Đắm mình vào Học Sâu

Release 0.17.5

Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola

Contents

Lời nói đầu **1**

Cài đặt **9**

1 Ký hiệu **13**
1.1 Các đối tượng số . . . 13
1.2 Lý thuyết đặt . . . 13
1.3 Chức năng và toán tử . . . 14
1.4 Calculus (Giải tích) . . . 14
1.5 Lý thuyết xác suất và thông tin . . . 14

2 Giới thiệu **17**
2.1 Một ví dụ động lực . . . 18
2.2 Các thành phần chính . . . 20
2.2.1 Dữ liệu . . . 20
2.2.2 Mô hình . . . 21
2.2.3 Hàm mục tiêu . . . 21
2.2.4 Thuật toán tối ưu hóa . . . 22
2.3 Các loại vấn đề về máy học . . . 22
2.3.1 Học được giám sát . . . 22
2.3.2 Học tập không được giám sát và tự giám sát . . . 30
2.3.3 Tương tác với môi trường . . . 31
2.3.4 Học tăng cường . . . 32
2.4 Rễ . . . 33
2.5 Con đường đến sâu học . . . 35
2.6 Câu chuyện thành công . . . 37
2.7 các điểm . . . 39
2.8 Tóm tắt . . . 40
2.9 Bài tập . . . 40

3 Sơ bộ **41**
3.1 Thao tác dữ liệu . . . 41
3.1.1 Bắt đầu . . . 42
3.1.2 Hoạt động . . . 44
3.1.3 Cơ chế phát sóng . . . 46
3.1.4 Lập chỉ mục và cắt . . . 46
3.1.5 Tiết kiệm bộ nhớ . . . 47
3.1.6 Chuyển đổi sang các đối tượng Python khác . . . 48
3.1.7 Tóm tắt . . . 49
3.1.8 Bài tập . . . 49
3.2 Xử lý sơ bộ dữ liệu . . . 49

3.2.1 Đọc tập dữ liệu . . . 49
3.2.2 Xử lý dữ liệu bị thiếu . . . 50
3.2.3 Chuyển đổi sang định dạng Tensor . . . 51
3.2.4 Tóm tắt . . . 51
3.2.5 Bài tập . . . 51
3.3 Đại số tuyến tính . . . 51
3.3.1 Vô hướng . . . 52
3.3.2 Vectơ . . . 52
3.3.3 Ma trận . . . 54
3.3.4 Tensors . . . 55
3.3.5 Tính chất cơ bản của Tensor Arithmetic . . . 56
3.3.6 Giảm . . . 57
3.3.7 Sản phẩm Dot . . . 59
3.3.8 Matrix-Vector Sản phẩm . . . 59
3.3.9 Phép nhân ma trận ma trận . . . 60
3.3.10 Định mức . . . 61
3.3.11 Tìm hiểu thêm về Linear Algebra . . . 63
3.3.12 Tóm tắt . . . 63
3.3.13 Bài tập . . . 63
3.4 Calculus (Giải tích) . . . 64
3.4.1 Các dẫn xuất và sự khác biệt . . . 65
3.4.2 Phái sinh một phần . . . 68
3.4.3 Độ dốc . . . 68
3.4.4 Quy tắc chuỗi . . . 69
3.4.5 Tóm tắt . . . 69
3.4.6 Bài tập . . . 69
3.5 Sự khác biệt tự động . . . 70
3.5.1 Một ví dụ đơn giản . . . 70
3.5.2 Lùi cho các biến không vô hướng . . . 71
3.5.3 Tách tính toán . . . 72
3.5.4 Tính toán Gradient của Python Control Flow . . . 72
3.5.5 Tóm tắt . . . 73
3.5.6 Bài tập . . . 73
3.6 Xác suất . . . 73
3.6.1 Lý thuyết xác suất cơ bản . . . 75
3.6.2 Xử lý nhiều biến ngẫu nhiên . . . 78
3.6.3 Kỳ vọng và phương sai . . . 81
3.6.4 Tóm tắt . . . 81
3.6.5 Bài tập . . . 81
3.7 Documentation . . . 82
3.7.1 Tìm tất cả các hàm và lớp học trong một mô-đun . . . 82
3.7.2 Tìm cách sử dụng các hàm và lớp cụ thể . . . 83
3.7.3 Tóm tắt . . . 84
3.7.4 Bài tập . . . 84

4 Mạng thần kinh tuyến tính **85**
4.1 Hồi quy tuyến tính . . . 85
4.1.1 Các yếu tố cơ bản của hồi quy tuyến tính . . . 85
4.1.2 Vectorization cho tốc độ . . . 89
4.1.3 Phân phối bình thường và tổn thất bình phương . . . 91
4.1.4 Từ hồi quy tuyến tính đến mạng sâu . . . 92

4.1.5 Tóm tắt . . . 94
4.1.6 Bài tập . . . 94
4.2 Thực hiện hồi quy tuyến tính từ đầu . . . 95
4.2.1 Tạo tập dữ liệu . . . 95
4.2.2 Đọc tập dữ liệu . . . 96
4.2.3 Khởi tạo các tham số mô hình . . . 98
4.2.4 Xác định mô hình . . . 98
4.2.5 Xác định chức năng mất . . . 98
4.2.6 Xác định thuật toán tối ưu hóa . . . 99
4.2.7 Đào tạo . . . 99
4.2.8 Tóm tắt . . . 100
4.2.9 Bài tập . . . 101
4.3 Thực hiện ngắn gọn của hồi quy tuyến tính . . . 101
4.3.1 Tạo tập dữ liệu . . . 101
4.3.2 Đọc tập dữ liệu . . . 102
4.3.3 Xác định mô hình . . . 103
4.3.4 Khởi tạo các tham số mô hình . . . 103
4.3.5 Xác định chức năng mất . . . 104
4.3.6 Xác định thuật toán tối ưu hóa . . . 104
4.3.7 Đào tạo . . . 104
4.3.8 Tóm tắt . . . 105
4.3.9 Bài tập . . . 105
4.4 Hồi quy Softmax . . . 105
4.4.1 Phân loại vấn đề . . . 106
4.4.2 Kiến trúc mạng . . . 106
4.4.3 Chi phí tham số của các lớp được kết nối hoàn toàn . . . 107
4.4.4 Softmax hoạt động . . . 107
4.4.5 Vectorization cho Minibatches . . . 108
4.4.6 Chức năng mất . . . 108
4.4.7 Thông tin cơ bản về lý thuyết . . . 110
4.4.8 Dự đoán và đánh giá mô hình . . . 111
4.4.9 Tóm tắt . . . 111
4.4.10 Bài tập . . . 111
4.5 Tập dữ liệu phân loại hình ảnh . . . 112
4.5.1 Đọc tập dữ liệu . . . 112
4.5.2 Đọc một Minibatch . . . 114
4.5.3 Đặt tất cả mọi thứ lại với nhau . . . 114
4.5.4 Tóm tắt . . . 115
4.5.5 Bài tập . . . 115
4.6 Thực hiện hồi quy Softmax từ đầu . . . 115
4.6.1 Khởi tạo các tham số mô hình . . . 116
4.6.2 Xác định hoạt động Softmax . . . 116
4.6.3 Xác định mô hình . . . 117
4.6.4 Xác định chức năng mất . . . 117
4.6.5 Phân loại chính xác . . . 118
4.6.6 Đào tạo . . . 119
4.6.7 Prediction . . . 122
4.6.8 Tóm tắt . . . 122
4.6.9 Bài tập . . . 122
4.7 Thực hiện ngắn gọn về hồi quy Softmax . . . 123
4.7.1 Khởi tạo các tham số mô hình . . . 123

4.7.2 Triển khai Softmax Revisited . . . 123
4.7.3 Thuật toán tối ưu hóa . . . 124
4.7.4 Đào tạo . . . 125
4.7.5 Tóm tắt . . . 125
4.7.6 Bài tập . . . 125

5 Multilayer Perceptrons **127**
5.1 Multilayer Perceptrons . . . 127
5.1.1 Các lớp ẩn . . . 127
5.1.2 Chức năng kích hoạt . . . 130
5.1.3 Tóm tắt . . . 135
5.1.4 Bài tập . . . 135
5.2 Thực hiện các Perceptrons đa lớp từ đầu . . . 136
5.2.1 Khởi tạo các tham số mô hình . . . 136
5.2.2 Chức năng kích hoạt . . . 136
5.2.3 Mô hình . . . 137
5.2.4 Chức năng mất . . . 137
5.2.5 Đào tạo . . . 137
5.2.6 Tóm tắt . . . 138
5.2.7 Bài tập . . . 138
5.3 Thực hiện ngắn gọn của Multilayer Perceptrons . . . 138
5.3.1 Mô hình . . . 139
5.3.2 Tóm tắt . . . 139
5.3.3 Bài tập . . . 140
5.4 Lựa chọn mô hình, Underfitting, và Overfitting . . . 140
5.4.1 Lỗi đào tạo và lỗi tổng quát . . . 141
5.4.2 Lựa chọn mô hình . . . 143
5.4.3 Underfitting hoặc Overfitting? . . . 144
5.4.4 Hồi quy đa thức . . . 145
5.4.5 Tóm tắt . . . 149
5.4.6 Bài tập . . . 149
5.5 Trọng lượng phân rã . . . 150
5.5.1 Định mức và phân rã trọng lượng . . . 150
5.5.2 High-Dimensional Linear Regression . . . 151
5.5.3 Thực hiện từ đầu . . . 152
5.5.4 Thiết tập . . . 154
5.5.5 Tóm tắt . . . 156
5.5.6 Bài tập . . . 156
5.6 Bỏ học . . . 156
5.6.1 Overfitting Revisited . . . 157
5.6.2 Mạnh mẽ thông qua nhiễu loạn . . . 157
5.6.3 Bỏ học trong thực hành . . . 158
5.6.4 Thực hiện từ đầu . . . 159
5.6.5 Thiết tập . . . 161
5.6.6 Tóm tắt . . . 162
5.6.7 Bài tập . . . 162
5.7 Chuyển tiếp tuyên truyền, tuyên truyền ngược, và đồ thị tính toán . . . 163
5.7.1 Chuyển tiếp tuyên truyền . . . 163
5.7.2 Đồ thị tính toán của chuyển tiếp tuyên truyền . . . 164
5.7.3 Backpropagation . . . 164
5.7.4 Đào tạo mạng nơ-ron . . . 165

5.7.5 Tóm tắt . . . 166
5.7.6 Bài tập . . . 166
5.8 Tính ổn định số và khởi tạo . . . 166
5.8.1 Biến mất và bùng nổ Gradient . . . 167
5.8.2 Khởi tạo tham số . . . 169
5.8.3 Tóm tắt . . . 171
5.8.4 Bài tập . . . 171
5.9 Môi trường và phân phối Shift . . . 171
5.9.1 Các loại dịch chuyển phân phối . . . 172
5.9.2 Ví dụ về Distribution Shift . . . 174
5.9.3 Sửa đổi sự thay đổi phân phối . . . 176
5.9.4 Một phân loại của các vấn đề học tập . . . 179
5.9.5 Công bằng, trách nhiệm giải trình và minh bạch trong học máy . . . 181
5.9.6 Tóm tắt . . . 181
5.9.7 Bài tập . . . 182
5.10 Dự đoán giá nhà trên Kaggle . . . 182
5.10.1 Tải xuống và bộ dữ liệu bộ nhớ đệm . . . 182
5.10.2 Kaggle . . . 184
5.10.3 Truy cập và đọc tập dữ liệu . . . 185
5.10.4 Xử lý sơ bộ dữ liệu . . . 186
5.10.5 Đào tạo . . . 187
5.10.6 K-Xác định chéo Fold . . . 188
5.10.7 Model Selection . . . 189
5.10.8 Nộp dự đoán trên Kaggle . . . 190
5.10.9 Tóm tắt . . . 192
5.10.10 Bài tập . . . 192

6 Tính toán học sâu **193**
6.1 Lớp và khối . . . 193
6.1.1 Một khối tùy chỉnh . . . 195
6.1.2 The Sequential Block . . . 196
6.1.3 Thihành mã trong chức năng tuyên truyền chuyển tiếp . . . 197
6.1.4 Hiệu quả . . . 199
6.1.5 Tóm tắt . . . 199
6.1.6 Bài tập . . . 199
6.2 Quản lý tham số . . . 200
6.2.1 Truy cập tham số . . . 200
6.2.2 Khởi tạo tham số . . . 203
6.2.3 Tham số Tied . . . 205
6.2.4 Tóm tắt . . . 206
6.2.5 Bài tập . . . 206
6.3 Layers tùy chỉnh . . . 206
6.3.1 Các lớp không có tham số . . . 206
6.3.2 Các lớp có tham số . . . 207
6.3.3 Tóm tắt . . . 208
6.3.4 Bài tập . . . 208
6.4 Tệp I/O . . . 208
6.4.1 Tải và tiết kiệm Tensors . . . 209
6.4.2 Tải và lưu thông số mô hình . . . 209
6.4.3 Tóm tắt . . . 210
6.4.4 Bài tập . . . 211

6.5 GPU . . . 211
6.5.1 Thiết bị vi tính . . . 213
6.5.2 Tensors và GPU . . . 214
6.5.3 Mạng nơ-ron và GPU . . . 216
6.5.4 Tóm tắt . . . 216
6.5.5 Bài tập . . . 217

7 Mạng thần kinh phức tạp 219
7.1 Từ các lớp được kết nối hoàn toàn đến sự phức tạp . . . 220
7.1.1 Bất biến . . . 220
7.1.2 Hạn chế MLP . . . 221
7.1.3 Sự phức tạp . . . 223
7.1.4 “Where’s Waldo” Revisited . . . 223
7.1.5 Tóm tắt . . . 224
7.1.6 Bài tập . . . 225
7.2 Sự phức tạp cho hình ảnh . . . 225
7.2.1 Hoạt động tương quan chéo . . . 225
7.2.2 Layers phức tạp . . . 227
7.2.3 Phát hiện cạnh đối tượng trong hình ảnh . . . 227
7.2.4 Học một hạt nhân . . . 228
7.2.5 Tương quan chéo và phức tạp . . . 229
7.2.6 Bản đồ tính năng và trường tiếp nhận . . . 230
7.2.7 Tóm tắt . . . 230
7.2.8 Bài tập . . . 230
7.3 Đệm và sải chân . . . 231
7.3.1 Đệm . . . 231
7.3.2 Sải chân . . . 233
7.3.3 Tóm tắt . . . 234
7.3.4 Bài tập . . . 234
7.4 Nhiều kênh đầu vào và nhiều đầu ra . . . 234
7.4.1 Nhiều kênh đầu vào . . . 235
7.4.2 Nhiều kênh đầu ra . . . 236
7.4.3 1×1 Lớp phức tạp . . . 237
7.4.4 Tóm tắt . . . 238
7.4.5 Bài tập . . . 238
7.5 Pooling . . . 239
7.5.1 Pooling tối đa và Pooling trung bình . . . 239
7.5.2 Padding và Stride . . . 241
7.5.3 Nhiều kênh . . . 242
7.5.4 Tóm tắt . . . 242
7.5.5 Bài tập . . . 243
7.6 Mạng thần kinh phức tạp (LeNet) . . . 243
7.6.1 LeNet . . . 244
7.6.2 Đào tạo . . . 246
7.6.3 Tóm tắt . . . 248
7.6.4 Bài tập . . . 248

8 Mạng thần kinh phức tạp hiện đại 249
8.1 Mạng thần kinh phức tạp sâu (AlexNet) . . . 249
8.1.1 Đại diện học tập . . . 250
8.1.2 AlexNet . . . 253

8.1.3 Đọc tập dữ liệu . . . 256
8.1.4 Đào tạo . . . 256
8.1.5 Tóm tắt . . . 256
8.1.6 Bài tập . . . 257
8.2 Mạng sử dụng khối (VGG) . . . 257
8.2.1 VGG Blocks . . . 258
8.2.2 VGG mạng . . . 258
8.2.3 Đào tạo . . . 260
8.2.4 Tóm tắt . . . 261
8.2.5 Bài tập . . . 261
8.3 Mạng trong mạng (Nin) . . . 262
8.3.1 NiN Blocks . . . 262
8.3.2 Mô hình NiN . . . 264
8.3.3 Đào tạo . . . 265
8.3.4 Tóm tắt . . . 265
8.3.5 Bài tập . . . 265
8.4 Mạng có kết nối song song (GoogLeNet) . . . 266
8.4.1 Chống** Bắt nối . . . 266
8.4.2 Mô hình GoogLeNet . . . 267
8.4.3 Đào tạo . . . 270
8.4.4 Tóm tắt . . . 270
8.4.5 Bài tập . . . 270
8.5 Chuẩn hóa hàng loạt . . . 271
8.5.1 Đào tạo mạng sâu . . . 271
8.5.2 Các lớp chuẩn hóa hàng loạt . . . 273
8.5.3 Implementation from Scratch . . . 274
8.5.4 Áp dụng Bình thường hóa hàng loạt trong LeNet . . . 275
8.5.5 Thiết tập . . . 276
8.5.6 Tranh cãi . . . 277
8.5.7 Tóm tắt . . . 278
8.5.8 Bài tập . . . 278
8.6 Mạng dư (ResNet) . . . 279
8.6.1 Các lớp hàm . . . 279
8.6.2 Khối** Residual . . . 280
8.6.3 Mô hình ResNet . . . 282
8.6.4 Đào tạo . . . 285
8.6.5 Tóm tắt . . . 285
8.6.6 Bài tập . . . 286
8.7 Mạng kết nối mật độ (DenseNet) . . . 286
8.7.1 Từ ResNet đến DenseNet . . . 286
8.7.2 Khối dày dàng . . . 287
8.7.3 Layers Transition . . . 288
8.7.4 Mô hình DenseNet . . . 289
8.7.5 Đào tạo . . . 289
8.7.6 Tóm tắt . . . 290
8.7.7 Bài tập . . . 290

9 Mạng nơ-ron tái phát **291**
9.1 Mô hình trình tự . . . 291
9.1.1 Công cụ thống kê . . . 293
9.1.2 Đào tạo . . . 295

9.1.3 Prediction . . . 297
9.1.4 Tóm tắt . . . 299
9.1.5 Bài tập . . . 300
9.2 Xử lý sơ bộ văn bản . . . 300
9.2.1 Đọc tập dữ liệu . . . 301
9.2.2 Mã hóa . . . 301
9.2.3 Từ vựng . . . 302
9.2.4 Đặt tất cả mọi thứ lại với nhau . . . 304
9.2.5 Tóm tắt . . . 304
9.2.6 Bài tập . . . 304
9.3 Mô hình ngôn ngữ và bộ dữ liệu . . . 305
9.3.1 Học một mô hình ngôn ngữ . . . 305
9.3.2 Mô hình Markov và n-gram . . . 306
9.3.3 Natural Language Thống kê . . . 307
9.3.4 Đọc dữ liệu trình tự dài . . . 310
9.3.5 Tóm tắt . . . 313
9.3.6 Bài tập . . . 313
9.4 Mạng nơ-ron tái phát . . . 314
9.4.1 Mạng thần kinh không có trạng thái ẩn . . . 314
9.4.2 Mạng thần kinh định kỳ với các trạng thái ẩn . . . 315
9.4.3 Mô hình ngôn ngữ cấp ký tự dựa trên RNN . . . 317
9.4.4 Bối rối . . . 317
9.4.5 Tóm tắt . . . 319
9.4.6 Bài tập . . . 319
9.5 Thực hiện các mạng nơ-ron tái phát từ đầu . . . 319
9.5.1 Mã hóa Một-Hot . . . 320
9.5.2 Khởi tạo các tham số mô hình . . . 320
9.5.3 Mô hình RNN . . . 321
9.5.4 Prediction . . . 322
9.5.5 Clipping Gradient . . . 323
9.5.6 Đào tạo . . . 324
9.5.7 Tóm tắt . . . 327
9.5.8 Bài tập . . . 327
9.6 Thực hiện ngắn gọn các mạng nơ-ron tái phát . . . 328
9.6.1 Định nghĩa mẫu . . . 328
9.6.2 Đào tạo và dự đoán . . . 329
9.6.3 Tóm tắt . . . 330
9.6.4 Bài tập . . . 330
9.7 Backpropagation qua thời gian . . . 331
9.7.1 Phân tích Gradient trong RNNs . . . 331
9.7.2 Backpropagation qua thời gian chi tiết . . . 334
9.7.3 Tóm tắt . . . 336
9.7.4 Bài tập . . . 336

10 Mạng thần kinh tái phát hiện đại **337**
10.1 Các đơn vị định kỳ cổng (GRU) . . . 337
10.1.1 Nhà nước ẩn Gated . . . 338
10.1.2 Thực hiện từ đầu . . . 341
10.1.3 Thiết tập . . . 343
10.1.4 Tóm tắt . . . 343
10.1.5 Bài tập . . . 344

10.2 Bộ nhớ ngắn hạn dài (LSTM) . . . 344
10.2.1 Gated Memory Cell . . . 344
10.2.2 Thực hiện từ đầu . . . 347
10.2.3 Thiết tập . . . 349
10.2.4 Tóm tắt . . . 350
10.2.5 Bài tập . . . 350
10.3 Mạng thần kinh tái phát sâu . . . 351
10.3.1 Phụ thuộc chức năng . . . 352
10.3.2 Thực hiện ngắn gọn . . . 352
10.3.3 Đào tạo và Dự đoán . . . 353
10.3.4 Tóm tắt . . . 353
10.3.5 Bài tập . . . 353
10.4 Mạng nơ-ron định kỳ hai chiều . . . 354
10.4.1 Lập trình động trong các mô hình Markov ẩn . . . 354
10.4.2 Mô hình hai chiều . . . 356
10.4.3 Đào tạo một RNN hai chiều cho một ứng dụng sai . . . 358
10.4.4 Tóm tắt . . . 359
10.4.5 Bài tập . . . 359
10.5 Dịch máy và bộ dữ liệu . . . 360
10.5.1 Tải xuống và xử lý sơ bộ dữ liệu . . . 360
10.5.2 Tokenization . . . 361
10.5.3 Từ vừ . . . 363
10.5.4 Đọc tập dữ liệu . . . 363
10.5.5 Putting All Things Together . . . 364
10.5.6 Tóm tắt . . . 365
10.5.7 Bài tập . . . 365
10.6 Kiến trúc mã hóa-Decoder . . . 365
10.6.1 Bộ mã hóa . . . 366
10.6.2 Decoder . . . 366
10.6.3 Đút bộ mã hóa và bộ giải mã cùng nhau . . . 367
10.6.4 Tóm tắt . . . 367
10.6.5 Bài tập . . . 367
10.7 Trình tự để học trình tự . . . 368
10.7.1 Bộ mã hóa . . . 369
10.7.2 Decoder . . . 370
10.7.3 Chức năng mất . . . 372
10.7.4 Đào tạo . . . 373
10.7.5 Prediction . . . 375
10.7.6 Đánh giá các chuỗi dự đoán . . . 376
10.7.7 Tóm tắt . . . 377
10.7.8 Bài tập . . . 377
10.8 Tìm kiếm chùm . . . 378
10.8.1 Tìm kiếm tham lam . . . 378
10.8.2 Tìm kiếm đầy đủ . . . 379
10.8.3 Tìm kiếm chùm . . . 379
10.8.4 Tóm tắt . . . 381
10.8.5 Bài tập . . . 381

11 Cơ chế chú ý **383**
11.1 Chú ý tín hiệu . . . 383
11.1.1 Các tín hiệu chú ý trong sinh học . . . 384

11.1.2 Truy vấn, Khóa và Giá trị . . . 385
11.1.3 Hình dung của sự chú ý . . . 386
11.1.4 Tóm tắt . . . 387
11.1.5 Bài tập . . . 387
11.2 Chú ý Pooling: Hồi quy hạt nhân Nadaraya-Watson . . . 388
11.2.1 Tạo ra bộ dữ liệu . . . 388
11.2.2 Pooling trung bình . . . 389
11.2.3 Không tham số Chú ý Pooling . . . 389
11.2.4 ** Tham số chú ý Pooling** . . . 391
11.2.5 Tóm tắt . . . 394
11.2.6 Bài tập . . . 395
11.3 Chức năng chấm điểm chú ý . . . 395
11.3.1 Masked Softmax Operation . . . 396
11.3.2 Phụ gia chú ý . . . 397
11.3.3 Quy mô Dot-Product Attention . . . 399
11.3.4 Tóm tắt . . . 400
11.3.5 Bài tập . . . 400
11.4 Bahdanau Chú ý . . . 400
11.4.1 Mô hình . . . 401
11.4.2 Xác định bộ giải mã với sự chú ý . . . 401
11.4.3 Đào tạo . . . 403
11.4.4 Tóm tắt . . . 405
11.4.5 Bài tập . . . 405
11.5 Chú ý nhiều đầu . . . 405
11.5.1 Mô hình . . . 406
11.5.2 Thực hiện . . . 406
11.5.3 Tóm tắt . . . 409
11.5.4 Bài tập . . . 409
11.6 Tự ghi nhận và mã hóa vị trí . . . 409
11.6.1 Self-Attention . . . 409
11.6.2 So sánh CNN s, RNNs, và Self-Attention . . . 410
11.6.3 Mã hóa vị trí . . . 411
11.6.4 Tóm tắt . . . 414
11.6.5 Bài tập . . . 414
11.7 Máy biến áp . . . 414
11.7.1 Mô hình . . . 414
11.7.2 Các mạng thức ăn chuyển tiếp vị trí . . . 416
11.7.3 Kết nối còn lại và chuẩn hóa lớp . . . 417
11.7.4 Bộ mã hóa . . . 418
11.7.5 Bộ giải mã . . . 419
11.7.6 Đào tạo . . . 421
11.7.7 Tóm tắt . . . 425
11.7.8 Bài tập . . . 425

12 Thuật toán tối ưu hóa **427**
12.1 Tối ưu hóa và học sâu . . . 427
12.1.1 Mục tiêu tối ưu hóa . . . 427
12.1.2 Tối ưu hóa thách thức trong Deep Learning . . . 429
12.1.3 Tóm tắt . . . 432
12.1.4 Bài tập . . . 432
12.2 Độ lồi . . . 433

12.2.1 Định nghĩa . . . 433
12.2.2 Thuộc tính . . . 436
12.2.3 Ràng buộc . . . 438
12.2.4 Tóm tắt . . . 440
12.2.5 Bài tập . . . 440
12.3 Gradient Descent . . . 441
12.3.1 Một chiều Gradient Descent . . . 441
12.3.2 Đa biến Gradient Descent . . . 445
12.3.3 Phương pháp thích ứng . . . 446
12.3.4 Tóm tắt . . . 450
12.3.5 Bài tập . . . 451
12.4 Stochastic Gradient Descent . . . 451
12.4.1 Stochastic Gradient Những Thông Tin Cập Nhập . . . 451
12.4.2 Tốc độ học động . . . 453
12.4.3 Phân tích hội tụ cho mục tiêu lồi . . . 455
12.4.4 Gradient Stochastic và mẫu hữu hạn . . . 457
12.4.5 Tóm tắt . . . 457
12.4.6 Bài tập . . . 458
12.5 Minibatch Stochastic Gradient Descent . . . 458
12.5.1 Vector hóa và bộ nhớ cache . . . 458
12.5.2 Minibatches . . . 460
12.5.3 Đọc tập dữ liệu . . . 461
12.5.4 Thực hiện từ đầu . . . 462
12.5.5 Thực hiện ngắn gọn . . . 465
12.5.6 Tóm tắt . . . 466
12.5.7 Bài tập . . . 467
12.6 Đà . . . 467
12.6.1 Khái niệm cơ bản . . . 467
12.6.2 Các thí nghiệm thực tế . . . 472
12.6.3 Phân tích lý thuyết . . . 474
12.6.4 Tóm tắt . . . 476
12.6.5 Bài tập . . . 477
12.7 Adagrad . . . 477
12.7.1 Tính năng thưa thớt và tỷ lệ học tập . . . 477
12.7.2 Điều hòa trước . . . 478
12.7.3 Các thuật toán . . . 479
12.7.4 Thực hiện từ đầu . . . 481
12.7.5 Thực hiện ngắn gọn . . . 482
12.7.6 Tóm tắt . . . 482
12.7.7 Bài tập . . . 483
12.8 RMSProp . . . 483
12.8.1 Các thuật toán . . . 484
12.8.2 Thực hiện từ đầu . . . 485
12.8.3 Thực hiện ngắn gọn . . . 486
12.8.4 Tóm tắt . . . 487
12.8.5 Bài tập . . . 487
12.9 Adadelta . . . 488
12.9.1 Các thuật toán . . . 488
12.9.2 Thực hiện . . . 488
12.9.3 Tóm tắt . . . 490
12.9.4 Bài tập . . . 490

12.10 Adam . . . 490
12.10.1 Các thuật toán . . . 491
12.10.2 Thực hiện . . . 492
12.10.3 Yogi . . . 493
12.10.4 Tóm tắt . . . 494
12.10.5 Bài tập . . . 494
12.11 Lập kế hoạch tỷ lệ học tập . . . 495
12.11.1 Vấn đề Toy . . . 495
12.11.2 Người lập lịch . . . 497
12.11.3 Chính sách . . . 499
12.11.4 Tóm tắt . . . 504
12.11.5 Bài tập . . . 504

13 Hiệu suất tính toán **505**
13.1 Trình biên dịch và phiên dịch . . . 505
13.1.1 Lập trình tượng trưng . . . 506
13.1.2 Lập trình lai . . . 507
13.1.3 Lai tạo lớp `Sequential` . . . 508
13.1.4 Tóm tắt . . . 510
13.1.5 Bài tập . . . 510
13.2 Tính toán không đồng bộ . . . 510
13.2.1 Asynchrony qua Backend . . . 510
13.2.2 Rào cản và chặn . . . 513
13.2.3 Cải thiện tính toán . . . 513
13.2.4 Tóm tắt . . . 513
13.2.5 Bài tập . . . 513
13.3 Song song tự động . . . 513
13.3.1 Tính toán song song trên GPU . . . 514
13.3.2 Tính toán song song và truyền thông . . . 515
13.3.3 Tóm tắt . . . 516
13.3.4 Bài tập . . . 516
13.4 Phần cứng . . . 517
13.4.1 Máy vi tính . . . 518
13.4.2 Bộ nhớ . . . 519
13.4.3 Lưu trữ . . . 520
13.4.4 CPU . . . 521
13.4.5 GPU và các bộ tăng tốc khác . . . 524
13.4.6 Mạng và xe buýt . . . 526
13.4.7 Nhiều số độ trễ hơn . . . 527
13.4.8 Tóm tắt . . . 528
13.4.9 Bài tập . . . 529
13.5 Đào tạo về nhiều GPU . . . 529
13.5.1 Chia tách vấn đề . . . 530
13.5.2 Dữ liệu song song . . . 532
13.5.3 Một mạng đồ chơi . . . 533
13.5.4 Đồng bộ hóa dữ liệu . . . 533
13.5.5 Phân phối dữ liệu . . . 534
13.5.6 Đào tạo . . . 535
13.5.7 Tóm tắt . . . 537
13.5.8 Bài tập . . . 537
13.6 Triển khai ngắn gọn cho nhiều GPU . . . 538

13.6.1 Một mạng đồ chơi . . . 538
13.6.2 Khởi tạo mạng . . . 539
13.6.3 Đào tạo . . . 539
13.6.4 Tóm tắt . . . 541
13.6.5 Bài tập . . . 541
13.7 Máy chủ tham số . . . 541
13.7.1 Đào tạo dữ liệu song song . . . 541
13.7.2 Đồng bộ hóa vòng . . . 544
13.7.3 Đào tạo đa máy . . . 546
13.7.4 Cửa hàng Key — Value . . . 548
13.7.5 Tóm tắt . . . 549
13.7.6 Bài tập . . . 549

14 Tầm nhìn máy tính **551**
14.1 Hình ảnh Augmentation . . . 551
14.1.1 Phương pháp Augmentation hình ảnh phổ biến . . . 552
14.1.2 Đào tạo với hình ảnh Augmentation . . . 556
14.1.3 Tóm tắt . . . 559
14.1.4 Bài tập . . . 559
14.2 Tinh chỉnh . . . 559
14.2.1 Các bước . . . 560
14.2.2 Nhận dạng chó nóng . . . 561
14.2.3 Tóm tắt . . . 565
14.2.4 Bài tập . . . 565
14.3 Hộp phát hiện và giới hạn đối tượng . . . 566
14.3.1 Hộp giới hạn . . . 567
14.3.2 Tóm tắt . . . 568
14.3.3 Bài tập . . . 568
14.4 Hộp neo . . . 569
14.4.1 Tạo nhiều hộp neo . . . 569
14.4.2 Giao lộ qua Union (IoU) . . . 572
14.4.3 Dán nhãn hộp neo trong dữ liệu đào tạo . . . 573
14.4.4 Dự đoán các hộp giới hạn với ức chế không tối đa . . . 578
14.4.5 Tóm tắt . . . 581
14.4.6 Bài tập . . . 581
14.5 Phát hiện đối tượng đa quy mô . . . 582
14.5.1 Multiscale Neo Boxes . . . 582
14.5.2 Phát hiện đa quy mô . . . 584
14.5.3 Tóm tắt . . . 585
14.5.4 Bài tập . . . 585
14.6 Bộ dữ liệu phát hiện đối tượng . . . 586
14.6.1 Tải xuống dữ liệu . . . 586
14.6.2 Đọc tập dữ liệu . . . 586
14.6.3 Demonstration . . . 588
14.6.4 Tóm tắt . . . 588
14.6.5 Bài tập . . . 588
14.7 Phát hiện Multibox Shot đơn . . . 589
14.7.1 Mô hình . . . 589
14.7.2 Đào tạo . . . 595
14.7.3 Prediction . . . 597
14.7.4 Tóm tắt . . . 598

14.7.5 Bài tập . . . 598
14.8 CNN dựa trên khu vực (R-CNN) . . . 600
14.8.1 R-CNN . . . 600
14.8.2 R-CNN nhanh . . . 601
14.8.3 R-CNN nhanh hơn . . . 604
14.8.4 Mặt nạ R-CNN . . . 605
14.8.5 Tóm tắt . . . 605
14.8.6 Bài tập . . . 606
14.9 Phân đoạn ngữ nghĩa và tập dữ liệu . . . 606
14.9.1 Phân đoạn hình ảnh và Phân đoạn phiên bản . . . 606
14.9.2 Tập dữ liệu phân đoạn ngữ nghĩa Pascal VOC2012 . . . 607
14.9.3 Tóm tắt . . . 612
14.9.4 Bài tập . . . 612
14.10 Chuyển đổi Convolution . . . 612
14.10.1 Hoạt động cơ bản . . . 612
14.10.2 Đệm, sải bước và nhiều kênh . . . 614
14.10.3 Kết nối với Ma trận Transposition . . . 615
14.10.4 Tóm tắt . . . 616
14.10.5 Bài tập . . . 617
14.11 Mạng kết nối hoàn toàn . . . 617
14.11.1 Mô hình . . . 617
14.11.2 Initializing Transposed Convolutional Layers . . . 619
14.11.3 Đọc dữ liệu . . . 621
14.11.4 Đào tạo . . . 621
14.11.5 Prediction . . . 622
14.11.6 Tóm tắt . . . 623
14.11.7 Bài tập . . . 624
14.12 Chuyển kiểu thần kinh . . . 624
14.12.1 Phương pháp . . . 625
14.12.2 Đọc nội dung và phong cách hình ảnh . . . 626
14.12.3 Tiền xử lý và xử lý sau . . . 627
14.12.4 tính năng chiết xuất . . . 627
14.12.5 Defining the Loss Function . . . 628
14.12.6 Initializing the Synthesized Image . . . 630
14.12.7 Đào tạo . . . 631
14.12.8 Tóm tắt . . . 632
14.12.9 Bài tập . . . 632
14.13 Phân loại hình ảnh (CIFAR-10) trên Kaggle . . . 632
14.13.1 Lấy và tổ chức tập dữ liệu . . . 633
14.13.2 Image Augmentation . . . 636
14.13.3 Đọc tập dữ liệu . . . 636
14.13.4 Xác định Mẫu . . . 637
14.13.5 Xác định đào tạo chức năng . . . 637
14.13.6 Đào tạo và xác thực mô hình . . . 638
14.13.7 Phân loại bộ thử nghiệm và gửi kết quả trên Kaggle . . . 639
14.13.8 Tóm tắt . . . 639
14.13.9 Bài tập . . . 640
14.14 Nhận dạng giống chó (Chó ImageNet) trên Kaggle . . . 640
14.14.1 Lấy và tổ chức tập dữ liệu . . . 641
14.14.2 Image Augmentation . . . 642
14.14.3 Đọc dữ liệu . . . 643

14.14.4 Tinh chỉnh một mô hình Pretrained Model . . . 643
14.14.5 Xác định chức năng đào tạo . . . 644
14.14.6 Đào tạo và xác thực mô hình . . . 645
14.14.7 Phân loại bộ thử nghiệm và gửi kết quả trên Kaggle . . . 646
14.14.8 Tóm tắt . . . 647
14.14.9 Bài tập . . . 647

15 Chế biến ngôn ngữ tự nhiên: Pretraining 649
15.1 Từ nhúng (word2vec) . . . 650
15.1.1 Vectơ một nóng là một lựa chọn tồi . . . 650
15.1.2 Word2vec tự giám sát . . . 651
15.1.3 Mô hình Skip-Gram . . . 651
15.1.4 Mô hình túi từ liên tục (CBOW) . . . 652
15.1.5 Tóm tắt . . . 654
15.1.6 Bài tập . . . 654
15.2 Đào tạo gần đúng . . . 654
15.2.1 Lấy mẫu âm . . . 655
15.2.2 Phân cấp Softmax . . . 656
15.2.3 Tóm tắt . . . 657
15.2.4 Bài tập . . . 657
15.3 Các Dataset cho Pretraining Word Embeddings . . . 657
15.3.1 Đọc tập dữ liệu . . . 658
15.3.2 Lấy mẫu phụ . . . 658
15.3.3 Trích xuất từ trung tâm và từ ngữ cảnh . . . 660
15.3.4 Lấy mẫu âm . . . 661
15.3.5 Tải ví dụ đào tạo trong Minibatches . . . 662
15.3.6 Đặt tất cả mọi thứ lại với nhau . . . 663
15.3.7 Tóm tắt . . . 664
15.3.8 Bài tập . . . 664
15.4 Pretraining word2vec . . . 665
15.4.1 Mô hình Skip-Gram . . . 665
15.4.2 Đào tạo . . . 666
15.4.3 Áp dụng Word Embeddings . . . 668
15.4.4 Tóm tắt . . . 669
15.4.5 Bài tập . . . 669
15.5 Word Nhúng với Vectơ toàn cầu (Glove) . . . 669
15.5.1 Skip-Gram với Thống kê Corpus toàn cầu . . . 670
15.5.2 Mô hình Glove . . . 670
15.5.3 Giải thích găng tay từ Tỷ lệ xác suất đồng xuất hiện . . . 671
15.5.4 Tóm tắt . . . 672
15.5.5 Bài tập . . . 672
15.6 Từ tương tự và tương tự . . . 673
15.6.1 Đang tải Pretrained Word Vectơ . . . 673
15.6.2 Áp dụng vectơ Word Pretrained . . . 675
15.6.3 Tóm tắt . . . 677
15.6.4 Bài tập . . . 677
15.7 Đại diện bộ mã hóa hai chiều từ Transformers (BERT) . . . 677
15.7.1 Từ bối cảnh độc lập đến bối cảnh nhạy cảm . . . 677
15.7.2 Từ nhiệm vụ cụ thể đến nhiệm vụ Agnostic . . . 678
15.7.3 BERT: Kết hợp tốt nhất của cả hai thế giới . . . 678
15.7.4 Đại diện đầu vào . . . 679

15.7.5 Nhiệm vụ Pretraining . . . 681
15.7.6 Đặt tất cả mọi thứ lại với nhau . . . 684
15.7.7 Tóm tắt . . . 684
15.7.8 Bài tập . . . 685
15.8 Tập dữ liệu cho Pretraining BERT . . . 685
15.8.1 Xác định hàm trợ giúp cho các nhiệm vụ Pretraining . . . 686
15.8.2 Chuyển văn bản thành tập dữ liệu Pretraining . . . 688
15.8.3 Tóm tắt . . . 690
15.8.4 Bài tập . . . 690
15.9 Pretraining BERT . . . 691
15.9.1 Pretraining BERT . . . 691
15.9.2 Đại diện cho văn bản với BERT . . . 693
15.9.3 Tóm tắt . . . 694
15.9.4 Bài tập . . . 694

16 Xử lý ngôn ngữ tự nhiên: Ứng dụng 695
16.1 Phân tích tình cảm và tập dữ liệu . . . 696
16.1.1 Đọc tập dữ liệu . . . 696
16.1.2 Xử lý trước bộ dữ liệu . . . 697
16.1.3 Tạo bộ lặp dữ liệu . . . 698
16.1.4 Đặt tất cả mọi thứ lại với nhau . . . 698
16.1.5 Tóm tắt . . . 699
16.1.6 Bài tập . . . 699
16.2 Phân tích tình cảm: Sử dụng mạng nơ-ron tái phát . . . 699
16.2.1 Đại diện cho văn bản đơn với RNNs . . . 700
16.2.2 Đang tải Pretrained Word Vectơ . . . 701
16.2.3 Đào tạo và đánh giá mô hình . . . 701
16.2.4 Tóm tắt . . . 702
16.2.5 Bài tập . . . 703
16.3 Phân tích tình cảm: Sử dụng mạng thần kinh phức tạp . . . 703
16.3.1 Một chiều Convolutions . . . 704
16.3.2 Max-Over-Time Pooling . . . 706
16.3.3 Mô hình textCNN . . . 706
16.3.4 Tóm tắt . . . 709
16.3.5 Bài tập . . . 710
16.4 Suy luận ngôn ngữ tự nhiên và tập dữ liệu . . . 710
16.4.1 Suy luận ngôn ngữ tự nhiên . . . 710
16.4.2 Bộ dữ liệu Suy luận ngôn ngữ tự nhiên Stanford (SNLI) . . . 711
16.4.3 Tóm tắt . . . 714
16.4.4 Bài tập . . . 714
16.5 Suy luận ngôn ngữ tự nhiên: Sử dụng sự chú ý . . . 714
16.5.1 Mô hình . . . 715
16.5.2 Đào tạo và đánh giá mô hình . . . 719
16.5.3 Tóm tắt . . . 721
16.5.4 Bài tập . . . 721
16.6 Tinh chỉnh BERT cho các ứng dụng cấp Sequence-Level và Token-Level . . . 721
16.6.1 Phân loại văn bản đơn . . . 722
16.6.2 Phân loại cặp văn bản hoặc hồi quy . . . 722
16.6.3 Gắn thẻ văn bản . . . 723
16.6.4 Câu hỏi trả lời . . . 724
16.6.5 Tóm tắt . . . 725

16.6.6 Bài tập . . . 726
16.7 Suy luận ngôn ngữ tự nhiên: Tinh chỉnh BERT . . . 726
16.7.1 Đang tải BERT Pretrained . . . 727
16.7.2 Tập dữ liệu cho tinh chỉnh BERT . . . 728
16.7.3 Tinh chỉnh BERT . . . 729
16.7.4 Tóm tắt . . . 731
16.7.5 Bài tập . . . 731

17 Hệ thống Recommender **733**
17.1 Tổng quan về hệ thống Recommender . . . 733
17.1.1 Lọc hợp tác . . . 734
17.1.2 Phản hồi rõ ràng và phản hồi tiềm ẩn . . . 734
17.1.3 Tác vụ đề xuất . . . 735
17.1.4 Tóm tắt . . . 735
17.1.5 Bài tập . . . 735

18 Mạng đối thủ thế hệ **737**
18.1 Mạng đối thủ thế hệ . . . 737
18.1.1 Tạo ra một số dữ liệu "thực" . . . 739
18.1.2 Máy phát điện . . . 740
18.1.3 Phân biệt đối xử . . . 740
18.1.4 Đào tạo . . . 740
18.1.5 Tóm tắt . . . 742
18.1.6 Bài tập . . . 742
18.2 Mạng đối thủ tạo phức tạp sâu . . . 743
18.2.1 Các Pokemon Dataset . . . 743
18.2.2 Các máy phát điện . . . 744
18.2.3 Phân biệt đối xử . . . 746
18.2.4 Đào tạo . . . 747
18.2.5 Tóm tắt . . . 749
18.2.6 Bài tập . . . 749

19 Phụ lục: Toán học cho Deep Learning **751**
19.1 Hình học và các hoạt động đại số tuyến tính . . . 752
19.1.1 Hình học của Vectơ Trước tiên, chúng ta cần thảo luận về hai cách giải thích hình học phổ biến của vectơ, như một trong hai điểm hoặc hướng trong không gian. Về cơ bản, một vectơ là danh sách các số như danh sách Python bên dưới. . . . 752
19.1.2 Dot Sản phẩm và Angles Như chúng ta đã thấy trong Section 3.3, nếu chúng ta lấy hai vectơ cột **u** và **v**, chúng ta có thể tạo thành sản phẩm chấm của chúng bằng cách tính toán: . . . 754
19.1.3 Siêu máy bay . . . 756
19.1.4 Hình học của biến đổi tuyến tính . . . 759
19.1.5 Sự phụ thuộc tuyến tính . . . 761
19.1.6 thứ hạng . . . 762
19.1.7 Khả năng đảo ngược . . . 762
19.1.8 Xác định Quan điểm hình học của đại số tuyến tính đưa ra một cách trực quan để diễn giải một đại lượng cơ bản được gọi là *quyết định*. Hãy xem xét hình ảnh lưới từ trước, nhưng bây giờ với một khu vực được tô sáng (Fig. 19.1.9). . . . 764
19.1.9 Tensors và hoạt động đại số tuyến tính phổ biến . . . 765

19.1.10 Tóm tắt * Vectơ có thể được giải thích về mặt hình học như một trong hai điểm hoặc hướng trong không gian * Các sản phẩm chấm xác định khái niệm góc đến không gian chiều cao tùy ý. Chúng có thể được sử dụng để xác định các mặt phẳng quyết định thường được sử dụng như là bước cuối cùng trong một nhiệm vụ phân loại. * Phép nhân ma trận có thể được giải thích về mặt hình học là biến dạng thống nhất của tọa độ bên dưới. Chúng đại diện cho một cách rất hạn chế, nhưng về mặt toán học sạch, để biến đổi vectơ. * Phụ thuộc tuyến tính là một cách để biết khi một tập hợp các vectơ nằm trong một không gian chiều thấp hơn chúng ta mong đợi (giả sử bạn có 3 vectơ sống trong một không gian 2 chiều). Thứ hạng của ma trận là kích thước của tập hợp con lớn nhất của các cột của nó độc lập tuyến tính * Khi nghịch đảo của ma trận được xác định, đảo ngược ma trận cho phép chúng ta tìm một ma trận khác hoàn tác hành động của đầu tiên. Ma trận đảo ngược là hữu ích trong lý thuyết, nhưng đòi hỏi sự chăm sóc trong thực tế do sự bất ổn số.* yếu tố quyết định cho phép chúng ta đo lường bao nhiêu ma trận mở rộng hoặc ký hợp đồng một không gian. Một yếu tố quyết định nonzero ngụ ý một ma trận đảo ngược (không số ít) và một yếu tố xác định có giá trị không có nghĩa là ma trận không thể đảo ngược (số ít). * co thắt tensor và tổng kết Einstein cung cấp cho một ký hiệu gọn gàng và sạch sẽ để thể hiện nhiều tính toán được nhìn thấy trong học máy. 767
19.2 Eigendecompositions 768
19.2.1 Tìm kiếm Eigenvalues Hãy để chúng tôi tìm ra cách tìm chúng. Bằng cách trừ $\lambda \mathbf{v}$ từ cả hai phía, và sau đó bao thanh toán ra vectơ, ta thấy ở trên tương đương với: . . . 769
19.2.2 Phân hủy ma trận Hãy để chúng tôi tiếp tục ví dụ trước một bước nữa. Hãy để . . . 770
19.2.3 Hoạt động trên Eigendecomposition Một điều hay về eigendecomposition (19.2.9) là chúng ta có thể viết nhiều thao tác mà chúng ta thường gặp phải một cách sạch sẽ về sự phân hủy. Như một ví dụ đầu tiên, hãy xem xét: 770
19.2.4 Eigendecompositions of Symmetric Matrận Không phải lúc nào cũng có thể tìm thấy đủ các eigenvectơ độc lập tuyến tính cho quá trình trên hoạt động. For instance ví dụ the matrix ma trận 771
19.2.5 Định lý vòng tròn Gershgorin Eigenvalues thường khó lý luận với trực giác. Nếu trình bày một ma trận tùy ý, có rất ít điều có thể nói về những gì eigenvalues là mà không tính toán chúng. Tuy nhiên, có một định lý có thể làm cho nó dễ dàng gần đúng nếu các giá trị lớn nhất nằm trên đường chéo. 771
19.2.6 Một ứng dụng hữu ích: Sự phát triển của bản đồ lặp 772
19.2.7 Kết luận 777
19.2.8 Tóm tắt * Eigenvectors là các vectơ được kéo dài bởi một ma trận mà không thay đổi hướng. * Eigenvalues là số tiền mà các eigenvectors được kéo dài bởi ứng dụng của ma trận. * Sự phân hủy của ma trận có thể cho phép nhiều phép toán được giảm xuống các hoạt động trên eigenvalues. Định lý vòng tròn Gershgorin có thể cung cấp các giá trị gần đúng cho eigenvalues của ma trận * hành vi của các công suất ma trận lặp lại phụ thuộc chủ yếu vào kích thước của eigenvalue lớn nhất. Sự hiểu biết này có nhiều ứng dụng trong lý thuyết khởi tạo mạng thần kinh. 777
19.3 Tính toán biến đơn 778
19.3.1 Vi phân Giải tích vi phân về cơ bản là nghiên cứu về cách các chức năng hoạt động dưới những thay đổi nhỏ. Để xem lý do tại sao điều này là cốt lõi để học sâu, chúng ta hãy xem xét một ví dụ. 778
19.3.2 Quy tắc của Calculus 781
19.3.3 Tóm tắt 788
19.3.4 Bài tập 788
19.4 Tính toán đa năng 788

19.4.1 Sự khác biệt chiều cao hơn Điều Section 19.3 nói với chúng ta là nếu chúng ta thay đổi một trong số hàng tỷ trọng này để lại mọi trọng lượng khác cố định, chúng ta biết điều gì sẽ xảy ra! Điều này không có gì khác hơn là một hàm của một biến duy nhất, vì vậy chúng ta có thể viết . . . 788
19.4.2 Hình học của Gradient và Gradient Descent Xem xét biểu thức từ (19.4.5) một lần nữa: . . . 790
19.4.3 Một lưu ý về tối ưu hóa toán học Trong suốt cuốn sách này, chúng tôi tập trung thẳng vào các kỹ thuật tối ưu hóa số vì lý do thực tế mà tất cả các hàm chúng ta gặp phải trong cài đặt deep learning là quá phức tạp để giảm thiểu rõ ràng. . . . 791
19.4.4 Multivariate Chain Rule . . . 792
19.4.5 Thuật toán Backpropagation . . . 794
19.4.6 Hessia Như với phép tính biến đơn lẻ, nó rất hữu ích để xem xét các dẫn xuất bậc cao hơn để có được một xử lý về cách chúng ta có thể có được một xấp xỉ tốt hơn với một hàm hơn là sử dụng gradient một mình. . . . 797
19.4.7 A Little Matrix Calculus Dẫn xuất của các hàm liên quan đến ma trận hóa ra là đặc biệt tốt đẹp. Phần này có thể trở nên nặng nề về mặt công thức, vì vậy có thể bị bỏ qua trong lần đọc đầu tiên, nhưng rất hữu ích khi biết các dẫn xuất của các hàm liên quan đến hoạt động ma trận phổ biến thường sạch hơn nhiều so với người ta có thể dự đoán ban đầu, đặc biệt là cho các hoạt động ma trận trung tâm như thế nào đối với các ứng dụng học sâu. . . . 799
19.4.8 Tóm tắt . . . 803
19.4.9 Bài tập 1. Cho một vector cột $\boldsymbol{\beta}$, tính toán các dẫn xuất của cả $f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$ và $g(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\beta}$. Tại sao bạn nhận được câu trả lời tương tự? 2. Hãy để $\mathbf{v}$ là một vector kích thước n. $\frac{\partial}{\partial \mathbf{v}} \|\mathbf{v}\|_2$ là cái gì? 3. Hãy để $L(x, y) = \log(e^x + e^y)$. Tính toán gradient. Tổng của các thành phần của gradient là gì? 4. Hãy để $f(x, y) = x^2 y + x y^2$. Cho thấy điểm quan trọng duy nhất là $(0, 0)$. Bằng cách xem xét $f(x, x)$, xác định xem $(0, 0)$ có phải là tối đa, tối thiểu hay không. Giả sử rằng chúng ta đang giảm thiểu một hàm $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$. Làm thế nào chúng ta có thể giải thích hình học tình trạng của $\nabla f = 0$ về g và h? . . . 803
19.5 Integral Calculus . . . 803
19.5.1 Giải thích hình học Giả sử chúng ta có hàm $f(x)$. Để đơn giản, chúng ta hãy giả định rằng $f(x)$ là không âm (không bao giờ có giá trị nhỏ hơn 0). Điều chúng ta muốn thử và hiểu là: khu vực chứa giữa $f(x)$ và trục x-trục là gì? . . . 804
19.5.2 Định lý cơ bản của giải tích . . . 806
19.5.3 Thay đổi các biến . . . 808
19.5.4 Một bình luận về Sign Conventions . . . 809
19.5.5 Nhiều tích hợp Trong một số trường hợp, chúng ta sẽ cần phải làm việc ở các kích thước cao hơn. Ví dụ, giả sử rằng chúng ta có một hàm của hai biến, như $f(x, y)$ và chúng ta muốn biết khối lượng dưới f khi x phạm vi trên $[a, b]$ và y phạm vi trên $[c, d]$. . . . 810
19.5.6 Thay đổi biến trong nhiều tích phân Như với các biến đơn trong (19.5.18), khả năng thay đổi các biến bên trong tích phân chiều cao hơn là một công cụ quan trọng. Hãy để chúng tôi tóm tắt kết quả mà không có nguồn gốc. . . . 812
19.5.7 Tóm tắt . . . 813
19.6 Biến ngẫu nhiên . . . 813
19.6.1 Biến ngẫu nhiên liên tục . . . 814

19.6.2 Tóm lượng* Biến ngẫu nhiên liên tục là các biến ngẫu nhiên có thể thực hiện một liên tục của các giá trị. Chúng có một số khó khăn kỹ thuật khiến chúng trở nên khó khăn hơn khi làm việc so với các biến ngẫu nhiên rời rạc * Hàm mật độ xác suất cho phép chúng ta làm việc với các biến ngẫu nhiên liên tục bằng cách đưa ra một hàm trong đó khu vực dưới đường cong trên một khoảng thời gian cho xác suất tìm thấy một điểm mẫu trong khoảng thời gian đó* Hàm phân phối tích lũy là xác suất quan sát biến ngẫu nhiên nhỏ hơn một ngưỡng nhất định. Nó có thể cung cấp một quan điểm thay thế hữu ích thống nhất các biến rời rạc và liên tục.* trung bình là giá trị trung bình của một biến ngẫu nhiên. * phương sai là bình phương dự kiến của sự khác biệt giữa biến ngẫu nhiên và trung bình của nó.* Độ lệch chuẩn là căn bậc hai của phương sai. Nó có thể được coi là đo phạm vi các giá trị mà biến ngẫu nhiên có thể mất. * Bất đẳng thức của Chebyshev cho phép chúng ta làm cho trực giác này nghiêm ngặt bằng cách đưa ra một khoảng rõ ràng chứa biến ngẫu nhiên hầu hết thời gian* Mật độ chung cho phép chúng ta làm việc với các biến ngẫu nhiên tương quan. Chúng ta có thể lề mật độ chung bằng cách tích hợp các biến ngẫu nhiên không mong muốn để có được sự phân bố của biến ngẫu nhiên mong muốn* hệ số đồng phương sai và hệ số tương quan cung cấp một cách để đo bất kỳ mối quan hệ tuyến tính nào giữa hai biến ngẫu nhiên tương quan. 830
19.7 Khả năng tối đa 830
19.7.1 Nguyên tắc khả năng tối đa 831
19.7.2 Tối ưu hóa số và tiêu cực Log-Likelihood 832
19.7.3 Khả năng tối đa cho các biến liên tục 834
19.7.4 Tóm tắt * Nguyên tắc khả năng tối đa cho chúng ta biết rằng mô hình phù hợp nhất cho một tập dữ liệu nhất định là mô hình tạo ra dữ liệu có xác suất cao nhất.* Thường mọi người làm việc với khả năng ghi âm thay vì nhiều lý do: ổn định số, chuyển đổi sản phẩm thành tổng (và kết quả đơn giản hóa tính toán gradient), và mối quan hệ lý thuyết với lý thuyết thông tin * Mặc dù đơn giản nhất để thúc đẩy trong cài đặt rời rạc, nó có thể được tự do khái quát hóa để cài đặt liên tục cũng như bằng cách tối đa hóa mật độ xác suất được gán cho các datapoints. 835
19.7.5 Bài tập 1. Giả sử rằng bạn biết rằng một biến ngẫu nhiên có mật độ $\frac{1}{\alpha}e^{-\alpha x}$ cho một số giá trị α. Bạn có được một quan sát duy nhất từ biến ngẫu nhiên đó là số 3. Ước tính khả năng tối đa cho α là gì? 2. Giả sử rằng bạn có một tập dữ liệu của các mẫu $\{x_i\}_{i=1}^N$ rút ra từ một Gaussian với trung bình không xác định, nhưng phương sai 1. Ước tính khả năng tối đa cho nghĩa là gì? 835
19.8 Bayes ngây thơ 836
19.8.1 Nhận dạng ký tự quang 836
19.8.2 Mô hình xác suất để phân loại 838
19.8.3 Phân loại Bayes ngây thơ 838
19.8.4 Đào tạo 839
19.8.5 Tóm tắt * Sử dụng quy tắc của Bayes', một phân loại có thể được thực hiện bằng cách giả sử tất cả các tính năng quan sát là độc lập. * Phân loại này có thể được đào tạo trên một tập dữ liệu bằng cách đếm số lần xuất hiện của các kết hợp của nhãn và giá trị pixel * Phân loại này là tiêu chuẩn vàng trong nhiều thập kỷ cho các tác vụ như thư rác phát hiện. 842

19.8.6 Bài tập 1. Xem xét tập dữ liệu $[[0,0],[0,1],[1,0],[1,1]]$ với các nhãn được đưa ra bởi XOR của hai phần tử $[0,1,1,0]$. Xác suất cho một phân loại ngây thơ Bayes được xây dựng trên bộ dữ liệu này là gì. Nó có phân loại thành công điểm của chúng tôi không? Nếu không, những giả định nào bị vi phạm? 1. Giả sử rằng chúng tôi đã không sử dụng Laplace làm mịn khi ước tính xác suất và một ví dụ dữ liệu đến lúc thử nghiệm trong đó có một giá trị không bao giờ quan sát thấy trong đào tạo. Sản lượng mô hình sẽ là gì? 1. Phân loại Bayes ngây thơ là một ví dụ cụ thể của một mạng Bayesian, trong đó sự phụ thuộc của các biến ngẫu nhiên được mã hóa bằng cấu trúc đồ thị. Trong khi lý thuyết đầy đủ nằm ngoài phạm vi của phần này (xem [Koller.Friedman.2009] để biết chi tiết đầy đủ), giải thích tại sao cho phép phụ thuộc rõ ràng giữa hai biến đầu vào trong mô hình XOR cho phép tạo ra một phân loại thành công. . . . 842
19.9 Thống kê . . . 843
19.9.1 Đánh giá và so sánh ước tính . . . 843
19.9.2 Tiến hành các bài kiểm tra giả thuyết . . . 847
19.9.3 Xây dựng khoảng tự tin . . . 851
19.9.4 Tóm tắt . . . 853
19.9.5 Bài tập . . . 853
19.10 Lý thuyết thông tin . . . 854
19.10.1 Thông tin . . . 854
19.10.2 Entropy . . . 855
19.10.3 Thông tin lẫn nhau . . . 858
19.10.4 Kullback — Phân kỳ Leibler . . . 862
19.10.5 Cross-Entropy . . . 864
19.10.6 Tóm tắt . . . 867
19.10.7 Bài tập . . . 867

20 Phụ lục: Các công cụ cho Deep Learning **869**
20.1 Sử dụng Jupyter . . . 869
20.1.1 Chỉnh sửa và chạy mã cục bộ . . . 869
20.1.2 Tùy chọn nâng cao . . . 873
20.1.3 Tóm tắt . . . 874
20.1.4 Bài tập . . . 874
20.2 Sử dụng Amazon SageMaker . . . 874
20.2.1 Đăng ký và đăng nhập . . . 874
20.2.2 Tạo một phiên bản SageMaker . . . 875
20.2.3 Chạy và dừng phiên bản . . . 876
20.2.4 Cập nhật máy tính xách tay . . . 877
20.2.5 Tóm tắt . . . 877
20.2.6 Bài tập . . . 878
20.3 Sử dụng phiên bản AWS EC2 . . . 878
20.3.1 Tạo và chạy Phiên bản EC2 . . . 878
20.3.2 Cài đặt CDA . . . 883
20.3.3 Cài đặt MXNet và Tải xuống máy tính xách tay D2L . . . 884
20.3.4 Chạy Jupyter . . . 885
20.3.5 Đóng phiên bản chưa sử dụng . . . 885
20.3.6 Tóm tắt . . . 886
20.3.7 Bài tập . . . 886
20.4 Sử dụng Google Colab . . . 886
20.4.1 Tóm tắt . . . 887
20.4.2 Bài tập . . . 887

20.5 Chọn máy chủ và GPU . . . 887
20.5.1 Chọn máy chủ . . . 887
20.5.2 Chọn GPU . . . 888
20.5.3 Tóm tắt . . . 890
20.6 Đóng góp cho cuốn sách này . . . 891
20.6.1 Thay đổi văn bản nhỏ . . . 891
20.6.2 Đề xuất một sự thay đổi lớn . . . 892
20.6.3 Thêm một phần mới hoặc một triển khai khung mới . . . 893
20.6.4 Gửi một thay đổi lớn . . . 893
20.6.5 Tóm tắt . . . 896
20.6.6 Bài tập . . . 896
20.7 d2l Tài liệu API . . . 896

Bibliography **927**

Python Module Index **935**

Index **937**

Lời nói đầu

Chỉ vài năm trước, không có quân đoàn của các nhà khoa học học sâu phát triển các sản phẩm và dịch vụ thông minh tại các công ty lớn và công ty khởi nghiệp. Khi chúng tôi bước vào lĩnh vực này, học máy đã không ra lệnh tiêu đề trên các tờ báo hàng ngày. Cha mẹ chúng ta không biết học máy là gì, chứ đừng nói đến lý do tại sao chúng ta có thể thích nó hơn một nghề nghiệp trong y học hoặc luật pháp. Học máy là một kỷ luật học thuật bầu trời xanh có ý nghĩa công nghiệp bị giới hạn trong một tập hợp hẹp các ứng dụng trong thế giới thực, bao gồm nhận dạng giọng nói và tầm nhìn máy tính. Hơn nữa, nhiều ứng dụng trong số này đòi hỏi rất nhiều kiến thức về miền đến mức chúng thường được coi là các lĩnh vực hoàn toàn riêng biệt mà machine learning là một thành phần nhỏ. Vào thời điểm đó, mạng thần kinh - những người tiền nhiệm của các phương pháp học sâu mà chúng ta tập trung vào trong cuốn sách này - thường được coi là lỗi thời.

Chỉ trong năm năm qua, deep learning đã gây bất ngờ cho thế giới, thúc đẩy tiến bộ nhanh chóng trong các lĩnh vực đa dạng như thị giác máy tính, xử lý ngôn ngữ tự nhiên, nhận dạng giọng nói tự động, học củng cố và tin học y sinh. Hơn nữa, sự thành công của học sâu về rất nhiều nhiệm vụ quan tâm thực tế thậm chí đã xúc tác những phát triển trong học máy lý thuyết và thống kê. Với những tiến bộ này trong tay, giờ đây chúng ta có thể chế tạo những chiếc xe tự lái với quyền tự chủ hơn bao giờ hết (và ít tự chủ hơn một số công ty có thể có bạn tin), hệ thống trả lời thông minh tự động soạn thảo các email trần tục nhất, giúp mọi người đào thoát khỏi các hộp thư đến lớn áp bức và phần mềm các đặc vụ thống trị con người giỏi nhất thế giới tại các trò chơi trên bàn như Go, một kỳ công từng được cho là cách xa hàng thập kỷ. Đã có, những công cụ này tạo ra những tác động ngày càng rộng hơn đến ngành công nghiệp và xã hội, thay đổi cách tạo ra phim ảnh, chẩn đoán bệnh tật và đóng vai trò ngày càng tăng trong các khoa học cơ bản — từ vật lý thiên văn đến sinh học.

Về cuốn sách này

Cuốn sách này đại diện cho nỗ lực của chúng tôi để làm cho việc học sâu dễ tiếp cận, dạy cho bạn các khái niệm *, *context*, và *mã*.

Một phương tiện kết hợp mã, toán, và HTML

Để bất kỳ công nghệ điện toán nào đạt được tác động đầy đủ của nó, nó phải được hiểu rõ, tài liệu tốt và được hỗ trợ bởi các công cụ trưởng thành, được duy trì tốt. Những ý tưởng quan trọng cần được chưng cất rõ ràng, giảm thiểu thời gian lên máy bay cần đưa các học viên mới cập nhật. Thư viện trưởng thành nên tự động hóa các tác vụ phổ biến, và mã mẫu sẽ giúp các học viên dễ dàng sửa đổi, áp dụng và mở rộng các ứng dụng phổ biến cho phù hợp với nhu cầu của họ. Lấy các ứng dụng web động làm ví dụ. Mặc dù có một số lượng lớn các công ty, như Amazon, phát triển các ứng dụng web dựa trên cơ sở dữ liệu thành công trong những năm 1990, tiềm năng của công nghệ này để hỗ trợ các doanh nhân sáng tạo đã được nhận ra ở mức độ lớn hơn nhiều trong mười năm qua, do một phần phát triển mạnh mẽ, có tài liệu tốt khuôn khổ.

Kiểm tra tiềm năng của deep learning đưa ra những thách thức độc đáo bởi vì bất kỳ ứng dụng duy nhất nào tập hợp các ngành khác nhau. Áp dụng học sâu đòi hỏi sự hiểu biết đồng thời (i) động lực để đúc một vấn đề

theo một cách cụ thể; (ii) hình thức toán học của một mô hình nhất định; (iii) các thuật toán tối ưu hóa để phù hợp với các mô hình với dữ liệu; (iv) các nguyên tắc thống kê cho chúng tôi biết khi nào chúng ta nên mong đợi các mô hình của chúng tôi để khái quát hóa dữ liệu vô hình và các phương pháp thực tế để xác nhận rằng chúng có, trên thực tế, khái quát hóa; và (v) các kỹ thuật kỹ thuật cần thiết để đào tạo mô hình một cách hiệu quả, điều hướng những cạm bẫy của máy tính số và tận dụng tối đa phần cứng có sẵn. Dạy cả kỹ năng tư duy phê phán cần thiết để xây dựng các vấn đề, toán học để giải quyết chúng và các công cụ phần mềm để thực hiện các giải pháp đó ở một nơi đều thể hiện những thách thức đáng gờm. Mục tiêu của chúng tôi trong cuốn sách này là trình bày một nguồn tài nguyên thống nhất để mang lại cho các học viên sẽ được tăng tốc.

Khi chúng tôi bắt đầu dự án sách này, không có tài nguyên nào đồng thời (i) được cập nhật; (ii) bao phủ toàn bộ bề rộng của máy học hiện đại với chiều sâu kỹ thuật đáng kể; và (iii) trình bày xen kẽ về chất lượng mà người ta mong đợi từ một cuốn sách giáo khoa hấp dẫn với mã runnable sạch mà một hy vọng sẽ tìm thấy trong hướng dẫn thực hành. Chúng tôi tìm thấy rất nhiều ví dụ về cách sử dụng một khuôn khổ học sâu nhất định (ví dụ: cách thực hiện điện toán số cơ bản với ma trận trong TensorFlow) hoặc để thực hiện các kỹ thuật cụ thể (ví dụ: đoạn mã cho LeNet, AlexNet, ResNet, v.v.) nằm rải rác trên các bài đăng blog khác nhau và kho GitHub. Tuy nhiên, những ví dụ này thường tập trung vào *làm thế nào* để thực hiện một cách tiếp cận nhất định, nhưng để lại cuộc thảo luận về *tại sao các quyết định thuật toán nhất định được đưa ra. Mặc dù một số tài nguyên tương tác đã xuất hiện lẻ tẻ để giải quyết một chủ đề cụ thể, ví dụ, các bài đăng trên blog hấp dẫn được xuất bản trên trang web Distill[1] hoặc blog cá nhân, chúng chỉ đề cập đến các chủ đề được chọn trong học sâu và thường thiếu mã liên quan. Mặt khác, trong khi một số sách giáo khoa học sâu đã xuất hiện — ví dụ, (Goodfellow et al., 2016), cung cấp một cuộc khảo sát toàn diện về những điều cơ bản về học sâu — những tài nguyên này không kết hôn với các mô tả để nhận ra các khái niệm trong mã, đôi khi khiến độc giả không biết gì về cách thực hiện chúng. Hơn nữa, quá nhiều tài nguyên được ẩn đằng sau các bức tường trả tiền của các nhà cung cấp khóa học thương mại.

Chúng tôi đặt ra để tạo ra một tài nguyên có thể (i) được cung cấp miễn phí cho tất cả mọi người; (ii) cung cấp đủ chiều sâu kỹ thuật để cung cấp một điểm khởi đầu trên con đường để thực sự trở thành một nhà khoa học máy học ứng dụng; (iii) bao gồm mã runnable, hiển thị độc giả *làm thế nào* để giải quyết các vấn đề trong thực tế; (iv) cho phép cập nhật nhanh chóng, bởi cả chúng tôi và cả cộng đồng nói chung; và (v) được bổ sung bởi một forum[2] để thảo luận tương tác về chi tiết kỹ thuật và trả lời các câu hỏi.

Những mục tiêu này thường bị xung đột. Phương trình, định lý và trích dẫn được quản lý tốt nhất và đặt ra trong LateX. Mã được mô tả tốt nhất trong Python. Và các trang web có nguồn gốc trong HTML và javascript. Hơn nữa, chúng tôi muốn nội dung có thể truy cập cả dưới dạng mã thực thi, dưới dạng sách vật lý, dưới dạng PDF có thể tải xuống và trên Internet dưới dạng trang web. Hiện tại không có công cụ và không có quy trình làm việc hoàn toàn phù hợp với những nhu cầu này, vì vậy chúng tôi phải lắp ráp riêng của chúng tôi. Chúng tôi mô tả cách tiếp cận của chúng tôi một cách chi tiết trong Section 20.6. Chúng tôi định cư trên GitHub để chia sẻ nguồn và tạo điều kiện cho cộng đồng đóng góp, máy tính xách tay Jupyter để trộn mã, phương trình và văn bản, Sphinx như một công cụ kết xuất để tạo ra nhiều đầu ra và Discourse cho diễn đàn. Mặc dù hệ thống của chúng tôi chưa hoàn hảo, nhưng những lựa chọn này cung cấp một sự thỏa hiệp tốt giữa các mối quan tâm cạnh tranh. Chúng tôi tin rằng đây có thể là cuốn sách đầu tiên được xuất bản bằng cách sử dụng quy trình làm việc tích hợp như vậy.

[1] http://distill.pub
[2] http://discuss.d2l.ai

Học bằng cách làm

Nhiều sách giáo khoa trình bày các khái niệm liên tiếp, bao gồm từng chi tiết đầy đủ. Ví dụ, sách giáo khoa xuất sắc của Chris Bishop (Bishop, 2006), dạy từng chủ đề kỹ lưỡng đến mức nhận được chương về hồi quy tuyến tính đòi hỏi một lượng công việc không tầm thường. Trong khi các chuyên gia yêu thích cuốn sách này chính xác vì sự triệt để của nó, đối với những người mới bắt đầu thực sự, tài sản này hạn chế tính hữu ích của nó như một văn bản giới thiệu.

Trong cuốn sách này, chúng tôi sẽ dạy hầu hết các khái niệm * chỉ trong thời gian*. Nói cách khác, bạn sẽ học các khái niệm vào lúc này rằng chúng cần thiết để hoàn thành một số kết thúc thực tế. Trong khi chúng tôi mất một thời gian ngay từ đầu để dạy các sơ bộ cơ bản, như đại số tuyến tính và xác suất, chúng tôi muốn bạn nếm thử sự hài lòng khi đào tạo mô hình đầu tiên của mình trước khi lo lắng về phân phối xác suất bí truyền hơn.

Bên cạnh một vài sổ ghi chép sơ bộ cung cấp một khóa học sụp đổ trong nền toán học cơ bản, mỗi chương tiếp theo giới thiệu cả một số khái niệm mới hợp lý và cung cấp các ví dụ làm việc khép kín duy nhất—sử dụng bộ dữ liệu thực. Điều này trình bày một thách thức tổ chức. Một số mô hình có thể được nhóm lại với nhau một cách hợp lý trong một sổ ghi chép duy nhất. Và một số ý tưởng có thể được dạy tốt nhất bằng cách thực hiện một số mô hình liên tiếp. Mặt khác, có một lợi thế lớn để tuân thủ chính sách của * một ví dụ làm việc, một máy tính xách túc*: Điều này giúp bạn dễ dàng bắt đầu các dự án nghiên cứu của riêng mình bằng cách tận dụng mã của chúng tôi. Chỉ cần sao chép một sổ ghi chép và bắt đầu sửa đổi nó.

Chúng tôi sẽ interleave mã runnable với tài liệu nền khi cần thiết. Nói chung, chúng ta thường sẽ sai về phía làm cho các công cụ có sẵn trước khi giải thích chúng đầy đủ (và chúng tôi sẽ theo dõi bằng cách giải thích nền sau). Ví dụ: chúng ta có thể sử dụng *stochastic gradient descent* trước khi giải thích đầy đủ lý do tại sao nó hữu ích hoặc tại sao nó hoạt động. Điều này giúp cho các học viên những đạn dược cần thiết để giải quyết vấn đề một cách nhanh chóng, với chi phí yêu cầu người đọc tin tưởng chúng tôi với một số quyết định giám tuyển.

Cuốn sách này sẽ dạy các khái niệm học sâu từ đầu. Đôi khi, chúng tôi muốn đi sâu vào các chi tiết tốt về các mô hình thường sẽ bị ẩn khỏi người dùng bằng các trừu tượng nâng cao của khung học sâu. Điều này đặc biệt xuất hiện trong các hướng dẫn cơ bản, nơi chúng tôi muốn bạn hiểu mọi thứ xảy ra trong một lớp hoặc trình tối ưu hóa nhất định. Trong những trường hợp này, chúng tôi thường sẽ trình bày hai phiên bản của ví dụ: một trong đó chúng tôi thực hiện mọi thứ từ đầu, chỉ dựa vào chức năng giống như NumPy và sự khác biệt tự động, và một ví dụ khác, thực tế hơn, nơi chúng tôi viết mã ngắn gọn bằng cách sử dụng API cấp cao của các khuôn khổ học sâu. Khi chúng tôi đã dạy bạn cách thức hoạt động của một số component, chúng ta chỉ có thể sử dụng các API cấp cao trong các hướng dẫn tiếp theo.

Nội dung và cấu trúc

Cuốn sách có thể được chia thành ba phần, tập trung vào sơ bộ, kỹ thuật học sâu, và các chủ đề nâng cao tập trung vào các hệ thống và ứng dụng thực tế (Fig. 1).

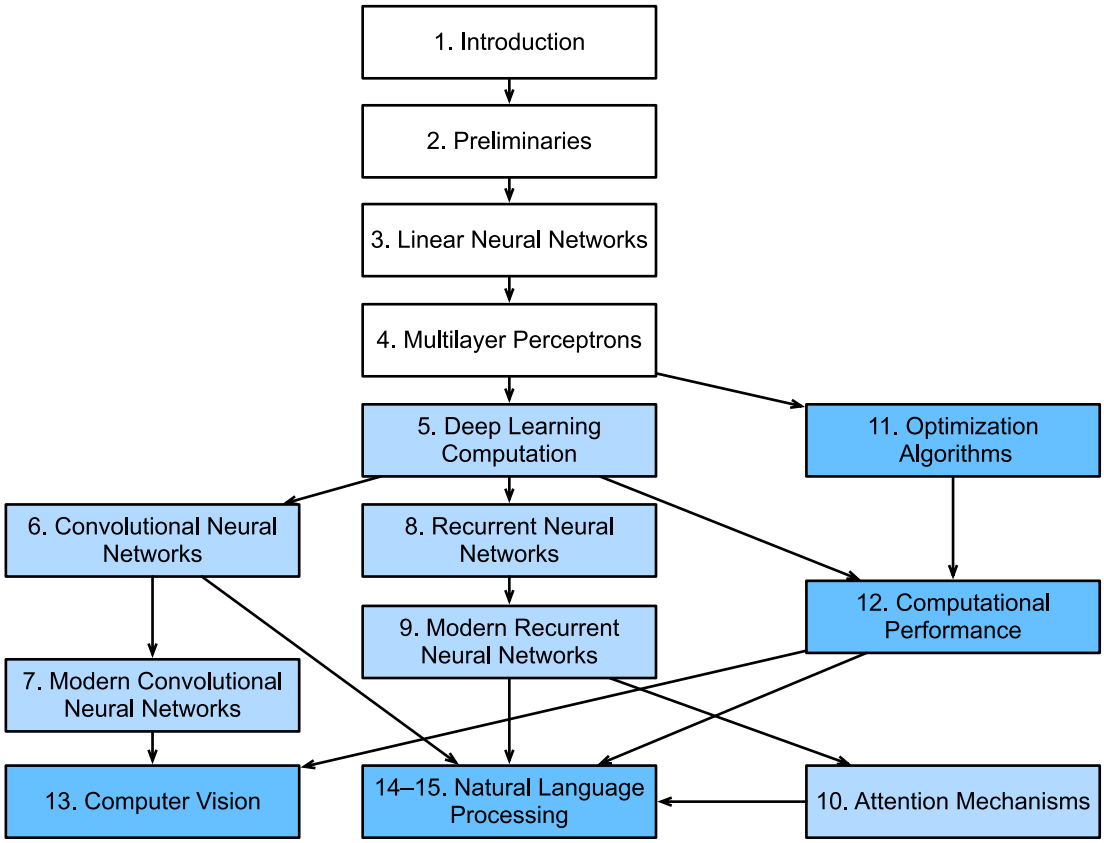


Fig. 1: Book structure

- Phần đầu tiên bao gồm những điều cơ bản và sơ bộ. Chapter 2 cung cấp một giới thiệu về học sâu. Sau đó, vào năm Chapter 3, chúng tôi nhanh chóng đưa bạn tăng tốc các điều kiện tiên quyết cần thiết cho việc học sâu thực hành, chẳng hạn như cách lưu trữ và thao tác dữ liệu và cách áp dụng các phép toán số khác nhau dựa trên các khái niệm cơ bản từ đại số tuyến tính, tính toán và xác suất. Chapter 4 và Chapter 5 bao gồm nhiều nhất các khái niệm và kỹ thuật cơ bản trong học sâu, bao gồm hồi quy và phân loại; mô hình tuyến tính và nhận thức đa lớp; và overfitting và regarization.
- Năm chương tiếp theo tập trung vào các kỹ thuật học sâu hiện đại. Chapter 6 mô tả các thành phần tính toán chính của các hệ thống học sâu và đặt nền tảng cho việc triển khai các mô hình phức tạp hơn tiếp theo của chúng tôi. Tiếp theo, Chapter 7 và Chapter 8, giới thiệu các mạng thần kinh phức tạp (CNN), các công cụ mạnh mẽ tạo thành xương sống của hầu hết các hệ thống thị giác máy tính hiện đại. Tương tự, Chapter 9 và Chapter 10 giới thiệu các mạng nơ-ron tái phát (RNNs), các mô hình khai thác cấu trúc tuần tự (ví dụ, tạm thời) trong dữ liệu và thường được sử dụng để xử lý ngôn ngữ tự nhiên và dự đoán chuỗi thời gian. Năm Chapter 11, chúng tôi giới thiệu một lớp mô hình tương đối mới dựa trên cái gọi là cơ chế chú ý đã thay thế RNN là kiến trúc thống trị cho hầu hết các nhiệm vụ xử lý ngôn ngữ tự nhiên. Những phần này sẽ mang lại cho bạn tốc độ trên các công cụ mạnh mẽ và chung nhất được sử dụng rộng rãi bởi các học viên học sâu.
- Phần ba thảo luận về khả năng mở rộng, hiệu quả và ứng dụng. Đầu tiên, vào năm Chapter 12, chúng tôi thảo luận về một số thuật toán tối ưu hóa phổ biến được sử dụng để đào tạo các mô hình học sâu. Chương tiếp theo, Chapter 13, xem xét một số yếu tố chính ảnh hưởng đến hiệu suất tính toán của mã

học sâu của bạn. Trong Chapter 14, chúng tôi minh họa các ứng dụng chính của deep learning trong tầm nhìn máy tính. Trong Chapter 15 và Chapter 16, chúng tôi chỉ ra cách chuẩn bị các mô hình biểu diễn ngôn ngữ và áp dụng chúng vào các nhiệm vụ xử lý ngôn ngữ tự nhiên.

Mã

Hầu hết các phần của cuốn sách này đều có mã thực thi. Chúng tôi tin rằng một số trực giác được phát triển tốt nhất thông qua thử và sai, tinh chỉnh mã theo những cách nhỏ và quan sát kết quả. Lý tưởng nhất, một lý thuyết toán học thanh lịch có thể cho chúng ta biết chính xác làm thế nào để tinh chỉnh mã của chúng tôi để đạt được kết quả mong muốn. Tuy nhiên, ngày nay các học viên học sâu ngày nay phải thường bước đi nơi không có lý thuyết cogent nào có thể cung cấp hướng dẫn vững chắc. Bất chấp những nỗ lực tốt nhất của chúng tôi, những lời giải thích chính thức về hiệu quả của các kỹ thuật khác nhau vẫn còn thiếu, cả vì toán học để mô tả các mô hình này có thể rất khó khăn và cũng bởi vì cuộc điều tra nghiêm túc về các chủ đề này chỉ mới bắt đầu vào thiết bị cao. Chúng tôi hy vọng rằng khi lý thuyết học sâu tiến triển, các phiên bản trong tương lai của cuốn sách này có thể cung cấp những hiểu biết mà nhật thực những người hiện có.

Để tránh sự lặp lại không cần thiết, chúng tôi đóng gói một số hàm và lớp được nhập và tham chiếu thường xuyên nhất của chúng tôi trong gói `d2l`. Để chỉ ra một khối mã, chẳng hạn như hàm, lớp hoặc tập hợp các câu lệnh import, sau đó sẽ được truy cập thông qua gói `d2l`, chúng ta sẽ đánh dấu nó bằng `# @save`. Chúng tôi cung cấp một cái nhìn tổng quan chi tiết về các chức năng và lớp trong Section 20.7. Gói `d2l` có trọng lượng nhẹ và chỉ yêu cầu các phụ thuộc sau:

```
#@save
import collections
import hashlib
import math
import os
import random
import re
import shutil
import sys
import tarfile
import time
import zipfile
from collections import defaultdict
import pandas as pd
import requests
from IPython import display
from matplotlib import pyplot as plt
from matplotlib_inline import backend_inline

d2l = sys.modules[__name__]
```

Hầu hết các mã trong cuốn sách này dựa trên PyTorch, một khuôn khổ mã nguồn mở cực kỳ phổ biến đã được cộng đồng nghiên cứu sâu đón nhận nhiệt tình. Tất cả các mã trong cuốn sách này đã vượt qua các bài kiểm tra theo phiên bản ổn định mới nhất của PyTorch. Tuy nhiên, do sự phát triển nhanh chóng của deep learning, một số mã*trong phiên bản in* có thể không hoạt động đúng trong các phiên bản PyTorch trong tương lai. Chúng tôi dự định cập nhật phiên bản trực tuyến. Trong trường hợp bạn gặp bất kỳ vấn đề nào, vui lòng tham khảo *Cài đặt* (page 9) để cập nhật mã và môi trường thời gian chạy của bạn.

Dưới đây là cách chúng tôi nhập các mô-đun từ PyTorch.

```
#@save
import numpy as np
import torch
import torchvision
from PIL import Image
from torch import nn
from torch.nn import functional as F
from torch.utils import data
from torchvision import transforms
```

Đối tượng mục tiêu

Cuốn sách này dành cho sinh viên (đại học hoặc sau đại học), kỹ sư và nhà nghiên cứu, những người tìm kiếm một nắm vững chắc về các kỹ thuật thực tế của học sâu. Bởi vì chúng tôi giải thích mọi khái niệm từ đầu, không có nền tảng trước đó trong học sâu hoặc học máy là bắt buộc. Giải thích đầy đủ các phương pháp học sâu đòi hỏi một số toán học và lập trình, nhưng chúng tôi sẽ chỉ cho rằng bạn đến với một số điều cơ bản, bao gồm một lượng khiêm tốn của đại số tuyến tính, giải tích, xác suất, và lập trình Python. Chỉ trong trường hợp bạn quên những điều cơ bản, Phụ lục cung cấp một bồi dưỡng về hầu hết các toán học bạn sẽ tìm thấy trong cuốn sách này. Hầu hết thời gian, chúng ta sẽ ưu tiên trực giác và ý tưởng hơn sự nghiêm ngặt về toán học. Nếu bạn muốn mở rộng các nền tảng này ngoài các điều kiện tiên quyết để hiểu cuốn sách của chúng tôi, chúng tôi vui vẻ đề xuất một số tài nguyên tuyệt vời khác: Phân tích tuyến tính của Bela Bollobas (Bollobas, 1999) bao gồm đại số tuyến tính và phân tích chức năng ở độ sâu lớn. Tất cả các thống kê (Wasserman, 2013) cung cấp một giới thiệu tuyệt vời về thống kê. books[3] của Joe Blitzstein và courses[4] về xác suất và suy luận là đá quý sư phạm. Và nếu bạn chưa sử dụng Python trước đây, bạn có thể muốn xem xét Python tutorial[5] này.

Diễn đàn

Liên quan đến cuốn sách này, chúng tôi đã đưa ra một diễn đàn thảo luận, đặt tại discuss.d2l.ai[6]. Khi bạn có câu hỏi về bất kỳ phần nào của cuốn sách, bạn có thể tìm thấy một liên kết đến trang thảo luận liên quan ở cuối mỗi sổ ghi chép.

Lời ghi nhận

Chúng tôi đang mắc nợ hàng trăm người đóng góp cho cả bản nháp tiếng Anh và tiếng Trung. Họ đã giúp cải thiện nội dung và cung cấp phản hồi có giá trị. Cụ thể, chúng tôi cảm ơn mọi người đóng góp của dự thảo tiếng Anh này đã làm cho nó tốt hơn cho tất cả mọi người. ID hoặc tên GitHub của họ là (không theo thứ tự cụ thể): alxnorden, avinashingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Stoui, Michael (u) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, buồn, sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, bún, Vishaal Kapoor, Vishwesh Ravi Shrimali, YaYaB, Yuhong Chen, Evgeniy Smirnov, lgov, Simon Corston-Oliver, Igor Dzreyev, Hà Nguyên, Lupmuens, Andrei venko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, Prasanth Buddareddygari, brianhendee, mani2106, mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim,

[3] https://www.amazon.com/Introduction-Probability-Chapman-Statistical-Science/dp/1138369918
[4] https://projects.iq.harvard.edu/stat110/home
[5] http://learnpython.org/
[6] https://discuss.d2l.ai/

dennismalmgren, adursun, Anirqudh Dagar, LiAnh, Pedro Larroy, lgov, và ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Hồi giáo Rakib, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, Ruslan Baratov, Rafael Schlatter, liusy182, Giannis Pappas, ati-ozgur, qbaza, dchoi77, Adam Gerson, Phúc Lê, Mark Atwood, christabella, vn09, Haibin Lin, jjangga0214, RichyChen, Noelo, hansent, Giel Dops, dvincent1337, WhiteD3vil , Peter Kulit, codypenta, joseppinilla, ahmaurya, karolszk, heytitle, Peter Goetz, rigtorp, Tiep Vu, sfilip, mlxd, Kale-ab Tessera, Sanjar Adilov, MatteoFerrara, hsneto, Katarzyna Biesialska, Gregory Bruss, Duy—Thanh Đoàn, paulaurel, graytowne, Đức Phạm, sl7423, Jaedong Hwang, Yida Wang, cys4, cys4, cysm, Jean Kaddour , austinmw, trebeljahr, tbaums, Cường V Nguyễn, pavelkomarov, vzlamal, NotAnotherSystem, J-Arun-Mani, jancio, eldarkurtic, the great-shazbot, doctorcolossus, gducharme, cclauss, Daniel-Mietchen, hoonose, avagiom, abhinagiom sp0730, jonathanhrandall, ysraell, Nodar Okroshiashvili, UgurKap, Jiyang Kang, StevenJokes, Tomer Kaftan, liweiwp, netyster, ypandya, NishantTharani, heiligerl, SportsTHU, Hoa Nguyen, manuel-arno-korfmann-webentwicklung, aterzis-personal, nxby, Xiaoting He, Josiah Yoder, mathresearch, mzz2017, jroberayalas, iluu, ejc, BSharmi, vkramdev, simonwardjones, LakshKD, TalNeoran, Djliden, Nikhil95, Oren Barkan, guoweis, haozhu233, pratikhack, Yue Ying, tayfununal, steinsag, charleybeller, Andrew Lumsdaine, Jiekui Zhang, Deepak Pathak, Florian Donhauser, Tim Gates, Adriaan Tijsseling, Ron Medina, Gaurav Saha, Murat Semerci, Lei Mao, Levi McClamenny, Joshua Broyde, jake221, jonbally, zyhazwraith, Brian Pulfer, Nick Tomasino, Lefan Zhang, Hongshen Yang, Vinney Cavallo, Yuntai, Yuanxiang Zhu, Amarazov, Pasricha, Ben Greenawald, Shivam Upadhyay, Quanshangze Du, Biswajit Sahoo, Parthe Pandit, Ishan Kumar, HomunculusK, Lane Schwartz, Varadgunjal, Jason Wiener, Armin Gholampoor, Shreshtha13, Eigen-arnav, Hyeonggyu Kim, EmilyOng, Bálint Mucsányi, Chase DuBois.

Chúng tôi cảm ơn Amazon Web Services, đặc biệt là Swami Sivasubramanian, Peter DeSantis, Adam Selipsky và Andrew Jassy vì sự hỗ trợ hào phóng của họ trong việc viết cuốn sách này. Nếu không có thời gian có sẵn, nguồn lực, thảo luận với đồng nghiệp và khuyến khích liên tục, cuốn sách này sẽ không xảy ra.

Tóm tắt

- Deep learning đã cách mạng hóa nhận dạng mẫu, giới thiệu công nghệ hiện nay cung cấp năng lượng cho một loạt các công nghệ, bao gồm tầm nhìn máy tính, xử lý ngôn ngữ tự nhiên, nhận dạng giọng nói tự động.
- Để áp dụng thành công deep learning, bạn phải hiểu làm thế nào để giải quyết một vấn đề, toán học của mô hình hóa, các thuật toán để phù hợp với mô hình của bạn với dữ liệu, và các kỹ thuật kỹ thuật để thực hiện tất cả.
- Cuốn sách này trình bày một nguồn tài nguyên toàn diện, bao gồm văn xuôi, số liệu, toán học và mã, tất cả ở một nơi.
- Để trả lời các câu hỏi liên quan đến cuốn sách này, hãy truy cập diễn đàn của chúng tôi tại https://discuss.d2l.ai/.
- Tất cả các máy tính xách tay đều có sẵn để tải xuống trên GitHub.

Bài tập

1. Đăng ký một tài khoản trên diễn đàn thảo luận của cuốn sách này discuss.d2l.ai[7].
2. Cài đặt Python trên máy tính của bạn.
3. Thực hiện theo các liên kết ở dưới cùng của phần đến diễn đàn, nơi bạn sẽ có thể tìm kiếm sự giúp đỡ và thảo luận về cuốn sách và tìm câu trả lời cho câu hỏi của bạn bằng cách thu hút các tác giả và cộng đồng rộng hơn.

Discussions[8]

[7] https://discuss.d2l.ai/
[8] https://discuss.d2l.ai/t/20

Cài đặt

Để giúp bạn có được và chạy trải nghiệm học tập thực hành, chúng tôi cần thiết lập cho bạn một môi trường để chạy Python, máy tính xách tay Jupyter, các thư viện có liên quan và mã cần thiết để tự chạy cuốn sách.

Cài đặt Miniconda

Cách đơn giản nhất để có được đi sẽ là cài đặt Miniconda[9]. Phiên bản Python 3.x là bắt buộc. Bạn có thể bỏ qua các bước sau nếu máy của bạn đã cài đặt conda.

Truy cập trang web Miniconda và xác định phiên bản thích hợp cho hệ thống của bạn dựa trên phiên bản Python 3.x và kiến trúc máy của bạn. Ví dụ: nếu bạn đang sử dụng macOS và Python 3.x, bạn sẽ tải xuống tập lệnh bash có tên chứa chuỗi "Miniconda3" và "MacOSX", điều hướng đến vị trí tải xuống và thực hiện cài đặt như sau:

```
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

Một người dùng Linux với Python 3.x sẽ tải xuống tệp có tên chứa chuỗi "Miniconda3" và "Linux" và thực hiện như sau tại vị trí tải xuống:

```
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Tiếp theo, khởi tạo shell để chúng ta có thể chạy trực tiếp `conda`.

```
~/miniconda3/bin/conda init
```

Bây giờ đóng và mở lại vỏ hiện tại của bạn. Bạn sẽ có thể tạo ra một môi trường mới như sau:

[9] https://conda.io/en/latest/miniconda.html

```
conda create --name d2l python=3.8 -y
```

Tải xuống máy tính xách tay D2L

Tiếp theo, chúng ta cần tải xuống mã của cuốn sách này. Bạn có thể nhấp vào tab "Tất cả máy tính xách tay" ở đầu bất kỳ trang HTML nào để tải xuống và giải nén mã. Ngoài ra, nếu bạn có `unzip` (nếu không chạy `sudo apt install unzip`) có sẵn:

```
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Bây giờ chúng ta có thể kích hoạt môi trường `d2l`:

```
conda activate d2l
```

Cài đặt Framework và gói `d2l`

Trước khi cài đặt bất kỳ khung học sâu nào, trước tiên hãy kiểm tra xem bạn có GPU thích hợp trên máy tính của mình hay không (GPU cung cấp năng lượng cho màn hình trên máy tính xách tay tiêu chuẩn không phù hợp với mục đích của chúng tôi). Nếu bạn đang làm việc trên máy chủ GPU, hãy tiến hành *Hỗ trợ GPU* (page 11) để được hướng dẫn về cách cài đặt các phiên bản thân thiện với GPU của các thư viện có liên quan.

Nếu máy của bạn không chứa bất kỳ GPU nào, không cần phải lo lắng. CPU của bạn cung cấp quá đủ mã lực để giúp bạn vượt qua một vài chương đầu tiên. Chỉ cần nhớ rằng bạn sẽ muốn truy cập GPU trước khi chạy các mô hình lớn hơn. Để cài đặt phiên bản CPU, hãy thực hiện lệnh sau.

```
pip install torch torchvision
```

Bước tiếp theo của chúng tôi là cài đặt gói `d2l` mà chúng tôi đã phát triển để đóng gói các chức năng và lớp thường được sử dụng được tìm thấy trong suốt cuốn sách này.

```
# -U: Upgrade all packages to the newest available version
pip install -U d2l
```

Khi bạn đã hoàn thành các bước cài đặt này, chúng tôi có thể máy chủ máy tính xách tay Jupyter bằng cách chạy:

```
jupyter notebook
```

Tại thời điểm này, bạn có thể mở http://localhost:8888 (nó có thể đã tự động mở) trong trình duyệt Web của bạn. Sau đó, chúng ta có thể chạy mã cho mỗi phần của cuốn sách. Vui lòng luôn thực hiện `conda activate d2l` để kích hoạt môi trường thời gian chạy trước khi chạy mã của cuốn sách hoặc cập nhật khung học sâu hoặc gói `d2l`. Để thoát khỏi môi trường, chạy `conda deactivate`.

Hỗ trợ GPU

Theo mặc định, khung học sâu được cài đặt với hỗ trợ GPU. Nếu máy tính của bạn có GPU NVIDIA và đã cài đặt CUDA[10], thì tất cả bạn đã thiết lập.

Bài tập

1. Tải xuống mã cho cuốn sách và cài đặt môi trường thời gian chạy.

Discussions[11]

[10] https://developer.nvidia.com/cuda-downloads
[11] https://discuss.d2l.ai/t/24

1 | Ký hiệu

Trong suốt cuốn sách này, chúng tôi tuân thủ các quy ước công ước sau đây. Lưu ý rằng một số ký hiệu này là giữ chỗ, trong khi những ký hiệu khác đề cập đến các đối tượng cụ thể. Theo nguyên tắc chung của ngón tay cái, bài viết không xác định "a" chỉ ra rằng ký hiệu là một giữ chỗ và các ký hiệu được định dạng tương tự có thể biểu thị các đối tượng khác cùng loại. Ví dụ, "x: một vô hướng" có nghĩa là các chữ cái viết thường đại diện cho các giá trị vô hướng.

1.1 Các đối tượng số

- x: một vô hướng
- $\mathbf{x}$: một vector
- $\mathbf{X}$: một ma trận
- X: một tensor chung
- $\mathbf{I}$: một ma trận nhận dạng - vuông, với 1 trên tất cả các mục chéo và 0 trên tất cả các đường chéo
- x_i, $[\mathbf{x}]_i$: yếu tố i^{th} của vector $\mathbf{x}$
- x_{ij}, $x_{i,j}$,$[\mathbf{X}]_{ij}$, $[\mathbf{X}]_{i,j}$: phần tử của ma trận $\mathbf{X}$ tại hàng i và cột j.

1.2 Lý thuyết đặt

- $\mathcal{X}$: một bộ
- $\mathbb{Z}$: tập hợp các số nguyên
- $\mathbb{Z}^+$: tập hợp các số nguyên dương
- $\mathbb{R}$: tập hợp các số thực
- $\mathbb{R}^n$: tập hợp các vectơ n chiều của số thực
- $\mathbb{R}^{a\times b}$: Tập hợp các ma trận số thực với a hàng và b cột
- $|\mathcal{X}|$: cardinality (số phần tử) của bộ $\mathcal{X}$
- $\mathcal{A} \cup \mathcal{B}$: liên minh các bộ $\mathcal{A}$ và $\mathcal{B}$
- $\mathcal{A} \cap \mathcal{B}$: giao điểm của bộ $\mathcal{A}$ và $\mathcal{B}$
- $\mathcal{A} \setminus \mathcal{B}$: đặt phép trừ $\mathcal{B}$ từ $\mathcal{A}$ (chỉ chứa những yếu tố của $\mathcal{A}$ không thuộc về $\mathcal{B}$)

1.3 Chức năng và toán tử

- $f(\cdot)$: một chức năng
- $\log(\cdot)$: logarit tự nhiên (cơ sở e)
- $\log_2(\cdot)$: logarit với cơ sở 2
- $\exp(\cdot)$: hàm mũ
- $\mathbf{1}(\cdot)$: hàm chỉ báo, đánh giá thành 1 nếu đối số boolean là đúng và 0 nếu không
- $\mathbf{1}_{\mathcal{X}}(z)$: chức năng chỉ báo thành viên thiết lập, đánh giá 1 nếu phần tử z thuộc về bộ $\mathcal{X}$ và 0 nếu không
- $(\cdot)^\top$: transpose của một vectơ hoặc ma trận
- $\mathbf{X}^{-1}$: nghịch đảo của ma trận $\mathbf{X}$
- $\odot$: Sản phẩm Hadamard (elementwise)
- $[\cdot, \cdot]$: nối
- $\|\cdot\|_p$:L_p định mức
- $\|\cdot\|$:L_2 định mức
- $\langle \mathbf{x}, \mathbf{y} \rangle$: sản phẩm chấm của vectơ $\mathbf{x}$ và $\mathbf{y}$
- $\sum$: tổng kết về một bộ sưu tập các yếu tố
- $\prod$: sản phẩm trên một bộ sưu tập các yếu tố
- $\stackrel{\text{def}}{=}$: một sự bình đẳng khẳng định như một định nghĩa của ký hiệu ở phía bên trái

1.4 Calculus (Giải tích)

- $\frac{dy}{dx}$: dẫn xuất của y đối với x
- $\frac{\partial y}{\partial x}$: phái sinh một phần của y đối với x
- $\nabla_{\mathbf{x}} y$: gradient của y đối với $\mathbf{x}$
- $\int_a^b f(x)\, dx$: tích phân xác định của f từ a đến b đối với x
- $\int f(x)\, dx$: tích phân không xác định của f đối với x

1.5 Lý thuyết xác suất và thông tin

- X: một biến ngẫu nhiên
- P: một phân phối xác suất
- $X \sim P$: biến ngẫu nhiên X có phân phối P
- $P(X = x)$: xác suất được gán cho sự kiện mà biến ngẫu nhiên X lấy giá trị x
- $P(X \mid Y)$: phân phối xác suất có điều kiện của X cho Y
- $p(\cdot)$: một hàm mật độ xác suất (PDF) liên quan đến phân phối P

- $E[X]$: kỳ vọng của một biến ngẫu nhiên X
- $X \perp Y$: các biến ngẫu nhiên X và Y độc lập
- $X \perp Y \mid Z$: các biến ngẫu nhiên X và Y có điều kiện độc lập cho Z
- σ_X: độ lệch chuẩn của biến ngẫu nhiên X
- $\text{Var}(X)$: phương sai của biến ngẫu nhiên X, bằng σ_X^2
- $\text{Cov}(X, Y)$: đồng phương sai của các biến ngẫu nhiên X và Y
- $\rho(X, Y)$: hệ số tương quan Pearson giữa X và Y, bằng $\frac{\text{Cov}(X,Y)}{\sigma_X \sigma_Y}$
- $H(X)$: entropy của biến ngẫu nhiên X
- $D_{\text{KL}}(P\|Q)$: phân kỳ KL-phân kỳ (hoặc entropy tương đối) từ phân phối Q đến phân phối P

Discussions[12]

[12] https://discuss.d2l.ai/t/25

2 | Giới thiệu

Cho đến gần đây, gần như mọi chương trình máy tính mà chúng tôi tương tác với hàng ngày được mã hóa bởi các nhà phát triển phần mềm từ các nguyên tắc đầu tiên. Nói rằng chúng tôi muốn viết một ứng dụng để quản lý một nền tảng thương mại điện tử. Sau khi huddling xung quanh một bảng trắng trong một vài giờ để suy nghĩ về vấn đề, chúng tôi sẽ đưa ra những nét rộng của một giải pháp làm việc có thể trông giống như thế này: (i) người dùng tương tác với các ứng dụng thông qua một giao diện chạy trong một trình duyệt web hoặc ứng dụng di động; (ii) ứng dụng của chúng tôi tương tác với một công cụ cơ sở dữ liệu cấp thương mại để theo dõi trạng thái của từng người dùng và duy trì hồ sơ về các giao dịch lịch sử; và (iii) ở trung tâm của ứng dụng của chúng tôi, * business logic* (bạn có thể nói, *nãi*) ứng dụng của chúng tôi đánh vần chi tiết về phương pháp hành động thích hợp của chúng tôi program chương trình should take in every mỗi conceivable có thể tưởng tượng circumstance hoàn cảnh.

Để xây dựng bộ não của ứng dụng của chúng tôi, chúng tôi sẽ phải bước qua mọi trường hợp góc có thể mà chúng tôi dự đoán gặp phải, đưa ra các quy tắc phù hợp. Mỗi lần khách hàng nhấp chuột để thêm một mặt hàng vào giỏ hàng của họ, chúng tôi thêm một mục nhập vào bảng cơ sở dữ liệu giỏ hàng, liên kết ID của người dùng đó với ID của sản phẩm được yêu cầu. Trong khi vài nhà phát triển bao giờ nhận được nó hoàn toàn đúng lần đầu tiên (nó có thể mất một số thử nghiệm chạy để làm việc ra những trở ngại), đối với hầu hết các phần, chúng tôi có thể viết một chương trình như vậy từ nguyên tắc đầu tiên và tự tin khởi động nó *trước* bao giờ nhìn thấy một khách hàng thực sự. Khả năng thiết kế các hệ thống tự động của chúng tôi từ các nguyên tắc đầu tiên thúc đẩy các sản phẩm và hệ thống hoạt động, thường là trong các tình huống mới lạ, là một kỳ công nhận thức đáng chú ý. Và khi bạn có thể đưa ra các giải pháp hoạt động $100% $ thời gian, bạn không nên sử dụng máy học.

May mắn thay cho cộng đồng ngày càng tăng của các nhà khoa học máy học, nhiều nhiệm vụ mà chúng tôi muốn tự động hóa không dễ uốn cong đến sự khéo léo của con người. Hãy tưởng tượng xoay quanh bảng trắng với những bộ óc thông minh nhất mà bạn biết, nhưng lần này bạn đang giải quyết một trong những vấn đề sau:

- Viết một chương trình dự đoán thời tiết ngày mai được đưa ra thông tin địa lý, hình ảnh vệ tinh và một cửa sổ theo dõi thời tiết trong quá khứ.
- Viết một chương trình có trong một câu hỏi, thể hiện bằng văn bản dạng tự do, và trả lời nó một cách chính xác.
- Viết một chương trình đưa ra một hình ảnh có thể xác định tất cả những người mà nó chứa, vẽ phác thảo xung quanh mỗi.
- Viết một chương trình trình bày cho người dùng các sản phẩm mà họ có khả năng thưởng thức nhưng không thể, trong quá trình duyệt tự nhiên, để gặp phải.

Trong mỗi trường hợp này, ngay cả các lập trình viên ưu tú cũng không có khả năng mã hóa các giải pháp từ đầu. Những lý do cho điều này có thể khác nhau. Đôi khi chương trình mà chúng tôi đang tìm kiếm theo một mô hình thay đổi theo thời gian và chúng tôi cần các chương trình của mình để điều chỉnh. Trong các trường hợp khác, mối quan hệ (nói giữa pixel và danh mục trừu tượng) có thể quá phức tạp, đòi hỏi hàng ngàn hoặc hàng triệu tính toán vượt quá sự hiểu biết có ý thức của chúng ta ngay cả khi mắt chúng ta quản lý nhiệm vụ

dễ dàng. *Học máy* là nghiên cứu mạnh mẽ kỹ thuật mà có thể học hỏi từ kinh nghiệm. Là một thuật toán học máy tích lũy nhiều kinh nghiệm hơn, điển hình dưới dạng dữ liệu quan sát hoặc tương tác với một môi trường, hiệu suất của nó được cải thiện. Tương phản điều này với nền tảng thương mại điện tử xác định của chúng tôi, thực hiện theo cùng một logic kinh doanh, bất kể kinh nghiệm tích lũy bao nhiêu, cho đến khi các nhà phát triển tự học và quyết định rằng đã đến lúc cập nhật phần mềm. Trong cuốn sách này, chúng tôi sẽ dạy cho bạn những nguyên tắc cơ bản của máy học và đặc biệt tập trung vào * deep learning*, một bộ kỹ thuật mạnh mẽ thúc đẩy đổi mới trong các lĩnh vực đa dạng như thị giác máy tính, xử lý ngôn ngữ tự nhiên, chăm sóc sức khỏe và bộ gen.

2.1 Một ví dụ động lực

Trước khi bắt đầu viết, các tác giả của cuốn sách này, giống như nhiều lực lượng lao động, đã phải trở thành caffein. Chúng tôi nhảy vào xe và bắt đầu lái xe. Sử dụng iPhone, Alex gọi ra "Hey Siri", đánh thức hệ thống nhận dạng giọng nói của điện thoại. Sau đó Mục chỉ huy "hướng đến quán cà phê Blue Bottle". Điện thoại nhanh chóng hiển thị phiên mã lệnh của mình. Nó cũng nhận ra rằng chúng tôi đang yêu cầu chỉ đường và khởi chạy ứng dụng Maps (ứng dụng) để đáp ứng yêu cầu của chúng tôi. Sau khi ra mắt, ứng dụng Maps đã xác định được một số tuyến đường. Bên cạnh mỗi tuyến đường, điện thoại hiển thị thời gian vận chuyển dự đoán. Trong khi chúng tôi chế tạo câu chuyện này để thuận tiện sư phạm, nó chứng minh rằng trong khoảng thời gian chỉ vài giây, các tương tác hàng ngày của chúng tôi với một chiếc điện thoại thông minh có thể tham gia vào một số mô hình máy học.

Hãy tưởng tượng chỉ cần viết một chương trình để trả lời một từ *wake* như "Alexa", "OK Google", và "Hey Siri". Hãy thử tự mình mã hóa nó trong một căn phòng mà không có gì ngoài máy tính và trình soạn thảo mã, như minh họa trong Fig. 2.1.1. Làm thế nào bạn sẽ viết một chương trình như vậy từ các nguyên tắc đầu tiên? Hãy suy nghĩ về nó… vấn đề là khó khăn. Mỗi giây, micrô sẽ thu thập khoảng 44000 mẫu. Mỗi mẫu là một phép đo biên độ của sóng âm. Quy tắc nào có thể ánh xạ đáng tin cậy từ một đoạn âm thanh thô để dự đoán tự tin $\{\text{yes}, \text{no}\}$ về việc liệu đoạn mã có chứa từ thức hay không? Nếu bạn bị mắc kẹt, đừng lo lắng. Chúng tôi cũng không biết làm thế nào để viết một chương trình như vậy từ đầu. Đó là lý do tại sao chúng tôi sử dụng máy học.

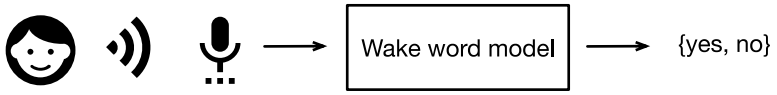


Fig. 2.1.1: Identify a wake word.

Đây là mánh khóe. Thông thường, ngay cả khi chúng ta không biết làm thế nào để nói với một máy tính một cách rõ ràng làm thế nào để ánh xạ từ đầu vào đến đầu ra, dù sao chúng ta vẫn có khả năng tự thực hiện chiến công nhận thức. Nói cách khác, ngay cả khi bạn không biết cách lập trình máy tính để nhận ra từ "Alexa", bản thân bạn có thể nhận ra nó. Được trang bị khả năng này, chúng ta có thể thu thập một bộ dữ liệu* khổng lồ* chứa các ví dụ về âm thanh và gắn nhãn những người làm và không chứa từ thức dậy. Trong phương pháp học máy, chúng tôi không cố gắng thiết kế một hệ thống *rõ ràng* để nhận ra các từ thức. Thay vào đó, chúng tôi xác định một chương trình linh hoạt có hành vi được xác định bởi một số tham số*. Sau đó, chúng tôi sử dụng tập dữ liệu để xác định bộ tham số tốt nhất có thể, những thông số cải thiện hiệu suất của chương trình của chúng tôi đối với một số biện pháp hiệu suất đối với nhiệm vụ quan tâm.

Bạn có thể nghĩ về các tham số như các nút bấm mà chúng ta có thể xoay, thao tác hành vi của chương trình. Sửa các tham số, chúng tôi gọi chương trình là mô hình **. Tập hợp tất cả các chương trình riêng biệt (ánh xạ đầu vào-đầu ra) mà chúng ta có thể tạo ra chỉ bằng cách thao tác các tham số được gọi là *family* của mô hình.

Và meta-chương trình sử dụng tập dữ liệu của chúng tôi để chọn các tham số được gọi là thuật toán *learning *.

Trước khi chúng ta có thể tiếp tục và tham gia vào thuật toán học tập, chúng ta phải xác định chính xác vấn đề, ghim bản chất chính xác của các đầu vào và đầu ra, và chọn một gia đình mô hình thích hợp. Trong trường hợp này, mô hình của chúng tôi nhận được một đoạn âm thanh dưới dạng * đầu vào* và mô hình tạo ra một lựa chọn trong số $\{\text{yes}, \text{no}\}$ dưới dạng * đầu ra*. Nếu tất cả diễn ra theo kế hoạch, các dự đoán của mô hình thường sẽ chính xác về việc đoạn mã có chứa từ đánh thức hay không.

Nếu chúng ta chọn đúng dòng mô hình, sẽ tồn tại một cài đặt của các nút bấm sao cho mô hình sẽ cháy "có" mỗi khi nó nghe từ "Alexa". Bởi vì sự lựa chọn chính xác của từ thức là tùy ý, chúng ta có thể sẽ cần một gia đình người mẫu đủ phong phú rằng, thông qua một thiết lập khác của các nút bấm, nó có thể bắn "có" chỉ khi nghe từ "Apricot". Chúng tôi hy vọng rằng cùng một gia đình mô hình nên phù hợp để công nhận "Alexa" và công nhận "Apricot" bởi vì chúng có vẻ, trực giác, là những nhiệm vụ tương tự. Tuy nhiên, chúng ta có thể cần một dòng mô hình khác hoàn toàn nếu chúng ta muốn đối phó với các đầu vào hoặc đầu ra khác nhau về cơ bản, hãy nói nếu chúng ta muốn ánh xạ từ hình ảnh sang chú thích, hoặc từ câu tiếng Anh sang câu Trung Quốc.

Như bạn có thể đoán, nếu chúng ta chỉ cần đặt tất cả các nút một cách ngẫu nhiên, không chắc mô hình của chúng tôi sẽ nhận ra "Alexa", "Apricot" hoặc bất kỳ từ tiếng Anh nào khác. Trong machine learning, * learning* là quá trình mà chúng tôi khám phá ra cài đặt phù hợp của các nút nhấn ép buộc hành vi mong muốn từ mô hình của chúng tôi. Nói cách khác, chúng tôi * đào tạo* mô hình của chúng tôi với dữ liệu. Như thể hiện trong Fig. 2.1.2, quá trình đào tạo thường trông như sau:

1. Bắt đầu với một mô hình được khởi tạo ngẫu nhiên mà không thể làm bất cứ điều gì hữu ích.
2. Lấy một số dữ liệu của bạn (ví dụ: đoạn âm thanh và nhãn $\{\text{yes}, \text{no}\}$ tương ứng).
3. Tinh chỉnh các núm để mô hình hút ít hơn so với những ví dụ đó.
4. Lặp lại Bước 2 và 3 cho đến khi mô hình tuyệt vời.

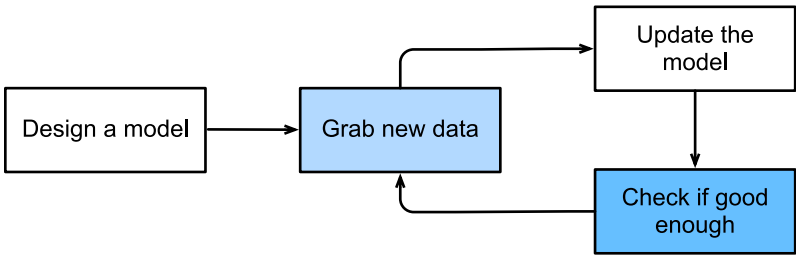


Fig. 2.1.2: A typical training process.

Để tóm tắt, thay vì mã hóa một công cụ nhận dạng từ thức, chúng tôi mã hóa một chương trình có thể * learn* để nhận ra các từ thức, nếu chúng tôi trình bày nó với một tập dữ liệu được dán nhãn lớn. Bạn có thể nghĩ đến hành động này để xác định hành vi của một chương trình bằng cách trình bày nó với một tập dữ liệu dưới dạng *lập trình với dữ liệu*. Điều đó có nghĩa là, chúng ta có thể "lập trình" một máy dò mèo bằng cách cung cấp hệ thống học máy của chúng tôi nhiều ví dụ về mèo và chó. Bằng cách này, máy dò cuối cùng sẽ học cách phát ra một số dương rất lớn nếu nó là một con mèo, một số âm rất lớn nếu nó là chó, và một cái gì đó gần bằng không nếu nó không chắc chắn, và điều này hầu như không làm trầy xước bề mặt của những gì máy học có thể làm. Deep learning, mà chúng tôi sẽ giải thích chi tiết hơn sau này, chỉ là một trong số nhiều phương pháp phổ biến để giải quyết các vấn đề máy học.

2.2 Các thành phần chính

Trong ví dụ từ thức dậy của chúng tôi, chúng tôi đã mô tả một tập dữ liệu bao gồm đoạn âm thanh và nhãn nhị phân, và chúng tôi đã đưa ra một cảm giác lượn sóng tay về cách chúng tôi có thể đào tạo một mô hình để xấp xỉ một ánh xạ từ đoạn trích đến phân loại. Loại vấn đề này, nơi chúng tôi cố gắng dự đoán một nhãn không xác định được chỉ định dựa trên các đầu vào đã biết được đưa ra một tập dữ liệu bao gồm các ví dụ mà các nhãn được biết đến, được gọi là * học được giám sách*. Đây chỉ là một trong số nhiều loại vấn đề máy học. Sau đó chúng ta sẽ đi sâu vào các vấn đề học máy khác nhau. Đầu tiên, chúng tôi muốn làm sáng tỏ nhiều hơn về một số thành phần cốt lõi sẽ theo dõi chúng tôi xung quanh, bất kể chúng tôi gặp phải vấn đề máy học nào:

1. *dữ liệu* mà chúng ta có thể học hỏi từ.
2. Một* mô hình* về cách chuyển đổi dữ liệu.
3. Chức năng *mục tiêu * định lượng mô hình đang hoạt động tốt như thế nào (hoặc xấu).
4. *algorithm * để điều chỉnh các tham số của mô hình để tối ưu hóa hàm mục tiêu.

2.2.1 Dữ liệu

Nó có thể đi mà không nói rằng bạn không thể làm khoa học dữ liệu mà không có dữ liệu. Chúng ta có thể mất hàng trăm trang suy ngẫm những gì chính xác cấu thành dữ liệu, nhưng bây giờ, chúng ta sẽ sai về mặt thực tế và tập trung vào các thuộc tính chính cần quan tâm. Nói chung, chúng tôi quan tâm đến một bộ sưu tập các ví dụ. Để làm việc với dữ liệu một cách hữu ích, chúng ta thường cần phải đưa ra một đại diện số phù hợp. Mỗi *ví dụ* (hoặc *data point*, *data instance*, *sample*) thường bao gồm một tập hợp các thuộc tính gọi là *features* (hoặc *covariates*), từ đó mô hình phải đưa ra dự đoán của nó. Trong các vấn đề học tập được giám sát ở trên, điều cần dự đoán là một thuộc tính đặc biệt được chỉ định là *label* (hoặc *target*).

Nếu chúng ta đang làm việc với dữ liệu hình ảnh, mỗi bức ảnh riêng lẻ có thể tạo thành một ví dụ, mỗi bức ảnh được biểu thị bằng một danh sách có thứ tự các giá trị số tương ứng với độ sáng của mỗi pixel. Một bức ảnh màu 200×200 sẽ bao gồm $200 \times 200 \times 3 = 120000$ giá trị số, tương ứng với độ sáng của các kênh màu đỏ, xanh lá cây và xanh cho mỗi vị trí không gian. Trong một nhiệm vụ truyền thống khác, chúng ta có thể cố gắng dự đoán liệu bệnh nhân có tồn tại hay không, với một tập hợp các tính năng tiêu chuẩn như tuổi tác, dấu hiệu quan trọng và chẩn đoán.

Khi mỗi ví dụ được đặc trưng bởi cùng một số giá trị số, chúng ta nói rằng dữ liệu bao gồm các vectơ có độ dài cố định và chúng ta mô tả độ dài không đổi của các vectơ là *dimensionality* của dữ liệu. Như bạn có thể tưởng tượng, chiều dài cố định có thể là một tài sản thuận tiện. Nếu chúng ta muốn đào tạo một mô hình để nhận ra ung thư trong hình ảnh kính hiển vi, các đầu vào có độ dài cố định có nghĩa là chúng ta có ít điều cần lo lắng hơn.

Tuy nhiên, không phải tất cả dữ liệu đều có thể dễ dàng được biểu diễn dưới dạng *chiều dài cố định* vectơ. Mặc dù chúng ta có thể mong đợi hình ảnh kính hiển vi đến từ thiết bị tiêu chuẩn, nhưng chúng ta không thể mong đợi hình ảnh được khai thác từ Internet để tất cả hiển thị với cùng độ phân giải hoặc hình dạng. Đối với hình ảnh, chúng ta có thể xem xét cắt tất cả chúng đến một kích thước tiêu chuẩn, nhưng chiến lược đó chỉ giúp chúng ta cho đến nay. Chúng tôi có nguy cơ mất thông tin trong các phần cắt ra. Hơn nữa, dữ liệu văn bản chống lại các biểu diễn độ dài cố định thậm chí còn bướng bỉnh hơn. Xem xét các đánh giá của khách hàng còn lại trên các trang thương mại điện tử như Amazon, IMDB và TripAdvisor. Một số là ngắn: "nó bốc mùi!". Những người khác ramble cho các trang. Một lợi thế lớn của học sâu so với các phương pháp truyền thống là ân sủng so sánh mà các mô hình hiện đại có thể xử lý dữ liệu * chiều dài thay đổi*.

Nói chung, chúng ta càng có nhiều dữ liệu, công việc của chúng ta càng trở nên dễ dàng hơn. Khi chúng ta có nhiều dữ liệu hơn, chúng ta có thể đào tạo các mô hình mạnh mẽ hơn và ít dựa vào các giả định được hình thành trước. Sự thay đổi chế độ từ (tương đối) nhỏ sang dữ liệu lớn là một đóng góp chính cho sự thành công

của học sâu hiện đại. Để thúc đẩy điểm về nhà, nhiều mô hình thú vị nhất trong học sâu không hoạt động mà không có bộ dữ liệu lớn. Một số người khác làm việc trong chế độ dữ liệu nhỏ, nhưng không tốt hơn các cách tiếp cận truyền thống.

Cuối cùng, nó không đủ để có nhiều dữ liệu và xử lý nó một cách khéo léo. Chúng tôi cần dữ liệu *phải*. Nếu dữ liệu đầy những sai lầm hoặc nếu các tính năng được chọn không dự đoán về số lượng mục tiêu quan tâm, việc học sẽ thất bại. Tình hình bị bắt tốt bởi sáo rỗng: *rác vào, rác ra*. Hơn nữa, hiệu suất dự đoán kém không phải là hậu quả tiềm năng duy nhất. Trong các ứng dụng nhạy cảm của học máy, như chính sách dự đoán, sàng lọc tiếp tục và các mô hình rủi ro được sử dụng để cho vay, chúng ta phải đặc biệt cảnh giác với hậu quả của dữ liệu rác. Một chế độ lỗi phổ biến xảy ra trong các bộ dữ liệu nơi một số nhóm người không được đại diện trong dữ liệu đào tạo. Hãy tưởng tượng áp dụng một hệ thống nhận dạng ung thư da trong tự nhiên chưa từng thấy da đen trước đây. Thất bại cũng có thể xảy ra khi dữ liệu không chỉ đơn thuần là đại diện cho một số nhóm mà phản ánh định kiến xã hội. Ví dụ: nếu các quyết định tuyển dụng trong quá khứ được sử dụng để đào tạo một mô hình dự đoán sẽ được sử dụng để sàng lọc sơ yếu lý lịch, thì các mô hình học máy có thể vô tình nắm bắt và tự động hóa các bất công lịch sử. Lưu ý rằng tất cả điều này có thể xảy ra mà không có nhà khoa học dữ liệu tích cực âm mưu, hoặc thậm chí nhận thức được.

2.2.2 Mô hình

Hầu hết các máy học liên quan đến việc chuyển đổi dữ liệu theo một số nghĩa. Chúng ta có thể muốn xây dựng một hệ thống chụp ảnh và dự đoán nụ cười. Ngoài ra, chúng ta có thể muốn ăn một tập hợp các bài đọc cảm biến và dự đoán mức độ bình thường so với các bài đọc dị thường như thế nào. Theo mô hình *, chúng tôi biểu thị máy tính toán để nhập dữ liệu của một loại và phun ra các dự đoán của một loại có thể khác. Đặc biệt, chúng tôi quan tâm đến các mô hình thống kê có thể được ước tính từ dữ liệu. Trong khi các mô hình đơn giản hoàn toàn có khả năng giải quyết các vấn đề đơn giản thích hợp, các vấn đề mà chúng tôi tập trung vào trong cuốn sách này kéo dài giới hạn của các phương pháp cổ điển. Học sâu được phân biệt với cách tiếp cận cổ điển chủ yếu bởi tập hợp các mô hình mạnh mẽ mà nó tập trung vào. Các mô hình này bao gồm nhiều biến đổi liên tiếp của dữ liệu được liên kết với nhau từ trên xuống dưới, do đó tên * deep learning*. Trên đường thảo luận về các mô hình sâu sắc, chúng tôi cũng sẽ thảo luận về một số phương pháp truyền thống hơn.

2.2.3 Hàm mục tiêu

Trước đó, chúng tôi giới thiệu machine learning là học hỏi từ kinh nghiệm. Bằng cách * learning* ở đây, chúng tôi có nghĩa là cải thiện tại một số nhiệm vụ theo thời gian. Nhưng ai là để nói những gì tạo thành một sự cải thiện? Bạn có thể tưởng tượng rằng chúng tôi có thể đề xuất cập nhật mô hình của chúng tôi và một số người có thể không đồng ý về việc bản cập nhật được đề xuất cấu thành một cải tiến hay suy giảm.

Để phát triển một hệ thống toán học chính thức của máy học, chúng ta cần phải có các biện pháp chính thức về mức độ tốt (hoặc xấu) mô hình của chúng ta. Trong học máy và tối ưu hóa nói chung hơn, chúng tôi gọi các chức năng * mục tiêu này*. Theo quy ước, chúng ta thường xác định các hàm khách quan để thấp hơn là tốt hơn. Đây chỉ đơn thuần là một quy ước. Bạn có thể thực hiện bất kỳ chức năng nào cao hơn tốt hơn và biến nó thành một chức năng mới giống hệt nhau về chất lượng nhưng thấp hơn là tốt hơn bằng cách lật dấu hiệu. Bởi vì thấp hơn là tốt hơn, các chức năng này đôi khi được gọi *chức năng mất*.

Khi cố gắng dự đoán các giá trị số, hàm mất mát phổ biến nhất là *squared error*, tức là bình phương của sự khác biệt giữa dự đoán và sự thật mặt đất. Để phân loại, mục tiêu phổ biến nhất là giảm thiểu tỷ lệ lỗi, tức là phần nhỏ các ví dụ mà các dự đoán của chúng ta không đồng ý với sự thật nền tảng. Một số mục tiêu (ví dụ, lỗi bình phương) rất dễ tối ưu hóa. Những người khác (ví dụ, tỷ lệ lỗi) rất khó để tối ưu hóa trực tiếp, do không phân biệt hoặc các biến chứng khác. Trong những trường hợp này, thông thường tối ưu hóa một * thay thế khách thể*.

Thông thường, hàm mất được xác định đối với các tham số của mô hình và phụ thuộc vào tập dữ liệu. Chúng tôi tìm hiểu các giá trị tốt nhất của các thông số mô hình của chúng tôi bằng cách giảm thiểu tổn thất phát sinh trên một bộ bao gồm một số ví dụ được thu thập để đào tạo. Tuy nhiên, làm tốt trên dữ liệu đào tạo không đảm bảo rằng chúng tôi sẽ làm tốt trên dữ liệu không nhìn thấy. Vì vậy, chúng tôi thường sẽ muốn chia dữ liệu có sẵn thành hai phân vùng: *training dataset* (hoặc *training set*, cho các tham số mô hình phù hợp) và * test dataset* (hoặc * test set*, được tổ chức ra để đánh giá), báo cáo cách mô hình thực hiện trên cả hai chúng. Bạn có thể nghĩ về hiệu suất đào tạo như là giống như một sinh viên 's điểm số trên các kỳ thi thực hành được sử dụng để chuẩn bị cho một số kỳ thi cuối cùng thực tế. Ngay cả khi kết quả là đáng khích lệ, điều đó không đảm bảo thành công trong kỳ thi cuối cùng. Nói cách khác, hiệu suất thử nghiệm có thể đi chệch đáng kể so với hiệu suất đào tạo. Khi một mô hình hoạt động tốt trên bộ đào tạo nhưng không khái quát hóa với dữ liệu không nhìn thấy, chúng tôi nói rằng đó là * overfitting*. Trong điều kiện thực tế, điều này giống như flunking kỳ thi thực sự mặc dù làm tốt trên các kỳ thi thực hành.

2.2.4 Thuật toán tối ưu hóa

Khi chúng ta đã có một số nguồn dữ liệu và đại diện, một mô hình và một hàm mục tiêu được xác định rõ, chúng ta cần một thuật toán có khả năng tìm kiếm các tham số tốt nhất có thể để giảm thiểu hàm mất mát. Các thuật toán tối ưu hóa phổ biến cho học sâu dựa trên cách tiếp cận gọi là * gradient descent*. Nói tóm lại, ở mỗi bước, phương pháp này kiểm tra để xem, cho mỗi tham số, theo cách mà bộ đào tạo mất sẽ di chuyển nếu bạn làm phiền tham số đó chỉ là một lượng nhỏ. Sau đó, nó cập nhật tham số theo hướng có thể làm giảm tổn thất.

2.3 Các loại vấn đề về máy học

Vấn đề từ thức trong ví dụ thúc đẩy của chúng tôi chỉ là một trong số nhiều vấn đề mà machine learning có thể giải quyết. Để thúc đẩy người đọc hơn nữa và cung cấp cho chúng tôi một số ngôn ngữ chung khi chúng tôi nói về nhiều vấn đề hơn trong suốt cuốn sách, trong phần sau, chúng tôi liệt kê một mẫu các vấn đề về máy học. Chúng tôi sẽ liên tục tham khảo các khái niệm đã nói ở trên của chúng tôi như dữ liệu, mô hình và kỹ thuật đào tạo.

2.3.1 Học được giám sát

Việc học được giám sát giải quyết nhiệm vụ dự đoán các nhãn được đưa ra các tính năng đầu vào. Mỗi tính năng—cặp nhãn được gọi là một ví dụ. Đôi khi, khi ngữ cảnh rõ ràng, chúng ta có thể sử dụng thuật ngữ *examples* để chỉ một tập hợp các đầu vào, ngay cả khi các nhãn tương ứng không được biết. Mục tiêu của chúng tôi là tạo ra một mô hình ánh xạ bất kỳ đầu vào nào đến dự đoán nhãn.

Để đưa ra mô tả này trong một ví dụ cụ thể, nếu chúng ta đang làm việc trong chăm sóc sức khỏe, thì chúng ta có thể muốn dự đoán liệu bệnh nhân có bị đau tim hay không. Quan sát này, "đau tim" hoặc "không đau tim", sẽ là nhãn hiệu của chúng tôi. Các tính năng đầu vào có thể là dấu hiệu quan trọng như nhịp tim, huyết áp tâm trương và huyết áp tâm thu.

Việc giám sát phát huy tác dụng vì để chọn các tham số, chúng tôi (các giám sát viên) cung cấp cho mô hình một tập dữ liệu bao gồm các ví dụ được dán nhãn, trong đó mỗi ví dụ được khớp với nhãn chân lý mặt đất. Trong thuật ngữ xác suất, chúng tôi thường quan tâm đến việc ước tính xác suất có điều kiện của một nhãn cho các tính năng đầu vào. Mặc dù nó chỉ là một trong số một số mô hình trong machine learning, nhưng việc học được giám sát chiếm phần lớn các ứng dụng thành công của machine learning trong ngành. Một phần, đó là do nhiều nhiệm vụ quan trọng có thể được mô tả rõ ràng là ước tính xác suất của một cái gì đó chưa biết được đưa ra một tập hợp dữ liệu có sẵn cụ thể:

- Dự đoán ung thư so với ung thư, được đưa ra một hình ảnh chụp cắt lớp vi tính.
- Dự đoán bản dịch chính xác bằng tiếng Pháp, được đưa ra một câu bằng tiếng Anh.
- Dự đoán giá cổ phiếu vào tháng tới dựa trên dữ liệu báo cáo tài chính của tháng này.

Ngay cả với mô tả đơn giản "dự đoán nhãn cho các tính năng đầu vào" học tập giám sát có thể có rất nhiều hình thức và đòi hỏi rất nhiều quyết định mô hình hóa, tùy thuộc vào (trong số các cân nhắc khác) loại, kích thước và số lượng đầu vào và đầu ra. Ví dụ, chúng ta sử dụng các mô hình khác nhau để xử lý các chuỗi có độ dài tùy ý và để xử lý các biểu diễn vectơ có độ dài cố định. Chúng tôi sẽ đến thăm nhiều vấn đề này một cách sâu sắc trong suốt cuốn sách này.

Không chính thức, quá trình học tập trông giống như sau. Đầu tiên, lấy một bộ sưu tập lớn các ví dụ mà các tính năng được biết đến và chọn từ chúng một tập hợp con ngẫu nhiên, có được các nhãn chân lý nền cho mỗi. Đôi khi các nhãn này có thể là dữ liệu có sẵn đã được thu thập (ví dụ: bệnh nhân có chết trong năm tiếp theo không?) và những lần khác, chúng ta có thể cần sử dụng các chú thích của con người để gắn nhãn dữ liệu, (ví dụ: gán hình ảnh cho các danh mục). Cùng nhau, các đầu vào và nhãn tương ứng này bao gồm bộ đào tạo. Chúng tôi cung cấp tập dữ liệu đào tạo vào một thuật toán học tập được giám sát, một chức năng lấy làm đầu vào một tập dữ liệu và xuất ra một chức năng khác: mô hình đã học. Cuối cùng, chúng ta có thể cung cấp các đầu vào chưa nhìn thấy trước đây cho mô hình đã học, sử dụng đầu ra của nó làm dự đoán của nhãn tương ứng. Quá trình đầy đủ được rút ra trong Fig. 2.3.1.

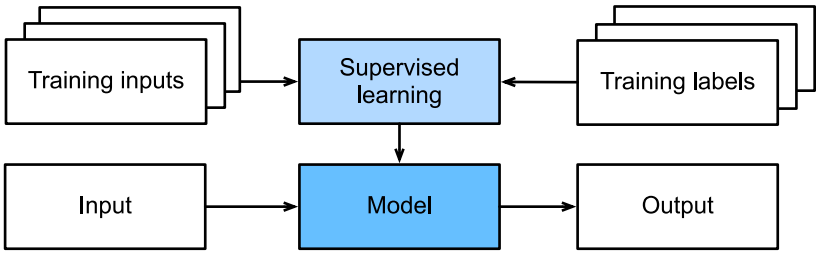


Fig. 2.3.1: Supervised learning.

Hồi quy

Có lẽ nhiệm vụ học tập được giám sát đơn giản nhất để quấn đầu của bạn là * hồi quy*. Hãy xem xét, ví dụ, một tập hợp dữ liệu thu hoạch từ cơ sở dữ liệu về doanh số bán nhà. Chúng ta có thể xây dựng một bảng, trong đó mỗi hàng tương ứng với một ngôi nhà khác nhau và mỗi cột tương ứng với một số thuộc tính có liên quan, chẳng hạn như cảnh vuông của một ngôi nhà, số phòng ngủ, số lượng phòng tắm, và số phút (đi bộ) đến trung tâm thị trấn. Trong tập dữ liệu này, mỗi ví dụ sẽ là một ngôi nhà cụ thể và vector tính năng tương ứng sẽ là một hàng trong bảng. Nếu bạn sống ở New York hoặc San Francisco, và bạn không phải là Giám đốc điều hành của Amazon, Google, Microsoft hoặc Facebook, (cảnh quay vuông, số phòng ngủ, số phòng tắm, khoảng cách đi bộ) có vector cho ngôi nhà của bạn có thể trông giống như: $[600, 1, 1, 60]$. Tuy nhiên, nếu bạn sống ở Pittsburgh, nó có thể trông giống như $[3000, 4, 3, 10]$ hơn. Các vectơ tính năng như thế này rất cần thiết cho hầu hết các thuật toán machine learning cổ điển.

Điều gì làm cho một vấn đề trở thành hồi quy thực sự là đầu ra. Nói rằng bạn đang ở trong thị trường cho một ngôi nhà mới. Bạn có thể muốn ước tính giá trị thị trường hợp lý của một ngôi nhà, với một số tính năng như trên. Nhãn, giá bán, là một giá trị số. Khi nhãn lấy các giá trị số tùy ý, chúng tôi gọi đây là vấn đề * regression*. Mục tiêu của chúng tôi là tạo ra một mô hình có dự đoán gần đúng các giá trị nhãn thực tế.

Rất nhiều vấn đề thực tế là các vấn đề hồi quy được mô tả tốt. Dự đoán đánh giá mà người dùng sẽ gán cho một bộ phim có thể được coi là một vấn đề hồi quy và nếu bạn thiết kế một thuật toán tuyệt vời để hoàn thành

kỳ công này trong 2009, bạn có thể đã giành được 1-million-dollar Netflix prize[13]. Dự đoán thời gian lưu trú cho bệnh nhân trong bệnh viện cũng là một vấn đề hồi quy. Một nguyên tắc tốt của ngón tay cái là bất kỳ * bao nhiêu? * hoặc *bao nhiêu? * vấn đề nên đề nghị hồi quy, chẳng hạn như:

- Phẫu thuật này sẽ mất bao nhiêu giờ?
- Thị trấn này sẽ có bao nhiêu lượng mưa trong sáu giờ tới?

Ngay cả khi bạn chưa bao giờ làm việc với machine learning trước đây, bạn có thể đã làm việc thông qua một vấn đề hồi quy một cách không chính thức. Hãy tưởng tượng, ví dụ, rằng bạn đã sửa chữa cống rãnh của bạn và nhà thầu của bạn đã dành 3 giờ để loại bỏ gunk khỏi đường ống nước thải của bạn. Sau đó, ông ta gửi cho bạn một hóa đơn 350 đô la. Bây giờ hãy tưởng tượng rằng bạn của bạn đã thuê cùng một nhà thầu trong 2 giờ và anh ta đã nhận được một hóa đơn 250 đô la. Nếu ai đó hỏi bạn mong đợi bao nhiêu vào hóa đơn loại bỏ gunk-removal sắp tới của họ, bạn có thể đưa ra một số giả định hợp lý, chẳng hạn như nhiều giờ làm việc chi phí nhiều đô la hơn. Bạn cũng có thể giả định rằng có một số phí cơ bản và nhà thầu sau đó tính phí mỗi giờ. Nếu những giả định này đúng, sau đó đưa ra hai ví dụ dữ liệu này, bạn đã có thể xác định cấu trúc giá của nhà thầu: 100 đô la mỗi giờ cộng với 50 đô la để xuất hiện tại nhà bạn. Nếu bạn làm theo nhiều điều đó thì bạn đã hiểu ý tưởng cấp cao đằng sau hồi quy tuyến tính.

Trong trường hợp này, chúng tôi có thể sản xuất các thông số phù hợp chính xác với giá của nhà thầu. Đôi khi điều này là không thể, ví dụ, nếu một số phương sai nợ một vài yếu tố bên cạnh hai tính năng của bạn. Trong những trường hợp này, chúng ta sẽ cố gắng tìm hiểu các mô hình giảm thiểu khoảng cách giữa các dự đoán của chúng tôi và các giá trị quan sát được. Trong hầu hết các chương của chúng tôi, chúng tôi sẽ tập trung vào việc giảm thiểu hàm mất lỗi bình phương. Như chúng ta sẽ thấy sau, sự mất mát này tương ứng với giả định rằng dữ liệu của chúng tôi đã bị hỏng bởi tiếng ồn Gaussian.

Phân loại

Trong khi các mô hình hồi quy là tuyệt vời để giải quyết * bao nhiêu? * câu hỏi, rất nhiều vấn đề không uốn cong thoải mái với mẫu này. Ví dụ: một ngân hàng muốn thêm quét séc vào ứng dụng di động của mình. Điều này sẽ liên quan đến khách hàng chụp ảnh séc bằng máy ảnh của điện thoại thông minh của họ và ứng dụng sẽ cần phải có thể tự động hiểu văn bản nhìn thấy trong hình ảnh. Cụ thể, nó cũng sẽ cần phải hiểu văn bản viết tay để trở nên mạnh mẽ hơn, chẳng hạn như ánh xạ một ký tự viết tay với một trong những ký tự đã biết. Đây là loại * cái nào? * vấn đề được gọi là *phân loại *. Nó được xử lý bằng một tập hợp các thuật toán khác với các thuật toán được sử dụng để hồi quy mặc dù nhiều kỹ thuật sẽ tiếp tục.

Trong *classification*, chúng tôi muốn mô hình của chúng tôi xem xét các tính năng, ví dụ, các giá trị pixel trong một hình ảnh, và sau đó dự đoán loại ** (chính thức gọi là *class*), trong số một tập hợp các tùy chọn rời rạc, một ví dụ thuộc về. Đối với chữ số viết tay, chúng ta có thể có mười lớp, tương ứng với các chữ số 0 đến 9. Hình thức phân loại đơn giản nhất là khi chỉ có hai lớp, một vấn đề mà chúng ta gọi là *phân loại nhị phân *. Ví dụ, tập dữ liệu của chúng ta có thể bao gồm hình ảnh của động vật và nhãn của chúng ta có thể là các lớp $\{\text{cat}, \text{dog}\}$. Trong khi trong hồi quy, chúng tôi tìm kiếm một regressor để xuất ra một giá trị số, trong phân loại, chúng tôi tìm kiếm một phân loại, có đầu ra là gán lớp dự đoán.

Vì những lý do mà chúng ta sẽ tham gia khi cuốn sách trở nên kỹ thuật hơn, có thể khó tối ưu hóa một mô hình chỉ có thể xuất ra một nhiệm vụ phân loại cứng, ví dụ: "mèo" hoặc "chó". Trong những trường hợp này, thay vào đó, việc thể hiện mô hình của chúng ta bằng ngôn ngữ xác suất thường dễ dàng hơn nhiều. Cho các tính năng của một ví dụ, mô hình của chúng tôi gán một xác suất cho mỗi lớp có thể. Quay trở lại ví dụ phân loại động vật của chúng tôi trong đó các lớp là $\{\text{cat}, \text{dog}\}$, một phân loại có thể thấy một hình ảnh và xuất ra xác suất rằng hình ảnh là một con mèo là 0.9. Chúng ta có thể giải thích con số này bằng cách nói rằng phân loại là 90% chắc chắn rằng hình ảnh mô tả một con mèo. Độ lớn của xác suất cho lớp dự đoán truyền tải một

[13] https://en.wikipedia.org/wiki/Netflix_Prize

khái niệm về sự không chắc chắn. Đó không phải là khái niệm duy nhất về sự không chắc chắn và chúng ta sẽ thảo luận về những người khác trong các chương nâng cao hơn.

Khi chúng ta có nhiều hơn hai lớp có thể, chúng ta gọi vấn đề * phân loại đa lượng*. Các ví dụ phổ biến bao gồm nhận dạng ký tự viết tay $\{0, 1, 2, ...9, a, b, c, ...\}$. Trong khi chúng ta tấn công các bài toán hồi quy bằng cách cố gắng giảm thiểu hàm mất lỗi bình phương, hàm tổn thất phổ biến cho các bài toán phân loại được gọi là *cross-entropy*, tên của nó có thể được demystified thông qua một giới thiệu về lý thuyết thông tin trong các chương tiếp theo.

Lưu ý rằng lớp có khả năng nhất không nhất thiết phải là lớp mà bạn sẽ sử dụng cho quyết định của mình. Giả sử rằng bạn tìm thấy một loại nấm đẹp ở sân sau của bạn như thể hiện trong Fig. 2.3.2.

Fig. 2.3.2: Death cap—do not eat!

Bây giờ, giả sử rằng bạn đã xây dựng một nhà phân loại và đào tạo nó để dự đoán liệu một loại nấm có độc dựa trên một bức ảnh hay không. Giả sử đầu ra phân loại phát hiện chất độc của chúng tôi rằng xác suất Fig. 2.3.2 chứa nắp tử vong là 0,2. Nói cách khác, phân loại là 80% chắc chắn rằng nấm của chúng tôi không phải là một cái chết nắp. Tuy nhiên, bạn sẽ phải là một kẻ ngốc để ăn nó. Đó là bởi vì lợi ích nhất định của một bữa tối ngon không đáng để có nguy cơ tử vong từ nó 20%. Nói cách khác, ảnh hưởng của rủi ro không chắc chắn lớn hơn lợi ích cho đến nay. Do đó, chúng ta cần tính toán rủi ro dự kiến mà chúng ta phải chịu như chức năng mất mát, tức là, chúng ta cần nhân xác suất của kết quả với lợi ích (hoặc tác hại) liên quan đến nó. Trong trường hợp này, sự mất mát phát sinh do ăn nấm có thể là $0.2 \times \infty + 0.8 \times 0 = \infty$, trong khi mất việc loại bỏ nó là $0.2 \times 0 + 0.8 \times 1 = 0.8$. Sự thận trọng của chúng tôi đã được chứng minh: như bất kỳ nhà nghiên cứu nấm nào cũng cho chúng tôi biết, nấm vào năm Fig. 2.3.2 thực sự là một cái mũ chết.

Phân loại có thể trở nên phức tạp hơn nhiều so với chỉ phân loại nhị phân, đa phân loại hoặc thậm chí nhiều nhãn. Ví dụ, có một số biến thể của phân loại để giải quyết các hệ thống phân cấp. Hệ thống phân cấp giả định rằng có tồn tại một số mối quan hệ giữa nhiều lớp. Vì vậy, không phải tất cả các lỗi đều bằng nhau - nếu chúng ta phải sai, chúng ta muốn phân loại sai thành một lớp liên quan hơn là đến một lớp xa. Thông thường, điều này được gọi là * phân loại phân học*. Một ví dụ ban đầu là do Linnaeus[14], người đã tổ chức các loài động vật theo hệ thống phân cấp.

Trong trường hợp phân loại động vật, có thể không quá tệ khi nhầm lẫn một con chó xù (giống chó) cho schnauzer (một giống chó khác), nhưng mô hình của chúng tôi sẽ phải trả một hình phạt lớn nếu nó nhầm lẫn một con chó xù cho một con khủng long. Hệ thống phân cấp nào có liên quan có thể phụ thuộc vào cách bạn định sử dụng mô hình. Ví dụ, rắn lục lạc và rắn garter có thể gần trên cây phát sinh loài, nhưng nhầm một con rattler cho một garter có thể gây chết người.

[14] https://en.wikipedia.org/wiki/Carl_Linnaeus

Gắn thẻ

Một số bài toán phân loại phù hợp gọn gàng vào các thiết lập phân loại nhị phân hoặc đa lớp. Ví dụ, chúng ta có thể huấn luyện một phân loại nhị phân bình thường để phân biệt mèo với chó. Với trạng thái hiện tại của tầm nhìn máy tính, chúng ta có thể làm điều này một cách dễ dàng, với các công cụ off-the-shelf. Tuy nhiên, cho dù mô hình của chúng tôi chính xác đến mức nào, chúng ta có thể thấy mình gặp rắc rối khi nhà phân loại gặp phải hình ảnh của * Town Musicians of Bremen*, một câu chuyện cổ tích nổi tiếng của Đức có bốn con vật trong Fig. 2.3.3.

Fig. 2.3.3: A donkey, a dog, a cat, and a rooster.

Như bạn có thể thấy, có một con mèo trong Fig. 2.3.3, và một con gà trống, chó và một con lừa, với một số cây trong nền. Tùy thuộc vào những gì chúng ta muốn làm với mô hình của chúng ta cuối cùng, coi đây là một vấn đề phân loại nhị phân có thể không có nhiều ý nghĩa. Thay vào đó, chúng ta có thể muốn cung cấp cho người mẫu tùy chọn nói rằng hình ảnh mô tả một con mèo, chó, một con lừa, *và* một con gà trống.

Vấn đề học cách dự đoán các lớp không loại trừ lẫn nhau được gọi là *phân loại nhiều nhãn mác*. Các vấn đề tự động gắn thẻ thường được mô tả tốt nhất là các vấn đề phân loại đa nhãn. Hãy suy nghĩ về các thẻ mà mọi người có thể áp dụng cho các bài đăng trên blog kỹ thuật, ví dụ: “machine learning”, “technology”, “tiện ích”, “ngôn ngữ lập trình”, “Linux”, “điện toán đám mây”, “AWS”. Một bài viết điển hình có thể có 5—10 thẻ được áp dụng bởi vì các khái niệm này có tương quan. Bài viết về “điện toán đám mây” có khả năng đề cập đến “AWS” và các bài đăng về “machine learning” cũng có thể đối phó với “ngôn ngữ lập trình”.

Chúng ta cũng phải đối phó với loại vấn đề này khi đối phó với các tài liệu y sinh, nơi gắn thẻ chính xác các bài báo là quan trọng vì nó cho phép các nhà nghiên cứu thực hiện các đánh giá đầy đủ về tài liệu. Tại Thư viện Y học Quốc gia, một số chú thích chuyên nghiệp đi qua mỗi bài viết được lập chỉ mục trong PubMed để liên kết nó với các điều khoản có liên quan từ mesh, một bộ sưu tập khoảng 28000 thẻ. Đây là một quá trình tốn thời gian và các chú thích thường có độ trễ một năm giữa lưu trữ và gắn thẻ. Học máy có thể được sử dụng

ở đây để cung cấp các thẻ tạm thời cho đến khi mỗi bài viết có thể có một đánh giá thủ công thích hợp. Thật vậy, trong vài năm, tổ chức BioASQ có hosted competitions[15] để làm chính xác điều này.

Tìm

Đôi khi chúng ta không chỉ muốn gán từng ví dụ cho một xô hoặc cho một giá trị thực. Trong lĩnh vực truy xuất thông tin, chúng tôi muốn áp đặt một bảng xếp hạng trên một tập hợp các mục. Lấy tìm kiếm trên web cho một ví dụ. Mục tiêu ít hơn để xác định xem một trang cụ thể có liên quan đến truy vấn hay không, mà là một trong rất nhiều kết quả tìm kiếm có liên quan nhất cho một người dùng cụ thể. Chúng tôi thực sự quan tâm đến việc sắp xếp các kết quả tìm kiếm có liên quan và thuật toán học tập của chúng tôi cần tạo ra các tập con có thứ tự của các yếu tố từ một tập hợp lớn hơn. Nói cách khác, nếu chúng ta được yêu cầu tạo ra 5 chữ cái đầu tiên từ bảng chữ cái, có sự khác biệt giữa việc trả về "A B C D E" và "C A B E D". Ngay cả khi tập kết quả là như nhau, thứ tự trong bộ vấn đề.

Một giải pháp khả thi cho vấn đề này là lần đầu tiên gán cho mọi yếu tố trong tập hợp một điểm liên quan tương ứng và sau đó lấy các yếu tố được xếp hạng hàng đầu. PageRank[16], nước sốt bí mật ban đầu đằng sau công cụ tìm kiếm của Google là một ví dụ ban đầu của một hệ thống tính điểm như vậy nhưng nó đặc biệt ở chỗ nó đã làm không phụ thuộc vào truy vấn thực tế. Ở đây, họ dựa vào một bộ lọc liên quan đơn giản để xác định tập hợp các mục có liên quan và sau đó vào PageRank để đặt hàng những kết quả có chứa thuật ngữ truy vấn. Ngày nay, các công cụ tìm kiếm sử dụng máy học và mô hình hành vi để có được điểm số liên quan phụ thuộc vào truy vấn. Có toàn bộ hội nghị học thuật dành cho môn học này.

Hệ thống Recommender

Hệ thống giới thiệu là một cài đặt vấn đề khác có liên quan đến tìm kiếm và xếp hạng. Các vấn đề tương tự như mục tiêu là hiển thị một tập hợp các mục có liên quan cho người dùng. Sự khác biệt chính là sự nhấn mạnh vào *cá nhân hóa* cho người dùng cụ thể trong bối cảnh của các hệ thống giới thiệu. Ví dụ: đối với các đề xuất phim, trang kết quả cho một người hâm mộ khoa học viễn tưởng và trang kết quả cho một người sành phim hài Peter Sellers có thể khác nhau đáng kể. Các vấn đề tương tự bật lên trong các cài đặt đề xuất khác, ví dụ, cho các sản phẩm bán lẻ, âm nhạc và khuyến nghị tin tức.

Trong một số trường hợp, khách hàng cung cấp phản hồi rõ ràng cho biết họ thích một sản phẩm cụ thể như thế nào (ví dụ: xếp hạng và đánh giá sản phẩm trên Amazon, IMDb và Goodreads). Trong một số trường hợp khác, họ cung cấp phản hồi ngầm, ví dụ: bằng cách bỏ qua tiêu đề trên danh sách phát, điều này có thể cho thấy sự không hài lòng nhưng có thể chỉ ra rằng bài hát không phù hợp trong ngữ cảnh. Trong các công thức đơn giản nhất, các hệ thống này được đào tạo để ước tính một số điểm, chẳng hạn như đánh giá ước tính hoặc xác suất mua hàng, cho người dùng và một mặt hàng.

Với một mô hình như vậy, đối với bất kỳ người dùng nhất định nào, chúng ta có thể lấy tập hợp các đối tượng với điểm số lớn nhất, sau đó có thể được đề xuất cho người dùng. Hệ thống sản xuất tiên tiến hơn đáng kể và tính đến hoạt động chi tiết của người dùng và đặc điểm mục khi tính toán điểm số đó. Fig. 2.3.4 là một ví dụ về sách học sâu được Amazon đề xuất dựa trên các thuật toán cá nhân hóa được điều chỉnh để nắm bắt sở thích của một người.

[15] http://bioasq.org/
[16] https://en.wikipedia.org/wiki/PageRank

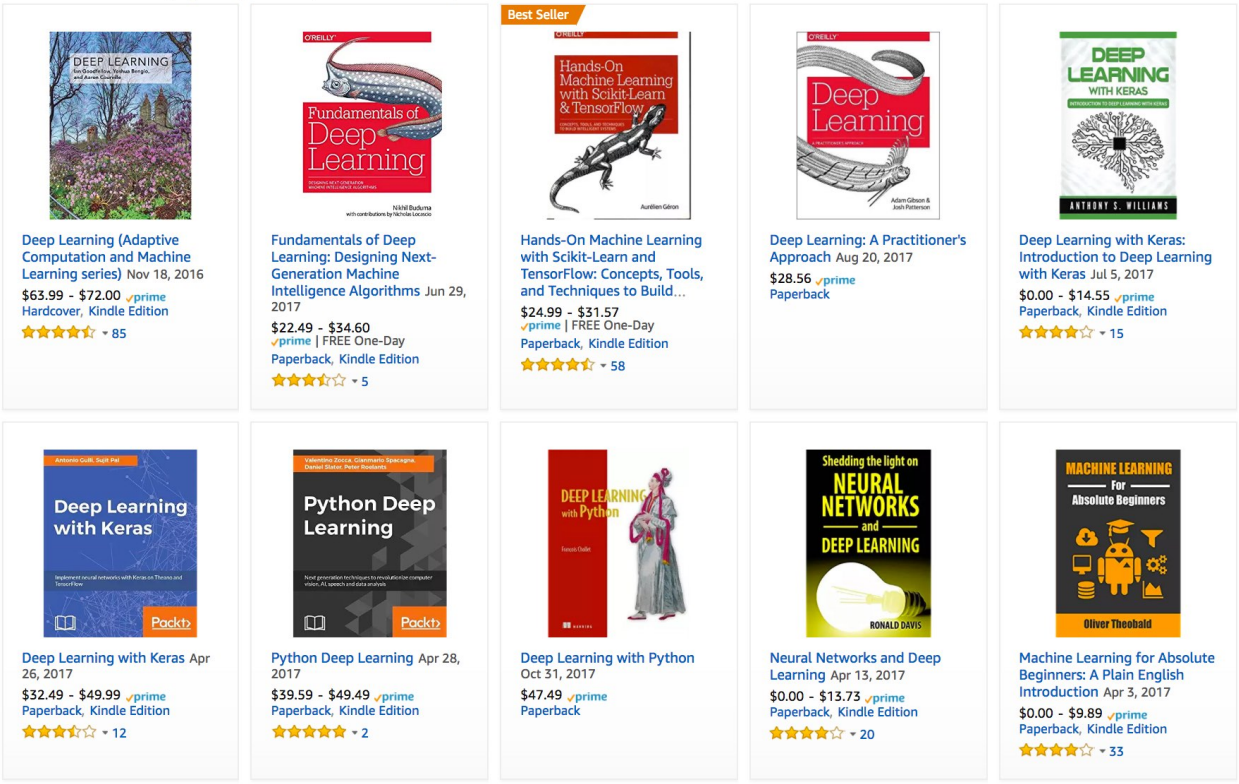


Fig. 2.3.4: Deep learning books recommended by Amazon.

Mặc dù giá trị kinh tế to lớn của họ, các hệ thống khuyến nghị ngây thơ được xây dựng trên đầu trang của các mô hình dự đoán phải chịu một số sai sót quan niệm nghiêm trọng. Để bắt đầu, chúng tôi chỉ quan sát *phản hồi kiểm duyệt *: người dùng ưu tiên đánh giá những bộ phim mà họ cảm thấy mạnh mẽ. Ví dụ: trên thang điểm năm điểm, bạn có thể nhận thấy rằng các mặt hàng nhận được nhiều xếp hạng năm và một sao nhưng có rất ít xếp hạng ba sao dễ thấy. Hơn nữa, thói quen mua hàng hiện tại thường là kết quả của thuật toán đề xuất hiện tại, nhưng các thuật toán học tập không phải lúc nào cũng tính đến chi tiết này. Do đó, các vòng phản hồi có thể hình thành nơi một hệ thống giới thiệu ưu tiên đẩy một mục sau đó được thực hiện tốt hơn (do mua hàng lớn hơn) và lần lượt được khuyến nghị thường xuyên hơn. Nhiều vấn đề trong số này về cách đối phó với kiểm duyệt, ưu đãi và vòng lặp phản hồi, là những câu hỏi nghiên cứu mở quan trọng.

Học trình tự

Cho đến nay, chúng tôi đã xem xét các vấn đề mà chúng tôi có một số cố định số đầu vào và tạo ra một số cố định của đầu ra. Ví dụ, chúng tôi đã xem xét dự đoán giá nhà từ một tập hợp các tính năng cố định: cảnh vuông, số phòng ngủ, số phòng tắm, thời gian đi bộ đến trung tâm thành phố. Chúng tôi cũng thảo luận về ánh xạ từ một hình ảnh (có kích thước cố định) đến xác suất dự đoán rằng nó thuộc về mỗi lớp cố định hoặc lấy ID người dùng và ID sản phẩm và dự đoán xếp hạng sao. Trong những trường hợp này, một khi chúng tôi cung cấp đầu vào chiều dài cố định của mình vào mô hình để tạo ra một đầu ra, mô hình ngay lập tức quên những gì nó vừa thấy.

Điều này có thể ổn nếu đầu vào của chúng ta thực sự tất cả đều có cùng kích thước và nếu các đầu vào liên tiếp thực sự không liên quan gì đến nhau. Nhưng làm thế nào chúng ta sẽ đối phó với đoạn video? Trong trường hợp này, mỗi đoạn mã có thể bao gồm một số khung khác nhau. Và đoán của chúng tôi về những gì đang xảy ra trong mỗi khung hình có thể mạnh hơn nhiều nếu chúng ta tính đến các khung trước đó hoặc thành công. Cũng vậy với ngôn ngữ. Một vấn đề học sâu phổ biến là dịch máy: nhiệm vụ nhập câu bằng một số ngôn ngữ

nguồn và dự đoán bản dịch của chúng bằng một ngôn ngữ khác.

Những vấn đề này cũng xảy ra trong y học. Chúng tôi có thể muốn một mô hình để theo dõi bệnh nhân trong phòng chăm sóc đặc biệt và bắn ra cảnh báo nếu nguy cơ tử vong của họ trong 24 giờ tới vượt quá ngưỡng. Chúng tôi chắc chắn sẽ không muốn mô hình này vứt bỏ mọi thứ mà nó biết về lịch sử bệnh nhân mỗi giờ và chỉ đưa ra dự đoán của nó dựa trên các phép đo gần đây nhất.

Những vấn đề này là một trong những ứng dụng thú vị nhất của machine learning và chúng là các trường hợp của học trình tự *. Chúng yêu cầu một mô hình để ăn các chuỗi đầu vào hoặc phát ra chuỗi đầu ra (hoặc cả hai). Cụ thể, *trình tự để học trình tự* xem xét các vấn đề trong đó đầu vào và đầu ra đều là chuỗi có độ dài thay đổi, chẳng hạn như dịch máy và phiên âm văn bản từ bài phát biểu nói. Mặc dù không thể xem xét tất cả các loại biến đổi trình tự, nhưng các trường hợp đặc biệt sau đây là đáng nói đến.

** Gắn thẻ và phân tích cú pháp**. Điều này liên quan đến việc chú thích một chuỗi văn bản với các thuộc tính. Nói cách khác, số lượng đầu vào và đầu ra về cơ bản là giống nhau. Ví dụ, chúng ta có thể muốn biết các động từ và chủ thể ở đâu. Ngoài ra, chúng ta có thể muốn biết từ nào là các thực thể được đặt tên. Nói chung, mục tiêu là phân hủy và chú thích văn bản dựa trên các giả định cấu trúc và ngữ pháp để có được một số chú thích. Điều này nghe có vẻ phức tạp hơn so với thực tế. Dưới đây là một ví dụ rất đơn giản về chú thích một câu với các thẻ cho biết từ nào đề cập đến các thực thể được đặt tên (được gắn thẻ là "Ent").

```
Tom has dinner in Washington with Sally
Ent  -    -    -     Ent      -    Ent
```

** Nhận dạng giọng nói tự động**. Với nhận dạng giọng nói, trình tự đầu vào là một bản ghi âm của một loa (hiển thị trong Fig. 2.3.5), và đầu ra là bảng điểm văn bản của những gì người nói. Thách thức là có nhiều khung âm thanh hơn (âm thanh thường được lấy mẫu ở tốc độ 8kHz hoặc 16kHz) so với văn bản, tức là, không có sự tương ứng 1:1 giữa âm thanh và văn bản, vì hàng ngàn mẫu có thể tương ứng với một từ nói duy nhất. Đây là những chuỗi để sắp xếp các vấn đề học tập trong đó đầu ra ngắn hơn nhiều so với đầu vào.

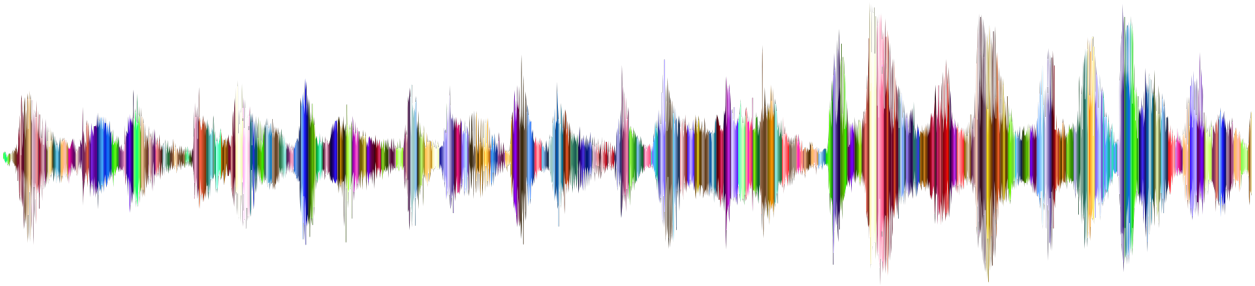


Fig. 2.3.5: `-D-e-e-p- L-ea-r-ni-ng-` in an audio recording.

Text to Speech. Đây là nghịch đảo của nhận dạng giọng nói tự động. Nói cách khác, đầu vào là văn bản và đầu ra là một tệp âm thanh. Trong trường hợp này, đầu ra dài hơn nhiều so với đầu vào. Mặc dù con người dễ dàng nhận ra một tệp âm thanh xấu, nhưng điều này không hoàn toàn tầm thường đối với máy tính.

** Dịch máy**. Không giống như trường hợp nhận dạng giọng nói, nơi tương ứng đầu vào và đầu ra xảy ra theo cùng một thứ tự (sau khi căn chỉnh), trong dịch máy, đảo ngược thứ tự có thể rất quan trọng. Nói cách khác, trong khi chúng ta vẫn đang chuyển đổi một chuỗi thành chuỗi khác, không phải số lượng đầu vào và đầu ra cũng như thứ tự của các ví dụ dữ liệu tương ứng được giả định là giống nhau. Hãy xem xét ví dụ minh họa sau đây về xu hướng đặc biệt của người Đức để đặt các động từ ở cuối câu.

```
German:           Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English:          Did you already check out this excellent tutorial?
Wrong alignment:  Did you yourself already this excellent tutorial looked-at?
```

Nhiều vấn đề liên quan bật lên trong các nhiệm vụ học tập khác. Ví dụ, xác định thứ tự mà người dùng đọc một trang web là một vấn đề phân tích bố cục hai chiều. Các vấn đề đối thoại thể hiện tất cả các loại biến chứng bổ sung, trong đó việc xác định những gì cần nói tiếp theo đòi hỏi phải tính đến kiến thức trong thế giới thực và trạng thái trước đó của cuộc trò chuyện trên khoảng cách thời gian dài. Đây là những lĩnh vực nghiên cứu tích cực.

2.3.2 Học tập không được giám sát và tự giám sát

Tất cả các ví dụ cho đến nay đều liên quan đến việc học được giám sát, tức là, các tình huống mà chúng tôi cung cấp cho mô hình một tập dữ liệu khổng lồ chứa cả các tính năng và giá trị nhãn tương ứng. Bạn có thể nghĩ về người học được giám sát là có một công việc cực kỳ chuyên biệt và một ông chủ cực kỳ tầm thường. Ông chủ đứng trên vai của bạn và cho bạn biết chính xác phải làm gì trong mọi tình huống cho đến khi bạn học cách lập bản đồ từ các tình huống đến hành động. Làm việc cho một ông chủ như vậy nghe có vẻ khá khập khiễng. Mặt khác, thật dễ dàng để làm hài lòng ông chủ này. Bạn chỉ cần nhận ra mô hình càng nhanh càng tốt và bắt chước hành động của họ.

Theo một cách hoàn toàn ngược lại, có thể là bực bội khi làm việc cho một ông chủ không biết họ muốn bạn làm gì. Tuy nhiên, nếu bạn có kế hoạch trở thành một nhà khoa học dữ liệu, bạn nên làm quen với nó. Ông chủ có thể chỉ đưa cho bạn một bãi dữ liệu khổng lồ và nói với bạn để * làm một số khoa học dữ liệu với nó! * Điều này nghe có vẻ mơ hồ vì nó là. Chúng tôi gọi lớp vấn đề này* học không được giám sách*, và loại và số câu hỏi chúng tôi có thể hỏi chỉ bị giới hạn bởi sự sáng tạo của chúng tôi. Chúng tôi sẽ giải quyết các kỹ thuật học tập không được giám sát trong các chương sau. Để kích thích sự thèm ăn của bạn bây giờ, chúng tôi mô tả một vài trong số các câu hỏi sau đây bạn có thể hỏi.

- Can we find a small nhỏ bé numbercon số of prototypenguyên mẫu that accurately chính xác summarize tóm tắt the data dữ liệu? Với một bộ ảnh, chúng ta có thể nhóm chúng thành ảnh phong cảnh, hình ảnh của chó, trẻ sơ sinh, mèo và đỉnh núi không? Tương tự như vậy, với một bộ sưu tập các hoạt động duyệt web của người dùng, chúng ta có thể nhóm chúng thành những người dùng có hành vi tương tự không? Vấn đề này thường được gọi là *cùng*.
- Chúng ta có thể tìm thấy một số lượng nhỏ các tham số mà nắm bắt chính xác các thuộc tính có liên quan của dữ liệu? Quỹ đạo của một quả bóng được mô tả khá tốt bằng vận tốc, đường kính và khối lượng của quả bóng. Thợ may đã phát triển một số lượng nhỏ các thông số mô tả hình dạng cơ thể con người khá chính xác với mục đích phù hợp với quần áo. Những vấn đề này được gọi là *ước tính không gian con*. Nếu sự phụ thuộc là tuyến tính, nó được gọi là * phân tích thành phần chính*.
- Có một đại diện của các đối tượng (cấu trúc tùy ý) trong không gian Euclide như vậy mà tính chất tượng trưng có thể được phù hợp tốt? Điều này có thể được sử dụng để mô tả các thực thể và quan hệ của họ, chẳng hạn như “Rome” $-$ “Italy” $+$ “Pháp” $=$ “Paris”.
- Có mô tả về nguyên nhân gốc rễ of much of the data dữ liệu that we observe quan sát? Ví dụ, nếu chúng ta có dữ liệu nhân khẩu học về giá nhà, ô nhiễm, tội phạm, vị trí, giáo dục và tiền lương, chúng ta có thể khám phá ra cách chúng liên quan đơn giản dựa trên dữ liệu thực nghiệm không? Các trường liên quan đến *causality* và *mô hình đồ họa xác suất* giải quyết vấn đề này.
- Một sự phát triển gần đây quan trọng và thú vị khác trong việc học không được giám sát là sự ra đời của * generative adversarial networks*. Những điều này cho chúng ta một cách thủ tục để tổng hợp dữ liệu, thậm chí dữ liệu có cấu trúc phức tạp như hình ảnh và âm thanh. Các cơ chế thống kê cơ bản là các thử nghiệm để kiểm tra xem dữ liệu thực và giả có giống nhau hay không.

As a formhình thức of unsupervised không giám sát learning học tập, *tự giám sát* tận dụng dữ liệu không được dán nhãn để cung cấp sự giám sát trong đào tạo, chẳng hạn như bằng cách dự đoán một số phần giữ lại của dữ liệu bằng cách sử dụng các phần khác. Đối với văn bản, chúng ta có thể đào tạo các mô hình để “điền vào khoảng trống” bằng cách dự đoán các từ được che giấu ngẫu nhiên bằng cách sử dụng các từ xung quanh (ngữ

cảnh) của chúng trong thể lớn mà không cần bất kỳ nỗ lực ghi nhãn (Devlin et al., 2018)! Đối với hình ảnh, chúng tôi có thể đào tạo các mô hình để cho biết vị trí tương đối giữa hai vùng bị cắt của cùng một hình ảnh [Doersch.Gupta.Efros.2015]. Trong hai ví dụ về việc học tự giám sát này, các mô hình đào tạo để dự đoán các từ có thể và vị trí tương đối đều là nhiệm vụ phân loại (từ việc học được giám sát).

2.3.3 Tương tác với môi trường

Cho đến nay, chúng ta chưa thảo luận về dữ liệu thực sự đến từ đâu, hoặc những gì thực sự xảy ra khi một mô hình máy học tạo ra một đầu ra. Đó là bởi vì việc học được giám sát và học tập không được giám sát không giải quyết những vấn đề này một cách rất tinh vi. Trong cả hai trường hợp, chúng tôi lấy trước một đống dữ liệu lớn, sau đó thiết lập các máy nhận dạng mẫu của chúng tôi chuyển động mà không bao giờ tương tác với môi trường một lần nữa. Bởi vì tất cả các quá trình học tập diễn ra sau khi thuật toán bị ngắt kết nối khỏi môi trường, điều này đôi khi được gọi là *học ngoại tuyết*. Đối với việc học được giám sát, quá trình này bằng cách xem xét thu thập dữ liệu từ một môi trường trông giống như Fig. 2.3.6.

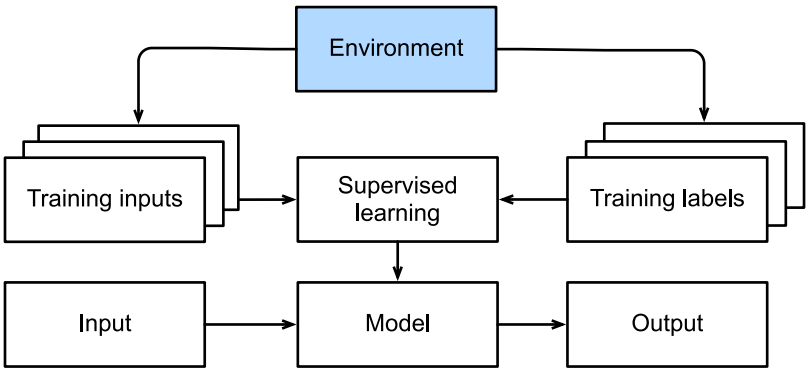


Fig. 2.3.6: Collecting data for supervised learning from an environment.

Sự đơn giản của việc học ngoại tuyến này có sự quyến rũ của nó. Nhược điểm là chúng ta có thể lo lắng về việc nhận dạng mô hình trong sự cô lập, mà không bị phân tâm từ những vấn đề khác. Nhưng nhược điểm là công thức vấn đề khá hạn chế. Nếu bạn tham vọng hơn, hoặc nếu bạn lớn lên đọc loạt Robot của Asimov, thì bạn có thể tưởng tượng các bot thông minh nhân tạo có khả năng không chỉ đưa ra dự đoán mà còn thực hiện các hành động trên thế giới. Chúng tôi muốn suy nghĩ về thông minh* đại lý*, không chỉ là các mô hình dự đoán. Điều này có nghĩa là chúng ta cần suy nghĩ về việc lựa chọn *actions*, không chỉ đưa ra dự đoán. Hơn nữa, không giống như dự đoán, các hành động thực sự ảnh hưởng đến môi trường. Nếu chúng ta muốn đào tạo một đại lý thông minh, chúng ta phải tính đến cách hành động của nó có thể ảnh hưởng đến các quan sát trong tương lai của tác nhân.

Xem xét sự tương tác với một môi trường mở ra một tập hợp các câu hỏi mô hình hóa mới. Sau đây chỉ là một vài ví dụ.

- Môi trường có nhớ những gì chúng ta đã làm trước đây không?
- Môi trường có muốn giúp chúng tôi, ví dụ: người dùng đọc văn bản vào bộ nhận dạng giọng nói không?
- Môi trường có muốn đánh bại chúng ta, tức là, một thiết lập đối thủ như lọc thư rác (chống lại kẻ gửi thư rác) hoặc chơi một trò chơi (so với đối thủ)?
- Môi trường không quan tâm?
- Môi trường có động lực dịch chuyển không? Ví dụ: dữ liệu trong tương lai luôn giống với quá khứ hay các mẫu thay đổi theo thời gian, một cách tự nhiên hoặc để đáp ứng với các công cụ tự động của chúng tôi?

Câu hỏi cuối cùng này đặt ra vấn đề của *dịch chuyển phân phố*, khi dữ liệu đào tạo và kiểm tra khác nhau. Đó là một vấn đề mà hầu hết chúng ta đã trải qua khi tham gia các kỳ thi được viết bởi một giảng viên, trong khi bài tập về nhà được sáng tác bởi các trợ lý giảng dạy của ông. Tiếp theo, chúng tôi sẽ mô tả ngắn gọn học tập củng cố, một thiết lập xem xét rõ ràng tương tác với một môi trường.

2.3.4 Học tăng cường

Nếu bạn quan tâm đến việc sử dụng máy học để phát triển một tác nhân tương tác với môi trường và thực hiện các hành động, thì có lẽ bạn sẽ tập trung vào * học tăng cường *. Điều này có thể bao gồm các ứng dụng cho robot, hệ thống đối thoại và thậm chí để phát triển trí tuệ nhân tạo (AI) cho trò chơi điện tử. *Học tăng cường sâu*, áp dụng học sâu để củng cố các vấn đề học tập, đã tăng lên phổ biến. Đột phá sâu Q-mạng đánh bại con người tại các trò chơi Atari chỉ sử dụng đầu vào trực quan, và chương trình AlphaGo đã truất ngôi nhà vô địch thế giới tại trò chơi board game Go là hai ví dụ nổi bật.

Học củng cố đưa ra một tuyên bố rất chung chung về một vấn đề, trong đó một tác nhân tương tác với một môi trường trong một loạt các bước thời gian. Tại mỗi bước thời gian, tác nhân nhận được một số *quan sách* từ môi trường và phải chọn một *hành động* sau đó được truyền trở lại môi trường thông qua một số cơ chế (đôi khi được gọi là bộ truyền động). Cuối cùng, đại lý nhận được phần thưởng từ môi trường. Quá trình này được minh họa trong Fig. 2.3.7. Các tác nhân sau đó nhận được một quan sát tiếp theo, và chọn một hành động tiếp theo, và như vậy. Hành vi của một đại lý học tập củng cố được điều chỉnh bởi một chính sách. Nói tóm lại, một *policy* chỉ là một chức năng ánh xạ từ quan sát môi trường đến hành động. Mục tiêu của việc học củng cố là tạo ra một chính sách tốt.

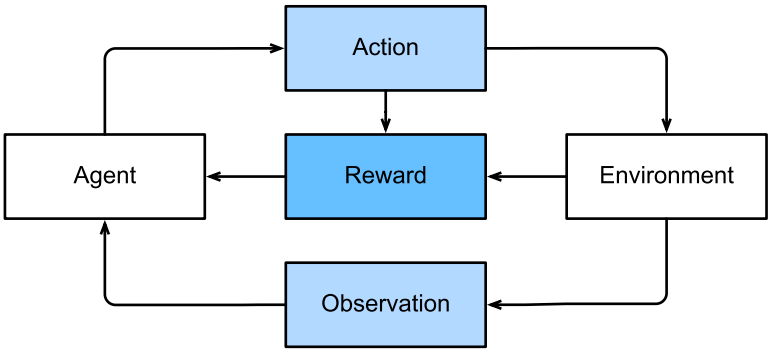


Fig. 2.3.7: The interaction between reinforcement learning and an environment.

Thật khó để nói quá mức tổng quát của khung học tập củng cố. Ví dụ, chúng ta có thể đưa bất kỳ vấn đề học tập được giám sát nào như một vấn đề học tập củng cố. Giả sử chúng tôi đã có một vấn đề phân loại. Chúng ta có thể tạo ra một đại lý học tập tăng cường với một hành động tương ứng với mỗi lớp. Sau đó chúng ta có thể tạo ra một môi trường cho một phần thưởng chính xác bằng với chức năng mất từ vấn đề học tập được giám sát ban đầu.

Điều đó đang được nói, việc học củng cố cũng có thể giải quyết nhiều vấn đề mà việc học được giám sát không thể. Ví dụ, trong việc học được giám sát, chúng tôi luôn mong đợi rằng đầu vào đào tạo được liên kết với nhãn chính xác. Nhưng trong học tập củng cố, chúng tôi không cho rằng đối với mỗi quan sát môi trường cho chúng ta biết hành động tối ưu. Nói chung, chúng tôi chỉ nhận được một số phần thưởng. Hơn nữa, môi trường thậm chí có thể không cho chúng ta biết hành động nào dẫn đến phần thưởng.

Xem xét ví dụ như các trò chơi của cờ vua. Tín hiệu phần thưởng thực sự duy nhất đến ở cuối trò chơi khi chúng ta giành chiến thắng, mà chúng ta có thể gán phần thưởng là 1, hoặc khi chúng ta thua, mà chúng ta có thể gán phần thưởng là -1. Vì vậy, người học củng cố phải giải quyết vấn đề *tín dục*: xác định hành động nào cần ghi nhận hoặc đổ lỗi cho một kết quả. Điều tương tự cũng xảy ra đối với một nhân viên được khuyến mãi

vào ngày 11 tháng 10. Chương trình khuyến mãi đó có thể phản ánh một số lượng lớn các hành động được lựa chọn tốt trong năm trước. Nhận được nhiều chương trình khuyến mãi trong tương lai đòi hỏi phải tìm ra những hành động nào trên đường đi dẫn đến chương trình khuyến mãi.

Người học củng cố cũng có thể phải đối phó với vấn đề quan sát từng phần. Đó là, quan sát hiện tại có thể không cho bạn biết mọi thứ về trạng thái hiện tại của bạn. Nói rằng một robot làm sạch thấy mình bị mắc kẹt trong một trong nhiều tủ quần áo giống hệt nhau trong một ngôi nhà. Suy ra vị trí chính xác (và do đó trạng thái) của robot có thể yêu cầu xem xét các quan sát trước đó của nó trước khi vào tủ quần áo.

Cuối cùng, tại bất kỳ thời điểm nào, người học củng cố có thể biết về một chính sách tốt, nhưng có thể có nhiều chính sách tốt hơn khác mà đại lý chưa bao giờ thử. Người học củng cố phải liên tục lựa chọn để * khai thác * chiến lược tốt nhất hiện nay được biết đến như một chính sách, hoặc để * khám phát* không gian của các chiến lược, có khả năng từ bỏ một số phần thưởng ngắn hạn để đổi lấy kiến thức.

Vấn đề học tập củng cố chung là một bối cảnh rất chung chung. Các hành động ảnh hưởng đến các quan sát tiếp theo. Phần thưởng chỉ được quan sát tương ứng với các hành động đã chọn. Môi trường có thể được quan sát đầy đủ hoặc một phần. Kế toán cho tất cả sự phức tạp này cùng một lúc có thể hỏi quá nhiều các nhà nghiên cứu. Hơn nữa, không phải mọi vấn đề thực tế thể hiện tất cả sự phức tạp này. Kết quả là, các nhà nghiên cứu đã nghiên cứu một số trường hợp đặc biệt của các vấn đề học tập củng cố.

Khi môi trường được quan sát đầy đủ, chúng tôi gọi vấn đề học tập củng cố là quy trình quyết định * Markov. *Khi trạng thái không phụ thuộc vào các hành động trước đó, chúng ta gọi vấn đề là vấn đề tên cướp theo ngữ cảnh *. *. Khi không có trạng thái, chỉ là một tập hợp các hành động có sẵn với phần thưởng ban đầu chưa biết, vấn đề này là vấn đề cướp đa vũ cổ* cổ điển*.

2.4 Rễ

Chúng tôi vừa xem xét một tập hợp nhỏ các vấn đề mà machine learning có thể giải quyết. Đối với một tập hợp các vấn đề máy học đa dạng, deep learning cung cấp các công cụ mạnh mẽ để giải quyết chúng. Mặc dù nhiều phương pháp học sâu là những phát minh gần đây, ý tưởng cốt lõi của lập trình với dữ liệu và mạng thần kinh (tên của nhiều mô hình học sâu) đã được nghiên cứu trong nhiều thế kỷ. Trên thực tế, con người đã tổ chức mong muốn phân tích dữ liệu và dự đoán kết quả trong tương lai lâu dài và phần lớn khoa học tự nhiên có nguồn gốc từ việc này. Ví dụ, bản phân bố Bernoulli được đặt theo tên [Jacob Bernoulli (1655—1705)[17], và bản phân bố Gaussian được phát hiện bởi Carl Friedrich Gauss (1777—1855)[18]. Ông đã phát minh ra, ví dụ, thuật toán ô vuông ít trung bình nhất, mà vẫn được sử dụng ngày nay cho vô số vấn đề từ tính toán bảo hiểm đến chẩn đoán y tế. Những công cụ này đã tạo ra một cách tiếp cận thực nghiệm trong khoa học tự nhiên - ví dụ, định luật Ohm liên quan đến dòng điện và điện áp trong một điện trở được mô tả hoàn hảo bởi một mô hình tuyến tính.

Ngay cả trong thời trung cổ, các nhà toán học đã có một trực giác quan tâm về ước tính. Ví dụ, cuốn sách hình học của [Jacob Köbel (1460—1533)[19] minh họa trung bình chiều dài của 16 chân nam trưởng thành để có được chiều dài chân trung bình.

[17] https://en.wikipedia.org/wiki/Jacob_Bernoulli
[18] https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss
[19] https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry

Fig. 2.4.1: Estimating the length of a foot.

Fig. 2.4.1 minh họa cách thức ước tính này hoạt động. 16 người đàn ông trưởng thành được yêu cầu xếp hàng liên tiếp, khi rời khỏi nhà thờ. Chiều dài tổng hợp của chúng sau đó được chia cho 16 để có được ước tính cho những gì bây giờ là 1 foot. "Thuật toán" này sau đó được cải tiến để đối phó với đôi chân sai mạnh—2 người đàn ông có bàn chân ngắn nhất và dài nhất lần lượt được gửi đi, trung bình chỉ hơn phần còn lại. Đây là một trong những ví dụ sớm nhất về ước tính trung bình được cắt tỉa.

Thống kê thực sự cất cánh với việc thu thập và tính sẵn có của dữ liệu. Một trong những người khổng lồ của nó, Ronald Fisher (1890—1962)[20], đóng góp đáng kể cho lý thuyết của nó và cũng là các ứng dụng của nó trong di truyền học. Nhiều thuật toán của ông (như phân tích phân biệt đối xử tuyến tính) và công thức (như ma trận thông tin Fisher) vẫn đang được sử dụng thường xuyên ngày nay. Trên thực tế, ngay cả bộ dữ liệu Iris mà Fisher phát hành vào năm 1936 vẫn còn được sử dụng đôi khi để minh họa cho các thuật toán học máy. Ông cũng là một người đề xuất ưu sinh, điều này nên nhắc nhở chúng ta rằng việc sử dụng khoa học dữ liệu đáng ngờ về mặt đạo đức đã lâu dài và lâu dài một lịch sử như việc sử dụng sản xuất của nó trong công nghiệp và khoa học tự nhiên.

Ảnh hưởng thứ hai đối với học máy đến từ lý thuyết thông tin của Claude Shannon (1916—2001)[21] và lý thuyết tính toán qua Alan Turing (1912—1954)[22]. Turing đặt ra câu hỏi "máy móc có thể nghĩ không?" trong bài báo nổi tiếng của mình * Computing Machinery and Intelligence* (Turing, 1950). Trong những gì ông mô tả là thử nghiệm Turing, một cỗ máy có thể được coi là *thông minh* nếu người đánh giá của con người khó phân biệt giữa các câu trả lời với một máy và một con người dựa trên tương tác văn bản.

Một ảnh hưởng khác có thể được tìm thấy trong khoa học thần kinh và tâm lý học. Rốt cuộc, con người thể hiện rõ hành vi thông minh. Do đó, chỉ hợp lý để hỏi liệu người ta có thể giải thích và có thể đảo ngược khả năng này. Một trong những thuật toán lâu đời nhất lấy cảm hứng từ thời trang này được xây dựng bởi Donald

[20] https://en.wikipedia.org/wiki/Ronald_Fisher
[21] https://en.wikipedia.org/wiki/Claude_Shannon
[22] https://en.wikipedia.org/wiki/Alan_Turing

Hebb (1904—1985)[23]. Trong cuốn sách đột phá của mình * Tổ chức hành vi* (Hebb & Hebb, 1949), ông khẳng định rằng các tế bào thần kinh học bằng cách củng cố tích cực. Điều này được biết đến như là quy tắc học tập Hebbian. Đây là nguyên mẫu của thuật toán học perceptron của Rosenblatt và nó đã đặt nền móng của nhiều thuật toán gốc gradient ngẫu nhiên làm nền tảng học sâu ngày nay: củng cố hành vi mong muốn và giảm bớt hành vi không mong muốn để có được cài đặt tốt của các thông số trong mạng thần kinh.

Cảm hứng sinh học là những gì đã cho * mạng thần kinh * tên của họ. Trong hơn một thế kỷ (có niên đại từ các mô hình của Alexander Bain, 1873 và James Sherrington, 1890), các nhà nghiên cứu đã cố gắng lắp ráp các mạch tính toán giống với mạng lưới các tế bào thần kinh tương tác. Theo thời gian, việc giải thích sinh học đã trở nên ít theo nghĩa đen hơn nhưng cái tên bị mắc kẹt. Tại trái tim của nó, nói dối một vài nguyên tắc chính có thể được tìm thấy trong hầu hết các mạng hiện nay:

- Sự xen kẽ của các đơn vị xử lý tuyến tính và phi tuyến, thường được gọi là * lớp*.
- Việc sử dụng quy tắc chuỗi (còn được gọi là *backpropagation*) để điều chỉnh các tham số trong toàn bộ mạng cùng một lúc.

Sau những tiến bộ nhanh chóng ban đầu, nghiên cứu trong các mạng thần kinh mệt mỏi từ khoảng năm 1995 cho đến năm 2005. Điều này chủ yếu là do hai lý do. Đầu tiên, đào tạo một mạng là tính toán rất tốn kém. Trong khi bộ nhớ truy cập ngẫu nhiên rất phong phú vào cuối thế kỷ qua, sức mạnh tính toán rất khan hiếm. Thứ hai, các bộ dữ liệu tương đối nhỏ. Trên thực tế, tập dữ liệu Iris của Fisher từ năm 1932 là một công cụ phổ biến để kiểm tra hiệu quả của các thuật toán. Tập dữ liệu MNIST với 60000 chữ số viết tay của nó được coi là rất lớn.

Với sự khan hiếm của dữ liệu và tính toán, các công cụ thống kê mạnh mẽ như phương pháp hạt nhân, cây quyết định và mô hình đồ họa tỏ ra vượt trội theo kinh nghiệm. Không giống như các mạng thần kinh, họ không cần nhiều tuần để đào tạo và cung cấp kết quả có thể dự đoán được với những đảm bảo lý thuyết mạnh mẽ.

2.5 Con đường đến sâu học

Phần lớn điều này đã thay đổi với sự sẵn có sẵn của một lượng lớn dữ liệu, do World Wide Web, sự ra đời của các công ty phục vụ hàng trăm triệu người dùng trực tuyến, phổ biến các cảm biến giá rẻ, chất lượng cao, lưu trữ dữ liệu giá rẻ (định luật Kryder) và tính toán giá rẻ (định luật Moore), đặc biệt là dạng GPU, ban đầu được thiết kế để chơi game trên máy tính. Đột nhiên các thuật toán và mô hình dường như không khả thi tính toán trở nên có liên quan (và ngược lại). Điều này được minh họa tốt nhất trong Chapter 2.5.

Decade	Dataset	Memory	Floating point calculations per second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

Table: Dataset so với bộ nhớ máy tính và sức mạnh tính toán

Rõ ràng là bộ nhớ truy cập ngẫu nhiên đã không theo kịp sự tăng trưởng của dữ liệu. Đồng thời, sự gia tăng sức mạnh tính toán đã vượt xa dữ liệu có sẵn. Điều này có nghĩa là các mô hình thống kê cần phải trở nên hiệu quả hơn bộ nhớ (điều này thường đạt được bằng cách thêm phi tuyến tính) trong khi đồng thời có thể dành

[23] https://en.wikipedia.org/wiki/Donald_O._Hebb

nhiều thời gian hơn để tối ưu hóa các thông số này, do ngân sách tính toán tăng lên. Do đó, điểm ngọt ngào trong máy học và thống kê đã chuyển từ các mô hình tuyến tính (tổng quát) và phương pháp hạt nhân sang các mạng thần kinh sâu. Đây cũng là một trong những lý do tại sao nhiều trụ cột của học sâu, chẳng hạn như nhận thức đa lớp (McCulloch & Pitts, 1943), mạng thần kinh phức tạp (LeCun et al., 1998), bộ nhớ ngắn hạn dài (Hochreiter & Schmidhuber, 1997) và Q-Learning (Watkins & Dayan, 1992), về cơ bản là "phát hiện lại" trong thập kỷ qua, sau khi đặt tương đối không hoạt động cho considerable đáng kể time.

Sự tiến bộ gần đây trong các mô hình thống kê, ứng dụng, và thuật toán đôi khi đã được so sánh với vụ nổ Cambri: một khoảnh khắc tiến bộ nhanh chóng trong quá trình tiến hóa của các loài. Thật vậy, nhà nước của nghệ thuật không chỉ là một hậu quả đơn thuần của các nguồn lực có sẵn, áp dụng cho các thuật toán hàng thập kỷ cũ. Lưu ý rằng danh sách dưới đây hầu như không làm trầy xước bề mặt của những ý tưởng đã giúp các nhà nghiên cứu đạt được tiến bộ to lớn trong thập kỷ qua.

- Các phương pháp mới để kiểm soát công suất, chẳng hạn như * dropout* (Srivastava et al., 2014), đã giúp giảm thiểu nguy cơ quá mức. Điều này đạt được bằng cách áp dụng tiếng ồn tiêm (Bishop, 1995) trên toàn mạng thần kinh, thay thế trọng lượng bằng các biến ngẫu nhiên cho mục đích đào tạo.

- Các cơ chế chú ý đã giải quyết một vấn đề thứ hai đã cản trở số liệu thống kê trong hơn một thế kỷ: làm thế nào để tăng bộ nhớ và độ phức tạp của một hệ thống mà không làm tăng số lượng các tham số có thể học được. Các nhà nghiên cứu đã tìm thấy một giải pháp thanh lịch bằng cách sử dụng những gì chỉ có thể được xem như một cấu trúc con trỏ có thể học được (Bahdanau et al., 2014). Thay vì phải nhớ toàn bộ một chuỗi văn bản, ví dụ, đối với dịch máy trong một biểu diễn chiều cố định, tất cả những gì cần được lưu trữ là một con trỏ đến trạng thái trung gian của quá trình dịch thuật. Điều này cho phép tăng độ chính xác đáng kể cho các chuỗi dài, vì mô hình không còn cần phải nhớ toàn bộ chuỗi trước khi bắt đầu tạo ra một chuỗi mới.

- Thiết kế đa giai đoạn, ví dụ, thông qua các mạng bộ nhớ (Sukhbaatar et al., 2015) và lập trình viên thần kinh (Reed & DeFreitas, 2015) cho phép các mô hình thống kê mô tả các phương pháp lặp đi lặp lại để lập luận. Những công cụ này cho phép một trạng thái nội bộ của mạng thần kinh sâu được sửa đổi nhiều lần, do đó thực hiện các bước tiếp theo trong một chuỗi lý luận, tương tự như cách một bộ xử lý có thể sửa đổi bộ nhớ cho một tính toán.

- Một phát triển quan trọng khác là phát minh ra các mạng đối thủ thế hệ (Goodfellow et al., 2014). Theo truyền thống, các phương pháp thống kê để ước tính mật độ và các mô hình tạo tập trung vào việc tìm kiếm các phân phối xác suất thích hợp và (thường là gần đúng) thuật toán lấy mẫu từ chúng. Kết quả là, các thuật toán này phần lớn bị giới hạn bởi sự thiếu linh hoạt vốn có trong các mô hình thống kê. Sự đổi mới quan trọng trong các mạng đối thủ thế hệ là thay thế bộ lấy mẫu bằng một thuật toán tùy ý với các tham số khác biệt. Chúng sau đó được điều chỉnh theo cách mà người phân biệt đối xử (hiệu quả là thử nghiệm hai mẫu) không thể phân biệt giả với dữ liệu thực. Thông qua khả năng sử dụng các thuật toán tùy ý để tạo ra dữ liệu, nó đã mở ra ước tính mật độ cho nhiều kỹ thuật khác nhau. Ví dụ về Zebras phi nước đại (Zhu et al., 2017) và của những người nổi tiếng giả mạo phải đối mặt (Karras et al., 2017) đều là lời khai cho sự tiến bộ này. Ngay cả những người vẽ tranh nghiệp dư cũng có thể tạo ra những hình ảnh thực tế dựa trên các bản phác thảo mô tả cách bố trí của một cảnh trông giống như (Park et al., 2019).

- Trong nhiều trường hợp, một GPU duy nhất là không đủ để xử lý một lượng lớn dữ liệu có sẵn để đào tạo. Trong thập kỷ qua, khả năng xây dựng các thuật toán đào tạo song song và phân tán đã được cải thiện đáng kể. Một trong những thách thức chính trong việc thiết kế các thuật toán có thể mở rộng là việc làm việc tối ưu hóa học tập sâu, gốc gradient ngẫu nhiên, dựa vào các minibatches dữ liệu tương đối nhỏ sẽ được xử lý. Đồng thời, các lô nhỏ hạn chế hiệu quả của GPU. Do đó, đào tạo trên 1024 GPU với kích thước minibatch, giả sử 32 hình ảnh mỗi lô lên tới một minibatch tổng hợp khoảng 32000 hình ảnh. Công việc gần đây, đầu tiên là Li (Li, 2017), và sau đó là (You et al., 2017) và (Jia et al., 2018) đã đẩy kích thước lên đến 64000 quan sát, giảm thời gian đào tạo cho mô hình ResNet-50 trên bộ dữ liệu ImageNet xuống chưa đầy 7 phút. Để so sánh, thời gian đào tạo ban đầu được đo theo thứ tự ngày.

- Khả năng tính toán song song cũng đã đóng góp khá quan trọng để tiến bộ trong học tập củng cố, ít nhất là bất cứ khi nào mô phỏng là một lựa chọn. Điều này đã dẫn đến những tiến bộ đáng kể trong các máy tính đạt được hiệu suất siêu phàm trong các game Go, Atari, Starcraft, và trong các mô phỏng vật lý (ví dụ, sử dụng MuJoCo). Xem ví dụ, (Silver et al., 2016) để biết mô tả về cách đạt được điều này trong AlphaGo. Tóm lại, việc học tăng cường hoạt động tốt nhất nếu có nhiều (trạng thái, hành động, phần thưởng) gấp ba lần, tức là, bất cứ khi nào có thể thử nhiều thứ để tìm hiểu cách chúng liên quan đến nhau. Mô phỏng cung cấp một đại lộ như vậy.
- Các khuôn khổ học sâu đã đóng một vai trò quan trọng trong việc phổ biến các ý tưởng. Thế hệ khung đầu tiên cho phép mô hình hóa dễ dàng bao gồm Caffe[24], Torch[25] và Theano[26]. Nhiều bài báo tinh dịch được viết bằng các công cụ này. Đến bây giờ, chúng đã được thay thế bởi TensorFlow[27] (thường được sử dụng thông qua API cấp cao Keras[28]), CNTK[29], Caffe 2[30], và Apache MXNet[31]. Thế hệ công cụ thứ ba, cụ thể là các công cụ bắt buộc để học sâu, được cho là dẫn đầu bởi Chainer[32], sử dụng một cú pháp tương tự như Python NumPy để mô tả các mô hình. Ý tưởng này đã được thông qua bởi cả PyTorch[33], Gluon API[34] của MXNet và Jax[35].

Việc phân chia lao động giữa các nhà nghiên cứu hệ thống xây dựng các công cụ tốt hơn và các nhà mô hình thống kê xây dựng mạng thần kinh tốt hơn đã đơn giản hóa mọi thứ rất nhiều. Ví dụ, đào tạo một mô hình hồi quy logistic tuyến tính từng là một vấn đề bài tập về nhà không tầm thường, xứng đáng để cung cấp cho sinh viên máy học mới tiến sĩ tại Đại học Carnegie Mellon vào năm 2014. Đến bây giờ, nhiệm vụ này có thể được thực hiện với ít hơn 10 dòng mã, đặt nó vững chắc vào nắm bắt các lập trình viên.

2.6 Câu chuyện thành công

AI có một lịch sử lâu dài của việc cung cấp kết quả sẽ rất khó để thực hiện nếu không. Ví dụ, các hệ thống phân loại thư sử dụng nhận dạng ký tự quang học đã được triển khai từ những năm 1990. Đây là, sau tất cả, nguồn gốc của tập dữ liệu MNIST nổi tiếng của các chữ số viết tay. Điều tương tự cũng áp dụng cho việc đọc séc cho tiền gửi ngân hàng và ghi điểm tín dụng của người nộp đơn. Các giao dịch tài chính được kiểm tra gian lận tự động. Điều này tạo thành xương sống của nhiều hệ thống thanh toán thương mại điện tử, chẳng hạn như PayPal, Stripe, AliPay, WeChat, Apple, Visa và MasterCard. Các chương trình máy tính cho cờ vua đã cạnh tranh trong nhiều thập kỷ. Máy học nguồn cấp dữ liệu tìm kiếm, khuyến nghị, cá nhân hóa và xếp hạng trên Internet. Nói cách khác, học máy rất phổ biến, mặc dù thường bị che giấu khỏi tầm nhìn.

Chỉ gần đây AI mới ở trong ánh đèn sân khấu, chủ yếu là do các giải pháp cho các vấn đề được coi là khó chữa trước đây và có liên quan trực tiếp đến người tiêu dùng. Nhiều tiến bộ như vậy được quy cho học sâu.

- Các trợ lý thông minh, chẳng hạn như Siri của Apple, Alexa của Amazon và trợ lý của Google, có thể trả lời các câu hỏi nói với mức độ chính xác hợp lý. Điều này bao gồm các nhiệm vụ quan trọng như bật công tắc đèn (một lợi ích cho người khuyết tật) để thực hiện các cuộc hẹn của thợ cắt tóc và cung cấp hộp thoại hỗ trợ điện thoại. Đây có thể là dấu hiệu đáng chú ý nhất cho thấy AI đang ảnh hưởng đến cuộc sống của chúng ta.

[24] https://github.com/BVLC/caffe
[25] https://github.com/torch
[26] https://github.com/Theano/Theano
[27] https://github.com/tensorflow/tensorflow
[28] https://github.com/keras-team/keras
[29] https://github.com/Microsoft/CNTK
[30] https://github.com/caffe2/caffe2
[31] https://github.com/apache/incubator-mxnet
[32] https://github.com/chainer/chainer
[33] https://github.com/pytorch/pytorch
[34] https://github.com/apache/incubator-mxnet
[35] https://github.com/google/jax

- Một thành phần quan trọng trong trợ lý kỹ thuật số là khả năng nhận dạng giọng nói chính xác. Dần dần độ chính xác của các hệ thống như vậy đã tăng lên đến mức chúng đạt đến độ chẵn lẻ của con người đối với một số ứng dụng nhất định (Xiong et al., 2018).
- Nhận dạng đối tượng tương tự như vậy đã đi một chặng đường dài. Ước tính đối tượng trong một bức tranh là một nhiệm vụ khá khó khăn trong năm 2010. Trên các nhà nghiên cứu điểm chuẩn ImageNet từ NEC Labs và Đại học Illinois tại Urbana-Champaign đã đạt được tỷ lệ lỗi top-5 là 28% (Lin et al., 2010). Đến năm 2017, tỷ lệ lỗi này đã giảm xuống còn 2,25% (Hu et al., 2018). Tương tự như vậy, kết quả tuyệt đẹp đã đạt được để xác định các loài chim hoặc chẩn đoán ung thư da.
- Trò chơi từng là một pháo đài của trí thông minh của con người. Bắt đầu từ TD-Gammon, một chương trình chơi backgammon bằng cách sử dụng học tăng cường chênh lệch thời gian, tiến trình thuật toán và tính toán đã dẫn đến các thuật toán cho một loạt các ứng dụng. Không giống như backgammon, cờ vua có một không gian trạng thái phức tạp hơn nhiều và tập hợp các hành động. DeepBlue đánh bại Garry Kasparov sử dụng song song lớn, phần cứng chuyên dụng và tìm kiếm hiệu quả thông qua cây trò chơi (Campbell et al., 2002). Đi vẫn còn khó khăn hơn, do không gian trạng thái khổng lồ của nó. AlphaGo đạt đến độ chẵn lẻ của con người vào năm 2015, sử dụng học sâu kết hợp với lấy mẫu cây Monte Carlo (Silver et al., 2016). Thách thức trong Poker là không gian nhà nước lớn và nó không được quan sát đầy đủ (chúng tôi không biết thẻ của đối thủ). Libratus vượt quá hiệu suất của con người trong Poker sử dụng chiến lược có cấu trúc hiệu quả (Brown & Sandholm, 2017). Điều này minh họa sự tiến bộ ấn tượng trong các trò chơi và thực tế là các thuật toán tiên tiến đóng một phần quan trọng trong đó.
- Một dấu hiệu khác của sự tiến bộ trong AI là sự ra đời của ô tô và xe tải tự lái. Mặc dù quyền tự chủ hoàn toàn chưa hoàn toàn nằm trong tầm tay, nhưng tiến bộ tuyệt vời đã được thực hiện theo hướng này, với các công ty như Tesla, NVIDIA và các sản phẩm vận chuyển Waymo cho phép ít nhất một phần tự chủ. Điều làm cho quyền tự chủ đầy đủ trở nên khó khăn là lái xe thích hợp đòi hỏi khả năng nhận thức, lý luận và kết hợp các quy tắc vào một hệ thống. Hiện nay, học sâu được sử dụng chủ yếu trong khía cạnh thị giác máy tính của những vấn đề này. Phần còn lại được điều chỉnh rất nhiều bởi các kỹ sư.

Một lần nữa, danh sách trên hầu như không làm trầy xước bề mặt của nơi máy học đã ảnh hưởng đến các ứng dụng thực tế. Ví dụ, robot, hậu cần, sinh học tính toán, vật lý hạt và thiên văn học nợ một số tiến bộ ấn tượng nhất gần đây của họ ít nhất là trong các phần của học máy. Do đó, học máy đang trở thành một công cụ phổ biến cho các kỹ sư và nhà khoa học.

Thông thường, câu hỏi về ngày tận thế AI, hoặc điểm kỳ dị AI đã được nêu ra trong các bài viết phi kỹ thuật về AI. Nỗi sợ hãi là bằng cách nào đó hệ thống học máy sẽ trở nên tình cảm và quyết định độc lập với các lập trình viên (và thạc sĩ) của họ về những thứ ảnh hưởng trực tiếp đến sinh kế của con người. Ở một mức độ nào đó, AI đã ảnh hưởng đến sinh kế của con người một cách ngay lập tức: tính tín dụng được đánh giá tự động, các phi công tự động chủ yếu điều hướng phương tiện, quyết định về việc có cấp bảo lãnh sử dụng dữ liệu thống kê làm đầu vào hay không. Phù phiếm hơn, chúng ta có thể yêu cầu Alexa bật máy pha cà phê.

May mắn thay, chúng ta còn xa một hệ thống AI có cảm giác sẵn sàng thao túng những người tạo ra con người (hoặc đốt cà phê của họ). Đầu tiên, các hệ thống AI được thiết kế, đào tạo và triển khai theo một cách cụ thể, hướng đến mục tiêu. Mặc dù hành vi của họ có thể tạo ra ảo giác về trí thông minh chung, nhưng đó là sự kết hợp của các quy tắc, heuristics và các mô hình thống kê làm nền tảng cho thiết kế. Thứ hai, hiện tại các công cụ cho * trí tuệ tổng quát nhân tạo* đơn giản là không tồn tại có thể cải thiện bản thân, lý do về bản thân và có thể sửa đổi, mở rộng và cải thiện kiến trúc của riêng họ trong khi cố gắng giải quyết các nhiệm vụ chung.

Một mối quan tâm cấp bách hơn nhiều là cách AI đang được sử dụng trong cuộc sống hàng ngày của chúng ta. Có khả năng nhiều nhiệm vụ menial được thực hiện bởi các tài xế xe tải và trợ lý cửa hàng có thể và sẽ được tự động hóa. Robot nông trại có thể sẽ giảm chi phí cho canh tác hữu cơ nhưng chúng cũng sẽ tự động hóa các hoạt động thu hoạch. Giai đoạn này của cuộc cách mạng công nghiệp có thể có hậu quả sâu sắc đối với những phần lớn của xã hội, vì tài xế xe tải và trợ lý cửa hàng là một số công việc phổ biến nhất ở nhiều quốc gia. Hơn nữa, các mô hình thống kê, khi áp dụng mà không cần chăm sóc có thể dẫn đến thiên vị chủng tộc, giới tính

hoặc tuổi tác và nêu ra những lo ngại hợp lý về sự công bằng về thủ tục nếu tự động thúc đẩy các quyết định do hậu quả. Điều quan trọng là đảm bảo rằng các thuật toán này được sử dụng cẩn thận. Với những gì chúng ta biết ngày nay, điều này gây ra cho chúng ta một mối quan tâm cấp bách hơn nhiều so với tiềm năng của siêu trí thông minh ác độc để tiêu diệt nhân loại.

2.7 các điểm

Cho đến nay, chúng ta đã nói về máy học một cách rộng rãi, vừa là một nhánh của AI vừa là cách tiếp cận với AI. Mặc dù deep learning là một tập hợp con của machine learning, nhưng tập hợp các thuật toán và ứng dụng chóng mặt gây khó khăn cho việc đánh giá các thành phần cụ thể cho học sâu có thể là gì. Điều này khó khăn như cố gắng pin xuống các thành phần cần thiết cho pizza vì hầu hết mọi thành phần đều có thể thay thế.

Như chúng tôi đã mô tả, machine learning có thể sử dụng dữ liệu để tìm hiểu các biến đổi giữa đầu vào và đầu ra, chẳng hạn như chuyển đổi âm thanh thành văn bản trong nhận dạng giọng nói. Khi làm như vậy, thường cần phải đại diện cho dữ liệu theo cách phù hợp với các thuật toán để biến đổi các biểu diễn như vậy thành đầu ra. *Học sâu* là * sâu* theo nghĩa chính xác rằng các mô hình của nó học nhiều * lớp* biến đổi, trong đó mỗi lớp cung cấp biểu diễn ở một cấp độ. Ví dụ, các lớp gần đầu vào có thể đại diện cho các chi tiết cấp thấp của dữ liệu, trong khi các lớp gần đầu ra phân loại có thể đại diện cho các khái niệm trừu tượng hơn được sử dụng để phân biệt đối xử. Vì học đại diện *nhằm mục đích tìm ra bản thân đại diện, học sâu có thể được gọi là học đại diện đa cấp.

Các vấn đề mà chúng ta đã thảo luận cho đến nay, chẳng hạn như học từ tín hiệu âm thanh thô, giá trị pixel thô của hình ảnh hoặc ánh xạ giữa các câu có độ dài tùy ý và các đối tác của chúng bằng ngoại ngữ, là những vấn đề mà học sâu vượt trội và nơi các phương pháp học máy truyền thống chùn bước. Nó chỉ ra rằng các mô hình nhiều lớp này có khả năng giải quyết dữ liệu nhận thức cấp thấp theo cách mà các công cụ trước đây không thể. Có thể cho là điểm chung quan trọng nhất trong các phương pháp học sâu là việc sử dụng * đào tạo end-to-end*. Đó là, thay vì lắp ráp một hệ thống dựa trên các thành phần được điều chỉnh riêng lẻ, người ta xây dựng hệ thống và sau đó điều chỉnh hiệu suất của chúng cùng nhau. Ví dụ, trong tầm nhìn máy tính các nhà khoa học sử dụng để tách quá trình kỹ thuật *tính năng* khỏi quá trình xây dựng các mô hình học máy. Máy dò cạnh Canny (Canny, 1987) và máy trích xuất tính năng SIFT của Lowe (Lowe, 2004) trị vì tối cao trong hơn một thập kỷ như các thuật toán để lập bản đồ hình ảnh thành các vectơ tính năng. Trong những ngày đã qua, phần quan trọng của việc áp dụng máy học vào những vấn đề này bao gồm việc đưa ra các cách được thiết kế thủ công để chuyển đổi dữ liệu thành một số hình thức phù hợp với các mô hình nông. Thật không may, chỉ có rất ít mà con người có thể thực hiện bằng sự khéo léo so với một đánh giá nhất quán trên hàng triệu lựa chọn được thực hiện tự động bởi một thuật toán. Khi học sâu tiếp quản, các bộ chiết xuất tính năng này được thay thế bằng các bộ lọc được điều chỉnh tự động, mang lại độ chính xác vượt trội.

Do đó, một lợi thế chính của học sâu là nó thay thế không chỉ các mô hình nông ở cuối các đường ống học tập truyền thống, mà còn là quá trình kỹ thuật tính năng tốn nhiều lao động. Hơn nữa, bằng cách thay thế phần lớn tiền xử lý miền cụ thể, deep learning đã loại bỏ nhiều ranh giới mà trước đây tách tầm nhìn máy tính, nhận dạng giọng nói, xử lý ngôn ngữ tự nhiên, tin học y tế và các lĩnh vực ứng dụng khác, cung cấp một bộ công cụ thống nhất để giải quyết đa dạng vấn đề.

Ngoài đào tạo từ đầu đến cuối, chúng tôi đang trải qua một sự chuyển đổi từ mô tả thống kê tham số sang các mô hình hoàn toàn không tham số. Khi dữ liệu khan hiếm, người ta cần dựa vào việc đơn giản hóa các giả định về thực tế để có được các mô hình hữu ích. Khi dữ liệu dồi dào, điều này có thể được thay thế bằng các mô hình không tham số phù hợp với thực tế chính xác hơn. Ở một mức độ nào đó, điều này phản ánh sự tiến bộ mà vật lý trải qua vào giữa thế kỷ trước với sự sẵn có của máy tính. Thay vì giải các xấp xỉ tham số về cách điện tử hoạt động bằng tay, giờ đây người ta có thể dùng đến các mô phỏng số của các phương trình vi phân từng phần liên quan. Điều này đã dẫn đến các mô hình chính xác hơn nhiều, mặc dù thường xuyên với chi phí giải thích.

Một điểm khác biệt khác đối với công việc trước đây là việc chấp nhận các giải pháp tối ưu, giải quyết các vấn đề tối ưu hóa phi tuyến không lồi và sẵn sàng thử mọi thứ trước khi chứng minh chúng. Chủ nghĩa kinh nghiệm mới được tìm thấy này trong việc đối phó với các vấn đề thống kê, kết hợp với một dòng tài năng nhanh chóng đã dẫn đến tiến bộ nhanh chóng của các thuật toán thực tế, mặc dù trong nhiều trường hợp với chi phí sửa đổi và tái phát minh các công cụ tồn tại trong nhiều thập kỷ.

Cuối cùng, cộng đồng deep learning tự hào về việc chia sẻ các công cụ trên các ranh giới học thuật và doanh nghiệp, phát hành nhiều thư viện tuyệt vời, mô hình thống kê và các mạng được đào tạo làm nguồn mở. Theo tinh thần này, các máy tính xách tay hình thành cuốn sách này có sẵn miễn phí để phân phối và sử dụng. Chúng tôi đã làm việc chăm chỉ để giảm bớt các rào cản tiếp cận để mọi người tìm hiểu về học sâu và chúng tôi hy vọng rằng độc giả của chúng tôi sẽ được hưởng lợi từ việc này.

2.8 Tóm tắt

- Machine learning nghiên cứu cách các hệ thống máy tính có thể tận dụng trải nghiệm (thường là dữ liệu) để cải thiện hiệu suất tại các tác vụ cụ thể. Nó kết hợp các ý tưởng từ thống kê, khai thác dữ liệu và tối ưu hóa. Thông thường, nó được sử dụng như một phương tiện triển khai các giải pháp AI.
- Là một lớp học máy học, việc học đại diện tập trung vào cách tự động tìm ra cách thể hiện dữ liệu thích hợp. Deep learning là học đại diện đa cấp thông qua việc học nhiều lớp biến đổi.
- Deep learning thay thế không chỉ các mô hình nông ở cuối các đường ống học máy truyền thống, mà còn là quá trình kỹ thuật tính năng tốn nhiều lao động.
- Phần lớn tiến bộ gần đây trong học sâu đã được kích hoạt bởi sự phong phú của dữ liệu phát sinh từ các cảm biến giá rẻ và các ứng dụng quy mô Internet, và bởi sự tiến bộ đáng kể trong tính toán, chủ yếu là thông qua GPU.
- Tối ưu hóa toàn bộ hệ thống là một thành phần quan trọng trong việc đạt được hiệu suất cao. Sự sẵn có của các khung học sâu hiệu quả đã giúp thiết kế và triển khai điều này dễ dàng hơn đáng kể.

2.9 Bài tập

1. Những phần nào của mã mà bạn hiện đang viết có thể được "học", tức là, được cải thiện bằng cách học và tự động xác định các lựa chọn thiết kế được thực hiện trong mã của bạn? Mã của bạn có bao gồm các lựa chọn thiết kế heuristic không?
2. Những vấn đề mà bạn gặp phải có nhiều ví dụ về cách giải quyết chúng, nhưng không có cách nào cụ thể để tự động hóa chúng? Đây có thể là những ứng cử viên chính để sử dụng deep learning.
3. Xem sự phát triển của AI như một cuộc cách mạng công nghiệp mới, mối quan hệ giữa các thuật toán và dữ liệu là gì? Nó có tương tự như động cơ hơi nước và than? Sự khác biệt cơ bản là gì?
4. Bạn có thể áp dụng phương pháp đào tạo từ đầu đến cuối ở đâu khác, chẳng hạn như trong Fig. 2.1.2, vật lý, kỹ thuật và kinh tế học?

Discussions[36]

[36] https://discuss.d2l.ai/t/22

3 | Sơ bộ

Để bắt đầu học sâu, chúng ta sẽ cần phát triển một vài kỹ năng cơ bản. Tất cả machine learning đều liên quan đến việc trích xuất thông tin từ dữ liệu. Vì vậy, chúng ta sẽ bắt đầu bằng cách học các kỹ năng thực tế để lưu trữ, thao tác và xử lý dữ liệu trước.

Hơn nữa, machine learning thường yêu cầu làm việc với các tập dữ liệu lớn, mà chúng ta có thể nghĩ là bảng, trong đó các hàng tương ứng với các ví dụ và các cột tương ứng với các thuộc tính. Đại số tuyến tính cho chúng ta một bộ kỹ thuật mạnh mẽ để làm việc với dữ liệu dạng bảng. Chúng tôi sẽ không đi quá xa vào cỏ dại mà tập trung vào cơ bản của các hoạt động ma trận và việc thực hiện chúng.

Ngoài ra, học sâu là tất cả về tối ưu hóa. Chúng tôi có một mô hình với một số thông số và chúng tôi muốn tìm những thông số phù hợp với dữ liệu của chúng tôi* tốt nhất*. Xác định cách nào để di chuyển từng tham số ở mỗi bước của một thuật toán đòi hỏi một chút giải tích, sẽ được giới thiệu ngắn gọn. May mắn thay, gói `autograd` tự động tính toán sự khác biệt cho chúng tôi và chúng tôi sẽ đề cập đến nó tiếp theo.

Tiếp theo, machine learning có liên quan đến việc đưa ra dự đoán: giá trị khả năng của một số thuộc tính không xác định là gì, với thông tin mà chúng ta quan sát? Để lý do nghiêm ngặt dưới sự không chắc chắn, chúng ta sẽ cần phải gọi ngôn ngữ xác suất.

Cuối cùng, tài liệu chính thức cung cấp nhiều mô tả và ví dụ vượt ra ngoài cuốn sách này. Để kết thúc chương, chúng tôi sẽ chỉ cho bạn cách tra cứu tài liệu cho các thông tin cần thiết.

Cuốn sách này đã giữ nội dung toán học ở mức tối thiểu cần thiết để có được sự hiểu biết đúng đắn về học sâu. Tuy nhiên, điều đó không có nghĩa là cuốn sách này là toán học miễn phí. Do đó, chương này cung cấp một giới thiệu nhanh chóng về toán học cơ bản và thường được sử dụng để cho phép bất cứ ai hiểu ít nhất *most* về nội dung toán học của cuốn sách. Nếu bạn muốn hiểu *tất cả* của nội dung toán học, xem xét thêm online appendix on mathematics[37] là đủ.

3.1 Thao tác dữ liệu

Để hoàn thành bất cứ điều gì, chúng ta cần một số cách để lưu trữ và thao tác dữ liệu. Nói chung, có hai điều quan trọng chúng ta cần làm với dữ liệu: (i) có được chúng; và (ii) xử lý chúng một khi chúng ở trong máy tính. Không có điểm nào trong việc thu thập dữ liệu mà không cần một cách nào đó để lưu trữ nó, vì vậy hãy để chúng tôi làm bẩn tay trước bằng cách chơi với dữ liệu tổng hợp. Để bắt đầu, chúng tôi giới thiệu mảng n chiều, còn được gọi là * tensor*.

Nếu bạn đã làm việc với NumPy, gói máy tính khoa học được sử dụng rộng rãi nhất trong Python, thì bạn sẽ thấy phần này quen thuộc. Cho dù bạn sử dụng khung nào, lớp tensor* của nó* (`ndarray` trong MXNet, `Tensor` trong cả PyTorch và TensorFlow) tương tự như `ndarray` của NumPy với một vài tính năng giết người. Đầu tiên, GPU được hỗ trợ tốt để tăng tốc tính toán trong khi NumPy chỉ hỗ trợ tính toán CPU. Thứ hai, lớp tensor hỗ trợ sự khác biệt tự động. Những tính chất này làm cho lớp tensor phù hợp với học sâu. Trong

[37] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/index.html

suốt cuốn sách, khi chúng tôi nói hàng chục, chúng tôi đang đề cập đến các trường hợp của lớp tensor trừ khi có quy định khác.

3.1.1 Bắt đầu

Trong phần này, chúng tôi mong muốn giúp bạn hoạt động, trang bị cho bạn các công cụ toán học và tính toán số cơ bản mà bạn sẽ xây dựng khi bạn tiến bộ qua cuốn sách. Đừng lo lắng nếu bạn đấu tranh để grok một số khái niệm toán học hoặc chức năng thư viện. Các phần sau đây sẽ xem xét lại tài liệu này trong bối cảnh các ví dụ thực tế và nó sẽ chìm vào. Mặt khác, nếu bạn đã có một số nền tảng và muốn đi sâu hơn vào nội dung toán học, chỉ cần bỏ qua phần này.

Để bắt đầu, chúng tôi nhập `torch`. Lưu ý rằng mặc dù nó được gọi là PyTorch, chúng ta nên nhập `torch` thay vì `pytorch`.

```
import torch
```

A tensor đại diện cho một mảng (có thể đa chiều) của các giá trị số. Với một trục, một tensor được gọi là *vector*. Với hai trục, một tensor được gọi là *matri*. Với $k > 2$ trục, chúng tôi thả các tên chuyên biệt và chỉ cần tham khảo đối tượng dưới dạng k^{th} *căng đơn hàng*.

PyTorch cung cấp một loạt các chức năng để tạo ra các hàng chục mới prepopulated với các giá trị. Ví dụ, bằng cách gọi `arange(n)`, chúng ta có thể tạo một vectơ có giá trị cách đều nhau, bắt đầu từ 0 (bao gồm) và kết thúc ở `n` (không bao gồm). Theo mặc định, kích thước khoảng thời gian là 1. Trừ khi có quy định khác, hàng chục mới được lưu trữ trong bộ nhớ chính và được chỉ định cho tính toán dựa trên CPU.

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

Chúng tôi có thể truy cập một tensor*shape* (chiều dài dọc theo mỗi trục) bằng cách kiểm tra thuộc tính `shape` của nó.

```
x.shape
```

```
torch.Size([12])
```

Nếu chúng ta chỉ muốn biết tổng số phần tử trong một tensor, tức là, sản phẩm của tất cả các yếu tố hình dạng, chúng ta có thể kiểm tra kích thước của nó. Bởi vì chúng ta đang xử lý một vector ở đây, phần tử duy nhất của `shape` của nó giống hệt với kích thước của nó.

```
x.numel()
```

```
12
```

Để thay đổi hình dạng của tensor mà không thay đổi số phần tử hoặc giá trị của chúng , chúng ta có thể gọi hàm `reshape`. Ví dụ, chúng ta có thể biến đổi tensor của mình, `x`, từ một vectơ hàng có hình dạng (12,) thành ma trận có hình dạng (3, 4). Tensor mới này chứa các giá trị chính xác giống nhau, nhưng xem chúng như một ma trận được tổ chức thành 3 hàng và 4 cột. Để nhắc lại, mặc dù hình dạng đã thay đổi, các yếu tố không có. Lưu ý rằng kích thước không bị thay đổi bằng cách định hình lại.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

Định hình lại bằng cách chỉ định thủ công mọi chiều là không cần thiết. Nếu hình dạng mục tiêu của chúng ta là một ma trận có hình dạng (chiều cao, chiều rộng), thì sau khi chúng ta biết chiều rộng, chiều cao được đưa ra ngầm. Tại sao chúng ta phải tự thực hiện việc phân chia? Trong ví dụ trên, để có được một ma trận với 3 hàng, chúng tôi đã chỉ định cả hai rằng nó phải có 3 hàng và 4 cột. May mắn thay, hàng chục có thể tự động làm việc ra một chiều cho phần còn lại. Chúng tôi gọi khả năng này bằng cách đặt `-1` cho kích thước mà chúng tôi muốn hàng chục tự động suy ra. Trong trường hợp của chúng tôi, thay vì gọi `x.reshape(3, 4)`, chúng tôi có thể gọi tương đương `x.reshape(-1, 4)` hoặc `x.reshape(3, -1)`.

Thông thường, chúng ta sẽ muốn ma trận của chúng tôi khởi tạo hoặc với số không, một số hằng số khác, hoặc số được lấy mẫu ngẫu nhiên từ một phân phối cụ thể. [**Chúng ta có thể tạo một tensor đại diện cho một tensor với tất cả các phần tử được đặt là 0**] và một hình dạng của (2, 3, 4) như sau:

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Tương tự, ta có thể tạo tenors với mỗi phần tử đặt thành 1 như sau:

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

Thông thường, chúng ta muốn lấy mẫu ngẫu nhiên các giá trị cho mỗi phần tử trong một tensor từ một số phân phối xác suất. Ví dụ, khi chúng ta xây dựng mảng để đóng vai trò là tham số trong mạng thần kinh, chúng ta thường sẽ khởi tạo các giá trị của chúng một cách ngẫu nhiên. Đoạn mã sau đây tạo ra một tensor với hình dạng (3, 4). Mỗi phần tử của nó được lấy mẫu ngẫu nhiên từ một phân bố Gaussian (bình thường) chuẩn với trung bình 0 và độ lệch chuẩn là 1.

```
torch.randn(3, 4)
```

```
tensor([[-0.6519,  0.2903,  1.9321, -0.7751],
        [ 1.2866, -0.1815,  1.2278,  1.9197],
        [-0.1778,  0.3601,  0.6502, -0.0875]])
```

Chúng ta cũng có thể chỉ định các giá trị chính xác cho mỗi element trong tensor mong muốn bằng cách cung cấp một danh sách Python (hoặc danh sách các danh sách) chứa các giá trị số. Ở đây, danh sách ngoài cùng tương ứng với trục 0 và danh sách bên trong đến trục 1.

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

3.1.2 Hoạt động

Cuốn sách này không phải là về kỹ thuật phần mềm. Sở thích của chúng tôi không giới hạn chỉ đơn giản là đọc và viết dữ liệu từ/đến mảng. Chúng tôi muốn thực hiện các hoạt động toán học trên những mảng. Một số thao tác đơn giản và hữu ích nhất là các hoạt động *elementwise*. Chúng áp dụng một phép toán vô hướng tiêu chuẩn cho mỗi phần tử của một mảng. Đối với các hàm lấy hai mảng làm đầu vào, các phép toán elementwise áp dụng một số toán tử nhị phân chuẩn trên mỗi cặp phần tử tương ứng từ hai mảng. Chúng ta có thể tạo ra một hàm elementwise từ bất kỳ hàm nào mà bản đồ từ vô hướng đến vô hướng.

Trong ký hiệu toán học, chúng ta sẽ biểu thị một toán tử vô hướng * unary* như vậy (lấy một đầu vào) bằng chữ ký $f : \mathbb{R} \rightarrow \mathbb{R}$. Điều này chỉ có nghĩa là chức năng đang ánh xạ từ bất kỳ số thực nào ($\mathbb{R}$) lên một số khác. Tương tự như vậy, chúng tôi biểu thị một toán tử vô hướng * binary* (lấy hai đầu vào thực và mang lại một đầu ra) bằng chữ ký $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$. Với bất kỳ hai vectơ **u** và **v** * có cùng hình dạng*, và một toán tử nhị phân f, chúng ta có thể tạo ra một vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ bằng cách đặt $c_i \leftarrow f(u_i, v_i)$ cho tất cả i, trong đó c_i, u_i, và v_i là các yếu tố i^{th} của vectơ **c**, **u**, và v_i 25. Ở đây, chúng tôi đã tạo ra các vectơ có giá trị $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$ bằng *lifting* hàm vô hướng cho một hoạt động vectơ elementwise.

Các toán tử số học tiêu chuẩn phổ biến (+, -, *, /, và **) đều được nâng lên * để hoạt động yếu tố cho bất kỳ hàng chục hình dạng giống hệt nhau nào có hình dạng tùy ý. Chúng ta có thể gọi các hoạt động elementwise trên bất kỳ hai hàng chục có cùng hình dạng. Trong ví dụ sau, chúng ta sử dụng dấu phẩy để xây dựng một tuple 5 phần tử, trong đó mỗi phần tử là kết quả của một phép toán elementwise.

Hoạt động

Các toán tử số học tiêu chuẩn phổ biến (+, -, *, /, và **) đều đã được nâng lên * thành các hoạt động elementwise

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y  # The ** operator is exponentiation
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
```

(continues on next page)

```
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

Nhiều nhiều thao tác hơn có thể được áp dụng elementwise, bao gồm các toán tử đơn nguyên như hàm mũ.

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

Ngoài tính toán elementwise, chúng ta cũng có thể thực hiện các phép toán đại số tuyến tính, bao gồm các sản phẩm chấm vector và phép nhân ma trận. Chúng tôi sẽ giải thích các bit quan trọng của đại số tuyến tính (không có kiến thức nào được giả định trước) trong Section 3.3.

Chúng ta cũng có thể * nối nhiều chục với nhau, xếp chúng từ đầu đến cuối để tạo thành một tensor lớn hơn. Chúng ta chỉ cần cung cấp một danh sách các hàng chục và nói với hệ thống dọc theo trục nào để nối. Ví dụ dưới đây cho thấy những gì xảy ra khi chúng ta nối hai ma trận dọc theo các hàng (trục 0, phần tử đầu tiên của hình dạng) so với các cột (trục 1, phần tử thứ hai của hình dạng). Chúng ta có thể thấy rằng độ dài trục 0 của tensor đầu ra đầu tiên (6) là tổng của chiều dài trục 0 của hai căng đầu vào ($3 + 3$); trong khi độ dài trục của bộ căng đầu ra thứ hai (8) là tổng của chiều dài trục 1 của hai căng đầu vào ($4 + 4$).

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Đôi khi, chúng ta muốn xây dựng một tensor nhị phân thông qua *trạng thái logic *. Lấy `X == Y` làm ví dụ. Đối với mỗi vị trí, nếu `X` và `Y` bằng nhau tại vị trí đó thì mục nhập tương ứng trong tensor mới lấy giá trị là 1, nghĩa là câu lệnh logic `X == Y` đúng tại vị trí đó; nếu không vị trí đó mất 0.

```
X == Y
```

```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

Tổng hợp tất cả các yếu tố trong tensor mang lại một tensor chỉ với một phần tử.

```
X.sum()
```

```
tensor(66.)
```

3.1.3 Cơ chế phát sóng

Trong phần trên, chúng ta đã thấy cách thực hiện các hoạt động elementwise trên hai hàng chục có cùng hình dạng. Trong một số điều kiện nhất định, ngay cả khi hình dạng khác nhau, chúng ta vẫn có thể thực hiện các hoạt động elementwise bằng cách gọi cơ chế phát sóng *. Cơ chế này hoạt động theo cách sau: Đầu tiên, mở rộng một hoặc cả hai mảng bằng cách sao chép các phần tử một cách thích hợp để sau khi chuyển đổi này, hai hàng chục có cùng một hình dạng. Thứ hai, thực hiện các hoạt động elementwise trên các mảng kết quả.

Trong hầu hết các trường hợp, chúng tôi phát dọc theo một trục mà một mảng ban đầu chỉ có chiều dài 1, chẳng hạn như trong ví dụ sau:

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

Vì `a` và `b` tương ứng là 3×1 và 1×2 ma trận, hình dạng của chúng không khớp nếu chúng ta muốn thêm chúng. Chúng tôi * phát sóng * các mục của cả hai ma trận thành ma trận 3×2 lớn hơn như sau: đối với ma trận `a`, nó sao chép các cột và cho ma trận `b` nó sao chép các hàng trước khi thêm cả hai elementwise.

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

3.1.4 Lập chỉ mục và cắt

Cũng giống như trong bất kỳ mảng Python nào khác, các phần tử trong một tensor có thể được truy cập bằng chỉ mục. Như trong bất kỳ mảng Python nào, phần tử đầu tiên có chỉ số 0 và phạm vi được chỉ định để bao gồm phần tử đầu tiên nhưng * trước* phần tử cuối cùng. Như trong danh sách Python tiêu chuẩn, chúng ta có thể truy cập các phần tử theo vị trí tương đối của chúng đến cuối danh sách bằng cách sử dụng các chỉ số âm.

Do đó, `[-1]` chọn phần tử cuối cùng và `[1:3]` chọn phần tử thứ hai và thứ ba như sau:

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Ngoài việc đọc, chúng ta cũng có thể viết các phần tử của ma trận bằng cách chỉ định các chỉ số.

```
X[1, 2] = 9
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  9.,  7.],
        [ 8.,  9., 10., 11.]])
```

Nếu chúng ta muốn để gán nhiều phần tử cùng một giá trị, chúng ta chỉ cần lập chỉ mục tất cả chúng và sau đó gán cho họ giá trị. Ví dụ, `[0:2, :]` truy cập các hàng đầu tiên và thứ hai, trong đó `:` lấy tất cả các phần tử dọc theo trục 1 (cột). Trong khi chúng tôi thảo luận về lập chỉ mục cho ma trận, điều này rõ ràng cũng hoạt động cho vectơ và cho hàng chục hơn 2 chiều.

```
X[0:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

3.1.5 Tiết kiệm bộ nhớ

Các thao tác chạy có thể khiến bộ nhớ mới được phân bổ cho kết quả máy chủ Ví dụ, nếu chúng ta viết `Y = X + Y`, chúng ta sẽ hủy bỏ tensor mà `Y` dùng để trỏ đến và thay vào đó chỉ `Y` vào bộ nhớ mới được phân bổ. Trong ví dụ sau, chúng ta chứng minh điều này với hàm `id()` của Python, cung cấp cho chúng ta địa chỉ chính xác của đối tượng tham chiếu trong bộ nhớ. Sau khi chạy `Y = Y + X`, chúng tôi sẽ thấy rằng `id(Y)` chỉ đến một vị trí khác. Đó là do Python lần đầu tiên đánh giá `Y + X`, phân bổ bộ nhớ mới cho kết quả và sau đó làm cho `Y` trỏ đến vị trí mới này trong bộ nhớ.

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

Điều này có thể là không mong muốn vì hai lý do. Đầu tiên, chúng tôi không muốn chạy xung quanh phân bổ bộ nhớ một cách không cần thiết tất cả các thời gian. Trong machine learning, chúng ta có thể có hàng trăm megabyte tham số và cập nhật tất cả chúng nhiều lần mỗi giây. Thông thường, chúng tôi sẽ muốn thực hiện các bản cập nhật này * tại chỗ *. Thứ hai, chúng ta có thể chỉ vào các tham số tương tự từ nhiều biến. Nếu chúng ta không cập nhật tại chỗ, các tham chiếu khác vẫn sẽ trỏ đến vị trí bộ nhớ cũ, làm cho nó có thể cho các phần của mã của chúng ta vô tình tham chiếu các tham số cũ.

May mắn thay, thực hiện hoạt động tại chỗ rất dễ dàng. Chúng ta có thể gán kết quả của một hoạt động cho một mảng được phân bổ trước đó với ký hiệu slice, ví dụ, `Y[:] = <expression>`. Để minh họa khái niệm này, trước tiên chúng ta tạo ra một ma trận mới `Z` với hình dạng tương tự như `Y` khác, sử dụng `zeros_like` để phân bổ một khối 0 mục nhập.

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 140201478059168
id(Z): 140201478059168
```

Nếu giá trị của `X` không được sử dụng lại trong các tính toán tiếp theo, chúng ta cũng có thể sử dụng `X[:] = X + Y` hoặc `X += Y` để giảm chi phí bộ nhớ của hoạt động.

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

3.1.6 Chuyển đổi sang các đối tượng Python khác

Chuyển đổi sang tensor NumPy (`ndarray`) , hoặc ngược lại, rất dễ dàng. Ngọn đuốc Tensor và mảng numpy sẽ chia sẻ vị trí bộ nhớ cơ bản của chúng, và thay đổi một thông qua một hoạt động tại chỗ cũng sẽ thay đổi vị trí khác.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

Để chuyển đổi một tensor size-1 thành một scalar Python , chúng ta có thể gọi hàm `item` hoặc các hàm tích hợp sẵn của Python.

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

3.1.7 Tóm tắt

- Giao diện chính để lưu trữ và thao tác dữ liệu cho học sâu là tensor (mảng n chiều). Nó cung cấp một loạt các chức năng bao gồm các hoạt động toán học cơ bản, phát sóng, lập chỉ mục, cắt lát, tiết kiệm bộ nhớ và chuyển đổi sang các đối tượng Python khác.

3.1.8 Bài tập

1. Chạy mã trong phần này. Thay đổi câu lệnh có điều kiện `X == Y` trong phần này thành `X < Y` hoặc `X > Y`, và sau đó xem loại tensor bạn có thể nhận được.
2. Thay thế hai hàng chục hoạt động theo phần tử trong cơ chế phát sóng bằng các hình dạng khác, ví dụ, các dụng cụ 3 chiều. Kết quả có giống như mong đợi không?

Discussions[38]

3.2 Xử lý sơ bộ dữ liệu

Cho đến nay chúng tôi đã giới thiệu một loạt các kỹ thuật để thao tác dữ liệu đã được lưu trữ trong hàng chục. Để áp dụng deep learning để giải quyết các vấn đề trong thế giới thực, chúng ta thường bắt đầu với tiền xử lý dữ liệu thô, thay vì những dữ liệu được chuẩn bị độc đáo ở định dạng tensor. Trong số các công cụ phân tích dữ liệu phổ biến trong Python, gói `pandas` thường được sử dụng. Giống như nhiều gói mở rộng khác trong hệ sinh thái rộng lớn của Python, `pandas` có thể làm việc cùng với hàng chục. Vì vậy, chúng tôi sẽ đi qua một thời gian ngắn các bước để xử lý trước dữ liệu thô với `pandas` và chuyển đổi chúng sang định dạng tensor. Chúng tôi sẽ đề cập đến nhiều kỹ thuật tiền xử lý dữ liệu hơn trong các chương sau.

3.2.1 Đọc tập dữ liệu

Ví dụ, chúng ta bắt đầu bằng cách tạo một tập dữ liệu nhân tạo được lưu trữ trong tệp csv (giá trị phân cách bằng dấu phẩy) `../data/house_tiny.csv`. Dữ liệu được lưu trữ ở các định dạng khác có thể được xử lý theo những cách tương tự.

Dưới đây chúng tôi viết từng hàng tập dữ liệu vào một tệp csv.

```
import os

os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n')  # Column names
    f.write('NA,Pave,127500\n')  # Each row represents a data example
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')
```

Để tải tập dữ liệu thô từ tệp csv đã tạo , chúng tôi nhập gói `pandas` và gọi hàm `read_csv`. Tập dữ liệu này có bốn hàng và ba cột, trong đó mỗi hàng mô tả số phòng (“NumRooms”), kiểu hẻm (“Alley”), và giá (“Price”) của một ngôi nhà.

[38] https://discuss.d2l.ai/t/27

```
# If pandas is not installed, just uncomment the following line:
# !pip install pandas
import pandas as pd

data = pd.read_csv(data_file)
print(data)
```

```
   NumRooms Alley   Price
0       NaN  Pave  127500
1       2.0   NaN  106000
2       4.0   NaN  178100
3       NaN   NaN  140000
```

3.2.2 Xử lý dữ liệu bị thiếu

Lưu ý rằng các mục "nan" bị thiếu giá trị. Để xử lý dữ liệu bị thiếu, các phương pháp điển hình bao gồm *imputation* và *xoá*, trong đó imputation thay thế các giá trị bị thiếu bằng các giá trị thay thế, trong khi xóa bỏ qua các giá trị bị thiếu. Ở đây chúng tôi sẽ xem xét imputation.

Bằng cách lập chỉ mục dựa trên vị trí số nguyên (`iloc`), chúng tôi chia `data` thành `inputs` và `outputs`, trong đó cái trước lấy hai cột đầu tiên trong khi cột sau chỉ giữ cột cuối cùng. Đối với các giá trị số trong `inputs` bị thiếu, chúng ta thay thế các mục "nan" bằng giá trị trung bình của cùng một cột.

```
inputs, outputs = data.iloc[:, 0:2], data.iloc[:, 2]
inputs = inputs.fillna(inputs.mean())
print(inputs)
```

```
   NumRooms Alley
0       3.0  Pave
1       2.0   NaN
2       4.0   NaN
3       3.0   NaN
```

Đối với các giá trị phân loại hoặc rời rạc trong `inputs`, chúng tôi coi "nan" là một thể loại. Vì cột "Alley" chỉ lấy hai loại giá trị phân loại "Pave" và "nan", `pandas` có thể tự động chuyển đổi cột này thành hai cột "Alley_Pave" và "Alley_nan". Một hàng có kiểu hẻm là "Pave" sẽ đặt giá trị của "Alley_Pave" và "Alley_nan" thành 1 và 0. Một hàng có kiểu hẻm bị thiếu sẽ đặt giá trị của chúng thành 0 và 1.

```
inputs = pd.get_dummies(inputs, dummy_na=True)
print(inputs)
```

```
   NumRooms  Alley_Pave  Alley_nan
0       3.0           1          0
1       2.0           0          1
2       4.0           0          1
3       3.0           0          1
```

3.2.3 Chuyển đổi sang định dạng Tensor

Bây giờ tất cả các mục trong `inputs` và `outputs` đều là số, chúng có thể được chuyển đổi sang định dạng tensor. Khi dữ liệu ở định dạng này, chúng có thể được thao tác thêm với các chức năng tensor mà chúng tôi đã giới thiệu trong Section 3.1.

```
import torch

X, y = torch.tensor(inputs.values), torch.tensor(outputs.values)
X, y
```

```
(tensor([[3., 1., 0.],
         [2., 0., 1.],
         [4., 0., 1.],
         [3., 0., 1.]], dtype=torch.float64),
 tensor([127500, 106000, 178100, 140000]))
```

3.2.4 Tóm tắt

- Giống như nhiều gói mở rộng khác trong hệ sinh thái rộng lớn của Python, `pandas` có thể làm việc cùng với hàng chục.
- Imputation và xóa có thể được sử dụng để xử lý dữ liệu bị thiếu.

3.2.5 Bài tập

Tạo một tập dữ liệu thô với nhiều hàng và cột hơn.

1. Xóa cột với các giá trị còn thiếu nhiều nhất.
2. Chuyển đổi tập dữ liệu được xử lý trước sang định dạng tensor.

Discussions[39]

3.3 Đại số tuyến tính

Bây giờ bạn có thể lưu trữ và thao tác dữ liệu, chúng ta hãy xem xét ngắn gọn tập hợp con của đại số tuyến tính cơ bản mà bạn sẽ cần hiểu và thực hiện hầu hết các mô hình được đề cập trong cuốn sách này. Dưới đây, chúng tôi giới thiệu các đối tượng toán học cơ bản, số học, và phép toán trong đại số tuyến tính, thể hiện từng đối tượng trong số chúng thông qua ký hiệu toán học và việc thực hiện tương ứng trong mã.

[39] https://discuss.d2l.ai/t/29

3.3.1 Vô hướng

Nếu bạn chưa bao giờ học đại số tuyến tính hoặc học máy, thì kinh nghiệm trong quá khứ của bạn với toán học có thể bao gồm suy nghĩ về một số tại một thời điểm. Và, nếu bạn đã bao giờ cân bằng một sổ séc hoặc thậm chí trả tiền cho bữa tối tại một nhà hàng thì bạn đã biết làm thế nào để làm những việc cơ bản như thêm và nhân các cặp số. Ví dụ, nhiệt độ ở Palo Alto là 52 độ F. Chính thức, chúng ta gọi các giá trị chỉ bao gồm một số lượng * vô số*. Nếu bạn muốn chuyển đổi giá trị này sang Celsius (thang nhiệt độ hợp lý hơn của hệ mét), bạn sẽ đánh giá biểu thức $c = \frac{5}{9}(f - 32)$, đặt f thành 52. Trong phương trình này, mỗi thuật ngữ — 5, 9, và 32—là những giá trị vô hướng. Các giữ chỗ c và f được gọi là *variables* và chúng đại diện cho các giá trị vô hướng không xác định.

Trong cuốn sách này, chúng ta áp dụng ký hiệu toán học trong đó các biến vô hướng được biểu thị bằng các chữ cái thường thấp hơn (ví dụ: x, y và z). Chúng tôi biểu thị không gian của tất cả (liên tục) * có giá trị thực* vô hướng bởi $\mathbb{R}$. Đối với sự nhanh chóng, chúng ta sẽ xem xét các định nghĩa nghiêm ngặt về chính xác *không gian* là gì, nhưng chỉ cần nhớ rằng biểu thức $x \in \mathbb{R}$ là một cách chính thức để nói rằng x là một vô hướng có giá trị thực. Ký hiệu $\in$ có thể được phát âm "trong" và chỉ đơn giản là biểu thị thành viên trong một bộ. Tương tự, chúng ta có thể viết $x, y \in \{0, 1\}$ để nói rằng x và y là những con số có giá trị chỉ có thể là 0 hoặc 1.

Một vô hướng được biểu diễn bằng một tensor chỉ với một yếu tố. Trong đoạn tiếp theo, chúng ta khởi tạo hai vô hướng và thực hiện một số phép toán số học quen thuộc với chúng, cụ thể là cộng, nhân, chia và hàm mũ.

```
import torch

x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

3.3.2 Vectơ

Bạn có thể nghĩ một vector chỉ đơn giản là một danh sách các giá trị vô hướng. Chúng ta gọi các giá trị này là * element* (*entries* hoặc *components*) của vector. Khi vectơ của chúng ta đại diện cho các ví dụ từ tập dữ liệu của chúng ta, các giá trị của chúng giữ một số ý nghĩa trong thế giới thực. Ví dụ: nếu chúng ta đang đào tạo một mô hình để dự đoán rủi ro mà một khoản vay mặc định, chúng ta có thể liên kết mỗi ứng viên với một vectơ có thành phần tương ứng với thu nhập, thời gian làm việc, số lượng mặc định trước đó và các yếu tố khác. Nếu chúng ta đang nghiên cứu nguy cơ đau tim bệnh nhân bệnh viện có khả năng phải đối mặt, chúng ta có thể đại diện cho mỗi bệnh nhân bằng một vectơ có thành phần nắm bắt các dấu hiệu quan trọng gần đây nhất của họ, mức cholesterol, phút tập thể dục mỗi ngày, vv Trong ký hiệu toán học, chúng ta thường sẽ biểu thị vectơ là mặt đậm, thấp hơn cased các chữ cái (ví dụ: **x**, **y** và **z**).

Chúng tôi làm việc với các vectơ thông qua các hàng chục một chiều. Trong hàng chục nói chung có thể có độ dài tùy ý, tùy thuộc vào giới hạn bộ nhớ của máy của bạn.

```
x = torch.arange(4)
x
```

```
tensor([0, 1, 2, 3])
```

Chúng ta có thể tham khảo bất kỳ phần tử nào của vectơ bằng cách sử dụng chỉ số dưới. Ví dụ: chúng ta có thể tham khảo phần tử i^{th} của $\mathbf{x}$ bởi x_i. Lưu ý rằng phần tử x_i là vô hướng, vì vậy chúng ta không phải đối mặt với phông chữ khi đề cập đến nó. Văn học mở rộng coi vectơ cột là định hướng mặc định của vectơ, cuốn sách này cũng vậy. Trong toán học, một vector $\mathbf{x}$ có thể được viết là

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{3.3.1}$$

trong đó $x_1, \ldots, x_n$ là các yếu tố của vectơ. Trong code, chúng ta truy cập bất kỳ phần tử nào bằng cách lập chỉ mục vào tensor.

```
x[3]
```

```
tensor(3)
```

Chiều dài, kích thước và hình dạng

Chúng ta hãy xem lại một số khái niệm từ Section 3.1. Một vectơ chỉ là một mảng các số. Và cũng giống như mỗi mảng có một chiều dài, vì vậy không mỗi vector. Trong ký hiệu toán học, nếu chúng ta muốn nói rằng một vector $\mathbf{x}$ bao gồm n vô hướng có giá trị thực, chúng ta có thể thể hiện điều này là $\mathbf{x} \in \mathbb{R}^n$. Độ dài của một vectơ thường được gọi là *dimension* của vectơ.

Như với một mảng Python thông thường, chúng ta có thể truy cập độ dài của một tensor bằng cách gọi hàm `len()` tích hợp của Python.

```
len(x)
```

```
4
```

Khi một tensor đại diện cho một vectơ (với chính xác một trục), chúng ta cũng có thể truy cập độ dài của nó thông qua thuộc tính `.shape`. Hình dạng là một tuple liệt kê chiều dài (chiều) dọc theo mỗi trục của tensor. Đối với hàng chục chỉ với một trục, hình dạng chỉ có một yếu tố.

```
x.shape
```

```
torch.Size([4])
```

Lưu ý rằng từ "chiều" có xu hướng bị quá tải trong các bối cảnh này và điều này có xu hướng gây nhầm lẫn cho mọi người. Để làm rõ, chúng ta sử dụng chiều của một * vector* hoặc một * axis* để chỉ chiều dài của nó, tức là số phần tử của một vectơ hoặc một trục. Tuy nhiên, chúng ta sử dụng chiều của một tensor để chỉ số trục mà một tensor có. Theo nghĩa này, chiều của một số trục của một tensor sẽ là chiều dài của trục đó.

3.3.3 Ma trận

Cũng giống như các vectơ tổng quát hóa vô hướng từ thứ tự số 0 để đặt hàng một, ma trận khái quát hóa vectơ từ thứ tự một đến thứ tự hai. Ma trận, mà chúng ta thường sẽ biểu thị bằng chữ in hoa, có mặt đậm (ví dụ: **X**, **Y** và **Z**), được biểu diễn bằng mã dưới dạng hàng chục với hai trục.

Trong ký hiệu toán, ta sử dụng $\mathbf{A} \in \mathbb{R}^{m \times n}$ để thể hiện rằng ma trận **A** bao gồm m hàng và n cột vô hướng có giá trị thực. Trực quan, chúng ta có thể minh họa bất kỳ ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ như một bảng, trong đó mỗi phần tử a_{ij} thuộc về hàng i^{th} và cột j^{th}:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{3.3.2}$$

Đối với bất kỳ $\mathbf{A} \in \mathbb{R}^{m \times n}$, hình dạng của **A** là (m, n) hoặc $m \times n$. Cụ thể, khi một ma trận có cùng số hàng và cột, hình dạng của nó trở thành một hình vuông; do đó, nó được gọi là matrận *vuông*.

Chúng ta có thể tạo một ma trận $m \times n$ bằng cách chỉ định một hình dạng với hai thành phần m và n khi gọi bất kỳ chức năng yêu thích nào của chúng tôi để khởi tạo một tensor.

```
A = torch.arange(20).reshape(5, 4)
A
```

```
tensor([[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11],
        [12, 13, 14, 15],
        [16, 17, 18, 19]])
```

Chúng ta có thể truy cập phần tử vô hướng a_{ij} của ma trận **A** trong (3.3.2) bằng cách chỉ định các chỉ số cho hàng (i) và cột (j), chẳng hạn như $[\mathbf{A}]_{ij}$. Khi các yếu tố vô hướng của ma trận **A**, chẳng hạn như trong (3.3.2), không được đưa ra, chúng ta có thể chỉ đơn giản sử dụng chữ thường của ma trận **A** với chỉ số dưới, a_{ij}, để tham khảo $[\mathbf{A}]_{ij}$. Để giữ ký hiệu đơn giản, dấu phẩy chỉ được chèn vào các chỉ số riêng biệt khi cần thiết, chẳng hạn như $a_{2,3j}$ và $[\mathbf{A}]_{2i-1,3}$.

Đôi khi, chúng tôi muốn lật các trục. Khi chúng ta trao đổi các hàng và cột của ma trận, kết quả được gọi là *transpose* của ma trận. Chính thức, chúng tôi biểu thị một ma trận **A** của transpose bởi $\mathbf{A}^\top$ và nếu $\mathbf{B} = \mathbf{A}^\top$, sau đó $b_{ij} = a_{ji}$ cho bất kỳ i và j. Như vậy, transpose của **A** trong (3.3.2) là một ma trận $n \times m$:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}. \tag{3.3.3}$$

Bây giờ chúng ta truy cập vào một matrix của transpose trong mã.

```
A.T
```

```
tensor([[ 0,  4,  8, 12, 16],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19]])
```

Là một loại đặc biệt của ma trận vuông, a *matrix đối xứng * $\mathbf{A}$ bằng chuyển vị của nó: $\mathbf{A} = \mathbf{A}^\top$. Ở đây chúng ta xác định ma trận đối xứng `B`.

```
B = torch.tensor([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

```
tensor([[1, 2, 3],
        [2, 0, 4],
        [3, 4, 5]])
```

Bây giờ chúng tôi so sánh `B` với transpose của nó.

```
B == B.T
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Ma trận là cấu trúc dữ liệu hữu ích: chúng cho phép chúng ta tổ chức dữ liệu có các phương thức biến thể khác nhau. Ví dụ: các hàng trong ma trận của chúng ta có thể tương ứng với các ngôi nhà khác nhau (ví dụ dữ liệu), trong khi các cột có thể tương ứng với các thuộc tính khác nhau. Điều này nghe có vẻ quen thuộc nếu bạn đã từng sử dụng phần mềm bảng tính hoặc đã đọc Section 3.2. Do đó, mặc dù định hướng mặc định của một vectơ duy nhất là một vectơ cột, trong một ma trận đại diện cho một tập dữ liệu dạng bảng, nó là thông thường hơn để coi mỗi ví dụ dữ liệu như một vectơ hàng trong ma trận. Và, như chúng ta sẽ thấy trong các chương sau, quy ước này sẽ cho phép các thực hành học sâu phổ biến. Ví dụ, dọc theo trục ngoài cùng của một tensor, chúng ta có thể truy cập hoặc liệt kê minibatches ví dụ dữ liệu, hoặc chỉ các ví dụ dữ liệu nếu không có minibatch tồn tại.

3.3.4 Tensors

Cũng giống như vectơ tổng quát hóa vô hướng, và ma trận tổng quát hóa vectơ, chúng ta có thể xây dựng cấu trúc dữ liệu với nhiều trục hơn nữa. **Tensors** cung cấp cho chúng ta một cách chung để mô tả các mảng n chiều với một số trục tùy ý. Vectơ, ví dụ, là các hàng chục bậc nhất, và ma trận là hàng chục bậc hai. Tensors được ký hiệu bằng chữ in hoa của một khuôn mặt phông chữ đặc biệt (ví dụ, X, Y, và Z) và cơ chế lập chỉ mục của chúng (ví dụ, x_{ijk} và $[\mathsf{X}]_{1,2i-1,3}$) tương tự như của ma trận.

Tensors sẽ trở nên quan trọng hơn khi chúng ta bắt đầu làm việc với hình ảnh, đến các mảng n chiều với 3 trục tương ứng với chiều cao, chiều rộng và trục * kênh* để xếp các kênh màu (đỏ, xanh lá cây và xanh dương). Hiện tại, chúng ta sẽ bỏ qua hàng chục bậc cao hơn và tập trung vào những điều cơ bản.

```
X = torch.arange(24).reshape(2, 3, 4)
X
```

```
tensor([[[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]],

        [[12, 13, 14, 15],
         [16, 17, 18, 19],
         [20, 21, 22, 23]]])
```

3.3.5 Tính chất cơ bản của Tensor Arithmetic

Vô hướng, vectơ, ma trận, và hàng chục ("tensors" trong tiểu mục này đề cập đến các đối tượng đại số) của một số trục tùy ý có một số tính chất tốt đẹp thường có ích. Ví dụ, bạn có thể nhận thấy từ định nghĩa của một phép toán elementwise rằng bất kỳ hoạt động đơn nguyên tố nào không thay đổi hình dạng của toán hạng của nó. Tương tự như vậy, với bất kỳ hai hàng chục có cùng hình dạng, kết quả của bất kỳ phép toán phần tử nhị phân nào sẽ là một tensor của cùng một hình dạng đó Ví dụ, thêm hai ma trận có cùng hình dạng thực hiện phép cộng elementwise trên hai ma trận này.

```
A = torch.arange(20, dtype=torch.float32).reshape(5, 4)
B = A.clone()  # Assign a copy of `A` to `B` by allocating new memory
A, A + B
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [12., 13., 14., 15.],
         [16., 17., 18., 19.]]),
 tensor([[ 0.,  2.,  4.,  6.],
         [ 8., 10., 12., 14.],
         [16., 18., 20., 22.],
         [24., 26., 28., 30.],
         [32., 34., 36., 38.]]))
```

Cụ thể, elementwise phép nhân của hai ma trận được gọi là sản phẩm * Hadamard* của họ (ký hiệu toán $\odot$). Xem xét ma trận $\mathbf{B} \in \mathbb{R}^{m \times n}$ có phần tử của hàng i và cột j là b_{ij}. Sản phẩm Hadamard của ma trận $\mathbf{A}$ (được định nghĩa trong (3.3.2)) và $\mathbf{B}$

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \tag{3.3.4}$$

```
A * B
```

```
tensor([[  0.,   1.,   4.,   9.],
        [ 16.,  25.,  36.,  49.],
        [ 64.,  81., 100., 121.],
        [144., 169., 196., 225.],
        [256., 289., 324., 361.]])
```

Nhân hoặc thêm một tensor với vô hướng cũng không thay đổi hình dạng của tensor, trong đó mỗi phần tử của tensor toán hạng sẽ được thêm hoặc nhân với vô hướng.

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

```
(tensor([[[ 2,  3,  4,  5],
          [ 6,  7,  8,  9],
```

(continues on next page)

```
          [10, 11, 12, 13]],

         [[14, 15, 16, 17],
          [18, 19, 20, 21],
          [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

3.3.6 Giảm

Một thao tác hữu ích mà chúng ta có thể thực hiện với các hàng chục tùy ý là tính toán tổng các phần tử của chúng. Trong ký hiệu toán học, chúng ta thể hiện các khoản tiền bằng ký hiệu $\sum$. Để thể hiện tổng của các phần tử trong một vector **x** chiều dài d, chúng tôi viết $\sum_{i=1}^{d} x_i$. Trong code, chúng ta chỉ có thể gọi hàm để tính tổng.

```
x = torch.arange(4, dtype=torch.float32)
x, x.sum()
```

```
(tensor([0., 1., 2., 3.]), tensor(6.))
```

Chúng ta có thể diễn đạt tổng trên các phần tử của hàng chục hình dạng tùy ý. Ví dụ, tổng các phần tử của ma trận $m \times n$ **A** có thể được viết $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum()
```

```
(torch.Size([5, 4]), tensor(190.))
```

Theo mặc định, gọi hàm để tính tổng *giảm* một tensor dọc theo tất cả các trục của nó đến một vô hướng. Chúng ta cũng có thể chỉ định các trục dọc theo đó tensor được giảm thông qua tổng hợp. Lấy ma trận làm ví dụ. Để giảm kích thước hàng (trục 0) bằng cách tổng hợp các phần tử của tất cả các hàng, chúng tôi chỉ định `axis=0` khi gọi hàm. Vì ma trận đầu vào giảm dọc theo trục 0 để tạo ra vectơ đầu ra, kích thước của trục 0 của đầu vào bị mất trong hình dạng đầu ra.

```
A_sum_axis0 = A.sum(axis=0)
A_sum_axis0, A_sum_axis0.shape
```

```
(tensor([40., 45., 50., 55.]), torch.Size([4]))
```

Chỉ định `axis=1` sẽ giảm kích thước cột (trục 1) bằng cách tổng hợp các phần tử của tất cả các cột. Do đó, kích thước của trục 1 của đầu vào bị mất trong hình dạng đầu ra.

```
A_sum_axis1 = A.sum(axis=1)
A_sum_axis1, A_sum_axis1.shape
```

```
(tensor([ 6., 22., 38., 54., 70.]), torch.Size([5]))
```

Giảm một ma trận dọc theo cả hàng và cột thông qua tổng tương đương với việc tổng hợp tất cả các phần tử của ma trận.

```
A.sum(axis=[0, 1])  # Same as `A.sum()`
```

```
tensor(190.)
```

Một số lượng liên quan là *mean*, còn được gọi là *trung bình*. Chúng tôi tính trung bình bằng cách chia tổng cho tổng số phần tử. Trong mã, chúng ta chỉ có thể gọi hàm để tính toán trung bình trên hàng chục hình dạng tùy ý.

```
A.mean(), A.sum() / A.numel()
```

```
(tensor(9.5000), tensor(9.5000))
```

Tương tự như vậy, chức năng tính trung bình cũng có thể làm giảm tensor dọc theo các trục được chỉ định.

```
A.mean(axis=0), A.sum(axis=0) / A.shape[0]
```

```
(tensor([ 8.,  9., 10., 11.]), tensor([ 8.,  9., 10., 11.]))
```

Tổng không giảm

Tuy nhiên, đôi khi có thể hữu ích khi giữ số trục không thay đổi khi gọi hàm tính tổng hoặc trung bình.

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A
```

```
tensor([[ 6.],
        [22.],
        [38.],
        [54.],
        [70.]])
```

Ví dụ, vì `sum_A` vẫn giữ hai trục của nó sau khi tổng hợp mỗi hàng, chúng ta có thể chia `A` cho `sum_A` với phát sóng.

```
A / sum_A
```

```
tensor([[0.0000, 0.1667, 0.3333, 0.5000],
        [0.1818, 0.2273, 0.2727, 0.3182],
        [0.2105, 0.2368, 0.2632, 0.2895],
        [0.2222, 0.2407, 0.2593, 0.2778],
        [0.2286, 0.2429, 0.2571, 0.2714]])
```

Nếu chúng ta muốn tính toán tổng tích lũy của các phần tử của `A` dọc theo một số trục , giả sử `axis=0` (từng hàng), chúng ta có thể gọi hàm `cumsum`. Chức năng này sẽ không làm giảm tensor đầu vào dọc theo bất kỳ trục nào.

```
A.cumsum(axis=0)
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  6.,  8., 10.],
        [12., 15., 18., 21.],
        [24., 28., 32., 36.],
        [40., 45., 50., 55.]])
```

3.3.7 Sản phẩm Dot

Cho đến nay, chúng tôi chỉ thực hiện các hoạt động elementwise, tổng và trung bình. Và nếu đây là tất cả những gì chúng ta có thể làm, đại số tuyến tính có lẽ sẽ không xứng đáng với phần riêng của nó. Tuy nhiên, một trong những hoạt động cơ bản nhất là sản phẩm chấm. Với hai vectơ $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, sản phẩm *dot của họ* $\mathbf{x}^\top \mathbf{y}$ (hoặc $\langle \mathbf{x}, \mathbf{y} \rangle$) là một tổng so với các sản phẩm của các phần tử ở cùng một vị trí: $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$.

```
y = torch.ones(4, dtype = torch.float32)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2., 3.]), tensor([1., 1., 1., 1.]), tensor(6.))
```

Lưu ý rằng chúng ta có thể diễn đạt tích chấm của hai vectơ tương đương bằng cách thực hiện phép nhân elementwise và sau đó là tổng:

```
torch.sum(x * y)
```

```
tensor(6.)
```

Các sản phẩm Dot rất hữu ích trong một loạt các bối cảnh. Ví dụ, cho một số tập hợp các giá trị, được biểu thị bằng một vectơ $\mathbf{x} \in \mathbb{R}^d$ và một tập hợp trọng số được ký hiệu bởi $\mathbf{w} \in \mathbb{R}^d$, tổng trọng số của các giá trị trong $\mathbf{x}$ theo trọng lượng $\mathbf{w}$ có thể được biểu thị dưới dạng sản phẩm chấm $\mathbf{x}^\top \mathbf{w}$. Khi trọng lượng không âm và tổng thành một (tức là $\left(\sum_{i=1}^{d} w_i = 1\right)$), sản phẩm chấm biểu thị trung bình * có trọng lự*. Sau khi bình thường hóa hai vectơ để có chiều dài đơn vị, các sản phẩm chấm thể hiện cosin của góc giữa chúng. Chúng tôi sẽ chính thức giới thiệu khái niệm này là * chiều dài* sau trong phần này.

3.3.8 Matrix-Vector Sản phẩm

Bây giờ chúng ta đã biết cách tính toán các sản phẩm chấm, chúng ta có thể bắt đầu hiểu các sản phẩm *ma thuật-vectơ *. Nhớ lại ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ và vector $\mathbf{x} \in \mathbb{R}^n$ được xác định và hình dung trong (3.3.2) và (3.3.1) tương ứng. Hãy để chúng tôi bắt đầu bằng cách hình dung ma trận $\mathbf{A}$ về vectơ hàng của nó

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \tag{3.3.5}$$

trong đó mỗi $\mathbf{a}_i^\top \in \mathbb{R}^n$ là một vectơ hàng đại diện cho hàng i^{th} của ma trận $\mathbf{A}$.

Sản phẩm ma trận-vector $\mathbf{Ax}$ chỉ đơn giản là một vector cột có chiều dài m, có i^{th} phần tử là sản phẩm chấm

$\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \tag{3.3.6}$$

Chúng ta có thể nghĩ đến phép nhân của một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$ như một biến đổi dự án vectơ từ $\mathbb{R}^n$ đến $\mathbb{R}^m$. Những biến đổi này trở nên hữu ích đáng kể. Ví dụ, chúng ta có thể biểu diễn các vòng quay dưới dạng phép nhân bằng một ma trận vuông. Như chúng ta sẽ thấy trong các chương tiếp theo, chúng ta cũng có thể sử dụng các sản phẩm ma thuật-vector để mô tả các phép tính chuyên sâu nhất cần thiết khi tính toán từng lớp trong mạng thần kinh với các giá trị của lớp trước đó.

Thể hiện các sản phẩm ma thuật-vector trong mã với hàng chục, chúng tôi sử dụng hàm `mv`. Khi chúng ta gọi `torch.mv(A, x)` với ma trận `A` và một vector `x`, sản phẩm ma thuật-vector được thực hiện. Lưu ý rằng kích thước cột của `A` (chiều dài của nó dọc theo trục 1) phải giống với kích thước `x` (chiều dài của nó).

```
A.shape, x.shape, torch.mv(A, x)
```

```
(torch.Size([5, 4]), torch.Size([4]), tensor([ 14.,  38.,  62.,  86., 110.]))
```

3.3.9 Phép nhân ma trận ma trận

Nếu bạn đã nhận được sự treo của các sản phẩm chấm và các sản phẩm ma thuật-vector, thì nhân * ma trận ma trận * phải đơn giản.

Nói rằng chúng ta có hai ma trận $\mathbf{A} \in \mathbb{R}^{n \times k}$ và $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \tag{3.3.7}$$

Biểu thị bằng $\mathbf{a}_i^\top \in \mathbb{R}^k$ vector hàng đại diện cho hàng i^{th} của ma trận $\mathbf{A}$, và để cho $\mathbf{b}_j \in \mathbb{R}^k$ là vectơ cột từ cột j^{th} của ma trận $\mathbf{B}$. Để sản xuất sản phẩm ma trận $\mathbf{C} = \mathbf{AB}$, dễ nhất là nghĩ đến $\mathbf{A}$ về vectơ hàng và $\mathbf{B}$ về vectơ cột của nó:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}. \tag{3.3.8}$$

Sau đó, sản phẩm ma trận $\mathbf{C} \in \mathbb{R}^{n \times m}$ được sản xuất như chúng ta chỉ đơn giản là tính toán từng phần tử c_{ij} như là sản phẩm chấm $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \tag{3.3.9}$$

Chúng ta có thể nghĩ đến phép nhân ma trận ma trận **AB** chỉ đơn giản là thực hiện m các sản phẩm ma trận vector và khâu kết quả lại với nhau để tạo thành ma trận $n \times m$ Trong đoạn mã sau, chúng tôi thực hiện phép

nhân ma trận trên `A` và `B`. Ở đây, `A` là một ma trận với 5 hàng và 4 cột, và `B` là một ma trận với 4 hàng và 3 cột. Sau khi nhân, chúng ta có được một ma trận với 5 hàng và 3 cột.

```
B = torch.ones(4, 3)
torch.mm(A, B)
```

```
tensor([[ 6.,  6.,  6.],
        [22., 22., 22.],
        [38., 38., 38.],
        [54., 54., 54.],
        [70., 70., 70.]])
```

Phép nhân ma trận ma trận có thể được gọi đơn giản là nhân *ma trận *, và không nên nhầm lẫn với sản phẩm Hadamard.

3.3.10 Định mức

Một số toán tử hữu ích nhất trong đại số tuyến tính là * chỉ tiêu *. Một cách không chính thức, định mức của một vector cho chúng ta biết cách * lớn* một vectơ là. Khái niệm về kích thước* đang được xem xét ở đây không liên quan đến chiều mà là độ lớn của các thành phần.

Trong đại số tuyến tính, một định mức vectơ là một hàm f ánh xạ một vectơ đến vô hướng, thỏa mãn một số ít các thuộc tính. Với bất kỳ vector $\mathbf{x}$ nào, thuộc tính đầu tiên nói rằng nếu chúng ta quy mô tất cả các phần tử của một vectơ bằng một hệ số không đổi α, định mức của nó cũng quy mô theo giá trị * tuyệt đối * của cùng một yếu tố không đổi:

$$f(\alpha\mathbf{x}) = |\alpha| f(\mathbf{x}). \tag{3.3.10}$$

Tính chất thứ hai là bất đẳng thức tam giác quen thuộc:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \tag{3.3.11}$$

Tài sản thứ ba chỉ đơn giản nói rằng định mức phải không tiêu cực:

$$f(\mathbf{x}) \geq 0. \tag{3.3.12}$$

Điều đó có ý nghĩa, như trong hầu hết các bối cảnh, * kích thước* nhỏ nhất cho bất cứ điều gì là 0. Thuộc tính cuối cùng yêu cầu định mức nhỏ nhất đạt được và chỉ đạt được bằng một vectơ bao gồm tất cả các số không.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \tag{3.3.13}$$

Bạn có thể nhận thấy rằng các định mức âm thanh rất giống với các biện pháp khoảng cách. Và nếu bạn nhớ khoảng cách Euclide (nghĩ rằng định lý Pythagoras) từ trường lớp, thì các khái niệm về không tiêu cực và bất bình đẳng tam giác có thể rung chuông. Trên thực tế, khoảng cách Euclide là một tiêu chuẩn: cụ thể nó là định mức L_2. Giả sử rằng các yếu tố trong vector n chiều $\mathbf{x}$ là $x_1, \ldots, x_n$.

L_2 norm của $\mathbf{x}$ là căn bậc hai của tổng các ô vuông của các phần tử vectơ:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}, \tag{3.3.14}$$

trong đó chỉ số dưới 2 thường bị bỏ qua trong L_2 chỉ tiêu, tức là $\|\mathbf{x}\|$ tương đương với $\|\mathbf{x}\|_2$. Trong mã, chúng ta có thể tính định mức L_2 của một vectơ như sau.

```
u = torch.tensor([3.0, -4.0])
torch.norm(u)
```

```
tensor(5.)
```

Trong học sâu, chúng tôi làm việc thường xuyên hơn với định mức L_2 bình phương.

Bạn cũng sẽ thường xuyên gặp phải the L_1 *norm*, được biểu thị dưới dạng tổng các giá trị tuyệt đối của các phần tử vectơ:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \,. \tag{3.3.15}$$

So với định mức L_2, nó ít bị ảnh hưởng bởi outliers. Để tính định mức L_1, ta soạn hàm giá trị tuyệt đối với một tổng trên các phần tử.

```
torch.abs(u).sum()
```

```
tensor(7.)
```

Cả định mức L_2 và định mức L_1 đều là những trường hợp đặc biệt của L_p * tiêu chuẩn* chung hơn:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} . \tag{3.3.16}$$

Tương tự như định mức L_2 của vectơ, chuẩn *Frobenius* của ma trận $\mathbf{X} \in \mathbb{R}^{m \times n}$ là căn bậc hai của tổng bình phương của các phần tử ma trận:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2}. \tag{3.3.17}$$

Định mức Frobenius thỏa mãn tất cả các thuộc tính của các chỉ tiêu vector. Nó hoạt động như thể nó là một định mức L_2 của một vectơ hình ma trận. Gọi hàm sau sẽ tính định mức Frobenius của một ma trận.

```
torch.norm(torch.ones((4, 9)))
```

```
tensor(6.)
```

Định mức và mục tiêu

Mặc dù chúng ta không muốn vượt quá xa bản thân, chúng ta có thể trồng một số trực giác về lý do tại sao những khái niệm này hữu ích. Trong học sâu, chúng ta thường cố gắng giải quyết các vấn đề tối ưu hóa: *tối đa hóa* xác suất được gán cho dữ liệu quan sát; *giảm xuống* khoảng cách giữa các dự đoán and the ground-truth đất sự thật observations quan sát. Gán biểu diễn vector cho các mục (như từ, sản phẩm hoặc bài báo tin tức) sao cho khoảng cách giữa các mục tương tự được giảm thiểu và khoảng cách giữa các mục khác nhau được tối đa hóa. Thông thường, các mục tiêu, có lẽ là các thành phần quan trọng nhất của các thuật toán học sâu (bên cạnh dữ liệu), được thể hiện dưới dạng định mức.

3.3.11 Tìm hiểu thêm về Linear Algebra

Chỉ trong phần này, chúng tôi đã dạy bạn tất cả các đại số tuyến tính mà bạn sẽ cần phải hiểu một phần đáng chú ý của học sâu hiện đại. Có rất nhiều hơn để đại số tuyến tính và rất nhiều toán học đó rất hữu ích cho việc học máy. Ví dụ, ma trận có thể bị phân hủy thành các yếu tố, và những phân hủy này có thể tiết lộ cấu trúc chiều thấp trong các bộ dữ liệu thế giới thực. Có toàn bộ trường con của machine learning tập trung vào việc sử dụng các phân hủy ma trận và khái quát hóa của chúng cho các hàng chục bậc cao để khám phá cấu trúc trong bộ dữ liệu và giải quyết các vấn đề dự đoán. Nhưng cuốn sách này tập trung vào việc học sâu. Và chúng tôi tin rằng bạn sẽ có xu hướng tìm hiểu thêm toán học hơn một khi bạn đã nhận được bàn tay của bạn bẩn triển khai các mô hình máy học hữu ích trên các bộ dữ liệu thực. Vì vậy, trong khi chúng tôi có quyền giới thiệu nhiều toán học hơn nhiều sau này, chúng tôi sẽ kết thúc phần này ở đây.

Nếu bạn muốn tìm hiểu thêm về đại số tuyến tính, bạn có thể tham khảo online appendix on linear algebraic operations[40] hoặc các tài nguyên tuyệt vời khác (Strang, 1993; Kolter, 2008; Petersen et al., 2008).

3.3.12 Tóm tắt

- Vô hướng, vectơ, ma trận, và hàng chục là các đối tượng toán học cơ bản trong đại số tuyến tính.
- Vectơ tổng quát hóa vô hướng, và ma trận tổng quát hóa vectơ.
- Vô hướng, vectơ, ma trận và hàng chục có số không, một, hai và một số trục tùy ý, tương ứng.
- Một tensor có thể được giảm dọc theo các trục được chỉ định bởi `sum` và `mean`.
- Nhân Elementwise của hai ma trận được gọi là sản phẩm Hadamard của họ. Nó khác với phép nhân ma trận.
- Trong học sâu, chúng ta thường làm việc với các tiêu chuẩn như định mức L_1, định mức L_2 và định mức Frobenius.
- Chúng ta có thể thực hiện một loạt các hoạt động trên vô hướng, vectơ, ma trận và hàng chục.

3.3.13 Bài tập

1. Chứng minh rằng sự chuyển vị của một ma trận **A** transpose là **A**:$(\mathbf{A}^\top)^\top = \mathbf{A}$.
2. Cho hai ma trận **A** và **B**, cho thấy tổng các transposes bằng chuyển vị của một tổng: $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$.
3. Cho bất kỳ ma trận vuông **A**, là $\mathbf{A} + \mathbf{A}^\top$ luôn đối xứng? Tại sao?
4. Chúng tôi xác định tensor `X` của hình dạng (2, 3, 4) trong phần này. Sản lượng của `len(X)` là gì?
5. Đối với một tensor `X` có hình dạng tùy ý, liệu `len(X)` luôn tương ứng với chiều dài của một trục nhất định là `X` không? Trục đó là gì?
6. Chạy `A / A.sum(axis=1)` và xem những gì sẽ xảy ra. Bạn có thể phân tích lý do?
7. Khi đi du lịch giữa hai điểm ở Manhattan, khoảng cách mà bạn cần bao gồm về tọa độ, tức là về con đường và đường phố là bao nhiêu? Bạn có thể đi du lịch theo đường chéo?
8. Xem xét một tensor với hình dạng (2, 3, 4). Các hình dạng của đầu ra tổng kết dọc theo trục 0, 1 và 2 là gì?

[40] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/geometry-linear-algebraic-ops.html

9. Nạp một tensor với 3 trục trở lên đến chức năng `linalg.norm` và quan sát đầu ra của nó. Chức năng này tính toán gì cho hàng chục hình dạng tùy ý?

Discussions[41]

3.4 Calculus (Giải tích)

Tìm diện tích của một đa giác vẫn còn bí ẩn cho đến ít nhất 2.500 năm trước, khi người Hy Lạp cổ đại chia một đa giác thành tam giác và tổng hợp các khu vực của chúng. Để tìm khu vực của các hình dạng cong, chẳng hạn như một vòng tròn, người Hy Lạp cổ đại được ghi đa giác trong các hình dạng như vậy. Như thể hiện trong Fig. 3.4.1, một đa giác được ghi với nhiều cạnh có chiều dài bằng nhau tốt hơn gần đúng vòng tròn. Quá trình này còn được gọi là *phương pháp kiệt thúc*.

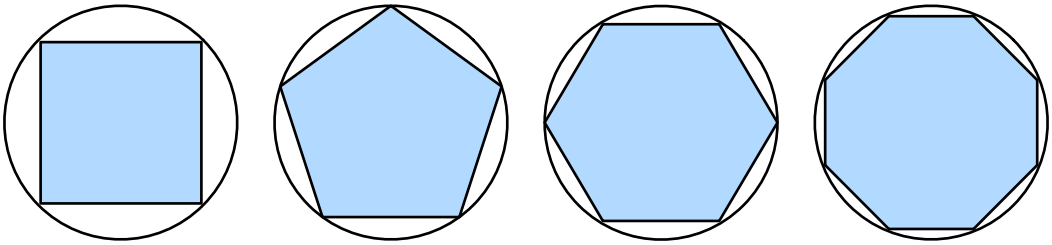


Fig. 3.4.1: Find the area of a circle with the method of exhaustion.

Trên thực tế, phương pháp kiệt sức là nơi * tích phân tích* (sẽ được mô tả trong Section 19.5) bắt nguồn từ. Hơn 2.000 năm sau, nhánh khác của giải tích, *vi phân tích*, đã được phát minh. Trong số các ứng dụng quan trọng nhất của phép tính vi phân, các bài toán tối ưu hóa xem xét cách làm một cái gì đó * tốt nhất*. Như đã thảo luận trong Section 3.3.10, những vấn đề như vậy là phổ biến trong học sâu.

Trong học sâu, chúng tôi * đào tạo* mô hình, cập nhật chúng liên tiếp để chúng trở nên tốt hơn và tốt hơn khi họ thấy ngày càng nhiều dữ liệu. Thông thường, nhận được tốt hơn có nghĩa là giảm thiểu chức năng *mất *, một điểm số trả lời câu hỏi "làm thế nào * xấu* là mô hình của chúng tôi?" Câu hỏi này tinh tế hơn nó xuất hiện. Cuối cùng, những gì chúng tôi thực sự quan tâm là tạo ra một mô hình hoạt động tốt trên dữ liệu mà chúng tôi chưa từng thấy trước đây. Nhưng chúng ta chỉ có thể phù hợp với mô hình để dữ liệu mà chúng ta thực sự có thể thấy. Do đó, chúng ta có thể phân hủy nhiệm vụ lắp các mô hình thành hai mối quan tâm chính: (i) *tối ưu hóa*: quá trình lắp các mô hình của chúng tôi với dữ liệu quan sát; (ii) *khái quát hóa*: các nguyên tắc toán học và trí tuệ của các học viên hướng dẫn về cách tạo ra các mô hình có giá trị vượt quá bộ dữ liệu chính xác examples ví dụ used to train đào tạo them.

Để giúp bạn hiểu các vấn đề và phương pháp tối ưu hóa trong các chương sau, ở đây chúng tôi đưa ra một mồi rất ngắn gọn về phép tính vi phân thường được sử dụng trong deep learning.

[41] https://discuss.d2l.ai/t/31

3.4.1 Các dẫn xuất và sự khác biệt

Chúng tôi bắt đầu bằng cách giải quyết việc tính toán các dẫn xuất, một bước quan trọng trong gần như tất cả các thuật toán tối ưu hóa học tập sâu. Trong deep learning, chúng ta thường chọn các chức năng mất mát có thể khác biệt đối với các thông số của mô hình của chúng tôi. Nói một cách đơn giản, điều này có nghĩa là đối với mỗi tham số, chúng ta có thể xác định mức độ tổn thất sẽ tăng hoặc giảm nhanh như thế nào, chúng tôi * tăng* hoặc * giảm tham số đó bằng một lượng nhỏ vô hạn.

Giả sử rằng chúng ta có một hàm $f : \mathbb{R} \to \mathbb{R}$, có đầu vào và đầu ra là cả vô hướng. *phái sinh* của f được định nghĩa là

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}, \tag{3.4.1}$$

if this limitgiới hạn exists tồn tại. Nếu $f'(a)$ tồn tại, f được cho là *khác biệt* tại a. Nếu f có thể khác biệt ở mọi số của một khoảng thời gian, thì chức năng này có thể phân biệt trong khoảng thời gian này. Chúng ta có thể giải thích đạo hàm $f'(x)$ trong (3.4.1) là tỷ lệ thay đổi tức thường* của $f(x)$ đối với x. Cái gọi là tỷ lệ thay đổi tức thời dựa trên sự thay đổi h trong x, tiếp cận 0.

Để minh họa các dẫn xuất, chúng ta hãy thử nghiệm với một ví dụ. Define $u = f(x) = 3x^2 - 4x$.

```
%matplotlib inline
import numpy as np
from matplotlib_inline import backend_inline
from d2l import torch as d2l


def f(x):
    return 3 * x ** 2 - 4 * x
```

Bằng cách thiết lập $x = 1$ và để h tiếp cận 0, kết quả số của $\frac{f(x+h)-f(x)}{h}$ trong (3.4.1) cách tiếp cận 2. Mặc dù thí nghiệm này không phải là một bằng chứng toán học, chúng ta sẽ thấy sau đó đạo hàm u' là 2 khi 2.

```
def numerical_lim(f, x, h):
    return (f(x + h) - f(x)) / h

h = 0.1
for i in range(5):
    print(f'h={h:.5f}, numerical limit={numerical_lim(f, 1, h):.5f}')
    h *= 0.1
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Chúng ta hãy làm quen với một vài ký hiệu tương đương cho các dẫn xuất. Cho $y = f(x)$, trong đó x và y là biến độc lập và biến phụ thuộc của hàm f, tương ứng. Các biểu thức sau là tương đương:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x), \tag{3.4.2}$$

trong đó các ký hiệu $\frac{d}{dx}$ và D là * toán tử khác biệt* cho biết hoạt động của *sự khác biệt*. Chúng ta có thể sử dụng các quy tắc sau để phân biệt các hàm chung:

- $DC = 0$ (C là một hằng số),
- $Dx^n = nx^{n-1}$ (quy tắc điện*, n là bất kỳ số thực nào),
- $De^x = e^x$,
- $D\ln(x) = 1/x$.

Để phân biệt một hàm được hình thành từ một vài chức năng đơn giản hơn như các chức năng phổ biến trên, các quy tắc sau đây có thể hữu ích cho chúng ta. Giả sử rằng các chức năng f và g đều có thể phân biệt và C là một hằng số, chúng ta có quy tắc nhiều * hằng số*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \tag{3.4.3}$$

quy tắc * sum*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \tag{3.4.4}$$

quy tắc *sản phẩm*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \tag{3.4.5}$$

và quy tắc *thương lượng*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \tag{3.4.6}$$

Bây giờ chúng ta có thể áp dụng một vài trong số các quy tắc trên để tìm $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$. Do đó, bằng cách đặt $x = 1$, chúng tôi có $u' = 2$: điều này được hỗ trợ bởi thí nghiệm trước đó của chúng tôi trong phần này, nơi kết quả số tiếp cận 2. Đạo hàm này cũng là độ dốc của đường tiếp tuyến với đường cong $u = f(x)$ khi $x = 1$.

Để hình dung một cách giải thích các dẫn xuất như vậy, chúng ta sẽ sử dụng `matplotlib`, một thư viện vẽ phổ biến trong Python. Để cấu hình các thuộc tính của các số liệu được tạo ra bởi `matplotlib`, chúng ta cần xác định một vài chức năng. Sau đây, hàm `use_svg_display` chỉ định gói `matplotlib` để xuất các số liệu svg cho hình ảnh sắc nét hơn. Lưu ý rằng nhận xét `#  @save`là một dấu đặc biệt trong đó hàm, lớp hoặc câu lệnh sau được lưu trong gói `d2l` để sau này chúng có thể được gọi trực tiếp (ví dụ, `d2l.use_svg_display()`) mà không được định nghĩa lại.

```
def use_svg_display():  #@save
    """Use the svg format to display a plot in Jupyter."""
    backend_inline.set_matplotlib_formats('svg')
```

Chúng tôi xác định hàm `set_figsize` để chỉ định kích thước hình. Lưu ý rằng ở đây chúng tôi trực tiếp sử dụng `d2l.plt` vì lệnh nhập khẩu `from matplotlib import pyplot as plt` đã được đánh dấu là được lưu trong gói `d2l` trong lời nói đầu.

```
def set_figsize(figsize=(3.5, 2.5)):  #@save
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

Chức năng `set_axes` sau đây đặt các thuộc tính của trục của các con số được tạo ra bởi `matplotlib`.

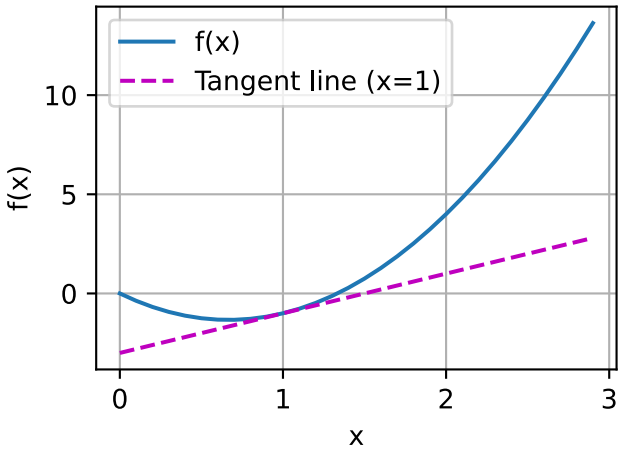
```
#@save
def set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend):
    """Set the axes for matplotlib."""
    axes.set_xlabel(xlabel)
    axes.set_ylabel(ylabel)
    axes.set_xscale(xscale)
    axes.set_yscale(yscale)
    axes.set_xlim(xlim)
    axes.set_ylim(ylim)
    if legend:
        axes.legend(legend)
    axes.grid()
```

Với ba chức năng này cho cấu hình hình, chúng tôi xác định hàm `plot` để vẽ nhiều đường cong một cách ngắn gọn vì chúng ta sẽ cần hình dung nhiều đường cong trong suốt cuốn sách.

```
#@save
def plot(X, Y=None, xlabel=None, ylabel=None, legend=None, xlim=None,
         ylim=None, xscale='linear', yscale='linear',
         fmts=('-', 'm--', 'g-.', 'r:'), figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    if legend is None:
        legend = []

    set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if `X` (tensor or list) has 1 axis
    def has_one_axis(X):
        return (hasattr(X, "ndim") and X.ndim == 1 or isinstance(X, list)
                and not hasattr(X[0], "__len__"))

    if has_one_axis(X):
        X = [X]
    if Y is None:
        X, Y = [[]] * len(X), X
    elif has_one_axis(Y):
        Y = [Y]
    if len(X) != len(Y):
        X = X * len(Y)
    axes.cla()
    for x, y, fmt in zip(X, Y, fmts):
        if len(x):
            axes.plot(x, y, fmt)
        else:
            axes.plot(y, fmt)
    set_axes(axes, xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
```

Bây giờ chúng ta có thể vẽ hàm $u = f(x)$ và đường tiếp tuyến của nó $y = 2x - 3$ tại $x = 1$, trong đó hệ số 2 là độ dốc của đường tiếp tuyến.

```
x = np.arange(0, 3, 0.1)
plot(x, [f(x), 2 * x - 3], 'x', 'f(x)', legend=['f(x)', 'Tangent line (x=1)'])
```



3.4.2 Phái sinh một phần

Cho đến nay chúng ta đã xử lý sự khác biệt của các chức năng của chỉ một biến. Trong deep learning, các hàm thường phụ thuộc vào biến * nhiều *. Do đó, chúng ta cần mở rộng các ý tưởng khác biệt cho các chức năng * đa lộ* này.

Hãy để $y = f(x_1, x_2, \ldots, x_n)$ là một hàm với n biến. Phái sinh một phần* của y đối với tham số i^{th} x_i của nó là

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}. \tag{3.4.7}$$

Để tính toán $\frac{\partial y}{\partial x_i}$, chúng ta chỉ có thể coi $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ là hằng số và tính toán đạo hàm của y đối với x_i. Đối với ký hiệu của các dẫn xuất từng phần, sau đây là tương đương:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \tag{3.4.8}$$

3.4.3 Độ dốc

Chúng ta có thể nối các dẫn xuất từng phần của một hàm đa biến đối với tất cả các biến của nó để có được vectơ *gradient* của hàm. Giả sử rằng đầu vào của hàm $f : \mathbb{R}^n \to \mathbb{R}$ là một vector n chiều $\mathbf{x} = [x_1, x_2, \ldots, x_n]^\top$ và đầu ra là vô hướng. Gradient của hàm $f(\mathbf{x})$ đối với $\mathbf{x}$ là một vectơ của n dẫn xuất một phần:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right]^\top, \tag{3.4.9}$$

trong đó $\nabla_{\mathbf{x}} f(\mathbf{x})$ thường được thay thế bằng $\nabla f(\mathbf{x})$ khi không có sự mơ hồ.

Để $\mathbf{x}$ là một vector n chiều, các quy tắc sau thường được sử dụng khi phân biệt các chức năng đa biến:

- For all $\mathbf{A} \in \mathbb{R}^{m \times n}$, $\nabla_{\mathbf{x}} \mathbf{A}\mathbf{x} = \mathbf{A}^\top$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times m}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A} = \mathbf{A}$,
- For all $\mathbf{A} \in \mathbb{R}^{n \times n}$, $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$,
- $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = \nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{x} = 2\mathbf{x}$.

Tương tự, đối với bất kỳ ma trận $\mathbf{X}$, chúng tôi có $\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$. Như chúng ta sẽ thấy sau, gradient rất hữu ích cho việc thiết kế các thuật toán tối ưu hóa trong deep learning.

3.4.4 Quy tắc chuỗi

Tuy nhiên, độ dốc như vậy có thể khó tìm. Điều này là do các chức năng đa biến trong học sâu thường là * composite*, vì vậy chúng tôi có thể không áp dụng bất kỳ quy tắc nào nói trên để phân biệt các chức năng này. May mắn thay, quy tắc chuỗi * cho phép chúng tôi phân biệt các chức năng tổng hợp.

Trước tiên chúng ta hãy xem xét các chức năng của một biến duy nhất. Giả sử rằng các chức năng $y = f(u)$ và $u = g(x)$ đều có thể khác biệt, thì quy tắc chuỗi nói rằng

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}. \tag{3.4.10}$$

Bây giờ chúng ta hãy chuyển sự chú ý của chúng tôi sang một kịch bản chung hơn trong đó các chức năng có một số tùy ý của biến. Giả sử rằng hàm phân biệt y có các biến $u_1, u_2, \ldots, u_m$, trong đó mỗi hàm phân biệt u_i có các biến $x_1, x_2, \ldots, x_n$. Lưu ý rằng y là một hàm của $x_1, x_2, \ldots, x_n$. Sau đó, quy tắc chuỗi cho

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_i} + \frac{dy}{du_2}\frac{du_2}{dx_i} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_i} \tag{3.4.11}$$

for any $i = 1, 2, \ldots, n$.

3.4.5 Tóm tắt

- Giải tích phân và giải tích phân là hai nhánh của giải tích, nơi mà trước đây có thể được áp dụng cho các bài toán tối ưu hóa phổ biến trong học sâu.
- Một đạo hàm có thể được hiểu là tốc độ thay đổi tức thời của một hàm đối với biến của nó. Nó cũng là độ dốc của đường tiếp tuyến với đường cong của hàm.
- Gradient là một vectơ có thành phần là các dẫn xuất từng phần của một hàm đa biến đối với tất cả các biến của nó.
- Quy tắc chuỗi cho phép chúng ta phân biệt các hàm tổng hợp.

3.4.6 Bài tập

1. Vẽ chức năng $y = f(x) = x^3 - \frac{1}{x}$ và đường tiếp tuyến của nó khi $x = 1$.
2. Tìm gradient của hàm $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$.
3. Gradient của hàm $f(\mathbf{x}) = \|\mathbf{x}\|_2$ là gì?
4. Bạn có thể viết ra quy tắc chuỗi cho trường hợp $u = f(x, y, z)$ và $x = x(a, b)$, $y = y(a, b)$ và $z = z(a, b)$ không?

Discussions[42]

[42] https://discuss.d2l.ai/t/33

3.5 Sự khác biệt tự động

Như chúng tôi đã giải thích trong Section 3.4, sự khác biệt là một bước quan trọng trong gần như tất cả các thuật toán tối ưu hóa học tập sâu. Trong khi các tính toán để lấy các dẫn xuất này rất đơn giản, chỉ đòi hỏi một số phép tính cơ bản, đối với các mô hình phức tạp, làm việc các bản cập nhật bằng tay có thể là một nỗi đau (và thường dễ bị lỗi).

Các khung học sâu đẩy nhanh công việc này bằng cách tự động tính toán các dẫn xuất, tức là *phân biệt tự động*. Trong thực tế, dựa trên mô hình được thiết kế của chúng tôi, hệ thống xây dựng một biểu đồ tính toán *, theo dõi dữ liệu kết hợp thông qua đó các hoạt động để tạo ra đầu ra. Sự khác biệt tự động cho phép hệ thống để sau đó backpropagate gradients. Ở đây, * backpropagate* chỉ đơn giản là có nghĩa là theo dõi thông qua biểu đồ tính toán, điền vào các dẫn xuất từng phần đối với mỗi tham số.

3.5.1 Một ví dụ đơn giản

Như một ví dụ đồ chơi, nói rằng chúng ta quan tâm đến phân biệt chức năng $y = 2\mathbf{x}^\top \mathbf{x}$ đối với vector cột $\mathbf{x}$. Để bắt đầu, chúng ta hãy tạo biến `x` và gán cho nó một giá trị ban đầu.

```
import torch

x = torch.arange(4.0)
x
```

```
tensor([0., 1., 2., 3.])
```

Trước khi chúng tôi thậm chí tính toán gradient của y đối với $\mathbf{x}$, chúng ta sẽ cần một nơi để lưu trữ nó. Điều quan trọng là chúng ta không phân bổ bộ nhớ mới mỗi khi chúng ta lấy một dẫn xuất đối với một tham số vì chúng ta thường sẽ cập nhật các tham số tương tự hàng ngàn hoặc hàng triệu lần và could quickly Nhanh chóng run chạy out of memory bộ nhớ. Lưu ý rằng một gradient của một hàm có giá trị vô hướng đối với một vector $\mathbf{x}$ chính nó có giá trị vectơ và có hình dạng tương tự như $\mathbf{x}$.

```
x.requires_grad_(True)  # Same as `x = torch.arange(4.0, requires_grad=True)`
x.grad  # The default value is None
```

Bây giờ chúng ta hãy tính y.

```
y = 2 * torch.dot(x, x)
y
```

```
tensor(28., grad_fn=<MulBackward0>)
```

Vì `x` là một vectơ có chiều dài 4, một tích chấm của `x` và `x` được thực hiện, mang lại đầu ra vô hướng mà chúng ta gán cho `y`. Tiếp theo, chúng ta có thể tự động tính toán gradient của `y` đối với mỗi thành phần của `x` bằng cách gọi hàm để truyền ngược và in gradient.

```
y.backward()
x.grad
```

```
tensor([ 0.,  4.,  8., 12.])
```

Độ dốc của hàm $y = 2\mathbf{x}^\top \mathbf{x}$ đối với $\mathbf{x}$ nên là $4\mathbf{x}$. Hãy để chúng tôi nhanh chóng xác minh rằng gradient mong muốn của chúng tôi đã được tính chính xác.

```
x.grad == 4 * x
```

```
tensor([True, True, True, True])
```

Bây giờ chúng ta hãy tính một hàm khác của `x`.

```
# PyTorch accumulates the gradient in default, we need to clear the previous
# values
x.grad.zero_()
y = x.sum()
y.backward()
x.grad
```

```
tensor([1., 1., 1., 1.])
```

3.5.2 Lùi cho các biến không vô hướng

Về mặt kỹ thuật, khi `y` không phải là vô hướng, cách giải thích tự nhiên nhất về sự khác biệt của một vectơ `y` đối với một vectơ `x` là một ma trận. Đối với bậc cao hơn và chiều cao hơn `y` và `x`, kết quả khác biệt có thể là một tensor bậc cao.

Tuy nhiên, trong khi những đối tượng kỳ lạ hơn này xuất hiện trong học máy nâng cao (bao gồm trong học sâu), thường xuyên hơn khi chúng ta gọi ngược trên một vectơ, , chúng ta đang cố gắng tính toán các dẫn xuất của các hàm mất mát cho mỗi thành phần của một * lô* ví dụ đào tạo. Ở đây, ý định của chúng tôi là không tính toán ma trận phân biệt mà thay vào đó tổng của các dẫn xuất từng phần được tính riêng cho mỗi ví dụ trong lô.

```
# Invoking `backward` on a non-scalar requires passing in a `gradient`␣
↪argument
# which specifies the gradient of the differentiated function w.r.t `self`.
# In our case, we simply want to sum the partial derivatives, so passing
# in a gradient of ones is appropriate
x.grad.zero_()
y = x * x
# y.backward(torch.ones(len(x))) equivalent to the below
y.sum().backward()
x.grad
```

```
tensor([0., 2., 4., 6.])
```

3.5.3 Tách tính toán

Đôi khi, chúng tôi muốn di chuyển một số tính toán bên ngoài đồ thị tính toán đã ghi lại. Ví dụ, nói rằng `y` được tính như một hàm `x`, và sau đó `z` được tính như một hàm của cả `y` và `x`. Bây giờ, hãy tưởng tượng rằng chúng tôi muốn tính toán gradient của `z` đối với `x`, nhưng muốn vì một lý do nào đó để đối xử với `y` như một hằng số, và chỉ tính đến vai trò mà `x` chơi sau khi `y` được tính toán.

Ở đây, chúng ta có thể tách `y` để trả về một biến mới `u` có cùng giá trị như `y` nhưng loại bỏ bất kỳ thông tin nào về cách `y` được tính toán trong biểu đồ tính toán. Nói cách khác, gradient sẽ không chảy ngược qua `u` đến `x`. Như vậy, hàm truyền ngược sau đây tính toán đạo hàm từng phần của `z = u * x` đối với `x` trong khi xử lý `u` như một hằng số, thay vì đạo hàm từng phần của `z = x * x * x` đối với `x`.

```
x.grad.zero_()
y = x * x
u = y.detach()
z = u * x

z.sum().backward()
x.grad == u
```

```
tensor([True, True, True, True])
```

Kể từ khi tính toán `y` đã được ghi lại, sau đó chúng ta có thể gọi truyền ngược trên `y` để có được dẫn xuất của `y = x * x` đối với `x`, đó là `2 * x`.

```
x.grad.zero_()
y.sum().backward()
x.grad == 2 * x
```

```
tensor([True, True, True, True])
```

3.5.4 Tính toán Gradient của Python Control Flow

Một lợi ích của việc sử dụng sự khác biệt tự động là thậm chí nếu xây dựng biểu đồ tính toán của một hàm yêu cầu đi qua một mê cung của dòng điều khiển Python (ví dụ, điều kiện, vòng lặp và các cuộc gọi hàm tùy ý), chúng ta vẫn có thể tính toán gradient của biến thể kết quả. Trong đoạn sau, lưu ý rằng số lần lặp của vòng lặp `while` và đánh giá câu lệnh `if` cả phụ thuộc vào giá trị của đầu vào `a`.

```
def f(a):
    b = a * 2
    while b.norm() < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

Hãy để chúng tôi tính toán gradient.

```
a = torch.randn(size=(), requires_grad=True)
d = f(a)
d.backward()
```

Bây giờ chúng ta có thể phân tích hàm `f` được xác định ở trên. Lưu ý rằng nó là piecewise tuyến tính trong đầu vào của nó `a`. Nói cách khác, đối với bất kỳ `a` có tồn tại một số vô hướng không đổi `k` sao cho `f(a) = k * a`, trong đó giá trị của `k` phụ thuộc vào `a` đầu vào. Do đó `d / a` cho phép chúng tôi xác minh rằng gradient là chính xác.

```
a.grad == d / a
```

```
tensor(True)
```

3.5.5 Tóm tắt

- Các khuôn khổ học sâu có thể tự động hóa việc tính toán các dẫn xuất. Để sử dụng nó, trước tiên chúng ta đính kèm gradient vào các biến đó đối với mà chúng ta mong muốn các dẫn xuất một phần. Sau đó, chúng tôi ghi lại tính toán giá trị mục tiêu của chúng tôi, thực hiện chức năng của nó để truyền ngược và truy cập gradient kết quả.

3.5.6 Bài tập

1. Tại sao đạo hàm thứ hai đắt hơn nhiều để tính toán so với đạo hàm đầu tiên?
2. Sau khi chạy chức năng để truyền ngược, ngay lập tức chạy lại và xem điều gì sẽ xảy ra.
3. Trong ví dụ dòng điều khiển, nơi chúng ta tính toán đạo hàm của `d` đối với `a`, điều gì sẽ xảy ra nếu chúng ta thay đổi biến `a` thành vectơ hoặc ma trận ngẫu nhiên. Tại thời điểm này, kết quả của phép tính `f(a)` không còn là vô hướng. Điều gì xảy ra với kết quả? Làm thế nào để chúng ta phân tích điều này?
4. Thiết kế lại một ví dụ về việc tìm gradient của luồng điều khiển. Chạy và phân tích kết quả.
5. Hãy để $f(x) = \sin(x)$. Lô $f(x)$ và $\frac{df(x)}{dx}$, nơi sau này được tính toán mà không cần khai thác $f'(x) = \cos(x)$ đó.

Discussions[43]

3.6 Xác suất

Trong một số hình thức này hay hình thức khác, machine learning là tất cả về việc đưa ra dự đoán. Chúng tôi có thể muốn dự đoán * xác suất* của một bệnh nhân bị đau tim trong năm tới, với tiền sử lâm sàng của họ. Trong phát hiện bất thường, chúng ta có thể muốn đánh giá mức độ* tương tự* một tập hợp các bài đọc từ động cơ phản lực của máy bay phản lực sẽ như thế nào, nó hoạt động bình thường. Trong học tập củng cố, chúng tôi muốn một đại lý hành động thông minh trong một môi trường. Điều này có nghĩa là chúng ta cần suy nghĩ về xác suất nhận được phần thưởng cao theo từng hành động có sẵn. Và khi chúng ta xây dựng hệ thống giới thiệu, chúng ta cũng cần phải suy nghĩ về xác suất. Ví dụ: giả thuyết *rằng chúng tôi đã làm việc cho một người bán sách trực tuyến lớn. Chúng tôi có thể muốn ước tính xác suất mà một người dùng cụ thể sẽ

[43] https://discuss.d2l.ai/t/35

mua một cuốn sách cụ thể. Đối với điều này, chúng ta cần sử dụng ngôn ngữ xác suất. Toàn bộ các khóa học, chuyên ngành, luận án, nghề nghiệp và thậm chí cả các bộ phận, được dành cho xác suất. Vì vậy, một cách tự nhiên, mục tiêu của chúng tôi trong phần này là không dạy toàn bộ chủ đề. Thay vào đó, chúng tôi hy vọng sẽ đưa bạn ra khỏi mặt đất, để dạy bạn vừa đủ để bạn có thể bắt đầu xây dựng các mô hình học sâu đầu tiên của mình và cung cấp cho bạn đủ hương vị cho chủ đề mà bạn có thể bắt đầu tự khám phá nó nếu bạn muốn.

Chúng tôi đã gọi xác suất trong các phần trước mà không cần khớp nối chính xác chúng là gì hoặc đưa ra một ví dụ cụ thể. Chúng ta hãy nghiêm túc hơn bây giờ bằng cách xem xét trường hợp đầu tiên: phân biệt mèo và chó dựa trên các bức ảnh. Điều này nghe có vẻ đơn giản nhưng nó thực sự là một thách thức ghê gớm. Để bắt đầu, khó khăn của vấn đề có thể phụ thuộc vào độ phân giải của hình ảnh.

Fig. 3.6.1: Images of varying resolutions (10×10, 20×20, 40×40, 80×80, and 160×160 pixels).

Như thể hiện trong Fig. 3.6.1, trong khi con người dễ dàng nhận ra mèo và chó ở độ phân giải 160×160 pixel, nó trở nên khó khăn ở 40×40 pixel và bên cạnh không thể ở 10×10 pixel. Nói cách khác, khả năng của chúng ta để nói với mèo và chó xa nhau ở một khoảng cách lớn (và do đó độ phân giải thấp) có thể tiếp cận đoán không hiểu biết. Xác suất cho chúng ta một cách lý luận chính thức về mức độ chắc chắn của chúng ta. Nếu chúng ta hoàn toàn chắc chắn rằng hình ảnh mô tả một con mèo, chúng ta nói rằng * xác suất* rằng nhãn tương ứng y là "mèo", ký hiệu $P(y =$ "mèo"$)$ bằng 1. Nếu chúng tôi không có bằng chứng để gợi ý rằng $y =$ "mèo" hoặc $y =$ "chó" đó, thì chúng ta có thể nói rằng hai khả năng là như nhau *likely* thể hiện điều này như $P(y =$ "mèo"$) = P(y =$ "chó"$) = 0.5$. Nếu chúng ta hợp lý tự tin, nhưng không chắc chắn rằng hình ảnh mô tả một con mèo, chúng ta có thể gán một xác suất $0.5 < P(y =$ "mèo"$) < 1$.

Bây giờ hãy xem xét trường hợp thứ hai: đưa ra một số dữ liệu theo dõi thời tiết, chúng tôi muốn dự đoán xác suất nó sẽ mưa ở Đài Bắc vào ngày mai. Nếu đó là mùa hè, mưa có thể đi kèm với xác suất 0,5.

Trong cả hai trường hợp, chúng tôi có một số giá trị quan tâm. Và trong cả hai trường hợp, chúng tôi không chắc chắn về kết quả. Nhưng có một sự khác biệt chính giữa hai trường hợp. Trong trường hợp đầu tiên này, hình ảnh trên thực tế là chó hoặc một con mèo, và chúng tôi chỉ không biết cái nào. Trong trường hợp thứ hai, kết quả thực sự có thể là một sự kiện ngẫu nhiên, nếu bạn tin vào những điều như vậy (và hầu hết các nhà vật lý làm). Vì vậy, xác suất là một ngôn ngữ linh hoạt để lý luận về mức độ chắc chắn của chúng ta, và nó có thể được áp dụng hiệu quả trong một tập hợp rộng lớn các ngữ cảnh.

3.6.1 Lý thuyết xác suất cơ bản

Nói rằng chúng tôi đã chết và muốn biết cơ hội nhìn thấy 1 chứ không phải là một chữ số khác. Nếu chết là công bằng, tất cả sáu kết quả $\{1, \ldots, 6\}$ đều có khả năng xảy ra như nhau, và do đó chúng ta sẽ thấy một 1 trong một trong sáu trường hợp. Chính thức chúng tôi tuyên bố rằng 1 xảy ra với xác suất $\frac{1}{6}$.

Đối với một cái chết thực sự mà chúng tôi nhận được từ một nhà máy, chúng ta có thể không biết những tỷ lệ đó và chúng tôi sẽ cần phải kiểm tra xem nó có bị nhiễm bẩn hay không. Cách duy nhất để điều tra khuôn là đúc nó nhiều lần và ghi lại kết quả. Đối với mỗi diễn viên của khuôn, chúng ta sẽ quan sát một giá trị trong $\{1, \ldots, 6\}$. Với những kết quả này, chúng tôi muốn điều tra xác suất quan sát từng kết quả.

Một cách tiếp cận tự nhiên cho mỗi giá trị là lấy số lượng cá nhân cho giá trị đó và chia nó cho tổng số lần tung. Điều này cho chúng ta một * ước tính* về xác suất của một *sự kiện* nhất định. Quy luật * số lượng lớn* cho chúng ta biết rằng khi số lượng ném tăng lên ước tính này sẽ tiến gần hơn và gần hơn với xác suất cơ bản thực sự. Trước khi đi sâu vào chi tiết về những gì đang xảy ra ở đây, chúng ta hãy thử nó.

Để bắt đầu, chúng ta hãy nhập các gói cần thiết.

```
%matplotlib inline
import torch
from torch.distributions import multinomial
from d2l import torch as d2l
```

Tiếp theo, chúng ta sẽ muốn có thể đúc chết. Trong thống kê, chúng tôi gọi quá trình vẽ ví dụ này từ phân phối xác suất * mẫu*. Phân phối gán xác suất cho một số lựa chọn rời rạc được gọi là *phân phối đa ngôn ngữ*. Chúng tôi sẽ đưa ra một định nghĩa chính thức hơn về *phân phối* sau đó, nhưng ở mức cao, hãy nghĩ về nó như chỉ là một nhiệm vụ probabilities xác suất to events các sự kiện.

Để vẽ một mẫu duy nhất, chúng tôi chỉ cần vượt qua một vector xác suất. Đầu ra là một vectơ khác có cùng độ dài: giá trị của nó tại chỉ số i là số lần kết quả lấy mẫu tương ứng với i.

```
fair_probs = torch.ones([6]) / 6
multinomial.Multinomial(1, fair_probs).sample()
```

```
tensor([0., 0., 0., 0., 1., 0.])
```

Nếu bạn chạy sampler một loạt các lần, bạn sẽ thấy rằng bạn nhận ra các giá trị ngẫu nhiên mỗi lần. Như với việc ước tính tính công bằng của khuôn, chúng ta thường muốn tạo ra nhiều mẫu từ cùng một phân phối. Nó sẽ là không thể chịu nổi chậm để làm điều này với một vòng lặp Python `for`, vì vậy chức năng chúng tôi đang sử dụng hỗ trợ vẽ nhiều mẫu cùng một lúc, trả về một loạt các mẫu độc lập trong bất kỳ hình dạng nào chúng ta có thể mong muốn.

```
multinomial.Multinomial(10, fair_probs).sample()
```

```
tensor([2., 2., 0., 2., 1., 3.])
```

Bây giờ chúng ta đã biết làm thế nào để lấy mẫu cuộn của một khuôn, chúng ta có thể mô phỏng 1000 cuộn. Sau đó chúng ta có thể đi qua và đếm, sau mỗi 1000 cuộn, bao nhiêu lần mỗi số được cuộn. Cụ thể, chúng tôi tính toán tần số tương đối như ước tính xác suất thực sự.

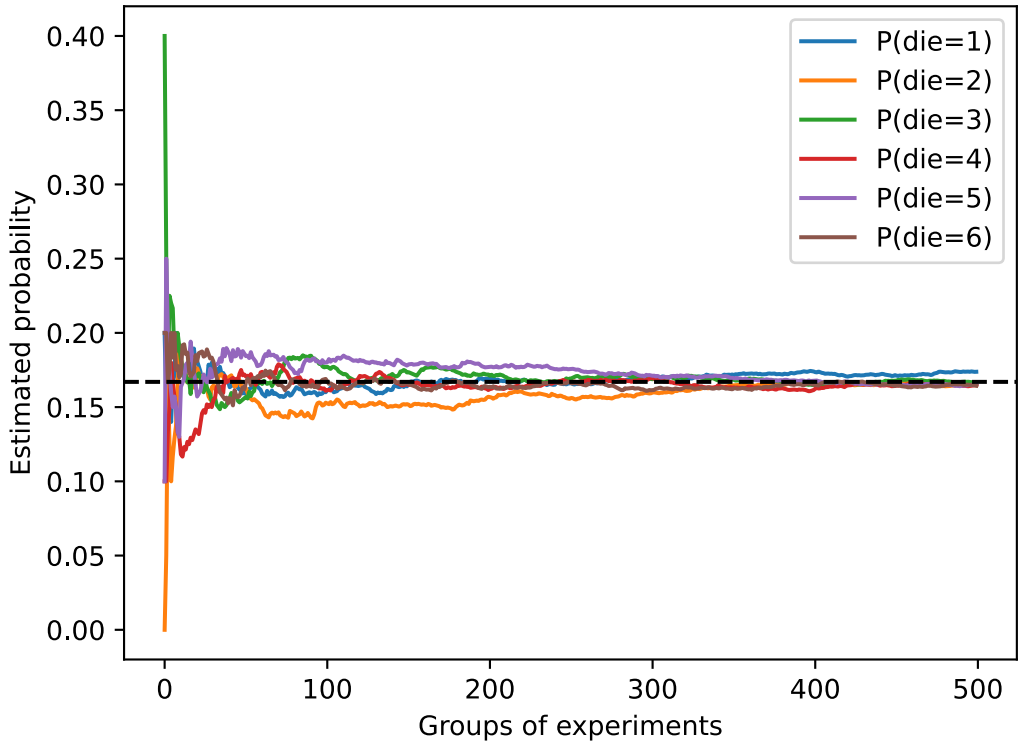
```
# Store the results as 32-bit floats for division
counts = multinomial.Multinomial(1000, fair_probs).sample()
counts / 1000  # Relative frequency as the estimate
```

```
tensor([0.1680, 0.1850, 0.1690, 0.1700, 0.1560, 0.1520])
```

Bởi vì chúng tôi tạo ra dữ liệu từ một chết công bằng, chúng tôi biết rằng mỗi kết quả có xác suất thực sự $\frac{1}{6}$, khoảng 0.167, vì vậy các ước tính đầu ra trên trông tốt.

Chúng ta cũng có thể hình dung cách các xác suất này hội tụ theo thời gian hướng tới xác suất thực sự. Hãy để chúng tôi tiến hành 500 nhóm thí nghiệm trong đó mỗi nhóm vẽ 10 mẫu.

```
counts = multinomial.Multinomial(10, fair_probs).sample((500,))
cum_counts = counts.cumsum(dim=0)
estimates = cum_counts / cum_counts.sum(dim=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].numpy(),
                 label=("P(die=" + str(i + 1) + ")"))
d2l.plt.axhline(y=0.167, color='black', linestyle='dashed')
d2l.plt.gca().set_xlabel('Groups of experiments')
d2l.plt.gca().set_ylabel('Estimated probability')
d2l.plt.legend();
```



Mỗi đường cong rắn tương ứng với một trong sáu giá trị của khuôn và cho xác suất ước tính của chúng ta rằng khuôn biến giá trị đó như được đánh giá sau mỗi nhóm thí nghiệm. Đường đen đứt nét cho xác suất cơ bản thực sự. Khi chúng ta nhận được nhiều dữ liệu hơn bằng cách tiến hành nhiều thí nghiệm hơn, các đường cong rắn 6 hội tụ về xác suất thực sự.

Tiên đề của lý thuyết xác suất

Khi xử lý các cuộn của một chết, chúng tôi gọi bộ $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ là * không gian mẫu* hoặc *không gian kết thúc*, trong đó mỗi phần tử là một *kết thúc*. Một *event* là một tập hợp các kết quả từ một không gian mẫu nhất định. Ví dụ: "nhìn thấy một 5" ($\{5\}$) và "nhìn thấy một số lẻ" ($\{1, 3, 5\}$) đều là những sự kiện hợp lệ của việc lăn chết. Lưu ý rằng nếu kết quả của một thí nghiệm ngẫu nhiên là trong sự kiện $\mathcal{A}$, thì sự kiện $\mathcal{A}$ đã xảy ra. Điều đó có nghĩa là, nếu 3 chấm đối mặt sau khi lăn chết, kể từ $3 \in \{1, 3, 5\}$, chúng ta có thể nói rằng sự kiện "nhìn thấy một số lẻ" đã xảy ra.

Về mặt chính thức, *xác suất* có thể được coi là một hàm ánh xạ một tập hợp thành giá trị thực. Xác suất của một sự kiện $\mathcal{A}$ trong không gian mẫu đã cho $\mathcal{S}$, ký hiệu là $P(\mathcal{A})$, thỏa mãn các thuộc tính sau:

- Đối với bất kỳ sự kiện nào $\mathcal{A}$, xác suất của nó không bao giờ tiêu cực, tức là, $P(\mathcal{A}) \geq 0$;
- Xác suất của toàn bộ không gian mẫu là 1, tức là, $P(\mathcal{S}) = 1$;
- Đối với bất kỳ chuỗi các sự kiện có thể đếm được $\mathcal{A}_1, \mathcal{A}_2, \ldots$ mà là * loại trừ lẫn nhau* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ cho tất cả $i \neq j$), xác suất rằng bất kỳ xảy ra là bằng tổng xác suất cá nhân của họ, tức là, $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$.

Đây cũng là các tiên đề của lý thuyết xác suất, được đề xuất bởi Kolmogorov năm 1933. Nhờ hệ thống tiên đề này, chúng ta có thể tránh được bất kỳ tranh chấp triết học nào về tính ngẫu nhiên; thay vào đó, chúng ta có thể lý luận nghiêm ngặt với một ngôn ngữ toán học. Ví dụ, bằng cách cho phép sự kiện $\mathcal{A}_1$ là toàn bộ không gian mẫu và $\mathcal{A}_i = \emptyset$ cho tất cả $i > 1$, chúng ta có thể chứng minh rằng $P(\emptyset) = 0$, tức là xác suất của một sự kiện bất khả thi là 0.

Biến ngẫu nhiên

Trong thí nghiệm ngẫu nhiên của chúng tôi về đúc chết, chúng tôi đã giới thiệu khái niệm về một biến thể ngẫu nhiên*. Một biến ngẫu nhiên có thể khá nhiều bất kỳ số lượng và không phải là xác định. Nó có thể mất một giá trị trong một tập hợp các khả năng trong một thí nghiệm ngẫu nhiên. Hãy xem xét một biến ngẫu nhiên X có giá trị nằm trong không gian mẫu $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$ lăn một khuôn. Chúng ta có thể biểu thị sự kiện "nhìn thấy một 5" là $\{X = 5\}$ hoặc $X = 5$, và xác suất của nó là $P(\{X = 5\})$ hoặc $P(X = 5)$. Đến $P(X = a)$, chúng ta phân biệt giữa biến ngẫu nhiên X và các giá trị (ví dụ, a) mà X có thể lấy. Tuy nhiên, pedantry như vậy dẫn đến một ký hiệu rườm rà. Đối với một ký hiệu nhỏ gọn, một mặt, chúng ta chỉ có thể biểu thị $P(X)$ là *distribution* so với biến ngẫu nhiên X: phân phối cho chúng ta biết xác suất X có bất kỳ giá trị nào. Mặt khác, chúng ta chỉ cần viết $P(a)$ để biểu thị xác suất một biến ngẫu nhiên lấy giá trị a. Vì một sự kiện trong lý thuyết xác suất là một tập hợp các kết quả từ không gian mẫu, chúng ta có thể chỉ định một phạm vi các giá trị cho một biến ngẫu nhiên cần lấy. Ví dụ, $P(1 \leq X \leq 3)$ biểu thị xác suất của sự kiện $\{1 \leq X \leq 3\}$, có nghĩa là $\{X = 1, 2, \text{or}, 3\}$. Tương đương, $P(1 \leq X \leq 3)$ thể hiện xác suất biến ngẫu nhiên X có thể lấy một giá trị từ $\{1, 2, 3\}$.

Lưu ý rằng có một sự khác biệt tinh tế giữa các biến ngẫu nhiên *discrete*, như các cạnh của một cái chết, và *liên tục* các biến, như trọng lượng và chiều cao của một người. Có rất ít điểm trong việc hỏi liệu hai người có chính xác cùng chiều cao hay không. Nếu chúng ta thực hiện các phép đo đủ chính xác, bạn sẽ thấy rằng không có hai người nào trên hành tinh có cùng chiều cao chính xác. Trên thực tế, nếu chúng ta thực hiện một phép đo đủ tốt, bạn sẽ không có cùng chiều cao khi bạn thức dậy và khi bạn đi ngủ. Vì vậy, không có mục đích nào trong việc hỏi về xác suất mà ai đó là 1.80139278291028719210196740527486202 mét cao. Với dân số thế giới của con người xác suất hầu như là 0. Nó có ý nghĩa hơn trong trường hợp này để hỏi xem chiều cao của ai đó có rơi vào một khoảng thời gian nhất định hay không, nói từ 1,79 đến 1,81 mét. Trong những trường hợp này, chúng tôi định lượng khả năng chúng tôi thấy giá trị là * mật độ*. Chiều cao chính xác 1,80 mét không có xác suất, nhưng mật độ nonzero. Trong khoảng thời gian giữa bất kỳ hai độ cao khác nhau chúng ta có xác

suất nonzero. Trong phần còn lại của phần này, chúng tôi xem xét xác suất trong không gian rời rạc. Để xác suất qua các biến ngẫu nhiên liên tục, bạn có thể tham khảo Section 19.6.

3.6.2 Xử lý nhiều biến ngẫu nhiên

Rất thường xuyên, chúng ta sẽ muốn xem xét nhiều hơn một biến ngẫu nhiên tại một thời điểm. Ví dụ, chúng tôi có thể muốn mô hình hóa mối quan hệ giữa các bệnh và triệu chứng. Với một bệnh và một triệu chứng, nói "cúm" và "ho", hoặc có thể xảy ra hoặc không thể xảy ra ở một bệnh nhân có một số xác suất. Mặc dù chúng tôi hy vọng rằng xác suất của cả hai sẽ gần bằng 0, chúng tôi có thể muốn ước tính những xác suất này và mối quan hệ của chúng với nhau để chúng tôi có thể áp dụng các suy luận của mình để có tác dụng chăm sóc y tế tốt hơn.

Như một ví dụ phức tạp hơn, hình ảnh chứa hàng triệu pixel, do đó hàng triệu biến ngẫu nhiên. Và trong nhiều trường hợp, hình ảnh sẽ đi kèm với một nhãn, xác định các đối tượng trong hình ảnh. Chúng ta cũng có thể nghĩ nhãn như một biến ngẫu nhiên. Chúng ta thậm chí có thể nghĩ về tất cả các siêu dữ liệu như các biến ngẫu nhiên như vị trí, thời gian, khẩu độ, độ dài tiêu cự, ISO, khoảng cách lấy nét và loại máy ảnh. Tất cả những điều này là các biến ngẫu nhiên xảy ra cùng nhau. Khi chúng ta đối phó với nhiều biến ngẫu nhiên, có một số lượng quan tâm.

Xác suất chung

Đầu tiên được gọi là xác suất khớp *:math:`P(A = a, B=b)`. Với bất kỳ giá trị :math:`a` và :math:`b`, xác suất chung cho phép chúng ta trả lời, xác suất :math:`A=a` và :math:`B=b` đồng thời là bao nhiêu? Lưu ý rằng đối với bất kỳ giá trị :math:`a` và :math:`b`, :math:`P(A=a, B=b) leq P(A=a)`. Điều này phải xảy ra, vì đối với :math:`A=a` và :math:`B=b` xảy ra, :math:`A=a` phải xảy ra*và* $B = b$ cũng phải xảy ra (và ngược lại). Do đó, $A = a$ và $B = b$ không thể có nhiều khả năng hơn $A = a$ hoặc $B = b$ riêng lẻ.

Xác suất có điều kiện

Điều này đưa chúng ta đến một tỷ lệ thú vị: $0 \leq \frac{P(A=a,B=b)}{P(A=a)} \leq 1$. Chúng tôi gọi tỷ lệ này là xác suất có điều kiện* và biểu thị nó bằng $P(B = b \mid A = a)$: đó là xác suất $B = b$, với điều kiện là $A = a$ đã xảy ra.

Định lý Bayes'

Sử dụng định nghĩa xác suất có điều kiện, chúng ta có thể lấy được một trong những phương trình hữu ích và nổi tiếng nhất trong thống kê: Định lý *Bayes'. Nó đi như sau. Bằng cách xây dựng, chúng tôi có quy tắc nhân ** rằng $P(A, B) = P(B \mid A)P(A)$. Theo đối xứng, điều này cũng giữ cho $P(A, B) = P(A \mid B)P(B)$. Giả sử rằng $P(B) > 0$. Giải quyết cho một trong các biến có điều kiện mà chúng ta nhận được

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}. \tag{3.6.1}$$

Lưu ý rằng ở đây chúng tôi sử dụng ký hiệu nhỏ gọn hơn trong đó $P(A, B)$ là bản phân phối * chung * và $P(A \mid B)$ là bản phân phối có điều kiện*. Các phân phối như vậy có thể được đánh giá cho các giá trị cụ thể $A = a, B = b$.

Marginalization

Định lý Bayes rất hữu ích nếu chúng ta muốn suy ra một điều từ cái kia, nói nguyên nhân và hiệu quả, nhưng chúng ta chỉ biết các thuộc tính theo hướng ngược lại, như chúng ta sẽ thấy sau trong phần này. Một hoạt động quan trọng mà chúng ta cần, để thực hiện công việc này, là *marginalization*. Đó là hoạt động xác định $P(B)$ từ $P(A, B)$. Chúng ta có thể thấy rằng xác suất B chiếm tất cả các lựa chọn có thể là A và tổng hợp xác suất chung trên tất cả chúng:

$$P(B) = \sum_A P(A, B), \tag{3.6.2}$$

còn được gọi là quy tắc tổng .. Xác suất hoặc phân phối là kết quả của việc biên giới được gọi là xác suất * biên độ* hoặc phân phối biên *.

Độc lập

Một thuộc tính hữu ích khác để kiểm tra là *dependence* so với *independence*. Hai biến ngẫu nhiên A và B độc lập có nghĩa là sự xuất hiện của một sự kiện A không tiết lộ bất kỳ thông tin nào về sự xuất hiện của một sự kiện B. Trong trường hợp này $P(B \mid A) = P(B)$. Các nhà thống kê thường thể hiện điều này là $A \perp B$. Từ định lý Bayes', nó đi theo ngay lập tức đó cũng là $P(A \mid B) = P(A)$. Trong tất cả các trường hợp khác, chúng tôi gọi A và B phụ thuộc. Ví dụ, hai cuộn liên tiếp của một khuôn là độc lập. Ngược lại, vị trí của công tắc ánh sáng và độ sáng trong phòng không phải là (mặc dù chúng không hoàn toàn xác định, vì chúng ta luôn có thể có bóng đèn bị hỏng, mất điện hoặc công tắc bị hỏng).

Vì $P(A \mid B) = \frac{P(A,B)}{P(B)} = P(A)$ tương đương với $P(A, B) = P(A)P(B)$, hai biến ngẫu nhiên độc lập nếu và chỉ khi phân phối chung của chúng là tích của các phân phối riêng lẻ của chúng. Tương tự như vậy, hai biến ngẫu nhiên A và B là * độc lập có điều kiện* cho một biến ngẫu nhiên khác C nếu và chỉ khi $P(A, B \mid C) = P(A \mid C)P(B \mid C)$. Điều này được thể hiện là $A \perp B \mid C$.

Ứng dụng

Hãy để chúng tôi đưa các kỹ năng của chúng tôi để kiểm tra. Giả sử rằng bác sĩ quản lý xét nghiệm HIV cho bệnh nhân. Xét nghiệm này khá chính xác và nó chỉ thất bại với xác suất 1% nếu bệnh nhân khỏe mạnh nhưng báo cáo anh ta là bệnh. Hơn nữa, nó không bao giờ không phát hiện ra HIV nếu bệnh nhân thực sự có nó. Chúng tôi sử dụng D_1 để chỉ ra chẩn đoán (1 nếu dương tính và 0 nếu âm tính) và H để biểu thị tình trạng HIV (1 nếu dương tính và 0 nếu âm tính). Section 3.6.2 liệt kê các xác suất có điều kiện như vậy.

Conditional probability	$H = 1$	$H = 0$
$P(D_1 = 1 \mid H)$	1	0.01
$P(D_1 = 0 \mid H)$	0	0.99

Table: Xác suất điều kiện của $P(D_1 \mid H)$.

Lưu ý rằng các tổng cột là tất cả 1 (nhưng tổng hàng không), vì xác suất có điều kiện cần phải tổng hợp lên đến 1, giống như xác suất. Chúng ta hãy tìm ra xác suất bệnh nhân nhiễm HIV nếu xét nghiệm trở lại dương tính, tức là $P(H = 1 \mid D_1 = 1)$. Rõ ràng điều này sẽ phụ thuộc vào mức độ phổ biến của bệnh, vì nó ảnh hưởng đến số lượng báo động sai. Giả sử rằng dân số khá khỏe mạnh, ví dụ, $P(H = 1) = 0.0015$. Để áp

dụng định lý Bayes', chúng ta cần áp dụng marginalization và quy tắc nhân để xác định

$$\begin{aligned}&P(D_1 = 1)\\ =&P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1)\\ =&P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1)\\ =&0.011485.\end{aligned} \tag{3.6.3}$$

Vì vậy, chúng tôi nhận được

$$\begin{aligned}&P(H = 1 \mid D_1 = 1)\\ =&\frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)}.\\ =&0.1306\end{aligned} \tag{3.6.4}$$

Nói cách khác, chỉ có 13,06% khả năng bệnh nhân thực sự bị nhiễm HIV, mặc dù sử dụng một xét nghiệm rất chính xác. Như chúng ta có thể thấy, xác suất có thể phản trực quan.

Bệnh nhân nên làm gì khi nhận được những tin tức đáng sợ như vậy? Có khả năng, bệnh nhân sẽ yêu cầu bác sĩ quản lý một xét nghiệm khác để có được sự rõ ràng. Thử nghiệm thứ hai có các đặc điểm khác nhau và nó không tốt bằng thử nghiệm đầu tiên, như thể hiện trong Section 3.6.2.

Conditional probability	$H = 1$	$H = 0$
$P(D_2 = 1 \mid H)$	0.98	0.03
$P(D_2 = 0 \mid H)$	0.02	0.97

Table: Xác suất điều kiện của $P(D_2 \mid H)$.

Thật không may, thử nghiệm thứ hai trở lại tích cực, quá. Chúng ta hãy tìm ra các xác suất cần thiết để gọi định lý Bayes' bằng cách giả định độc lập có điều kiện:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 0)\\ =&P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0)\\ =&0.0003,\end{aligned} \tag{3.6.5}$$

$$\begin{aligned}&P(D_1 = 1, D_2 = 1 \mid H = 1)\\ =&P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1)\\ =&0.98.\end{aligned} \tag{3.6.6}$$

Now we can apply marginalization and the multiplication rule:

$$\begin{aligned}&P(D_1 = 1, D_2 = 1)\\ =&P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1)\\ =&P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)\\ =&0.00176955.\end{aligned} \tag{3.6.7}$$

In the end, the probability of the patient having HIV given both positive tests is

$$\begin{aligned}&P(H = 1 \mid D_1 = 1, D_2 = 1)\\ =&\frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)}\\ =&0.8307.\end{aligned} \tag{3.6.8}$$

Đó là, bài kiểm tra thứ hai cho phép chúng tôi đạt được sự tự tin cao hơn nhiều rằng không phải tất cả đều tốt. Mặc dù thử nghiệm thứ hai kém chính xác hơn đáng kể so với thử nghiệm đầu tiên, nhưng nó vẫn cải thiện đáng kể ước tính của chúng tôi.

3.6.3 Kỳ vọng và phương sai

Để tóm tắt các đặc điểm chính của phân phối xác suất, chúng ta cần một số biện pháp. *kỳ vị* (hoặc trung bình) của biến ngẫu nhiên X được ký hiệu là

$$E[X] = \sum_x xP(X = x). \tag{3.6.9}$$

Khi đầu vào của một hàm $f(x)$ là một biến ngẫu nhiên rút ra từ phân phối P với các giá trị khác nhau x, kỳ vọng của $f(x)$ được tính là

$$E_{x\sim P}[f(x)] = \sum_x f(x)P(x). \tag{3.6.10}$$

Trong nhiều trường hợp, chúng tôi muốn đo lường bằng bao nhiêu biến ngẫu nhiên X lệch khỏi kỳ vọng của nó. Điều này có thể được định lượng bởi phương sai

$$\text{Var}[X] = E\left[(X - E[X])^2\right] = E[X^2] - E[X]^2. \tag{3.6.11}$$

Căn bậc hai của nó được gọi là độ lệch tiêu chuẩn*. Phương sai của một hàm số của một phép đo biến ngẫu nhiên bằng bao nhiêu hàm lệch so với kỳ vọng của hàm, vì các giá trị khác nhau x của biến ngẫu nhiên được lấy mẫu từ phân phối của nó:

$$\text{Var}[f(x)] = E\left[\left(f(x) - E[f(x)]\right)^2\right]. \tag{3.6.12}$$

3.6.4 Tóm tắt

- Chúng tôi có thể lấy mẫu từ phân phối xác suất.
- Chúng ta có thể phân tích nhiều biến ngẫu nhiên bằng cách sử dụng phân phối chung, phân phối có điều kiện, định lý Bayes', marginalization, và giả định độc lập.
- Kỳ vọng và phương sai đưa ra các biện pháp hữu ích để tóm tắt các đặc điểm chính của phân phối xác suất.

3.6.5 Bài tập

1. Chúng tôi đã tiến hành $m = 500$ nhóm thí nghiệm trong đó mỗi nhóm rút ra $n = 10$ mẫu. Vary m và n. Quan sát và phân tích kết quả thử nghiệm.
2. Cho hai sự kiện với xác suất $P(\mathcal{A})$ và $P(\mathcal{B})$, tính toán giới hạn trên và dưới trên $P(\mathcal{A} \cup \mathcal{B})$ và $P(\mathcal{A} \cap \mathcal{B})$. (Gợi ý: hiển thị tình hình bằng cách sử dụng Venn Diagram[44].)
3. Giả sử rằng chúng ta có một chuỗi các biến ngẫu nhiên, nói A, B, và C, trong đó B chỉ phụ thuộc vào A, và C chỉ phụ thuộc vào B, bạn có thể đơn giản hóa xác suất chung $P(A, B, C)$? (Gợi ý: đây là một Markov Chain[45].)
4. Năm Section 3.6.2, bài kiểm tra đầu tiên chính xác hơn. Tại sao không chạy thử nghiệm đầu tiên hai lần thay vì chạy cả thử nghiệm thứ nhất và thứ hai?

Discussions[46]

[44] https://en.wikipedia.org/wiki/Venn_diagram
[45] https://en.wikipedia.org/wiki/Markov_chain
[46] https://discuss.d2l.ai/t/37

3.7 Documentation

Do những hạn chế về độ dài của cuốn sách này, chúng ta không thể giới thiệu mọi hàm và lớp PyTorch (và có lẽ bạn sẽ không muốn chúng tôi). Các tài liệu API và các hướng dẫn bổ sung và ví dụ cung cấp nhiều tài liệu ngoài cuốn sách. Trong phần này, chúng tôi cung cấp cho bạn một số hướng dẫn để khám phá API PyTorch.

3.7.1 Tìm tất cả các hàm và lớp học trong một mô-đun

Để biết các hàm và lớp nào có thể được gọi trong một mô-đun, chúng ta gọi hàm `dir`. Ví dụ, chúng ta có thể truy vấn tất cả các thuộc tính trong mô-đun để tạo ra số ngẫu nhiên:

```
import torch

print(dir(torch.distributions))
```

```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial',
↪'CatTransform', 'Categorical', 'Cauchy', 'Chi2', 'ComposeTransform',
↪'ContinuousBernoulli', 'CorrCholeskyTransform', 'Dirichlet', 'Distribution
↪', 'ExpTransform', 'Exponential', 'ExponentialFamily', 'FisherSnedecor',
↪'Gamma', 'Geometric', 'Gumbel', 'HalfCauchy', 'HalfNormal', 'Independent',
↪'IndependentTransform', 'Kumaraswamy', 'LKJCholesky', 'Laplace', 'LogNormal
↪', 'LogisticNormal', 'LowRankMultivariateNormal', 'LowerCholeskyTransform',
↪'MixtureSameFamily', 'Multinomial', 'MultivariateNormal', 'NegativeBinomial
↪', 'Normal', 'OneHotCategorical', 'OneHotCategoricalStraightThrough
↪', 'Pareto', 'Poisson', 'PowerTransform', 'RelaxedBernoulli',
↪'RelaxedOneHotCategorical', 'ReshapeTransform', 'SigmoidTransform',
↪'SoftmaxTransform', 'StackTransform', 'StickBreakingTransform', 'StudentT
↪', 'TanhTransform', 'Transform', 'TransformedDistribution', 'Uniform',
↪'VonMises', 'Weibull', 'Wishart', '__all__', '__builtins__', '__cached__
↪', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__
↪', '__spec__', 'bernoulli', 'beta', 'biject_to', 'binomial', 'categorical
↪', 'cauchy', 'chi2', 'constraint_registry', 'constraints', 'continuous_
↪bernoulli', 'dirichlet', 'distribution', 'exp_family', 'exponential',
↪'fishersnedecor', 'gamma', 'geometric', 'gumbel', 'half_cauchy', 'half_
↪normal', 'identity_transform', 'independent', 'kl', 'kl_divergence',
↪'kumaraswamy', 'laplace', 'lkj_cholesky', 'log_normal', 'logistic_normal
↪', 'lowrank_multivariate_normal', 'mixture_same_family', 'multinomial',
↪'multivariate_normal', 'negative_binomial', 'normal', 'one_hot_categorical
↪', 'pareto', 'poisson', 'register_kl', 'relaxed_bernoulli', 'relaxed_
↪categorical', 'studentT', 'transform_to', 'transformed_distribution',
↪'transforms', 'uniform', 'utils', 'von_mises', 'weibull', 'wishart']
```

Nói chung, chúng ta có thể bỏ qua các hàm bắt đầu và kết thúc bằng __ (các đối tượng đặc biệt trong Python) hoặc các hàm bắt đầu bằng một _ duy nhất (thường là hàm nội bộ). Dựa trên hàm hoặc tên thuộc tính còn lại, chúng ta có thể nguy hiểm đoán rằng mô-đun này cung cấp các phương pháp khác nhau để tạo ra các số ngẫu nhiên, bao gồm lấy mẫu từ phân phối thống nhất (`uniform`), phân phối bình thường (`normal`), và phân phối đa phương thức (`multinomial`).

3.7.2 Tìm cách sử dụng các hàm và lớp cụ thể

Đối với các hướng dẫn cụ thể hơn về cách sử dụng một hàm hoặc lớp nhất định, chúng ta có thể gọi hàm `help`. Ví dụ, chúng ta hãy khám phá các hướng dẫn sử dụng cho hàm `ones` của tensor.

```
help(torch.ones)
```

```
Help on built-in function ones:

ones(...)
    ones(*size, *, out=None, dtype=None, layout=torch.strided,␣
↪device=None, requires_grad=False) -> Tensor

    Returns a tensor filled with the scalar value 1, with the shape␣
↪defined
    by the variable argument size.

    Args:
        size (int...): a sequence of integers defining the shape of␣
↪the output tensor.
            Can be a variable number of arguments or a collection like␣
↪a list or tuple.

    Keyword arguments:
        out (Tensor, optional): the output tensor.
        dtype (torch.dtype, optional): the desired data type of␣
↪returned tensor.
            Default: if None, uses a global default (see torch.set_
↪default_tensor_type()).
        layout (torch.layout, optional): the desired layout of␣
↪returned Tensor.
            Default: torch.strided.
        device (torch.device, optional): the desired device of␣
↪returned tensor.
            Default: if None, uses the current device for the default␣
↪tensor type
            (see torch.set_default_tensor_type()). device will be the␣
↪CPU
            for CPU tensor types and the current CUDA device for CUDA␣
↪tensor types.
        requires_grad (bool, optional): If autograd should record␣
↪operations on the
            returned tensor. Default: False.

    Example::

        >>> torch.ones(2, 3)
        tensor([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])

        >>> torch.ones(5)
```

```
        tensor([ 1.,  1.,  1.,  1.,  1.])
```

Từ tài liệu, chúng ta có thể thấy rằng hàm `ones` tạo ra một tensor mới với hình dạng được chỉ định và đặt tất cả các phần tử thành giá trị của 1. Bất cứ khi nào có thể, bạn nên chạy một bài kiểm tra nhanh để xác nhận giải thích của bạn:

```
torch.ones(4)
```

```
tensor([1., 1., 1., 1.])
```

Trong sổ ghi chép Jupyter, chúng ta có thể sử dụng `?`để hiển thị tài liệu trong một cửa sổ khác. Ví dụ: `list?`sẽ tạo nội dung gần giống với `help(list)`, hiển thị nó trong một cửa sổ trình duyệt mới. Ngoài ra, nếu chúng ta sử dụng hai dấu hỏi, chẳng hạn như `list??`, mã Python thực hiện hàm cũng sẽ được hiển thị.

3.7.3 Tóm tắt

- Tài liệu chính thức cung cấp rất nhiều mô tả và ví dụ vượt ra ngoài cuốn sách này.
- Chúng ta có thể tra cứu tài liệu cho việc sử dụng API bằng cách gọi các hàm `dir` và `help`, hoặc `?` and `??`trong máy tính xách tay Jupyter.

3.7.4 Bài tập

1. Tra cứu tài liệu cho bất kỳ chức năng hoặc lớp học nào trong khuôn khổ học sâu. Bạn cũng có thể tìm thấy tài liệu trên trang web chính thức của khuôn khổ?

Discussions[47]

[47] https://discuss.d2l.ai/t/39

4 | Mạng thần kinh tuyến tính

Trước khi chúng ta đi sâu vào các chi tiết của các mạng thần kinh sâu, chúng ta cần phải đề cập đến những điều cơ bản của đào tạo mạng thần kinh. Trong chương này, chúng tôi sẽ bao gồm toàn bộ quá trình đào tạo, bao gồm xác định kiến trúc mạng thần kinh đơn giản, xử lý dữ liệu, chỉ định chức năng mất mát và đào tạo mô hình. Để làm cho mọi thứ dễ nắm bắt hơn, chúng ta bắt đầu với các khái niệm đơn giản nhất. May mắn thay, các kỹ thuật học thống kê cổ điển như hồi quy tuyến tính và softmax có thể được đúc dưới dạng mạng nơ-ron * tuyến tính*. Bắt đầu từ các thuật toán cổ điển này, chúng tôi sẽ giới thiệu cho bạn những điều cơ bản, cung cấp cơ sở cho các kỹ thuật phức tạp hơn trong phần còn lại của cuốn sách.

4.1 Hồi quy tuyến tính

Hồi quy đề cập đến một tập hợp các phương pháp để mô hình hóa mối quan hệ giữa một hoặc nhiều biến độc lập và một biến phụ thuộc. Trong khoa học tự nhiên và khoa học xã hội, mục đích của hồi quy thường là *characterize* mối quan hệ giữa các đầu vào và đầu ra. Mặt khác, học máy thường liên quan đến * dự đoán *.

Vấn đề hồi quy bật lên bất cứ khi nào chúng ta muốn dự đoán một giá trị số. Các ví dụ phổ biến bao gồm dự đoán giá (nhà cửa, cổ phiếu, v.v.), dự đoán thời gian lưu trú (đối với bệnh nhân trong bệnh viện), dự báo nhu cầu (đối với doanh số bán lẻ), trong số vô số những người khác. Không phải mọi vấn đề dự đoán đều là một vấn đề hồi quy cổ điển. Trong các phần tiếp theo, chúng tôi sẽ giới thiệu các vấn đề phân loại, trong đó mục tiêu là dự đoán tư cách thành viên trong một tập hợp các loại.

4.1.1 Các yếu tố cơ bản của hồi quy tuyến tính

Hồi quy tuyến tính có thể là đơn giản nhất và phổ biến nhất trong số các công cụ tiêu chuẩn để hồi quy. Có niên đại từ bình minh của thế kỷ 19, hồi quy tuyến tính chảy từ một vài giả định đơn giản. Đầu tiên, chúng ta giả định rằng mối quan hệ giữa các biến độc lập $\mathbf{x}$ và biến phụ thuộc y là tuyến tính, tức là y có thể được biểu thị dưới dạng tổng trọng số của các phần tử trong $\mathbf{x}$, cho một số tiếng ồn trên các quan sát. Thứ hai, chúng tôi giả định rằng bất kỳ tiếng ồn nào cũng được cư xử tốt (theo phân phối Gaussian).

Để thúc đẩy cách tiếp cận, chúng ta hãy bắt đầu với một ví dụ đang chạy. Giả sử rằng chúng tôi muốn ước tính giá nhà ở (tính bằng đô la) dựa trên diện tích của họ (tính bằng feet vuông) và tuổi (tính bằng năm). Để thực sự phát triển một mô hình để dự đoán giá nhà, chúng ta sẽ cần phải có được bàn tay của chúng tôi trên một bộ dữ liệu bao gồm doanh số bán hàng mà chúng tôi biết giá bán, diện tích và tuổi cho mỗi ngôi nhà. Trong thuật ngữ học máy, tập dữ liệu được gọi là tập dữ liệu đào tạo* hoặc * bộ đào tạo* và mỗi hàng (ở đây dữ liệu tương ứng với một lần bán) được gọi là *ví dụ* (hoặc điểm dữ liệu, **dữ liệu instance*, *mẫu*). Điều chúng tôi đang cố gắng dự đoán (giá) được gọi là * nhãy* (hoặc *mục tiêu *). Các biến độc lập (tuổi và khu vực) dựa trên dự đoán được gọi là *features* (hoặc *covariates*).

Thông thường, chúng ta sẽ sử dụng n để biểu thị số ví dụ trong tập dữ liệu của chúng ta. Chúng tôi lập chỉ mục các ví dụ dữ liệu bằng i, biểu thị mỗi đầu vào là $\mathbf{x}^{(i)} = [x_1^{(i)}, x_2^{(i)}]^\top$ và nhãn tương ứng là $y^{(i)}$.

Mô hình tuyến tính

Giả định tuyến tính chỉ nói rằng mục tiêu (giá) có thể được biểu thị dưới dạng tổng trọng số của các tính năng (diện tích và tuổi):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{4.1.1}$$

Trong (4.1.1), w_{area} và w_{age} được gọi là *trọng lượng*, và b được gọi là *bias* (còn được gọi là *offset* hoặc *intercept*). Trọng lượng xác định ảnh hưởng của từng tính năng đối với dự đoán của chúng tôi và sự thiên vị chỉ nói giá trị mà giá dự đoán nên mất khi tất cả các tính năng có giá trị 0. Ngay cả khi chúng ta sẽ không bao giờ nhìn thấy bất kỳ ngôi nhà nào có diện tích bằng 0, hoặc chính xác là 0 năm tuổi, chúng ta vẫn cần sự thiên vị nếu không chúng ta sẽ hạn chế tính biểu cảm của mô hình của mình. Nói đúng ra, (4.1.1) là một biến đổi * affine* của các tính năng đầu vào, được đặc trưng bởi sự biến đổi tuyến tính* của các tính năng thông qua tổng trọng số, kết hợp với một * dịch* thông qua sự thiên vị được thêm vào.

Với một tập dữ liệu, mục tiêu của chúng tôi là chọn trọng lượng $\mathbf{w}$ và thiên vị b sao cho trung bình, các dự đoán được thực hiện theo mô hình của chúng tôi phù hợp nhất với giá thực sự quan sát thấy trong dữ liệu. Các mô hình có dự đoán đầu ra được xác định bởi sự biến đổi affine của các tính năng đầu vào là * mô hình tuyến tính*, trong đó chuyển đổi affine được chỉ định bởi các trọng lượng và thiên vị đã chọn.

Trong các ngành mà người ta thường tập trung vào các bộ dữ liệu chỉ với một vài tính năng, thể hiện rõ ràng các mô hình dạng dài như thế này là phổ biến. Trong học máy, chúng ta thường làm việc với các bộ dữ liệu chiều cao, vì vậy thuận tiện hơn khi sử dụng ký hiệu đại số tuyến tính. Khi đầu vào của chúng tôi bao gồm các tính năng d, chúng tôi thể hiện dự đoán của chúng tôi $\hat{y}$ (nói chung là biểu tượng "mũ" biểu thị ước tính) như

$$\hat{y} = w_1 x_1 + ... + w_d x_d + b. \tag{4.1.2}$$

Thu thập tất cả các tính năng thành một vector $\mathbf{x} \in \mathbb{R}^d$ và tất cả các trọng lượng thành một vector $\mathbf{w} \in \mathbb{R}^d$, chúng ta có thể thể hiện mô hình của mình một cách nhỏ gọn bằng cách sử dụng một sản phẩm chấm:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{4.1.3}$$

Trong (4.1.3), vector $\mathbf{x}$ tương ứng với các tính năng của một ví dụ dữ liệu duy nhất. Chúng ta thường sẽ thấy thuận tiện khi tham khảo các tính năng của toàn bộ dữ liệu của chúng tôi n ví dụ thông qua ma trận thiết kế*$\mathbf{X} \in \mathbb{R}^{n \times d}$. Ở đây, $\mathbf{X}$ chứa một hàng cho mỗi ví dụ và một cột cho mọi tính năng.

Đối với một bộ sưu tập các tính năng $\mathbf{X}$, các dự đoán $\hat{\mathbf{y}} \in \mathbb{R}^n$ có thể được thể hiện thông qua sản phẩm ma trục-vector:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{4.1.4}$$

nơi phát sóng (xem Section 3.1.3) được áp dụng trong quá trình tổng kết. Với các tính năng của tập dữ liệu đào tạo $\mathbf{X}$ và các nhãn tương ứng (đã biết) $\mathbf{y}$, mục tiêu của hồi quy tuyến tính là tìm vector trọng lượng $\mathbf{w}$ và thuật ngữ thiên vị b đưa ra các tính năng của một ví dụ dữ liệu mới được lấy mẫu từ cùng một phân phối như $\mathbf{X}$, nhãn của ví dụ mới sẽ (trong kỳ vọng) được dự đoán với lỗi thấp nhất.

Ngay cả khi chúng tôi tin rằng mô hình tốt nhất để dự đoán y cho $\mathbf{x}$ là tuyến tính, chúng tôi sẽ không mong đợi để tìm thấy một bộ dữ liệu thế giới thực của n ví dụ trong đó $y^{(i)}$ chính xác bằng $\mathbf{w}^\top \mathbf{x}^{(i)} + b$ cho tất cả $1 \leq i \leq n$. Ví dụ, bất kỳ công cụ nào chúng tôi sử dụng để quan sát các tính năng $\mathbf{X}$ và nhãn $\mathbf{y}$ có thể bị sai số đo nhỏ. Do đó, ngay cả khi chúng tôi tự tin rằng mối quan hệ cơ bản là tuyến tính, chúng tôi sẽ kết hợp một thuật ngữ tiếng ồn để tính đến các lỗi như vậy.

Trước khi chúng ta có thể tìm kiếm các thông số* tốt nhất* (hoặc tham số *model *) $\mathbf{w}$ và b, chúng ta sẽ cần thêm hai điều nữa: (i) một biện pháp chất lượng cho một số mô hình nhất định; và (ii) một quy trình để cập nhật mô hình để cải thiện chất lượng của nó.

Chức năng mất

Trước khi chúng ta bắt đầu suy nghĩ về cách * phù hợp* dữ liệu với mô hình của chúng tôi, chúng ta cần xác định thước đo* phù hợp. *Chức năng *loss* định lượng khoảng cách giữa giá trị *real* và *dự đoán * của mục tiêu. Sự mất mát thường sẽ là một số không âm trong đó các giá trị nhỏ hơn là dự đoán tốt hơn và hoàn hảo sẽ bị mất 0. Chức năng mất phổ biến nhất trong các bài toán hồi quy là lỗi bình phương. Khi dự đoán của chúng ta cho một ví dụ i là $\hat{y}^{(i)}$ và nhãn đúng tương ứng là $y^{(i)}$, lỗi bình phương được đưa ra bởi:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{4.1.5}$$

Hằng số $\frac{1}{2}$ không tạo ra sự khác biệt thực sự nhưng sẽ chứng minh một cách rõ ràng thuận tiện, hủy bỏ khi chúng ta lấy dẫn xuất của sự mất mát. Kể từ khi tập dữ liệu đào tạo được cung cấp cho chúng tôi, và do đó ngoài tầm kiểm soát của chúng tôi, lỗi thực nghiệm chỉ là một chức năng của các tham số mô hình. Để làm cho mọi thứ cụ thể hơn, hãy xem xét ví dụ dưới đây nơi chúng ta vẽ một bài toán hồi quy cho một trường hợp một chiều như thể hiện trong Fig. 4.1.1.

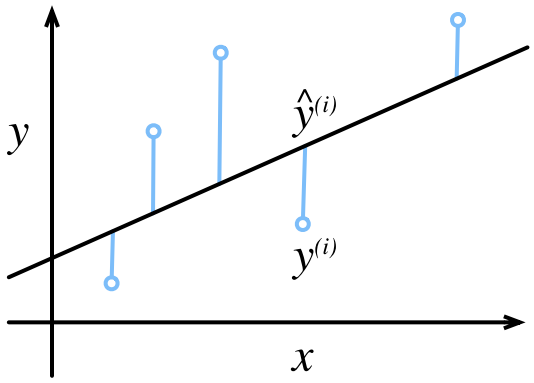


Fig. 4.1.1: Fit data with a linear model.

Lưu ý rằng sự khác biệt lớn giữa các ước tính $\hat{y}^{(i)}$ và quan sát $y^{(i)}$ dẫn đến những đóng góp thậm chí còn lớn hơn cho sự mất mát, do sự phụ thuộc bậc hai. Để đo lường chất lượng của một mô hình trên toàn bộ dữ liệu n ví dụ, chúng tôi chỉ đơn giản là trung bình (hoặc tương đương, tổng hợp) các tổn thất trên bộ đào tạo.

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{4.1.6}$$

Khi đào tạo mô hình, chúng tôi muốn tìm các tham số $(\mathbf{w}^*, b^*)$ giúp giảm thiểu tổng tổn thất trên tất cả các ví dụ đào tạo:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}}\ L(\mathbf{w}, b). \tag{4.1.7}$$

Giải pháp phân tích

Hồi quy tuyến tính xảy ra là một vấn đề tối ưu hóa đơn giản bất thường. Không giống như hầu hết các mô hình khác mà chúng ta sẽ gặp phải trong cuốn sách này, hồi quy tuyến tính có thể được giải quyết một cách phân tích bằng cách áp dụng một công thức đơn giản. Để bắt đầu, chúng ta có thể đặt sai lệch b vào tham số $\mathbf{w}$ bằng cách thêm một cột vào ma trận thiết kế bao gồm tất cả các cột. Sau đó, vấn đề dự đoán của chúng tôi là giảm thiểu $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$. Chỉ có một điểm quan trọng trên bề mặt mất mát và nó tương ứng với mức tối thiểu tổn thất trên toàn bộ miền. Lấy đạo hàm của tổn thất đối với $\mathbf{w}$ và đặt nó bằng 0 mang lại giải pháp phân tích (dạng đóng):

$$\mathbf{w}^* = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}. \tag{4.1.8}$$

Trong khi các vấn đề đơn giản như hồi quy tuyến tính có thể thừa nhận các giải pháp phân tích, bạn không nên quen với may mắn như vậy. Mặc dù các giải pháp phân tích cho phép phân tích toán học tốt đẹp, yêu cầu của một giải pháp phân tích rất hạn chế đến mức nó sẽ loại trừ tất cả các học sâu.

Minibatch Stochastic Gradient Descent

Ngay cả trong trường hợp chúng ta không thể giải quyết các mô hình một cách phân tích, hóa ra chúng ta vẫn có thể đào tạo các mô hình hiệu quả trong thực tế. Hơn nữa, đối với nhiều nhiệm vụ, những mô hình khó tối ưu hóa đó hóa ra trở nên tốt hơn nhiều đến mức tìm ra cách đào tạo chúng sẽ rất đáng để gặp rắc rối.

Kỹ thuật chính để tối ưu hóa gần như bất kỳ mô hình học sâu nào và chúng ta sẽ kêu gọi trong suốt cuốn sách này, bao gồm việc giảm lỗi lặp đi lặp lại bằng cách cập nhật các tham số theo hướng làm giảm chức năng mất dần. Thuật toán này được gọi là *gradient descent*.

Ứng dụng ngây thơ nhất của gradient descent bao gồm lấy đạo hàm của hàm mất, là trung bình của các tổn thất được tính toán trên mỗi ví dụ duy nhất trong tập dữ liệu. Trong thực tế, điều này có thể cực kỳ chậm: chúng ta phải vượt qua toàn bộ tập dữ liệu trước khi thực hiện một bản cập nhật duy nhất. Do đó, chúng ta thường sẽ giải quyết để lấy mẫu một minibatch ngẫu nhiên các ví dụ mỗi khi chúng ta cần tính toán bản cập nhật, một biến thể được gọi là * minibatch stochastic gradient descent*.

Trong mỗi lần lặp lại, lần đầu tiên chúng ta lấy mẫu ngẫu nhiên một minibatch $\mathcal{B}$ bao gồm một số ví dụ đào tạo cố định. Sau đó chúng ta tính toán đạo hàm (gradient) của tổn thất trung bình trên minibatch liên quan đến các tham số mô hình. Cuối cùng, chúng ta nhân gradient với giá trị dương được xác định trước η và trừ thuật ngữ kết quả từ các giá trị tham số hiện tại.

Chúng ta có thể thể hiện bản cập nhật về mặt toán học như sau (∂ biểu thị đạo hàm một phần):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b). \tag{4.1.9}$$

Tóm lại, các bước của thuật toán như sau: (i) chúng ta khởi tạo các giá trị của các tham số mô hình, thường là ngẫu nhiên; (ii) chúng ta lặp lại lấy mẫu minibatches ngẫu nhiên từ dữ liệu, cập nhật các tham số theo hướng của gradient âm. Đối với tổn thất bậc hai và biến đổi affine, chúng ta có thể viết ra điều này một cách rõ ràng như sau:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right). \end{aligned} \tag{4.1.10}$$

Lưu ý rằng $\mathbf{w}$ và $\mathbf{x}$ là vectơ trong (4.1.10). Ở đây, ký hiệu vector thanh lịch hơn làm cho toán học dễ đọc hơn nhiều so với thể hiện mọi thứ về hệ số, nói $w_1, w_2, \ldots, w_d$. Tập cardinality $|\mathcal{B}|$ đại diện cho số lượng ví dụ

trong mỗi minibatch (* batch size*) và :math:`eta` biểu thị tỷ lệ học tập *. Chúng tôi nhấn mạnh rằng các giá trị của quy mô lô và tốc độ học tập được chỉ định trước bằng tay và thường không được học thông qua đào tạo mô hình. Các tham số này có thể điều chỉnh nhưng không được cập nhật trong vòng đào tạo được gọi là *hyperparameters. Điều chỉnh siêu tham số* là quá trình mà các siêu tham số được chọn, và thường yêu cầu chúng tôi điều chỉnh chúng dựa trên kết quả của vòng lặp đào tạo như được đánh giá trên một bộ dữ liệu xác thực * riêng biệt* (hoặc bộ xác thực *).

Sau khi đào tạo cho một số lần lặp lại được xác định trước (hoặc cho đến khi đáp ứng một số tiêu chí dừng khác), chúng tôi ghi lại các tham số mô hình ước tính, ký hiệu là $\hat{\mathbf{w}}, \hat{b}$. Lưu ý rằng ngay cả khi chức năng của chúng ta thực sự tuyến tính và không ồn ào, các tham số này sẽ không phải là bộ giảm thiểu chính xác của sự mất mát bởi vì, mặc dù thuật toán hội tụ chậm về phía các bộ giảm thiểu, nó không thể đạt được chính xác trong một số bước hữu hạn.

Hồi quy tuyến tính xảy ra là một vấn đề học tập mà chỉ có một mức tối thiểu so với toàn bộ miền. Tuy nhiên, đối với các mô hình phức tạp hơn, như mạng sâu, các bề mặt mất mát chứa nhiều minima. May mắn thay, vì những lý do chưa được hiểu đầy đủ, các học viên học sâu hiếm khi đấu tranh để tìm ra các thông số giảm thiểu tổn thất * trong bộ đào tạo*. Nhiệm vụ ghê gớm hơn là tìm các tham số sẽ đạt được tổn thất thấp đối với dữ liệu mà chúng ta chưa thấy trước đây, một thách thức được gọi là *generalization*. Chúng tôi quay trở lại các chủ đề này trong suốt cuốn sách.

Đưa ra dự đoán với mô hình đã học

Với mô hình hồi quy tuyến tính đã học $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$, bây giờ chúng ta có thể ước tính giá của một ngôi nhà mới (không chứa trong dữ liệu đào tạo) cho khu vực của nó x_1 và x_2 tuổi. Ước tính các mục tiêu được đưa ra các tính năng thường được gọi là *prediction* hoặc *inference*.

Chúng tôi sẽ cố gắng gắn bó với *prediction* vì gọi bước này* suy tiế*, mặc dù nổi lên như là thuật ngữ tiêu chuẩn trong học sâu, nhưng có phần là một sự sai lầm. Trong thống kê, *inference* thường biểu thị các tham số ước tính dựa trên tập dữ liệu. Việc sử dụng sai thuật ngữ này là một nguồn gây nhầm lẫn phổ biến khi các học viên học sâu nói chuyện với các nhà thống kê.

4.1.2 Vectorization cho tốc độ

Khi đào tạo các mô hình của chúng tôi, chúng tôi thường muốn xử lý toàn bộ minibatches ví dụ cùng một lúc. Làm điều này một cách hiệu quả đòi hỏi rằng we vectorize các phép tính và tận dụng các thư viện đại số tuyến tính nhanh hơn là viết tốn kém cho - vòng lặp trong Python.

```
%matplotlib inline
import math
import time
import numpy as np
import torch
from d2l import torch as d2l
```

Để minh họa lý do tại sao điều này quan trọng rất nhiều, chúng ta có thể xem xét hai phương pháp để thêm vectors. Để bắt đầu chúng ta khởi tạo hai vectơ 10000 chiều chứa tất cả các vectơ. Trong một phương pháp, chúng ta sẽ lặp lại các vectơ bằng Python for-loop. Trong phương pháp khác, chúng tôi sẽ dựa vào một cuộc gọi duy nhất đến +.

```
n = 10000
a = torch.ones(n)
b = torch.ones(n)
```

Vì chúng ta sẽ chuẩn hóa thời gian chạy thường xuyên trong cuốn sách này, hãy để chúng tôi định nghĩa một timer.

```
class Timer:  #@save
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """Start the timer."""
        self.tik = time.time()

    def stop(self):
        """Stop the timer and record the time in a list."""
        self.times.append(time.time() - self.tik)
        return self.times[-1]

    def avg(self):
        """Return the average time."""
        return sum(self.times) / len(self.times)

    def sum(self):
        """Return the sum of time."""
        return sum(self.times)

    def cumsum(self):
        """Return the accumulated time."""
        return np.array(self.times).cumsum().tolist()
```

Bây giờ chúng ta có thể chuẩn khối lượng công việc. Đầu tiên, chúng tôi thêm chúng, một tọa độ tại một thời điểm, sử dụng một vòng lặp cho.

```
c = torch.zeros(n)
timer = Timer()
for i in range(n):
    c[i] = a[i] + b[i]
f'{timer.stop():.5f} sec'
```

```
'0.08788 sec'
```

Ngoài ra, chúng tôi dựa vào toán tử + nạp lại để tính toán tổng elementwise

```
timer.start()
d = a + b
f'{timer.stop():.5f} sec'
```

```
'0.00025 sec'
```

Bạn có thể nhận thấy rằng phương pháp thứ hai nhanh hơn đáng kể so với phương pháp đầu tiên. Mã vector hóa thường mang lại tốc độ lớn theo thứ tự. Hơn nữa, chúng tôi đẩy nhiều toán học vào thư viện và không cần phải tự viết nhiều tính toán, giảm khả năng xảy ra lỗi.

4.1.3 Phân phối bình thường và tổn thất bình phương

Mặc dù bạn đã có thể làm bẩn tay chỉ bằng cách sử dụng thông tin ở trên, nhưng sau đây chúng ta có thể chính thức thúc đẩy mục tiêu mất bình phương thông qua các giả định về sự phân bố tiếng ồn.

Hồi quy tuyến tính được phát minh bởi Gauss vào năm 1795, người cũng phát hiện ra sự phân bố bình thường (còn gọi là *Gaussian*). Nó chỉ ra rằng kết nối giữa phân bố bình thường và hồi quy tuyến tính chạy sâu hơn cha mẹ chung. Để làm mới bộ nhớ của bạn, mật độ xác suất của một phân phối bình thường với trung bình μ và phương sai σ^2 (độ lệch chuẩn σ) được đưa ra như

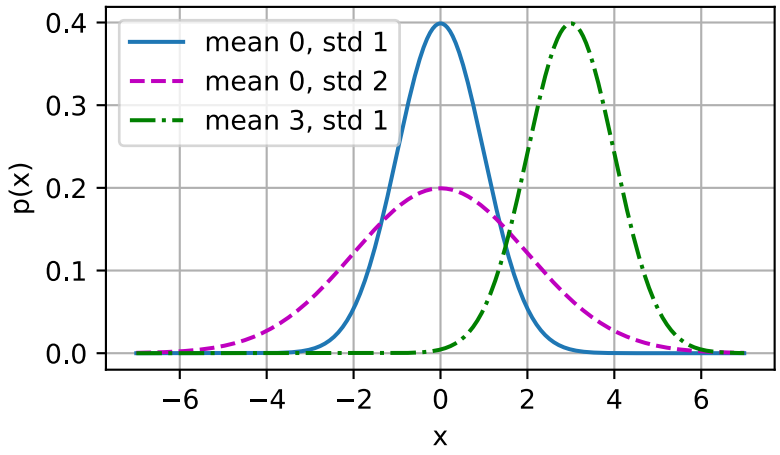
$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right). \tag{4.1.11}$$

Bên dưới chúng tôi định nghĩa một hàm Python để tính toán bản phân phối bình thường.

```
def normal(x, mu, sigma):
    p = 1 / math.sqrt(2 * math.pi * sigma**2)
    return p * np.exp(-0.5 / sigma**2 * (x - mu)**2)
```

Bây giờ chúng ta có thể hình dung các bản phân phối bình thường.

```
# Use numpy again for visualization
x = np.arange(-7, 7, 0.01)

# Mean and standard deviation pairs
params = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in params], xlabel='x',
         ylabel='p(x)', figsize=(4.5, 2.5),
         legend=[f'mean {mu}, std {sigma}' for mu, sigma in params])
```



Như chúng ta có thể thấy, việc thay đổi trung bình tương ứng với sự dịch chuyển dọc theo trục x-và tăng phương sai lan rộng phân phối ra, hạ thấp đỉnh của nó.

Một cách để thúc đẩy hồi quy tuyến tính với hàm mất lỗi bình phương trung bình (hoặc đơn giản là mất bình phương) là chính thức cho rằng các quan sát phát sinh từ các quan sát ồn ào, trong đó tiếng ồn thường được phân phối như sau:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{4.1.12}$$

Do đó, bây giờ chúng ta có thể viết ra * likelihood* của việc nhìn thấy một y cụ thể cho $\mathbf{x}$ qua

$$P(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \tag{4.1.13}$$

Bây giờ, theo nguyên tắc khả năng tối đa, các giá trị tốt nhất của các tham số $\mathbf{w}$ và b là những giá trị tối đa hóa * likelihood* của toàn bộ dữ liệu:

$$P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)}). \tag{4.1.14}$$

Ước tính được lựa chọn theo nguyên tắc khả năng tối đa được gọi là * dự kiến khả năng tối đa*. Trong khi, tối đa hóa sản phẩm của nhiều hàm mũ, có thể trông khó khăn, chúng ta có thể đơn giản hóa mọi thứ đáng kể, mà không thay đổi mục tiêu, bằng cách tối đa hóa nhật ký của khả năng thay thế. Vì lý do lịch sử, tối ưu hóa thường được thể hiện dưới dạng giảm thiểu hơn là tối đa hóa. Vì vậy, mà không thay đổi bất cứ điều gì chúng ta có thể giảm thiểu * âm log-likelihood* $-\log P(\mathbf{y} \mid \mathbf{X})$. Làm việc ra toán học cho chúng ta:

$$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \tag{4.1.15}$$

Bây giờ chúng ta chỉ cần thêm một giả định rằng σ là một số hằng số cố định. Do đó chúng ta có thể bỏ qua thuật ngữ đầu tiên vì nó không phụ thuộc vào $\mathbf{w}$ hoặc b. Bây giờ thuật ngữ thứ hai giống hệt với tổn thất lỗi bình phương được giới thiệu trước đó, ngoại trừ hằng số nhân $\frac{1}{\sigma^2}$. May mắn thay, giải pháp không phụ thuộc vào σ. Theo đó, việc giảm thiểu sai số bình phương trung bình tương đương với ước tính khả năng tối đa của một mô hình tuyến tính theo giả định của tiếng ồn Gaussian phụ gia.

4.1.4 Từ hồi quy tuyến tính đến mạng sâu

Cho đến nay chúng ta chỉ nói về các mô hình tuyến tính. Trong khi các mạng thần kinh bao gồm một họ mô hình phong phú hơn nhiều, chúng ta có thể bắt đầu nghĩ về mô hình tuyến tính như một mạng thần kinh bằng cách thể hiện nó bằng ngôn ngữ của mạng thần kinh. Để bắt đầu, chúng ta hãy bắt đầu bằng cách viết lại mọi thứ trong một ký hiệu "lớp".

Sơ đồ mạng thần kinh

Các học viên học sâu thích vẽ sơ đồ để hình dung những gì đang xảy ra trong mô hình của họ. Trong Fig. 4.1.2, chúng tôi mô tả mô hình hồi quy tuyến tính của chúng tôi như một mạng thần kinh. Lưu ý rằng các sơ đồ này làm nổi bật mô hình kết nối như cách mỗi đầu vào được kết nối với đầu ra, nhưng không phải các giá trị được thực hiện bởi trọng lượng hoặc thành kiến.

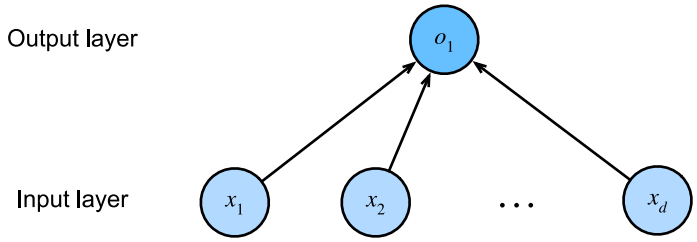


Fig. 4.1.2: Linear regression is a single-layer neural network.

Đối với mạng nơ-ron được hiển thị trong Fig. 4.1.2, các đầu vào là $x_1, \ldots, x_d$, do đó, *số đầu vào* (hoặc *tính năng dimensionality*) trong lớp đầu vào là d. Đầu ra của mạng trong Fig. 4.1.2 là o_1, do đó * số đầu ra* trong lớp đầu ra là 1. Lưu ý rằng các giá trị đầu vào là tất cả * given* và chỉ có một tế bào thần kinh * computed* duy nhất. Tập trung vào nơi tính toán diễn ra, thông thường chúng ta không xem xét lớp đầu vào khi đếm các lớp. Điều đó có nghĩa là, * số lớp* cho mạng thần kinh trong Fig. 4.1.2 là 1. Chúng ta có thể nghĩ về các mô hình hồi quy tuyến tính như các mạng thần kinh chỉ bao gồm một tế bào thần kinh nhân tạo duy nhất, hoặc như các mạng thần kinh một lớp.

Vì đối với hồi quy tuyến tính, mọi đầu vào được kết nối với mọi đầu ra (trong trường hợp này chỉ có một đầu ra), chúng ta có thể coi chuyển đổi này (lớp đầu ra trong Fig. 4.1.2) như một lớp * kết nối đầy đủ* hoặc * lớp dày dài*. Chúng ta sẽ nói nhiều hơn về các mạng bao gồm các lớp như vậy trong chương tiếp theo.

Sinh học

Kể từ khi hồi quy tuyến tính (được phát minh vào năm 1795) có trước khoa học thần kinh tính toán, có vẻ như dị ứng để mô tả hồi quy tuyến tính như một mạng lưới thần kinh. Để xem tại sao các mô hình tuyến tính là một nơi tự nhiên để bắt đầu khi các nhà mạng mạch/nhà sinh lý thần kinh Warren McCulloch và Walter Pitt bắt đầu phát triển các mô hình tế bào thần kinh nhân tạo, hãy xem xét hình ảnh hoạt hình của một tế bào thần kinh sinh học trong Fig. 4.1.3, bao gồm *dendrites* (thiết bị đầu cuối đầu vào), *nucleus* (CPU), * axon* (dây đầu ra) và thiết bị đầu cuối * axon* (thiết bị đầu cuối đầu ra), cho phép kết nối với các tế bào thần kinh khác thông qua * synapses*.

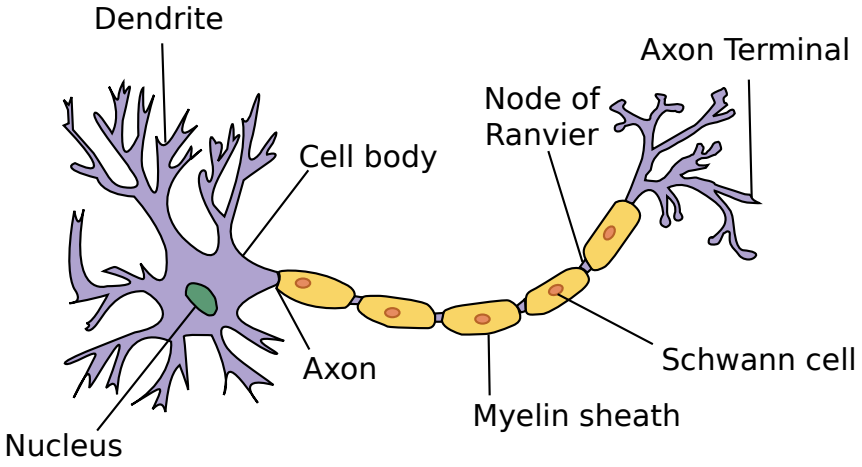


Fig. 4.1.3: The real neuron.

Thông tin x_i đến từ các tế bào thần kinh khác (hoặc cảm biến môi trường như võng mạc) được nhận trong các dendrites. Đặc biệt, thông tin đó được cân bằng * synaptic trọng lượng* w_i xác định ảnh hưởng của các đầu vào (ví dụ: kích hoạt hoặc ức chế thông qua sản phẩm $x_i w_i$). Các đầu vào có trọng số đến từ nhiều nguồn

được tổng hợp trong hạt nhân dưới dạng tổng trọng số $y = \sum_i x_i w_i + b$ và thông tin này sau đó được gửi để xử lý thêm trong axon y, thường là sau một số xử lý phi tuyến thông qua $\sigma(y)$. Từ đó nó hoặc đến đích của nó (ví dụ, một cơ bắp) hoặc được đưa vào một tế bào thần kinh khác thông qua các dendrites của nó.

Chắc chắn, ý tưởng cấp cao rằng nhiều đơn vị như vậy có thể được rải sỏi cùng với kết nối phù hợp và thuật toán học tập đúng đắn, để tạo ra hành vi thú vị và phức tạp hơn nhiều so với bất kỳ tế bào thần kinh nào một mình có thể thể hiện nợ nghiên cứu của chúng tôi về các hệ thống thần kinh sinh học thực sự.

Đồng thời, hầu hết các nghiên cứu trong học sâu ngày nay thu hút rất ít cảm hứng trực tiếp trong khoa học thần kinh. Chúng tôi gọi Stuart Russell và Peter Norvig, người trong cuốn sách giáo khoa AI cổ điển của họ *Artificial Intelligence: A Modern Approach* (Russell & Norvig, 2016), chỉ ra rằng mặc dù máy bay có thể đã được truyền cảm hứng * bởi các loài chim, điểu học đã không phải là động lực chính của đổi mới hàng không trong một số thế kỷ. Tương tự như vậy, cảm hứng trong học sâu những ngày này có thước đo bằng hoặc lớn hơn từ toán học, thống kê và khoa học máy tính.

4.1.5 Tóm tắt

- Các thành phần chính trong mô hình học máy là dữ liệu đào tạo, chức năng mất mát, thuật toán tối ưu hóa và khá rõ ràng là bản thân mô hình.
- Vector hóa làm cho mọi thứ tốt hơn (chủ yếu là toán học) và nhanh hơn (chủ yếu là mã).
- Giảm thiểu một chức năng khách quan và thực hiện ước tính khả năng tối đa có thể có nghĩa là điều tương tự.
- Các mô hình hồi quy tuyến tính cũng là mạng thần kinh.

4.1.6 Bài tập

1. Giả sử rằng chúng ta có một số dữ liệu $x_1, \ldots, x_n \in \mathbb{R}$. Mục tiêu của chúng tôi là tìm một hằng số b sao cho $\sum_i (x_i - b)^2$ được giảm thiểu.
    1. Tìm một giải pháp phân tích cho giá trị tối ưu là b.
    2. Làm thế nào để vấn đề này và giải pháp của nó liên quan đến phân phối bình thường?
2. Lấy được giải pháp phân tích cho bài toán tối ưu hóa cho hồi quy tuyến tính với lỗi bình phương. Để giữ cho mọi thứ đơn giản, bạn có thể bỏ qua sự thiên vị b khỏi vấn đề (chúng ta có thể làm điều này theo kiểu nguyên tắc bằng cách thêm một cột vào $\mathbf{X}$ bao gồm tất cả các cột).
    1. Viết ra bài toán tối ưu hóa trong ký hiệu ma trận và vector (coi tất cả dữ liệu như một ma trận duy nhất, và tất cả các giá trị đích như một vectơ duy nhất).
    2. Tính toán độ dốc của sự mất mát đối với w.
    3. Tìm giải pháp phân tích bằng cách đặt gradient bằng 0 và giải phương trình ma trận.
    4. Khi nào điều này có thể tốt hơn so với sử dụng stochastic gradient descent? Khi nào phương pháp này có thể phá vỡ?
3. Giả sử rằng mô hình tiếng ồn điều chỉnh tiếng ồn phụ gia ϵ là phân phối theo cấp số nhân. Đó là, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
    1. Viết ra khả năng log âm của dữ liệu theo mô hình $-\log P(\mathbf{y} \mid \mathbf{X})$.
    2. Bạn có thể tìm thấy một giải pháp hình thức khép kín?

3. Đề xuất một thuật toán gốc gradient ngẫu nhiên để giải quyết vấn đề này. Điều gì có thể xảy ra sai (gợi ý: điều gì xảy ra gần điểm cố định khi chúng ta tiếp tục cập nhật các tham số)? Bạn có thể sửa chữa điều này?

Discussions[48]

4.2 Thực hiện hồi quy tuyến tính từ đầu

Bây giờ bạn đã hiểu những ý tưởng quan trọng đằng sau hồi quy tuyến tính, chúng ta có thể bắt đầu làm việc thông qua việc thực hiện thực hành trong mã. Trong phần này, chúng tôi sẽ thực hiện toàn bộ phương pháp từ đầu, bao gồm đường ống dữ liệu, mô hình, chức năng mất mát và trình tối ưu hóa giảm dần dần ngẫu nhiên minibatch. Trong khi các khuôn khổ học sâu hiện đại có thể tự động hóa gần như tất cả công việc này, thực hiện mọi thứ từ đầu là cách duy nhất để đảm bảo rằng bạn thực sự biết những gì bạn đang làm. Hơn nữa, khi đến lúc tùy chỉnh các mô hình, xác định các lớp hoặc chức năng mất mát của riêng chúng ta, hiểu cách mọi thứ hoạt động dưới mui xe sẽ chứng minh tiện dụng. Trong phần này, chúng tôi sẽ chỉ dựa vào hàng chục và sự khác biệt tự động. Sau đó, chúng tôi sẽ giới thiệu một triển khai ngắn gọn hơn, tận dụng chuông và còi của các khuôn khổ học sâu.

```
%matplotlib inline
import random
import torch
from d2l import torch as d2l
```

4.2.1 Tạo tập dữ liệu

Để giữ cho mọi thứ đơn giản, chúng ta sẽ xây dựng một tập dữ liệu nhân tạo theo một mô hình tuyến tính với tiếng ồn phụ gia. Nhiệm vụ của chúng ta là khôi phục các tham số của mô hình này bằng cách sử dụng tập hợp các ví dụ hữu hạn có trong tập dữ liệu của chúng ta. Chúng tôi sẽ giữ dữ liệu chiều thấp để chúng tôi có thể hình dung nó một cách dễ dàng. Trong đoạn mã sau, chúng ta tạo ra một tập dữ liệu chứa 1000 ví dụ, mỗi ví dụ gồm 2 tính năng được lấy mẫu từ một phân phối bình thường chuẩn. Do đó tập dữ liệu tổng hợp của chúng tôi sẽ là một ma trận $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$.

Các thông số thực sự tạo ra bộ dữ liệu của chúng tôi sẽ là $\mathbf{w} = [2, -3.4]^\top$ và $b = 4.2$ và nhãn tổng hợp của chúng tôi sẽ được gán theo mô hình tuyến tính sau với thuật ngữ nhiễu ϵ:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \tag{4.2.1}$$

Bạn có thể nghĩ về ϵ là nắm bắt các lỗi đo lường tiềm ẩn trên các tính năng và nhãn. Chúng tôi sẽ giả định rằng các giả định tiêu chuẩn giữ và do đó ϵ tuân theo một phân phối bình thường với trung bình 0. Để làm cho vấn đề của chúng tôi dễ dàng, chúng tôi sẽ đặt độ lệch chuẩn của nó thành 0,01. Mã sau đây tạo ra tập dữ liệu tổng hợp của chúng tôi.

```
def synthetic_data(w, b, num_examples):  #@save
    """Generate y = Xw + b + noise."""
    X = torch.normal(0, 1, (num_examples, len(w)))
    y = torch.matmul(X, w) + b
    y += torch.normal(0, 0.01, y.shape)
    return X, y.reshape((-1, 1))
```

[48] https://discuss.d2l.ai/t/258

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

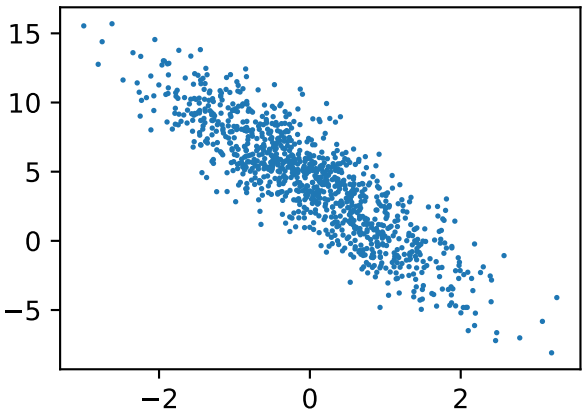
Lưu ý rằng mỗi hàng trong `features` bao gồm một ví dụ dữ liệu 2 chiều và mỗi hàng trong `labels` bao gồm một giá trị nhãn 1 chiều (vô hướng) .

```
print('features:', features[0],'\nlabel:', labels[0])
```

```
features: tensor([-0.6500,  0.6735])
label: tensor([0.6193])
```

Bằng cách tạo ra một âm mưu phân tán bằng cách sử dụng tính năng thứ hai `features[:, 1]` và `labels`, chúng ta có thể quan sát rõ mối tương quan tuyến tính giữa hai.

```
d2l.set_figsize()
# The semicolon is for displaying the plot only
d2l.plt.scatter(features[:, (1)].detach().numpy(), labels.detach().numpy(),␣
↪1);
```



4.2.2 Đọc tập dữ liệu

Nhớ lại rằng các mô hình đào tạo bao gồm thực hiện nhiều lần vượt qua tập dữ liệu, lấy một minibatch ví dụ tại một thời điểm và sử dụng chúng để cập nhật mô hình của chúng tôi. Vì quá trình này rất cơ bản để đào tạo các thuật toán học máy, nên nó đáng để xác định một chức năng tiện ích để xáo trộn bộ dữ liệu và truy cập nó trong các minibatches.

Trong đoạn code sau, chúng ta [**define the ``data_iter`` function**] để chứng minh một thực hiện có thể thực hiện chức năng này. Chức năng có kích thước lô, một ma trận của các tính năng và một vector nhãn, mang lại minibatches kích thước `batch_size`. Mỗi minibatch bao gồm một loạt các tính năng và nhãn.

```
def data_iter(batch_size, features, labels):
    num_examples = len(features)
    indices = list(range(num_examples))
    # The examples are read at random, in no particular order
    random.shuffle(indices)
```

(continues on next page)

(continued from previous page)

```
    for i in range(0, num_examples, batch_size):
        batch_indices = torch.tensor(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Nói chung, lưu ý rằng chúng tôi muốn sử dụng minibatches có kích thước hợp lý để tận dụng lợi thế của phần cứng GPU, vượt trội ở các hoạt động song song. Bởi vì mỗi ví dụ có thể được cung cấp thông qua các mô hình của chúng tôi song song và gradient của hàm mất cho mỗi ví dụ cũng có thể được thực hiện song song, GPU cho phép chúng tôi xử lý hàng trăm ví dụ trong thời gian ít hơn so với việc xử lý chỉ là một ví dụ duy nhất.

Để xây dựng một số trực giác, chúng ta hãy đọc và in hàng loạt ví dụ dữ liệu nhỏ đầu tiên. Hình dạng của các tính năng trong mỗi minibatch cho chúng ta biết cả kích thước minibatch và số lượng tính năng đầu vào. Tương tự như vậy, minibatch nhãn của chúng tôi sẽ có một hình dạng được đưa ra bởi `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

```
tensor([[-0.6999,  1.1784],
        [-0.9279,  1.1579],
        [-0.0039,  0.4110],
        [-0.5786, -2.2632],
        [-0.1232, -0.3288],
        [ 1.5607, -0.2430],
        [ 0.6993, -0.5202],
        [ 0.6211, -0.4483],
        [ 1.1416, -1.1832],
        [-0.6357,  1.1181]])
 tensor([[-1.2099],
        [-1.5779],
        [ 2.8061],
        [10.7431],
        [ 5.0564],
        [ 8.1567],
        [ 7.3705],
        [ 6.9707],
        [10.5014],
        [-0.8699]])
```

Khi chúng ta chạy lặp lại, chúng ta có được các minibatches riêng biệt liên tiếp cho đến khi toàn bộ bộ dữ liệu đã cạn kiệt (hãy thử điều này). Mặc dù việc lặp lại được thực hiện ở trên là tốt cho mục đích giáo khoa, nhưng nó không hiệu quả theo những cách có thể khiến chúng ta gặp rắc rối về các vấn đề thực sự. Ví dụ: nó yêu cầu chúng tôi tải tất cả dữ liệu trong bộ nhớ và chúng tôi thực hiện nhiều truy cập bộ nhớ ngẫu nhiên. Các bộ lặp tích hợp được triển khai trong một khuôn khổ học sâu hiệu quả hơn đáng kể và chúng có thể xử lý cả dữ liệu được lưu trữ trong tệp và dữ liệu được cung cấp thông qua các luồng dữ liệu.

4.2.3 Khởi tạo các tham số mô hình

Trước khi chúng ta có thể bắt đầu tối ưu hóa các tham số của mô hình của mình bằng cách hạ xuống gradient ngẫu nhiên minibatch, chúng ta cần có một số tham số ở vị trí đầu tiên. Trong mã sau, chúng ta khởi tạo trọng lượng bằng cách lấy mẫu các số ngẫu nhiên từ phân phối bình thường với 0 trung bình và độ lệch chuẩn 0,01, và thiết lập sự thiên vị thành 0.

```
w = torch.normal(0, 0.01, size=(2,1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

Sau khi khởi tạo các tham số của chúng tôi, nhiệm vụ tiếp theo của chúng tôi là cập nhật chúng cho đến khi chúng phù hợp với dữ liệu của chúng tôi đủ tốt. Mỗi bản cập nhật yêu cầu lấy gradient của hàm mất của chúng tôi đối với các tham số. Với gradient này, chúng ta có thể cập nhật từng tham số theo hướng có thể làm giảm tổn thất.

Vì không ai muốn tính toán độ dốc một cách rõ ràng (điều này là tẻ nhạt và dễ bị lỗi), chúng tôi sử dụng sự khác biệt tự động, như được giới thiệu trong Section 3.5, để tính toán gradient.

4.2.4 Xác định mô hình

Tiếp theo, chúng ta phải xác định mô hình của chúng tôi, liên quan đến các đầu vào và tham số của nó với đầu racủa nó. Nhớ lại rằng để tính toán đầu ra của mô hình tuyến tính, chúng ta chỉ cần lấy sản phẩm chấm ma thuật-vector của các tính năng đầu vào $\mathbf{X}$ và trọng lượng mô hình $\mathbf{w}$ và thêm b bù vào mỗi ví dụ. Lưu ý rằng dưới $\mathbf{Xw}$ là một vectơ và b là vô hướng. Nhớ lại cơ chế phát sóng như được mô tả trong Section 3.1.3. Khi chúng ta thêm một vectơ và vô hướng, vô hướng được thêm vào mỗi thành phần của vectơ.

```
def linreg(X, w, b):  #@save
    """The linear regression model."""
    return torch.matmul(X, w) + b
```

4.2.5 Xác định chức năng mất

Vì update model của chúng tôi yêu cầu dùng gradient của hàm mất của chúng ta, chúng ta nên define the loss function first. Ở đây chúng ta sẽ sử dụng hàm mất bình phương như mô tả trong Section 4.1. Trong quá trình thực hiện, chúng ta cần chuyển đổi giá trị thực `y` thành hình dạng của giá trị dự đoán `y_hat`. Kết quả được trả về bởi hàm sau cũng sẽ có hình dạng giống như `y_hat`.

```
def squared_loss(y_hat, y):  #@save
    """Squared loss."""
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

4.2.6 Xác định thuật toán tối ưu hóa

Như chúng ta đã thảo luận trong Section 4.1, hồi quy tuyến tính có một giải pháp dạng kín. Tuy nhiên, đây không phải là một cuốn sách về hồi quy tuyến tính: nó là một cuốn sách về học sâu. Vì không có mô hình nào khác mà cuốn sách này giới thiệu có thể được giải quyết một cách phân tích, chúng tôi sẽ nhân cơ hội này để giới thiệu ví dụ làm việc đầu tiên của bạn về dòng dốc ngẫu nhiên minibatch.

Ở mỗi bước, sử dụng một minibatch được vẽ ngẫu nhiên từ tập dữ liệu của chúng tôi, chúng tôi sẽ ước tính độ dốc của sự mất mát đối với các tham số của chúng tôi. Tiếp theo, chúng tôi sẽ cập nhật các thông số của chúng tôi theo hướng có thể làm giảm tổn thất. Mã sau áp dụng bản cập nhật gradient gốc minibatch stochastic, cho một tập hợp các tham số, tốc độ học tập và kích thước lô. Kích thước của bước cập nhật được xác định bởi tỷ lệ học tập `lr`. Bởi vì tổn thất của chúng tôi được tính như một tổng so với các ví dụ nhỏ, chúng tôi bình thường hóa kích thước bước của chúng tôi theo kích thước lô (`batch_size`), do đó độ lớn của một kích thước bước điển hình không phụ thuộc nhiều vào sự lựa chọn của chúng tôi về kích thước lô.

```
def sgd(params, lr, batch_size):  #@save
    """Minibatch stochastic gradient descent."""
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

4.2.7 Đào tạo

Bây giờ chúng tôi đã có tất cả các phần tại chỗ, chúng tôi đã sẵn sàng để thực hiện vòng lặp đào tạo chính Điều quan trọng là bạn phải hiểu mã này bởi vì bạn sẽ thấy các vòng đào tạo gần giống hệt nhau hơn và hơn nữa trong suốt sự nghiệp của bạn trong học sâu.

Trong mỗi lần lặp lại, chúng ta sẽ lấy một loạt các ví dụ đào tạo và chuyển chúng thông qua mô hình của chúng tôi để có được một tập hợp các dự đoán. Sau khi tính toán tổn thất, chúng tôi bắt đầu đi ngược qua mạng, lưu trữ các gradient đối với mỗi tham số. Cuối cùng, chúng ta sẽ gọi thuật toán tối ưu hóa `sgd` để cập nhật các tham số mô hình.

Tóm lại, chúng tôi sẽ thực hiện vòng lặp sau:

- Khởi tạo tham số $(\mathbf{w}, b)$
- Lặp lại cho đến khi xong
  - Tính gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b)$
  - Cập nhật thông số $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Trong mỗi *epoch*, chúng ta sẽ lặp qua toàn bộ tập dữ liệu (sử dụng hàm `data_iter`) một lần đi qua mọi ví dụ trong tập dữ liệu đào tạo (giả sử rằng số ví dụ được chia hết cho kích thước lô). Số epochs `num_epochs` và tỷ lệ học tập `lr` là cả hai siêu tham số, mà chúng tôi đặt ở đây là 3 và 0,03, tương ứng. Thật không may, việc thiết lập các siêu tham số là khó khăn và yêu cầu một số điều chỉnh bằng cách dùng thử và lỗi. Chúng tôi elide những chi tiết này cho bây giờ nhưng sửa đổi chúng sau này trong Chapter 12.

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss
```

```
for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)  # Minibatch loss in `X` and `y`
        # Compute gradient on `l` with respect to [`w`, `b`]
        l.sum().backward()
        sgd([w, b], lr, batch_size)  # Update parameters using their gradient
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

```
epoch 1, loss 0.032702
epoch 2, loss 0.000131
epoch 3, loss 0.000048
```

Trong trường hợp này, bởi vì chúng tôi tự tổng hợp bộ dữ liệu, chúng tôi biết chính xác các tham số thực sự là gì. Do đó, chúng ta có thể đánh giá thành công của mình trong đào tạo bằng cách so sánh các thông số thực sự với những thông số mà chúng tôi đã học thông qua vòng đào tạo của chúng tôi. Thật vậy, họ hóa ra rất gần nhau.

```
print(f'error in estimating w: {true_w - w.reshape(true_w.shape)}')
print(f'error in estimating b: {true_b - b}')
```

```
error in estimating w: tensor([-0.0004, -0.0004], grad_fn=<SubBackward0>)
error in estimating b: tensor([0.0008], grad_fn=<RsubBackward1>)
```

Lưu ý rằng chúng ta không nên coi là điều hiển nhiên rằng chúng ta có thể khôi phục các tham số một cách hoàn hảo. Tuy nhiên, trong machine learning, chúng ta thường ít quan tâm đến việc khôi phục các tham số cơ bản thực sự và quan tâm nhiều hơn đến các tham số dẫn đến dự đoán chính xác cao. May mắn thay, ngay cả trên các vấn đề tối ưu hóa khó khăn, gốc gradient stochastic thường có thể tìm thấy các giải pháp tốt đáng kể, một phần do thực tế là, đối với các mạng sâu, tồn tại nhiều cấu hình của các thông số dẫn đến dự đoán chính xác cao.

4.2.8 Tóm tắt

- Chúng tôi đã thấy cách một mạng sâu có thể được triển khai và tối ưu hóa từ đầu, chỉ sử dụng hàng chục và sự khác biệt tự động, mà không cần xác định các lớp hoặc tối ưu hóa ưa thích.
- Phần này chỉ làm trầy xước bề mặt của những gì có thể. Trong các phần sau, chúng tôi sẽ mô tả các mô hình bổ sung dựa trên các khái niệm mà chúng tôi vừa giới thiệu và tìm hiểu cách thực hiện chúng một cách chính xác hơn.

4.2.9 Bài tập

1. Điều gì sẽ xảy ra nếu chúng ta khởi tạo trọng lượng bằng không. Liệu thuật toán vẫn hoạt động?
2. Giả sử rằng bạn đang Georg Simon Ohm[49] đang cố gắng đưa ra một mô hình giữa điện áp và dòng điện. Bạn có thể sử dụng sự khác biệt tự động để tìm hiểu các thông số của mô hình của bạn?
3. Bạn có thể sử dụng Định luật Planck[50] để xác định nhiệt độ của một vật thể sử dụng mật độ năng lượng quang phổ không?
4. Các vấn đề bạn có thể gặp phải là gì nếu bạn muốn tính toán các dẫn xuất thứ hai? Bạn sẽ sửa chúng như thế nào?
5. Tại sao chức năng `reshape` cần thiết trong hàm `squared_loss`?
6. Thử nghiệm sử dụng các tỷ lệ học tập khác nhau để tìm hiểu giá trị hàm mất giảm nhanh như thế nào.
7. Nếu số lượng ví dụ không thể chia cho kích thước lô, điều gì sẽ xảy ra với hành vi của hàm `data_iter`?

Discussions[51]

4.3 Thực hiện ngắn gọn của hồi quy tuyến tính

Sự quan tâm rộng rãi và mãnh liệt trong việc học sâu trong nhiều năm qua đã truyền cảm hứng cho các công ty, học giả và những người có sở thích phát triển một loạt các khuôn khổ nguồn mở trưởng thành để tự động hóa công việc lặp đi lặp lại của việc thực hiện các thuật toán học tập dựa trên độ dốc. Năm Section 4.2, chúng tôi chỉ dựa vào (i) hàng chục để lưu trữ dữ liệu và đại số tuyến tính; và (ii) tự động phân biệt để tính toán độ dốc. Trong thực tế, bởi vì các bộ lặp dữ liệu, chức năng mất mát, tối ưu hóa và các lớp mạng thần kinh rất phổ biến, các thư viện hiện đại cũng triển khai các thành phần này cho chúng ta.

Trong phần này, chúng tôi sẽ chỉ cho bạn cách thực hiện mô hình hồi quy tuyến tính từ Section 4.2 chính xác bằng cách sử dụng APIs cấp cao của các framework học sâu.

4.3.1 Tạo tập dữ liệu

Để bắt đầu, chúng ta sẽ tạo ra cùng một tập dữ liệu như trong Section 4.2.

```
import numpy as np
import torch
from torch.utils import data
from d2l import torch as d2l
```

```
true_w = torch.tensor([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

[49] https://en.wikipedia.org/wiki/Georg_Ohm
[50] https://en.wikipedia.org/wiki/Planck%27s_law
[51] https://discuss.d2l.ai/t/43

4.3.2 Đọc tập dữ liệu

Thay vì lăn iterator của riêng mình, chúng ta có thể gọi API hiện có trong một framework để đọc dữ liệu Chúng tôi vượt qua `features` và `labels` làm đối số và chỉ định `batch_size` khi khởi tạo một đối tượng lặp dữ liệu. Bên cạnh đó, giá trị boolean `is_train` cho biết liệu chúng ta có muốn đối tượng lặp dữ liệu xáo trộn dữ liệu trên mỗi kỷ nguyên hay không (đi qua bộ dữ liệu).

```
def load_array(data_arrays, batch_size, is_train=True):  #@save
    """Construct a PyTorch data iterator."""
    dataset = data.TensorDataset(*data_arrays)
    return data.DataLoader(dataset, batch_size, shuffle=is_train)
```

```
batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Bây giờ chúng ta có thể sử dụng `data_iter` theo cách tương tự như chúng ta gọi hàm `data_iter` trong Section 4.2. Để xác minh rằng nó đang hoạt động, chúng ta có thể đọc và in minibatch đầu tiên của các ví dụ. So sánh với Section 4.2, ở đây chúng ta sử dụng `iter` để xây dựng một iterator Python và sử dụng `next` để lấy mục đầu tiên từ iterator.

```
next(iter(data_iter))
```

```
[tensor([[-1.0768, -1.1891],
         [-0.8970, -0.1291],
         [ 0.5670, -0.2283],
         [-1.2042, -0.2695],
         [-0.8318,  0.5172],
         [-0.5357,  1.0894],
         [-0.3336,  0.1212],
         [ 0.9926, -0.1793],
         [ 0.2501,  0.5808],
         [ 1.4849,  0.8004]]),
 tensor([[ 6.0941],
         [ 2.8334],
         [ 6.1015],
         [ 2.7120],
         [ 0.7793],
         [-0.5745],
         [ 3.1419],
         [ 6.7983],
         [ 2.7231],
         [ 4.4351]])]
```

4.3.3 Xác định mô hình

Khi chúng tôi thực hiện hồi quy tuyến tính từ đầu vào năm Section 4.2, chúng tôi đã xác định các tham số mô hình của mình một cách rõ ràng và mã hóa các phép tính để tạo ra đầu ra bằng cách sử dụng các phép toán đại số tuyến tính cơ bản. Bạn * nên biết làm thế nào để làm điều này. Nhưng một khi mô hình của bạn trở nên phức tạp hơn, và một khi bạn phải làm điều này gần như mỗi ngày, bạn sẽ rất vui vì được hỗ trợ. Tình hình tương tự như mã hóa blog của riêng bạn từ đầu. Làm điều đó một hoặc hai lần là bổ ích và hướng dẫn, nhưng bạn sẽ là một nhà phát triển web tệ hại nếu mỗi khi bạn cần một blog bạn đã dành một tháng để phát minh lại bánh xe.

Đối với các phép toán chuẩn, chúng ta có thể sử dụng các lớp được xác định trước của framework, cho phép chúng ta tập trung đặc biệt vào các layer dùng để xây dựng mô hình chứ không phải tập trung vào việc triển khai. Trước tiên chúng ta sẽ xác định một biến mô hình `net`, sẽ đề cập đến một phiên bản của lớp `Sequential`. Lớp `Sequential` định nghĩa một container cho nhiều lớp sẽ được xích lại với nhau. Cho dữ liệu đầu vào, một trường hợp `Sequential` truyền nó qua lớp đầu tiên, lần lượt đi qua đầu ra dưới dạng đầu vào của lớp thứ hai và vân vân. Trong ví dụ sau, mô hình của chúng tôi chỉ bao gồm một lớp, vì vậy chúng tôi không thực sự cần `Sequential`. Nhưng vì gần như tất cả các mô hình trong tương lai của chúng tôi sẽ liên quan đến nhiều lớp, dù sao chúng tôi sẽ sử dụng nó chỉ để làm quen với bạn với quy trình làm việc tiêu chuẩn nhất.

Nhớ lại kiến trúc của một mạng một lớp như thể hiện trong Fig. 4.1.2. Lớp được cho là * kết nối đầy đủ* bởi vì mỗi đầu vào của nó được kết nối với mỗi đầu ra của nó bằng phương pháp nhân ma thuật-vector.

Trong PyTorch, lớp kết nối hoàn toàn được định nghĩa trong lớp `Linear`. Lưu ý rằng chúng tôi đã thông qua hai đối số vào `nn.Linear`. Cái đầu tiên xác định kích thước đối tượng đầu vào, đó là 2, và thứ hai là kích thước tính năng đầu ra, đó là một vô hướng duy nhất và do đó 1.

```
# `nn` is an abbreviation for neural networks
from torch import nn

net = nn.Sequential(nn.Linear(2, 1))
```

4.3.4 Khởi tạo các tham số mô hình

Trước khi sử dụng `net`, chúng ta cần khởi tạo các tham số model chẳng hạn như trọng lượng và thiên vị trong mô hình hồi quy tuyến tính. Các khuôn khổ học sâu thường có cách xác định trước để khởi tạo các tham số. Ở đây chúng tôi chỉ định rằng mỗi tham số trọng lượng nên được lấy mẫu ngẫu nhiên từ phân phối bình thường với 0 trung bình và độ lệch chuẩn 0,01. Tham số thiên vị sẽ được khởi tạo thành 0.

Như chúng ta đã chỉ định kích thước đầu vào và đầu ra khi xây dựng `nn.Linear`, bây giờ chúng ta có thể truy cập trực tiếp các tham số để chỉ định các giá trị ban đầu của chúng. Đầu tiên chúng tôi xác định vị trí lớp `net[0]`, là lớp đầu tiên trong mạng, sau đó sử dụng các phương thức `weight.data` và `bias.data` để truy cập các tham số. Tiếp theo chúng ta sử dụng các phương pháp thay thế `normal_` và `fill_` để ghi đè lên các giá trị tham số.

```
net[0].weight.data.normal_(0, 0.01)
net[0].bias.data.fill_(0)
```

```
tensor([0.])
```

4.3.5 Xác định chức năng mất

Lớp `MSELoss` tính toán lỗi bình phương trung bình (không có hệ số $1/2$ trong (4.1.5)) . Theo mặc định, nó trả về mức lỗ trung bình so với các ví dụ.

```
loss = nn.MSELoss()
```

4.3.6 Xác định thuật toán tối ưu hóa

Minibatch stochastic gradient descent là một công cụ tiêu chuẩn để tối ưu hóa các mạng thần kinh và do đó PyTorch hỗ trợ nó cùng với một số biến thể trên thuật toán này trong mô-đun `optim`. Khi chúng ta khởi tạo một phiên bản `SGD` chúng ta sẽ chỉ định các tham số để tối ưu hóa (có thể đạt được từ mạng của chúng tôi thông qua `net.parameters()`), với một từ điển các siêu tham số theo yêu cầu của thuật toán tối ưu hóa của chúng tôi. Minibatch stochastic gradient gốc chỉ yêu cầu chúng ta đặt giá trị `lr`, được đặt thành 0,03 ở đây.

```
trainer = torch.optim.SGD(net.parameters(), lr=0.03)
```

4.3.7 Đào tạo

Bạn có thể nhận thấy rằng thể hiện mô hình của chúng tôi thông qua các API cấp cao của một khuôn khổ học sâu đòi hỏi tương đối ít dòng mã. Chúng tôi không phải phân bổ các thông số riêng lẻ, xác định chức năng mất mát của chúng tôi hoặc thực hiện minibatch stochastic gradient descent. Khi chúng tôi bắt đầu làm việc với các mô hình phức tạp hơn nhiều, lợi thế của API cấp cao sẽ tăng lên đáng kể. Tuy nhiên, một khi chúng tôi có tất cả các phần cơ bản tại chỗ, bản thân vòng đào tạo rất giống với những gì chúng tôi đã làm khi thực hiện mọi thứ từ vết xước.

Để làm mới bộ nhớ của bạn: đối với một số kỷ nguyên, chúng tôi sẽ thực hiện một vượt qua toàn bộ dữ liệu (`train_data`), lặp đi lặp lại lấy một minibatch đầu vào và các nhãn chân lý mặt đất tương ứng. Đối với mỗi minibatch, chúng tôi trải qua các nghi thức sau:

- Tạo ra dự đoán bằng cách gọi `net(X)` và tính toán tổn thất `l` (sự lan truyền về phía trước).
- Tính toán gradient bằng cách chạy backpropagation.
- Cập nhật các tham số mô hình bằng cách gọi trình tối ưu hóa của chúng tôi.

Để có biện pháp tốt, chúng tôi tính toán tổn thất sau mỗi kỷ nguyên và in nó để theo dõi tiến độ.

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        l = loss(net(X) ,y)
        trainer.zero_grad()
        l.backward()
        trainer.step()
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

```
epoch 1, loss 0.000260
epoch 2, loss 0.000097
epoch 3, loss 0.000096
```

Dưới đây, chúng ta so sánh các tham số mô hình học được bằng cách đào tạo về dữ liệu hữu hạn và các tham số thực tế đã tạo ra tập dữ liệu của chúng tôi. Để truy cập các tham số, trước tiên chúng ta truy cập vào lớp mà chúng ta cần từ `net` và sau đó truy cập vào trọng lượng và thiên vị của lớp đó. Như trong triển khai từ đầu của chúng tôi, lưu ý rằng các thông số ước tính của chúng tôi gần với các đối tác thực tế mặt đất của chúng.

```
w = net[0].weight.data
print('error in estimating w:', true_w - w.reshape(true_w.shape))
b = net[0].bias.data
print('error in estimating b:', true_b - b)
```

```
error in estimating w: tensor([0.0007, 0.0007])
error in estimating b: tensor([0.0002])
```

4.3.8 Tóm tắt

- Sử dụng API cấp cao của PyTorch, chúng ta có thể triển khai các mô hình chính xác hơn nhiều.
- Trong PyTorch, mô-đun `data` cung cấp các công cụ để xử lý dữ liệu, mô-đun `nn` định nghĩa một số lượng lớn các lớp mạng thần kinh và các chức năng mất phổ biến.
- Chúng ta có thể khởi tạo các tham số bằng cách thay thế các giá trị của chúng bằng các phương thức kết thúc bằng _.

4.3.9 Bài tập

1. Nếu chúng ta thay thế `nn.MSELoss(reduction='sum')` bằng `nn.MSELoss()`, làm thế nào chúng ta có thể thay đổi tốc độ học tập để mã hoạt động giống hệt nhau. Tại sao?
2. Xem lại tài liệu PyTorch để xem các chức năng mất mát và phương pháp khởi tạo được cung cấp. Thay thế tổn thất bằng mất mát của Huber.
3. Làm thế nào để bạn truy cập gradient của `net[0].weight`?

Discussions[52]

4.4 Hồi quy Softmax

Trong Section 4.1, chúng tôi giới thiệu hồi quy tuyến tính, làm việc thông qua việc triển khai từ đầu vào năm Section 4.2 và một lần nữa sử dụng API cấp cao của một khung học sâu trong Section 4.3 để thực hiện nâng nặng.

Hồi quy là cái búa chúng ta đạt được khi chúng ta muốn trả lời * bao nhiêu? * hoặc *bao nhiêu? * câu hỏi. Nếu bạn muốn dự đoán số đô la (giá cả) mà tại đó một ngôi nhà sẽ được bán, hoặc số chiến thắng một đội

[52] https://discuss.d2l.ai/t/45

bóng chày có thể có, hoặc số ngày mà một bệnh nhân sẽ vẫn nhập viện trước khi được xuất viện, sau đó bạn có thể đang tìm kiếm một mô hình hồi quy.

Trong thực tế, chúng tôi thường quan tâm đến *phân loại *: hỏi không phải "bao nhiêu" mà là "cái nào":

- Email này có thuộc về thư mục spam hay hộp thư đến không?
- Khách hàng này có nhiều khả năng * để đăng ký* hoặc * không đăng ký* cho một dịch vụ đăng ký?
- Hình ảnh này mô tả một con lừa, chó, một con mèo, hoặc một con gà trống?
- Aston có thể xem bộ phim nào nhất tiếp theo?

Thông thường, các học viên học máy làm quá tải từ * phân loại * để mô tả hai vấn đề tinh tế khác nhau: (i) những vấn đề mà chúng tôi chỉ quan tâm đến các bài tập cứng của các ví dụ cho các loại (lớp); và (ii) những vấn đề mà chúng tôi muốn thực hiện các bài tập mềm, tức là, để đánh giá xác suất mà mỗi loại áp dụng. Sự khác biệt có xu hướng bị mờ, một phần, bởi vì thường xuyên, ngay cả khi chúng ta chỉ quan tâm đến các bài tập cứng, chúng ta vẫn sử dụng các mô hình tạo ra các bài tập mềm.

4.4.1 Phân loại vấn đề

Để có được bàn chân của chúng tôi ướt, chúng ta hãy bắt đầu với một vấn đề phân loại hình ảnh đơn giản. Ở đây, mỗi đầu vào bao gồm một hình ảnh thang màu xám 2×2. Chúng ta có thể đại diện cho mỗi giá trị pixel với một vô hướng duy nhất, cho chúng ta bốn tính năng x_1, x_2, x_3, x_4. Hơn nữa, chúng ta hãy giả định rằng mỗi hình ảnh thuộc về một trong số các loại "mèo", "gà" và "chó".

Tiếp theo, chúng ta phải chọn cách đại diện cho các nhãn. Chúng tôi có hai lựa chọn rõ ràng. Có lẽ xung tự nhiên nhất sẽ là chọn $y \in \{1, 2, 3\}$, trong đó các số nguyên đại diện cho $\{\text{dog}, \text{cat}, \text{chicken}\}$ tương ứng. Đây là một cách tuyệt vời để * lưu trữ* thông tin như vậy trên máy tính. Nếu các danh mục có một số thứ tự nhiên trong số đó, hãy nói nếu chúng ta đang cố gắng dự đoán $\{\text{baby}, \text{toddler}, \text{adolescent}, \text{young adult}, \text{adult}, \text{geriatric}\}$, thì thậm chí có thể khiến vấn đề này làm hồi quy và giữ nhãn ở định dạng này.

Nhưng các vấn đề phân loại chung không đi kèm với trật tự tự nhiên giữa các lớp học. May mắn thay, các nhà thống kê từ lâu đã phát minh ra một cách đơn giản để đại diện cho dữ liệu phân loại: mã hóa* một*. Mã hóa một nóng là một vector với nhiều thành phần như chúng ta có các loại. Thành phần tương ứng với thể loại của phiên bản cụ thể được đặt thành 1 và tất cả các thành phần khác được đặt thành 0. Trong trường hợp của chúng tôi, một nhãn y sẽ là một vector ba chiều, với $(1, 0, 0)$ tương ứng với "mèo", $(0, 1, 0)$ đến "gà" và $(0, 0, 1)$ thành "chó":

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \tag{4.4.1}$$

4.4.2 Kiến trúc mạng

Để ước tính xác suất có điều kiện liên quan đến tất cả các lớp có thể, chúng ta cần một mô hình với nhiều đầu ra, một cho mỗi lớp. Để giải quyết phân loại với các mô hình tuyến tính, chúng ta sẽ cần nhiều hàm affine như chúng ta có đầu ra. Mỗi đầu ra sẽ tương ứng với chức năng affine riêng của nó. Trong trường hợp của chúng tôi, vì chúng tôi có 4 tính năng và 3 danh mục đầu ra có thể, chúng tôi sẽ cần 12 vô hướng để đại diện cho trọng lượng (w với các chỉ mục) và 3 vô hướng để đại diện cho các thành kiến (b với chỉ mục). Chúng tôi tính toán ba *logits* này, o_1, o_2, và o_3, cho mỗi đầu vào:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \tag{4.4.2}$$

Chúng ta có thể mô tả tính toán này với sơ đồ mạng thần kinh được hiển thị trong Fig. 4.4.1. Cũng giống như trong hồi quy tuyến tính, hồi quy softmax cũng là một mạng thần kinh một lớp. Và kể từ khi tính toán của mỗi đầu ra, o_1, o_2 và o_3, phụ thuộc vào tất cả các đầu vào, x_1, x_2, x_3 và x_4, lớp đầu ra của hồi quy softmax cũng có thể được mô tả là lớp kết nối hoàn toàn.

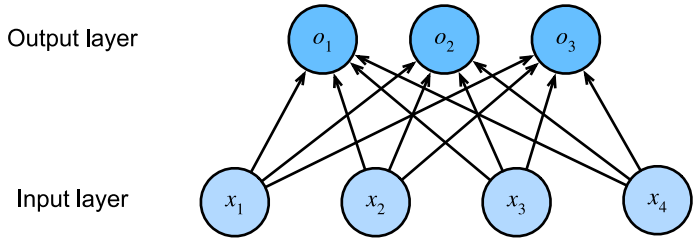


Fig. 4.4.1: Softmax regression is a single-layer neural network.

Để thể hiện mô hình nhỏ gọn hơn, chúng ta có thể sử dụng ký hiệu đại số tuyến tính. Ở dạng vector, chúng tôi đến $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, một hình thức phù hợp hơn cho cả toán học và viết mã. Lưu ý rằng chúng tôi đã thu thập tất cả các trọng lượng của chúng tôi thành ma trận 3×4 và đối với các tính năng của một ví dụ dữ liệu nhất định $\mathbf{x}$, đầu ra của chúng tôi được đưa ra bởi một sản phẩm ma thuật-vector có trọng lượng của chúng tôi bởi các tính năng đầu vào của chúng tôi cộng với thiên vị của chúng tôi $\mathbf{b}$.

4.4.3 Chi phí tham số của các lớp được kết nối hoàn toàn

Như chúng ta sẽ thấy trong các chương tiếp theo, các lớp được kết nối hoàn toàn có mặt khắp nơi trong học sâu. Tuy nhiên, như tên cho thấy, các lớp được kết nối hoàn toàn là * đầy đủ* kết nối với nhiều tham số có khả năng học được. Cụ thể, đối với bất kỳ lớp kết nối hoàn toàn nào với đầu vào d và đầu ra q, chi phí tham số hóa là $\mathcal{O}(dq)$, có thể rất cao trong thực tế. May mắn thay, chi phí chuyển đổi đầu vào d này thành đầu ra q có thể được giảm xuống còn $\mathcal{O}(\frac{dq}{n})$, trong đó siêu tham số n có thể được chúng tôi xác định linh hoạt để cân bằng giữa tiết kiệm thông số và hiệu quả mô hình trong các ứng dụng thế giới thực [Zhang.Tay.Zhang.ea.2021].

4.4.4 Softmax hoạt động

Cách tiếp cận chính mà chúng ta sẽ thực hiện ở đây là giải thích các kết quả đầu ra của mô hình của chúng tôi là xác suất. Chúng tôi sẽ tối ưu hóa các thông số của mình để tạo ra xác suất tối đa hóa khả năng của dữ liệu quan sát. Sau đó, để tạo ra dự đoán, chúng tôi sẽ đặt một ngưỡng, ví dụ, chọn nhãn với xác suất dự đoán tối đa.

Đặt chính thức, chúng tôi muốn bất kỳ đầu ra $\hat{y}_j$ được hiểu là xác suất mà một mục nhất định thuộc về lớp j. Sau đó, chúng ta có thể chọn lớp có giá trị đầu ra lớn nhất như dự đoán của chúng ta $\operatorname{argmax}_j y_j$. Ví dụ: nếu $\hat{y}_1$, $\hat{y}_2$ và $\hat{y}_3$ lần lượt là 0,1, 0,8 và 0,1, thì chúng tôi dự đoán loại 2, trong đó (trong ví dụ của chúng tôi) đại diện cho "gà".

Bạn có thể bị cám dỗ để đề nghị rằng chúng tôi giải thích các bản ghi o trực tiếp như đầu ra của chúng tôi quan tâm. Tuy nhiên, có một số vấn đề với việc giải thích trực tiếp đầu ra của lớp tuyến tính là một xác suất. Một mặt, không có gì hạn chế những con số này để tổng cộng với 1. Mặt khác, tùy thuộc vào đầu vào, chúng có thể lấy giá trị âm. Những vi phạm tiên đề cơ bản của xác suất được trình bày trong Section 3.6

Để hiểu kết quả đầu ra của chúng tôi là xác suất, chúng tôi phải đảm bảo rằng (ngay cả trên dữ liệu mới), chúng sẽ không âm và tổng hợp lên đến 1. Hơn nữa, chúng ta cần một mục tiêu đào tạo khuyến khích mô hình ước tính xác suất trung thực. Trong tất cả các trường hợp khi một phân loại đầu ra 0.5, chúng tôi hy vọng rằng một nửa trong số các ví dụ đó sẽ thực sự thuộc về lớp dự đoán. Đây là một thuộc tính gọi là *calibration*.

Chức năng *softmax*, được phát minh vào năm 1959 bởi nhà khoa học xã hội R.Duncan Luce trong bối cảnh mô hình *lựa chọn*, thực hiện chính xác điều này. Để chuyển đổi nhật ký của chúng tôi sao cho chúng trở nên không âm và tổng thành 1, trong khi yêu cầu mô hình vẫn có thể khác biệt, trước tiên chúng ta cấp mũ mỗi logit (đảm bảo không tiêu cực) và sau đó chia cho tổng của chúng (đảm bảo rằng chúng tổng thành 1):

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}. \tag{4.4.3}$$

Thật dễ dàng để xem $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ với $0 \leq \hat{y}_j \leq 1$ cho tất cả j. Do đó, $\hat{\mathbf{y}}$ là một phân phối xác suất thích hợp mà các giá trị phần tử có thể được diễn giải cho phù hợp. Lưu ý rằng thao tác softmax không thay đổi thứ tự trong số các bản ghi $\mathbf{o}$, đơn giản là các giá trị pre-softmax xác định xác suất được gán cho mỗi lớp. Do đó, trong quá trình dự đoán, chúng ta vẫn có thể chọn ra lớp có khả năng nhất bằng

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j. \tag{4.4.4}$$

Mặc dù softmax là một hàm phi tuyến, các đầu ra của hồi quy softmax vẫn là * quyết định* bởi một biến đổi affine của các tính năng đầu vào; do đó, hồi quy softmax là một mô hình tuyến tính.

4.4.5 Vectorization cho Minibatches

Để cải thiện hiệu quả tính toán và tận dụng lợi thế của GPU, chúng tôi thường thực hiện các phép tính vector cho các minibatches dữ liệu. Giả sử rằng chúng ta được đưa ra một minibatch $\mathbf{X}$ ví dụ với kích thước tính năng (số lượng đầu vào) d và kích thước lô n. Hơn nữa, giả sử rằng chúng tôi có q danh mục trong đầu ra. Sau đó, các tính năng minibatch $\mathbf{X}$ là trong $\mathbb{R}^{n \times d}$, trọng lượng $\mathbf{W} \in \mathbb{R}^{d \times q}$ và sự thiên vị đáp ứng $\mathbf{b} \in \mathbb{R}^{1 \times q}$.

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}). \end{aligned} \tag{4.4.5}$$

Điều này tăng tốc hoạt động thống trị thành một sản phẩm ma trận ma trận $\mathbf{XW}$ so với các sản phẩm ma trận vector mà chúng tôi sẽ thực hiện nếu chúng tôi xử lý một ví dụ tại một thời điểm. Vì mỗi hàng trong $\mathbf{X}$ đại diện cho một ví dụ dữ liệu, nên bản thân thao tác softmax có thể được tính toán * rowwise*: cho mỗi hàng $\mathbf{O}$, cấp số mũ tất cả các mục nhập và sau đó bình thường hóa chúng bằng tổng. Kích hoạt phát sóng trong tổng kết $\mathbf{XW} + \mathbf{b}$ trong (4.4.5), cả hai minibatch đăng nhập $\mathbf{O}$ và xác suất đầu ra $\hat{\mathbf{Y}}$ là $n \times q$ ma trận.

4.4.6 Chức năng mất

Tiếp theo, chúng ta cần một chức năng mất mát để đo lường chất lượng xác suất dự đoán của chúng tôi. Chúng ta sẽ dựa vào ước tính khả năng tối đa, khái niệm rất giống nhau mà chúng ta gặp phải khi cung cấp một biện minh xác suất cho mục tiêu sai số bình phương trung bình trong hồi quy tuyến tính (Section 4.1.3).

Log-Likelihood

Hàm softmax cung cấp cho chúng ta một vector $\hat{\mathbf{y}}$, mà chúng ta có thể giải thích như xác suất có điều kiện ước tính của mỗi lớp cho bất kỳ đầu vào $\mathbf{x}$, ví dụ, $\hat{y}_1$ = $P(y = \text{cat} \mid \mathbf{x})$. Giả sử rằng toàn bộ dữ liệu $\{\mathbf{X}, \mathbf{Y}\}$ có n ví dụ, trong đó ví dụ được lập chỉ mục bởi i bao gồm một vector tính năng $\mathbf{x}^{(i)}$ và một vector nhãn một nóng $\mathbf{y}^{(i)}$. Chúng ta có thể so sánh các ước tính với thực tế bằng cách kiểm tra xem các lớp thực tế có thể xảy ra như thế nào theo mô hình của chúng tôi, với các tính năng:

$$P(\mathbf{Y} \mid \mathbf{X}) = \prod_{i=1}^{n} P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}). \tag{4.4.6}$$

Theo ước tính khả năng tối đa, chúng tôi tối đa hóa $P(\mathbf{Y} \mid \mathbf{X})$, tương đương với việc giảm thiểu khả năng log âm:

$$-\log P(\mathbf{Y} \mid \mathbf{X}) = \sum_{i=1}^{n} -\log P(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{4.4.7}$$

nơi đối với bất kỳ cặp nhãn $\mathbf{y}$ và dự đoán mô hình $\hat{\mathbf{y}}$ trên các lớp q, hàm mất l là

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j. \tag{4.4.8}$$

Vì lý do được giải thích sau này, hàm mất trong (4.4.8) thường được gọi là mất *cross-entropy*. Kể từ $\mathbf{y}$ là một vectơ một nóng có chiều dài q, tổng trên tất cả các tọa độ của nó j biến mất cho tất cả trừ một thuật ngữ. Vì tất cả $\hat{y}_j$ được dự đoán xác suất, logarit của chúng không bao giờ lớn hơn 0. Do đó, hàm mất không thể được giảm thiểu thêm nữa nếu chúng ta dự đoán chính xác nhãn thực tế với *chắc chắn*, tức là, nếu xác suất dự đoán $P(\mathbf{y} \mid \mathbf{x}) = 1$ cho nhãn thực tế $\mathbf{y}$. Lưu ý rằng điều này thường là không thể. Ví dụ: có thể có nhiễu nhãn trong tập dữ liệu (một số ví dụ có thể bị dán nhãn sai). Cũng có thể không thể thực hiện được khi các tính năng đầu vào không đủ thông tin để phân loại mọi ví dụ một cách hoàn hảo.

Softmax và các dẫn xuất

Vì softmax và tổn thất tương ứng rất phổ biến, nên hiểu rõ hơn một chút về cách nó được tính toán. Cắm (4.4.3) vào định nghĩa về sự mất mát trong (4.4.8) và sử dụng định nghĩa của softmax chúng ta có được:

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j. \end{aligned} \tag{4.4.9}$$

Để hiểu rõ hơn một chút những gì đang xảy ra, hãy xem xét phái sinh liên quan đến bất kỳ logit o_j nào. Chúng tôi nhận được

$$\partial_{o_j} l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j. \tag{4.4.10}$$

Nói cách khác, đạo hàm là sự khác biệt giữa xác suất được gán bởi mô hình của chúng ta, như thể hiện bằng phép toán softmax, và những gì thực sự đã xảy ra, như thể hiện bằng các phần tử trong vectơ nhãn một nóng. Theo nghĩa này, nó rất giống với những gì chúng ta đã thấy trong hồi quy, trong đó gradient là sự khác biệt giữa quan sát y và ước tính $\hat{y}$. Đây không phải là sự trùng hợp ngẫu nhiên. Trong bất kỳ họ cấp số nhân nào (xem mô hình online appendix on distributions[53]), độ dốc của khả năng log được đưa ra bởi chính xác thuật ngữ này. Thực tế này làm cho gradient tính toán dễ dàng trong thực tế.

[53] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/distributions.html

Mất Cross-Entropy

Bây giờ hãy xem xét trường hợp mà chúng ta quan sát không chỉ là một kết quả duy nhất mà còn là toàn bộ phân phối trên kết quả. Chúng ta có thể sử dụng đại diện tương tự như trước cho nhãn $\mathbf{y}$. Sự khác biệt duy nhất là thay vì một vector chỉ chứa các mục nhị phân, giả sử $(0, 0, 1)$, bây giờ chúng ta có một vector xác suất chung, nói $(0.1, 0.2, 0.7)$. Toán học mà chúng tôi đã sử dụng trước đây để xác định mất l trong (4.4.8) vẫn hoạt động tốt, chỉ là việc giải thích là tổng quát hơn một chút. Đó là giá trị dự kiến của sự mất mát cho một phân phối trên nhãn. Tổn thất này được gọi là mất *cross-entropy* và nó là một trong những tổn thất được sử dụng phổ biến nhất cho các vấn đề phân loại. Chúng ta có thể demystify tên bằng cách giới thiệu chỉ những điều cơ bản của lý thuyết thông tin. Nếu bạn muốn hiểu thêm chi tiết về lý thuyết thông tin, bạn có thể tham khảo thêm online appendix on information theory[54].

4.4.7 Thông tin cơ bản về lý thuyết

Lý lịch thông tin đề cập đến vấn đề mã hóa, giải mã, truyền tải, và thao tác thông tin (còn được gọi là dữ liệu) ở dạng ngắn gọn nhất có thể.

Entropy

Ý tưởng trung tâm trong lý thuyết thông tin là định lượng nội dung thông tin trong dữ liệu. Số lượng này đặt một giới hạn cứng về khả năng nén dữ liệu của chúng tôi. Trong lý thuyết thông tin, đại lượng này được gọi là *entropy* của một phân phối P, và nó được bắt bởi phương trình sau:

$$H[P] = \sum_j -P(j) \log P(j). \tag{4.4.11}$$

Một trong những định lý cơ bản của lý thuyết thông tin nói rằng để mã hóa dữ liệu được rút ra ngẫu nhiên từ phân phối P, chúng ta cần ít nhất $H[P]$ "net" để mã hóa nó. Nếu bạn tự hỏi một "nat" là gì, nó tương đương với bit nhưng khi sử dụng một mã với base e chứ không phải là một với base 2. Như vậy, một nat là $\frac{1}{\log(2)} \approx 1.44$ bit.

Surprisal

Bạn có thể tự hỏi nén có liên quan gì đến dự đoán. Hãy tưởng tượng rằng chúng ta có một luồng dữ liệu mà chúng ta muốn nén. Nếu chúng ta luôn dễ dàng dự đoán token tiếp theo, thì dữ liệu này rất dễ nén! Lấy ví dụ cực đoan trong đó mọi token trong luồng luôn có cùng giá trị. Đó là một luồng dữ liệu rất nhàm chán! Và không chỉ nó là nhàm chán, nhưng nó cũng dễ dàng để dự đoán. Bởi vì chúng luôn giống nhau, chúng tôi không phải truyền tải bất kỳ thông tin nào để truyền đạt nội dung của luồng. Dễ dự đoán, dễ nén.

Tuy nhiên, nếu chúng ta không thể dự đoán hoàn hảo mọi sự kiện, thì đôi khi chúng ta có thể ngạc nhiên. Sự ngạc nhiên của chúng tôi là lớn hơn khi chúng tôi chỉ định một sự kiện xác suất thấp hơn. Claude Shannon định cư vào $\log \frac{1}{P(j)} = -\log P(j)$ để định lượng của một người *ngạc nhiên* khi quan sát một sự kiện j đã gán cho nó một xác suất (chủ quan) $P(j)$. Entropy được định nghĩa trong (4.4.11) sau đó là sự ngạc nhiên dự kiến * khi người ta gán xác suất chính xác thực sự phù hợp với quá trình tạo dữ liệu.

[54] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

Cross-Entropy Revisited

Vì vậy, nếu entropy là mức độ bất ngờ trải qua bởi một người biết xác suất thực sự, thì bạn có thể tự hỏi, cross-entropy là gì? Các cross-entropy* từ* P * đến* Q, được ký hiệu là $H(P, Q)$, là sự ngạc nhiên dự kiến của một người quan sát với xác suất chủ quan Q khi nhìn thấy dữ liệu thực sự được tạo ra theo xác suất P. Entropy chéo thấp nhất có thể đạt được khi $P = Q$. Trong trường hợp này, entropy chéo từ P đến Q là $H(P, P) = H(P)$.

Nói tóm lại, chúng ta có thể nghĩ đến mục tiêu phân loại chéo entropy theo hai cách: (i) là tối đa hóa khả năng của dữ liệu quan sát; và (ii) là giảm thiểu sự ngạc nhiên của chúng ta (và do đó số lượng bit) cần thiết để truyền đạt các nhãn.

4.4.8 Dự đoán và đánh giá mô hình

Sau khi đào tạo mô hình hồi quy softmax, cho bất kỳ tính năng ví dụ nào, chúng ta có thể dự đoán xác suất của mỗi lớp đầu ra. Thông thường, chúng ta sử dụng class có xác suất dự đoán cao nhất làm class output. Dự đoán là chính xác nếu nó phù hợp với lớp thực tế (label). Trong phần tiếp theo của thử nghiệm, chúng tôi sẽ sử dụng *accuracy* để đánh giá hiệu suất của mô hình. Điều này bằng tỷ lệ giữa số dự đoán chính xác và tổng số dự đoán.

4.4.9 Tóm tắt

- Thao tác softmax lấy một vector và ánh xạ nó thành xác suất.
- Hồi quy Softmax áp dụng cho các bài toán phân loại. Nó sử dụng phân phối xác suất của lớp đầu ra trong hoạt động softmax.
- Cross-entropy là một thước đo tốt về sự khác biệt giữa hai phân phối xác suất. Nó đo số lượng bit cần thiết để mã hóa dữ liệu cho mô hình của chúng tôi.

4.4.10 Bài tập

1. Chúng ta có thể khám phá mối liên hệ giữa các gia đình cấp số nhân và softmax ở một số chiều sâu hơn.
    1. Tính toán đạo hàm thứ hai của tổn thất chéo entropy $l(\mathbf{y}, \hat{\mathbf{y}})$ cho softmax.
    2. Tính toán phương sai của phân phối được đưa ra bởi $\text{softmax}(\mathbf{o})$ và cho thấy nó khớp với đạo hàm thứ hai được tính toán ở trên.
2. Giả sử rằng chúng ta có ba lớp xảy ra với xác suất bằng nhau, tức là vector xác suất là $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.
    1. Vấn đề là gì nếu chúng ta cố gắng thiết kế một mã nhị phân cho nó?
    2. Bạn có thể thiết kế một mã tốt hơn? Gợi ý: điều gì sẽ xảy ra nếu chúng ta cố gắng mã hóa hai quan sát độc lập? Điều gì sẽ xảy ra nếu chúng ta mã hóa n quan sát cùng nhau?
3. Softmax là một sai lầm cho bản đồ được giới thiệu ở trên (nhưng tất cả mọi người trong deep learning đều sử dụng nó). Softmax thực được định nghĩa là $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.
    1. Chứng minh rằng $\text{RealSoftMax}(a, b) > \max(a, b)$.
    2. Chứng minh rằng điều này giữ cho $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, với điều kiện là $\lambda > 0$.
    3. Cho thấy rằng đối với $\lambda \to \infty$ chúng tôi có $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.

4. Soft-min trông như thế nào?
5. Mở rộng điều này đến hơn hai số.

Discussions[55]

4.5 Tập dữ liệu phân loại hình ảnh

Một trong những tập dữ liệu được sử dụng rộng rãi để phân loại hình ảnh là tập dữ liệu MNIST (LeCun et al., 1998). Mặc dù nó có một chạy tốt như một tập dữ liệu chuẩn, thậm chí các mô hình đơn giản theo tiêu chuẩn ngày nay cũng đạt được độ chính xác phân loại trên 95%, khiến nó không phù hợp để phân biệt giữa các mô hình mạnh hơn và các mô hình yếu hơn. Ngày nay, MNIST phục vụ như là kiểm tra sự tỉnh táo hơn là một chuẩn mực. Để lên ante chỉ một chút, chúng tôi sẽ tập trung thảo luận của chúng tôi trong các phần sắp tới về chất lượng tương tự, nhưng tương đối phức tạp Fashion-MNIST dataset (Xiao et al., 2017), được phát hành vào năm 2017.

```
%matplotlib inline
import torch
import torchvision
from torch.utils import data
from torchvision import transforms
from d2l import torch as d2l

d2l.use_svg_display()
```

4.5.1 Đọc tập dữ liệu

Chúng ta có thể tải xuống và đọc tập dữ liệu Fashion-MNIST vào bộ nhớ thông qua các chức năng tích hợp trong khuôn việt.

```
# `ToTensor` converts the image data from PIL type to 32-bit floating point
# tensors. It divides all numbers by 255 so that all pixel values are between
# 0 and 1
trans = transforms.ToTensor()
mnist_train = torchvision.datasets.FashionMNIST(
    root="../data", train=True, transform=trans, download=True)
mnist_test = torchvision.datasets.FashionMNIST(
    root="../data", train=False, transform=trans, download=True)
```

Fashion-MNIST bao gồm các hình ảnh từ 10 loại, mỗi loại được đại diện bởi 6000 hình ảnh trong tập dữ liệu đào tạo và 1000 trong tập dữ liệu thử nghiệm. Một *test dataset* (hoặc * test set*) được sử dụng để đánh giá hiệu suất mô hình và không cho đào tạo. Do đó, bộ đào tạo và bộ thử nghiệm chứa 60000 và 10000 hình ảnh, tương ứng.

```
len(mnist_train), len(mnist_test)
```

```
(60000, 10000)
```

[55] https://discuss.d2l.ai/t/46

Chiều cao và chiều rộng của mỗi hình ảnh đầu vào đều là 28 pixel. Lưu ý rằng tập dữ liệu bao gồm các hình ảnh thang màu xám, có số kênh là 1. Đối với ngắn gọn, trong suốt cuốn sách này, chúng tôi lưu trữ hình dạng của bất kỳ hình ảnh nào có chiều cao h chiều rộng w pixel là $h \times w$ hoặc (h, w).

```
mnist_train[0][0].shape
```

```
torch.Size([1, 28, 28])
```

Các hình ảnh trong Fashion-MNIST được liên kết với các loại sau: áo phông, quần tây, áo thun, váy, áo khoác, sandal, áo sơ mi, giày thể thao, túi xách và khởi động mắt cá chân. Hàm sau chuyển đổi giữa các chỉ số nhãn số và tên của chúng trong văn bản.

```
def get_fashion_mnist_labels(labels):  #@save
    """Return text labels for the Fashion-MNIST dataset."""
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                   'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

Bây giờ chúng ta có thể tạo ra một hàm để hình dung các ví dụ này.

```
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):  #@save
    """Plot a list of images."""
    figsize = (num_cols * scale, num_rows * scale)
    _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
    axes = axes.flatten()
    for i, (ax, img) in enumerate(zip(axes, imgs)):
        if torch.is_tensor(img):
            # Tensor Image
            ax.imshow(img.numpy())
        else:
            # PIL Image
            ax.imshow(img)
        ax.axes.get_xaxis().set_visible(False)
        ax.axes.get_yaxis().set_visible(False)
        if titles:
            ax.set_title(titles[i])
    return axes
```

Dưới đây là ** hình ảnh và nhãn tương ứng của chúng** cho một vài ví dụ đầu tiên trong tập dữ liệu đào tạo.

```
X, y = next(iter(data.DataLoader(mnist_train, batch_size=18)))
show_images(X.reshape(18, 28, 28), 2, 9, titles=get_fashion_mnist_labels(y));
```

4.5.2 Đọc một Minibatch

Để làm cho cuộc sống của chúng ta dễ dàng hơn khi đọc từ các bộ đào tạo và kiểm tra, chúng tôi sử dụng bộ lặp dữ liệu tích hợp hơn là tạo một từ đầu. Nhớ lại rằng tại mỗi lần lặp lại, một bộ lặp dữ liệu đọc một minibatch dữ liệu với kích thước `batch_size` mỗi lần. Chúng tôi cũng ngẫu nhiên xáo trộn các ví dụ cho bộ lặp dữ liệu đào tạo.

```
batch_size = 256

def get_dataloader_workers():  #@save
    """Use 4 processes to read the data."""
    return 4

train_iter = data.DataLoader(mnist_train, batch_size, shuffle=True,
                             num_workers=get_dataloader_workers())
```

Chúng ta hãy nhìn vào thời gian cần thiết để đọc dữ liệu đào tạo.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
f'{timer.stop():.2f} sec'
```

```
'2.54 sec'
```

4.5.3 Đặt tất cả mọi thứ lại với nhau

Bây giờ chúng ta định nghĩa hàm `load_data_fashion_mnist` có được và đọc bộ dữ liệu Fashion-MNIST. Nó trả về các bộ lặp dữ liệu cho cả bộ đào tạo và bộ xác nhận. Ngoài ra, nó chấp nhận một đối số tùy chọn để thay đổi kích thước hình ảnh thành một hình dạng khác.

```
def load_data_fashion_mnist(batch_size, resize=None):  #@save
    """Download the Fashion-MNIST dataset and then load it into memory."""
    trans = [transforms.ToTensor()]
    if resize:
        trans.insert(0, transforms.Resize(resize))
    trans = transforms.Compose(trans)
    mnist_train = torchvision.datasets.FashionMNIST(
        root="../data", train=True, transform=trans, download=True)
    mnist_test = torchvision.datasets.FashionMNIST(
        root="../data", train=False, transform=trans, download=True)
    return (data.DataLoader(mnist_train, batch_size, shuffle=True,
                            num_workers=get_dataloader_workers()),
            data.DataLoader(mnist_test, batch_size, shuffle=False,
                            num_workers=get_dataloader_workers()))
```

Dưới đây chúng tôi kiểm tra tính năng thay đổi kích thước hình ảnh của hàm `load_data_fashion_mnist` bằng cách chỉ định đối số `resize`.

```
train_iter, test_iter = load_data_fashion_mnist(32, resize=64)
for X, y in train_iter:
```

(continues on next page)

(continued from previous page)

```
    print(X.shape, X.dtype, y.shape, y.dtype)
    break
```

```
torch.Size([32, 1, 64, 64]) torch.float32 torch.Size([32]) torch.int64
```

Bây giờ chúng tôi đã sẵn sàng để làm việc với tập dữ liệu Fashion-MNIST trong các phần tiếp theo.

4.5.4 Tóm tắt

- Fashion-MNIST là một tập dữ liệu phân loại trang phục bao gồm các hình ảnh đại diện cho 10 loại. Chúng tôi sẽ sử dụng tập dữ liệu này trong các phần và chương tiếp theo để đánh giá các thuật toán phân loại khác nhau.
- Chúng tôi lưu trữ hình dạng của bất kỳ hình ảnh nào với chiều cao h chiều rộng w pixel là $h \times w$ hoặc (h, w).
- Bộ lặp dữ liệu là một thành phần quan trọng cho hiệu suất hiệu quả. Dựa vào các bộ lặp dữ liệu được triển khai tốt khai thác tính toán hiệu suất cao để tránh làm chậm vòng đào tạo của bạn.

4.5.5 Bài tập

1. Việc giảm `batch_size` (ví dụ, xuống 1) có ảnh hưởng đến hiệu suất đọc không?
2. Hiệu suất lặp dữ liệu rất quan trọng. Bạn có nghĩ rằng việc thực hiện hiện tại là đủ nhanh? Khám phá các tùy chọn khác nhau để cải thiện nó.
3. Kiểm tra tài liệu API trực tuyến của framework. Những bộ dữ liệu nào khác có sẵn?

Discussions[56]

4.6 Thực hiện hồi quy Softmax từ đầu

Cũng giống như chúng tôi thực hiện hồi quy tuyến tính từ đầu, chúng tôi tin rằng that hồi quy softmax là tương tự cơ bản và bạn nên biết các chi tiết gory của và làm thế nào để thực hiện nó cho mình. Chúng tôi sẽ làm việc với bộ dữ liệu Fashion-MNIST, vừa được giới thiệu trong Section 4.5, thiết lập một bộ lặp dữ liệu với kích thước lô 256.

```
import torch
from IPython import display
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

[56] https://discuss.d2l.ai/t/49

4.6.1 Khởi tạo các tham số mô hình

Như trong ví dụ hồi quy tuyến tính của chúng ta, mỗi ví dụ ở đây sẽ được biểu diễn bằng một vectơ có độ dài cố định. Mỗi ví dụ trong tập dữ liệu thô là một hình ảnh 28×28. Trong phần này, chúng ta sẽ làm phẳng từng hình ảnh, coi chúng là vectơ có độ dài 784. Trong tương lai, chúng ta sẽ nói về các chiến lược phức tạp hơn để khai thác cấu trúc không gian trong hình ảnh, nhưng bây giờ chúng ta coi từng vị trí pixel như một tính năng khác.

Nhớ lại rằng trong hồi quy softmax, chúng ta có nhiều đầu ra như có các lớp. Vì tập dữ liệu của chúng tôi có 10 lớp, mạng của chúng tôi sẽ có kích thước đầu ra là 10. Do đó, trọng lượng của chúng tôi sẽ tạo thành ma trận 784×10 và các thành kiến sẽ tạo thành một vector hàng 1×10. Như với hồi quy tuyến tính, chúng tôi sẽ khởi tạo trọng lượng của chúng tôi `W` với tiếng ồn Gaussian và thành kiến của chúng tôi để lấy giá trị ban đầu 0.

```
num_inputs = 784
num_outputs = 10

W = torch.normal(0, 0.01, size=(num_inputs, num_outputs), requires_grad=True)
b = torch.zeros(num_outputs, requires_grad=True)
```

4.6.2 Xác định hoạt động Softmax

Trước khi thực hiện mô hình hồi quy softmax, chúng ta hãy xem xét ngắn gọn cách toán tử tổng hoạt động dọc theo các kích thước cụ thể trong một tensor, như đã thảo luận trong Section 3.3.6 và Section 3.3.6. Cho một ma trận `X` chúng ta có thể tổng hợp tất cả các phần tử (theo mặc định) hoặc chỉ trên các phần tử trong cùng một trục, tức là, cùng một cột (trục 0) hoặc cùng một hàng (trục 1). Lưu ý rằng nếu `X` là một tensor với hình dạng (2, 3) và chúng ta tổng hợp trên các cột, kết quả sẽ là một vectơ có hình dạng (3,). Khi gọi toán tử tổng, chúng ta có thể chỉ định để giữ số trục trong tensor ban đầu, thay vì thu gọn kích thước mà chúng ta tóm tắt. Điều này sẽ dẫn đến một tensor hai chiều với hình dạng (1, 3).

```
X = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
X.sum(0, keepdim=True), X.sum(1, keepdim=True)
```

```
(tensor([[5., 7., 9.]]),
 tensor([[ 6.],
         [15.]]))
```

Bây giờ chúng tôi đã sẵn sàng để thực hiện các hoạt động softmax. Nhớ lại rằng softmax bao gồm ba bước: (i) chúng ta cấp mũ mỗi thuật ngữ (sử dụng `exp`); (ii) chúng ta tổng hợp trên mỗi hàng (chúng ta có một hàng cho mỗi ví dụ trong lô) để lấy hằng số bình thường hóa cho mỗi ví dụ; (iii) chúng ta chia mỗi hàng bằng hằng số bình thường hóa của nó, đảm bảo rằng kết quả tổng thành 1. Trước khi nhìn vào mã, chúng ta hãy nhớ lại cách này trông thể hiện như một phương trình:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{X}_{ij})}{\sum_k \exp(\mathbf{X}_{ik})}. \tag{4.6.1}$$

Mẫu số, hoặc hằng số chuẩn hóa, đôi khi cũng được gọi là hàm phân vùng * (và logarit của nó được gọi là hàm logarit). Nguồn gốc của tên đó nằm trong vật lý thống kê[57] trong đó một phương trình liên quan mô hình sự phân bố trên một nhóm các hạt.

[57] https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics)

```
def softmax(X):
    X_exp = torch.exp(X)
    partition = X_exp.sum(1, keepdim=True)
    return X_exp / partition  # The broadcasting mechanism is applied here
```

Như bạn có thể thấy, đối với bất kỳ đầu vào ngẫu nhiên nào, chúng tôi biến mỗi phần tử thành một số không âm. Hơn nữa, mỗi hàng tổng cộng lên đến 1, như là cần thiết cho một xác suất.

```
X = torch.normal(0, 1, (2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(1)
```

```
(tensor([[0.2157, 0.0482, 0.0647, 0.4588, 0.2126],
         [0.0683, 0.0946, 0.1220, 0.3124, 0.4028]]),
 tensor([1.0000, 1.0000]))
```

Lưu ý rằng mặc dù điều này có vẻ chính xác về mặt toán học, chúng tôi hơi cẩu thả trong việc thực hiện bởi vì chúng tôi không thực hiện các biện pháp phòng ngừa chống lại tràn số hoặc tràn do các yếu tố lớn hoặc rất nhỏ của ma trận.

4.6.3 Xác định mô hình

Bây giờ chúng ta đã xác định hoạt động softmax, chúng ta có thể triển khai mô hình hồi quy softmax. Đoạn mã dưới đây định nghĩa cách nhập liệu được ánh xạ đến đầu ra thông qua mạng. Lưu ý rằng chúng ta làm phẳng từng ảnh gốc trong lô thành một vectơ bằng hàm `reshape` trước khi truyền dữ liệu qua mô hình của chúng ta.

```
def net(X):
    return softmax(torch.matmul(X.reshape((-1, W.shape[0])), W) + b)
```

4.6.4 Xác định chức năng mất

Tiếp theo, chúng ta cần thực hiện hàm mất chéo entropy, như được giới thiệu trong Section 4.4. Đây có thể là chức năng mất mát phổ biến nhất trong tất cả các học sâu bởi vì, tại thời điểm này, các vấn đề phân loại vượt xa vấn đề hồi quy.

Nhớ lại rằng cross-entropy lấy khả năng log âm của xác suất dự đoán được gán cho nhãn thật. Thay vì lặp qua các dự đoán bằng Python for-loop (có xu hướng không hiệu quả), chúng ta có thể chọn tất cả các phần tử bằng một toán tử duy nhất. Dưới đây, chúng ta tạo dữ liệu mẫu `y_hat` với 2 ví dụ về xác suất dự đoán trên 3 lớp và nhãn tương ứng của chúng `y`. Với `y` chúng ta biết rằng trong ví dụ đầu tiên, lớp đầu tiên là dự đoán chính xác và trong ví dụ thứ hai, lớp thứ ba là sự thật. Sử dụng `y` làm chỉ số xác suất trong `y_hat`, chúng ta chọn xác suất của lớp đầu tiên trong ví dụ đầu tiên và xác suất của lớp thứ ba trong ví dụ thứ hai.

```
y = torch.tensor([0, 2])
y_hat = torch.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

Bây giờ chúng ta có thể triển khai hàm mất chéo entropy một cách hiệu quả chỉ với một dòng mã.

```
def cross_entropy(y_hat, y):
    return - torch.log(y_hat[range(len(y_hat)), y])

cross_entropy(y_hat, y)
```

```
tensor([2.3026, 0.6931])
```

4.6.5 Phân loại chính xác

Với phân phối xác suất dự đoán `y_hat`, chúng ta thường chọn lớp có xác suất dự đoán cao nhất bất cứ khi nào chúng ta phải đưa ra một dự đoán cứng. Thật vậy, nhiều ứng dụng yêu cầu chúng tôi đưa ra lựa chọn. Gmail phải phân loại email thành "Chính", "Xã hội", "Cập nhật" hoặc "Diễn đàn". Nó có thể ước tính xác suất trong nội bộ, nhưng vào cuối ngày, nó phải chọn một trong số các lớp học.

Khi dự đoán phù hợp với lớp nhãn `y`, chúng là chính xác. Độ chính xác phân loại là phần nhỏ của tất cả các dự đoán là chính xác. Mặc dù có thể khó tối ưu hóa độ chính xác trực tiếp (nó không phân biệt được), nhưng nó thường là biện pháp hiệu suất mà chúng tôi quan tâm nhất và gần như chúng tôi sẽ luôn báo cáo khi đào tạo phân loại.

Để tính toán độ chính xác, chúng tôi làm như sau. Đầu tiên, nếu `y_hat` là một ma trận, chúng ta giả định rằng chiều thứ hai lưu trữ điểm dự đoán cho mỗi lớp. Chúng tôi sử dụng `argmax` để có được lớp dự đoán bằng chỉ số cho mục nhập lớn nhất trong mỗi hàng. Sau đó, chúng ta so sánh lớp dự đoán với `y` elementwise. Vì toán tử bình đẳng == nhạy cảm với các kiểu dữ liệu, chúng tôi chuyển đổi kiểu dữ liệu của `y_hat` để khớp với `y`. Kết quả là một tensor chứa các mục của 0 (false) và 1 (true). Lấy tổng sản lượng số dự đoán chính xác.

```
def accuracy(y_hat, y):  #@save
    """Compute the number of correct predictions."""
    if len(y_hat.shape) > 1 and y_hat.shape[1] > 1:
        y_hat = y_hat.argmax(axis=1)
    cmp = y_hat.type(y.dtype) == y
    return float(cmp.type(y.dtype).sum())
```

Chúng tôi sẽ tiếp tục sử dụng các biến `y_hat` và `y` được xác định trước đó như là phân phối xác suất dự đoán và nhãn, tương ứng. Chúng ta có thể thấy lớp dự đoán của ví dụ đầu tiên là 2 (phần tử lớn nhất của hàng là 0,6 với chỉ số 2), không phù hợp với nhãn thực tế, 0. Lớp dự đoán của ví dụ thứ hai là 2 (phần tử lớn nhất của hàng là 0,5 với chỉ số 2), phù hợp với nhãn thực tế, 2. Do đó, tỷ lệ chính xác phân loại cho hai ví dụ này là 0,5.

```
accuracy(y_hat, y) / len(y)
```

```
0.5
```

Tương tự, chúng ta có thể đánh giá độ chính xác cho bất kỳ model `net` nào trên một bộ dữ liệu được truy cập thông qua bộ lặp dữ liệu `data_iter`.

```
def evaluate_accuracy(net, data_iter):  #@save
    """Compute the accuracy for a model on a dataset."""
    if isinstance(net, torch.nn.Module):
        net.eval()  # Set the model to evaluation mode
```

(continues on next page)

(continued from previous page)

```
    metric = Accumulator(2)  # No. of correct predictions, no. of predictions

    with torch.no_grad():
        for X, y in data_iter:
            metric.add(accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Ở đây `Accumulator` là một lớp tiện ích để tích lũy tổng trên nhiều biến. Trong hàm `evaluate_accuracy` trên, chúng ta tạo ra 2 biến trong phiên bản `Accumulator` để lưu trữ cả số dự đoán đúng và số dự đoán, tương ứng. Cả hai sẽ được tích lũy theo thời gian khi chúng ta lặp lại tập dữ liệu.

```
class Accumulator:  #@save
    """For accumulating sums over `n` variables."""
    def __init__(self, n):
        self.data = [0.0] * n

    def add(self, *args):
        self.data = [a + float(b) for a, b in zip(self.data, args)]

    def reset(self):
        self.data = [0.0] * len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]
```

Bởi vì chúng tôi đã khởi tạo mô hình `net` với trọng lượng ngẫu nhiên, độ chính xác của mô hình này phải gần với đoán ngẫu nhiên, tức là 0,1 cho 10 lớp.

```
evaluate_accuracy(net, test_iter)
```

```
0.1485
```

4.6.6 Đào tạo

Vòng đào tạo cho hồi quy softmax sẽ trông nổi bật quen thuộc nếu bạn đọc thông qua việc thực hiện hồi quy tuyến tính của chúng tôi trong Section 4.2. Ở đây chúng tôi tái cấu trúc việc thực hiện để làm cho nó có thể tái sử dụng. Đầu tiên, chúng ta định nghĩa một hàm để đào tạo cho một kỷ nguyên. Lưu ý rằng `updater` là một hàm chung để cập nhật các tham số mô hình, chấp nhận kích thước lô làm đối số. Nó có thể là một trình bao bọc của hàm `d2l.sgd` hoặc chức năng tối ưu hóa tích hợp của khung.

```
def train_epoch_ch3(net, train_iter, loss, updater):  #@save
    """The training loop defined in Chapter 3."""
    # Set the model to training mode
    if isinstance(net, torch.nn.Module):
        net.train()
    # Sum of training loss, sum of training accuracy, no. of examples
    metric = Accumulator(3)
    for X, y in train_iter:
        # Compute gradients and update parameters
        y_hat = net(X)
```

(continues on next page)

(continued from previous page)

```
        l = loss(y_hat, y)
        if isinstance(updater, torch.optim.Optimizer):
            # Using PyTorch in-built optimizer & loss criterion
            updater.zero_grad()
            l.mean().backward()
            updater.step()
        else:
            # Using custom built optimizer & loss criterion
            l.sum().backward()
            updater(X.shape[0])
        metric.add(float(l.sum()), accuracy(y_hat, y), y.numel())
    # Return training loss and training accuracy
    return metric[0] / metric[2], metric[1] / metric[2]
```

Trước khi hiển thị việc thực hiện hàm đào tạo, chúng ta định nghĩa một lớp tiện ích vẽ dữ liệu trong hoạt hình. Một lần nữa, nó nhằm mục đích đơn giản hóa mã trong phần còn lại của cuốn sách.

```
class Animator:  #@save
    """For plotting data in animation."""
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        d2l.use_svg_display()
        self.fig, self.axes = d2l.plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a lambda function to capture arguments
        self.config_axes = lambda: d2l.set_axes(
            self.axes[0], xlabel, ylabel, xlim, ylim, xscale, yscale, legend)
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
```

(continues on next page)

```
        display.display(self.fig)
        display.clear_output(wait=True)
```

Chức năng đào tạo sau đây sau đó đào tạo một mô hình `net` trên một tập dữ liệu đào tạo được truy cập qua `train_iter` cho nhiều kỷ nguyên, được chỉ định bởi `num_epochs`. Vào cuối mỗi kỷ nguyên, mô hình được đánh giá trên một tập dữ liệu thử nghiệm truy cập qua `test_iter`. Chúng tôi sẽ tận dụng lớp `Animator` để hình dung tiến độ đào tạo.

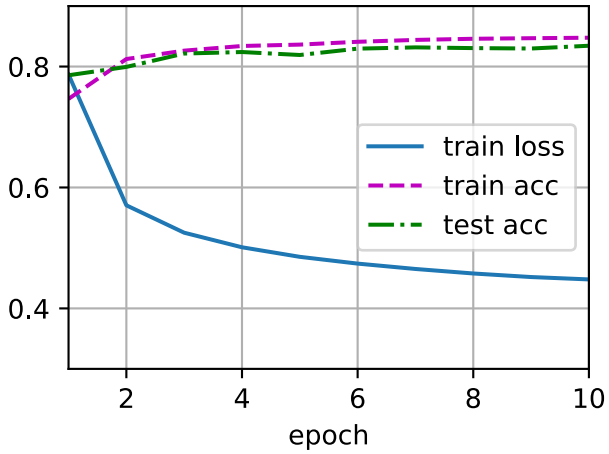
```
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):  #@save
    """Train a model (defined in Chapter 3)."""
    animator = Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0.3, 0.9],
                        legend=['train loss', 'train acc', 'test acc'])
    for epoch in range(num_epochs):
        train_metrics = train_epoch_ch3(net, train_iter, loss, updater)
        test_acc = evaluate_accuracy(net, test_iter)
        animator.add(epoch + 1, train_metrics + (test_acc,))
    train_loss, train_acc = train_metrics
    assert train_loss < 0.5, train_loss
    assert train_acc <= 1 and train_acc > 0.7, train_acc
    assert test_acc <= 1 and test_acc > 0.7, test_acc
```

Là một triển khai từ đầu, chúng tôi sử dụng minibatch stochastic gradient descent được định nghĩa trong Section 4.2 để tối ưu hóa chức năng mất mát của mô hình với tốc độ học tập 0.1.

```
lr = 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)
```

Bây giờ chúng ta đào tạo mô hình với 10 kỷ nguyên Lưu ý rằng cả số epochs (`num_epochs`) và tốc độ học tập (`lr`) đều có thể điều chỉnh các siêu tham số. Bằng cách thay đổi giá trị của chúng, chúng ta có thể tăng độ chính xác phân loại của mô hình.

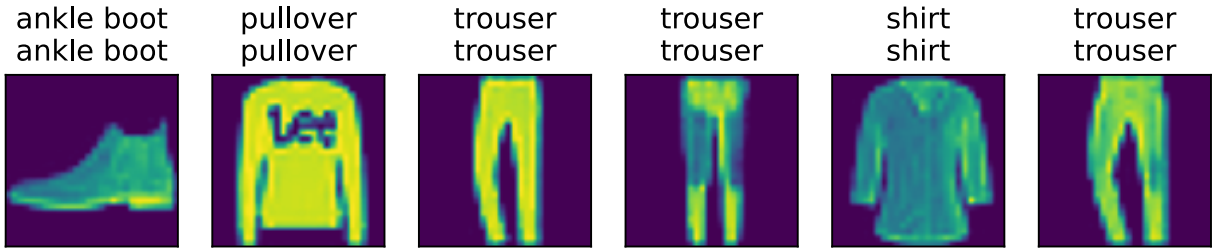
```
num_epochs = 10
train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)
```



4.6.7 Prediction

Bây giờ việc đào tạo đã hoàn tất, mô hình của chúng tôi đã sẵn sàng để phân loại một số hình ảnh Với một loạt các hình ảnh, chúng tôi sẽ so sánh các nhãn thực tế của chúng (dòng đầu ra văn bản đầu tiên) và dự đoán từ mô hình (dòng thứ hai của đầu ra văn bản).

```
def predict_ch3(net, test_iter, n=6):  #@save
    """Predict labels (defined in Chapter 3)."""
    for X, y in test_iter:
        break
    trues = d2l.get_fashion_mnist_labels(y)
    preds = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1))
    titles = [true +'\n' + pred for true, pred in zip(trues, preds)]
    d2l.show_images(
        X[0:n].reshape((n, 28, 28)), 1, n, titles=titles[0:n])

predict_ch3(net, test_iter)
```



4.6.8 Tóm tắt

- Với hồi quy softmax, chúng ta có thể đào tạo các mô hình để phân loại đa lớp.
- Vòng đào tạo của hồi quy softmax rất giống với trong hồi quy tuyến tính: lấy và đọc dữ liệu, xác định mô hình và chức năng mất mát, sau đó đào tạo các mô hình sử dụng các thuật toán tối ưu hóa. Như bạn sẽ sớm tìm ra, hầu hết các mô hình học sâu phổ biến đều có các quy trình đào tạo tương tự.

4.6.9 Bài tập

1. Trong phần này, chúng tôi trực tiếp triển khai hàm softmax dựa trên định nghĩa toán học của phép toán softmax. Những vấn đề này có thể gây ra? Gợi ý: cố gắng tính toán kích thước của $\exp(50)$.
2. Hàm `cross_entropy` trong phần này được thực hiện theo định nghĩa của hàm mất chéo entropy. Điều gì có thể là vấn đề với việc thực hiện này? Gợi ý: xem xét tên miền của logarit.
3. Những giải pháp bạn có thể nghĩ đến để khắc phục hai vấn đề ở trên?
4. Có phải luôn luôn là một ý tưởng tốt để trả lại nhãn có khả năng nhất? Ví dụ, bạn sẽ làm điều này để chẩn đoán y tế?
5. Giả sử rằng chúng ta muốn sử dụng hồi quy softmax để dự đoán từ tiếp theo dựa trên một số tính năng. Một số vấn đề có thể phát sinh từ một từ vựng lớn là gì?

Discussions[58]

[58] https://discuss.d2l.ai/t/51

4.7 Thực hiện ngắn gọn về hồi quy Softmax

Cũng giống như APIs cấp cao của các khuôn khổ học sâu làm cho nó dễ dàng hơn nhiều để thực hiện hồi quy tuyến tính trong Section 4.3, chúng tôi sẽ tìm thấy nó tương tự (hoặc có thể nhiều hơn) thuận tiện cho việc thực hiện các mô hình phân loại. Hãy để chúng tôi gắn bó với tập dữ liệu Fashion-MNIST và giữ kích thước lô ở mức 256 như trong Section 4.6.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.7.1 Khởi tạo các tham số mô hình

Như đã đề cập trong Section 4.4, lớp đầu ra của hồi quy softmax là một lớp kết nối đầy đủ Do đó, để thực hiện mô hình của chúng tôi, chúng ta chỉ cần thêm một lớp kết nối hoàn toàn với 10 đầu ra vào `Sequential` của chúng tôi. Một lần nữa, ở đây, `Sequential` không thực sự cần thiết, nhưng chúng ta cũng có thể hình thành thói quen vì nó sẽ phổ biến khi thực hiện các mô hình sâu. Một lần nữa, chúng tôi khởi tạo các trọng lượng một cách ngẫu nhiên với 0 trung bình và độ lệch chuẩn 0,01.

```
# PyTorch does not implicitly reshape the inputs. Thus we define the flatten
# layer to reshape the inputs before the linear layer in our network
net = nn.Sequential(nn.Flatten(), nn.Linear(784, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

4.7.2 Triển khai Softmax Revisited

Trong ví dụ trước của Section 4.6, chúng tôi đã tính toán đầu ra của mô hình và sau đó chạy đầu ra này thông qua tổn thất ngẫu nhiên chéo. Về mặt toán học, đó là một điều hoàn toàn hợp lý để làm. Tuy nhiên, từ góc độ tính toán, số mũ có thể là một nguồn của các vấn đề ổn định số.

Nhớ lại rằng hàm softmax tính toán $\hat{y}_j = \frac{\exp(o_j)}{\sum_k \exp(o_k)}$, trong đó $\hat{y}_j$ là phần tử j^{th} của phân phối xác suất dự đoán $\hat{\mathbf{y}}$ và o_j là phần tử j^{th} của logits $\mathbf{o}$. Nếu một số o_k rất lớn (tức là rất tích cực), thì $\exp(o_k)$ có thể lớn hơn số lớn nhất chúng ta có thể có cho một số loại dữ liệu nhất định (tức là * đầu*). Điều này sẽ làm cho mẫu số (và/hoặc tử số) `inf` (vô cùng) và chúng tôi gặp phải 0, `inf` hoặc `nan` (không phải là một số) cho $\hat{y}_j$. Trong những tình huống này, chúng ta không nhận được giá trị trả về được xác định rõ cho cross-entropy.

Một mẹo để giải quyết vấn đề này là lần đầu tiên trừ $\max(o_k)$ khỏi tất cả o_k trước khi tiến hành tính toán softmax. Bạn có thể thấy rằng sự dịch chuyển này của mỗi o_k theo hệ số không đổi không thay đổi giá trị trả

về của softmax:

$$\begin{aligned}\hat{y}_j &= \frac{\exp(o_j - \max(o_k))\exp(\max(o_k))}{\sum_k \exp(o_k - \max(o_k))\exp(\max(o_k))} \\ &= \frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}.\end{aligned} \tag{4.7.1}$$

Sau bước trừ và bình thường hóa, có thể một số $o_j - \max(o_k)$ có giá trị âm lớn và do đó $\exp(o_j - \max(o_k))$ tương ứng sẽ lấy giá trị gần bằng 0. Chúng có thể được làm tròn thành 0 do độ chính xác hữu hạn (ví dụ: * underflow*), làm cho $\hat{y}_j$ không và cho chúng tôi `-inf` cho $\log(\hat{y}_j)$. Một vài bước xuống con đường trong backpropagation, chúng ta có thể thấy mình phải đối mặt với một màn hình của đáng sợ `nan` kết quả.

May mắn thay, chúng tôi được lưu bởi thực tế là mặc dù chúng tôi đang tính toán các hàm mũ, cuối cùng chúng tôi có ý định lấy nhật ký của chúng (khi tính toán mất chéo entropy). Bằng cách kết hợp hai toán tử softmax và cross-entropy với nhau, chúng ta có thể thoát khỏi các vấn đề ổn định số mà nếu không có thể gây ra chúng ta trong quá trình truyền ngược. Như thể hiện trong phương trình dưới đây, chúng ta tránh tính toán $\exp(o_j - \max(o_k))$ và có thể sử dụng thay thế $o_j - \max(o_k)$ trực tiếp do việc hủy bỏ trong $\log(\exp(\cdot))$:

$$\begin{aligned}\log\left(\hat{y}_j\right) &= \log\left(\frac{\exp(o_j - \max(o_k))}{\sum_k \exp(o_k - \max(o_k))}\right) \\ &= \log\left(\exp(o_j - \max(o_k))\right) - \log\left(\sum_k \exp(o_k - \max(o_k))\right) \\ &= o_j - \max(o_k) - \log\left(\sum_k \exp(o_k - \max(o_k))\right).\end{aligned} \tag{4.7.2}$$

Chúng tôi sẽ muốn giữ chức năng softmax thông thường tiện dụng trong trường hợp chúng tôi muốn đánh giá xác suất đầu ra theo mô hình của chúng tôi. Nhưng thay vì truyền xác suất softmax vào chức năng mất mát mới của chúng ta, chúng ta sẽ chỉ vượt qua các logits và tính toán softmax và nhật ký của nó cùng một lúc bên trong hàm loss cross-entropy, mà thực hiện những điều thông minh như "LogSumExp trick"[59].

```
loss = nn.CrossEntropyLoss(reduction='none')
```

4.7.3 Thuật toán tối ưu hóa

Ở đây, chúng ta sử dụng minibatch stochastic gradient descent với tốc độ học tập là 0.1 làm thuật toán tối ưu hóa. Lưu ý rằng điều này giống như chúng ta đã áp dụng trong ví dụ hồi quy tuyến tính và nó minh họa khả năng ứng dụng chung của các trình tối ưu hóa.

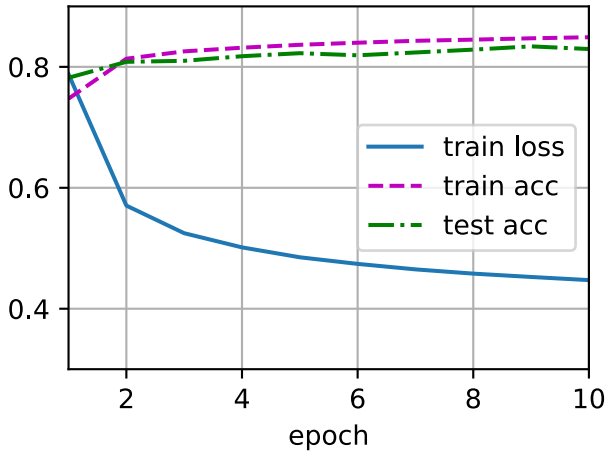
```
trainer = torch.optim.SGD(net.parameters(), lr=0.1)
```

[59] https://en.wikipedia.org/wiki/LogSumExp

4.7.4 Đào tạo

Tiếp theo chúng ta [**gọi hàm đào tạo**] trong Section 4.6 để đào tạo mô hình.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



Như trước đây, thuật toán này hội tụ thành một giải pháp đạt được độ chính xác khá, mặc dù lần này với ít dòng mã hơn trước.

4.7.5 Tóm tắt

- Sử dụng API cấp cao, chúng ta có thể thực hiện hồi quy softmax chính xác hơn nhiều.
- Từ góc độ tính toán, việc thực hiện hồi quy softmax có những phức tạp. Lưu ý rằng trong nhiều trường hợp, một khuôn khổ học sâu thực hiện các biện pháp phòng ngừa bổ sung ngoài các thủ thuật nổi tiếng nhất này để đảm bảo sự ổn định về số, cứu chúng ta khỏi những cạm bẫy hơn nữa mà chúng ta sẽ gặp phải nếu chúng ta cố gắng mã hóa tất cả các mô hình của mình từ đầu trong thực tế.

4.7.6 Bài tập

1. Hãy thử điều chỉnh các siêu tham số, chẳng hạn như kích thước lô, số kỷ nguyên và tốc độ học tập, để xem kết quả là gì.
2. Tăng số lượng kỷ nguyên để đào tạo. Tại sao độ chính xác thử nghiệm có thể giảm sau một thời gian? Làm thế nào chúng ta có thể sửa chữa điều này?

Discussions[60]

[60] https://discuss.d2l.ai/t/53

5 | Multilayer Perceptrons

Trong chương này, chúng tôi sẽ giới thiệu mạng thực sự * sâu * đầu tiên của bạn. Các mạng sâu đơn giản nhất được gọi là nhận thức đa lớp, và chúng bao gồm nhiều lớp tế bào thần kinh mỗi lớp được kết nối hoàn toàn với những người trong lớp bên dưới (từ đó chúng nhận được đầu vào) và những người ở trên (đến lượt nó, ảnh hưởng). Khi chúng tôi đào tạo các mô hình công suất cao, chúng tôi có nguy cơ vượt quá. Vì vậy, chúng tôi sẽ cần phải cung cấp giới thiệu nghiêm ngặt đầu tiên của bạn về các khái niệm về overfitting, underfitting, và lựa chọn mô hình. Để giúp bạn chống lại những vấn đề này, chúng tôi sẽ giới thiệu các kỹ thuật chính quy hóa như phân rã cân và bỏ học. Chúng tôi cũng sẽ thảo luận về các vấn đề liên quan đến tính ổn định số và khởi tạo tham số là chìa khóa để đào tạo thành công các mạng sâu. Trong suốt, chúng tôi mong muốn cung cấp cho bạn một nắm bắt vững chắc không chỉ về các khái niệm mà còn về việc thực hành sử dụng các mạng sâu. Vào cuối chương này, chúng tôi áp dụng những gì chúng tôi đã giới thiệu cho đến nay cho một trường hợp thực sự: dự đoán giá nhà. Chúng tôi đánh giá các vấn đề liên quan đến hiệu suất tính toán, khả năng mở rộng và hiệu quả của các mô hình của chúng tôi cho các chương tiếp theo.

5.1 Multilayer Perceptrons

Trong Chapter 4, chúng tôi giới thiệu hồi quy softmax (Section 4.4), thực hiện thuật toán từ đầu (Section 4.6) và sử dụng API cấp cao (Section 4.7), và đào tạo phân loại để nhận ra 10 loại quần áo từ hình ảnh có độ phân giải thấp. Trên đường đi, chúng tôi đã học cách wrangle dữ liệu, ép buộc đầu ra của chúng tôi vào một phân phối xác suất hợp lệ, áp dụng một chức năng mất mát thích hợp và giảm thiểu nó đối với các tham số của mô hình của chúng tôi. Bây giờ chúng ta đã thành thạo các cơ chế này trong bối cảnh các mô hình tuyến tính đơn giản, chúng ta có thể khởi động khám phá các mạng thần kinh sâu, lớp mô hình tương đối phong phú mà cuốn sách này chủ yếu quan tâm.

5.1.1 Các lớp ẩn

Chúng tôi đã mô tả sự biến đổi affine trong Section 4.1.1, là một biến đổi tuyến tính được thêm vào bởi một sự thiên vị. Để bắt đầu, hãy nhớ lại kiến trúc mô hình tương ứng với ví dụ hồi quy softmax của chúng tôi, minh họa trong Fig. 4.4.1. Mô hình này ánh xạ đầu vào của chúng tôi trực tiếp đến đầu ra của chúng tôi thông qua một biến đổi affine duy nhất, tiếp theo là một hoạt động softmax. Nếu nhãn của chúng tôi thực sự có liên quan đến dữ liệu đầu vào của chúng tôi bằng cách chuyển đổi affine, thì cách tiếp cận này sẽ là đủ. Nhưng tuyến tính trong biến đổi affine là một giả định * mạnh mẽ*.

Mô hình tuyến tính có thể đi sai

Ví dụ, tuyến tính ngụ ý giả định *yếu hơn* về *monotonicity*: rằng bất kỳ sự gia tăng tính năng nào của chúng tôi phải luôn gây ra sự gia tăng sản lượng của mô hình của chúng tôi (nếu trọng lượng tương ứng là dương) hoặc luôn làm giảm sản lượng của mô hình của chúng tôi (nếu trọng lượng tương ứng là âm). Đôi khi điều đó có ý nghĩa. Ví dụ, nếu chúng tôi đang cố gắng dự đoán liệu một cá nhân có trả nợ hay không, chúng ta có thể tưởng tượng một cách hợp lý rằng giữ tất cả đều bằng nhau, một người nộp đơn có thu nhập cao hơn sẽ luôn có nhiều khả năng trả nợ hơn một người có thu nhập thấp hơn. Trong khi đơn điệu, mối quan hệ này có khả năng không liên quan đến xác suất trả nợ. Tăng thu nhập từ 0 lên 50 nghìn có khả năng tương ứng với mức tăng khả năng trả nợ lớn hơn mức tăng từ 1 triệu lên 1.05 triệu. Một cách để xử lý điều này có thể là xử lý trước dữ liệu của chúng tôi sao cho tuyến tính trở nên hợp lý hơn, ví dụ, bằng cách sử dụng logarit thu nhập làm tính năng của chúng tôi.

Lưu ý rằng chúng ta có thể dễ dàng đưa ra các ví dụ vi phạm tính đơn điệu. Nói ví dụ rằng chúng ta muốn dự đoán xác suất tử vong dựa trên nhiệt độ cơ thể. Đối với những người có nhiệt độ cơ thể trên 37° C (98,6° F), nhiệt độ cao hơn cho thấy nguy cơ cao hơn. Tuy nhiên, đối với những người có nhiệt độ cơ thể dưới 37° C, nhiệt độ cao hơn cho thấy nguy cơ thấp hơn! Trong trường hợp này, chúng tôi cũng có thể giải quyết vấn đề với một số tiền xử lý thông minh. Cụ thể, chúng ta có thể sử dụng khoảng cách từ 37° C làm tính năng của chúng tôi.

Nhưng những gì về việc phân loại hình ảnh của mèo và chó? Có nên tăng cường độ của pixel tại vị trí (13, 17) luôn tăng (hoặc luôn giảm) khả năng hình ảnh mô tả một chú chó? Sự phụ thuộc vào một mô hình tuyến tính tương ứng với giả định ngầm rằng yêu cầu duy nhất để phân biệt mèo so với chó là đánh giá độ sáng của từng pixel. Cách tiếp cận này cam chịu thất bại trong một thế giới nơi đảo ngược một hình ảnh bảo tồn danh mục.

Tuy nhiên, mặc dù sự vô lý rõ ràng của tuyến tính ở đây, so với các ví dụ trước đây của chúng tôi, nó là ít rõ ràng rằng chúng tôi có thể giải quyết vấn đề với một sửa chữa tiền xử lý đơn giản. Đó là do ý nghĩa của bất kỳ pixel nào phụ thuộc theo những cách phức tạp vào bối cảnh của nó (các giá trị của các pixel xung quanh). Mặc dù có thể tồn tại một đại diện của dữ liệu của chúng tôi có thể tính đến các tương tác có liên quan giữa các tính năng của chúng tôi, trên đó một mô hình tuyến tính sẽ phù hợp, chúng tôi chỉ đơn giản là không biết cách tính toán nó bằng tay. Với các mạng nơ-ron sâu, chúng tôi đã sử dụng dữ liệu quan sát để cùng tìm hiểu cả một biểu diễn thông qua các lớp ẩn và một bộ dự đoán tuyến tính hoạt động dựa trên biểu diễn đó.

Kết hợp các lớp ẩn

Chúng ta có thể khắc phục những hạn chế này của các mô hình tuyến tính và xử lý một lớp hàm tổng quát hơn bằng cách kết hợp một hoặc nhiều lớp ẩn. Cách dễ nhất để làm điều này là xếp chồng nhiều lớp được kết nối hoàn toàn lên nhau. Mỗi lớp nạp vào lớp phía trên nó, cho đến khi chúng ta tạo ra các đầu ra. Chúng ta có thể nghĩ về $L - 1$ lớp đầu tiên là đại diện của chúng ta và lớp cuối cùng là bộ dự đoán tuyến tính của chúng ta. Kiến trúc này thường được gọi là một *multilayer perceptron*, thường được viết tắt là *MLP*. Dưới đây, chúng tôi mô tả một MLP diagrammatically (Fig. 5.1.1).

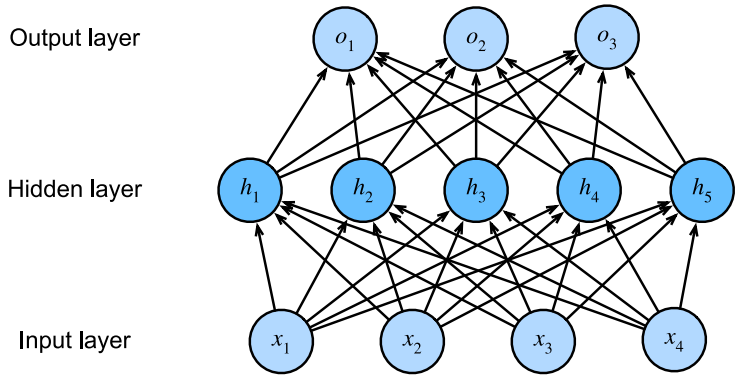


Fig. 5.1.1: An MLP with a hidden layer of 5 hidden units.

MLP này có 4 đầu vào, 3 đầu ra và lớp ẩn của nó chứa 5 đơn vị ẩn. Vì lớp đầu vào không liên quan đến bất kỳ tính toán nào, việc tạo ra kết quả đầu ra với mạng này đòi hỏi phải thực hiện các tính toán cho cả hai lớp ẩn và đầu ra; do đó, số lớp trong MLP này là 2. Lưu ý rằng các lớp này đều được kết nối hoàn toàn. Mỗi đầu vào ảnh hưởng đến mọi tế bào thần kinh trong lớp ẩn, và mỗi lần lượt ảnh hưởng đến mọi tế bào thần kinh trong lớp đầu ra. Tuy nhiên, theo đề xuất của Section 4.4.3, chi phí tham số hóa của MLP với các lớp được kết nối hoàn toàn có thể cao, điều này có thể thúc đẩy sự cân bằng giữa tiết kiệm tham số và hiệu quả mô hình ngay cả khi không thay đổi kích thước đầu vào hoặc đầu ra [Zhang.Tay.Zhang.ea.2021].

Từ tuyến tính đến phi tuyến

Như trước đây, bởi ma trận $\mathbf{X} \in \mathbb{R}^{n \times d}$, chúng tôi biểu thị một minibatch gồm n ví dụ trong đó mỗi ví dụ có d đầu vào (tính năng). Đối với một MLP một lớp ẩn có lớp ẩn có h đơn vị ẩn, biểu thị bằng $\mathbf{H} \in \mathbb{R}^{n \times h}$ các đầu ra của lớp ẩn, đó là *đại diện ẩn*. Trong toán học hoặc mã, $\mathbf{H}$ còn được gọi là một biến thể * ẩn lớp* hoặc một biến * hidden*. Vì các lớp ẩn và đầu ra đều được kết nối hoàn toàn, chúng ta có trọng lượng lớp ẩn $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ và thiên vị $\mathbf{b}^{(1)} \in \mathbb{R}^{1 \times h}$ và trọng lượng lớp đầu ra $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ và thiên vị $\mathbf{b}^{(2)} \in \mathbb{R}^{1 \times q}$. Chính thức, chúng tôi tính toán các đầu ra $\mathbf{O} \in \mathbb{R}^{n \times q}$ của MLP một lớp ẩn như sau:

$$\begin{aligned} \mathbf{H} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{5.1.1}$$

Lưu ý rằng sau khi thêm lớp ẩn, mô hình của chúng tôi bây giờ yêu cầu chúng tôi theo dõi và cập nhật các bộ tham số bổ sung. Vì vậy, những gì chúng ta đã đạt được trong trao đổi? Bạn có thể ngạc nhiên khi tìm hiểu điều đó - trong mô hình được xác định ở trên —* chúng tôi không đạt được gì cho sự cố của chúng tôi*! Lý do là đơn giản. Các đơn vị ẩn ở trên được đưa ra bởi một hàm affine của các đầu vào, và các đầu ra (pre-softmax) chỉ là một hàm affine của các đơn vị ẩn. Một hàm affine của một hàm affine tự nó là một hàm affine. Hơn nữa, mô hình tuyến tính của chúng tôi đã có khả năng đại diện cho bất kỳ chức năng affine nào.

Chúng ta có thể xem sự tương đương chính thức bằng cách chứng minh rằng đối với bất kỳ giá trị nào của trọng lượng, chúng ta chỉ có thể thu gọn lớp ẩn, mang lại một mô hình một lớp tương đương với các tham số $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ và $\mathbf{b} = \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$:

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}. \tag{5.1.2}$$

Để nhận ra tiềm năng của các kiến trúc đa lớp, chúng ta cần thêm một thành phần quan trọng: chức năng kích hoạt* phi tuyến * σ được áp dụng cho mỗi đơn vị ẩn sau khi chuyển đổi affine. Các đầu ra của các chức năng kích hoạt (ví dụ, $\sigma(\cdot)$) được gọi là *kích hoạt*. Nói chung, với các chức năng kích hoạt tại chỗ, không còn có

thể thu gọn MLP của chúng ta thành một mô hình tuyến tính:

$$\begin{aligned}\mathbf{H} &= \sigma(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}),\\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}.\end{aligned} \tag{5.1.3}$$

Vì mỗi hàng trong $\mathbf{X}$ tương ứng với một ví dụ trong minibatch, với một số lạm dụng ký hiệu, chúng tôi xác định tính phi tuyến σ để áp dụng cho các đầu vào của nó theo kiểu hàng, tức là một ví dụ tại một thời điểm. Lưu ý rằng chúng ta đã sử dụng ký hiệu cho softmax theo cùng một cách để biểu thị một hoạt động theo hàng trong Section 4.4.5. Thông thường, như trong phần này, các chức năng kích hoạt mà chúng ta áp dụng cho các lớp ẩn không chỉ đơn thuần là hàng, mà là elementwise. Điều đó có nghĩa là sau khi tính toán phần tuyến tính của lớp, chúng ta có thể tính toán từng kích hoạt mà không cần nhìn vào các giá trị được thực hiện bởi các đơn vị ẩn khác. Điều này đúng đối với hầu hết các chức năng kích hoạt.

Để xây dựng các MLP chung hơn, chúng ta có thể tiếp tục xếp chồng các lớp ẩn như vậy, ví dụ, $\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ và $\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$, một trên đỉnh khác, mang lại các mô hình biểu cảm hơn bao giờ hết.

Phổ Approximators

MLP có thể nắm bắt các tương tác phức tạp giữa các đầu vào của chúng ta thông qua các tế bào thần kinh ẩn của chúng, phụ thuộc vào các giá trị của từng đầu vào. Chúng ta có thể dễ dàng thiết kế các nút ẩn để thực hiện tính toán tùy ý, ví dụ, các phép toán logic cơ bản trên một cặp đầu vào. Hơn nữa, đối với một số lựa chọn nhất định của chức năng kích hoạt, người ta biết rằng MLPlà gần đúng phổ quát. Ngay cả với một mạng lưới một lớp ẩn, cho đủ nút (có thể vô lý nhiều), và tập hợp đúng trọng lượng, chúng ta có thể mô hình hóa bất kỳ chức năng, mặc dù thực sự học chức năng đó là phần cứng. Bạn có thể nghĩ về mạng thần kinh của bạn như là một chút giống như ngôn ngữ lập trình C. Ngôn ngữ, giống như bất kỳ ngôn ngữ hiện đại nào khác, có khả năng thể hiện bất kỳ chương trình có thể tính toán nào. Nhưng thực sự đến với một chương trình đáp ứng thông số kỹ thuật của bạn là phần khó khăn.

Hơn nữa, chỉ vì một mạng lưới một lớp ẩn *có thể* học bất kỳ chức năng nào không có nghĩa là bạn nên cố gắng giải quyết tất cả các vấn đề của mình với các mạng một lớp ẩn. Trên thực tế, chúng ta có thể xấp xỉ nhiều chức năng nhỏ gọn hơn nhiều bằng cách sử dụng các mạng sâu hơn (so với rộng hơn). Chúng tôi sẽ đề cập đến những lập luận nghiêm ngặt hơn trong các chương tiếp theo.

5.1.2 Chức năng kích hoạt

Các chức năng kích hoạt quyết định xem một tế bào thần kinh có nên được kích hoạt hay không bằng cách tính tổng trọng số và thêm thiên vị với nó. Chúng là các toán tử khác biệt để chuyển đổi tín hiệu đầu vào thành đầu ra, trong khi hầu hết chúng thêm tính phi tuyến tính. Bởi vì các chức năng kích hoạt là nền tảng cho việc học sâu, hãy để chúng tôi khảo sát ngắn gọn một số chức năng kích hoạt phổ biến.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

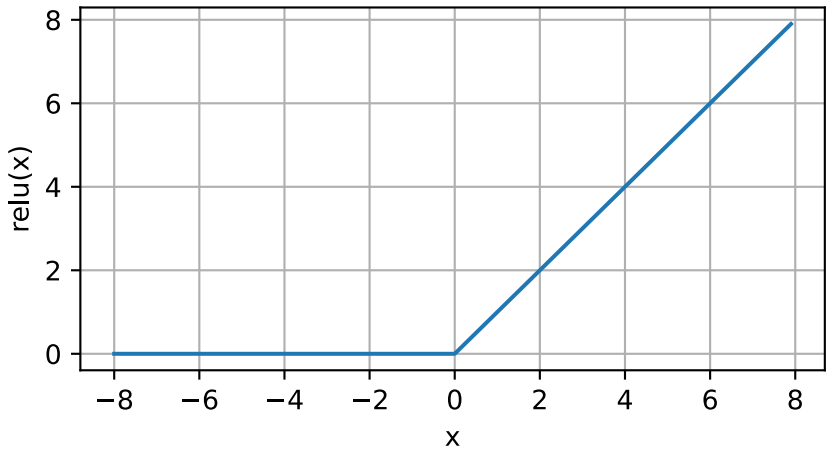
Chức năng ReLU

Sự lựa chọn phổ biến nhất, do cả sự đơn giản của việc thực hiện và hiệu suất tốt của nó trên một loạt các nhiệm vụ dự đoán, là đơn vị tuyến tính * sửa chữ* (* Relu*). ReLU cung cấp một biến đổi phi tuyến rất đơn giản. Cho một phần tử x, hàm được định nghĩa là tối đa của phần tử đó và 0:

$$\text{ReLU}(x) = \max(x, 0). \tag{5.1.4}$$

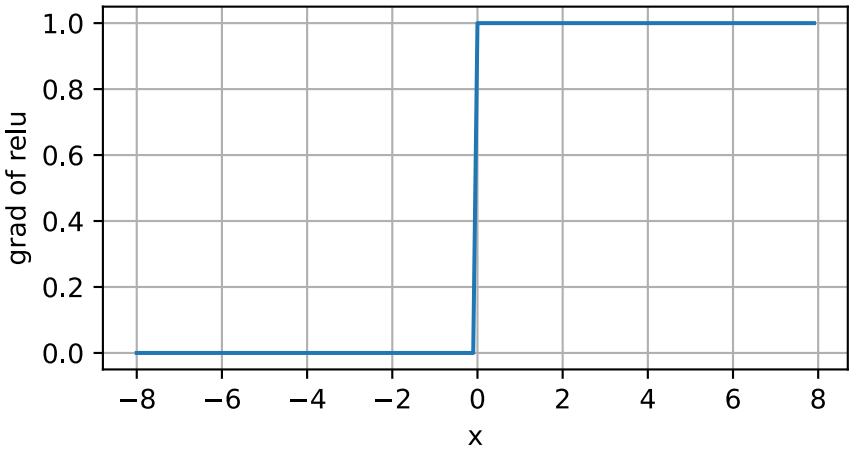
Không chính thức, hàm ReLU chỉ giữ lại các phần tử dương và loại bỏ tất cả các yếu tố âm bằng cách đặt các kích hoạt tương ứng thành 0. Để đạt được một số trực giác, chúng ta có thể vẽ hàm. Như bạn có thể thấy, chức năng kích hoạt là piecewise tuyến tính.

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Khi đầu vào là âm, đạo hàm của hàm ReLU là 0, và khi đầu vào là dương, đạo hàm của hàm ReLU là 1. Lưu ý rằng hàm ReLU không phân biệt được khi đầu vào lấy giá trị chính xác bằng 0. Trong những trường hợp này, chúng ta mặc định là đạo hàm bên trái và nói rằng đạo hàm là 0 khi đầu vào là 0. Chúng ta có thể nhận được đi với điều này bởi vì đầu vào có thể không bao giờ thực sự bằng không. Có một câu ngạn ngữ cũ rằng nếu điều kiện ranh giới tinh tế quan trọng, chúng ta có thể đang làm (* real*) toán học, chứ không phải kỹ thuật. Trí tuệ thông thường đó có thể áp dụng ở đây. Chúng tôi vẽ đạo hàm của hàm ReLU được vẽ bên dưới.

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2.5))
```



Lý do để sử dụng ReLU là các dẫn xuất của nó được cư xử đặc biệt tốt: hoặc chúng biến mất hoặc họ chỉ để cho cuộc tranh luận thông qua. Điều này làm cho tối ưu hóa hoạt động tốt hơn và nó giảm thiểu vấn đề được tài liệu tốt về độ dốc biến mất cản các phiên bản trước của mạng thần kinh (thêm về điều này sau).

Lưu ý rằng có nhiều biến thể cho hàm ReLU, bao gồm hàm ReLU* (*pReLU*) được tham số hóa (He et al., 2015). Biến thể này thêm một thuật ngữ tuyến tính vào ReLU, vì vậy một số thông tin vẫn được thông qua, ngay cả khi đối số là âm:

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \tag{5.1.5}$$

Chức năng Sigmoid

Chức năng * sigmoid* biến đổi đầu vào của nó, mà các giá trị nằm trong miền $\mathbb{R}$, để đầu ra nằm trong khoảng thời gian (0, 1) . Vì lý do đó, sigmoid thường được gọi là hàm *squashing*: nó đè bẹp bất kỳ đầu vào trong phạm vi (-inf, inf) đến một số giá trị trong phạm vi (0, 1):

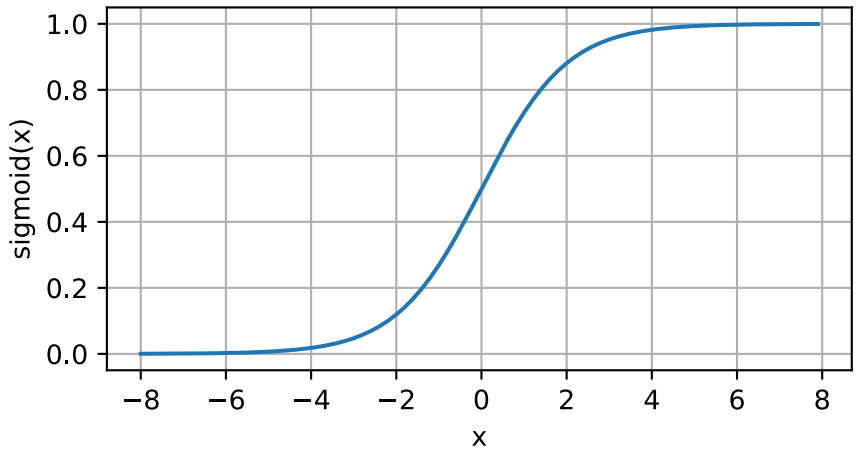
$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \tag{5.1.6}$$

Trong các mạng thần kinh sớm nhất, các nhà khoa học quan tâm đến việc mô hình hóa các tế bào thần kinh sinh học mà * cháy* hoặc * không cháy*. Do đó, những người tiên phong của lĩnh vực này, đi tất cả các con đường trở lại McCulloch và Pitt, những nhà phát minh của tế bào thần kinh nhân tạo, tập trung vào các đơn vị ngưỡng. Kích hoạt ngưỡng có giá trị 0 khi đầu vào của nó nằm dưới ngưỡng và giá trị 1 khi đầu vào vượt quá ngưỡng.

Khi sự chú ý chuyển sang học tập dựa trên gradient, chức năng sigmoid là một lựa chọn tự nhiên bởi vì nó là một xấp xỉ trơn tru, khác biệt với một đơn vị ngưỡng. Sigmoids vẫn được sử dụng rộng rãi như các hàm kích hoạt trên các đơn vị đầu ra, khi chúng ta muốn diễn giải các đầu ra như là xác suất cho các bài toán phân loại nhị phân (bạn có thể nghĩ về sigmoid như một trường hợp đặc biệt của softmax). Tuy nhiên, sigmoid chủ yếu được thay thế bằng ReLU đơn giản và dễ dàng hơn để sử dụng hầu hết trong các lớp ẩn. Trong các chương sau trên các mạng thần kinh định kỳ, chúng tôi sẽ mô tả các kiến trúc tận dụng các đơn vị sigmoid để kiểm soát luồng thông tin theo thời gian.

Dưới đây, chúng tôi vẽ hàm sigmoid. Lưu ý rằng khi đầu vào gần 0, hàm sigmoid tiếp cận một biến đổi tuyến tính.

```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```

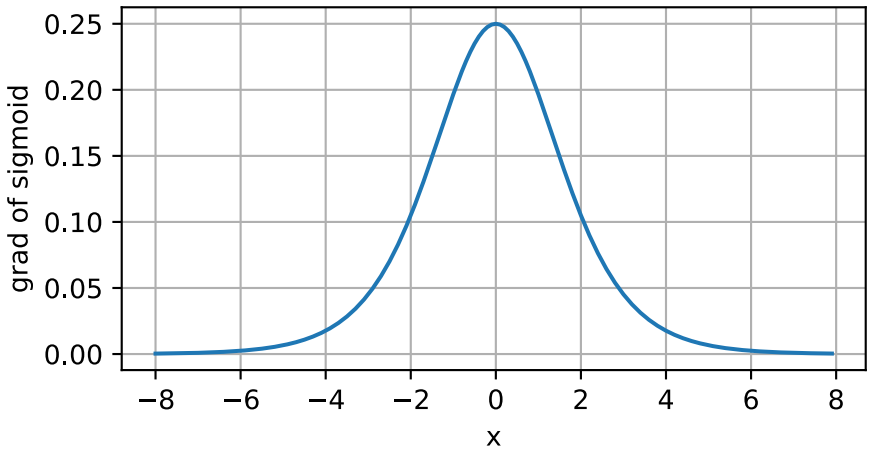


Đạo hàm của hàm sigmoid được cho bởi phương trình sau:

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))\,. \tag{5.1.7}$$

Đạo hàm của hàm sigmoid được vẽ bên dưới. Lưu ý rằng khi đầu vào là 0, đạo hàm của hàm sigmoid đạt tối đa 0,25. Khi đầu vào phân kỳ từ 0 theo một trong hai hướng, đạo hàm tiếp cận 0.

```
# Clear out previous gradients
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```



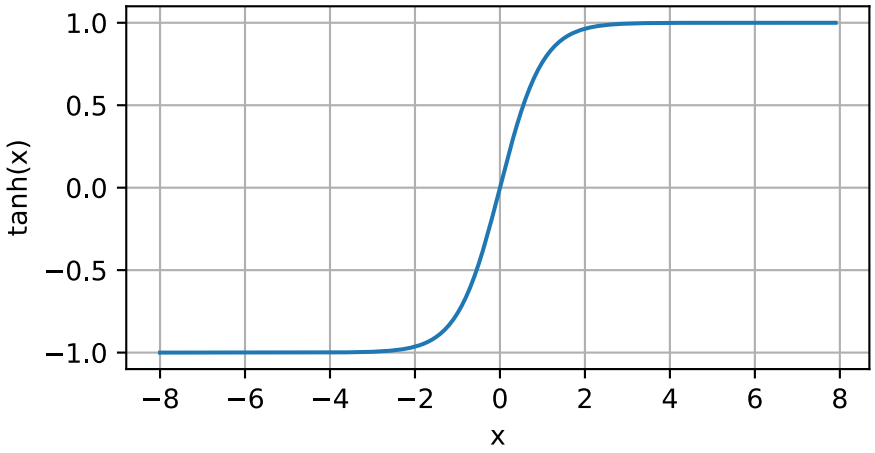
Chức năng Tanh

Giống như hàm sigmoid, hàm tanh (hyperbol tangent) cũng đè bẹp các đầu vào của nó, biến chúng thành các phần tử trong khoảng thời gian giữa -1 và 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{5.1.8}$$

Chúng tôi vẽ hàm tanh dưới đây. Lưu ý rằng khi đầu vào gần 0, hàm tanh tiếp cận một phép biến đổi tuyến tính. Mặc dù hình dạng của hàm tương tự như của hàm sigmoid, hàm tánh thể hiện tính đối xứng điểm về nguồn gốc của hệ tọa độ.

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```

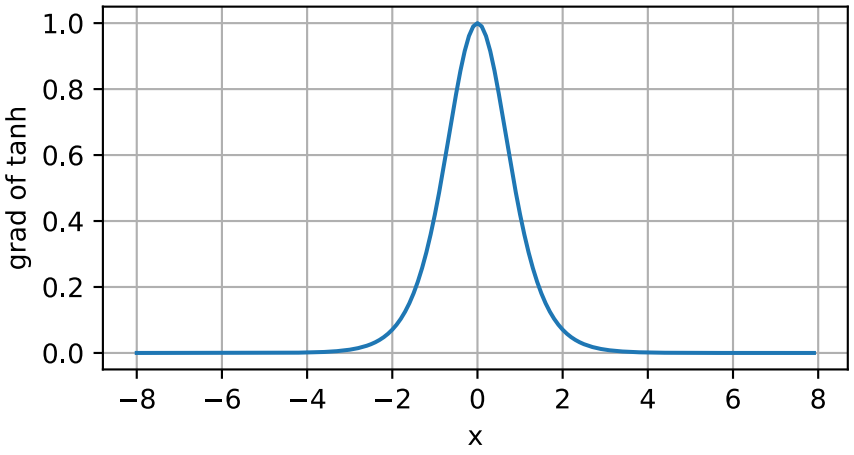


Đạo hàm của hàm tánh là:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x). \tag{5.1.9}$$

Đạo hàm của hàm tánh được vẽ bên dưới. Khi đầu vào gần 0, đạo hàm của hàm tanh tiếp cận tối đa là 1. Và như chúng ta đã thấy với hàm sigmoid, khi đầu vào di chuyển ra khỏi 0 theo một trong hai hướng, đạo hàm của hàm tanh tiếp cận 0.

```
# Clear out previous gradients.
x.grad.data.zero_()
y.backward(torch.ones_like(x),retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



Tóm lại, bây giờ chúng ta biết làm thế nào để kết hợp phi tuyến tính để xây dựng kiến trúc mạng thần kinh đa lớp biểu cảm. Như một lưu ý phụ, kiến thức của bạn đã đặt bạn trong chỉ huy của một bộ công cụ tương tự cho một học viên khoảng 1990. Theo một số cách nào đó, bạn có lợi thế hơn bất kỳ ai làm việc trong những năm 1990, bởi vì bạn có thể tận dụng các khuôn khổ deep learning mã nguồn mở mạnh mẽ để xây dựng các mô hình nhanh chóng, chỉ sử dụng một vài dòng mã. Trước đây, đào tạo các mạng này đòi hỏi các nhà nghiên cứu phải mã hóa hàng ngàn dòng C và Fortran.

5.1.3 Tóm tắt

- MLP thêm một hoặc nhiều lớp ẩn được kết nối hoàn toàn giữa các lớp đầu ra và đầu vào và biến đổi đầu ra của lớp ẩn thông qua chức năng kích hoạt.
- Các chức năng kích hoạt được sử dụng phổ biến bao gồm hàm ReLU, hàm sigmoid và hàm tanh.

5.1.4 Bài tập

1. Tính toán đạo hàm của hàm kích hoạt pReLU.
2. Cho thấy một MLP chỉ sử dụng ReLU (hoặc pReLU) xây dựng một hàm tuyến tính piecewise liên tục.
3. Cho thấy rằng $\tanh(x) + 1 = 2 \operatorname{sigmoid}(2x)$.
4. Giả sử rằng chúng ta có một phi tuyến tính áp dụng cho một minibatch tại một thời điểm. Những loại vấn đề nào bạn mong đợi điều này gây ra?

Discussions[61]

[61] https://discuss.d2l.ai/t/91

5.2 Thực hiện các Perceptrons đa lớp từ đầu

Bây giờ chúng ta đã đặc trưng các nhận thức đa lớp (MLPs) về mặt toán học, chúng ta hãy cố gắng thực hiện một chính mình. Để so sánh với các kết quả trước đây của chúng tôi đạt được với hồi quy softmax (Section 4.6), chúng tôi sẽ tiếp tục làm việc với tập dữ liệu phân loại hình ảnh Fashion-MNIST (Section 4.5).

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

5.2.1 Khởi tạo các tham số mô hình

Nhớ lại rằng Fashion-MNIST chứa 10 lớp, và rằng mỗi hình ảnh bao gồm một $28 \times 28 = 784$ lưới giá trị điểm ảnh xám. Một lần nữa, chúng ta sẽ bỏ qua cấu trúc không gian giữa các pixel cho bây giờ, vì vậy chúng ta có thể nghĩ về điều này chỉ đơn giản là một tập dữ liệu phân loại với 784 tính năng đầu vào và 10 lớp. Để bắt đầu, chúng ta sẽ triển khai MLP với một lớp ẩn và 256 đơn vị ẩn. Lưu ý rằng chúng ta có thể coi cả hai đại lượng này là siêu tham số. Thông thường, chúng ta chọn độ rộng lớp trong quyền hạn của 2, mà có xu hướng được tính toán hiệu quả vì cách bộ nhớ được phân bổ và giải quyết trong phần cứng.

Một lần nữa, chúng tôi sẽ đại diện cho các thông số của chúng tôi với một số hàng chục. Lưu ý rằng * cho mỗi lớp*, chúng ta phải theo dõi một ma trận trọng lượng và một vector thiên vị. Như mọi khi, chúng tôi phân bổ bộ nhớ cho gradient của sự mất mát đối với các tham số này.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(
    num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
    num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

5.2.2 Chức năng kích hoạt

Để đảm bảo rằng chúng ta biết mọi thứ hoạt động như thế nào, chúng ta sẽ triển khai kích hoạt ReLU bằng cách sử dụng hàm tối đa thay vì gọi trực tiếp hàm `relu` tích hợp sẵn.

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

5.2.3 Mô hình

Bởi vì chúng ta đang bỏ qua cấu trúc không gian, chúng ta `reshape` mỗi hình ảnh hai chiều thành một vector phẳng có chiều dài `num_inputs`. Cuối cùng, chúng tôi triển khai mô hình của chúng tôi chỉ với một vài dòng code.

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # Here '@' stands for matrix multiplication
    return (H@W2 + b2)
```

5.2.4 Chức năng mất

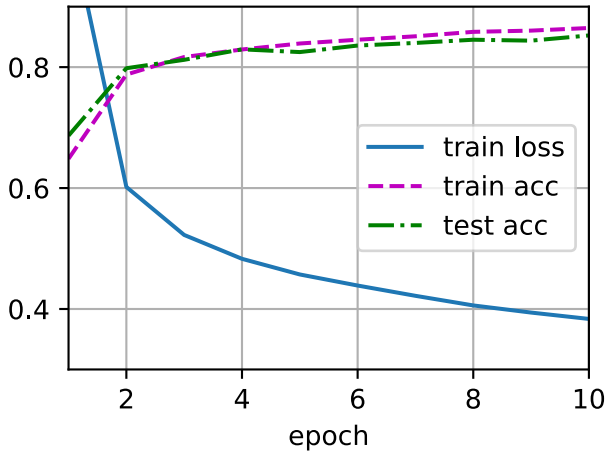
Để đảm bảo tính ổn định số, và bởi vì chúng tôi đã triển khai chức năng softmax từ đầu (Section 4.6), chúng tôi tận dụng chức năng tích hợp từ các API cấp cao để tính toán sự mất mát softmax và cross-entropy. Nhớ lại cuộc thảo luận trước đó của chúng tôi về những phức tạp này trong Section 4.7.2. Chúng tôi khuyến khích người đọc quan tâm kiểm tra mã nguồn cho chức năng mất mát để đào sâu kiến thức của họ về chi tiết thực hiện.

```
loss = nn.CrossEntropyLoss(reduction='none')
```

5.2.5 Đào tạo

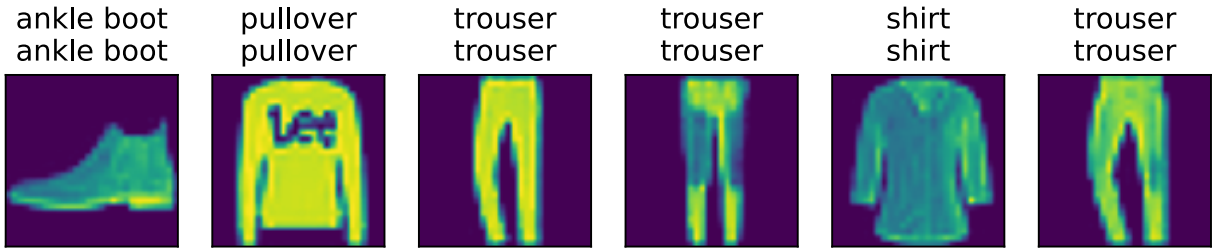
May mắn thay, vòng đào tạo cho MLP hoàn toàn giống như đối với hồi quy softmax. Tận dụng gói `d2l` một lần nữa, chúng tôi gọi hàm `train_ch3` (xem Section 4.6), đặt số epochs thành 10 và tỷ lệ học tập là 0.1.

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```



Để đánh giá mô hình đã học, chúng tôi áp dụng nó trên một số dữ liệu thử nghiệm.

```
d2l.predict_ch3(net, test_iter)
```



5.2.6 Tóm tắt

- Chúng tôi thấy rằng việc thực hiện một MLP đơn giản là dễ dàng, ngay cả khi thực hiện thủ công.
- Tuy nhiên, với một số lượng lớn các lớp, việc thực hiện MLP từ đầu vẫn có thể trở nên lộn xộn (ví dụ, đặt tên và theo dõi các thông số của mô hình của chúng tôi).

5.2.7 Bài tập

1. Thay đổi giá trị của hyperparameters `num_hiddens` và xem siêu tham số này ảnh hưởng như thế nào đến kết quả của bạn. Xác định giá trị tốt nhất của siêu tham số này, giữ cho tất cả những người khác không đổi.
2. Hãy thử thêm một layer ẩn bổ sung để xem nó ảnh hưởng đến kết quả như thế nào.
3. Làm thế nào để thay đổi tỷ lệ học tập làm thay đổi kết quả của bạn? Sửa chữa kiến trúc mô hình và các siêu tham số khác (bao gồm số thời đại), tỷ lệ học tập nào mang lại cho bạn kết quả tốt nhất?
4. Kết quả tốt nhất bạn có thể nhận được bằng cách tối ưu hóa tất cả các siêu tham số (tốc độ học tập, số lượng kỷ nguyên, số lớp ẩn, số lượng đơn vị ẩn trên mỗi lớp) cùng nhau?
5. Mô tả lý do tại sao việc đối phó với nhiều siêu tham số khó khăn hơn nhiều.
6. Chiến lược thông minh nhất bạn có thể nghĩ đến để cấu trúc tìm kiếm trên nhiều siêu tham số là gì?

Discussions[62]

5.3 Thực hiện ngắn gọn của Multilayer Perceptrons

Như bạn có thể mong đợi, bằng cách dựa vào các API cấp cao, chúng tôi có thể triển khai MLP thậm chí còn ngắn gọn hơn.

```
import torch
from torch import nn
from d2l import torch as d2l
```

[62] https://discuss.d2l.ai/t/93

5.3.1 Mô hình

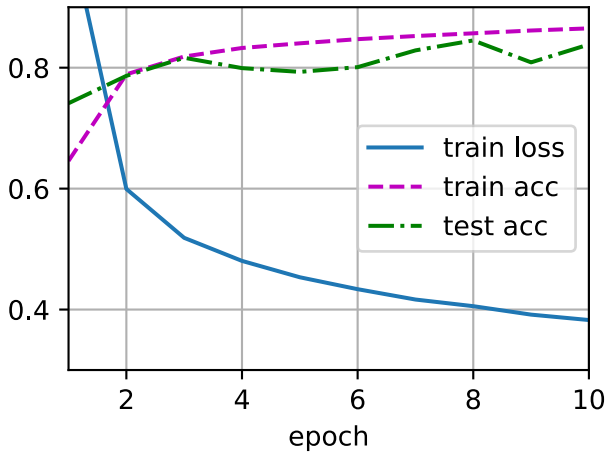
So với việc thực hiện súc tích của chúng tôi về triển khai hồi quy softmax (Section 4.7), sự khác biệt duy nhất là chúng tôi thêm *hai* lớp được kết nối hoàn toàn (trước đây, chúng tôi đã thêm * một*). Đầu tiên là lớp ẩn của chúng tôi, chứa 256 đơn vị ẩn và áp dụng chức năng kích hoạt ReLU. Thứ hai là lớp đầu ra của chúng tôi.

```
net = nn.Sequential(nn.Flatten(),
                    nn.Linear(784, 256),
                    nn.ReLU(),
                    nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

Vòng đào tạo chính xác giống như khi chúng tôi thực hiện hồi quy softmax. Mô đun này cho phép chúng ta tách các vấn đề liên quan đến kiến trúc mô hình khỏi các cân nhắc trực giao.

```
batch_size, lr, num_epochs = 256, 0.1, 10
loss = nn.CrossEntropyLoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=lr)
```

```
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



5.3.2 Tóm tắt

- Sử dụng API cấp cao, chúng ta có thể triển khai MLP chính xác hơn nhiều.
- Đối với bài toán phân loại tương tự, việc thực hiện một MLP giống như của hồi quy softmax ngoại trừ các lớp ẩn bổ sung có hàm kích hoạt.

5.3.3 Bài tập

1. Hãy thử thêm các số lớp ẩn khác nhau (bạn cũng có thể sửa đổi tốc độ học tập). Cài đặt nào hoạt động tốt nhất?
2. Hãy thử các chức năng kích hoạt khác nhau. Cái nào hoạt động tốt nhất?
3. Hãy thử các sơ đồ khác nhau để khởi tạo các trọng lượng. Phương pháp nào hoạt động tốt nhất?

Discussions[63]

5.4 Lựa chọn mô hình, Underfitting, và Overfitting

Là các nhà khoa học máy học, mục tiêu của chúng tôi là khám phá * mẫu*. Nhưng làm thế nào chúng ta có thể chắc chắn rằng chúng tôi đã thực sự phát hiện ra một mô hình * general* và không chỉ đơn giản là ghi nhớ dữ liệu của chúng tôi? Ví dụ, hãy tưởng tượng rằng chúng ta muốn săn lùng các mô hình giữa các dấu hiệu di truyền liên kết bệnh nhân với tình trạng sa sút trí tuệ của họ, nơi các nhãn được rút ra từ bộ $\{\text{dementia}, \text{mild cognitive impairment}, \text{healthy}\}$. Bởi vì gen của mỗi người xác định chúng một cách duy nhất (bỏ qua các anh chị em giống hệt nhau), có thể ghi nhớ toàn bộ tập dữ liệu.

Chúng tôi không muốn mô hình của chúng tôi nói *"Đó là Bob! Tôi nhớ anh ấy! Anh ấy bị mất trí nhớ!" * Lý do tại sao đơn giản. Khi chúng tôi triển khai mô hình trong tương lai, chúng tôi sẽ gặp những bệnh nhân mà mô hình chưa từng thấy trước đây. Dự đoán của chúng tôi sẽ chỉ hữu ích nếu mô hình của chúng tôi đã thực sự phát hiện ra một mô hình * general*.

Để tóm lại chính thức hơn, mục tiêu của chúng tôi là khám phá các mô hình nắm bắt được sự đều đặn trong dân số cơ bản mà từ đó bộ đào tạo của chúng tôi được rút ra. Nếu chúng ta thành công trong nỗ lực này, thì chúng ta có thể đánh giá thành công rủi ro ngay cả đối với những cá nhân mà chúng ta chưa bao giờ gặp phải trước đây. Vấn đề này—làm thế nào để khám phá các mẫu mà * tổng hợp* — là vấn đề cơ bản của học máy.

Nguy hiểm là khi chúng tôi đào tạo các mô hình, chúng tôi chỉ truy cập một mẫu dữ liệu nhỏ. Các tập dữ liệu hình ảnh công cộng lớn nhất chứa khoảng một triệu hình ảnh. Thường xuyên hơn, chúng ta phải học hỏi từ chỉ hàng ngàn hoặc hàng chục ngàn ví dụ dữ liệu. Trong một hệ thống bệnh viện lớn, chúng tôi có thể truy cập hàng trăm ngàn hồ sơ y tế. Khi làm việc với các mẫu hữu hạn, chúng tôi gặp rủi ro rằng chúng tôi có thể phát hiện ra các hiệp hội rõ ràng hóa ra không giữ khi chúng tôi thu thập nhiều dữ liệu hơn.

Hiện tượng phù hợp với dữ liệu đào tạo của chúng tôi chặt chẽ hơn chúng tôi phù hợp với phân phối cơ bản được gọi là * overfitting*, và các kỹ thuật được sử dụng để chống lại quá mức được gọi là *regarization*. Trong các phần trước, bạn có thể đã quan sát thấy hiệu ứng này trong khi thử nghiệm với bộ dữ liệu Fashion-MNIST. Nếu bạn thay đổi cấu trúc mô hình hoặc các siêu tham số trong quá trình thử nghiệm, bạn có thể nhận thấy rằng với đủ tế bào thần kinh, lớp và kỷ nguyên đào tạo, mô hình cuối cùng có thể đạt được độ chính xác hoàn hảo trên bộ đào tạo, ngay cả khi độ chính xác trên dữ liệu thử nghiệm xấu đi.

[63] https://discuss.d2l.ai/t/95

5.4.1 Lỗi đào tạo và lỗi tổng quát

Để thảo luận về hiện tượng này một cách chính thức hơn, chúng ta cần phân biệt giữa lỗi đào tạo và lỗi tổng quát hóa. Lỗi*training lỗi* là lỗi của mô hình của chúng tôi như được tính trên tập dữ liệu đào tạo, trong khi lỗi tổng quát * là kỳ vọng về lỗi mô hình của chúng tôi là chúng tôi áp dụng nó vào một dòng vô hạn các ví dụ dữ liệu bổ sung được rút ra từ cùng một phân phối dữ liệu cơ bản như mẫu ban đầu của chúng tôi.

Vấn đề, chúng ta không bao giờ có thể tính toán chính xác lỗi tổng quát hóa. Đó là bởi vì dòng dữ liệu vô hạn là một đối tượng tưởng tượng. Trong thực tế, chúng ta phải * ước tính* lỗi tổng quát bằng cách áp dụng mô hình của chúng tôi vào một tập kiểm tra độc lập cấu thành một lựa chọn ngẫu nhiên các ví dụ dữ liệu đã được giữ lại từ bộ đào tạo của chúng tôi.

Ba thí nghiệm suy nghĩ sau đây sẽ giúp minh họa tình huống này tốt hơn. Hãy xem xét một sinh viên đại học đang cố gắng chuẩn bị cho kỳ thi cuối cùng của mình. Một sinh viên siêng năng sẽ cố gắng luyện tập tốt và kiểm tra khả năng của mình bằng cách sử dụng các kỳ thi từ những năm trước. Tuy nhiên, làm tốt trong các kỳ thi trong quá khứ là không có gì đảm bảo rằng anh ấy sẽ xuất sắc khi nó quan trọng. Ví dụ, học sinh có thể cố gắng chuẩn bị bằng cách rote học câu trả lời cho các câu hỏi thi. Điều này đòi hỏi học sinh phải ghi nhớ nhiều thứ. Cô ấy thậm chí có thể nhớ câu trả lời cho các kỳ thi trong quá khứ một cách hoàn hảo. Một sinh viên khác có thể chuẩn bị bằng cách cố gắng hiểu lý do để đưa ra câu trả lời nhất định. Trong hầu hết các trường hợp, học sinh sau này sẽ làm tốt hơn nhiều.

Tương tự như vậy, hãy xem xét một mô hình chỉ đơn giản là sử dụng bảng tra cứu để trả lời các câu hỏi. Nếu tập hợp các đầu vào cho phép là rời rạc và hợp lý nhỏ, thì có lẽ sau khi xem * nhiều ví dụ đào tạo, cách tiếp cận này sẽ hoạt động tốt. Tuy nhiên, mô hình này không có khả năng làm tốt hơn so với đoán ngẫu nhiên khi đối mặt với các ví dụ mà nó chưa từng thấy trước đây. Trong thực tế, không gian đầu vào quá lớn để ghi nhớ các câu trả lời tương ứng với mọi đầu vào có thể tưởng tượng được. Ví dụ: hãy xem xét các hình ảnh 28×28 đen và trắng. Nếu mỗi pixel có thể lấy một trong số 256 giá trị thang màu xám, thì có 256^{784} hình ảnh có thể. Điều đó có nghĩa là có những hình ảnh có kích thước thu nhỏ có độ phân giải thấp hơn nhiều so với các nguyên tử trong vũ trụ. Ngay cả khi chúng ta có thể gặp phải dữ liệu như vậy, chúng ta không bao giờ có thể đủ khả năng để lưu trữ bảng tra cứu.

Cuối cùng, hãy xem xét vấn đề cố gắng phân loại kết quả của việc ném tiền xu (lớp 0: đầu, lớp 1: đuôi) dựa trên một số tính năng theo ngữ cảnh có thể có sẵn. Giả sử rằng đồng xu là công bằng. Không có vấn đề gì thuật toán chúng tôi đưa ra, lỗi tổng quát sẽ luôn là $\frac{1}{2}$. Tuy nhiên, đối với hầu hết các thuật toán, chúng ta nên mong đợi lỗi đào tạo của chúng ta thấp hơn đáng kể, tùy thuộc vào may mắn của trận hòa, ngay cả khi chúng ta không có bất kỳ tính năng nào! Xem xét tập dữ liệu {0, 1, 1, 1, 0, 1}. Thuật toán ít tính năng của chúng tôi sẽ phải rơi trở lại luôn dự đoán các lớp *đa số*, xuất hiện từ mẫu giới hạn của chúng tôi là *1*. Trong trường hợp này, mô hình luôn dự đoán lớp 1 sẽ phát sinh lỗi $\frac{1}{3}$, tốt hơn đáng kể so với lỗi tổng quát của chúng tôi. Khi chúng ta tăng lượng dữ liệu, xác suất phần đầu sẽ đi chệch đáng kể so với $\frac{1}{2}$ giảm và lỗi đào tạo của chúng tôi sẽ phù hợp với lỗi tổng quát.

Lý thuyết học thống kê

Vì khái quát hóa là vấn đề cơ bản trong học máy, bạn có thể không ngạc nhiên khi biết rằng nhiều nhà toán học và nhà lý luận đã cống hiến cuộc sống của họ để phát triển các lý thuyết chính thức để mô tả hiện tượng này. Trong [định lý cùng tên] của họ (https://en.wikipedia.org/wiki/Glivenko%E2%80%93Cantelli_theorem), Glivenko và Cantelli bắt nguồn tốc độ mà tại đó lỗi đào tạo hội tụ đến lỗi tổng quát hóa. Trong một loạt các bài báo seminal, Vapnik và Chervonenkis[64] đã mở rộng lý thuyết này sang các lớp hàm tổng quát hơn. Công trình này đặt nền tảng của lý thuyết học thống kê.

[64] https://en.wikipedia.org/wiki/Vapnik%E2%80%93Chervonenkis_theory

Trong cài đặt học tập được giám sát tiêu chuẩn, mà chúng tôi đã giải quyết cho đến bây giờ và sẽ gắn bó với trong suốt hầu hết cuốn sách này, chúng tôi giả định rằng cả dữ liệu đào tạo và dữ liệu thử nghiệm đều được rút ra* độc lập* từ các bản phân phối * giống hệt nhau*. Điều này thường được gọi là giả định *i.d., có nghĩa là quá trình lấy mẫu dữ liệu của chúng tôi không có bộ nhớ. Nói cách khác, ví dụ thứ hai được vẽ và vẽ thứ ba không tương quan hơn mẫu thứ hai và hai triệu được vẽ.

Trở thành một nhà khoa học máy học giỏi đòi hỏi phải suy nghĩ nghiêm túc, và bạn đã nên chọc lỗ hổng trong giả định này, đưa ra những trường hợp phổ biến mà giả định thất bại. Điều gì sẽ xảy ra nếu chúng ta đào tạo một dự báo rủi ro tử vong trên dữ liệu thu thập từ bệnh nhân tại Trung tâm Y tế UCSF và áp dụng nó cho bệnh nhân tại Bệnh viện Đa khoa Massachusetts? Những bản phân phối này chỉ đơn giản là không giống hệt nhau. Hơn nữa, rút thăm có thể tương quan trong thời gian. Điều gì sẽ xảy ra nếu chúng ta phân loại các chủ đề của Tweets? Chu kỳ tin tức sẽ tạo ra sự phụ thuộc thời gian trong các chủ đề đang được thảo luận, vi phạm bất kỳ giả định nào về sự độc lập.

Đôi khi chúng ta có thể thoát khỏi những vi phạm nhỏ của giả định i.i.d. và các mô hình của chúng tôi sẽ tiếp tục hoạt động tốt đáng kể. Rốt cuộc, gần như mọi ứng dụng trong thế giới thực đều liên quan đến ít nhất một số vi phạm nhỏ đối với giả định i.i..d., nhưng chúng tôi có nhiều công cụ hữu ích cho các ứng dụng khác nhau như nhận dạng khuôn mặt, nhận dạng giọng nói và dịch ngôn ngữ.

Các vi phạm khác chắc chắn sẽ gây rắc rối. Hãy tưởng tượng, ví dụ, nếu chúng ta cố gắng đào tạo một hệ thống nhận dạng khuôn mặt bằng cách đào tạo nó độc quyền trên sinh viên đại học và sau đó muốn triển khai nó như một công cụ để theo dõi lão khoa trong một dân số viện dưỡng lão. Điều này không có khả năng làm việc tốt vì sinh viên đại học có xu hướng trông khác biệt đáng kể so với người cao tuổi.

Trong các chương tiếp theo, chúng tôi sẽ thảo luận về các vấn đề phát sinh từ vi phạm giả định i.i.d. Hiện tại, thậm chí lấy giả định i.i.d. cho phép, hiểu khái quát hóa là một vấn đề ghê gớm. Hơn nữa, làm sáng tỏ các nền tảng lý thuyết chính xác có thể giải thích tại sao các mạng thần kinh sâu khái quát hóa cũng như chúng tiếp tục làm suy nghĩ vĩ đại nhất trong lý thuyết học tập.

Khi chúng tôi đào tạo các mô hình của mình, chúng tôi cố gắng tìm kiếm một chức năng phù hợp với dữ liệu đào tạo cũng như có thể. Nếu chức năng linh hoạt đến mức nó có thể bắt kịp các mẫu giả dễ dàng như các liên kết thực sự, thì nó có thể thực hiện * quá tốt* mà không tạo ra một mô hình khái quát hóa tốt với dữ liệu không nhìn thấy. Đây chính xác là những gì chúng ta muốn tránh hoặc ít nhất là kiểm soát. Nhiều kỹ thuật trong học sâu là heuristics và thủ thuật nhằm bảo vệ chống lại quá mức.

Độ phức tạp của mô hình

Khi chúng tôi có các mô hình đơn giản và dữ liệu phong phú, chúng tôi hy vọng lỗi tổng quát hóa giống với lỗi đào tạo. Khi chúng tôi làm việc với các mô hình phức tạp hơn và ít ví dụ hơn, chúng tôi hy vọng lỗi đào tạo sẽ đi xuống nhưng khoảng cách tổng quát sẽ phát triển. Điều chính xác cấu thành sự phức tạp của mô hình là một vấn đề phức tạp. Nhiều yếu tố chi phối liệu một mô hình sẽ khái quát hóa tốt hay không. Ví dụ, một mô hình có nhiều tham số hơn có thể được coi là phức tạp hơn. Một mô hình có tham số có thể lấy một phạm vi giá trị rộng hơn có thể phức tạp hơn. Thông thường với các mạng thần kinh, chúng tôi nghĩ về một mô hình có nhiều lần lặp đào tạo phức tạp hơn và một đối tượng để * stopping sớm* (ít lặp lại đào tạo hơn) là ít phức tạp hơn.

Có thể khó so sánh sự phức tạp giữa các thành viên của các lớp mô hình khác nhau đáng kể (ví dụ, cây quyết định so với mạng thần kinh). Hiện tại, một quy tắc đơn giản của ngón tay cái là khá hữu ích: một mô hình có thể dễ dàng giải thích các sự kiện tùy ý là những gì các nhà thống kê xem là phức tạp, trong khi một trong đó chỉ có một sức mạnh biểu cảm hạn chế nhưng vẫn quản lý để giải thích tốt dữ liệu có lẽ là gần gũi hơn với sự thật. Trong triết học, điều này có liên quan chặt chẽ đến tiêu chí Popper về tính giả mạo của một lý thuyết khoa học: một lý thuyết là tốt nếu nó phù hợp với dữ liệu và nếu có những thử nghiệm cụ thể có thể được sử dụng để bác bỏ nó. Điều này rất quan trọng vì tất cả các ước tính thống kê là *bài hoc*, tức là, chúng tôi ước

tính sau khi chúng tôi quan sát các sự kiện, do đó dễ bị tổn thương bởi sự sai lầm liên quan. Hiện tại, chúng tôi sẽ đặt triết lý sang một bên và dính vào các vấn đề hữu hình hơn.

Trong phần này, để cung cấp cho bạn một số trực giác, chúng ta sẽ tập trung vào một vài yếu tố có xu hướng ảnh hưởng đến tính tổng quát của một lớp model:

1. Số lượng các thông số có thể điều chỉnh. Khi số lượng các thông số có thể điều chỉnh, đôi khi được gọi là * độ tự do*, lớn, các mô hình có xu hướng dễ bị quá mức hơn.
2. Các giá trị được thực hiện bởi các tham số. Khi trọng lượng có thể mất một phạm vi rộng hơn các giá trị, các mô hình có thể dễ bị quá mức hơn.
3. Số lượng ví dụ đào tạo. Nó rất dễ dàng để overfit một tập dữ liệu chỉ chứa một hoặc hai ví dụ ngay cả khi mô hình của bạn là đơn giản. Nhưng quá nhiều tập dữ liệu với hàng triệu ví dụ đòi hỏi một mô hình cực kỳ linh hoạt.

5.4.2 Lựa chọn mô hình

Trong học máy, chúng tôi thường chọn mô hình cuối cùng của mình sau khi đánh giá một số mô hình ứng cử viên. Quá trình này được gọi là *lựa chọn mô hình*. Đôi khi các mô hình có thể so sánh về cơ bản khác nhau về bản chất (nói, cây quyết định so với mô hình tuyến tính). Vào những thời điểm khác, chúng tôi đang so sánh các thành viên của cùng một lớp mô hình đã được đào tạo với các cài đặt siêu tham số khác nhau.

Ví dụ, với MLP, chúng ta có thể muốn so sánh các mô hình với các số lớp ẩn khác nhau, số lượng đơn vị ẩn khác nhau và các lựa chọn khác nhau của các chức năng kích hoạt được áp dụng cho mỗi lớp ẩn. Để xác định tốt nhất trong số các mô hình ứng viên của chúng tôi, chúng tôi thường sẽ sử dụng một tập dữ liệu xác thực.

Validation Dataset

Về nguyên tắc, chúng ta không nên chạm vào bộ thử nghiệm của mình cho đến khi chúng tôi đã chọn tất cả các siêu tham số của mình. Nếu chúng tôi sử dụng dữ liệu thử nghiệm trong quá trình lựa chọn mô hình, có một rủi ro rằng chúng tôi có thể overfit dữ liệu thử nghiệm. Sau đó, chúng tôi sẽ gặp rắc rối nghiêm trọng. Nếu chúng tôi vượt quá dữ liệu đào tạo của mình, luôn có đánh giá về dữ liệu thử nghiệm để giữ cho chúng tôi trung thực. Nhưng nếu chúng ta vượt quá dữ liệu thử nghiệm, làm thế nào chúng ta sẽ biết?

Do đó, chúng ta không bao giờ nên dựa vào dữ liệu thử nghiệm để lựa chọn mô hình. Tuy nhiên, chúng tôi không thể chỉ dựa vào dữ liệu đào tạo để lựa chọn mô hình vì chúng tôi không thể ước tính lỗi tổng quát hóa trên dữ liệu mà chúng tôi sử dụng để đào tạo mô hình.

Trong các ứng dụng thực tế, hình ảnh trở nên muddier. Mặc dù lý tưởng nhất là chúng tôi sẽ chỉ chạm vào dữ liệu thử nghiệm một lần, để đánh giá mô hình tốt nhất hoặc so sánh một số lượng nhỏ các mô hình với nhau, dữ liệu thử nghiệm trong thế giới thực hiếm khi bị loại bỏ chỉ sau một lần sử dụng. Chúng ta hiếm khi có thể đủ khả năng một bộ thử nghiệm mới cho mỗi vòng thí nghiệm.

Thực tiễn phổ biến để giải quyết vấn đề này là chia dữ liệu của chúng tôi ba cách, kết hợp một tập dữ liệu xác thức* (hoặc * bộ xác thực *) ngoài các tập dữ liệu đào tạo và kiểm tra. Kết quả là một thực hành âm u trong đó ranh giới giữa xác nhận và dữ liệu thử nghiệm đáng lo ngại một cách mơ hồ. Trừ khi được nêu rõ ràng khác, trong các thí nghiệm trong cuốn sách này, chúng tôi thực sự đang làm việc với những gì nên được gọi là dữ liệu đào tạo và dữ liệu xác nhận, không có bộ kiểm tra thực sự. Do đó, độ chính xác được báo cáo trong mỗi thí nghiệm của cuốn sách thực sự là độ chính xác xác thực chứ không phải là độ chính xác của bộ thử nghiệm thực sự.

K-Xác định chéo Fold

Khi dữ liệu đào tạo khan hiếm, chúng ta thậm chí có thể không đủ khả năng để giữ đủ dữ liệu để tạo thành một bộ xác nhận thích hợp. Một giải pháp phổ biến cho vấn đề này là sử dụng K* -fold cross-valation*. Tại đây, dữ liệu đào tạo ban đầu được chia thành K các tập con không chồng chéo. Sau đó đào tạo mô hình và xác nhận được thực hiện K lần, mỗi lần đào tạo trên $K - 1$ tập con và xác nhận trên một tập con khác (tập hợp không được sử dụng để đào tạo trong vòng đó). Cuối cùng, các lỗi đào tạo và xác nhận được ước tính bằng cách trung bình trên các kết quả từ các thí nghiệm K.

5.4.3 Underfitting hoặc Overfitting?

Khi chúng tôi so sánh các lỗi đào tạo và xác nhận, chúng tôi muốn lưu ý đến hai tình huống phổ biến. Đầu tiên, chúng tôi muốn xem ra cho các trường hợp khi lỗi đào tạo và lỗi xác nhận của chúng tôi đều đáng kể nhưng có một khoảng cách nhỏ giữa chúng. Nếu mô hình không thể giảm lỗi đào tạo, điều đó có thể có nghĩa là mô hình của chúng tôi quá đơn giản (tức là không đủ biểu cảm) để nắm bắt mô hình mà chúng tôi đang cố gắng mô hình hóa. Hơn nữa, vì khoảng cách tổng quát *giữa các lỗi đào tạo và xác nhận của chúng tôi là nhỏ, chúng tôi có lý do để tin rằng chúng tôi có thể thoát khỏi một mô hình phức tạp hơn. Hiện tượng này được gọi là * underfitting*.

Mặt khác, như chúng tôi đã thảo luận ở trên, chúng tôi muốn chú ý các trường hợp khi lỗi đào tạo của chúng tôi thấp hơn đáng kể so với lỗi xác thực của chúng tôi, cho thấy nghiêm trọng * overfitting*. Lưu ý rằng overfitting không phải lúc nào cũng là một điều xấu. Đặc biệt với việc học sâu, người ta biết rằng các mô hình dự đoán tốt nhất thường hoạt động tốt hơn nhiều về dữ liệu đào tạo so với dữ liệu lưu trữ. Cuối cùng, chúng tôi thường quan tâm nhiều hơn đến lỗi xác nhận hơn là khoảng cách giữa các lỗi đào tạo và xác nhận.

Cho dù chúng ta overfit hay underfit có thể phụ thuộc cả vào sự phức tạp của mô hình của chúng tôi và kích thước của các tập dữ liệu đào tạo có sẵn, hai chủ đề mà chúng tôi thảo luận dưới đây.

Độ phức tạp của mô hình

Để minh họa một số trực giác cổ điển về độ phức tạp overfitting và mô hình, chúng tôi đưa ra một ví dụ bằng cách sử dụng đa thức. Với dữ liệu đào tạo bao gồm một tính năng duy nhất x và một nhãn có giá trị thực tương ứng y, chúng tôi cố gắng tìm đa thức của độ d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i \tag{5.4.1}$$

to estimate ước tính the labels nhãn y. Đây chỉ là một vấn đề hồi quy tuyến tính trong đó các tính năng của chúng tôi được đưa ra bởi sức mạnh của x, trọng lượng của mô hình được đưa ra bởi w_i và sự thiên vị được đưa ra bởi w_0 kể từ $x^0 = 1$ cho tất cả x. Vì đây chỉ là một bài toán hồi quy tuyến tính, chúng ta có thể sử dụng lỗi bình phương làm hàm mất mát của chúng ta.

Một hàm đa thức bậc cao hơn phức tạp hơn một hàm đa thức bậc thấp hơn, vì đa thức bậc cao hơn có nhiều tham số hơn và phạm vi lựa chọn của hàm mô hình rộng hơn. Sửa tập dữ liệu đào tạo, hàm đa thức bậc cao hơn phải luôn đạt được sai số đào tạo thấp hơn (tồi tệ nhất, bằng nhau) so với đa thức mức độ thấp hơn. Trên thực tế, bất cứ khi nào các ví dụ dữ liệu mỗi ví dụ có giá trị riêng biệt là x, một hàm đa thức có mức độ bằng với số ví dụ dữ liệu có thể phù hợp với bộ đào tạo một cách hoàn hảo. Chúng tôi hình dung mối quan hệ giữa mức độ đa thức và underfitting so với overfitting trong Fig. 5.4.1.

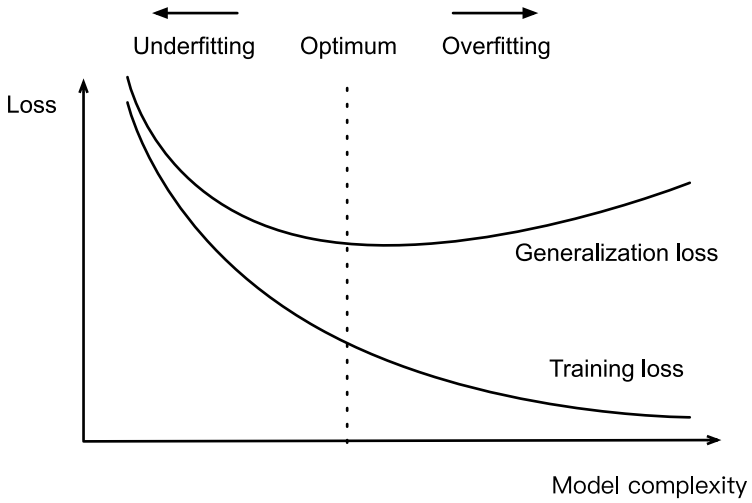


Fig. 5.4.1: Influence of model complexity on underfitting and overfitting

Kích tập dữ liệu

Sự cân nhắc lớn khác cần lưu ý là kích thước tập dữ liệu. Sửa mô hình của chúng tôi, chúng ta càng có ít mẫu trong tập dữ liệu đào tạo, chúng ta càng gặp phải nhiều khả năng (và nghiêm trọng hơn). Khi chúng tôi tăng lượng dữ liệu đào tạo, lỗi tổng quát thường giảm. Hơn nữa, nói chung, nhiều dữ liệu hơn không bao giờ bị tổn thương. Đối với một tác vụ cố định và phân phối dữ liệu, thường có mối quan hệ giữa độ phức tạp của mô hình và kích thước tập dữ liệu. Với nhiều dữ liệu hơn, chúng tôi có thể có lợi nhuận cố gắng để phù hợp với một mô hình phức tạp hơn. Vắng mặt đủ dữ liệu, các mô hình đơn giản hơn có thể khó đánh bại hơn. Đối với nhiều nhiệm vụ, deep learning chỉ vượt trội hơn các mô hình tuyến tính khi có hàng ngàn ví dụ đào tạo. Một phần, sự thành công hiện tại của deep learning nợ sự phong phú hiện tại của các tập dữ liệu khổng lồ do các công ty Internet, lưu trữ giá rẻ, thiết bị kết nối và số hóa rộng rãi của nền kinh tế.

5.4.4 Hồi quy đa thức

Bây giờ chúng ta có thể khám phá các khái niệm này tương tác bằng cách lắp đa thức với dữ liệu

```
import math
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l
```

Tạo tập dữ liệu

Đầu tiên chúng ta cần dữ liệu. Cho x, chúng tôi sẽ sử dụng đa thức khối sau đây để tạo nhãn về dữ liệu đào tạo và thử nghiệm:

$$y = 5 + 1.2x - 3.4\ fracx^2 2! + 5.6\ fracx^3 3! +\ epsilon\ textwhere\ epsilon\ sim\ mathcalN(0, 0.1^2). \tag{5.4.2}$$

Thuật ngữ tiếng ồn ϵ tuân theo sự phân bố bình thường với trung bình 0 và độ lệch chuẩn là 0,1. Để tối ưu hóa, chúng tôi thường muốn tránh các giá trị rất lớn của gradient hoặc tổn thất. Đây là lý do tại sao các tính

năng* được thay đổi lại từ x^i thành $ frac {x^i} {i!} $. Nó cho phép chúng ta tránh các giá trị rất lớn cho số mũ lớn i. Chúng tôi sẽ tổng hợp 100 mẫu mỗi mẫu cho bộ đào tạo và bộ kiểm tra.

```
max_degree = 20  # Maximum degree of the polynomial
n_train, n_test = 100, 100  # Training and test dataset sizes
true_w = np.zeros(max_degree)  # Allocate lots of empty space
true_w[0:4] = np.array([5, 1.2, -3.4, 5.6])

features = np.random.normal(size=(n_train + n_test, 1))
np.random.shuffle(features)
poly_features = np.power(features, np.arange(max_degree).reshape(1, -1))
for i in range(max_degree):
    poly_features[:, i] /= math.gamma(i + 1)  # `gamma(n)` = (n-1)!
# Shape of `labels`: (`n_train` + `n_test`,)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Một lần nữa, các monomials được lưu trữ trong `poly_features` được rescaled bởi hàm gamma, trong đó $ Gamma (n) = (n-1)! $. Hãy xem 2 mẫu đầu tiên từ tập dữ liệu được tạo ra. Giá trị 1 về mặt kỹ thuật là một tính năng, cụ thể là tính năng không đổi tương ứng với sự thiên vị.

```
# Convert from NumPy ndarrays to tensors
true_w, features, poly_features, labels = [torch.tensor(x, dtype=
    torch.float32) for x in [true_w, features, poly_features, labels]]
```

```
features[:2], poly_features[:2, :], labels[:2]
```

```
(tensor([[ 0.3741],
         [-0.2897]]),
 tensor([[ 1.0000e+00,  3.7412e-01,  6.9981e-02,  8.7270e-03,  8.1622e-04,
           6.1072e-05,  3.8080e-06,  2.0352e-07,  9.5175e-09,  3.9563e-10,
           1.4801e-11,  5.0339e-13,  1.5694e-14,  4.5163e-16,  1.2069e-17,
           3.0101e-19,  7.0382e-21,  1.5489e-22,  3.2192e-24,  6.3388e-26],
         [ 1.0000e+00, -2.8971e-01,  4.1966e-02, -4.0527e-03,  2.9352e-04,
          -1.7007e-05,  8.2120e-07, -3.3987e-08,  1.2308e-09, -3.9620e-11,
           1.1478e-12, -3.0231e-14,  7.2984e-16, -1.6265e-17,  3.3658e-19,
          -6.5006e-21,  1.1771e-22, -2.0059e-24,  3.2285e-26, -4.9228e-28]]),
 tensor([5.3687, 4.3416]))
```

Đào tạo và kiểm tra mô hình

Đầu tiên chúng ta hãy thực hiện một hàm để đánh giá sự mất mát trên một tập dữ liệu nhất định.

```
def evaluate_loss(net, data_iter, loss):  #@save
    """Evaluate the loss of a model on the given dataset."""
    metric = d2l.Accumulator(2)  # Sum of losses, no. of examples
    for X, y in data_iter:
        out = net(X)
        y = y.reshape(out.shape)
        l = loss(out, y)
        metric.add(l.sum(), l.numel())
    return metric[0] / metric[1]
```

Bây giờ xác định chức năng đào tạo.

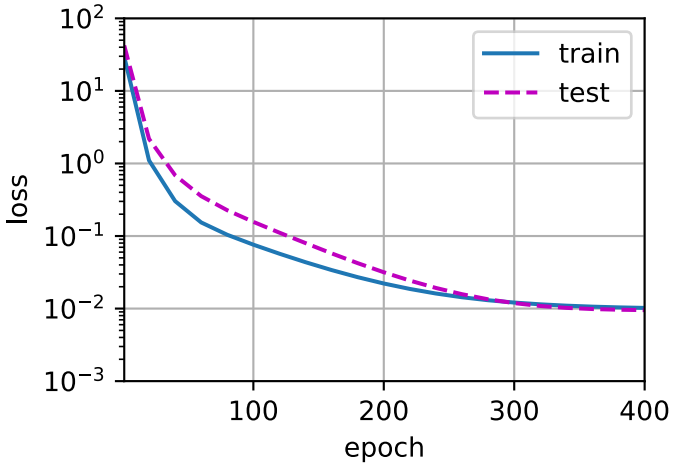
```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=400):
    loss = nn.MSELoss(reduction='none')
    input_shape = train_features.shape[-1]
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net = nn.Sequential(nn.Linear(input_shape, 1, bias=False))
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels.reshape(-1,1)),
                                batch_size)
    test_iter = d2l.load_array((test_features, test_labels.reshape(-1,1)),
                               batch_size, is_train=False)
    trainer = torch.optim.SGD(net.parameters(), lr=0.01)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss', yscale='log',
                            xlim=[1, num_epochs], ylim=[1e-3, 1e2],
                            legend=['train', 'test'])
    for epoch in range(num_epochs):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch == 0 or (epoch + 1) % 20 == 0:
            animator.add(epoch + 1, (evaluate_loss(net, train_iter, loss),
                                     evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data.numpy())
```

Phụ kiện đa thức đa thức thứ ba (Bình thường)

Chúng ta sẽ bắt đầu bằng cách sử dụng hàm đa thức bậc ba, có cùng thứ tự như hàm tạo dữ liệu. Kết quả cho thấy việc đào tạo và tổn thất thử nghiệm của mô hình này có thể được giảm một cách hiệu quả. Các tham số mô hình đã học cũng gần với các giá trị thực sự $w = [5, 1.2, -3.4, 5.6]$.

```
# Pick the first four dimensions, i.e., 1, x, x^2/2!, x^3/3! from the
# polynomial features
train(poly_features[:n_train, :4], poly_features[n_train:, :4],
      labels[:n_train], labels[n_train:])
```

```
weight: [[ 5.0026083  1.206844  -3.446796   5.5330935]]
```

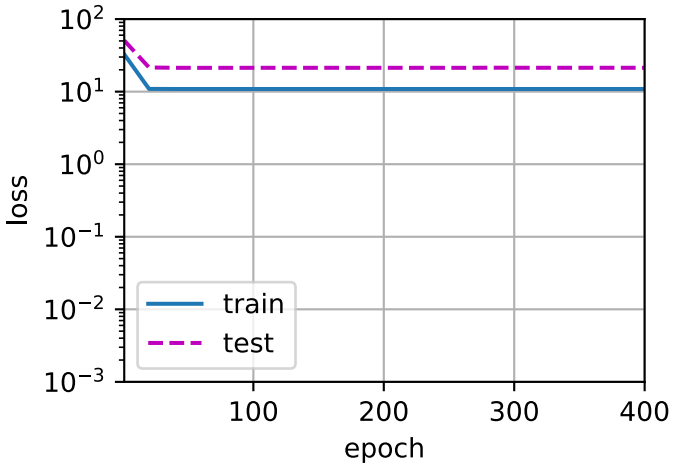


Phụ kiện chức năng tuyến tính (Underfitting)

Chúng ta hãy xem xét một chức năng tuyến tính phù hợp. Sau sự suy giảm trong thời đại đầu, việc giảm thêm sự mất mát huấn luyện của mô hình này trở nên khó khăn. Sau khi lần lặp kỷ nguyên cuối cùng đã được hoàn thành, tổn thất đào tạo vẫn còn cao. Khi được sử dụng để phù hợp với các mẫu phi tuyến (như hàm đa thức bậc ba ở đây) các mô hình tuyến tính có thể chịu trách nhiệm không phù hợp.

```
# Pick the first two dimensions, i.e., 1, x, from the polynomial features
train(poly_features[:n_train, :2], poly_features[n_train:, :2],
      labels[:n_train], labels[n_train:])
```

```
weight: [[3.147179 4.65393 ]]
```

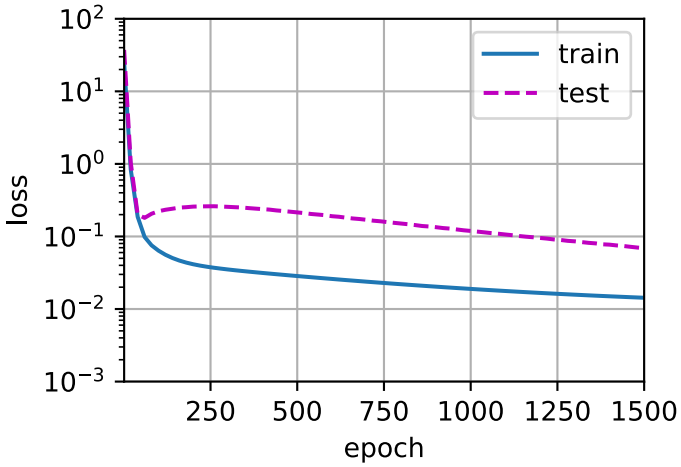


Khớp nối đa thức đa thức bậc cao hơn (Overfitting)

Bây giờ chúng ta hãy cố gắng đào tạo mô hình bằng cách sử dụng một đa thức của mức độ quá cao. Ở đây, không có đủ dữ liệu để biết rằng các hệ số mức độ cao hơn nên có giá trị gần bằng không. Do đó, mô hình quá phức tạp của chúng tôi rất dễ bị ảnh hưởng bởi tiếng ồn trong dữ liệu đào tạo. Mặc dù tổn thất đào tạo có thể được giảm hiệu quả, nhưng tổn thất thử nghiệm vẫn cao hơn nhiều. Nó cho thấy mô hình phức tạp vượt quá dữ liệu.

```
# Pick all the dimensions from the polynomial features
train(poly_features[:n_train, :], poly_features[n_train:, :],
      labels[:n_train], labels[n_train:], num_epochs=1500)
```

```
weight: [[ 4.962039    1.3130691  -3.2638512   5.021225   -0.57493603  0.
↪9147651
  -0.11779631  0.22265539  0.1897087  -0.08537436 -0.11553093  0.17667438
   0.07173117 -0.1627829   0.1078734  -0.17700529  0.04046748 -0.07227355
   0.1876146   0.05505766]]
```



Trong các phần tiếp theo, chúng tôi sẽ tiếp tục thảo luận về các vấn đề và phương pháp quá mức để đối phó với chúng, chẳng hạn như phân rã trọng lượng và bỏ học.

5.4.5 Tóm tắt

- Vì lỗi tổng quát hóa không thể được ước tính dựa trên lỗi đào tạo, chỉ cần giảm thiểu lỗi đào tạo sẽ không nhất thiết có nghĩa là giảm lỗi tổng quát hóa. Các mô hình học máy cần phải cẩn thận để bảo vệ chống quá mức để giảm thiểu lỗi tổng quát hóa.
- Một bộ xác thực có thể được sử dụng để lựa chọn mô hình, với điều kiện là nó không được sử dụng quá tự do.
- Underfitting có nghĩa là một mô hình không thể giảm lỗi đào tạo. Khi lỗi đào tạo thấp hơn nhiều so với lỗi xác nhận, có overfitting.
- Chúng ta nên chọn một mô hình phức tạp thích hợp và tránh sử dụng các mẫu đào tạo không đủ.

5.4.6 Bài tập

1. Bạn có thể giải quyết vấn đề hồi quy đa thức chính xác? Gợi ý: sử dụng đại số tuyến tính.
2. Xem xét lựa chọn mô hình cho đa thức:
    1. Vẽ sự mất mát đào tạo so với độ phức tạp mô hình (mức độ đa thức). Bạn quan sát điều gì? Bạn cần mức độ đa thức nào để giảm tổn thất đào tạo xuống 0?
    2. Vẽ tổn thất thử nghiệm trong trường hợp này.
    3. Tạo ra âm mưu tương tự như một hàm của lượng dữ liệu.
3. Điều gì xảy ra nếu bạn thả bình thường hóa ($1/i!$) of the polynomial features x^i? Bạn có thể khắc phục điều này theo một cách khác không?
4. Bạn có thể bao giờ mong đợi để thấy lỗi tổng quát không?

Discussions[65]

[65] https://discuss.d2l.ai/t/97

5.5 Trọng lượng phân rã

Bây giờ chúng tôi đã đặc trưng vấn đề overfitting, chúng tôi có thể giới thiệu một số kỹ thuật tiêu chuẩn để điều chỉnh các mô hình. Nhớ lại rằng chúng ta luôn có thể giảm thiểu quá mức bằng cách đi ra ngoài và thu thập thêm dữ liệu đào tạo. Điều đó có thể tốn kém, tốn thời gian hoặc hoàn toàn nằm ngoài tầm kiểm soát của chúng tôi, khiến nó không thể xảy ra trong thời gian ngắn. Hiện tại, chúng ta có thể giả định rằng chúng ta đã có nhiều dữ liệu chất lượng cao như tài nguyên của chúng tôi cho phép và tập trung vào các kỹ thuật chính quy hóa.

Nhớ lại rằng trong ví dụ hồi quy đa thức của chúng tôi (Section 5.4), chúng ta có thể giới hạn năng lực của mô hình của chúng tôi chỉ đơn giản bằng cách điều chỉnh mức độ của đa thức được trang bị. Thật vậy, hạn chế số lượng các tính năng là một kỹ thuật phổ biến để giảm thiểu quá mức. Tuy nhiên, chỉ cần ném các tính năng sang một bên có thể quá cùn một công cụ cho công việc. Gắn bó với ví dụ hồi quy đa thức, xem xét những gì có thể xảy ra với các đầu vào chiều cao. Các phần mở rộng tự nhiên của đa thức cho dữ liệu đa biến được gọi là * monomials*, đơn giản là sản phẩm của quyền hạn của các biến. Mức độ của một monomial là tổng của các cường quốc. Ví dụ, $x_1^2 x_2$, và $x_3 x_5^2$ đều là nguyên khối của độ 3.

Lưu ý rằng số lượng thuật ngữ với mức độ d thổi lên nhanh chóng khi d phát triển lớn hơn. Với k biến số, số lượng monomials của độ d (tức là k multichoose d) là $\binom{k-1+d}{k-1}$. Ngay cả những thay đổi nhỏ về mức độ, nói từ 2 đến 3, làm tăng đáng kể độ phức tạp của mô hình của chúng tôi. Vì vậy chúng ta thường cần một công cụ hạt mịn hơn để điều chỉnh độ phức tạp của chức năng.

5.5.1 Định mức và phân rã trọng lượng

Chúng tôi đã mô tả cả định mức L_2 và định mức L_1, đây là những trường hợp đặc biệt của định mức L_p chung hơn trong Section 3.3.10. * Phân rã trọng lượng* (thường được gọi là L_2 regarization), có thể là kỹ thuật được sử dụng rộng rãi nhất để điều chỉnh các mô hình học máy tham số. Kỹ thuật này được thúc đẩy bởi trực giác cơ bản trong số tất cả các chức năng f, chức năng $f = 0$ (gán giá trị 0 cho tất cả các đầu vào) trong một số nghĩa là * đơn giản nhất*, và rằng chúng ta có thể đo lường sự phức tạp của một hàm bằng khoảng cách của nó từ 0. Nhưng chính xác như thế nào chúng ta nên đo khoảng cách giữa một hàm và không? Không có câu trả lời đúng duy nhất. Trên thực tế, toàn bộ nhánh toán học, bao gồm các phần phân tích chức năng và lý thuyết về không gian Banach, được dành để trả lời vấn đề này.

Một cách giải thích đơn giản có thể là đo độ phức tạp của một hàm tuyến tính $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ theo một số định mức của vectơ trọng lượng của nó, ví dụ, $\|\mathbf{w}\|^2$. Phương pháp phổ biến nhất để đảm bảo một vector trọng lượng nhỏ là thêm định mức của nó như một thuật ngữ phạt cho vấn đề giảm thiểu tổn thất. Do đó chúng tôi thay thế mục tiêu ban đầu của chúng tôi, *giảm thiểu tổn thất dự đoán trên nhãn đào tạo*, with new Mới objective mục tiêu, *giảm thiểu tổng số tổn thất dự đoán và thuật ngữ phạt phút*. Bây giờ, nếu vector trọng lượng của chúng ta phát triển quá lớn, thuật toán học tập của chúng ta có thể tập trung vào việc giảm thiểu định mức trọng lượng $\|\mathbf{w}\|^2$ so với giảm thiểu lỗi đào tạo. Đó chính xác là những gì chúng tôi muốn. Để minh họa những điều trong code, chúng ta hãy hồi sinh ví dụ trước đây của chúng tôi từ Section 4.1 cho hồi quy tuyến tính. Ở đó, sự mất mát của chúng tôi đã được đưa ra bởi

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{5.5.1}$$

Nhớ lại rằng $\mathbf{x}^{(i)}$ là các tính năng, $y^{(i)}$ là nhãn cho tất cả các ví dụ dữ liệu i và $(\mathbf{w}, b)$ là các thông số trọng lượng và thiên vị, tương ứng. Để phạt kích thước của vector trọng lượng, bằng cách nào đó chúng ta phải thêm $\|\mathbf{w}\|^2$ vào chức năng mất mát, nhưng làm thế nào mô hình nên thương mại mất tiêu chuẩn cho hình phạt phụ gia mới này? Trong thực tế, chúng tôi mô tả sự cân bằng này thông qua hằng số * thường xuyệt* λ, một siêu

tham số không âm mà chúng tôi phù hợp bằng dữ liệu xác thực:

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2, \tag{5.5.2}$$

Đối với $\lambda = 0$, chúng tôi phục hồi chức năng mất mát ban đầu của chúng tôi. Đối với $\lambda > 0$, chúng tôi hạn chế kích thước của $\|\mathbf{w}\|$. Chúng tôi chia cho 2 theo quy ước: khi chúng ta lấy đạo hàm của một hàm bậc hai, 2 và $1/2$ hủy bỏ, đảm bảo rằng biểu thức cho bản cập nhật trông đẹp và đơn giản. Người đọc tinh xảo có thể tự hỏi tại sao chúng ta làm việc với định mức bình phương chứ không phải định mức tiêu chuẩn (tức là khoảng cách Euclide). Chúng tôi làm điều này để thuận tiện tính toán. Bằng cách bình phương định mức L_2, chúng ta loại bỏ căn bậc hai, để lại tổng hình vuông của mỗi thành phần của vector trọng lượng. Điều này làm cho đạo hàm của hình phạt dễ tính toán: tổng các dẫn xuất bằng đạo hàm của tổng.

Hơn nữa, bạn có thể hỏi tại sao chúng tôi làm việc với định mức L_2 ở nơi đầu tiên và không, nói, định mức L_1. Trên thực tế, các lựa chọn khác là hợp lệ và phổ biến trong suốt số liệu thống kê. Trong khi L_2-mô hình tuyến tính được điều chỉnh tạo thành thuật toán * hồi quy sườn núi cổ điển*, hồi quy tuyến tính :math:`L_1` là một mô hình cơ bản tương tự trong thống kê, thường được gọi là hồi quy lasso.*

Một lý do để làm việc với định mức L_2 là nó đặt một hình phạt lớn trên các thành phần lớn của vector trọng lượng. Điều này làm thiên vị thuật toán học tập của chúng tôi đối với các mô hình phân phối trọng lượng đều trên một số lượng lớn các tính năng. Trong thực tế, điều này có thể làm cho chúng mạnh mẽ hơn với lỗi đo lường trong một biến duy nhất. Ngược lại, L_1 hình phạt dẫn đến các mô hình tập trung trọng lượng trên một tập hợp nhỏ các tính năng bằng cách xóa các trọng lượng khác về 0. Điều này được gọi là lựa chọn tính năng*, có thể mong muốn vì các lý do khác.

Sử dụng cùng một ký hiệu trong (4.1.10), các bản cập nhật xuống dốc ngẫu nhiên minibatch cho hồi quy L_2-regarized theo sau:

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \tag{5.5.3}$$

Như trước đây, chúng tôi cập nhật **w** dựa trên số tiền mà ước tính của chúng tôi khác với quan sát. Tuy nhiên, chúng tôi cũng thu nhỏ kích thước **w** về phía không. Đó là lý do tại sao phương pháp đôi khi được gọi là "phân rã trọng lượng": chỉ cho thuật ngữ phạt, thuật toán tối ưu hóa của chúng tôi* phân tác* trọng lượng ở mỗi bước tập luyện. Trái ngược với lựa chọn tính năng, trọng lượng phân rã cung cấp cho chúng ta một cơ chế liên tục để điều chỉnh độ phức tạp của một hàm. Các giá trị nhỏ hơn của λ tương ứng với ít bị hạn chế **w**, trong khi các giá trị lớn hơn của λ hạn chế **w** đáng kể hơn.

Cho dù chúng ta bao gồm một hình phạt thiên vị tương ứng b^2 có thể thay đổi giữa các triển khai, và có thể thay đổi giữa các lớp của mạng thần kinh. Thông thường, chúng ta không thường xuyên hóa thuật ngữ thiên vị của lớp đầu ra của mạng.

5.5.2 High-Dimensional Linear Regression

Chúng ta có thể minh họa những lợi ích của phân rã trọng lượng thông qua một ví dụ tổng hợp đơn giản.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

Đầu tiên, chúng tôi tạo ra một số dữ liệu như trước

$$y = 0.05 + \ sum_{i=1}^{d} 0.01 x_i + \ epsilon \ textwhere \ epsilon \ sim \ mathcal N(0, 0.01^2). \tag{5.5.4}$$

Chúng tôi chọn nhãn của chúng tôi là một chức năng tuyến tính của đầu vào của chúng tôi, bị hỏng bởi tiếng ồn Gaussian với 0 trung bình và độ lệch chuẩn 0,01. Để làm cho các hiệu ứng của overfitting rõ rệt, chúng ta có thể tăng chiều của vấn đề của chúng tôi lên $d = 200$ và làm việc với một bộ đào tạo nhỏ chỉ chứa 20 ví dụ.

```
n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = torch.ones((num_inputs, 1)) * 0.01, 0.05
train_data = d2l.synthetic_data(true_w, true_b, n_train)
train_iter = d2l.load_array(train_data, batch_size)
test_data = d2l.synthetic_data(true_w, true_b, n_test)
test_iter = d2l.load_array(test_data, batch_size, is_train=False)
```

5.5.3 Thực hiện từ đầu

Sau đây, chúng tôi sẽ thực hiện phân rã trọng lượng từ đầu, chỉ bằng cách thêm hình phạt L_2 bình phương vào hàm mục tiêu ban đầu.

Initializing Model Parameters

Đầu tiên, chúng ta sẽ định nghĩa một hàm để khởi tạo ngẫu nhiên các tham số model của chúng ta.

```
def init_params():
    w = torch.normal(0, 1, size=(num_inputs, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return [w, b]
```

Định nghĩa L_2 Hình phạt Norm

Có lẽ cách thuận tiện nhất để thực hiện hình phạt này là để vuông tất cả các điều khoản tại chỗ và tổng hợp chúng lên.

```
def l2_penalty(w):
    return torch.sum(w.pow(2)) / 2
```

Xác định vòng đào tạo

Mã sau phù hợp với một mô hình trên bộ đào tạo và đánh giá nó trên bộ thử nghiệm. Mạng tuyến tính và tổn thất bình phương không thay đổi kể từ Chapter 4, vì vậy chúng tôi sẽ chỉ nhập chúng qua `d2l.linreg` và `d2l.squared_loss`. Thay đổi duy nhất ở đây là mất mát của chúng tôi bây giờ bao gồm thời hạn phạt.

```
def train(lambd):
    w, b = init_params()
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    num_epochs, lr = 100, 0.003
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            # The L2 norm penalty term has been added, and broadcasting
```

(continues on next page)

(continued from previous page)

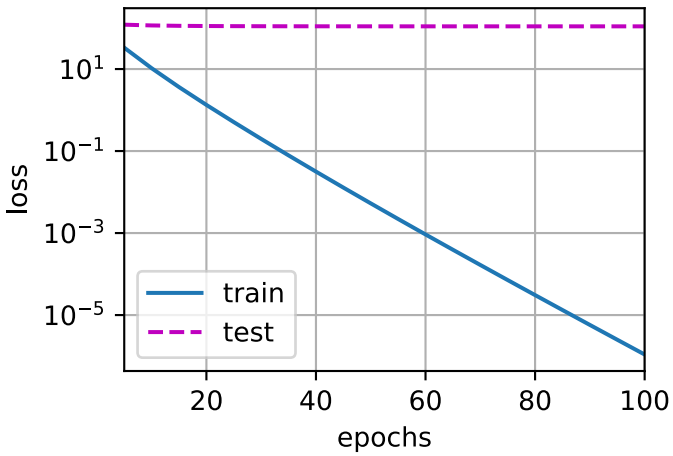
```
            # makes `l2_penalty(w)` a vector whose length is `batch_size`
            l = loss(net(X), y) + lambd * l2_penalty(w)
            l.sum().backward()
            d2l.sgd([w, b], lr, batch_size)
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', torch.norm(w).item())
```

Đào tạo mà không cần Regularization

Bây giờ chúng ta chạy mã này với `lambd = 0`, vô hiệu hóa phân rã trọng lượng. Lưu ý rằng chúng tôi quá phù hợp xấu, giảm lỗi đào tạo nhưng không phải lỗi kiểm tra - một trường hợp sách giáo khoa của overfitting.

```
train(lambd=0)
```

```
L2 norm of w: 12.982525825500488
```

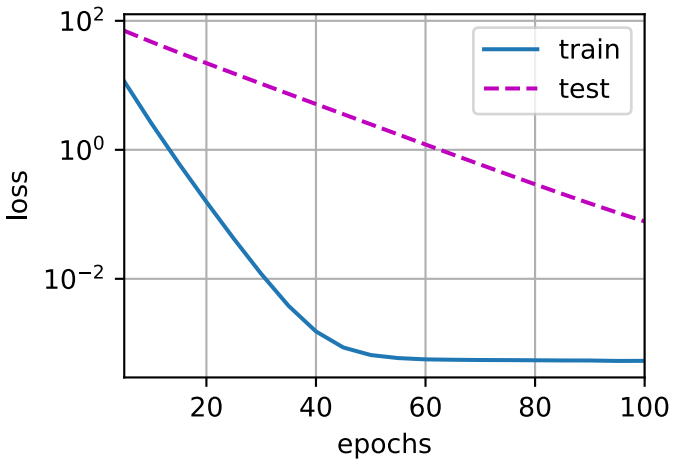


Sử dụng trọng lượng

Dưới đây, chúng tôi chạy với sự phân rã trọng lượng đáng kể. Lưu ý rằng lỗi đào tạo tăng nhưng lỗi thử nghiệm giảm. Đây chính xác là hiệu quả mà chúng ta mong đợi từ việc chính quy hóa.

```
train(lambd=3)
```

```
L2 norm of w: 0.3708249628543854
```



5.5.4 Thiết tập

Bởi vì sự phân rã trọng lượng có mặt khắp nơi trong việc tối ưu hóa mạng thần kinh, khung học sâu làm cho nó đặc biệt thuận tiện, tích hợp phân rã trọng lượng vào thuật toán tối ưu hóa để dễ sử dụng kết hợp với bất kỳ chức năng mất mát nào. Hơn nữa, tích hợp này phục vụ một lợi ích tính toán, cho phép các thủ thuật thực hiện thêm phân rã trọng lượng vào thuật toán, mà không cần bất kỳ chi phí tính toán bổ sung nào. Vì phần phân rã trọng lượng của bản cập nhật chỉ phụ thuộc vào giá trị hiện tại của mỗi tham số, trình tối ưu hóa phải chạm vào từng tham số một lần.

Trong mã sau, chúng tôi chỉ định siêu tham số phân rã trọng lượng trực tiếp thông qua `weight_decay` khi khởi tạo trình tối ưu hóa của chúng tôi. Theo mặc định, PyTorch phân rã cả trọng lượng và thành kiến cùng một lúc. Ở đây chúng tôi chỉ đặt `weight_decay` cho trọng lượng, vì vậy tham số thiên vị b sẽ không phân rã.

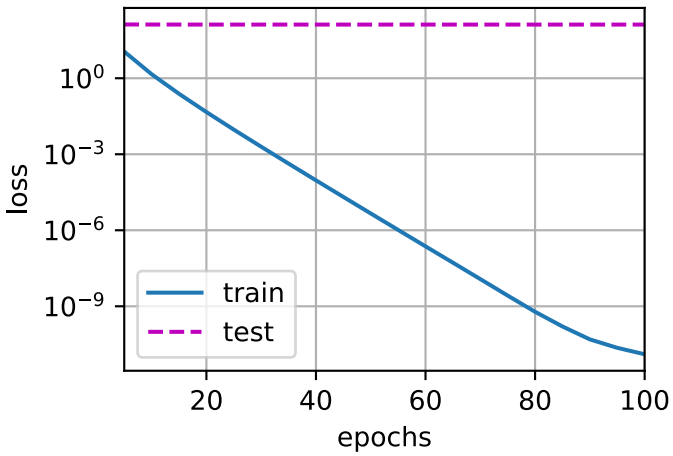
```
def train_concise(wd):
    net = nn.Sequential(nn.Linear(num_inputs, 1))
    for param in net.parameters():
        param.data.normal_()
    loss = nn.MSELoss(reduction='none')
    num_epochs, lr = 100, 0.003
    # The bias parameter has not decayed
    trainer = torch.optim.SGD([
        {"params":net[0].weight,'weight_decay': wd},
        {"params":net[0].bias}], lr=lr)
    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                            xlim=[5, num_epochs], legend=['train', 'test'])
    for epoch in range(num_epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.mean().backward()
            trainer.step()
        if (epoch + 1) % 5 == 0:
            animator.add(epoch + 1, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L2 norm of w:', net[0].weight.norm().item())
```

Các ô trông giống hệt với những lô khi chúng tôi thực hiện phân rã trọng lượng từ vết trầy xước. Tuy nhiên, chúng chạy nhanh hơn đáng kể và dễ thực hiện hơn, một lợi ích sẽ trở nên rõ rệt hơn đối với các vấn đề lớn

hơn.

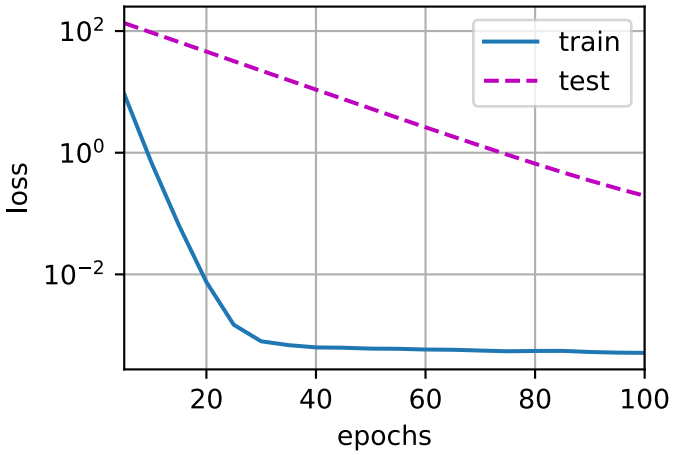
```
train_concise(0)
```

```
L2 norm of w: 12.434950828552246
```




```
train_concise(3)
```

```
L2 norm of w: 0.37203601002693176
```



Cho đến nay, chúng ta chỉ chạm vào một khái niệm về những gì tạo thành một hàm tuyến tính đơn giản. Hơn nữa, những gì tạo thành một hàm phi tuyến đơn giản có thể là một câu hỏi thậm chí còn phức tạp hơn. Ví dụ, tái tạo hạt nhân Hilbert space (RKHS)[66] cho phép người ta áp dụng các công cụ được giới thiệu cho các hàm tuyến tính trong bối cảnh phi tuyến tính. Thật không may, các thuật toán dựa trên RKHS có xu hướng mở rộng quy mô dữ liệu lớn, chiều cao. Trong cuốn sách này, chúng tôi sẽ mặc định để heuristic đơn giản của việc áp dụng phân rã trọng lượng trên tất cả các lớp của một mạng sâu.

[66] https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space

5.5.5 Tóm tắt

- Thường xuyên hóa là một phương pháp phổ biến để đối phó với overfitting. Nó bổ sung một thuật ngữ phạt cho chức năng mất trên bộ đào tạo để giảm độ phức tạp của mô hình đã học.
- Một lựa chọn đặc biệt để giữ cho mô hình đơn giản là giảm cân bằng cách sử dụng hình phạt L_2. Điều này dẫn đến phân rã trọng lượng trong các bước cập nhật của thuật toán học tập.
- Chức năng phân rã trọng lượng được cung cấp trong các trình tối ưu hóa từ các khuôn khổ học sâu.
- Các bộ tham số khác nhau có thể có các hành vi cập nhật khác nhau trong cùng một vòng đào tạo.

5.5.6 Bài tập

1. Thử nghiệm với giá trị của λ trong bài toán ước lượng trong phần này. Đào tạo cốt truyện và độ chính xác kiểm tra như một chức năng của λ. Bạn quan sát điều gì?
2. Sử dụng bộ xác thực để tìm giá trị tối ưu là λ. Nó có thực sự là giá trị tối ưu? Điều này có quan trọng không?
3. Các phương trình cập nhật sẽ trông như thế nào nếu thay vì $\|\mathbf{w}\|^2$ chúng tôi sử dụng $\sum_i |w_i|$ làm hình phạt của sự lựa chọn của chúng tôi (L_1 chính quy hóa)?
4. Chúng ta biết rằng $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Bạn có thể tìm thấy một phương trình tương tự cho ma trận (xem định mức Frobenius trong Section 3.3.10)?
5. Xem lại mối quan hệ giữa lỗi đào tạo và lỗi tổng quát hóa. Ngoài phân rã trọng lượng, tăng cường đào tạo, và sử dụng một mô hình phức tạp phù hợp, những cách khác bạn có thể nghĩ đến để đối phó với overfitting?
6. Trong thống kê Bayesian, chúng tôi sử dụng sản phẩm trước và khả năng để đến một phía sau thông qua $P(w \mid x) \propto P(x \mid w)P(w)$. Làm thế nào bạn có thể xác định $P(w)$ với quy định hóa?

Discussions[67]

5.6 Bỏ học

Năm Section 5.5, chúng tôi đã giới thiệu cách tiếp cận cổ điển để điều chỉnh các mô hình thống kê bằng cách phạt định mức L_2 của trọng lượng. Về xác suất, chúng ta có thể biện minh cho kỹ thuật này bằng cách lập luận rằng chúng ta đã giả định một niềm tin trước đó rằng trọng lượng lấy giá trị từ một phân phối Gaussian với trung bình 0. Trực quan hơn, chúng ta có thể lập luận rằng chúng tôi khuyến khích mô hình trải ra trọng lượng của nó giữa nhiều tính năng thay vì phụ thuộc quá nhiều vào một số lượng nhỏ các hiệp hội có khả năng giả mạo.

[67] https://discuss.d2l.ai/t/99

5.6.1 Overfitting Revisited

Đối mặt với nhiều tính năng hơn ví dụ, các mô hình tuyến tính có xu hướng overfit. Nhưng đưa ra nhiều ví dụ hơn các tính năng, chúng ta thường có thể tin tưởng vào các mô hình tuyến tính không quá mức. Thật không may, độ tin cậy mà các mô hình tuyến tính khái quát hóa đi kèm với chi phí. Các mô hình tuyến tính được áp dụng một cách ngây thơ không tính đến các tương tác giữa các tính năng. Đối với mọi tính năng, một mô hình tuyến tính phải gán một trọng lượng dương hoặc âm, bỏ qua bối cảnh.

Trong các văn bản truyền thống, sự căng thẳng cơ bản này giữa tính tổng quát và tính linh hoạt được mô tả là sự cân bằng phương sai *bias-variance*. Các mô hình tuyến tính có thiên vị cao: chúng chỉ có thể đại diện cho một lớp hàm nhỏ. Tuy nhiên, các mô hình này có phương sai thấp: chúng cho kết quả tương tự trên các mẫu ngẫu nhiên khác nhau của dữ liệu.

Các mạng thần kinh sâu sống ở đầu đối diện của phổ phương sai thiên vị. Không giống như các mô hình tuyến tính, mạng thần kinh không bị giới hạn để nhìn vào từng tính năng riêng lẻ. Họ có thể học tương tác giữa các nhóm tính năng. Ví dụ, họ có thể suy ra rằng "Nigeria" và "Western Union" xuất hiện cùng nhau trong một email chỉ ra thư rác nhưng điều đó riêng biệt họ không.

Ngay cả khi chúng ta có nhiều ví dụ hơn nhiều so với các tính năng, các mạng thần kinh sâu có khả năng vượt trội. Năm 2017, một nhóm các nhà nghiên cứu đã chứng minh tính linh hoạt cực đoan của mạng thần kinh bằng cách đào tạo các mạng lưới sâu trên các hình ảnh được dán nhãn ngẫu nhiên. Mặc dù không có bất kỳ mô hình thực sự nào liên kết đầu vào với đầu ra, họ phát hiện ra rằng mạng thần kinh được tối ưu hóa bởi gốc gradient ngẫu nhiên có thể gắn nhãn mọi hình ảnh trong bộ đào tạo một cách hoàn hảo. Hãy xem xét điều này có nghĩa là gì. Nếu các nhãn được gán thống nhất một cách ngẫu nhiên và có 10 lớp, thì không có bộ phân loại nào có thể làm tốt hơn 10% độ chính xác trên dữ liệu holdout. Khoảng cách tổng quát ở đây là một con số khổng lồ 90%. Nếu các mô hình của chúng tôi rất biểu cảm đến mức chúng có thể quá mức phù hợp với điều này, thì khi nào chúng ta nên mong đợi chúng không quá mức?

Các nền tảng toán học cho các thuộc tính tổng quát khó hiểu của các mạng sâu vẫn là các câu hỏi nghiên cứu mở, và chúng tôi khuyến khích người đọc định hướng lý thuyết để đào sâu hơn vào chủ đề. Hiện tại, chúng tôi chuyển sang điều tra các công cụ thực tế có xu hướng cải thiện thực nghiệm sự tổng quát của lưới sâu.

5.6.2 Mạnh mẽ thông qua nhiễu loạn

Chúng ta hãy suy nghĩ ngắn gọn về những gì chúng ta mong đợi từ một mô hình dự đoán tốt. Chúng tôi muốn nó để peform tốt trên dữ liệu không nhìn thấy. Lý thuyết tổng quát hóa cổ điển cho thấy rằng để thu hẹp khoảng cách giữa tàu và hiệu suất thử nghiệm, chúng ta nên hướng đến một mô hình đơn giản. Sự đơn giản có thể đến dưới dạng một số lượng nhỏ kích thước. Chúng tôi đã khám phá điều này khi thảo luận về các hàm cơ sở đơn nguyên của các mô hình tuyến tính trong Section 5.4. Ngoài ra, như chúng ta đã thấy khi thảo luận về phân rã trọng lượng (L_2 chính quy hóa) trong Section 5.5, định mức (nghịch đảo) của các tham số cũng đại diện cho một thước đo đơn giản hữu ích. Một khái niệm hữu ích khác về sự đơn giản là sự trơn tru, tức là chức năng không nên nhạy cảm với những thay đổi nhỏ đối với đầu vào của nó. Ví dụ: khi chúng tôi phân loại hình ảnh, chúng tôi hy vọng rằng việc thêm một số nhiễu ngẫu nhiên vào các pixel chủ yếu là vô hại.

Năm 1995, Christopher Bishop chính thức hóa ý tưởng này khi ông chứng minh rằng việc đào tạo với tiếng ồn đầu vào tương đương với việc chính quy hóa Tikhonov (Bishop, 1995). Công việc này đã vẽ một kết nối toán học rõ ràng giữa yêu cầu rằng một hàm được trơn tru (và do đó đơn giản), và yêu cầu rằng nó có khả năng phục hồi với nhiễu loạn trong đầu vào.

Sau đó, vào năm 2014, Srivastava et al. (Srivastava et al., 2014) đã phát triển một ý tưởng thông minh về cách áp dụng ý tưởng của Giám mục vào các lớp nội bộ của một mạng, quá. Cụ thể, họ đề xuất tiêm tiếng ồn vào mỗi lớp của mạng trước khi tính toán lớp tiếp theo trong quá trình đào tạo. Họ nhận ra rằng khi đào tạo một mạng lưới sâu với nhiều lớp, việc tiêm tiếng ồn sẽ thực hiện sự trơn tru chỉ trên bản đồ đầu vào-đầu ra.

Ý tưởng của họ, được gọi là * dropout, *liên quan đến việc tiêm tiếng ồn trong khi tính toán từng lớp bên trong trong quá trình truyền chuyển tiếp và nó đã trở thành một kỹ thuật tiêu chuẩn để đào tạo mạng thần kinh. Phương pháp này được gọi là *dropout* bởi vì chúng tôi theo nghĩa đen *thả ra* một số tế bào thần kinh trong quá trình đào tạo. Trong suốt quá trình đào tạo, trên mỗi lần lặp lại, bỏ học tiêu chuẩn bao gồm zeroing ra một số phần của các nút trong mỗi lớp trước khi tính toán lớp tiếp theo.

Để rõ ràng, chúng tôi đang áp đặt câu chuyện của riêng mình với liên kết với Giám mục. Bài báo gốc về bỏ học cung cấp trực giác thông qua một sự tương tự đáng ngạc nhiên với sinh sản tình dục. Các tác giả cho rằng mạng thần kinh overfitting được đặc trưng bởi một trạng thái trong đó mỗi lớp dựa vào một mô hình kích hoạt specifc trong lớp trước, gọi điều kiện này * đồng thích nghi*. Bỏ học, họ tuyên bố, phá vỡ sự đồng thích nghi giống như sinh sản tình dục được lập luận là phá vỡ các gen đồng thích nghi.

Thách thức quan trọng sau đó là làm thế nào để tiêm tiếng ồn này. Một ý tưởng là tiêm nhiễu theo cách *unbiased* sao cho giá trị kỳ vọng của mỗi lớp — trong khi sửa chữa các lớp khác - bằng với giá trị mà nó sẽ mất tiếng ồn vắng mặt.

Trong tác phẩm của Bishop, ông đã thêm tiếng ồn Gaussian vào các đầu vào cho một mô hình tuyến tính. Tại mỗi lần lặp lại đào tạo, ông đã thêm tiếng ồn lấy mẫu từ một phân phối với trung bình 0 $\epsilon \sim \mathcal{N}(0, \sigma^2)$ đến đầu vào $\mathbf{x}$, mang lại một điểm nhiễu loạn $\mathbf{x}' = \mathbf{x} + \epsilon$. Trong kỳ vọng, $E[\mathbf{x}'] = \mathbf{x}$.

Trong quy định bỏ học tiêu chuẩn, người ta làm giảm từng lớp bằng cách bình thường hóa bởi phần nhỏ của các nút được giữ lại (không bị loại bỏ). Nói cách khác, với xác suất bỏ lọc* p, mỗi kích hoạt trung gian h được thay thế bằng một biến ngẫu nhiên h' như sau:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases} \tag{5.6.1}$$

Theo thiết kế, kỳ vọng vẫn không thay đổi, tức là $E[h'] = h$.

5.6.3 Bỏ học trong thực hành

Nhớ lại MLP với một lớp ẩn và 5 đơn vị ẩn trong Fig. 5.1.1. Khi chúng ta áp dụng dropout cho một lớp ẩn, zeroing ra từng đơn vị ẩn với xác suất p, kết quả có thể được xem như là một mạng chỉ chứa một tập hợp con của các tế bào thần kinh ban đầu. Trong Fig. 5.6.1, h_2 và h_5 bị loại bỏ. Do đó, việc tính toán các đầu ra không còn phụ thuộc vào h_2 hoặc h_5 và gradient tương ứng của chúng cũng biến mất khi thực hiện truyền ngược. Bằng cách này, việc tính toán lớp đầu ra không thể phụ thuộc quá mức vào bất kỳ một phần tử nào của $h_1, \ldots, h_5$.

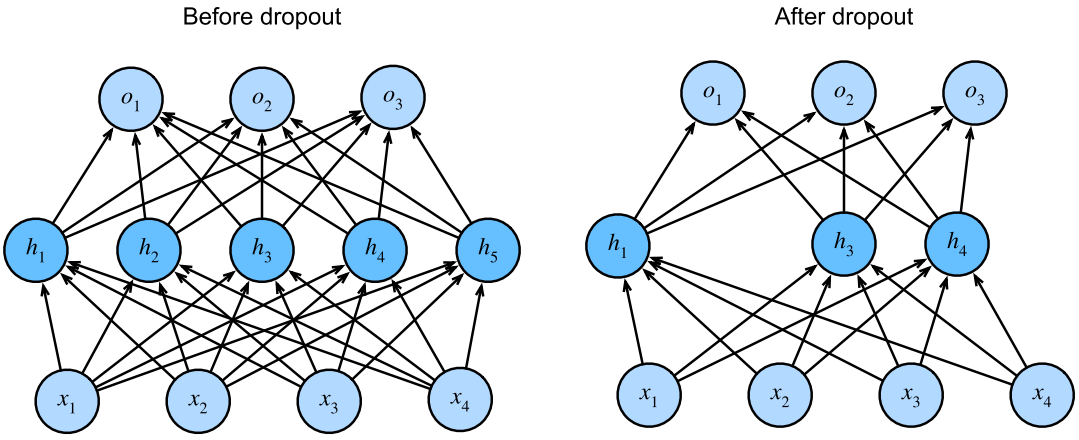


Fig. 5.6.1: MLP before and after dropout.

Thông thường, chúng tôi vô hiệu hóa bỏ học tại thời điểm thử nghiệm. Với một mô hình được đào tạo và một ví dụ mới, chúng tôi không bỏ bất kỳ nút nào và do đó không cần phải bình thường hóa. Tuy nhiên, có một số ngoại lệ: một số nhà nghiên cứu sử dụng bỏ học tại thời điểm thử nghiệm như một heuristic để ước tính * không chắc chắn* của dự đoán mạng thần kinh: nếu các dự đoán đồng ý trên nhiều mặt nạ bỏ học khác nhau, thì chúng ta có thể nói rằng mạng tự tin hơn.

5.6.4 Thực hiện từ đầu

Để thực hiện hàm dropout cho một lớp duy nhất, chúng ta phải vẽ nhiều mẫu từ một biến ngẫu nhiên Bernoulli (nhị phân) như lớp của chúng ta có kích thước, trong đó biến ngẫu nhiên có giá trị 1 (giữ) với xác suất $1 - p$ và 0 (thả) với xác suất p. Một cách dễ dàng để thực hiện điều này là lần đầu tiên vẽ các mẫu từ phân phối thống nhất $U[0, 1]$. Sau đó, chúng ta có thể giữ các nút đó mà mẫu tương ứng lớn hơn p, thả phần còn lại.

Trong đoạn code sau đây, ta thực hiện hàm `dropout_layer` loại bỏ các phần tử trong đầu vào tensor `X` với xác suất `dropout`, rescaling phần còn lại như mô tả ở trên: chia những người sống sót cho `1.0-dropout`.

```
import torch
from torch import nn
from d2l import torch as d2l


def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return torch.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = (torch.rand(X.shape) > dropout).float()
    return mask * X / (1.0 - dropout)
```

Chúng ta có thể test out the `dropout_layer` function on a few examples. Trong các dòng mã sau đây, chúng tôi vượt qua đầu vào `X` thông qua thao tác bỏ học, với xác suất lần lượt là 0, 0.5 và 1.

```
X= torch.arange(16, dtype = torch.float32).reshape((2, 8))
print(X)
print(dropout_layer(X, 0.))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1.))
```

```
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
tensor([[ 0.,  0.,  0.,  6.,  8., 10., 12., 14.],
        [16., 18., 20., 22.,  0.,  0.,  0.,  0.]])
tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
```

Xác định các tham số mô hình

Một lần nữa, chúng tôi làm việc với bộ dữ liệu Fashion-MNIST được giới thiệu trong Section 4.5. Chúng ta xác định một MLP với hai lớp ẩn chứa 256 đơn vị mỗi hột.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256
```

Xác định mô hình

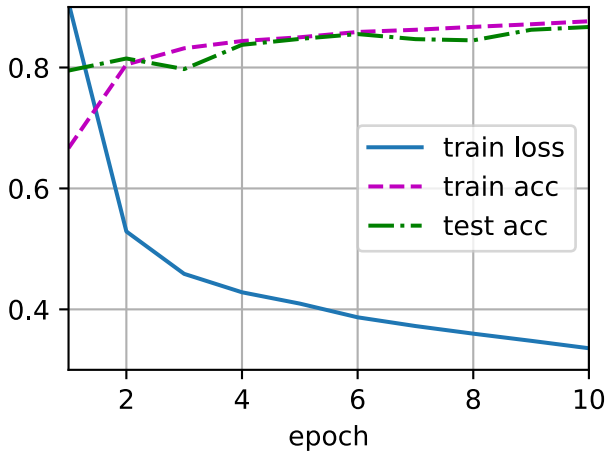
Mô hình dưới đây áp dụng dropout cho đầu ra của mỗi lớp ẩn (theo chức năng kích hoạt). Chúng ta có thể đặt xác suất bỏ học cho từng lớp riêng biệt. Một xu hướng phổ biến là đặt xác suất bỏ học thấp hơn gần với lớp đầu vào. Dưới đây chúng tôi đặt nó thành 0,2 và 0,5 cho các lớp ẩn đầu tiên và thứ hai, tương ứng. Chúng tôi đảm bảo rằng việc bỏ học chỉ hoạt động trong quá trình đào tạo.

```
dropout1, dropout2 = 0.2, 0.5

class Net(nn.Module):
    def __init__(self, num_inputs, num_outputs, num_hiddens1, num_hiddens2,
                 is_training = True):
        super(Net, self).__init__()
        self.num_inputs = num_inputs
        self.training = is_training
        self.lin1 = nn.Linear(num_inputs, num_hiddens1)
        self.lin2 = nn.Linear(num_hiddens1, num_hiddens2)
        self.lin3 = nn.Linear(num_hiddens2, num_outputs)
        self.relu = nn.ReLU()

    def forward(self, X):
        H1 = self.relu(self.lin1(X.reshape((-1, self.num_inputs))))
        # Use dropout only when training the model
        if self.training == True:
            # Add a dropout layer after the first fully connected layer
            H1 = dropout_layer(H1, dropout1)
        H2 = self.relu(self.lin2(H1))
        if self.training == True:
            # Add a dropout layer after the second fully connected layer
            H2 = dropout_layer(H2, dropout2)
        out = self.lin3(H2)
        return out


net = Net(num_inputs, num_outputs, num_hiddens1, num_hiddens2)
```

Đào tạo và kiểm tra

Điều này tương tự như đào tạo và thử nghiệm MLP s được mô tả trước đó.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = nn.CrossEntropyLoss(reduction='none')
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



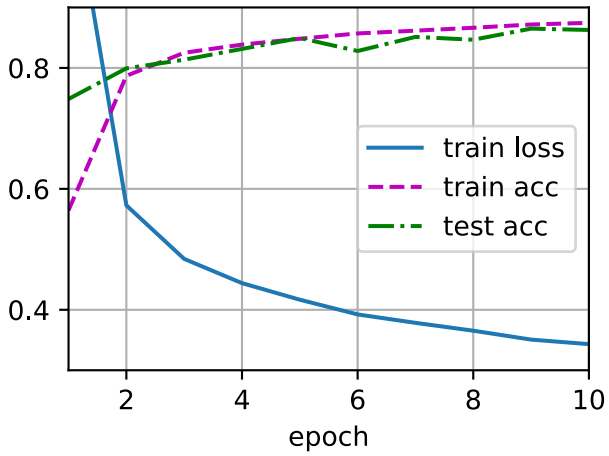
5.6.5 Thiết tập

Với API cấp cao, tất cả những gì chúng ta cần làm là thêm một lớp `Dropout` sau mỗi lớp được kết nối hoàn toàn, truyền trong xác suất bỏ học làm đối số duy nhất cho hàm tạo của nó. Trong quá trình đào tạo, lớp `Dropout` sẽ thả ngẫu nhiên các đầu ra của lớp trước đó (hoặc tương đương, các đầu vào cho lớp tiếp theo) theo xác suất bỏ học được chỉ định. Khi không ở chế độ đào tạo, lớp `Dropout` chỉ đơn giản là truyền dữ liệu qua trong quá trình thử nghiệm.

```
net = nn.Sequential(nn.Flatten(),
        nn.Linear(784, 256),
        nn.ReLU(),
        # Add a dropout layer after the first fully connected layer
        nn.Dropout(dropout1),
        nn.Linear(256, 256),
        nn.ReLU(),
        # Add a dropout layer after the second fully connected layer
        nn.Dropout(dropout2),
        nn.Linear(256, 10))

def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, std=0.01)

net.apply(init_weights);
```

Tiếp theo, chúng tôi đào tạo và kiểm tra mô hình.

```
trainer = torch.optim.SGD(net.parameters(), lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```



5.6.6 Tóm tắt

- Ngoài việc kiểm soát số lượng kích thước và kích thước của vectơ trọng lượng, bỏ học là một công cụ khác để tránh quá mức. Thường thì chúng được sử dụng cùng nhau.
- Dropout thay thế một kích hoạt h bằng một biến ngẫu nhiên với giá trị kỳ vọng h.
- Dropout chỉ được sử dụng trong quá trình đào tạo.

5.6.7 Bài tập

1. Điều gì sẽ xảy ra nếu bạn thay đổi xác suất bỏ học cho các lớp thứ nhất và thứ hai? Đặc biệt, điều gì sẽ xảy ra nếu bạn chuyển đổi những cái cho cả hai lớp? Thiết kế một thử nghiệm để trả lời những câu hỏi này, mô tả kết quả của bạn một cách định lượng và tóm tắt các takeaways định tính.
2. Tăng số lượng thời đại và so sánh kết quả thu được khi sử dụng bỏ học với những người khi không sử dụng nó.
3. Phương sai của các kích hoạt trong mỗi lớp ẩn khi bỏ học và không được áp dụng là gì? Vẽ một âm mưu để hiển thị số lượng này phát triển như thế nào theo thời gian cho cả hai mô hình.
4. Tại sao bỏ học thường không được sử dụng tại thời điểm thử nghiệm?
5. Sử dụng mô hình trong phần này làm ví dụ, so sánh các hiệu ứng của việc sử dụng bỏ học và phân rã trọng lượng. Điều gì xảy ra khi bỏ học và phân rã trọng lượng được sử dụng cùng một lúc? Có phụ gia kết quả không? Có lợi nhuận giảm (hoặc tệ hơn)? Họ có hủy bỏ nhau không?
6. Điều gì sẽ xảy ra nếu chúng ta áp dụng bỏ học cho các trọng lượng riêng lẻ của ma trận trọng lượng chứ không phải là kích hoạt?
7. Phát minh ra một kỹ thuật khác để tiêm nhiễu ngẫu nhiên ở mỗi lớp khác với kỹ thuật bỏ học tiêu chuẩn. Bạn có thể phát triển một phương pháp vượt trội hơn việc bỏ học trên tập dữ liệu Fashion-MNIST (cho một kiến trúc cố định)?

Discussions[68]

5.7 Chuyển tiếp tuyên truyền, tuyên truyền ngược, và đồ thị tính toán

Cho đến nay, chúng tôi đã đào tạo các mô hình của chúng tôi với dòng dốc ngẫu nhiên minibatch. Tuy nhiên, khi chúng tôi thực hiện thuật toán, chúng tôi chỉ lo lắng về các tính toán liên quan đến *forward propagation* thông qua mô hình. Khi đến lúc tính toán gradient, chúng ta chỉ gọi hàm backpropagation được cung cấp bởi khung học sâu.

Việc tính toán tự động độ dốc (phân biệt tự động) đơn giản hóa sâu sắc việc thực hiện các thuật toán học sâu. Trước khi phân biệt tự động, ngay cả những thay đổi nhỏ đối với các mô hình phức tạp đòi hỏi phải tính toán lại các dẫn xuất phức tạp bằng tay. Đáng ngạc nhiên thường xuyên, các bài báo học thuật đã phải phân bổ nhiều trang để đưa ra các quy tắc cập nhật. Mặc dù chúng ta phải tiếp tục dựa vào sự khác biệt tự động để chúng ta có thể tập trung vào các phần thú vị, bạn nên biết làm thế nào các gradient được tính toán dưới mui xe nếu bạn muốn vượt ra ngoài một sự hiểu biết nông cạn về học sâu.

Trong phần này, chúng tôi đi sâu vào các chi tiết của * tuyên truyền ngược * (thường được gọi là *backpropagation*). Để truyền tải một số cái nhìn sâu sắc cho cả kỹ thuật và triển khai của chúng, chúng tôi dựa vào một số toán học cơ bản và đồ thị tính toán. Để bắt đầu, chúng tôi tập trung triển lãm của mình vào MLP một lớp ẩn với sự phân rã trọng lượng (L_2 đều đặn).

5.7.1 Chuyển tiếp tuyên truyền

Tuyên truyền chuyển tiếp (hoặc * chuyển tiếp*) đề cập đến việc tính toán và lưu trữ của các biến trung gian (bao gồm cả đầu ra) cho một mạng thần kinh theo thứ tự từ lớp đầu vào đến lớp đầu ra. Bây giờ chúng tôi làm việc từng bước thông qua cơ chế của một mạng thần kinh với một lớp ẩn. Điều này có vẻ tẻ nhạt nhưng theo lời vĩnh cửu của nghệ sĩ funk James Brown, bạn phải "trả chi phí để trở thành ông chủ".

Vì lợi ích của sự đơn giản, chúng ta hãy giả định rằng ví dụ đầu vào là $\mathbf{x} \in \mathbb{R}^d$ và lớp ẩn của chúng ta không bao gồm một thuật ngữ thiên vị. Ở đây biến trung gian là:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \tag{5.7.1}$$

trong đó $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ là tham số trọng lượng của lớp ẩn. Sau khi chạy biến trung gian $\mathbf{z} \in \mathbb{R}^h$ thông qua chức năng kích hoạt ϕ, chúng tôi có được vector kích hoạt ẩn của chúng tôi có chiều dài h,

$$\mathbf{h} = \phi(\mathbf{z}). \tag{5.7.2}$$

Biến ẩn $\mathbf{h}$ cũng là một biến trung gian. Giả sử rằng các tham số của lớp đầu ra chỉ sở hữu trọng lượng $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, ta có thể thu được một biến lớp đầu ra với một vectơ có chiều dài q:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \tag{5.7.3}$$

Giả sử hàm mất là l và nhãn ví dụ là y, sau đó chúng ta có thể tính toán thuật ngữ mất mát cho một ví dụ dữ liệu duy nhất,

$$L = l(\mathbf{o}, y). \tag{5.7.4}$$

Theo định nghĩa của L_2 chính quy hóa, với siêu tham số λ, thuật ngữ chính quy hóa là

$$s = \frac{\lambda}{2}\left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2\right), \tag{5.7.5}$$

[68] https://discuss.d2l.ai/t/101

trong đó định mức Frobenius của ma trận chỉ đơn giản là định mức L_2 được áp dụng sau khi làm phẳng ma trận thành một vectơ. Cuối cùng, tổn thất thường xuyên của mô hình trên một ví dụ dữ liệu nhất định là:

$$J = L + s. \tag{5.7.6}$$

Chúng tôi đề cập đến J là chức năng mục tiêu *trong cuộc thảo luận sau đây.

5.7.2 Đồ thị tính toán của chuyển tiếp tuyên truyền

Vẽ đồ thị tính toán * giúp chúng ta hình dung các phụ thuộc của toán tử và biến trong phép tính. Fig. 5.7.1 chứa đồ thị liên kết với mạng đơn giản được mô tả ở trên, trong đó các ô vuông biểu thị các biến và vòng tròn biểu thị các toán tử. Góc dưới bên trái biểu thị đầu vào và góc trên bên phải là đầu ra. Lưu ý rằng các hướng của các mũi tên (minh họa luồng dữ liệu) chủ yếu là bên phải và hướng lên.

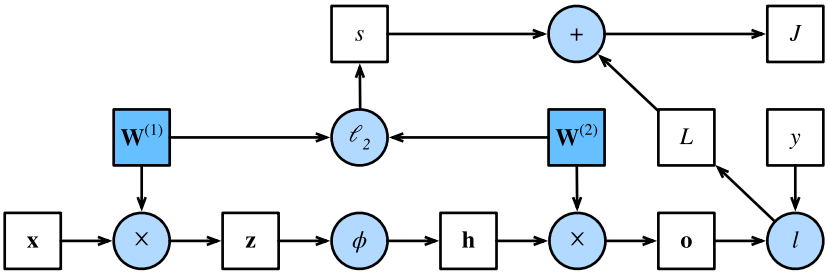


Fig. 5.7.1: Computational graph of forward propagation.

5.7.3 Backpropagation

Backpropagation đề cập đến phương pháp tính toán độ dốc của các tham số mạng thần kinh. Nói tóm lại, phương pháp đi qua mạng theo thứ tự ngược lại, từ đầu ra đến lớp đầu vào, theo quy tắc chuỗi *từ tính toán. Thuật toán lưu trữ bất kỳ biến trung gian nào (dẫn xuất từng phần) cần thiết trong khi tính toán gradient đối với một số tham số. Giả sử rằng chúng ta có chức năng $\mathsf{Y} = f(\mathsf{X})$ và $\mathsf{Z} = g(\mathsf{Y})$, trong đó đầu vào và đầu ra $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ là hàng chục hình dạng tùy ý. Bằng cách sử dụng quy tắc chuỗi, chúng ta có thể tính toán đạo hàm của Z đối với X qua

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \text{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}}\right). \tag{5.7.7}$$

Ở đây chúng tôi sử dụng toán tử prod để nhân các đối số của nó sau các hoạt động cần thiết, chẳng hạn như chuyển vị và trao đổi vị trí đầu vào, đã được thực hiện. Đối với vectơ, điều này rất đơn giản: nó chỉ đơn giản là phép nhân ma trận ma trận. Đối với hàng chục chiều cao hơn, chúng tôi sử dụng đối tác thích hợp. Các nhà điều hành prod ẩn tất cả các ký hiệu trên cao.

Nhớ lại rằng các tham số của mạng đơn giản với một lớp ẩn, có biểu đồ tính toán là trong Fig. 5.7.1, là $\mathbf{W}^{(1)}$ và $\mathbf{W}^{(2)}$. Mục tiêu của sự lan truyền ngược là tính toán độ dốc $\partial J/\partial \mathbf{W}^{(1)}$ và $\partial J/\partial \mathbf{W}^{(2)}$. Để thực hiện điều này, chúng ta áp dụng quy tắc chuỗi và tính toán, lần lượt, gradient của mỗi biến và tham số trung gian. Thứ tự các tính toán được đảo ngược so với các tính toán được thực hiện trong quá trình truyền chuyển tiếp, vì chúng ta cần bắt đầu với kết quả của biểu đồ tính toán và làm việc theo cách của chúng tôi hướng tới các tham số. Bước đầu tiên là tính toán độ dốc của hàm khách quan $J = L + s$ liên quan đến thuật ngữ mất L và thuật ngữ chính quy hóa s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1. \tag{5.7.8}$$

Tiếp theo, chúng ta tính toán gradient của hàm mục tiêu đối với biến của lớp đầu ra **o** theo quy tắc chuỗi:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \tag{5.7.9}$$

Tiếp theo, chúng ta tính toán độ dốc của thuật ngữ chính quy hóa đối với cả hai tham số:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \tag{5.7.10}$$

Bây giờ chúng ta có thể tính toán gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ của các tham số mô hình gần nhất với lớp đầu ra. Sử dụng sản lượng quy tắc chuỗi:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \tag{5.7.11}$$

Để có được gradient đối với $\mathbf{W}^{(1)}$, chúng ta cần tiếp tục lan truyền ngược dọc theo lớp đầu ra đến lớp ẩn. Gradient liên quan đến đầu ra của lớp ẩn $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ được đưa ra bởi

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}. \tag{5.7.12}$$

Vì hàm kích hoạt ϕ áp dụng elementwise, tính gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ của biến trung gian **z** yêu cầu chúng ta sử dụng toán tử nhân elementwise, mà chúng ta biểu thị bằng $\odot$:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right). \tag{5.7.13}$$

Cuối cùng, chúng ta có thể lấy gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ của các tham số mô hình gần nhất với lớp đầu vào. Theo quy tắc chuỗi, chúng tôi nhận được

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \tag{5.7.14}$$

5.7.4 Đào tạo mạng nơ-ron

Khi đào tạo mạng thần kinh, sự lan truyền về phía trước và lạc hậu phụ thuộc vào nhau. Đặc biệt, để tuyên truyền chuyển tiếp, chúng ta đi qua đồ thị tính toán theo hướng phụ thuộc và tính toán tất cả các biến trên đường đi của nó. Chúng sau đó được sử dụng để truyền ngược nơi thứ tự tính toán trên đồ thị bị đảo ngược.

Lấy mạng đơn giản đã nói ở trên làm ví dụ để minh họa. Một mặt, tính toán thuật ngữ chính quy hóa (5.7.5) trong quá trình truyền chuyển tiếp phụ thuộc vào các giá trị hiện tại của các tham số mô hình $\mathbf{W}^{(1)}$ và $\mathbf{W}^{(2)}$. Chúng được đưa ra bởi thuật toán tối ưu hóa theo backpropagation trong lần lặp mới nhất. Mặt khác, phép tính gradient cho tham số (5.7.11) trong quá trình truyền ngược phụ thuộc vào giá trị hiện tại của biến ẩn **h**, được đưa ra bằng cách lan truyền chuyển tiếp.

Do đó khi đào tạo mạng nơ-ron, sau khi các tham số mô hình được khởi tạo, chúng ta thay thế lan truyền chuyển tiếp với truyền ngược, cập nhật các tham số mô hình bằng cách sử dụng gradient được đưa ra bởi backpropagation. Lưu ý rằng backpropagation sử dụng lại các giá trị trung gian được lưu trữ từ chuyển tiếp tuyên truyền để tránh tính toán trùng lặp. Một trong những hậu quả là chúng ta cần giữ lại các giá trị trung gian cho đến khi truyền ngược hoàn tất. Đây cũng là một trong những lý do tại sao đào tạo đòi hỏi nhiều trí nhớ hơn đáng kể so với dự đoán đơn giản. Bên cạnh đó, kích thước của các giá trị trung gian như vậy là gần tỷ lệ thuận với số lượng lớp mạng và kích thước lô. Do đó, đào tạo các mạng sâu hơn bằng cách sử dụng kích thước lô lớn hơn dễ dàng dẫn đến * ra khỏi bộ nhớ* lỗi.

5.7.5 Tóm tắt

- Chuyển tiếp tuyên truyền tuần tự tính toán và lưu trữ các biến trung gian trong đồ thị tính toán được xác định bởi mạng nơ-ron. Nó tiến hành từ đầu vào đến lớp đầu ra.
- Backpropagation tuần tự tính toán và lưu trữ các gradient của các biến và tham số trung gian trong mạng nơ-ron theo thứ tự đảo ngược.
- Khi đào tạo các mô hình học sâu, sự lan truyền về phía trước và lan truyền trở lại phụ thuộc lẫn nhau.
- Đào tạo đòi hỏi nhiều trí nhớ hơn đáng kể so với dự đoán.

5.7.6 Bài tập

1. Giả sử rằng các đầu vào $\mathbf{X}$ đến một số hàm vô hướng f là ma trận $n \times m$. Chiều của gradient của f đối với $\mathbf{X}$ là gì?
2. Thêm một thiên vị vào lớp ẩn của mô hình được mô tả trong phần này (bạn không cần phải bao gồm thiên vị trong thuật ngữ chính quy hóa).
    1. Vẽ đồ thị tính toán tương ứng.
    2. Lấy được các phương trình tuyên truyền tiến và lạc hậu.
3. Tính toán dấu chân bộ nhớ để đào tạo và dự đoán trong mô hình được mô tả trong phần này.
4. Giả sử rằng bạn muốn tính toán các dẫn xuất thứ hai. Điều gì xảy ra với đồ thị tính toán? Bạn mong đợi tính toán mất bao lâu?
5. Giả sử rằng đồ thị tính toán quá lớn đối với GPU của bạn.
    1. Bạn có thể phân vùng nó trên nhiều GPU không?
    2. Những ưu điểm và nhược điểm so với đào tạo trên một minibatch nhỏ hơn là gì?

Discussions[69]

5.8 Tính ổn định số và khởi tạo

Cho đến nay, mọi mô hình mà chúng tôi đã thực hiện đều yêu cầu chúng tôi khởi tạo các tham số của nó theo một số phân phối được chỉ định trước. Cho đến bây giờ, chúng tôi đã thực hiện sơ đồ khởi tạo là điều hiển nhiên, nêu rõ các chi tiết về cách các lựa chọn này được thực hiện. Bạn thậm chí có thể đã nhận được ấn tượng rằng những lựa chọn này không đặc biệt quan trọng. Ngược lại, việc lựa chọn sơ đồ khởi tạo đóng một vai trò quan trọng trong việc học mạng thần kinh, và nó có thể rất quan trọng để duy trì sự ổn định số. Hơn nữa, những lựa chọn này có thể được gắn lên theo những cách thú vị với sự lựa chọn của chức năng kích hoạt phi tuyến. Chức năng nào chúng ta chọn và cách chúng ta khởi tạo các tham số có thể xác định thuật toán tối ưu hóa của chúng ta hội tụ nhanh như thế nào. Những lựa chọn kém ở đây có thể khiến chúng ta gặp phải sự bùng nổ hoặc biến mất gradient trong khi đào tạo. Trong phần này, chúng tôi đi sâu vào các chủ đề này với chi tiết hơn và thảo luận về một số heuristics hữu ích mà bạn sẽ thấy hữu ích trong suốt sự nghiệp của mình trong việc học sâu.

[69] https://discuss.d2l.ai/t/102

5.8.1 Biến mất và bùng nổ Gradient

Hãy xem xét một mạng sâu với L lớp, đầu vào $\mathbf{x}$ và đầu ra $\mathbf{o}$. Với mỗi lớp l được xác định bởi một biến đổi f_l tham số hóa bởi trọng lượng $\mathbf{W}^{(l)}$, có biến ẩn là $\mathbf{h}^{(l)}$ (để $\mathbf{h}^{(0)} = \mathbf{x}$), mạng của chúng tôi có thể được biểu thị như:

$$\mathbf{h}^{(l)} = f_l(\mathbf{h}^{(l-1)}) \text{ and thus } \mathbf{o} = f_L \circ \ldots \circ f_1(\mathbf{x}). \tag{5.8.1}$$

Nếu tất cả các biến ẩn và đầu vào là vectơ, chúng ta có thể viết gradient của $\mathbf{o}$ đối với bất kỳ tập hợp các tham số $\mathbf{W}^{(l)}$ như sau:

$$\partial_{\mathbf{W}^{(l)}} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{(L-1)}} \mathbf{h}^{(L)}}_{\mathbf{M}^{(L)} \stackrel{\text{def}}{=}} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^{(l)}} \mathbf{h}^{(l+1)}}_{\mathbf{M}^{(l+1)} \stackrel{\text{def}}{=}} \underbrace{\partial_{\mathbf{W}^{(l)}} \mathbf{h}^{(l)}}_{\mathbf{v}^{(l)} \stackrel{\text{def}}{=}}. \tag{5.8.2}$$

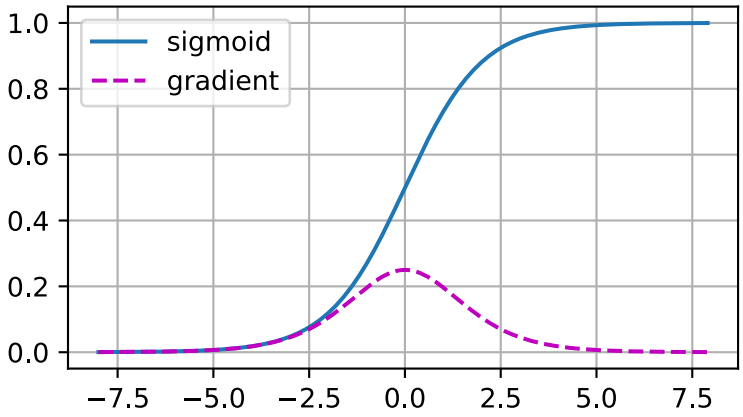
Nói cách khác, gradient này là sản phẩm của ma trận $L - l$ $\mathbf{M}^{(L)} \cdot \ldots \cdot \mathbf{M}^{(l+1)}$ và vector gradient $\mathbf{v}^{(l)}$. Do đó, chúng ta dễ bị các vấn đề tương tự của dòng chảy số thường cắt lên khi nhân với nhau quá nhiều xác suất. Khi đối phó với xác suất, một mẹo phổ biến là chuyển sang không gian đăng nhập, tức là, chuyển áp lực từ mantissa sang số mũ của biểu diễn số. Thật không may, vấn đề của chúng tôi ở trên nghiêm trọng hơn: ban đầu các ma trận $\mathbf{M}^{(l)}$ có thể có nhiều eigenvalues. Chúng có thể nhỏ hoặc lớn, và sản phẩm của họ có thể là * rất lớn* hoặc * rất nhỏ*.

Các rủi ro gây ra bởi gradient không ổn định vượt ra ngoài biểu diễn số. Độ dốc có độ lớn không thể đoán trước cũng đe dọa sự ổn định của các thuật toán tối ưu hóa của chúng tôi. Chúng ta có thể phải đối mặt với các bản cập nhật tham số (i) quá lớn, phá hủy mô hình của chúng tôi (vấn đề * exploding gradient*); hoặc (ii) quá nhỏ (vấn đề * vanishing gradient*), khiến việc học không thể như các tham số hầu như không di chuyển trên mỗi bản cập nhật.

Vanishing Gradients

Một thủ phạm thường xuyên gây ra vấn đề gradient biến mất là sự lựa chọn của hàm kích hoạt σ được nối sau các phép toán tuyến tính của mỗi lớp. Trong lịch sử, hàm sigmoid $1/(1 + \exp(-x))$ (được giới thiệu vào năm Section 5.1) rất phổ biến vì nó giống như một hàm ngưỡng. Vì các mạng thần kinh nhân tạo ban đầu được lấy cảm hứng từ các mạng thần kinh sinh học, ý tưởng về các tế bào thần kinh bắn * đầy đủ* hoặc * không* (như tế bào thần kinh sinh học) dường như hấp dẫn. Chúng ta hãy xem xét kỹ hơn sigmoid để xem lý do tại sao nó có thể gây ra gradient biến mất.

```
%matplotlib inline
import torch
from d2l import torch as d2l

x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.sigmoid(x)
y.backward(torch.ones_like(x))

d2l.plot(x.detach().numpy(), [y.detach().numpy(), x.grad.numpy()],
         legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```



Như bạn có thể thấy, gradient của sigmoid biến mất cả khi đầu vào của nó lớn và khi chúng nhỏ. Hơn nữa, khi truyền ngược qua nhiều lớp, trừ khi chúng ta ở trong vùng Goldilocks, nơi các đầu vào cho nhiều sigmoids gần bằng 0, độ dốc của sản phẩm tổng thể có thể biến mất. Khi mạng của chúng tôi tự hào có nhiều lớp, trừ khi chúng tôi cẩn thận, gradient có thể sẽ bị cắt ở một số lớp. Thật vậy, vấn đề này được sử dụng để đào tạo mạng sâu bệnh dịch hạch. Do đó, ReLUs, ổn định hơn (nhưng ít hợp lý về mặt thần kinh), đã nổi lên như là lựa chọn mặc định cho các học viên.

Bùng nổ Gradients

Vấn đề ngược lại, khi gradient nổ tung, có thể tương tự như vexing. Để minh họa điều này tốt hơn một chút, chúng tôi vẽ 100 ma trận ngẫu nhiên Gaussian và nhân chúng với một số ma trận ban đầu. Đối với thang đo mà chúng tôi đã chọn (sự lựa chọn của phương sai $\sigma^2 = 1$), sản phẩm ma trận phát nổ. Khi điều này xảy ra do khởi tạo một mạng sâu, chúng ta không có cơ hội nhận được trình tối ưu hóa gradient descent để hội tụ.

```
M = torch.normal(0, 1, size=(4,4))
print('a single matrix \n',M)
for i in range(100):
    M = torch.mm(M,torch.normal(0, 1, size=(4, 4)))

print('after multiplying 100 matrices\n', M)
```

```
a single matrix
 tensor([[-0.1369, -0.3347, -0.3115, -1.6260],
        [ 1.4966,  1.0600, -1.2315, -0.5771],
        [ 1.4038, -0.4152,  0.4013,  0.1608],
        [ 1.3121,  0.5267, -1.3593,  1.8891]])
after multiplying 100 matrices
 tensor([[ 4.8421e+26, -1.6528e+27,  8.2065e+26,  2.0016e+26],
        [-2.2945e+25,  7.8321e+25, -3.8888e+25, -9.4848e+24],
        [ 9.6323e+26, -3.2878e+27,  1.6325e+27,  3.9816e+26],
        [-6.7803e+26,  2.3144e+27, -1.1491e+27, -2.8027e+26]])
```

Phá vỡ đối xứng

Một vấn đề khác trong thiết kế mạng thần kinh là sự đối xứng vốn có trong tham số hóa của chúng. Giả sử rằng chúng ta có một MLP đơn giản với một lớp ẩn và hai đơn vị. Trong trường hợp này, chúng ta có thể hoán vị trọng lượng $\mathbf{W}^{(1)}$ của lớp đầu tiên và tương tự như vậy hoán vị trọng lượng của lớp đầu ra để có được chức năng tương tự. Không có gì đặc biệt phân biệt đơn vị ẩn đầu tiên so với đơn vị ẩn thứ hai. Nói cách khác, chúng ta có sự đối xứng hoán vị giữa các đơn vị ẩn của mỗi lớp.

Đây không chỉ là một phiền toái lý thuyết. Hãy xem xét MLP một lớp ẩn đã nói ở trên với hai đơn vị ẩn. Để minh họa, giả sử lớp đầu ra biến đổi hai đơn vị ẩn thành chỉ một đơn vị đầu ra. Hãy tưởng tượng điều gì sẽ xảy ra nếu chúng ta khởi tạo tất cả các tham số của lớp ẩn là $\mathbf{W}^{(1)} = c$ cho một số hằng số c. Trong trường hợp này, trong quá trình tuyên truyền chuyển tiếp hoặc đơn vị ẩn có cùng một đầu vào và tham số, tạo ra cùng một kích hoạt, được đưa vào đơn vị đầu ra. Trong quá trình lan truyền ngược, phân biệt đơn vị đầu ra đối với các tham số $\mathbf{W}^{(1)}$ cho một gradient có các phần tử có cùng giá trị. Do đó, sau khi lặp lại dựa trên gradient (ví dụ, minibatch stochastic gradient descent), tất cả các phần tử của $\mathbf{W}^{(1)}$ vẫn có cùng giá trị. Các lần lặp như vậy sẽ không bao giờ * tự phá vỡ đối xưởng* và chúng ta có thể không bao giờ có thể nhận ra sức mạnh biểu cảm của mạng. Lớp ẩn sẽ hoạt động như thể nó chỉ có một đơn vị duy nhất. Lưu ý rằng trong khi minibatch stochastic gradient gốc sẽ không phá vỡ đối xứng này, việc bỏ học thường xuyên sẽ!

5.8.2 Khởi tạo tham số

Một cách để giải quyết - hoặc ít nhất là giảm thiểu - các vấn đề nêu trên là thông qua việc khởi tạo cẩn thận. Chăm sóc bổ sung trong quá trình tối ưu hóa và chính quy hóa phù hợp có thể tăng cường hơn nữa sự ổn định.

Khởi tạo mặc định

Trong các phần trước, ví dụ, trong Section 4.3, chúng tôi đã sử dụng một phân phối bình thường để khởi tạo các giá trị của trọng lượng của chúng tôi. Nếu chúng ta không chỉ định phương thức khởi tạo, framework sẽ sử dụng phương thức khởi tạo ngẫu nhiên mặc định, thường hoạt động tốt trong thực tế cho kích thước vấn đề vừa phải.

Khởi tạo Xavier

Chúng ta hãy xem xét phân phối tỷ lệ của một đầu ra (ví dụ, một biến ẩn) o_i cho một số lớp được kết nối hoàn toàn *không có phi tuyến tính*. Với n_{in} đầu vào x_j và trọng lượng liên quan của chúng w_{ij} cho lớp này, một đầu ra được đưa ra bởi

$$o_i = \sum_{j=1}^{n_{\text{in}}} w_{ij} x_j. \tag{5.8.3}$$

Các trọng lượng w_{ij} đều được vẽ độc lập từ cùng một phân phối. Hơn nữa, chúng ta hãy giả định rằng phân phối này có 0 trung bình và phương sai σ^2. Lưu ý rằng điều này không có nghĩa là phân phối phải là Gaussian, chỉ là trung bình và phương sai cần phải tồn tại. Hiện tại, chúng ta hãy giả định rằng các đầu vào cho lớp x_j cũng có 0 trung bình và phương sai γ^2 và chúng độc lập với w_{ij} và độc lập với nhau. Trong trường hợp này,

chúng ta có thể tính toán trung bình và phương sai của o_i như sau:

$$\begin{aligned}
E[o_i] &= \sum_{j=1}^{n_{\text{in}}} E[w_{ij} x_j] \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}] E[x_j] \\
&= 0, \\
\text{Var}[o_i] &= E[o_i^2] - (E[o_i])^2 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2 x_j^2] - 0 \\
&= \sum_{j=1}^{n_{\text{in}}} E[w_{ij}^2] E[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2.
\end{aligned} \tag{5.8.4}$$

Một cách để giữ phương sai cố định là đặt $n_{\text{in}}\sigma^2 = 1$. Bây giờ hãy xem xét backpropagation. Ở đó chúng ta phải đối mặt với một vấn đề tương tự, mặc dù với gradient được truyền từ các lớp gần đầu ra hơn. Sử dụng lý luận tương tự như đối với tuyên truyền chuyển tiếp, chúng ta thấy rằng phương sai của gradient có thể thổi lên trừ khi $n_{\text{out}}\sigma^2 = 1$, trong đó n_{out} là số lượng đầu ra của lớp này. Điều này khiến chúng ta gặp khó xử: chúng ta không thể thỏa mãn cả hai điều kiện cùng một lúc. Thay vào đó, chúng tôi chỉ đơn giản là cố gắng thỏa mãn:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{5.8.5}$$

Đây là lý do dựa trên chuẩn hiện nay và thực tế có lợi * Xavier khởi hóa*, được đặt tên theo tác giả đầu tiên của người tạo ra nó (Glorot & Bengio, 2010). Thông thường, các mẫu khởi tạo Xavier trọng lượng từ phân phối Gaussian với 0 trung bình và phương sai $\sigma^2 = \frac{2}{n_{\text{in}}+n_{\text{out}}}$. Chúng ta cũng có thể điều chỉnh trực giác của Xavier để chọn phương sai khi lấy mẫu trọng lượng từ phân phối đồng đều. Lưu ý rằng sự phân bố thống nhất $U(-a, a)$ có phương sai $\frac{a^2}{3}$. Cắm $\frac{a^2}{3}$ vào điều kiện của chúng tôi trên σ^2 mang lại gợi ý khởi tạo theo

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{5.8.6}$$

Mặc dù giả định cho sự không tồn tại của phi tuyến tính trong lý luận toán học trên có thể dễ dàng vi phạm trong các mạng thần kinh, phương pháp khởi tạo Xavier hóa ra hoạt động tốt trong thực tế.

Beyond

Lý do trên hầu như không làm trầy xước bề mặt của các phương pháp tiếp cận hiện đại để khởi tạo tham số. Một khuôn khổ học sâu thường thực hiện hơn một chục heuristics khác nhau. Hơn nữa, khởi tạo tham số tiếp tục là một lĩnh vực nóng của nghiên cứu cơ bản trong học sâu. Trong số này có heuristics chuyên cho các tham số gắn (chia sẻ), siêu phân giải, mô hình trình tự và các tình huống khác. Ví dụ, Xiao et al. đã chứng minh khả năng đào tạo mạng thần kinh 10000 lớp mà không cần thủ thuật kiến trúc bằng cách sử dụng phương pháp khởi tạo được thiết kế cẩn thận (Xiao et al., 2018).

Nếu chủ đề mà bạn quan tâm, chúng tôi đề nghị đi sâu vào các dịch vụ của mô-đun này, đọc các bài báo đề xuất và phân tích từng heuristic, sau đó khám phá các ấn phẩm mới nhất về chủ đề này. Có lẽ bạn sẽ vấp ngã hoặc thậm chí phát minh ra một ý tưởng thông minh và đóng góp thực hiện cho các khuôn khổ học sâu.

5.8.3 Tóm tắt

- Biến mất và bùng nổ gradient là những vấn đề phổ biến trong các mạng sâu. Chăm sóc tuyệt vời trong khởi tạo tham số là cần thiết để đảm bảo rằng độ dốc và tham số vẫn được kiểm soát tốt.
- Khởi tạo heuristics là cần thiết để đảm bảo rằng các gradient ban đầu không quá lớn cũng không quá nhỏ.
- Chức năng kích hoạt ReLU giảm thiểu vấn đề gradient biến mất. Điều này có thể đẩy nhanh sự hội tụ.
- Khởi tạo ngẫu nhiên là chìa khóa để đảm bảo rằng đối xứng bị phá vỡ trước khi tối ưu hóa.
- Khởi tạo Xavier gợi ý rằng, đối với mỗi lớp, phương sai của bất kỳ đầu ra nào không bị ảnh hưởng bởi số lượng đầu vào và phương sai của bất kỳ gradient nào không bị ảnh hưởng bởi số lượng đầu ra.

5.8.4 Bài tập

1. Bạn có thể thiết kế các trường hợp khác trong đó một mạng thần kinh có thể thể hiện đối xứng đòi hỏi phải phá vỡ bên cạnh sự đối xứng hoán vị trong các lớp của MLP?
2. Chúng ta có thể khởi tạo tất cả các tham số trọng lượng trong hồi quy tuyến tính hoặc hồi quy softmax đến cùng một giá trị?
3. Tra cứu giới hạn phân tích trên eigenvalues của sản phẩm của hai ma trận. Điều này cho bạn biết gì về việc đảm bảo rằng độ dốc được điều hòa tốt?
4. Nếu chúng ta biết rằng một số thuật ngữ phân kỳ, chúng ta có thể khắc phục điều này sau khi thực tế? Nhìn vào bài báo về tỷ lệ thích ứng layerwise tỷ lệ cho cảm hứng (You et al., 2017).

Discussions[70]

5.9 Môi trường và phân phối Shift

Trong các phần trước, chúng tôi đã làm việc thông qua một số ứng dụng thực hành của machine learning, phù hợp với các mô hình cho nhiều bộ dữ liệu khác nhau. Tuy nhiên, chúng tôi không bao giờ dừng lại để chiêm ngưỡng dữ liệu đến từ đâu ở nơi đầu tiên hoặc những gì chúng tôi dự định cuối cùng làm với các đầu ra từ các mô hình của chúng tôi. Quá thường xuyên, các nhà phát triển machine learning sở hữu dữ liệu vội vàng phát triển các mô hình mà không dừng lại để xem xét những vấn đề cơ bản này.

Nhiều triển khai machine learning thất bại có thể được truy trở lại mô hình này. Đôi khi các mô hình dường như thực hiện tuyệt vời như được đo bằng độ chính xác của bộ thử nghiệm nhưng thất bại thảm khốc trong việc triển khai khi phân phối dữ liệu đột ngột thay đổi. Ngấm ngầm hơn, đôi khi việc triển khai một mô hình có thể là chất xúc tác làm xáo trộn phân phối dữ liệu. Ví dụ, ví dụ, chúng tôi đã đào tạo một mô hình để dự đoán ai sẽ trả nợ so với mặc định cho vay, thấy rằng lựa chọn giày dép của người nộp đơn có liên quan đến rủi ro mặc định (Oxfords cho biết trả nợ, giày thể thao chỉ ra mặc định). Sau đó chúng tôi có thể có xu hướng cấp các khoản vay cho tất cả các ứng viên mặc Oxfords và từ chối tất cả các ứng viên mang giày thể thao.

Trong trường hợp này, bước nhảy vọt không được coi là của chúng tôi từ công nhận mô hình đến ra quyết định và việc chúng tôi không xem xét nghiêm túc môi trường có thể có hậu quả tai hại. Để bắt đầu, ngay khi chúng tôi bắt đầu đưa ra quyết định dựa trên giày dép, khách hàng sẽ bắt kịp và thay đổi hành vi của họ. Trước lâu, tất cả các ứng viên sẽ mặc Oxfords, mà không có bất kỳ sự cải thiện trùng hợp nào về giá trị tín dụng. Dành

[70] https://discuss.d2l.ai/t/104

một phút để tiêu hóa điều này bởi vì các vấn đề tương tự rất nhiều trong nhiều ứng dụng học máy: bằng cách giới thiệu các quyết định dựa trên mô hình của chúng tôi cho môi trường, chúng tôi có thể phá vỡ mô hình.

Mặc dù chúng tôi không thể cung cấp cho các chủ đề này một phương pháp điều trị hoàn chỉnh trong một phần, chúng tôi đặt mục tiêu ở đây để phơi bày một số mối quan tâm phổ biến và kích thích tư duy phê phán cần thiết để phát hiện sớm những tình huống này, giảm thiểu thiệt hại và sử dụng máy học một cách có trách nhiệm. Một số giải pháp rất đơn giản (yêu cầu dữ liệu "đúng"), một số là khó khăn về mặt kỹ thuật (thực hiện một hệ thống học tập tăng cường), và những người khác yêu cầu chúng ta bước ra ngoài lĩnh vực dự đoán thống kê hoàn toàn và vật lộn với các câu hỏi triết học khó liên quan đến ứng dụng đạo đức của algorithms thuật toán.

5.9.1 Các loại dịch chuyển phân phối

Để bắt đầu, chúng tôi gắn bó với cài đặt dự đoán thụ động xem xét các cách khác nhau mà các phân phối dữ liệu có thể thay đổi và những gì có thể được thực hiện để cứu vớt hiệu suất mô hình. Trong một thiết lập cổ điển, chúng tôi giả định rằng dữ liệu đào tạo của chúng tôi đã được lấy mẫu từ một số phân phối $p_S(\mathbf{x}, y)$ nhưng dữ liệu thử nghiệm của chúng tôi sẽ bao gồm các ví dụ không có nhãn được rút ra từ một số phân phối khác nhau $p_T(\mathbf{x}, y)$. Đã, chúng ta phải đối đầu với một thực tế tỉnh táo. Vắng mặt bất kỳ giả định nào về cách p_S và p_T liên quan đến nhau, việc học một bộ phân loại mạnh mẽ là không thể.

Hãy xem xét một vấn đề phân loại nhị phân, nơi chúng ta muốn phân biệt giữa chó và mèo. Nếu phân phối có thể thay đổi theo những cách tùy ý, thì thiết lập của chúng tôi cho phép trường hợp bệnh lý trong đó phân phối trên đầu vào vẫn không đổi: $p_S(\mathbf{x}) = p_T(\mathbf{x})$, nhưng các nhãn đều bị lật: $p_S(y|\mathbf{x}) = 1 - p_T(y|\mathbf{x})$. Nói cách khác, nếu Thiên Chúa đột nhiên có thể quyết định rằng trong tương lai tất cả các "mèo" bây giờ là chó và những gì trước đây chúng ta gọi là "chó" bây giờ là mèo - mà không có bất kỳ thay đổi nào trong phân phối đầu vào $p(\mathbf{x})$, thì chúng ta không thể phân biệt được cài đặt này với một trong đó phân phối không thay đổi chút nào.

May mắn thay, dưới một số giả định bị hạn chế về cách dữ liệu của chúng ta có thể thay đổi trong tương lai, các thuật toán nguyên tắc có thể phát hiện sự thay đổi và đôi khi thậm chí thích ứng ngay lập tức, cải thiện độ chính xác của phân loại ban đầu.

Covariate Shift

Trong số các loại dịch chuyển phân phối, sự thay đổi đồng biến có thể được nghiên cứu rộng rãi nhất. Ở đây, chúng tôi giả định rằng trong khi phân phối các đầu vào có thể thay đổi theo thời gian, chức năng ghi nhãn, tức là phân phối có điều kiện $P(y \mid \mathbf{x})$ không thay đổi. Các nhà thống kê gọi đây là * covariate shift* vì vấn đề phát sinh do sự thay đổi trong sự phân bố của các covariates (tính năng). Mặc dù đôi khi chúng ta có thể lý do về sự thay đổi phân phối mà không gọi nhân quả, chúng tôi lưu ý rằng sự thay đổi đồng biến là giả định tự nhiên để gọi trong các cài đặt mà chúng tôi tin rằng $\mathbf{x}$ gây ra y.

Hãy xem xét thách thức phân biệt mèo và chó. Dữ liệu đào tạo của chúng tôi có thể bao gồm các hình ảnh thuộc loại trong Fig. 5.9.1.

Fig. 5.9.1: Training data for distinguishing cats and dogs.

Tại thời điểm thử nghiệm, chúng tôi được yêu cầu phân loại hình ảnh trong Fig. 5.9.2.

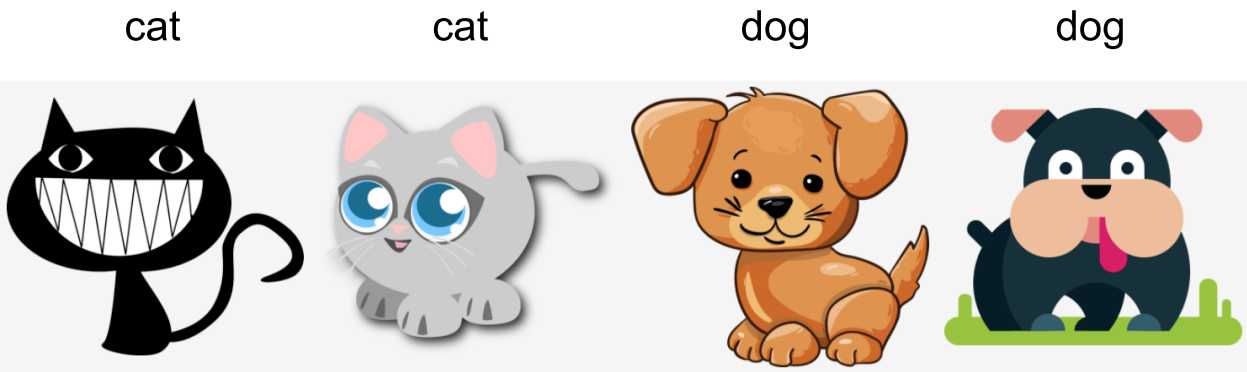


Fig. 5.9.2: Test data for distinguishing cats and dogs.

Bộ đào tạo bao gồm các bức ảnh, trong khi bộ thử nghiệm chỉ chứa phim hoạt hình. Đào tạo về một tập dữ liệu với các đặc điểm khác biệt đáng kể so với bộ thử nghiệm có thể đánh vần rắc rối vắng mặt một kế hoạch mạch lạc về cách thích ứng với tên miền mới.

Thay đổi nhãn

Label shift mô tả vấn đề converse. Ở đây, chúng tôi giả định rằng nhãn biên $P(y)$ có thể thay đổi nhưng phân phối lớp có điều kiện $P(\mathbf{x} \mid y)$ vẫn cố định trên các miền. Sự thay đổi nhãn là một giả định hợp lý để đưa ra khi chúng tôi tin rằng y gây ra $\mathbf{x}$. Ví dụ, chúng tôi có thể muốn dự đoán chẩn đoán cho các triệu chứng của họ (hoặc các biểu hiện khác), ngay cả khi tỷ lệ tương đối của chẩn đoán đang thay đổi theo thời gian. Thay đổi nhãn là giả định thích hợp ở đây vì bệnh gây ra các triệu chứng. Trong một số trường hợp thoái hóa, sự dịch chuyển nhãn và các giả định thay đổi thay đổi có thể giữ đồng thời. Ví dụ, khi nhãn là xác định, giả định thay đổi đồng biến sẽ được thỏa mãn, ngay cả khi y gây ra $\mathbf{x}$. Thật thú vị, trong những trường hợp này, nó thường thuận lợi để làm việc với các phương pháp chảy từ giả định thay đổi nhãn. Đó là bởi vì các phương pháp này có xu hướng liên quan đến việc thao tác các đối tượng trông giống như nhãn (thường là chiều thấp), trái ngược với các đối tượng trông giống như đầu vào, có xu hướng có chiều cao trong học sâu.

Concept Shift

Chúng tôi cũng có thể gặp phải vấn đề liên quan của *concept shift*, phát sinh khi các định nghĩa của nhãn có thể thay đổi. Điều này nghe có vẻ kỳ quặc một * cat* là một * cat*, không? Tuy nhiên, các danh mục khác có thể thay đổi sử dụng theo thời gian. Tiêu chí chẩn đoán cho bệnh tâm thần, những gì trôi qua cho thời trang và chức danh công việc, tất cả đều phải chịu một lượng đáng kể sự thay đổi khái niệm. Hóa ra nếu chúng ta điều hướng khắp Hoa Kỳ, chuyển nguồn dữ liệu của chúng ta theo địa lý, chúng ta sẽ tìm thấy sự thay đổi khái niệm đáng kể liên quan đến việc phân phối tên cho * đồ uống mềm* như thể hiện trong Fig. 5.9.3.

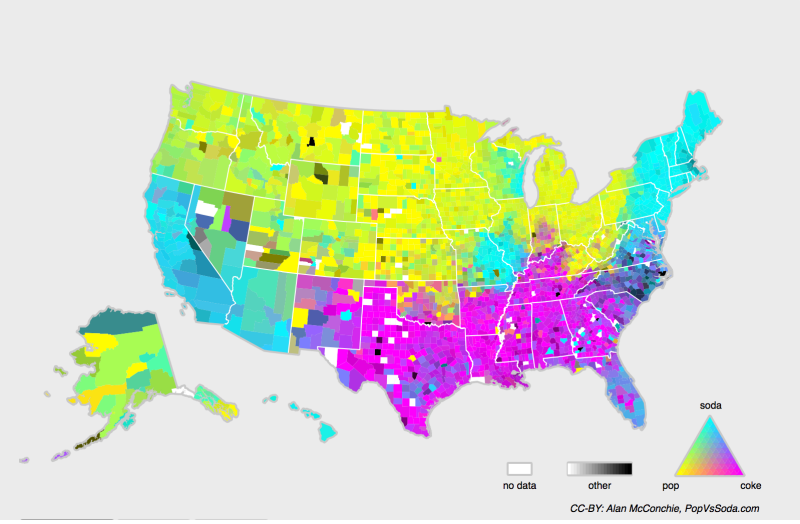


Fig. 5.9.3: Concept shift on soft drink names in the United States.

Nếu chúng ta xây dựng một hệ thống dịch máy, phân phối $P(y \mid \mathbf{x})$ có thể khác nhau tùy thuộc vào vị trí của chúng tôi. Vấn đề này có thể khó phát hiện. Chúng ta có thể hy vọng sẽ khai thác kiến thức rằng sự thay đổi chỉ diễn ra dần dần theo nghĩa thời gian hoặc địa lý.

5.9.2 Ví dụ về Distribution Shift

Trước khi đi sâu vào chủ nghĩa hình thức và thuật toán, chúng ta có thể thảo luận về một số tình huống cụ thể mà sự thay đổi chung hoặc khái niệm có thể không rõ ràng.

Chẩn đoán y tế

Hãy tưởng tượng rằng bạn muốn thiết kế một thuật toán để phát hiện ung thư. Bạn thu thập dữ liệu từ những người khỏe mạnh và bệnh tật và bạn đào tạo thuật toán của mình. Nó hoạt động tốt, mang lại cho bạn độ chính xác cao và bạn kết luận rằng bạn đã sẵn sàng cho một sự nghiệp thành công trong chẩn đoán y tế. *Không quá nhanh. *

Các bản phân phối đã tạo ra dữ liệu đào tạo và những người bạn sẽ gặp phải trong tự nhiên có thể khác nhau đáng kể. Điều này đã xảy ra với một công ty khởi nghiệp đáng tiếc mà một số người trong chúng ta (tác giả) đã làm việc với nhiều năm trước. Họ đang phát triển xét nghiệm máu cho một căn bệnh chủ yếu ảnh hưởng đến những người đàn ông lớn tuổi và hy vọng nghiên cứu nó bằng cách sử dụng các mẫu máu mà họ đã thu thập từ bệnh nhân. Tuy nhiên, việc lấy mẫu máu từ những người đàn ông khỏe mạnh là khó khăn hơn đáng kể so với bệnh nhân bị bệnh đã có trong hệ thống. Để bù đắp, công ty khởi nghiệp đã yêu cầu quyên góp máu từ các sinh viên trong khuôn viên trường đại học để đóng vai trò là biện pháp kiểm soát lành mạnh trong việc

phát triển bài kiểm tra của họ. Sau đó, họ hỏi liệu chúng tôi có thể giúp họ xây dựng một phân loại để phát hiện bệnh.

Như chúng tôi đã giải thích cho họ, thực sự sẽ dễ dàng phân biệt giữa các nhóm khỏe mạnh và bệnh tật với độ chính xác gần như hoàn hảo. Tuy nhiên, đó là do các đối tượng xét nghiệm khác nhau về độ tuổi, mức độ hormone, hoạt động thể chất, chế độ ăn uống, tiêu thụ rượu và nhiều yếu tố khác không liên quan đến bệnh. Điều này không có khả năng là trường hợp với bệnh nhân thực sự. Do quy trình lấy mẫu của họ, chúng ta có thể mong đợi sẽ gặp phải sự thay đổi đồng biến cực đoan. Hơn nữa, trường hợp này không có khả năng sửa chữa thông qua các phương pháp thông thường. Nói tóm lại, họ đã lãng phí một khoản tiền đáng kể.

Ô tô tự lái

Nói rằng một công ty muốn tận dụng máy học để phát triển những chiếc xe tự lái. Một thành phần quan trọng ở đây là một máy dò bên đường. Vì dữ liệu chú thích thực sự rất tốn kém để có được, họ đã có ý tưởng (thông minh và có vấn đề) để sử dụng dữ liệu tổng hợp từ một công cụ kết xuất trò chơi làm dữ liệu đào tạo bổ sung. Điều này hoạt động thực sự tốt trên "dữ liệu thử nghiệm" được rút ra từ công cụ kết xuất. Than ôi, bên trong một chiếc xe thực sự đó là một thảm họa. Khi nó bật ra, lề đường đã được kết xuất với một kết cấu rất đơn giản. Quan trọng hơn, * tất cả* bên lề đường đã được kết xuất với kết cấu * same* và máy dò bên đường đã tìm hiểu về "tính năng" này rất nhanh chóng.

Một điều tương tự cũng xảy ra với Quân đội Mỹ khi lần đầu tiên họ cố gắng phát hiện xe tăng trong rừng. Họ chụp những bức ảnh trên không của khu rừng không có xe tăng, sau đó lái xe tăng vào rừng và chụp một bộ ảnh khác. Bộ phân loại dường như hoạt động * hoàn hảo*. Thật không may, nó chỉ học cách phân biệt cây có bóng với cây không có bóng - bộ ảnh đầu tiên được chụp vào sáng sớm, bộ thứ hai vào buổi trưa.

Phân phối Nonstationary

Một tình huống tinh tế hơn nhiều phát sinh khi phân phối thay đổi chậm (còn được gọi là phân phối không cố định *) và mô hình không được cập nhật đầy đủ. Dưới đây là một số trường hợp điển hình.

- Chúng tôi đào tạo một mô hình quảng cáo tính toán và sau đó không cập nhật nó thường xuyên (ví dụ: chúng tôi quên kết hợp rằng một thiết bị mới tối nghĩa gọi là iPad vừa được khởi chạy).
- Chúng tôi xây dựng một bộ lọc thư rác. Nó hoạt động tốt trong việc phát hiện tất cả thư rác mà chúng ta đã thấy cho đến nay. Nhưng sau đó những kẻ gửi thư rác wisen lên và tạo ra những tin nhắn mới trông không giống như bất cứ điều gì chúng ta đã thấy trước đây.
- Chúng tôi xây dựng một hệ thống khuyến nghị sản phẩm. Nó hoạt động trong suốt mùa đông nhưng sau đó tiếp tục giới thiệu mũ ông già Noel từ lâu sau Giáng sinh.

Thêm giai thoại

- Chúng tôi xây dựng một máy dò khuôn mặt. Nó hoạt động tốt trên tất cả các điểm chuẩn. Thật không may, nó thất bại trên dữ liệu thử nghiệm - các ví dụ vi phạm là cận cảnh trong đó khuôn mặt lấp đầy toàn bộ hình ảnh (không có dữ liệu như vậy trong bộ đào tạo).
- Chúng tôi xây dựng một công cụ tìm kiếm Web cho thị trường Mỹ và muốn triển khai nó ở Anh.
- Chúng tôi đào tạo một bộ phân loại hình ảnh bằng cách biên dịch một tập dữ liệu lớn trong đó mỗi tập hợp lớn các lớp được biểu diễn như nhau trong tập dữ liệu, giả sử 1000 danh mục, được đại diện bởi 1000 hình ảnh mỗi. Sau đó, chúng tôi triển khai hệ thống trong thế giới thực, nơi phân phối nhãn thực tế của ảnh là quyết định không đồng nhất.

5.9.3 Sửa đổi sự thay đổi phân phối

Như chúng ta đã thảo luận, có nhiều trường hợp đào tạo và phân phối thử nghiệm $P(\mathbf{x}, y)$ khác nhau. Trong một số trường hợp, chúng tôi nhận được may mắn và các mô hình hoạt động mặc dù covariate, nhãn, hoặc thay đổi khái niệm. Trong các trường hợp khác, chúng ta có thể làm tốt hơn bằng cách sử dụng các chiến lược nguyên tắc để đối phó với sự thay đổi. Phần còn lại của phần này phát triển kỹ thuật hơn đáng kể. Người đọc thiếu kiên nhẫn có thể tiếp tục đến phần tiếp theo vì tài liệu này không phải là điều kiện tiên quyết cho các khái niệm tiếp theo.

Rủi ro và rủi ro thực nghiệm

Trước tiên chúng ta hãy phản ánh về những gì chính xác đang xảy ra trong quá trình đào tạo mô hình: chúng tôi lặp lại các tính năng và nhãn liên quan của dữ liệu đào tạo $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ và cập nhật các thông số của một mô hình f sau mỗi minibatch. Để đơn giản, chúng tôi không xem xét chính quy hóa, vì vậy chúng tôi phần lớn giảm thiểu sự mất mát trong đào tạo:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}_i), y_i), \tag{5.9.1}$$

trong đó l là chức năng mất đo "xấu như thế nào" dự đoán $f(\mathbf{x}_i)$ được đưa ra nhãn liên quan y_i. Các nhà thống kê gọi thuật ngữ này trong (5.9.1) * rủi ro thực tế*. Rủi ro thực nghiệm * là tổn thất trung bình so với dữ liệu đào tạo để xấp xỉ * rủi ro*, đó là kỳ vọng về sự mất mát đối với toàn bộ dân số dữ liệu được rút ra từ phân phối thực sự của họ $p(\mathbf{x}, y)$:

$$E_{p(\mathbf{x},y)}[l(f(\mathbf{x}), y)] = \int \int l(f(\mathbf{x}), y) p(\mathbf{x}, y) \, d\mathbf{x} dy. \tag{5.9.2}$$

Tuy nhiên, trong thực tế, chúng ta thường không thể có được toàn bộ dân số dữ liệu. Do đó, *giảm thiểu rủi ro thực nghiệm *, đang giảm thiểu rủi ro thực nghiệm trong (5.9.1), là một chiến lược thực tế cho việc học máy, với hy vọng sẽ giảm thiểu rủi ro gần đúng.

Covariate Shift Correction

Giả sử rằng chúng tôi muốn ước tính một số phụ thuộc $P(y \mid \mathbf{x})$ mà chúng tôi đã dán nhãn dữ liệu $(\mathbf{x}_i, y_i)$. Thật không may, các quan sát $\mathbf{x}_i$ được rút ra từ một số phân phối nguồn *$q(\mathbf{x})$ thay vì phân phối mục tiêu *$p(\mathbf{x})$. May mắn thay, giả định phụ thuộc có nghĩa là phân phối có điều kiện không thay đổi: $p(y \mid \mathbf{x}) = q(y \mid \mathbf{x})$. Nếu phân phối nguồn $q(\mathbf{x})$ là "sai", chúng ta có thể sửa chữa điều đó bằng cách sử dụng danh tính đơn giản sau đây trong rủi ro:

$$\int \int l(f(\mathbf{x}), y) p(y \mid \mathbf{x}) p(\mathbf{x}) \, d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(y \mid \mathbf{x}) q(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} \, d\mathbf{x} dy. \tag{5.9.3}$$

Nói cách khác, chúng ta cần cân nhắc lại từng ví dụ dữ liệu theo tỷ lệ xác suất mà nó sẽ được rút ra từ phân phối chính xác đến đó từ sai:

$$\beta_i \stackrel{\text{def}}{=} \frac{p(\mathbf{x}_i)}{q(\mathbf{x}_i)}. \tag{5.9.4}$$

Cắm trọng lượng β_i cho mỗi ví dụ dữ liệu $(\mathbf{x}_i, y_i)$ chúng ta có thể đào tạo mô hình của mình bằng cách sử dụng *giảm thiểu rủi ro thực nghiệm có trọng lự*:

$$\underset{f}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} \beta_i l(f(\mathbf{x}_i), y_i). \tag{5.9.5}$$

Than ôi, chúng ta không biết tỷ lệ đó, vì vậy trước khi chúng ta có thể làm bất cứ điều gì hữu ích, chúng ta cần ước tính nó. Nhiều phương pháp có sẵn, bao gồm một số cách tiếp cận lý thuyết vận hành ưa thích mà cố gắng tái hiệu chỉnh lại toán tử kỳ vọng trực tiếp bằng cách sử dụng một tiêu chuẩn tối thiểu hoặc một nguyên tắc entropy tối đa. Lưu ý rằng đối với bất kỳ cách tiếp cận nào như vậy, chúng ta cần các mẫu được rút ra từ cả hai bản phân phối - p "đúng", ví dụ, bằng cách truy cập vào dữ liệu thử nghiệm và mẫu được sử dụng để tạo bộ đào tạo q (sau này có sẵn tầm thường). Tuy nhiên, lưu ý rằng chúng tôi chỉ cần các tính năng $\mathbf{x} \sim p(\mathbf{x})$; chúng tôi không cần phải truy cập nhãn $y \sim p(y)$.

Trong trường hợp này, tồn tại một cách tiếp cận rất hiệu quả sẽ cho kết quả gần như tốt như bản gốc: hồi quy hậu cần, đó là một trường hợp đặc biệt của hồi quy softmax (xem Section 4.4) để phân loại nhị phân. Đây là tất cả những gì cần thiết để tính toán tỷ lệ xác suất ước tính. Chúng tôi học một phân loại để phân biệt giữa dữ liệu được rút ra từ $p(\mathbf{x})$ và dữ liệu được rút ra từ $q(\mathbf{x})$. Nếu không thể phân biệt giữa hai bản phân phối thì điều đó có nghĩa là các trường hợp liên quan đều có khả năng đến từ một trong hai bản phân phối. Mặt khác, bất kỳ trường hợp nào có thể bị phân biệt đối xử tốt nên bị quá trọng lượng hoặc giảm cân đáng kể cho phù hợp.

Vì lợi ích của đơn giản giả định rằng chúng ta có một số lượng tương đương của các phiên bản từ cả hai bản phân phối $p(\mathbf{x})$ và $q(\mathbf{x})$, tương ứng. Bây giờ biểu thị bằng z nhãn là 1 cho dữ liệu được rút ra từ p và -1 cho dữ liệu được rút ra từ q. Sau đó, xác suất trong một tập dữ liệu hỗn hợp được đưa ra bởi

$$P(z = 1 \mid \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ and hence } \frac{P(z = 1 \mid \mathbf{x})}{P(z = -1 \mid \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \tag{5.9.6}$$

Do đó, nếu chúng ta sử dụng phương pháp hồi quy logistic, trong đó $P(z = 1 \mid \mathbf{x}) = \frac{1}{1+\exp(-h(\mathbf{x}))}$ (h là một hàm tham số hóa), nó theo sau đó

$$\beta_i = \frac{1/(1 + \exp(-h(\mathbf{x}_i)))}{\exp(-h(\mathbf{x}_i))/(1 + \exp(-h(\mathbf{x}_i)))} = \exp(h(\mathbf{x}_i)). \tag{5.9.7}$$

Kết quả là, chúng ta cần giải quyết hai vấn đề: đầu tiên là phân biệt giữa dữ liệu được rút ra từ cả hai bản phân phối, và sau đó là một vấn đề giảm thiểu rủi ro theo kinh nghiệm có trọng số trong (5.9.5), nơi chúng ta cân nhắc các thuật ngữ β_i.

Bây giờ chúng tôi đã sẵn sàng để mô tả một thuật toán hiệu chỉnh. Giả sử rằng chúng tôi có một bộ đào tạo $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ và một bộ thử nghiệm không có nhãn $\{\mathbf{u}_1, \ldots, \mathbf{u}_m\}$. Đối với sự thay đổi hợp phương, chúng tôi giả định rằng $\mathbf{x}_i$ cho tất cả $1 \leq i \leq n$ được rút ra từ một số phân phối nguồn và $\mathbf{u}_i$ cho tất cả $1 \leq i \leq m$ được rút ra từ phân phối mục tiêu. Dưới đây là một thuật toán nguyên mẫu để sửa đổi dịch chuyển đồng biến:

1. Tạo một bộ đào tạo phân loại nhị phân: $\{(\mathbf{x}_1, -1), \ldots, (\mathbf{x}_n, -1), (\mathbf{u}_1, 1), \ldots, (\mathbf{u}_m, 1)\}$.
2. Đào tạo một phân loại nhị phân sử dụng hồi quy logistic để có được chức năng h.
3. Cân dữ liệu đào tạo sử dụng $\beta_i = \exp(h(\mathbf{x}_i))$ hoặc tốt hơn $\beta_i = \min(\exp(h(\mathbf{x}_i)), c)$ cho một số hằng số c.
4. Sử dụng trọng lượng β_i để đào tạo trên $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ trong (5.9.5).

Lưu ý rằng thuật toán trên dựa vào một giả định quan trọng. Để chương trình này hoạt động, chúng ta cần rằng mỗi ví dụ dữ liệu trong phân phối mục tiêu (ví dụ: thời gian thử nghiệm) có xác suất không xảy ra tại thời điểm đào tạo. Nếu chúng ta tìm thấy một điểm mà $p(\mathbf{x}) > 0$ nhưng $q(\mathbf{x}) = 0$, thì trọng lượng tầm quan trọng tương ứng nên là vô cùng.

Hiệu chỉnh thay đổi nhãn

Giả sử rằng chúng tôi đang đối phó với một nhiệm vụ phân loại với k loại. Sử dụng cùng một ký hiệu trong Section 5.9.3, q và p là phân phối nguồn (ví dụ, thời gian đào tạo) và phân phối mục tiêu (ví dụ, thời gian thử nghiệm), tương ứng. Giả sử rằng việc phân phối nhãn thay đổi theo thời gian: $q(y) \neq p(y)$, nhưng phân phối lớp có điều kiện vẫn giữ nguyên: $q(\mathbf{x} \mid y) = p(\mathbf{x} \mid y)$. Nếu phân phối nguồn $q(y)$ là "sai", chúng ta có thể sửa chữa điều đó theo danh tính sau trong rủi ro như được định nghĩa trong (5.9.2):

$$\int \int l(f(\mathbf{x}), y) p(\mathbf{x} \mid y) p(y) \, d\mathbf{x} dy = \int \int l(f(\mathbf{x}), y) q(\mathbf{x} \mid y) q(y) \frac{p(y)}{q(y)} \, d\mathbf{x} dy. \tag{5.9.8}$$

Ở đây, trọng lượng tầm quan trọng của chúng tôi sẽ tương ứng với tỷ lệ khả năng nhãn

$$\beta_i \stackrel{\text{def}}{=} \frac{p(y_i)}{q(y_i)}. \tag{5.9.9}$$

Một điều hay về sự thay đổi nhãn là nếu chúng ta có một mô hình hợp lý tốt về phân phối nguồn, thì chúng ta có thể nhận được ước tính nhất quán về các trọng lượng này mà không cần phải đối phó với kích thước xung quanh. Trong học sâu, các đầu vào có xu hướng là các đối tượng có chiều cao như hình ảnh, trong khi nhãn thường là các đối tượng đơn giản hơn như các danh mục.

Để ước tính phân phối nhãn mục tiêu, trước tiên chúng tôi lấy bộ phân loại ngoài giá trị hợp lý của chúng tôi (thường được đào tạo về dữ liệu đào tạo) và tính toán ma trận nhầm lẫn của nó bằng cách sử dụng bộ xác thực (cũng từ phân phối đào tạo). Ma trận *nhầm lúc*, $\mathbf{C}$, chỉ đơn giản là ma trận $k \times k$, trong đó mỗi cột tương ứng với danh mục nhãn (sự thật mặt đất) và mỗi hàng tương ứng với danh mục dự đoán của mô hình của chúng tôi. Giá trị của mỗi ô c_{ij} là một phần của tổng dự đoán trên bộ xác thực trong đó nhãn thực là j và mô hình của chúng tôi dự đoán i.

Bây giờ, chúng ta không thể tính toán ma trận nhầm lẫn trên dữ liệu mục tiêu trực tiếp, bởi vì chúng ta không thấy nhãn cho các ví dụ mà chúng ta thấy trong tự nhiên, trừ khi chúng ta đầu tư vào một đường ống chú thích thời gian thực phức tạp. Tuy nhiên, những gì chúng ta có thể làm là trung bình tất cả các dự đoán mô hình của chúng tôi tại thời điểm thử nghiệm cùng nhau, mang lại đầu ra mô hình trung bình $\mu(\hat{\mathbf{y}}) \in \mathbb{R}^k$, có i^{th} yếu tố $\mu(\hat{y}_i)$ là phần nhỏ của tổng dự đoán trên bộ thử nghiệm nơi mô hình của chúng tôi dự đoán i.

Nó chỉ ra rằng trong một số điều kiện nhẹ - nếu bộ phân loại của chúng tôi là chính xác hợp lý ngay từ đầu, và nếu dữ liệu mục tiêu chỉ chứa các danh mục mà chúng ta đã thấy trước đây, và nếu giả định thay đổi nhãn giữ ở nơi đầu tiên (giả định mạnh nhất ở đây), thì chúng ta có thể ước tính nhãn tập kiểm tra phân phối bằng cách giải quyết một hệ thống tuyến tính đơn giản

$$\mathbf{C} p(\mathbf{y}) = \mu(\hat{\mathbf{y}}), \tag{5.9.10}$$

bởi vì như một ước tính $\sum_{j=1}^{k} c_{ij} p(y_j) = \mu(\hat{y}_i)$ giữ cho tất cả $1 \leq i \leq k$, trong đó $p(y_j)$ là yếu tố j^{th} của vector phân phối nhãn k chiều $p(\mathbf{y})$. Nếu phân loại của chúng tôi đủ chính xác để bắt đầu, thì ma trận nhầm lẫn $\mathbf{C}$ sẽ có thể đảo ngược và chúng tôi nhận được một giải pháp $p(\mathbf{y}) = \mathbf{C}^{-1} \mu(\hat{\mathbf{y}})$.

Bởi vì chúng tôi quan sát các nhãn trên dữ liệu nguồn, rất dễ dàng để ước tính phân phối $q(y)$. Sau đó, đối với bất kỳ ví dụ đào tạo nào i có nhãn y_i, chúng ta có thể lấy tỷ lệ ước tính của chúng tôi $p(y_i)/q(y_i)$ để tính trọng lượng β_i và cắm điều này vào giảm thiểu rủi ro thực nghiệm có trọng số trong (5.9.5).

Concept Shift Correction

Sự thay đổi khái niệm khó khăn hơn nhiều để sửa chữa một cách nguyên tắc. Ví dụ, trong tình huống đột nhiên vấn đề thay đổi từ việc phân biệt mèo với chó sang một trong những phân biệt màu trắng với động vật đen, sẽ không hợp lý khi cho rằng chúng ta có thể làm tốt hơn nhiều so với việc chỉ thu thập nhãn mới và huấn luyện từ đầu. May mắn thay, trong thực tế, những thay đổi cực đoan như vậy là rất hiếm. Thay vào đó, những gì thường xảy ra là nhiệm vụ tiếp tục thay đổi chậm. Để làm cho mọi thứ cụ thể hơn, đây là một số ví dụ:

- Trong quảng cáo tính toán, các sản phẩm mới được tung ra, sản phẩm cũ trở nên ít phổ biến hơn. Điều này có nghĩa là việc phân phối trên quảng cáo và mức độ phổ biến của chúng thay đổi dần dần và bất kỳ bộ dự đoán tỷ lệ nhấp qua nào cần phải thay đổi dần dần với nó.
- Ống kính camera giao thông suy giảm dần do hao mòn môi trường, ảnh hưởng đến chất lượng hình ảnh dần dần.
- Nội dung tin tức thay đổi dần dần (tức là hầu hết các tin tức vẫn không thay đổi nhưng những câu chuyện mới xuất hiện).

Trong những trường hợp như vậy, chúng ta có thể sử dụng cách tiếp cận tương tự mà chúng tôi đã sử dụng cho các mạng đào tạo để làm cho chúng thích ứng với sự thay đổi trong dữ liệu. Nói cách khác, chúng tôi sử dụng trọng lượng mạng hiện có và chỉ cần thực hiện một vài bước cập nhật với dữ liệu mới thay vì đào tạo từ đầu.

5.9.4 Một phân loại của các vấn đề học tập

Được trang bị kiến thức về cách đối phó với những thay đổi trong phân phối, giờ đây chúng ta có thể xem xét một số khía cạnh khác của việc xây dựng vấn đề học máy.

Học hàng loạt

Trong * học theo lô*, chúng tôi có quyền truy cập vào các tính năng đào tạo và nhãn $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$, mà chúng tôi sử dụng để đào tạo một mô hình $f(\mathbf{x})$. Sau đó, chúng tôi triển khai mô hình này để ghi dữ liệu mới $(\mathbf{x}, y)$ được rút ra từ cùng một phân phối. Đây là giả định mặc định cho bất kỳ vấn đề nào mà chúng tôi thảo luận ở đây. Ví dụ, chúng ta có thể huấn luyện một máy dò mèo dựa trên rất nhiều hình ảnh của mèo và chó. Khi chúng tôi đào tạo nó, chúng tôi vận chuyển nó như một phần của hệ thống thị giác máy tính catdoor thông minh chỉ cho phép mèo vào. Điều này sau đó được cài đặt trong nhà của khách hàng và không bao giờ được cập nhật một lần nữa (cấm hoàn cảnh cực đoan).

Học trực tuyến

Bây giờ hãy tưởng tượng rằng dữ liệu $(\mathbf{x}_i, y_i)$ đến một mẫu tại một thời điểm. Cụ thể hơn, giả sử rằng lần đầu tiên chúng ta quan sát $\mathbf{x}_i$, sau đó chúng ta cần phải đưa ra một ước tính $f(\mathbf{x}_i)$ và chỉ một khi chúng tôi đã làm điều này, chúng tôi quan sát y_i và với nó, chúng tôi nhận được phần thưởng hoặc phải chịu một sự mất mát, đưa ra quyết định của chúng tôi. Nhiều vấn đề thực sự rơi vào thể loại này. Ví dụ: chúng ta cần dự đoán giá cổ phiếu của ngày mai, điều này cho phép chúng tôi giao dịch dựa trên ước tính đó và vào cuối ngày chúng tôi tìm hiểu xem ước tính của chúng tôi có cho phép chúng tôi kiếm lợi nhuận hay không. Nói cách khác, trong * học trực tuyến*, chúng tôi có chu kỳ sau đây, nơi chúng tôi liên tục cải tiến mô hình của mình cho những quan sát mới.

$$\text{model } f_t \longrightarrow \text{data } \mathbf{x}_t \longrightarrow \text{estimate } f_t(\mathbf{x}_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(\mathbf{x}_t)) \longrightarrow \text{model } f_{t+1} \tag{5.9.11}$$

Kẻ cướp

Bandits là một trường hợp đặc biệt của vấn đề trên. Trong khi trong hầu hết các vấn đề học tập, chúng ta có hàm tham số liên tục f nơi chúng ta muốn tìm hiểu các tham số của nó (ví dụ: mạng sâu), trong một vấn đề *bandit* chúng ta chỉ có một số cánh tay hữu hạn mà chúng ta có thể kéo, tức là, một số hành động hữu hạn mà chúng ta có thể thực hiện. Nó không phải là rất đáng ngạc nhiên rằng đối với vấn đề đơn giản hơn này đảm bảo lý thuyết mạnh mẽ hơn về mặt tối ưu có thể thu được. Chúng tôi liệt kê nó chủ yếu vì vấn đề này thường bị xử lý (gây nhầm lẫn) như thể đó là một bối cảnh học tập riêng biệt.

Kiểm soát

Trong nhiều trường hợp, môi trường nhớ những gì chúng tôi đã làm. Không nhất thiết phải theo cách đối nghịch nhưng nó sẽ chỉ nhớ và phản ứng sẽ phụ thuộc vào những gì đã xảy ra trước đây. Ví dụ, một bộ điều khiển nồi hơi cà phê sẽ quan sát nhiệt độ khác nhau tùy thuộc vào việc nó đã được sưởi ấm nồi hơi trước đây hay không. Thuật toán điều khiển PID (tỷ lệ tích hợp) là một lựa chọn phổ biến ở đó. Tương tự như vậy, hành vi của người dùng trên một trang tin tức sẽ phụ thuộc vào những gì chúng tôi đã cho anh ta thấy trước đây (ví dụ: anh ta sẽ chỉ đọc hầu hết tin tức một lần). Nhiều thuật toán như vậy tạo thành một mô hình của môi trường mà chúng hành động như để đưa ra quyết định của họ có vẻ ít ngẫu nhiên hơn. Gần đây, lý thuyết điều khiển (ví dụ, các biến thể PID) cũng đã được sử dụng để tự động điều chỉnh các siêu tham số để đạt được chất lượng disentangling và tái thiết tốt hơn, và cải thiện sự đa dạng của văn bản được tạo ra và chất lượng tái thiết của hình ảnh được tạo ra (Shao et al., 2020).

Học tăng cường

Trong trường hợp chung hơn của một môi trường có bộ nhớ, chúng ta có thể gặp phải các tình huống mà môi trường đang cố gắng hợp tác với chúng tôi (các trò chơi hợp tác, đặc biệt là cho các trò chơi không tổng bằng không) hoặc những người khác nơi môi trường sẽ cố gắng giành chiến thắng. Cờ vua, Go, Backgammon hoặc StarCraft là một số trường hợp trong * tăng cường học về*. Tương tự như vậy, chúng ta có thể muốn xây dựng một bộ điều khiển tốt cho xe ô tô tự trị. Những chiếc xe khác có khả năng đáp ứng với phong cách lái xe tự hành theo những cách không tầm thường, ví dụ, cố gắng tránh nó, cố gắng gây ra tai nạn và cố gắng hợp tác với nó.

Xem xét môi trường

Một điểm khác biệt chính giữa các tình huống khác nhau ở trên là cùng một chiến lược có thể đã hoạt động trong suốt trường hợp môi trường cố định, có thể không hoạt động trong suốt khi môi trường có thể thích nghi. Ví dụ, một cơ hội chênh lệch giá được phát hiện bởi một nhà giao dịch có khả năng biến mất khi anh ta bắt đầu khai thác nó. Tốc độ và cách thức mà môi trường thay đổi quyết định ở mức độ lớn loại thuật toán mà chúng ta có thể mang lại. Ví dụ, nếu chúng ta biết rằng mọi thứ chỉ có thể thay đổi chậm, chúng ta có thể buộc bất kỳ ước tính nào chỉ thay đổi chậm, quá. Nếu chúng ta biết rằng môi trường có thể thay đổi ngay lập tức, nhưng chỉ rất không thường xuyên, chúng ta có thể làm cho phụ cấp cho điều đó. Những loại kiến thức này rất quan trọng đối với các nhà khoa học dữ liệu tham vọng để đối phó với sự thay đổi khái niệm, tức là, khi vấn đề mà ông đang cố gắng giải quyết những thay đổi theo thời gian.

5.9.5 Công bằng, trách nhiệm giải trình và minh bạch trong học máy

Cuối cùng, điều quan trọng cần nhớ là khi bạn triển khai các hệ thống machine learning, bạn không chỉ đơn thuần là tối ưu hóa một mô hình dự đoán — bạn thường cung cấp một công cụ sẽ được sử dụng để tự động hóa các quyết định (một phần hoặc toàn bộ). Các hệ thống kỹ thuật này có thể ảnh hưởng đến cuộc sống của các cá nhân phải chịu các quyết định kết quả. Bước nhảy vọt từ việc xem xét các dự đoán cho các quyết định đặt ra không chỉ các câu hỏi kỹ thuật mới, mà còn là một loạt các câu hỏi đạo đức phải được xem xét cẩn thận. Nếu chúng ta đang triển khai một hệ thống chẩn đoán y tế, chúng ta cần biết quần thể nào nó có thể hoạt động và nó có thể không. Nhìn ra những rủi ro có thể thấy trước đối với phúc lợi của một nhóm dân số có thể khiến chúng ta phải chăm sóc kém hơn. Hơn nữa, một khi chúng ta chiêm ngưỡng các hệ thống ra quyết định, chúng ta phải lùi lại và xem xét lại cách chúng ta đánh giá công nghệ của mình. Trong số các hậu quả khác của sự thay đổi phạm vi này, chúng tôi sẽ thấy rằng *độ chính xác* hiếm khi là biện pháp phù hợp. Ví dụ, khi dịch dự đoán thành hành động, chúng ta thường sẽ muốn tính đến độ nhạy chi phí tiềm năng của việc sai phạm theo nhiều cách khác nhau. Nếu một cách phân loại sai hình ảnh có thể được coi là một sự buồn ngủ về chủng tộc, trong khi việc phân loại sai cho một danh mục khác sẽ vô hại, thì chúng ta có thể muốn điều chỉnh ngưỡng của mình cho phù hợp, chiếm các giá trị xã hội trong việc thiết kế giao thức ra quyết định. Chúng tôi cũng muốn cẩn thận về cách các hệ thống dự đoán có thể dẫn đến các vòng phản hồi. Ví dụ, xem xét các hệ thống chính sách dự đoán, phân bổ các sĩ quan tuần tra cho các khu vực có tội phạm dự báo cao. Thật dễ dàng để thấy một mô hình đáng lo ngại có thể xuất hiện như thế nào:

1. Các khu phố có nhiều tội phạm hơn nhận được nhiều tuần tra hơn.
2. Do đó, nhiều tội ác được phát hiện ở các khu phố này, nhập dữ liệu đào tạo có sẵn cho các lần lặp lại trong tương lai.
3. Tiếp xúc với những tích cực hơn, mô hình dự đoán nhiều tội phạm hơn ở những khu phố này.
4. Trong lần lặp tiếp theo, mô hình được cập nhật nhắm vào cùng một khu phố thậm chí còn nhiều hơn dẫn đến nhiều tội ác hơn được phát hiện, v.v.

Thông thường, các cơ chế khác nhau mà các dự đoán của một mô hình trở nên kết hợp với dữ liệu đào tạo của nó không được tính đến trong quá trình mô hình hóa. Điều này có thể dẫn đến những gì các nhà nghiên cứu gọi là * vòng lặp phản hồi runaway*. Ngoài ra, chúng tôi muốn cẩn thận về việc liệu chúng tôi có đang giải quyết đúng vấn đề ngay từ đầu hay không. Các thuật toán tiên đoán bây giờ đóng một vai trò lớn hơn trong việc trung gian việc phổ biến thông tin. Có nên tin tức rằng một cuộc gặp gỡ cá nhân được xác định bởi tập hợp các trang Facebook mà họ có *Liked* không? Đây chỉ là một vài trong số nhiều tình huống khó xử đạo đức cấp bách mà bạn có thể gặp phải trong sự nghiệp học máy.

5.9.6 Tóm tắt

- Trong nhiều trường hợp đào tạo và bộ thử nghiệm không đến từ cùng một phân phối. Đây được gọi là dịch chuyển phân phối.
- Rủi ro là kỳ vọng về sự mất mát đối với toàn bộ dân số dữ liệu được rút ra từ phân phối thực sự của họ. Tuy nhiên, toàn bộ dân số này thường không có sẵn. Rủi ro thực nghiệm là một tổn thất trung bình so với dữ liệu đào tạo để gần đúng rủi ro. Trong thực tế, chúng tôi thực hiện giảm thiểu rủi ro thực nghiệm.
- Theo các giả định tương ứng, covariate và thay đổi nhãn có thể được phát hiện và sửa chữa tại thời điểm thử nghiệm. Việc không tính đến sự thiên vị này có thể trở nên có vấn đề tại thời điểm thử nghiệm.
- Trong một số trường hợp, môi trường có thể nhớ các hành động tự động và phản ứng theo những cách đáng ngạc nhiên. Chúng ta phải tính đến khả năng này khi xây dựng các mô hình và tiếp tục giám sát các hệ thống trực tiếp, mở ra khả năng các mô hình và môi trường của chúng tôi sẽ bị vướng vào những cách không lường trước.

5.9.7 Bài tập

1. Điều gì có thể xảy ra khi chúng ta thay đổi hành vi của một công cụ tìm kiếm? Người dùng có thể làm gì? Còn các nhà quảng cáo thì sao?
2. Thực hiện một máy dò thay đổi covariate. Gợi ý: xây dựng một phân loại.
3. Thực hiện một corrector thay đổi covariate.
4. Bên cạnh sự thay đổi phân phối, điều gì khác có thể ảnh hưởng đến rủi ro thực nghiệm gần đúng rủi ro như thế nào?

Discussions[71]

5.10 Dự đoán giá nhà trên Kaggle

Bây giờ chúng tôi đã giới thiệu một số công cụ cơ bản để xây dựng và đào tạo các mạng lưới sâu và điều chỉnh chúng bằng các kỹ thuật bao gồm phân rã trọng lượng và bỏ học, chúng tôi sẵn sàng đưa tất cả kiến thức này vào thực tế bằng cách tham gia vào một cuộc thi Kaggle. Cuộc thi dự đoán giá nhà là một nơi tuyệt vời để bắt đầu. Dữ liệu khá chung chung và không thể hiện cấu trúc kỳ lạ có thể yêu cầu các mô hình chuyên dụng (như âm thanh hoặc video có thể). Tập dữ liệu này, được thu thập bởi Bart de Cock năm 2011 (DeCock, 2011), bao gồm giá nhà ở Ames, IA từ giai đoạn 2006—2010. Nó lớn hơn đáng kể so với Boston housing dataset[72] nổi tiếng của Harrison và Rubinfeld (1978), tự hào với cả ví dụ hơn và nhiều tính năng hơn.

Trong phần này, chúng tôi sẽ hướng dẫn bạn thông qua các chi tiết về tiền xử lý dữ liệu, thiết kế mô hình và lựa chọn siêu tham số. Chúng tôi hy vọng rằng thông qua một cách tiếp cận thực hành, bạn sẽ có được một số trực giác sẽ hướng dẫn bạn trong sự nghiệp của bạn với tư cách là một nhà khoa học dữ liệu.

5.10.1 Tải xuống và bộ dữ liệu bộ nhớ đệm

Trong suốt cuốn sách, chúng tôi sẽ đào tạo và kiểm tra các mô hình trên các bộ dữ liệu được tải xuống khác nhau. Ở đây, chúng tôi triển khai một số chức năng tiện ích để tạo điều kiện tải dữ liệu. Đầu tiên, chúng ta duy trì một từ điển `DATA_HUB` ánh xạ một chuỗi (*tên* của tập dữ liệu) thành một tuple chứa cả URL để định vị tập dữ liệu và khóa SHA-1 xác minh tính toàn vẹn của tệp. Tất cả các bộ dữ liệu như vậy được lưu trữ tại trang web có địa chỉ là `DATA_URL`.

```
import hashlib
import os
import tarfile
import zipfile
import requests

#@save
DATA_HUB = dict()
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

Hàm `download` sau tải xuống một tập dữ liệu, lưu trữ nó trong một thư mục cục bộ (`../data` theo mặc định) và trả về tên của tệp đã tải xuống. Nếu một tệp tương ứng với tập dữ liệu này đã tồn tại trong thư mục bộ nhớ cache và SHA-1 của nó khớp với tệp được lưu trữ trong `DATA_HUB`, mã của chúng tôi sẽ sử dụng tệp được lưu trữ để tránh làm tắc nghẽn internet của bạn với tải xuống dự phòng.

[71] https://discuss.d2l.ai/t/105
[72] https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names

```
def download(name, cache_dir=os.path.join('..', 'data')):  #@save
    """Download a file inserted into DATA_HUB, return the local filename."""
    assert name in DATA_HUB, f"{name} does not exist in {DATA_HUB}."
    url, sha1_hash = DATA_HUB[name]
    os.makedirs(cache_dir, exist_ok=True)
    fname = os.path.join(cache_dir, url.split('/')[-1])
    if os.path.exists(fname):
        sha1 = hashlib.sha1()
        with open(fname, 'rb') as f:
            while True:
                data = f.read(1048576)
                if not data:
                    break
                sha1.update(data)
        if sha1.hexdigest() == sha1_hash:
            return fname  # Hit cache
    print(f'Downloading {fname} from {url}...')
    r = requests.get(url, stream=True, verify=True)
    with open(fname, 'wb') as f:
        f.write(r.content)
    return fname
```

Chúng tôi cũng triển khai hai chức năng tiện ích bổ sung: một là tải xuống và trích xuất tệp zip hoặc tar và một để tải xuống tất cả các bộ dữ liệu được sử dụng trong cuốn sách này từ `DATA_HUB` vào thư mục bộ nhớ cache.

```
def download_extract(name, folder=None):  #@save
    """Download and extract a zip/tar file."""
    fname = download(name)
    base_dir = os.path.dirname(fname)
    data_dir, ext = os.path.splitext(fname)
    if ext == '.zip':
        fp = zipfile.ZipFile(fname, 'r')
    elif ext in ('.tar', '.gz'):
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted.'
    fp.extractall(base_dir)
    return os.path.join(base_dir, folder) if folder else data_dir

def download_all():  #@save
    """Download all files in the DATA_HUB."""
    for name in DATA_HUB:
        download(name)
```

5.10.2 Kaggle

Kaggle[73] là một nền tảng phổ biến tổ chức các cuộc thi machine learning. Mỗi cuộc thi tập trung vào một tập dữ liệu và nhiều người được tài trợ bởi các bên liên quan cung cấp giải thưởng cho các giải pháp chiến thắng. Nền tảng này giúp người dùng tương tác qua các diễn đàn và mã được chia sẻ, thúc đẩy cả hợp tác và cạnh tranh. Trong khi bảng xếp hạng theo đuổi thường xoắn ốc ngoài tầm kiểm soát, với các nhà nghiên cứu tập trung vào các bước xử lý sơ bộ thay vì đặt câu hỏi cơ bản, nhưng cũng có giá trị to lớn trong tính khách quan của một nền tảng tạo điều kiện so sánh định lượng trực tiếp giữa các phương pháp cạnh tranh cũng như mã chia sẻ để mọi người có thể tìm hiểu những gì đã làm và không làm việc. Nếu bạn muốn tham gia vào một cuộc thi Kaggle, trước tiên bạn sẽ cần đăng ký tài khoản (xem Fig. 5.10.1).

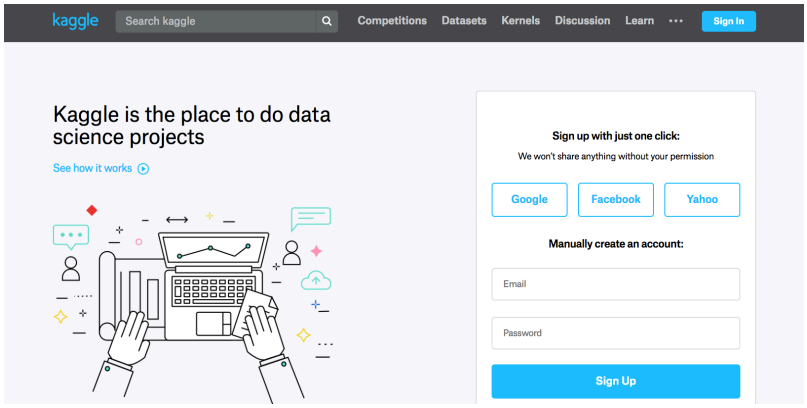


Fig. 5.10.1: The Kaggle website.

Trên trang cạnh tranh dự đoán giá nhà, như minh họa trong Fig. 5.10.2, bạn có thể tìm thấy bộ dữ liệu (trong tab "Dữ liệu"), gửi dự đoán và xem thứ hạng của bạn, URL ở ngay tại đây:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques

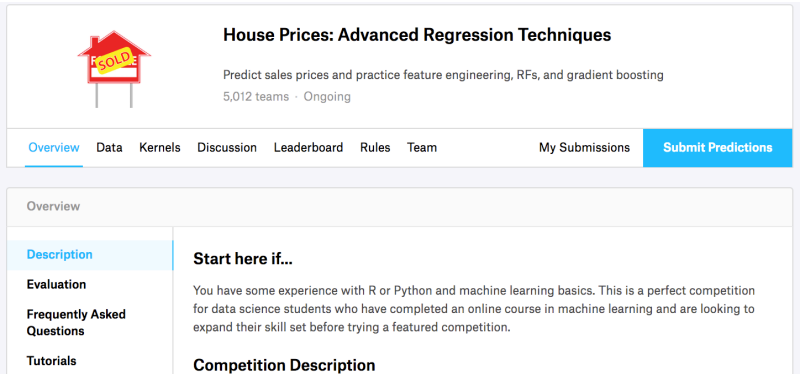


Fig. 5.10.2: The house price prediction competition page.

[73] https://www.kaggle.com

5.10.3 Truy cập và đọc tập dữ liệu

Lưu ý rằng dữ liệu cạnh tranh được tách thành các bộ đào tạo và kiểm tra. Mỗi hồ sơ bao gồm giá trị tài sản của ngôi nhà và các thuộc tính như loại đường phố, năm xây dựng, loại mái nhà, điều kiện tầng hầm, vv Các tính năng bao gồm các loại dữ liệu khác nhau. Ví dụ, năm xây dựng được thể hiện bằng một số nguyên, kiểu mái bằng cách gán phân loại rời rạc và các tính năng khác bằng số điểm nổi. Và đây là nơi thực tế làm phức tạp mọi thứ: đối với một số ví dụ, một số dữ liệu hoàn toàn bị thiếu với giá trị còn thiếu được đánh dấu đơn giản là "na". Giá của mỗi ngôi nhà chỉ được bao gồm cho bộ đào tạo (nó là một cuộc thi sau khi tất cả). Chúng tôi sẽ muốn phân vùng bộ đào tạo để tạo một bộ xác thực, nhưng chúng tôi chỉ nhận được để đánh giá các mô hình của chúng tôi trên thử nghiệm chính thức thiết lập sau khi tải dự đoán lên Kaggle. Tab "Dữ liệu" trên tab cạnh tranh trong Fig. 5.10.2 có các liên kết để tải xuống dữ liệu.

Để bắt đầu, chúng tôi sẽ đọc và xử lý dữ liệu bằng `pandas`, mà chúng tôi đã giới thiệu trong Section 3.2. Vì vậy, bạn sẽ muốn đảm bảo rằng bạn đã cài đặt `pandas` trước khi tiếp tục tiếp tục. May mắn thay, nếu bạn đang đọc trong Jupyter, chúng tôi có thể cài đặt gấu trúc mà không cần rời khỏi máy tính xách tay.

```
# If pandas is not installed, please uncomment the following line:
# !pip install pandas

%matplotlib inline
import numpy as np
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

Để thuận tiện, chúng tôi có thể tải xuống và lưu trữ bộ dữ liệu nhà ở Kaggle bằng cách sử dụng tập lệnh mà chúng tôi đã định nghĩa ở trên.

```
DATA_HUB['kaggle_house_train'] = (  #@save
    DATA_URL + 'kaggle_house_pred_train.csv',
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')

DATA_HUB['kaggle_house_test'] = (  #@save
    DATA_URL + 'kaggle_house_pred_test.csv',
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

Chúng tôi sử dụng `pandas` để tải hai tệp csv chứa dữ liệu đào tạo và kiểm tra tương ứng.

```
train_data = pd.read_csv(download('kaggle_house_train'))
test_data = pd.read_csv(download('kaggle_house_test'))
```

```
Downloading ../data/kaggle_house_pred_train.csv from http://d2l-data.s3-
↪accelerate.amazonaws.com/kaggle_house_pred_train.csv...
Downloading ../data/kaggle_house_pred_test.csv from http://d2l-data.s3-
↪accelerate.amazonaws.com/kaggle_house_pred_test.csv...
```

Tập dữ liệu đào tạo bao gồm 1460 ví dụ, 80 tính năng và 1 nhãn, trong khi dữ liệu thử nghiệm chứa 1459 ví dụ và 80 tính năng.

```
print(train_data.shape)
print(test_data.shape)
```

```
(1460, 81)
(1459, 80)
```

Hãy để chúng tôi hãy xem bốn tính năng đầu tiên và hai tính năng cuối cùng cũng như nhãn (SalePrice) từ bốn ví dụ đầu tiên.

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

Chúng ta có thể thấy rằng trong mỗi ví dụ, tính năng đầu tiên là ID. Điều này giúp mô hình xác định từng ví dụ đào tạo. Mặc dù điều này thuận tiện, nó không mang bất kỳ thông tin nào cho mục đích dự đoán. Do đó, chúng tôi loại bỏ nó khỏi tập dữ liệu trước khi đưa dữ liệu vào mô hình.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

5.10.4 Xử lý sơ bộ dữ liệu

Như đã nêu ở trên, chúng tôi có nhiều loại dữ liệu. Chúng ta sẽ cần xử lý trước dữ liệu trước khi chúng ta có thể bắt đầu mô hình hóa. Hãy để chúng tôi bắt đầu với các tính năng số. Đầu tiên, chúng ta áp dụng một heuristic, thay thế tất cả các giá trị còn thiếu theo ý nghĩa của tính năng tương ứng. Sau đó, để đặt tất cả các tính năng trên thang điểm chung, chúng ta *tiêu chuẩn* dữ liệu bằng cách thay đổi tỷ lệ các tính năng về 0 trung bình và phương sai đơn vị :

$$x \leftarrow \frac{x - \mu}{\sigma}, \tag{5.10.1}$$

trong đó μ và σ biểu thị độ lệch trung bình và chuẩn, tương ứng. Để xác minh rằng điều này thực sự biến đổi tính năng của chúng tôi (biến) sao cho nó có không trung bình và phương sai đơn vị, lưu ý rằng $E[\frac{x-\mu}{\sigma}] = \frac{\mu-\mu}{\sigma} = 0$ và $E[(x-\mu)^2] = (\sigma^2 + \mu^2) - 2\mu^2 + \mu^2 = \sigma^2$ đó. Trực giác, chúng tôi chuẩn hóa dữ liệu vì hai lý do. Đầu tiên, nó chứng minh thuận tiện cho việc tối ưu hóa. Thứ hai, bởi vì chúng tôi không biết *a priori* tính năng nào sẽ có liên quan, chúng tôi không muốn phạt các hệ số được gán cho một tính năng nhiều hơn bất kỳ tính năng nào khác.

```
# If test data were inaccessible, mean and standard deviation could be
# calculated from training data
numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
all_features[numeric_features] = all_features[numeric_features].apply(
    lambda x: (x - x.mean()) / (x.std()))
# After standardizing the data all means vanish, hence we can set missing
# values to 0
all_features[numeric_features] = all_features[numeric_features].fillna(0)
```

Tiếp theo chúng tôi đối phó với các giá trị rời rạc Điều này bao gồm các tính năng như "MSZoning". Chúng tôi thay thế chúng bằng mã hóa một nóng giống như cách mà trước đây chúng tôi đã chuyển đổi nhãn đa lớp thành vectơ (xem Section 4.4.1). Ví dụ, "MSZoning" giả định các giá trị "RL" và "RM". Thả tính năng "MSZoning", hai tính năng chỉ báo mới "MSZoning_RL" và "MSZoning_RM" được tạo ra với các giá trị là 0 hoặc 1. Theo

mã hóa một nóng, nếu giá trị ban đầu của "MSZoning" là "RL", thì "MSZoning_RL" là 1 và "MSZoning_RM" là 0. Gói `pandas` thực hiện điều này tự động cho chúng tôi.

```
# `Dummy_na=True` considers "na" (missing value) as a valid feature value, and
# creates an indicator feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

```
(2919, 331)
```

Bạn có thể thấy rằng chuyển đổi này làm tăng số lượng tính năng từ 79 lên 331. Cuối cùng, thông qua thuộc tính `values`, chúng ta có thể trích xuất định dạng NumPy từ định dạng `pandas` và chuyển đổi nó thành biểu diễn tensor để đào tạo.

```
n_train = train_data.shape[0]
train_features = torch.tensor(all_features[:n_train].values, dtype=torch.
↪float32)
test_features = torch.tensor(all_features[n_train:].values, dtype=torch.
↪float32)
train_labels = torch.tensor(
    train_data.SalePrice.values.reshape(-1, 1), dtype=torch.float32)
```

5.10.5 Đào tạo

Để bắt đầu, chúng tôi đào tạo một mô hình tuyến tính với tổn thất bình phương. Không có gì đáng ngạc nhiên, mô hình tuyến tính của chúng tôi sẽ không dẫn đến một bài nộp chiến thắng trong cuộc thi nhưng nó cung cấp kiểm tra sự tỉnh táo để xem liệu có thông tin có ý nghĩa trong dữ liệu hay không. Nếu chúng ta không thể làm tốt hơn so với đoán ngẫu nhiên ở đây, thì có thể có một cơ hội tốt mà chúng ta có một lỗi xử lý dữ liệu. Và nếu mọi thứ hoạt động, mô hình tuyến tính sẽ đóng vai trò là một đường cơ sở cho chúng ta một số trực giác về việc mô hình đơn giản gần với các mô hình được báo cáo tốt nhất như thế nào, cho chúng ta cảm giác rằng chúng ta nên mong đợi bao nhiêu từ các mô hình fancier.

```
loss = nn.MSELoss()
in_features = train_features.shape[1]

def get_net():
    net = nn.Sequential(nn.Linear(in_features,1))
    return net
```

Với giá nhà, như với giá cổ phiếu, chúng tôi quan tâm đến số lượng tương đối nhiều hơn số lượng tuyệt đối. Do đó chúng tôi có xu hướng quan tâm nhiều hơn về lỗi tương đối $\frac{y-\hat{y}}{y}$ so với lỗi tuyệt đối $y - \hat{y}$. Ví dụ, nếu dự đoán của chúng tôi giảm 100.000 USD khi ước tính giá của một ngôi nhà ở nông thôn Ohio, nơi giá trị của một ngôi nhà điển hình là 125.000 USD, thì có lẽ chúng ta đang làm một công việc khủng khiếp. Mặt khác, nếu chúng ta sai lệch bởi số tiền này ở Los Altos Hills, California, điều này có thể đại diện cho một dự đoán chính xác đáng kinh ngạc (ở đó, giá nhà trung bình vượt quá 4 triệu USD).

Một cách để giải quyết vấn đề này là đo lường sự khác biệt trong logarit của ước tính giá. Trên thực tế, đây cũng là biện pháp lỗi chính thức được sử dụng bởi đối thủ để đánh giá chất lượng đệ trình. Rốt cuộc, một giá trị nhỏ δ cho $|\log y - \log \hat{y}| \leq \delta$ dịch thành $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. Điều này dẫn đến lỗi gốc có nghĩa là bình phương

sau đây giữa logarit của giá dự đoán và logarit của giá nhãn:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log y_i - \log \hat{y}_i)^2}. \tag{5.10.2}$$

```
def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the
    # value less than 1 as 1
    clipped_preds = torch.clamp(net(features), 1, float('inf'))
    rmse = torch.sqrt(loss(torch.log(clipped_preds),
                           torch.log(labels)))
    return rmse.item()
```

Không giống như trong các phần trước, chức năng đào tạo của chúng tôi sẽ dựa vào trình tối ưu hóa Adam (chúng tôi sẽ mô tả chi tiết hơn sau) . Sự hấp dẫn chính của trình tối ưu hóa này là, mặc dù không làm tốt hơn (và đôi khi tệ hơn) được cung cấp tài nguyên không giới hạn để tối ưu hóa siêu tham số, mọi người có xu hướng thấy rằng nó ít nhạy cảm hơn đáng kể với tốc độ học tập ban đầu.

```
def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    train_ls, test_ls = [], []
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    # The Adam optimization algorithm is used here
    optimizer = torch.optim.Adam(net.parameters(),
                                 lr = learning_rate,
                                 weight_decay = weight_decay)
    for epoch in range(num_epochs):
        for X, y in train_iter:
            optimizer.zero_grad()
            l = loss(net(X), y)
            l.backward()
            optimizer.step()
        train_ls.append(log_rmse(net, train_features, train_labels))
        if test_labels is not None:
            test_ls.append(log_rmse(net, test_features, test_labels))
    return train_ls, test_ls
```

5.10.6 K-Xác định chéo Fold

Bạn có thể nhớ lại rằng chúng tôi đã giới thiệu K lần xác thực chéo trong phần mà chúng tôi đã thảo luận về cách đối phó với lựa chọn mô hình (Section 5.4). Chúng tôi sẽ sử dụng điều này để sử dụng tốt để chọn thiết kế mô hình và điều chỉnh các siêu tham số. Trước tiên chúng ta cần một hàm trả về gấp i^{th} của dữ liệu trong quy trình xác thực chéo K lần. Nó tiến hành bằng cách cắt phân đoạn i^{th} dưới dạng dữ liệu xác thực và trả lại phần còn lại dưới dạng dữ liệu đào tạo. Lưu ý rằng đây không phải là cách xử lý dữ liệu hiệu quả nhất và chúng tôi chắc chắn sẽ làm điều gì đó thông minh hơn nhiều nếu tập dữ liệu của chúng tôi lớn hơn đáng kể. Nhưng sự phức tạp bổ sung này có thể làm xáo trộn mã của chúng tôi một cách không cần thiết để chúng tôi có thể bỏ qua nó một cách an toàn ở đây do sự đơn giản của vấn đề của chúng tôi.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
```

(continues on next page)

(continued from previous page)

```
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = torch.cat([X_train, X_part], 0)
            y_train = torch.cat([y_train, y_part], 0)
    return X_train, y_train, X_valid, y_valid
```

Trung bình lỗi đào tạo và xác minh được trả lại khi chúng tôi đào tạo K lần trong xác nhận chéo K lần.

```
def k_fold(k, X_train, y_train, num_epochs, learning_rate, weight_decay,
           batch_size):
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        data = get_k_fold_data(k, i, X_train, y_train)
        net = get_net()
        train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                   weight_decay, batch_size)
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
        if i == 0:
            d2l.plot(list(range(1, num_epochs + 1)), [train_ls, valid_ls],
                     xlabel='epoch', ylabel='rmse', xlim=[1, num_epochs],
                     legend=['train', 'valid'], yscale='log')
        print(f'fold {i + 1}, train log rmse {float(train_ls[-1]):f}, '
              f'valid log rmse {float(valid_ls[-1]):f}')
    return train_l_sum / k, valid_l_sum / k
```

5.10.7 Model Selection

Trong ví dụ này, chúng tôi chọn một bộ siêu tham số không điều chỉnh và để nó lên cho người đọc để cải thiện mô hình. Tìm một lựa chọn tốt có thể mất thời gian, tùy thuộc vào số lượng biến một tối ưu hóa hơn. Với một tập dữ liệu đủ lớn và các loại siêu tham số bình thường, xác nhận chéo K có xu hướng có khả năng phục hồi hợp lý chống lại nhiều thử nghiệm. Tuy nhiên, nếu chúng tôi thử một số lượng lớn các tùy chọn bất hợp lý, chúng tôi có thể gặp may mắn và thấy rằng hiệu suất xác thực của chúng tôi không còn đại diện cho lỗi thực sự.

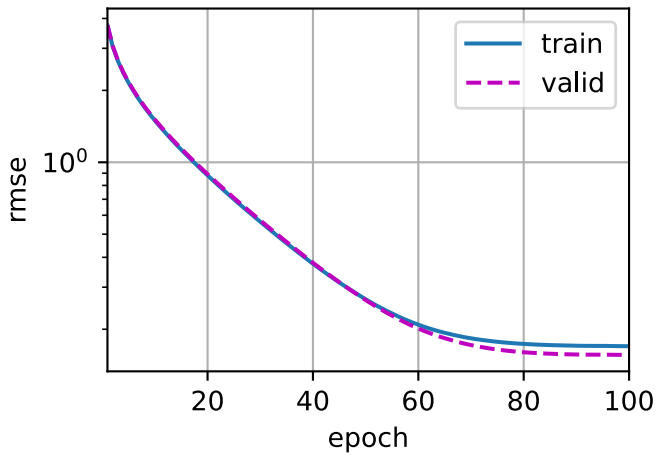
```
k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                          weight_decay, batch_size)
print(f'{k}-fold validation: avg train log rmse: {float(train_l):f}, '
      f'avg valid log rmse: {float(valid_l):f}')
```

```
fold 1, train log rmse 0.169845, valid log rmse 0.156327
fold 2, train log rmse 0.162059, valid log rmse 0.189429
fold 3, train log rmse 0.164017, valid log rmse 0.168371
```

(continues on next page)

(continued from previous page)

```
fold 4, train log rmse 0.168083, valid log rmse 0.154609
fold 5, train log rmse 0.163392, valid log rmse 0.182799
5-fold validation: avg train log rmse: 0.165479, avg valid log rmse: 0.170307
```



Lưu ý rằng đôi khi số lượng lỗi đào tạo cho một tập hợp các siêu tham số có thể rất thấp, ngay cả khi số lỗi trên K lần xác nhận chéo cao hơn đáng kể. Điều này chỉ ra rằng chúng tôi đang overfitting. Trong suốt quá trình đào tạo, bạn sẽ muốn theo dõi cả hai số. Ít quá mức có thể chỉ ra rằng dữ liệu của chúng tôi có thể hỗ trợ một mô hình mạnh mẽ hơn. Đồ sộ có thể gợi ý rằng chúng ta có thể đạt được bằng cách kết hợp các kỹ thuật chính quy hóa.

5.10.8 Nộp dự đoán trên Kaggle

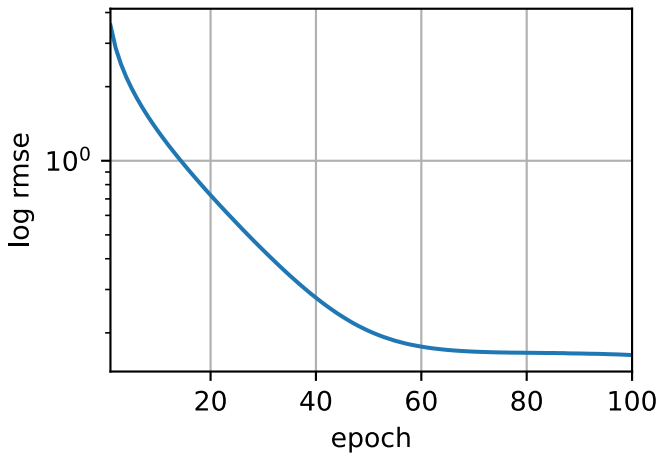
Bây giờ chúng ta biết một lựa chọn tốt của các siêu tham số nên là gì, chúng ta cũng có thể sử dụng tất cả dữ liệu để đào tạo trên nó (thay vì chỉ $1 - 1/K$ dữ liệu được sử dụng trong các lát xác nhận chéo). Mô hình mà chúng ta có được theo cách này sau đó có thể được áp dụng cho bộ thử nghiệm. Lưu dự đoán trong tệp csv sẽ đơn giản hóa việc tải kết quả lên Kaggle.

```
def train_and_pred(train_features, test_features, train_labels, test_data,
                   num_epochs, lr, weight_decay, batch_size):
    net = get_net()
    train_ls, _ = train(net, train_features, train_labels, None, None,
                        num_epochs, lr, weight_decay, batch_size)
    d2l.plot(np.arange(1, num_epochs + 1), [train_ls], xlabel='epoch',
             ylabel='log rmse', xlim=[1, num_epochs], yscale='log')
    print(f'train log rmse {float(train_ls[-1]):f}')
    # Apply the network to the test set
    preds = net(test_features).detach().numpy()
    # Reformat it to export to Kaggle
    test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
    submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
    submission.to_csv('submission.csv', index=False)
```

Một kiểm tra sự tỉnh táo tốt đẹp là xem liệu các dự đoán trên bộ thử nghiệm có giống với quy trình xác thực chéo K lần hay không. Nếu họ làm vậy, đã đến lúc tải chúng lên Kaggle. Đoạn mã sau sẽ tạo ra một tập tin gọi là `submission.csv`.

```
train_and_pred(train_features, test_features, train_labels, test_data,
               num_epochs, lr, weight_decay, batch_size)
```

```
train log rmse 0.162519
```



Tiếp theo, như đã được chứng minh trong Fig. 5.10.3, chúng tôi có thể gửi dự đoán của mình về Kaggle và xem cách chúng so sánh với giá nhà thực tế (nhãn) trên bộ thử nghiệm. Các bước khá đơn giản:

- Đăng nhập vào trang web Kaggle và truy cập trang cạnh tranh dự đoán giá nhà.
- Nhấp vào nút "Gửi dự đoán" hoặc "Nộp muộn" (như văn bản này, nút nằm ở bên phải).
- Nhấp vào nút "Tải lên tệp gửi" trong hộp đứt nét ở cuối trang và chọn tệp dự đoán bạn muốn tải lên.
- Nhấp vào nút "Make Submission" ở cuối trang để xem kết quả của bạn.

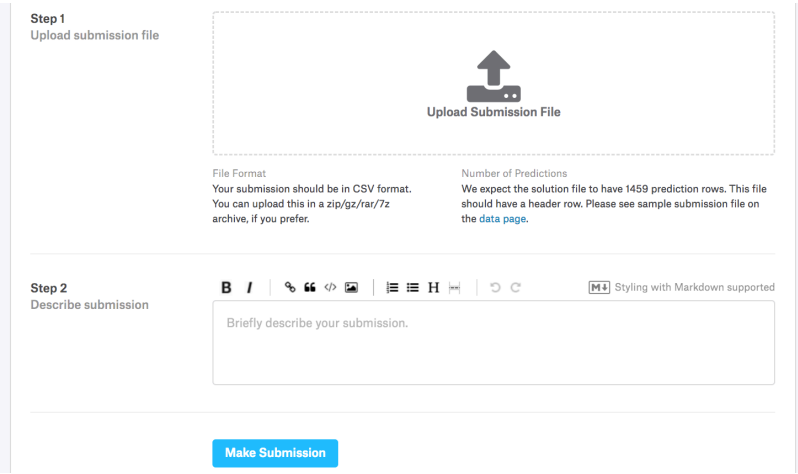


Fig. 5.10.3: Submitting data to Kaggle

5.10.9 Tóm tắt

- Dữ liệu thực thường chứa hỗn hợp các loại dữ liệu khác nhau và cần được xử lý trước.
- Rescaling dữ liệu có giá trị thực thành 0 trung bình và phương sai đơn vị là một mặc định tốt. Vì vậy, đang thay thế các giá trị bị thiếu với trung bình của chúng.
- Chuyển đổi các tính năng phân loại thành các tính năng chỉ báo cho phép chúng ta đối xử với chúng như các vectơ một nóng.
- Chúng ta có thể sử dụng xác nhận chéo K lần để chọn mô hình và điều chỉnh các siêu tham số.
- Logarit rất hữu ích cho các lỗi tương đối.

5.10.10 Bài tập

1. Gửi dự đoán của bạn cho phần này cho Kaggle. Dự đoán của bạn tốt như thế nào?
2. Bạn có thể cải thiện mô hình của mình bằng cách giảm thiểu lôgarit giá trực tiếp không? Điều gì sẽ xảy ra nếu bạn cố gắng dự đoán logarit của giá chứ không phải là giá?
3. Có phải luôn luôn là một ý tưởng tốt để thay thế các giá trị bị thiếu bằng ý nghĩa của họ? Gợi ý: bạn có thể xây dựng một tình huống mà các giá trị không bị thiếu ngẫu nhiên?
4. Cải thiện điểm số trên Kaggle bằng cách điều chỉnh các siêu tham số thông qua xác nhận chéo K lần.
5. Cải thiện điểm số bằng cách cải thiện mô hình (ví dụ: lớp, phân rã trọng lượng và bỏ học).
6. Điều gì sẽ xảy ra nếu chúng ta không chuẩn hóa các tính năng số liên tục như những gì chúng ta đã làm trong phần này?

Discussions[74]

[74] https://discuss.d2l.ai/t/107

6 | Tính toán học sâu

Bên cạnh các bộ dữ liệu khổng lồ và phần cứng mạnh mẽ, các công cụ phần mềm tuyệt vời đã đóng một vai trò không thể thiếu trong tiến bộ nhanh chóng của học sâu. Bắt đầu với thư viện Theano phá vỡ được phát hành vào năm 2007, các công cụ mã nguồn mở linh hoạt đã cho phép các nhà nghiên cứu nhanh chóng các mô hình nguyên mẫu, tránh làm việc lặp đi lặp lại khi tái chế các thành phần tiêu chuẩn trong khi vẫn duy trì khả năng thực hiện các sửa đổi cấp thấp. Theo thời gian, các thư viện của deep learning đã phát triển để cung cấp các trừu tượng ngày càng thô. Cũng giống như các nhà thiết kế bán dẫn đã đi từ chỉ định bóng bán dẫn đến các mạch logic sang viết mã, các nhà nghiên cứu mạng thần kinh đã chuyển từ suy nghĩ về hành vi của các tế bào thần kinh nhân tạo riêng lẻ sang hình thành mạng về toàn bộ lớp, và bây giờ thường thiết kế kiến trúc với thô hơn xa * khối* trong tâm trí.

Cho đến nay, chúng tôi đã giới thiệu một số khái niệm máy học cơ bản, tăng cường các mô hình học sâu đầy đủ chức năng. Trong chương cuối, chúng tôi đã triển khai từng thành phần của MLP từ đầu và thậm chí cho thấy cách tận dụng các API cấp cao để triển khai các mô hình tương tự một cách dễ dàng. Để giúp bạn có được đến mức nhanh như vậy, chúng tôi * gọi lên* các thư viện, nhưng bỏ qua các chi tiết nâng cao hơn về * cách chúng hoạt động. *Trong chương này, chúng ta sẽ bóc lại bức màn, đào sâu hơn vào các thành phần chính của tính toán học sâu, cụ thể là xây dựng mô hình, truy cập và khởi tạo tham số, thiết kế các lớp và khối tùy chỉnh, đọc và ghi các mô hình vào đĩa và tận dụng GPU để đạt được tốc độ ấn tượng. Những thông tin chi tiết này sẽ chuyển bạn từ * người dùng cùng* sang *power user*, cung cấp cho bạn các công cụ cần thiết để gặt hái những lợi ích của thư viện deep learning trưởng thành trong khi vẫn giữ được sự linh hoạt để triển khai các mô hình phức tạp hơn, bao gồm cả những người bạn tự phát minh ra! Mặc dù chương này không giới thiệu bất kỳ mô hình hoặc bộ dữ liệu mới nào, nhưng các chương mô hình nâng cao theo sau phụ thuộc rất nhiều vào các kỹ thuật này.

6.1 Lớp và khối

Khi chúng tôi lần đầu tiên giới thiệu các mạng thần kinh, chúng tôi tập trung vào các mô hình tuyến tính với một đầu ra duy nhất. Ở đây, toàn bộ mô hình chỉ bao gồm một tế bào thần kinh duy nhất. Lưu ý rằng một tế bào thần kinh đơn (i) lấy một số tập hợp các đầu vào; (ii) tạo ra một đầu ra vô hướng tương ứng; và (iii) có một tập hợp các tham số liên quan có thể được cập nhật để tối ưu hóa một số chức năng quan tâm. Sau đó, một khi chúng tôi bắt đầu suy nghĩ về các mạng có nhiều đầu ra, chúng tôi đã tận dụng số học vector hóa để mô tả toàn bộ một lớp tế bào thần kinh. Cũng giống như các tế bào thần kinh riêng lẻ, các lớp (i) lấy một tập hợp các đầu vào, (ii) tạo ra các đầu ra tương ứng, và (iii) được mô tả bởi một tập hợp các tham số có thể điều chỉnh. Khi chúng tôi làm việc thông qua hồi quy softmax, một lớp duy nhất chính nó là mô hình. Tuy nhiên, ngay cả khi chúng tôi sau đó giới thiệu MLP, chúng ta vẫn có thể nghĩ về mô hình là giữ lại cấu trúc cơ bản tương tự này.

Điều thú vị là đối với MLP, cả toàn bộ mô hình và các lớp cấu thành của nó đều chia sẻ cấu trúc này. Toàn bộ mô hình lấy đầu vào thô (các tính năng), tạo ra các đầu ra (dự đoán) và sở hữu các tham số (các tham số kết hợp từ tất cả các lớp cấu thành). Tương tự như vậy, mỗi lớp riêng lẻ ăn vào đầu vào (được cung cấp bởi lớp

trước đó) tạo ra các đầu ra (đầu vào cho lớp tiếp theo) và sở hữu một tập hợp các tham số có thể điều chỉnh được cập nhật theo tín hiệu chảy ngược từ lớp tiếp theo.

Mặc dù bạn có thể nghĩ rằng tế bào thần kinh, lớp và mô hình cung cấp cho chúng ta đủ trừu tượng để đi về kinh doanh của chúng tôi, hóa ra chúng ta thường thấy thuận tiện khi nói về các thành phần lớn hơn một lớp riêng lẻ nhưng nhỏ hơn toàn bộ mô hình. Ví dụ, kiến trúc ResNet-152, rất phổ biến trong tầm nhìn máy tính, sở hữu hàng trăm lớp. Các lớp này bao gồm các mô hình lặp lại của * nhóm lớp*. Thực hiện một mạng như vậy một lớp tại một thời điểm có thể phát triển tẻ nhạt. Mối quan tâm này không chỉ là giả thuyết, các mẫu thiết kế như vậy là phổ biến trong thực tế. Kiến trúc ResNet được đề cập ở trên đã giành được các cuộc thi tầm nhìn máy tính ImageNet và COCO 2015 cho cả công nhận và phát hiện (He et al., 2016a) và vẫn là một kiến trúc đi đến cho nhiều nhiệm vụ thị giác. Các kiến trúc tương tự trong đó các lớp được sắp xếp theo nhiều mẫu lặp lại khác nhau hiện đang phổ biến trong các lĩnh vực khác, bao gồm xử lý ngôn ngữ tự nhiên và lời nói.

Để thực hiện các mạng phức tạp này, chúng tôi giới thiệu khái niệm về một mạng nơ-ron* block*. Một khối có thể mô tả một lớp duy nhất, một thành phần bao gồm nhiều lớp, hoặc toàn bộ mô hình chính nó! Một lợi ích của việc làm việc với trừu tượng khối là chúng có thể được kết hợp thành các hiện vật lớn hơn, thường đệ quy. Điều này được minh họa trong Fig. 6.1.1. Bằng cách xác định mã để tạo ra các khối phức tạp tùy ý theo yêu cầu, chúng ta có thể viết mã nhỏ gọn đáng ngạc nhiên và vẫn triển khai các mạng nơ-ron phức tạp.

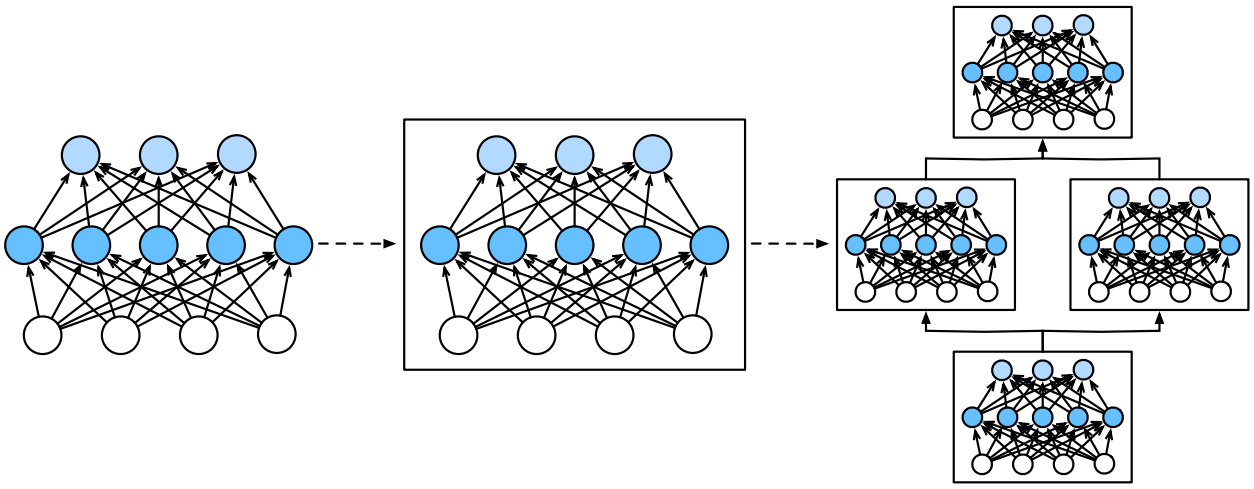


Fig. 6.1.1: Multiple layers are combined into blocks, forming repeating patterns of larger models.

Từ quan điểm lập trình, một khối được biểu diễn bằng một lớp * *. Bất kỳ lớp con nào của nó phải xác định một hàm tuyên truyền chuyển tiếp biến đổi đầu vào của nó thành đầu ra và phải lưu trữ bất kỳ tham số cần thiết nào. Lưu ý rằng một số khối không yêu cầu bất kỳ tham số nào cả. Cuối cùng một khối phải sở hữu một hàm backpropagation, cho mục đích tính toán gradient. May mắn thay, do một số phép thuật hậu trường được cung cấp bởi sự khác biệt tự động (được giới thiệu trong Section 3.5) khi xác định khối của riêng mình, chúng ta chỉ cần lo lắng về các thông số và chức năng lan truyền chuyển tiếp.

Để bắt đầu, chúng tôi xem lại mã mà chúng tôi đã sử dụng để thực hiện MLPs. Mã sau tạo ra một mạng với một lớp ẩn được kết nối hoàn toàn với 256 đơn vị và kích hoạt ReLU, tiếp theo là lớp đầu ra được kết nối hoàn toàn với 10 đơn vị (không có chức năng kích hoạt).

```
import torch
from torch import nn
from torch.nn import functional as F

net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
```

(continues on next page)

(continued from previous page)

```
X = torch.rand(2, 20)
net(X)
```

```
tensor([[ 0.2154, -0.2353, -0.0835,  0.0533,  0.0887,  0.0856, -0.1998, -0.
↪0017,
          0.3100, -0.1677],
        [ 0.2044, -0.2373, -0.1043,  0.0157, -0.0221, -0.0964, -0.2414, -0.
↪0525,
          0.3308, -0.0361]], grad_fn=<AddmmBackward0>)
```

Trong ví dụ này, chúng tôi xây dựng mô hình của mình bằng cách khởi tạo một `nn.Sequential`, với các lớp theo thứ tự mà chúng nên được thực hiện thông qua dưới dạng đối số. Nói tóm lại, `nn.Sequential` định nghĩa một loại đặc biệt của `Module`, lớp trình bày một khối trong PyTorch. Nó duy trì một danh sách đặt hàng của `Module`s cấu thành. Lưu ý rằng mỗi lớp trong hai lớp được kết nối hoàn toàn là một thể hiện của lớp `Linear` mà chính nó là một lớp con của `Module`. Chức năng lan truyền chuyển tiếp (`forward`) cũng rất đơn giản: nó chuỗi mỗi khối trong danh sách với nhau, truyền đầu ra của mỗi như đầu vào cho tiếp theo. Lưu ý rằng cho đến bây giờ, chúng tôi đã gọi các mô hình của chúng tôi thông qua xây dựng `net(X)` để có được kết quả đầu ra của chúng. Điều này thực sự chỉ là viết tắt cho `net.__call__(X)`.

6.1.1 Một khối tùy chỉnh

Có lẽ cách dễ nhất để phát triển trực giác về cách thức hoạt động của một khối là tự thực hiện một. Trước khi chúng tôi triển khai khối tùy chỉnh của riêng mình, chúng tôi tóm tắt ngắn gọn các chức năng cơ bản mà mỗi khối phải cung cấp:

1. Nhập dữ liệu đầu vào dưới dạng đối số cho chức năng tuyên truyền chuyển tiếp của nó.
2. Tạo ra một đầu ra bằng cách có hàm tuyên truyền chuyển tiếp trả về một giá trị. Lưu ý rằng đầu ra có thể có một hình dạng khác với đầu vào. Ví dụ: lớp kết nối hoàn toàn đầu tiên trong mô hình của chúng ta ở trên sẽ nhập một đầu vào của kích thước 20 nhưng trả về đầu ra của kích thước 256.
3. Tính gradient của đầu ra của nó đối với đầu vào của nó, có thể được truy cập thông qua chức năng truyền ngược của nó. Thông thường điều này xảy ra tự động.
4. Lưu trữ và cung cấp quyền truy cập vào các tham số cần thiết để thực hiện tính toán tuyên truyền chuyển tiếp.
5. Khởi tạo các tham số mô hình khi cần thiết.

Trong đoạn mã sau, chúng tôi mã hóa một khối từ đầu tương ứng với một MLP với một lớp ẩn với 256 đơn vị ẩn, và một lớp đầu ra 10 chiều. Lưu ý rằng lớp `MLP` bên dưới kế thừa lớp đại diện cho một khối. Chúng ta sẽ dựa rất nhiều vào các hàm của lớp cha, chỉ cung cấp hàm tạo riêng của chúng ta (hàm `__init__` trong Python) và hàm tuyên truyền chuyển tiếp.

```
class MLP(nn.Module):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers
    def __init__(self):
        # Call the constructor of the `MLP` parent class `Module` to perform
        # the necessary initialization. In this way, other function arguments
        # can also be specified during class instantiation, such as the model
        # parameters, `params` (to be described later)
```

(continues on next page)

```
        super().__init__()
        self.hidden = nn.Linear(20, 256)  # Hidden layer
        self.out = nn.Linear(256, 10)  # Output layer

    # Define the forward propagation of the model, that is, how to return the
    # required model output based on the input `X`
    def forward(self, X):
        # Note here we use the funtional version of ReLU defined in the
        # nn.functional module.
        return self.out(F.relu(self.hidden(X)))
```

Trước tiên chúng ta hãy tập trung vào chức năng lan truyền về phía trước. Lưu ý rằng phải mất `X` làm đầu vào, tính toán biểu diễn ẩn với chức năng kích hoạt được áp dụng và đầu ra các bản ghi của nó. Trong triển khai `MLP` này, cả hai lớp đều là các biến thể. Để xem lý do tại sao điều này là hợp lý, hãy tưởng tượng khởi tạo hai MLP s, `net1` và `net2`, và đào tạo chúng trên các dữ liệu khác nhau. Đương nhiên, chúng tôi mong đợi họ đại diện cho hai mô hình đã học khác nhau.

Chúng ta khởi tạo các lớp MLP trong hàm tạo và sau đó gọi các lớp này trên mỗi lần gọi đến hàm tuyên truyền chuyển tiếp. Lưu ý một vài chi tiết chính. Đầu tiên, chức năng `__init__` tùy chỉnh của chúng tôi gọi chức năng `__init__` của lớp mẹ thông qua `super().__init__()` tiết kiệm cho chúng tôi nỗi đau của việc đặt lại mã boilerplate áp dụng cho hầu hết các khối. Sau đó, chúng tôi khởi tạo hai lớp được kết nối hoàn toàn của chúng tôi, gán chúng cho `self.hidden` và `self.out`. Lưu ý rằng trừ khi chúng ta thực hiện một toán tử mới, chúng ta không cần phải lo lắng về hàm backpropagation hoặc khởi tạo tham số. Hệ thống sẽ tự động tạo ra các chức năng này. Hãy để chúng tôi thử điều này ra.

```
net = MLP()
net(X)
```

```
tensor([[-0.0966,  0.1394, -0.1274,  0.3354,  0.1572, -0.0922, -0.0228, -0.
↪2199,
         -0.1109, -0.2341],
        [-0.0501,  0.1194, -0.0566,  0.2452,  0.2531, -0.1992,  0.0007, -0.
↪2337,
         -0.1568, -0.2085]], grad_fn=<AddmmBackward0>)
```

Một đức tính chính của trừu tượng khối là tính linh hoạt của nó. Chúng ta có thể phân lớp một khối để tạo ra các lớp (chẳng hạn như lớp lớp lớp được kết nối hoàn toàn), toàn bộ mô hình (chẳng hạn như lớp `MLP` ở trên) hoặc các thành phần khác nhau có độ phức tạp trung gian. Chúng tôi khai thác tính linh hoạt này trong suốt các chương sau, chẳng hạn như khi giải quyết các mạng thần kinh phức tạp.

6.1.2 The Sequential Block

Bây giờ chúng ta có thể xem xét kỹ hơn cách thức hoạt động của lớp `Sequential`. Nhớ lại rằng `Sequential` được thiết kế để chuỗi các khối khác với nhau. Để xây dựng `MySequential` đơn giản hóa của riêng mình, chúng ta chỉ cần định nghĩa hai hàm chính: 1. Một hàm để nối các khối từng cái một vào một danh sách. 2. Một chức năng tuyên truyền chuyển tiếp để truyền một đầu vào thông qua chuỗi các khối, theo thứ tự như chúng được nối thêm.

Lớp `MySequential` sau đây cung cấp chức năng tương tự của lớp `Sequential` mặc định.

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            # Here, `module` is an instance of a `Module` subclass. We save it
            # in the member variable `_modules` of the `Module` class, and its
            # type is OrderedDict
            self._modules[str(idx)] = module

    def forward(self, X):
        # OrderedDict guarantees that members will be traversed in the order
        # they were added
        for block in self._modules.values():
            X = block(X)
        return X
```

Trong phương pháp `__init__`, chúng tôi thêm từng mô-đun vào từ điển được đặt hàng `_modules` từng cái một. Bạn có thể tự hỏi tại sao mỗi `Module` đều sở hữu một thuộc tính `_modules` và tại sao chúng ta sử dụng nó thay vì chỉ tự xác định một danh sách Python. Tóm lại, lợi thế chính của `_modules` là trong quá trình khởi tạo tham số mô-đun của chúng tôi, hệ thống biết nhìn vào bên trong từ điển `_modules` để tìm các mô-đun phụ có tham số cũng cần được khởi tạo.

Khi hàm tuyên truyền chuyển tiếp `MySequential` của chúng tôi được gọi, mỗi khối được thêm vào được thực hiện theo thứ tự mà chúng được thêm vào. Bây giờ chúng ta có thể triển khai lại MLP bằng cách sử dụng lớp `MySequential` của chúng tôi.

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X)
```

```
tensor([[-0.3238, -0.0582,  0.3331,  0.0385, -0.0048, -0.5133,  0.1399,  0.
↪1480,
         -0.1763,  0.0246],
        [-0.3501, -0.0605,  0.1158, -0.0148, -0.0063, -0.4010,  0.0360,  0.
↪0430,
         -0.1625,  0.0111]], grad_fn=<AddmmBackward0>)
```

Lưu ý rằng việc sử dụng `MySequential` này giống với mã mà chúng tôi đã viết trước đây cho lớp `Sequential` (như được mô tả trong Section 5.3).

6.1.3 Thihành mã trong chức năng tuyên truyền chuyển tiếp

Lớp `Sequential` giúp việc xây dựng mô hình dễ dàng, cho phép chúng tôi lắp ráp các kiến trúc mới mà không cần phải xác định lớp học của riêng mình. Tuy nhiên, không phải tất cả các kiến trúc đều là chuỗi daisy đơn giản. Khi cần tính linh hoạt cao hơn, chúng tôi sẽ muốn xác định các khối của riêng mình. Ví dụ, chúng ta có thể muốn thực thi luồng điều khiển của Python trong hàm tuyên truyền chuyển tiếp. Hơn nữa, chúng ta có thể muốn thực hiện các phép toán tùy ý, không chỉ dựa vào các lớp mạng thần kinh được xác định trước.

Bạn có thể nhận thấy rằng cho đến bây giờ, tất cả các hoạt động trong mạng của chúng tôi đã hoạt động dựa trên các kích hoạt mạng và các thông số của mạng của chúng tôi. Tuy nhiên, đôi khi, chúng ta có thể muốn kết hợp các thuật ngữ không phải là kết quả của các lớp trước đó cũng như các tham số có thể cập nhật. Chúng tôi gọi các tham số * không đổi này*. Ví dụ, chúng ta muốn một lớp tính hàm $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$, trong đó

x là đầu vào, **w** là tham số của chúng ta và c là một số hằng số được chỉ định không được cập nhật trong quá trình tối ưu hóa. Vì vậy, chúng tôi thực hiện một lớp `FixedHiddenMLP` như sau.

```
class FixedHiddenMLP(nn.Module):
    def __init__(self):
        super().__init__()
        # Random weight parameters that will not compute gradients and
        # therefore keep constant during training
        self.rand_weight = torch.rand((20, 20), requires_grad=False)
        self.linear = nn.Linear(20, 20)

    def forward(self, X):
        X = self.linear(X)
        # Use the created constant parameters, as well as the `relu` and `mm`
        # functions
        X = F.relu(torch.mm(X, self.rand_weight) + 1)
        # Reuse the fully-connected layer. This is equivalent to sharing
        # parameters with two fully-connected layers
        X = self.linear(X)
        # Control flow
        while X.abs().sum() > 1:
            X /= 2
        return X.sum()
```

Trong mô hình `FixedHiddenMLP` này, chúng tôi triển khai một lớp ẩn có trọng lượng (`self.rand_weight`) được khởi tạo ngẫu nhiên khi khởi tạo và sau đó là hằng số. Trọng lượng này không phải là một tham số mô hình và do đó nó không bao giờ được cập nhật bằng cách truyền ngược. Sau đó, mạng truyền đầu ra của lớp "cố định" này thông qua một lớp được kết nối hoàn toàn.

Lưu ý rằng trước khi trả lại đầu ra, mô hình của chúng tôi đã làm một cái gì đó bất thường. Chúng tôi chạy một vòng lặp trong khi, thử nghiệm với điều kiện định mức L_1 của nó lớn hơn 1 và chia vector đầu ra của chúng tôi cho 2 cho đến khi nó thỏa mãn điều kiện. Cuối cùng, chúng tôi trả lại tổng các mục trong `X`. Theo kiến thức của chúng tôi, không có mạng thần kinh tiêu chuẩn nào thực hiện hoạt động này. Lưu ý rằng hoạt động cụ thể này có thể không hữu ích trong bất kỳ nhiệm vụ thực tế nào. Quan điểm của chúng tôi chỉ là chỉ cho bạn cách tích hợp mã tùy ý vào luồng tính toán mạng thần kinh của bạn.

```
net = FixedHiddenMLP()
net(X)
```

```
tensor(0.1593, grad_fn=<SumBackward0>)
```

Chúng ta có thể trộn và kết hợp các cách khác nhau để lắp ráp các khối với nhau. Trong ví dụ sau, chúng ta tổ các khối theo một số cách sáng tạo.

```
class NestMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(20, 64), nn.ReLU(),
                                 nn.Linear(64, 32), nn.ReLU())
        self.linear = nn.Linear(32, 16)

    def forward(self, X):
        return self.linear(self.net(X))
```

(continues on next page)

(continued from previous page)

```
chimera = nn.Sequential(NestMLP(), nn.Linear(16, 20), FixedHiddenMLP())
chimera(X)
```

```
tensor(0.0163, grad_fn=<SumBackward0>)
```

6.1.4 Hiệu quả

Người đọc avid có thể bắt đầu lo lắng về hiệu quả của một số hoạt động này. Rốt cuộc, chúng ta có rất nhiều tra cứu từ điển, thực thi mã và rất nhiều thứ Pythonic khác diễn ra trong những gì được cho là một thư viện deep learning hiệu suất cao. Các vấn đề về global interpreter lock[75] của Python được biết đến nhiều. Trong bối cảnh học sâu, chúng ta có thể lo lắng rằng (các) GPU cực nhanh của chúng ta có thể phải đợi cho đến khi một CPU nhỏ chạy mã Python trước khi nó có được một công việc khác để chạy.

6.1.5 Tóm tắt

- Lớp là các khối.
- Nhiều lớp có thể bao gồm một khối.
- Nhiều khối có thể bao gồm một khối.
- Một khối có thể chứa mã.
- Các khối chăm sóc rất nhiều dịch vụ vệ sinh, bao gồm khởi tạo tham số và truyền ngược.
- Các kết nối tuần tự của các lớp và khối được xử lý bởi khối `Sequential`.

6.1.6 Bài tập

1. Những loại vấn đề sẽ xảy ra nếu bạn thay đổi `MySequential` để lưu trữ các khối trong một danh sách Python?
2. Thực hiện một khối lấy hai khối làm đối số, giả sử `net1` và `net2` và trả về đầu ra nối của cả hai mạng trong tuyên truyền chuyển tiếp. Đây còn được gọi là một khối song song.
3. Giả sử rằng bạn muốn nối nhiều phiên bản của cùng một mạng. Triển khai một hàm factory tạo ra nhiều trường hợp của cùng một khối và xây dựng một mạng lớn hơn từ nó.

Discussions[76]

[75] https://wiki.python.org/moin/GlobalInterpreterLock
[76] https://discuss.d2l.ai/t/55

6.2 Quản lý tham số

Khi chúng tôi đã chọn một kiến trúc và đặt các siêu tham số của mình, chúng tôi tiến hành vòng đào tạo, nơi mục tiêu của chúng tôi là tìm các giá trị tham số giảm thiểu chức năng mất mát của chúng tôi. Sau khi đào tạo, chúng tôi sẽ cần các thông số này để đưa ra dự đoán trong tương lai. Ngoài ra, đôi khi chúng tôi sẽ muốn trích xuất các tham số để sử dụng lại chúng trong một số bối cảnh khác, để lưu mô hình của chúng tôi vào đĩa để nó có thể được thực thi trong phần mềm khác hoặc để kiểm tra với hy vọng đạt được sự hiểu biết khoa học.

Hầu hết thời gian, chúng ta sẽ có thể bỏ qua các chi tiết nitty-gritty về cách các tham số được khai báo và thao tác, dựa vào các khuôn khổ học sâu để thực hiện việc nâng nặng. Tuy nhiên, khi chúng ta di chuyển ra khỏi các kiến trúc xếp chồng lên nhau với các lớp tiêu chuẩn, đôi khi chúng ta sẽ cần phải vào cỏ dại khai báo và thao tác các tham số. Trong phần này, chúng tôi đề cập đến những điều sau:

- Truy cập các tham số để gỡ lỗi, chẩn đoán và trực quan hóa.
- Khởi tạo tham số.
- Chia sẻ các thông số trên các thành phần mô hình khác nhau.

Chúng tôi bắt đầu bằng cách tập trung vào MLP với một lớp ẩn.

```
import torch
from torch import nn

net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(), nn.Linear(8, 1))
X = torch.rand(size=(2, 4))
net(X)
```

```
tensor([[0.0251],
        [0.0025]], grad_fn=<AddmmBackward0>)
```

6.2.1 Truy cập tham số

Hãy để chúng tôi bắt đầu với cách truy cập các tham số từ các mô hình mà bạn đã biết. Khi một mô hình được định nghĩa thông qua lớp `Sequential`, trước tiên chúng ta có thể truy cập bất kỳ lớp nào bằng cách lập chỉ mục vào mô hình như thể nó là một danh sách. Các tham số của mỗi lớp được đặt thuận tiện trong thuộc tính của nó. Chúng ta có thể kiểm tra các tham số của lớp được kết nối hoàn toàn thứ hai như sau.

```
print(net[2].state_dict())
```

```
OrderedDict([('weight', tensor([[-0.1270,  0.2448,  0.3073, -0.3091,  0.0385,␣
↪-0.2082,  0.2561, -0.1132]])), ('bias', tensor([-0.1036]))])
```

Đầu ra cho chúng ta biết một vài điều quan trọng. Đầu tiên, lớp kết nối hoàn toàn này chứa hai tham số, tương ứng với trọng lượng và thành kiến của lớp đó, tương ứng. Cả hai đều được lưu trữ dưới dạng phao chính xác đơn (float32). Lưu ý rằng tên của các tham số cho phép chúng ta xác định duy nhất các tham số của từng lớp, ngay cả trong một mạng chứa hàng trăm lớp.

Tham số được nhắm mục tiêu

Lưu ý rằng mỗi tham số được biểu diễn dưới dạng một đối tượng của lớp tham số. Để làm bất cứ điều gì hữu ích với các tham số, trước tiên chúng ta cần truy cập các giá trị số cơ bản. Có một số cách để làm điều này. Một số đơn giản hơn trong khi những người khác nói chung hơn. Đoạn code sau trích xuất sự thiên vị từ lớp mạng nơ-ron thứ hai, nó trả về một đối tượng lớp tham số, và truy cập thêm giá trị của tham số đó.

```
print(type(net[2].bias))
print(net[2].bias)
print(net[2].bias.data)
```

```
<class 'torch.nn.parameter.Parameter'>
Parameter containing:
tensor([-0.1036], requires_grad=True)
tensor([-0.1036])
```

Tham số là các đối tượng phức tạp, chứa các giá trị, độ dốc và thông tin bổ sung. Đó là lý do tại sao chúng ta cần yêu cầu giá trị một cách rõ ràng.

Ngoài giá trị, mỗi tham số cũng cho phép chúng ta truy cập gradient. Bởi vì chúng tôi chưa gọi backpropagation cho mạng này được nêu ra, nó ở trạng thái ban đầu của nó.

```
net[2].weight.grad == None
```

```
True
```

Tất cả các thông số tại một lần

Khi chúng ta cần thực hiện các thao tác trên tất cả các tham số, truy cập chúng từng cái một có thể phát triển tẻ nhạt. Tình hình có thể phát triển đặc biệt khó sử dụng khi chúng ta làm việc với các khối phức tạp hơn (ví dụ, các khối lồng nhau), vì chúng ta cần đệ quy qua toàn bộ cây để trích xuất các tham số của từng khối phụ. Dưới đây chúng tôi chứng minh việc truy cập các tham số của lớp được kết nối hoàn toàn đầu tiên so với truy cập tất cả các lớp.

```
print(*[(name, param.shape) for name, param in net[0].named_parameters()])
print(*[(name, param.shape) for name, param in net.named_parameters()])
```

```
('weight', torch.Size([8, 4])) ('bias', torch.Size([8]))
('0.weight', torch.Size([8, 4])) ('0.bias', torch.Size([8])) ('2.weight',␣
↪torch.Size([1, 8])) ('2.bias', torch.Size([1]))
```

Điều này cung cấp cho chúng tôi một cách khác để truy cập các tham số của mạng như sau.

```
net.state_dict()['2.bias'].data
```

```
tensor([-0.1036])
```

Thu thập các thông số từ các khối lồng nhựa

Chúng ta hãy xem làm thế nào các quy ước đặt tên tham số hoạt động nếu chúng ta tổ nhiều khối bên trong nhau. Đối với điều đó đầu tiên chúng ta xác định một hàm tạo ra các khối (một nhà máy khối, có thể nói) và sau đó kết hợp các khối bên trong nhưng lớn hơn.

```
def block1():
    return nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                         nn.Linear(8, 4), nn.ReLU())

def block2():
    net = nn.Sequential()
    for i in range(4):
        # Nested here
        net.add_module(f'block {i}', block1())
    return net

rgnet = nn.Sequential(block2(), nn.Linear(4, 1))
rgnet(X)
```

```
tensor([[-0.5720],
        [-0.5720]], grad_fn=<AddmmBackward0>)
```

Bây giờ chúng tôi đã thiết kế mạng, chúng ta hãy xem nó được tổ chức như thế nào.

```
print(rgnet)
```

```
Sequential(
  (0): Sequential(
    (block 0): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 1): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 2): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
    (block 3): Sequential(
      (0): Linear(in_features=4, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=4, bias=True)
      (3): ReLU()
    )
  )
```

(continues on next page)

(continued from previous page)

```
  (1): Linear(in_features=4, out_features=1, bias=True)
)
```

Kể từ khi các lớp được phân cấp lồng nhau, chúng ta cũng có thể truy cập chúng như thể lập chỉ mục thông qua các danh sách lồng nhau. Ví dụ, chúng ta có thể truy cập khối chính đầu tiên, bên trong đó khối phụ thứ hai và trong đó sự thiên vị của lớp đầu tiên, với như sau.

```
rgnet[0][1][0].bias.data
```

```
tensor([-0.0210,  0.2743,  0.0433, -0.1879, -0.3919, -0.1085,  0.0300, -0.
↪4445])
```

6.2.2 Khởi tạo tham số

Bây giờ chúng ta đã biết cách truy cập các tham số, chúng ta hãy xem cách khởi tạo chúng đúng cách. Chúng tôi đã thảo luận về sự cần thiết phải khởi tạo thích hợp trong Section 5.8. Khung học sâu cung cấp các khởi tạo ngẫu nhiên mặc định cho các lớp của nó. Tuy nhiên, chúng tôi thường muốn khởi tạo trọng lượng của mình theo các giao thức khác nhau. Khung cung cấp các giao thức được sử dụng phổ biến nhất và cũng cho phép tạo một trình khởi tạo tùy chỉnh.

Theo mặc định, PyTorch khởi tạo các ma trận trọng lượng và thiên vị đồng đều bằng cách vẽ từ một phạm vi được tính toán theo kích thước đầu vào và đầu ra. mô-đun `nn.init` của PyTorch cung cấp một loạt các phương pháp khởi tạo cài sẵn.

Khởi tạo tích hợp

Hãy để chúng tôi bắt đầu bằng cách gọi trên built-in initializers. Mã dưới đây khởi tạo tất cả các tham số trọng lượng dưới dạng biến ngẫu nhiên Gaussian với độ lệch chuẩn 0,01, trong khi các tham số thiên vị bị xóa về 0.

```
def init_normal(m):
    if type(m) == nn.Linear:
        nn.init.normal_(m.weight, mean=0, std=0.01)
        nn.init.zeros_(m.bias)
net.apply(init_normal)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([-0.0152, -0.0023, -0.0003,  0.0055]), tensor(0.))
```

Chúng ta cũng có thể khởi tạo tất cả các tham số thành một giá trị không đổi cho trước (ví dụ, 1).

```
def init_constant(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 1)
        nn.init.zeros_(m.bias)
net.apply(init_constant)
net[0].weight.data[0], net[0].bias.data[0]
```

```
(tensor([1., 1., 1., 1.]), tensor(0.))
```

Chúng tôi cũng có thể áp dụng các trình khởi tạo khác nhau cho các khối nhất định Ví dụ, bên dưới chúng ta khởi tạo lớp đầu tiên với trình khởi tạo Xavier và khởi tạo lớp thứ hai thành giá trị không đổi là 42.

```
def xavier(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
def init_42(m):
    if type(m) == nn.Linear:
        nn.init.constant_(m.weight, 42)

net[0].apply(xavier)
net[2].apply(init_42)
print(net[0].weight.data[0])
print(net[2].weight.data)
```

```
tensor([ 0.1975,  0.2599, -0.1798,  0.4012])
tensor([[42., 42., 42., 42., 42., 42., 42., 42.]])
```

Initializationtùy chỉnh

Đôi khi, các phương pháp khởi tạo chúng ta cần không được cung cấp bởi khung học sâu. Trong ví dụ dưới đây, chúng ta định nghĩa một bộ khởi tạo cho bất kỳ tham số trọng lượng nào w bằng cách sử dụng phân phối lạ sau:

$$w \sim \begin{cases} U(5, 10) & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U(-10, -5) & \text{with probability } \frac{1}{4} \end{cases} \tag{6.2.1}$$

Một lần nữa, chúng tôi triển khai một hàm `my_init` để áp dụng cho `net`.

```
def my_init(m):
    if type(m) == nn.Linear:
        print("Init", *[(name, param.shape)
                        for name, param in m.named_parameters()][0])
        nn.init.uniform_(m.weight, -10, 10)
        m.weight.data *= m.weight.data.abs() >= 5

net.apply(my_init)
net[0].weight[:2]
```

```
Init weight torch.Size([8, 4])
Init weight torch.Size([1, 8])
```

```
tensor([[-9.0877, -7.0061,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  5.4938, -0.0000]], grad_fn=<SliceBackward0>)
```

Lưu ý rằng chúng tôi luôn có tùy chọn đặt tham số trực tiếp.

```
net[0].weight.data[:] += 1
net[0].weight.data[0, 0] = 42
net[0].weight.data[0]
```

```
tensor([42.0000, -6.0061,  1.0000,  1.0000])
```

6.2.3 Tham số Tied

Thông thường, chúng tôi muốn chia sẻ các tham số trên nhiều lớp. Hãy để chúng tôi xem làm thế nào để làm điều này một cách thanh lịch. Sau đây, chúng tôi phân bổ một lớp dày đặc và sau đó sử dụng các tham số của nó đặc biệt để đặt các thông số của một lớp khác.

```
# We need to give the shared layer a name so that we can refer to its
# parameters
shared = nn.Linear(8, 8)
net = nn.Sequential(nn.Linear(4, 8), nn.ReLU(),
                    shared, nn.ReLU(),
                    shared, nn.ReLU(),
                    nn.Linear(8, 1))
net(X)
# Check whether the parameters are the same
print(net[2].weight.data[0] == net[4].weight.data[0])
net[2].weight.data[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[2].weight.data[0] == net[4].weight.data[0])
```

```
tensor([True, True, True, True, True, True, True, True])
tensor([True, True, True, True, True, True, True, True])
```

Ví dụ này cho thấy các tham số của lớp thứ hai và thứ ba được gắn. Chúng không chỉ bằng nhau, chúng được đại diện bởi cùng một tensor chính xác. Do đó, nếu chúng ta thay đổi một trong các tham số, cái kia cũng thay đổi. Bạn có thể tự hỏi, khi tham số được gắn những gì sẽ xảy ra với gradient? Vì các tham số mô hình chứa gradient, các gradient của lớp ẩn thứ hai và lớp ẩn thứ ba được thêm vào với nhau trong quá trình truyền ngược.

6.2.4 Tóm tắt

- Chúng tôi có một số cách để truy cập, khởi tạo và buộc các tham số mô hình.
- Chúng ta có thể sử dụng khởi tạo tùy chỉnh.

6.2.5 Bài tập

1. Sử dụng mô hình `FancyMLP` được xác định trong Section 6.1 và truy cập các tham số của các lớp khác nhau.
2. Nhìn vào tài liệu mô-đun khởi tạo để khám phá các trình khởi tạo khác nhau.
3. Xây dựng một MLP chứa một lớp tham số được chia sẻ và đào tạo nó. Trong quá trình đào tạo, quan sát các thông số mô hình và gradient của mỗi lớp.
4. Tại sao chia sẻ các thông số là một ý tưởng tốt?

Discussions[77]

6.3 Layers tùy chỉnh

Một yếu tố đằng sau sự thành công của deep learning là sự sẵn có của một loạt các lớp có thể được sáng tác theo những cách sáng tạo để thiết kế kiến trúc phù hợp với nhiều nhiệm vụ khác nhau. Ví dụ, các nhà nghiên cứu đã phát minh ra các lớp đặc biệt để xử lý hình ảnh, văn bản, lặp lại dữ liệu tuần tự và thực hiện lập trình động. Sớm hay muộn, bạn sẽ gặp hoặc phát minh ra một lớp chưa tồn tại trong khuôn khổ học sâu. Trong những trường hợp này, bạn phải xây dựng một lớp tùy chỉnh. Trong phần này, chúng tôi chỉ cho bạn như thế nào.

6.3.1 Các lớp không có tham số

Để bắt đầu, chúng tôi xây dựng một lớp tùy chỉnh không có bất kỳ tham số nào của riêng nó. Điều này sẽ trông quen thuộc nếu bạn nhớ lại giới thiệu của chúng tôi để chặn trong Section 6.1. Lớp `CenteredLayer` sau chỉ đơn giản là trừ trung bình từ đầu vào của nó. Để xây dựng nó, chúng ta chỉ cần kế thừa từ lớp lớp cơ sở và thực hiện hàm tuyên truyền chuyển tiếp.

```
import torch
from torch import nn
from torch.nn import functional as F


class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

Hãy để chúng tôi xác minh rằng lớp của chúng tôi hoạt động như dự định bằng cách cung cấp một số dữ liệu thông qua nó.

[77] https://discuss.d2l.ai/t/57

```
layer = CenteredLayer()
layer(torch.FloatTensor([1, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

Bây giờ chúng ta có thể kết hợp layer của chúng ta như một component trong việc xây dựng các mô hình phức tạp hơn.

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
```

Là một kiểm tra thêm sự tỉnh táo, chúng tôi có thể gửi dữ liệu ngẫu nhiên thông qua mạng và kiểm tra xem trung bình có thực tế 0 không. Bởi vì chúng ta đang xử lý các số điểm nổi, chúng ta vẫn có thể thấy một số nonzero rất nhỏ do lượng tử hóa.

```
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(-4.6566e-09, grad_fn=<MeanBackward0>)
```

6.3.2 Các lớp có tham số

Bây giờ chúng ta đã biết cách xác định các lớp đơn giản, chúng ta hãy chuyển sang xác định các lớp với các tham số có thể được điều chỉnh thông qua đào tạo. Chúng ta có thể sử dụng các chức năng tích hợp để tạo các tham số, cung cấp một số chức năng dọn phòng cơ bản. Đặc biệt, họ chi phối quyền truy cập, khởi tạo, chia sẻ, lưu và tải các tham số mô hình. Bằng cách này, trong số các lợi ích khác, chúng tôi sẽ không cần phải viết các thói quen serialization tùy chỉnh cho mọi lớp tùy chỉnh.

Bây giờ chúng ta hãy thực hiện phiên bản riêng của chúng ta của lớp được kết nối hoàn toàn. Nhớ lại rằng lớp này yêu cầu hai tham số, một để đại diện cho trọng lượng và một cho sự thiên vị. Trong triển khai này, chúng tôi nướng trong kích hoạt ReLU dưới dạng mặc định. Lớp này yêu cầu nhập đối số: `in_units` và `units`, biểu thị số lượng đầu vào và đầu ra, tương ứng.

```
class MyLinear(nn.Module):
    def __init__(self, in_units, units):
        super().__init__()
        self.weight = nn.Parameter(torch.randn(in_units, units))
        self.bias = nn.Parameter(torch.randn(units,))
    def forward(self, X):
        linear = torch.matmul(X, self.weight.data) + self.bias.data
        return F.relu(linear)
```

Tiếp theo, chúng tôi khởi tạo lớp `MyLinear` và truy cập các tham số mô hình của nó.

```
linear = MyLinear(5, 3)
linear.weight
```

```
Parameter containing:
tensor([[-0.5447, -0.4223, -0.7320],
        [-0.3532,  1.4021,  0.3656],
```

(continues on next page)

```
        [ 0.4039,  0.0149, -1.7429],
        [-0.9663, -2.5671, -1.0107],
        [ 0.4001, -0.4510,  0.6005]], requires_grad=True)
```

Chúng ta có thể trực tiếp thực hiện các phép tính tuyên truyền chuyển tiếp bằng cách sử dụng các lớp tùy chỉnh

```
linear(torch.rand(2, 5))
```

```
tensor([[1.8603, 0.0000, 0.0000],
        [1.9760, 0.0000, 0.0000]])
```

Chúng ta cũng có thể xây dựng các mô hình bằng cách sử dụng lớp tùy chỉnh Khi chúng ta có rằng chúng ta có thể sử dụng nó giống như lớp được kết nối hoàn toàn tích hợp.

```
net = nn.Sequential(MyLinear(64, 8), MyLinear(8, 1))
net(torch.rand(2, 64))
```

```
tensor([[0.],
        [0.]])
```

6.3.3 Tóm tắt

- Chúng ta có thể thiết kế các lớp tùy chỉnh thông qua lớp lớp cơ bản. Điều này cho phép chúng ta xác định các lớp mới linh hoạt hoạt hoạt khác với bất kỳ lớp hiện có nào trong thư viện.
- Sau khi được xác định, các lớp tùy chỉnh có thể được gọi trong bối cảnh và kiến trúc tùy ý.
- Các lớp có thể có các tham số cục bộ, có thể được tạo thông qua các chức năng tích hợp.

6.3.4 Bài tập

1. Thiết kế một lớp lấy đầu vào và tính toán giảm tensor, tức là nó trả về $y_k = \sum_{i,j} W_{ijk} x_i x_j$.
2. Thiết kế một lớp trả về một nửa hàng đầu của hệ số Fourier của dữ liệu.

Discussions[78]

6.4 Tệp I/O

Cho đến nay chúng tôi đã thảo luận về cách xử lý dữ liệu và cách xây dựng, đào tạo và kiểm tra các mô hình học sâu. Tuy nhiên, tại một số điểm, chúng tôi hy vọng sẽ đủ hạnh phúc với các mô hình đã học mà chúng tôi sẽ muốn lưu kết quả để sử dụng sau này trong các bối cảnh khác nhau (thậm chí có thể đưa ra dự đoán trong triển khai). Ngoài ra, khi chạy một quá trình đào tạo dài, thực hành tốt nhất là lưu định kỳ kết quả trung gian (checkpointing) để đảm bảo rằng chúng tôi không mất vài ngày giá trị tính toán nếu chúng ta đi qua dây nguồn của máy chủ của chúng tôi. Vì vậy, đã đến lúc học cách tải và lưu trữ cả vectơ trọng lượng riêng lẻ và toàn bộ mô hình. Phần này giải quyết cả hai vấn đề.

[78] https://discuss.d2l.ai/t/59

6.4.1 Tải và tiết kiệm Tensors

Đối với hàng chục cá nhân, chúng ta có thể gọi trực tiếp các hàm `load` và `save` để đọc và viết chúng tương ứng. Cả hai chức năng yêu cầu chúng tôi cung cấp một tên, và `save` yêu cầu như là đầu vào biến được lưu.

```
import torch
from torch import nn
from torch.nn import functional as F

x = torch.arange(4)
torch.save(x, 'x-file')
```

Bây giờ chúng ta có thể đọc dữ liệu từ tệp được lưu trữ trở lại vào bộ nhớ.

```
x2 = torch.load('x-file')
x2
```

```
tensor([0, 1, 2, 3])
```

Chúng ta có thể lưu trữ một danh sách các hàng chục và đọc lại vào bộ nhớ.

```
y = torch.zeros(4)
torch.save([x, y],'x-files')
x2, y2 = torch.load('x-files')
(x2, y2)
```

```
(tensor([0, 1, 2, 3]), tensor([0., 0., 0., 0.]))
```

Chúng ta thậm chí có thể viết và đọc một từ điển bản đồ từ chuỗi đến tensors. Điều này thuận tiện khi chúng ta muốn đọc hoặc viết tất cả các trọng lượng trong một mô hình.

```
mydict = {'x': x, 'y': y}
torch.save(mydict, 'mydict')
mydict2 = torch.load('mydict')
mydict2
```

```
{'x': tensor([0, 1, 2, 3]), 'y': tensor([0., 0., 0., 0.])}
```

6.4.2 Tải và lưu thông số mô hình

Tiết kiệm vectơ trọng lượng riêng lẻ (hoặc các hàng chục khác) rất hữu ích, nhưng nó trở nên rất tẻ nhạt nếu chúng ta muốn lưu (và sau đó tải) toàn bộ mô hình. Rốt cuộc, chúng ta có thể có hàng trăm nhóm tham số rắc khắp. Vì lý do này, khung học sâu cung cấp các chức năng tích hợp để tải và lưu toàn bộ mạng. Một chi tiết quan trọng cần lưu ý là điều này lưu mô hình *tham số* chứ không phải toàn bộ mô hình. Ví dụ: nếu chúng ta có MLP 3 lớp, chúng ta cần chỉ định kiến trúc riêng biệt. Lý do cho điều này là bản thân các mô hình có thể chứa mã tùy ý, do đó chúng không thể được serialized như một cách tự nhiên. Do đó, để khôi phục lại một mô hình, chúng ta cần tạo kiến trúc trong mã và sau đó tải các tham số từ đĩa. Hãy để chúng tôi bắt đầu với MLP.quen thuộc của chúng tôi

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(size=(2, 20))
Y = net(X)
```

Tiếp theo, chúng ta lưu trữ các tham số của mô hình dưới dạng tệp với tên "mlp.params".

```
torch.save(net.state_dict(), 'mlp.params')
```

Để khôi phục mô hình, chúng tôi khởi tạo một bản sao của mô hình MLP ban đầu. Thay vì khởi tạo ngẫu nhiên các tham số model, chúng ta đọc các tham số được lưu trữ trong tập tin trực tiếp.

```
clone = MLP()
clone.load_state_dict(torch.load('mlp.params'))
clone.eval()
```

```
MLP(
  (hidden): Linear(in_features=20, out_features=256, bias=True)
  (output): Linear(in_features=256, out_features=10, bias=True)
)
```

Vì cả hai trường hợp đều có cùng tham số mô hình, kết quả tính toán của cùng một đầu vào `X` phải giống nhau. Hãy để chúng tôi xác minh điều này.

```
Y_clone = clone(X)
Y_clone == Y
```

```
tensor([[True, True, True, True, True, True, True, True, True, True],
        [True, True, True, True, True, True, True, True, True, True]])
```

6.4.3 Tóm tắt

- Các chức năng `save` và `load` có thể được sử dụng để thực hiện I/O tệp cho các đối tượng tensor.
- Chúng ta có thể lưu và tải toàn bộ bộ tham số cho một mạng thông qua một từ điển tham số.
- Lưu kiến trúc phải được thực hiện trong mã chứ không phải trong các tham số.

6.4.4 Bài tập

1. Ngay cả khi không cần triển khai các mô hình được đào tạo cho một thiết bị khác, lợi ích thiết thực của việc lưu trữ các thông số mô hình là gì?
2. Giả sử rằng chúng ta chỉ muốn sử dụng lại các phần của mạng để được kết hợp vào một mạng của một kiến trúc khác nhau. Làm thế nào bạn sẽ đi về sử dụng, nói hai lớp đầu tiên từ một mạng trước đó trong một mạng mới?
3. Làm thế nào bạn sẽ đi về lưu kiến trúc mạng và các tham số? Bạn sẽ áp đặt những hạn chế nào đối với kiến trúc?

Discussions[79]

6.5 GPU

Năm Chapter 2.5, chúng tôi đã thảo luận về sự tăng trưởng nhanh chóng của tính toán trong hai thập kỷ qua. Tóm lại, hiệu suất GPU đã tăng hệ số 1000 mỗi thập kỷ kể từ năm 2000. Điều này mang lại những cơ hội tuyệt vời nhưng nó cũng cho thấy một nhu cầu đáng kể để cung cấp hiệu suất như vậy.

Trong phần này, chúng tôi bắt đầu thảo luận về cách khai thác hiệu suất tính toán này cho nghiên cứu của bạn. Đầu tiên bằng cách sử dụng GPU duy nhất và tại một thời điểm sau đó, làm thế nào để sử dụng nhiều GPU và nhiều máy chủ (với nhiều GPU).

Cụ thể, chúng tôi sẽ thảo luận về cách sử dụng một GPU NVIDIA duy nhất để tính toán. Trước tiên, hãy chắc chắn rằng bạn đã cài đặt ít nhất một GPU NVIDIA. Sau đó, tải xuống NVIDIA driver and CUDA[80] và làm theo lời nhắc để đặt đường dẫn thích hợp. Khi các chế phẩm này hoàn tất, lệnh `nvidia-smi` có thể được sử dụng để xem thông tin card đồ hoạ.

```
!nvidia-smi
```

```
Tue Apr 26 12:55:46 2022
+-----------------------------------------------------------------------------
↪+
| NVIDIA-SMI 460.27.04    Driver Version: 460.27.04    CUDA Version: 11.2    ␣
↪|
|-------------------------------+----------------------+----------------------
↪+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC␣
↪|
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M.␣
↪|
|                               |                      |               MIG M.␣
↪|
|===============================+======================+======================|
|   0  Tesla V100-SXM2...  Off  | 00000000:00:1B.0 Off |                    0␣
↪|
| N/A   39C    P0    36W / 300W |      3MiB / 16160MiB |      0%      Default␣
↪|
|                               |                      |                  N/A␣
↪|
```

(continues on next page)

[79] https://discuss.d2l.ai/t/61
[80] https://developer.nvidia.com/cuda-downloads

(continued from previous page)

```
+-------------------------------+----------------------+----------------------
↪+
|   1  Tesla V100-SXM2...  Off  | 00000000:00:1C.0 Off |                    0␣
↪|
| N/A   37C    P0    48W / 300W |      0MiB / 16160MiB |      0%      Default␣
↪|
|                               |                      |                  N/A␣
↪|
+-------------------------------+----------------------+----------------------
↪+
|   2  Tesla V100-SXM2...  Off  | 00000000:00:1D.0 Off |                    0␣
↪|
| N/A   40C    P0    37W / 300W |      3MiB / 16160MiB |      0%      Default␣
↪|
|                               |                      |                  N/A␣
↪|
+-------------------------------+----------------------+----------------------
↪+
|   3  Tesla V100-SXM2...  Off  | 00000000:00:1E.0 Off |                    0␣
↪|
| N/A   39C    P0    52W / 300W |      0MiB / 16160MiB |      4%      Default␣
↪|
|                               |                      |                  N/A␣
↪|
+-------------------------------+----------------------+----------------------
↪+

+-----------------------------------------------------------------------------
↪+
| Processes:                                                                  ␣
↪|
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory␣
↪|
|        ID   ID                                                   Usage      ␣
↪|
|=============================================================================|
|  No running processes found                                                 ␣
↪|
+-----------------------------------------------------------------------------
↪+
```

Trong PyTorch, mỗi mảng có một thiết bị, chúng ta thường gọi nó như là một bối cảnh. Cho đến nay, theo mặc định, tất cả các biến và tính toán liên quan đã được gán cho CPU. Thông thường, các ngữ cảnh khác có thể là các GPU khác nhau. Mọi thứ có thể nhận được thậm chí hairier khi chúng tôi triển khai công việc trên nhiều máy chủ. Bằng cách gán mảng cho ngữ cảnh một cách thông minh, chúng ta có thể giảm thiểu thời gian truyền dữ liệu giữa các thiết bị. Ví dụ: khi đào tạo mạng thần kinh trên máy chủ có GPU, chúng tôi thường thích các tham số của mô hình sống trên GPU.

Tiếp theo, chúng ta cần xác nhận rằng phiên bản GPU của PyTorch được cài đặt. Nếu phiên bản CPU của PyTorch đã được cài đặt, chúng ta cần gỡ cài đặt nó trước. Ví dụ: sử dụng lệnh `pip uninstall torch`, sau đó cài đặt phiên bản PyTorch tương ứng theo phiên bản CIDA của bạn. Giả sử bạn đã cài đặt CDA 10.0, bạn có thể cài đặt phiên bản PyTorch hỗ trợ CDA 10.0 qua `pip install torch-cu100`.

Để chạy các chương trình trong phần này, bạn cần ít nhất hai GPU. Lưu ý rằng điều này có thể là xa hoa đối với hầu hết các máy tính để bàn nhưng nó có thể dễ dàng có sẵn trên đám mây, ví dụ, bằng cách sử dụng các

phiên bản đa GPU AWS EC2. Hầu như tất cả các phần khác làm * không* yêu cầu nhiều GPU. Thay vào đó, điều này chỉ đơn giản là để minh họa cách dữ liệu chảy giữa các thiết bị khác nhau.

6.5.1 Thiết bị vi tính

Chúng tôi có thể chỉ định các thiết bị, chẳng hạn như CPU và GPU, để lưu trữ và tính toán. Theo mặc định, hàng chục được tạo trong bộ nhớ chính và sau đó sử dụng CPU để tính toán nó.

Trong PyTorch, CPU và GPU có thể được chỉ định bởi `torch.device('cpu')` và `torch.device('cuda')`. Cần lưu ý rằng thiết bị `cpu` có nghĩa là tất cả các CPU vật lý và bộ nhớ. Điều này có nghĩa là các tính toán của PyTorch sẽ cố gắng sử dụng tất cả các lõi CPU. Tuy nhiên, một thiết bị `gpu` chỉ đại diện cho một thẻ và bộ nhớ tương ứng. Nếu có nhiều GPU, chúng tôi sử dụng `torch.device(f'cuda:{i}')` để đại diện cho GPU i^{th} (i bắt đầu từ 0). Ngoài ra, `gpu:0` và `gpu` là tương đương.

```
import torch
from torch import nn

torch.device('cpu'), torch.device('cuda'), torch.device('cuda:1')
```

```
(device(type='cpu'), device(type='cuda'), device(type='cuda', index=1))
```

Chúng tôi có thể truy vấn số lượng GPU có sẵn.

```
torch.cuda.device_count()
```

```
2
```

Bây giờ chúng ta xác định hai hàm tiện lợi cho phép chúng tôi chạy mã ngay cả khi GPU được yêu cầu không tồn tại.

```
def try_gpu(i=0):  #@save
    """Return gpu(i) if exists, otherwise return cpu()."""
    if torch.cuda.device_count() >= i + 1:
        return torch.device(f'cuda:{i}')
    return torch.device('cpu')

def try_all_gpus():  #@save
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    devices = [torch.device(f'cuda:{i}')
             for i in range(torch.cuda.device_count())]
    return devices if devices else [torch.device('cpu')]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
 [device(type='cuda', index=0), device(type='cuda', index=1)])
```

6.5.2 Tensors và GPU

Theo mặc định, hàng chục được tạo trên CPU. Chúng ta có thể truy vấn thiết bị nơi tensor được đặt.

```
x = torch.tensor([1, 2, 3])
x.device
```

```
device(type='cpu')
```

Điều quan trọng cần lưu ý là bất cứ khi nào chúng tôi muốn hoạt động theo nhiều điều khoản, chúng cần phải ở trên cùng một thiết bị. Ví dụ, nếu chúng ta tổng hợp hai chục, chúng ta cần đảm bảo rằng cả hai đối số đều sống trên cùng một thiết bị—nếu không khung sẽ không biết lưu trữ kết quả ở đâu hoặc thậm chí làm thế nào để quyết định nơi thực hiện tính toán.

Lưu trữ trên GPU

Có một số cách để lưu trữ một tensor trên GPU. Ví dụ: chúng ta có thể chỉ định một thiết bị lưu trữ khi tạo tensor. Tiếp theo, chúng ta tạo biến tensor `X` trên `gpu` đầu tiên. Tensor được tạo trên GPU chỉ tiêu thụ bộ nhớ của GPU này. Chúng ta có thể sử dụng lệnh `nvidia-smi` để xem mức sử dụng bộ nhớ GPU. Nói chung, chúng ta cần đảm bảo rằng chúng ta không tạo dữ liệu vượt quá giới hạn bộ nhớ GPU.

```
X = torch.ones(2, 3, device=try_gpu())
X
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
```

Giả sử rằng bạn có ít nhất hai GPU, mã sau sẽ tạo tensor ngẫu nhiên trên GPU. thứ hai

```
Y = torch.rand(2, 3, device=try_gpu(1))
Y
```

```
tensor([[7.7027e-01, 1.6829e-01, 2.2538e-01],
        [1.5524e-01, 5.3015e-04, 5.3232e-01]], device='cuda:1')
```

Sao chép

Nếu chúng ta muốn tính `X + Y`, chúng ta cần quyết định nơi thực hiện thao tác này. Ví dụ, như thể hiện trong Fig. 6.5.1, chúng ta có thể chuyển `X` sang GPU thứ hai và thực hiện thao tác ở đó. *Làm không* chỉ cần thêm `X` và `Y`, vì điều này sẽ dẫn đến một ngoại lệ. Công cụ thời gian chạy sẽ không biết phải làm gì: nó không thể tìm thấy dữ liệu trên cùng một thiết bị và nó không thành công. Vì `Y` sống trên GPU thứ hai, chúng ta cần phải di chuyển `X` ở đó trước khi chúng ta có thể thêm hai.

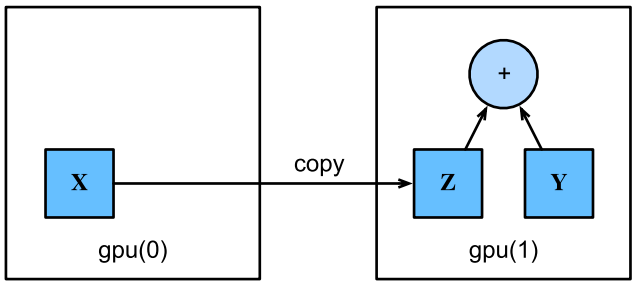


Fig. 6.5.1: Copy data to perform an operation on the same device.

```
Z = X.cuda(1)
print(X)
print(Z)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:0')
tensor([[1., 1., 1.],
        [1., 1., 1.]], device='cuda:1')
```

Bây giờ dữ liệu nằm trên cùng một GPU (cả `Z` và `Y` đều là), chúng ta có thể thêm chúng lên.

```
Y + Z
```

```
tensor([[1.7703, 1.1683, 1.2254],
        [1.1552, 1.0005, 1.5323]], device='cuda:1')
```

Hãy tưởng tượng rằng biến `Z` của bạn đã sống trên GPU thứ hai của bạn. Điều gì sẽ xảy ra nếu chúng ta vẫn gọi `Z.cuda(1)`? Nó sẽ trả về `Z` thay vì tạo một bản sao và phân bổ bộ nhớ mới.

```
Z.cuda(1) is Z
```

```
True
```

Ghi chú bên

Mọi người sử dụng GPU để học máy vì họ mong đợi chúng sẽ nhanh chóng. Nhưng chuyển các biến giữa các thiết bị chậm. Vì vậy, chúng tôi muốn bạn chắc chắn 100% rằng bạn muốn làm điều gì đó chậm trước khi chúng tôi cho phép bạn làm điều đó. Nếu khung học sâu chỉ thực hiện bản sao tự động mà không bị rơi sau đó bạn có thể không nhận ra rằng bạn đã viết một số mã chậm.

Ngoài ra, truyền dữ liệu giữa các thiết bị (CPU, GPU và các máy khác) là thứ chậm hơn nhiều so với tính toán. Nó cũng làm cho việc song song trở nên khó khăn hơn rất nhiều, vì chúng ta phải đợi dữ liệu được gửi (hay đúng hơn là được nhận) trước khi chúng ta có thể tiến hành nhiều thao tác hơn. Đây là lý do tại sao các thao tác sao chép nên được thực hiện cẩn thận. Theo nguyên tắc chung, nhiều hoạt động nhỏ tồi tệ hơn nhiều so với một hoạt động lớn. Hơn nữa, một số hoạt động tại một thời điểm tốt hơn nhiều so với nhiều thao tác đơn lẻ xen kẽ trong mã trừ khi bạn biết những gì bạn đang làm. Đây là trường hợp vì các hoạt động như vậy có thể chặn nếu một thiết bị phải đợi thiết bị kia trước khi nó có thể làm một cái gì đó khác. Nó là một chút

giống như đặt hàng cà phê của bạn trong một hàng đợi thay vì đặt hàng trước nó qua điện thoại và phát hiện ra rằng nó đã sẵn sàng khi bạn đang có.

Cuối cùng, khi chúng ta in hàng chục hoặc chuyển đổi hàng chục sang định dạng NumPy, nếu dữ liệu không nằm trong bộ nhớ chính, khung sẽ sao chép nó vào bộ nhớ chính trước, dẫn đến chi phí truyền bổ sung. Thậm chí tệ hơn, bây giờ nó phải tuân theo khóa thông dịch viên toàn cầu đáng sợ khiến mọi thứ chờ đợi Python hoàn thành.

6.5.3 Mạng nơ-ron và GPU

Tương tự, một mô hình mạng thần kinh có thể chỉ định các thiết bị. Mã sau đặt các tham số mô hình trên GPU.

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
```

Chúng ta sẽ thấy nhiều ví dụ khác về cách chạy các mô hình trên GPU trong các chương sau, đơn giản vì chúng sẽ trở nên chuyên sâu hơn về mặt tính toán.

Khi đầu vào là một tensor trên GPU, mô hình sẽ tính toán kết quả trên cùng một GPU.

```
net(X)
```

```
tensor([[0.1271],
        [0.1271]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

Hãy để chúng tôi xác nhận rằng các tham số mô hình được lưu trữ trên cùng một GPU.

```
net[0].weight.data.device
```

```
device(type='cuda', index=0)
```

Nói tóm lại, miễn là tất cả dữ liệu và tham số đều trên cùng một thiết bị, chúng ta có thể học các mô hình một cách hiệu quả. Trong các chương sau, chúng ta sẽ thấy một số ví dụ như vậy.

6.5.4 Tóm tắt

- Chúng tôi có thể chỉ định các thiết bị để lưu trữ và tính toán, chẳng hạn như CPU hoặc GPU. Theo mặc định, dữ liệu được tạo trong bộ nhớ chính và sau đó sử dụng CPU để tính toán.
- Khung học sâu yêu cầu tất cả dữ liệu đầu vào để tính toán phải nằm trên cùng một thiết bị, có thể là CPU hoặc cùng một GPU.
- Bạn có thể mất hiệu suất đáng kể bằng cách di chuyển dữ liệu mà không cần quan tâm. Một sai lầm điển hình như sau: tính toán tổn thất cho mỗi minibatch trên GPU và báo cáo lại cho người dùng trên dòng lệnh (hoặc đăng nhập nó trong NumPy `ndarray`) sẽ kích hoạt một khóa thông dịch viên toàn cầu ngăn chặn tất cả các GPU. Tốt hơn là phân bổ bộ nhớ để đăng nhập vào GPU và chỉ di chuyển các bản ghi lớn hơn.

6.5.5 Bài tập

1. Hãy thử một nhiệm vụ tính toán lớn hơn, chẳng hạn như phép nhân của ma trận lớn và xem sự khác biệt về tốc độ giữa CPU và GPU. Còn một nhiệm vụ với một lượng nhỏ tính toán thì sao?
2. Làm thế nào chúng ta nên đọc và viết các tham số mô hình trên GPU?
3. Đo thời gian cần thiết để tính toán 1000 phép nhân ma trận ma trận của 100×100 ma trận và ghi lại định mức Frobenius của ma trận đầu ra một kết quả tại một thời điểm so với giữ nhật ký trên GPU và chỉ chuyển kết quả cuối cùng.
4. Đo lường thời gian cần thiết để thực hiện hai phép nhân ma trận ma trận trên hai GPU cùng một lúc so với theo thứ tự trên một GPU. Gợi ý: bạn sẽ thấy tỷ lệ gần như tuyến tính.

Discussions[81]

[81] https://discuss.d2l.ai/t/63

7 | Mạng thần kinh phức tạp

Trong các chương trước đó, chúng tôi đã đưa ra dữ liệu hình ảnh, trong đó mỗi ví dụ bao gồm một lưới pixel hai chiều. Tùy thuộc vào việc chúng ta đang xử lý hình ảnh đen trắng hay màu, mỗi vị trí pixel có thể được liên kết với một hoặc nhiều giá trị số, tương ứng. Cho đến bây giờ, cách đối phó với cấu trúc phong phú này là vô cùng không thỏa mãn. Chúng tôi chỉ đơn giản là loại bỏ cấu trúc không gian của mỗi hình ảnh bằng cách làm phẳng chúng thành các vectơ một chiều, cho chúng ăn thông qua một MLP được kết nối hoàn toàn. Bởi vì các mạng này không thay đổi theo thứ tự của các tính năng, chúng ta có thể nhận được kết quả tương tự bất kể chúng ta giữ nguyên một thứ tự tương ứng với cấu trúc không gian của pixel hay nếu chúng ta hoán vị các cột của ma trận thiết kế trước khi lắp các tham số của MLP. Tốt nhất, chúng tôi sẽ tận dụng kiến thức trước đây của chúng tôi rằng các pixel gần đó thường liên quan đến nhau, để xây dựng các mô hình hiệu quả để học từ dữ liệu hình ảnh.

Chương này giới thiệu *mạng thần kinh phức lượng* (CNN), một họ mạng thần kinh mạnh mẽ được thiết kế cho mục đích chính xác này. Các kiến trúc dựa trên CNN hiện đang phổ biến trong lĩnh vực thị giác máy tính, và đã trở nên chiếm ưu thế đến mức hầu như không ai ngày nay sẽ phát triển một ứng dụng thương mại hoặc tham gia một cuộc thi liên quan đến nhận dạng hình ảnh, phát hiện đối tượng hoặc phân khúc ngữ nghĩa, mà không cần xây dựng cách tiếp cận này.

CNN hiện đại, vì chúng được gọi là thông tục nợ thiết kế của họ để truyền cảm hứng từ sinh học, lý thuyết nhóm và một liều lượng lành mạnh của mày mò thử nghiệm. Ngoài hiệu quả mẫu của chúng trong việc đạt được các mô hình chính xác, CNN có xu hướng hiệu quả về mặt tính toán, cả vì chúng yêu cầu ít tham số hơn các kiến trúc được kết nối hoàn toàn và vì các phức tạp dễ song song trên các lõi GPU. Do đó, các học viên thường áp dụng CNN bất cứ khi nào có thể, và ngày càng họ nổi lên như đối thủ cạnh tranh đáng tin cậy ngay cả trên các nhiệm vụ có cấu trúc trình tự một chiều, chẳng hạn như phân tích chuỗi âm thanh, văn bản và chuỗi thời gian, nơi các mạng thần kinh tái phát được sử dụng thông thường. Một số sự thích nghi thông minh của CNN cũng đã đưa chúng chịu đựng dữ liệu có cấu trúc đồ thị và trong các hệ thống giới thiệu.

Đầu tiên, chúng ta sẽ đi qua các hoạt động cơ bản bao gồm xương sống của tất cả các mạng phức tạp. Chúng bao gồm các lớp phức tạp, chi tiết nitty-gritty bao gồm đệm và sải chân, các lớp tổng hợp được sử dụng để tổng hợp thông tin trên các vùng không gian liền kề, sử dụng nhiều kênh ở mỗi lớp, và một cuộc thảo luận cẩn thận về cấu trúc của kiến trúc hiện đại. Chúng tôi sẽ kết thúc chương với một ví dụ làm việc đầy đủ về LeNet, mạng phức tạp đầu tiên được triển khai thành công, rất lâu trước khi sự gia tăng của học sâu hiện đại. Trong chương tiếp theo, chúng ta sẽ đi sâu vào triển khai đầy đủ một số kiến trúc CNN phổ biến và tương đối gần đây có thiết kế đại diện cho hầu hết các kỹ thuật thường được sử dụng bởi các học viên hiện đại.

7.1 Từ các lớp được kết nối hoàn toàn đến sự phức tạp

Cho đến ngày nay, các mô hình mà chúng tôi đã thảo luận cho đến nay vẫn là các tùy chọn phù hợp khi chúng tôi đang xử lý dữ liệu dạng bảng. Theo bảng, chúng tôi có nghĩa là dữ liệu bao gồm các hàng tương ứng với các ví dụ và cột tương ứng với các tính năng. Với dữ liệu dạng bảng, chúng tôi có thể dự đoán rằng các mẫu chúng tôi tìm kiếm có thể liên quan đến tương tác giữa các tính năng, nhưng chúng tôi không giả định bất kỳ cấu trúc nào * a priori* liên quan đến cách các tính năng tương tác.

Đôi khi, chúng tôi thực sự thiếu kiến thức để hướng dẫn việc xây dựng các kiến trúc craftier. Trong những trường hợp này, MLP có thể là tốt nhất mà chúng ta có thể làm. Tuy nhiên, đối với dữ liệu nhận thức chiều cao, các mạng không có cấu trúc như vậy có thể phát triển không tiện dụng.

Ví dụ, chúng ta hãy quay lại ví dụ chạy của chúng tôi về việc phân biệt mèo với chó. Giả sử rằng chúng tôi thực hiện một công việc kỹ lưỡng trong việc thu thập dữ liệu, thu thập một bộ dữ liệu chú thích của các bức ảnh một megapixel. Điều này có nghĩa là mỗi đầu vào vào mạng có một triệu kích thước. Theo các cuộc thảo luận của chúng tôi về chi phí tham số hóa của các lớp được kết nối hoàn toàn trong Section 4.4.3, ngay cả việc giảm tích cực xuống một nghìn kích thước ẩn sẽ đòi hỏi một lớp được kết nối hoàn toàn đặc trưng bởi các tham số $10^6 \times 10^3 = 10^9$. Trừ khi chúng ta có rất nhiều GPU, một tài năng để tối ưu hóa phân tán và một lượng kiên nhẫn phi thường, việc học các thông số của mạng này có thể trở nên không khả thi.

Một người đọc cẩn thận có thể phản đối lập luận này trên cơ sở rằng độ phân giải một megapixel có thể không cần thiết. Tuy nhiên, trong khi chúng ta có thể thoát khỏi một trăm nghìn pixel, lớp ẩn có kích thước 1000 của chúng tôi đánh giá thấp số lượng đơn vị ẩn cần thiết để tìm hiểu các biểu diễn hình ảnh tốt, vì vậy một hệ thống thực tế vẫn sẽ yêu cầu hàng tỷ tham số. Hơn nữa, học một bộ phân loại bằng cách lắp rất nhiều tham số có thể yêu cầu thu thập một bộ dữ liệu khổng lồ. Nhưng ngày nay cả con người và máy tính đều có thể phân biệt mèo với chó khá tốt, dường như mâu thuẫn với những trực giác này. Đó là do hình ảnh thể hiện cấu trúc phong phú có thể được con người và các mô hình học máy khai thác như nhau. Mạng thần kinh phức tạp (CNN) là một cách sáng tạo mà machine learning đã chấp nhận để khai thác một số cấu trúc đã biết trong hình ảnh tự nhiên.

7.1.1 Bất biến

Hãy tưởng tượng rằng bạn muốn phát hiện một đối tượng trong một hình ảnh. Có vẻ hợp lý rằng bất cứ phương pháp nào chúng ta sử dụng để nhận ra các đối tượng không nên quá quan tâm đến vị trí chính xác của đối tượng trong ảnh. Lý tưởng nhất, hệ thống của chúng ta nên khai thác kiến thức này. Lợn thường không bay và máy bay thường không bơi. Tuy nhiên, chúng ta vẫn nên nhận ra một con lợn là một xuất hiện ở trên cùng của hình ảnh. Chúng ta có thể rút ra một số cảm hứng ở đây từ trò chơi dành cho trẻ em "Where's Waldo" (được mô tả trong Fig. 7.1.1). Trò chơi bao gồm một số cảnh hỗn loạn bùng nổ với các hoạt động. Waldo xuất hiện ở đâu đó trong mỗi, thường ẩn nấp ở một số vị trí khó xảy ra. Mục tiêu của người đọc là xác định vị trí anh ta. Mặc dù trang phục đặc trưng của mình, điều này có thể khó khăn đáng ngạc nhiên, do số lượng lớn các phiền nhiễu. Tuy nhiên, * Waldo trông như vật* không phụ thuộc vào * nơi Waldo được định trong*. Chúng ta có thể quét hình ảnh bằng máy dò Waldo có thể gán điểm cho mỗi bản vá, cho biết khả năng bản vá có chứa Waldo. CNN hệ thống hóa ý tưởng này về * bất biến không gian*, khai thác nó để tìm hiểu các biểu diễn hữu ích với ít tham số hơn.

Fig. 7.1.1: An image of the “Where’s Waldo” game.

Bây giờ chúng ta có thể làm cho những trực giác này trở nên cụ thể hơn bằng cách liệt kê một vài desiderata để hướng dẫn thiết kế kiến trúc mạng thần kinh phù hợp với tầm nhìn máy tính:

1. Trong các lớp sớm nhất, mạng của chúng tôi sẽ phản hồi tương tự như cùng một bản vá, bất kể nó xuất hiện ở đâu trong hình ảnh. Nguyên tắc này được gọi là *translation invariance*.
2. Các lớp sớm nhất của mạng nên tập trung vào các vùng địa phương, mà không quan tâm đến nội dung của hình ảnh ở các vùng xa xôi. Đây là nguyên tắc *locality*. Cuối cùng, các đại diện địa phương này có thể được tổng hợp để đưa ra dự đoán ở cấp độ toàn bộ hình ảnh.

Chúng ta hãy xem điều này chuyển thành toán học như thế nào.

7.1.2 Hạn chế MLP

Để bắt đầu, chúng ta có thể xem xét một MLP với hình ảnh hai chiều $\mathbf{X}$ là đầu vào và các đại diện ẩn ngay lập tức của chúng $\mathbf{H}$ tương tự được biểu diễn dưới dạng ma trận trong toán học và là hàng chục hai chiều trong mã, trong đó cả $\mathbf{X}$ và $\mathbf{H}$ đều có hình dạng giống nhau. Hãy để cái đó chìm vào. Bây giờ chúng ta quan niệm không chỉ các đầu vào mà còn là các đại diện ẩn như sở hữu cấu trúc không gian.

Hãy để $[\mathbf{X}]_{i,j}$ và $[\mathbf{H}]_{i,j}$ biểu thị điểm ảnh tại vị trí (i, j) trong hình ảnh đầu vào và biểu diễn ẩn, tương ứng. Do đó, để mỗi đơn vị ẩn nhận được đầu vào từ mỗi pixel đầu vào, chúng tôi sẽ chuyển từ sử dụng ma trận trọng lượng (như chúng tôi đã làm trước đây trong MLP s) để biểu thị các tham số của chúng tôi dưới dạng trọng lượng thứ tư W. Giả sử rằng $\mathbf{U}$ chứa các thành kiến, chúng ta có thể chính thức thể hiện lớp được kết nối hoàn toàn như

$$\begin{aligned}[\mathbf{H}]_{i,j} &= [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l} [\mathbf{X}]_{k,l} \\ &= [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b} [\mathbf{X}]_{i+a,j+b}.\end{aligned}, \tag{7.1.1}$$

trong đó việc chuyển đổi từ W sang V là hoàn toàn mỹ phẩm cho bây giờ vì có sự tương ứng một-một giữa các hệ số trong cả hai hàng chục bậc bốn. Chúng tôi chỉ cần lập chỉ mục lại các bản đăng ký (k, l) sao cho $k = i + a$ và $l = j + b$. Nói cách khác, chúng tôi đặt $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$. Các chỉ số a và b chạy trên

cả độ lệch tích cực và âm, bao phủ toàn bộ hình ảnh. Đối với bất kỳ vị trí nhất định nào (i, j) trong biểu diễn ẩn $[\mathbf{H}]_{i,j}$, chúng tôi tính giá trị của nó bằng cách tổng hợp các pixel trong x, tập trung vào khoảng (i, j) và có trọng số $[\mathsf{V}]_{i,j,a,b}$.

Dịch Bất biến

Bây giờ chúng ta hãy gọi nguyên tắc đầu tiên được thiết lập ở trên: bất biến dịch. Điều này ngụ ý rằng một sự thay đổi trong đầu vào **X** chỉ đơn giản là dẫn đến một sự thay đổi trong đại diện ẩn **H**. Điều này chỉ có thể xảy ra nếu V và **U** không thực sự phụ thuộc vào (i, j), tức là, chúng tôi có $[\mathsf{V}]_{i,j,a,b} = [\mathbf{V}]_{a,b}$ và **U** là một hằng số, nói u. Kết quả là, chúng ta có thể đơn giản hóa định nghĩa cho **H**:

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{7.1.2}$$

Đây là một * convolution*! Chúng tôi có hiệu quả trọng điểm ảnh tại $(i + a, j + b)$ trong vùng lân cận của vị trí (i, j) với hệ số $[\mathbf{V}]_{a,b}$ để có được giá trị $[\mathbf{H}]_{i,j}$. Lưu ý rằng $[\mathbf{V}]_{a,b}$ cần nhiều hệ số ít hơn $[\mathsf{V}]_{i,j,a,b}$ vì nó không còn phụ thuộc vào vị trí trong ảnh. Chúng tôi đã đạt được những tiến bộ đáng kể!

Địa phương

Bây giờ chúng ta hãy gọi nguyên tắc thứ hai: địa phương. Như động lực ở trên, chúng tôi tin rằng chúng tôi không nên phải nhìn rất xa vị trí (i, j) để thu thập thông tin liên quan để đánh giá những gì đang xảy ra tại $[\mathbf{H}]_{i,j}$. Điều này có nghĩa là bên ngoài một số phạm vi $|a| > \Delta$ hoặc $|b| > \Delta$, chúng ta nên đặt $[\mathbf{V}]_{a,b} = 0$. Tương đương, chúng ta có thể viết lại $[\mathbf{H}]_{i,j}$ như

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{V}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{7.1.3}$$

Lưu ý rằng (7.1.3), tóm lại, là một lớp* phức tạp . **Mạng thần kinh hội lượng* (CNN) là một họ đặc biệt của các mạng thần kinh có chứa các lớp phức tạp. Trong cộng đồng nghiên cứu deep learning, :math:`mathbf{V}` được gọi là một hạt nhân * convolution*, một * filter*, hoặc đơn giản là trọng lượng* của lớp thường là các tham số có thể học được. Khi khu vực địa phương nhỏ, sự khác biệt so với mạng được kết nối hoàn toàn có thể rất ấn tượng. Trong khi trước đây, chúng ta có thể đã yêu cầu hàng tỷ tham số để đại diện cho chỉ một lớp duy nhất trong một mạng xử lý hình ảnh, bây giờ chúng ta thường chỉ cần một vài trăm, mà không làm thay đổi chiều chiều của một trong hai đầu vào hoặc biểu diễn ẩn. Giá phải trả cho việc giảm mạnh mẽ này trong các thông số là các tính năng của chúng tôi hiện đang dịch bất biến và lớp của chúng tôi chỉ có thể kết hợp thông tin cục bộ, khi xác định giá trị của mỗi kích hoạt ẩn. Tất cả việc học phụ thuộc vào sự thiên vị quy nạp áp đặt. Khi sự thiên vị đó đồng ý với thực tế, chúng tôi nhận được các mô hình hiệu quả mẫu mà khái quát hóa tốt với dữ liệu vô hình. Nhưng tất nhiên, nếu những thành kiến đó không đồng ý với thực tế, ví dụ, nếu hình ảnh hóa ra không phải là bất biến dịch thuật, các mô hình của chúng tôi có thể phải vật lộn ngay cả để phù hợp với dữ liệu đào tạo của chúng tôi.

7.1.3 Sự phức tạp

Trước khi đi xa hơn, chúng ta nên xem xét ngắn gọn lý do tại sao hoạt động trên được gọi là một sự phức tạp. Trong toán học, * convolution* giữa hai hàm, giả sử $f, g : \mathbb{R}^d \to \mathbb{R}$ được định nghĩa là

$$(f * g)(\mathbf{x}) = \int f(\mathbf{z})g(\mathbf{x} - \mathbf{z})d\mathbf{z}. \tag{7.1.4}$$

Đó là, chúng tôi đo sự chồng chéo giữa f và g khi một hàm được "lật" và dịch chuyển bởi $\mathbf{x}$. Bất cứ khi nào chúng ta có các đối tượng rời rạc, tích phân biến thành một tổng. Ví dụ, đối với vectơ từ tập hợp các vectơ chiều vô hạn có thể tổng hợp vuông với chỉ số chạy trên $\mathbb{Z}$ chúng ta có được định nghĩa sau:

$$(f * g)(i) = \sum_a f(a)g(i - a). \tag{7.1.5}$$

Đối với hàng chục hai chiều, chúng tôi có một tổng tương ứng với các chỉ số (a, b) cho f và $(i - a, j - b)$ cho g, tương ứng:

$$(f * g)(i, j) = \sum_a \sum_b f(a, b)g(i - a, j - b). \tag{7.1.6}$$

Điều này trông tương tự như (7.1.3), với một sự khác biệt lớn. Thay vì sử dụng $(i + a, j + b)$, chúng tôi đang sử dụng sự khác biệt thay thế. Tuy nhiên, lưu ý rằng sự khác biệt này chủ yếu là mỹ phẩm vì chúng ta luôn có thể phù hợp với ký hiệu giữa (7.1.3) và (7.1.6). Định nghĩa ban đầu của chúng tôi trong (7.1.3) mô tả đúng hơn một * tương quan chéo*. Chúng tôi sẽ trở lại với điều này trong phần sau.

7.1.4 "Where's Waldo" Revisited

Quay trở lại máy dò Waldo của chúng tôi, hãy để chúng tôi xem điều này trông như thế nào. Lớp phức tạp chọn các cửa sổ có kích thước nhất định và nặng cường độ theo bộ lọc V, như được chứng minh trong Fig. 7.1.2. Chúng ta có thể hướng đến việc tìm hiểu một mô hình để bất cứ nơi nào "waldoness" là cao nhất, chúng ta nên tìm một đỉnh trong các biểu diễn lớp ẩn.

Fig. 7.1.2: Detect Waldo.

Kênh

Chỉ có một vấn đề với cách tiếp cận này. Cho đến nay, chúng tôi hạnh phúc bỏ qua rằng hình ảnh bao gồm 3 kênh: đỏ, xanh lá cây và xanh dương. Trong thực tế, hình ảnh không phải là đối tượng hai chiều mà là hàng chục bậc ba, đặc trưng bởi chiều cao, chiều rộng và kênh, ví dụ, với hình dạng $1024 \times 1024 \times 3$ pixel. Trong khi hai trục đầu tiên trong số các trục này liên quan đến các mối quan hệ không gian, trục thứ ba có thể được coi là gán một biểu diễn đa chiều cho mỗi vị trí pixel. Do đó, chúng tôi chỉ số X là $[\mathsf{X}]_{i,j,k}$. Bộ lọc phức tạp phải thích ứng cho phù hợp. Thay vì $[\mathbf{V}]_{a,b}$, bây giờ chúng ta có $[\mathsf{V}]_{a,b,c}$.

Hơn nữa, cũng giống như đầu vào của chúng tôi bao gồm tensor bậc ba, hóa ra là một ý tưởng tốt để xây dựng tương tự các đại diện ẩn của chúng tôi như là hàng chục bậc ba H. Nói cách khác, thay vì chỉ có một biểu diễn ẩn duy nhất tương ứng với từng vị trí không gian, chúng ta muốn có toàn bộ vectơ biểu diễn ẩn tương ứng với từng vị trí không gian. Chúng ta có thể nghĩ về các biểu diễn ẩn như bao gồm một số lưới hai chiều xếp chồng lên nhau. Như trong các đầu vào, đôi khi chúng được gọi là * kênh*. Đôi khi chúng cũng được gọi là *feature maps*, vì mỗi người cung cấp một tập hợp các đối tượng đã học được không gian cho lớp tiếp theo. Trực giác, bạn có thể tưởng tượng rằng ở các lớp thấp hơn gần với đầu vào hơn, một số kênh có thể trở nên chuyên biệt để nhận ra các cạnh trong khi những người khác có thể nhận ra kết cấu.

Để hỗ trợ nhiều kênh trong cả hai đầu vào (X) và biểu diễn ẩn (H), chúng ta có thể thêm tọa độ thứ tư vào V:$[\mathsf{V}]_{a,b,c,d}$. Đặt mọi thứ lại với nhau chúng ta có:

$$[\mathsf{H}]_{i,j,d} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} [\mathsf{V}]_{a,b,c,d} [\mathsf{X}]_{i+a,j+b,c}, \tag{7.1.7}$$

trong đó d lập chỉ mục các kênh đầu ra trong các đại diện ẩn H. Lớp phức tạp tiếp theo sẽ tiếp tục lấy tensor bậc ba, H, làm đầu vào. Nói chung hơn, (7.1.7) là định nghĩa của một lớp phức tạp cho nhiều kênh, trong đó V là một hạt nhân hoặc bộ lọc của lớp.

Vẫn còn nhiều hoạt động mà chúng ta cần giải quyết. Ví dụ, chúng ta cần tìm ra cách kết hợp tất cả các biểu diễn ẩn với một đầu ra duy nhất, ví dụ, liệu có Waldo * bất cứ nơi nào* trong hình ảnh hay không. Chúng ta cũng cần quyết định cách tính toán mọi thứ một cách hiệu quả, cách kết hợp nhiều lớp, chức năng kích hoạt phù hợp và cách đưa ra các lựa chọn thiết kế hợp lý để mang lại các mạng có hiệu quả trong thực tế. Chúng tôi chuyển sang những vấn đề này trong phần còn lại của chương.

7.1.5 Tóm tắt

- Sự bất biến dịch trong hình ảnh ngụ ý rằng tất cả các bản vá của một hình ảnh sẽ được xử lý theo cách tương tự.
- Địa phương có nghĩa là chỉ một khu phố nhỏ của pixel sẽ được sử dụng để tính toán các biểu diễn ẩn tương ứng.
- Trong xử lý hình ảnh, các lớp tích hợp thường yêu cầu ít tham số hơn nhiều so với các lớp được kết nối hoàn toàn.
- CNNS là một họ đặc biệt của các mạng thần kinh có chứa các lớp phức tạp.
- Các kênh trên đầu vào và đầu ra cho phép mô hình của chúng tôi chụp nhiều khía cạnh của một hình ảnh tại mỗi vị trí không gian.

7.1.6 Bài tập

1. Giả sử rằng kích thước của hạt nhân phức tạp là $\Delta = 0$. Cho thấy rằng trong trường hợp này hạt nhân phức tạp thực hiện một MLP độc lập cho mỗi tập hợp các kênh.
2. Tại sao có thể dịch bất biến không phải là một ý tưởng tốt sau khi tất cả?
3. Chúng ta phải giải quyết vấn đề gì khi quyết định cách xử lý các biểu diễn ẩn tương ứng với vị trí pixel ở ranh giới của hình ảnh?
4. Mô tả một lớp phức tạp tương tự cho âm thanh.
5. Bạn có nghĩ rằng các lớp phức tạp cũng có thể được áp dụng cho dữ liệu văn bản? Tại sao hoặc tại sao không?
6. Chứng minh rằng $f * g = g * f$.

Discussions[82]

7.2 Sự phức tạp cho hình ảnh

Bây giờ chúng ta đã hiểu cách các lớp phức tạp hoạt động theo lý thuyết, chúng ta đã sẵn sàng để xem chúng hoạt động như thế nào trong thực tế. Dựa trên động lực của chúng tôi về các mạng thần kinh phức tạp như kiến trúc hiệu quả để khám phá cấu trúc trong dữ liệu hình ảnh, chúng tôi gắn bó với hình ảnh làm ví dụ chạy của chúng tôi.

7.2.1 Hoạt động tương quan chéo

Nhớ lại rằng nói đúng, các lớp phức tạp là một sự nhầm lẫn, vì các hoạt động mà chúng thể hiện được mô tả chính xác hơn là tương quan chéo. Dựa trên mô tả của chúng tôi về các lớp phức tạp trong Section 7.1, trong một lớp như vậy, một tensor đầu vào và một tensor hạt nhân được kết hợp để tạo ra một tensor đầu ra thông qua một cross-correlation operation.

Chúng ta hãy bỏ qua các kênh ngay bây giờ và xem cách điều này hoạt động với dữ liệu hai chiều và biểu diễn ẩn. Trong Fig. 7.2.1, đầu vào là tensor hai chiều với chiều cao 3 và chiều rộng 3. Chúng tôi đánh dấu hình dạng của tensor là 3×3 hoặc (3, 3). Chiều cao và chiều rộng của hạt nhân đều là 2. Hình dạng của cửa sổ *kernel * (hoặc cửa sổ *convolution*) được cho bởi chiều cao và chiều rộng của hạt nhân (ở đây nó là 2×2).

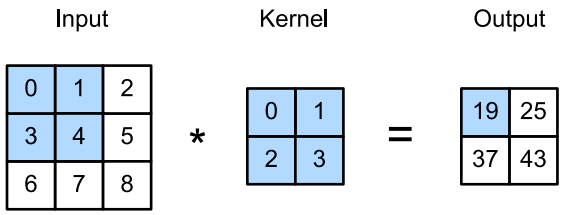


Fig. 7.2.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

Trong hoạt động tương quan chéo hai chiều, chúng ta bắt đầu với cửa sổ covolution được đặt ở góc trên bên trái của tensor đầu vào và trượt nó qua tensor đầu vào, cả từ trái sang phải và trên xuống dưới. Khi cửa sổ

[82] https://discuss.d2l.ai/t/64

covolution trượt đến một vị trí nhất định, subtensor đầu vào chứa trong cửa sổ đó và tensor kernel được nhân elementwise và tensor kết quả được tổng hợp lên mang lại một giá trị vô hướng duy nhất. Kết quả này cho giá trị của tensor đầu ra tại vị trí tương ứng. Ở đây, tensor đầu ra có chiều cao 2 và chiều rộng 2 và bốn phần tử có nguồn gốc từ hoạt động tương quan chéo hai chiều:

$$\begin{aligned}0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\ 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\ 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\ 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.\end{aligned} \tag{7.2.1}$$

Lưu ý rằng dọc theo mỗi trục, kích thước đầu ra nhỏ hơn một chút so với kích thước đầu vào. Bởi vì hạt nhân có chiều rộng và chiều cao lớn hơn một, chúng ta chỉ có thể tính toán tương quan chéo cho các vị trí mà hạt nhân vừa vặn hoàn toàn trong hình ảnh, kích thước đầu ra được đưa ra bởi kích thước đầu vào $n_h \times n_w$ trừ đi kích thước của hạt nhân phức tạp $k_h \times k_w$ qua

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \tag{7.2.2}$$

Đây là trường hợp vì chúng ta cần đủ không gian để "thay đổi" hạt nhân phức tạp trên hình ảnh. Sau đó chúng ta sẽ thấy làm thế nào để giữ cho kích thước không thay đổi bằng cách đệm hình ảnh với các số không xung quanh ranh giới của nó để có đủ không gian để thay đổi hạt nhân. Tiếp theo, chúng tôi triển khai quá trình này trong hàm `corr2d`, chấp nhận tensor đầu vào `X` và tensor kernel `K` và trả về tensor đầu ra `Y`.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def corr2d(X, K):  #@save
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

Chúng ta có thể xây dựng tensor đầu vào `X` và tensor kernel `K` từ Fig. 7.2.1 đến xác nhận đầu ra của việc triển khai ở trên của hoạt động tương quan chéo hai chiều.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

7.2.2 Layers phức tạp

Một lớp phức hợp chéo tương quan đầu vào và hạt nhân và thêm một thiên vị vô hướng để tạo ra một đầu ra. Hai tham số của một lớp tổ hợp là hạt nhân và thiên vị vô hướng. Khi đào tạo các mô hình dựa trên các lớp phức tạp, chúng ta thường khởi tạo các hạt nhân một cách ngẫu nhiên, giống như chúng ta sẽ với một lớp được kết nối hoàn toàn.

Bây giờ chúng ta đã sẵn sàng để triển khai một lớp ghép hai chiều dựa trên hàm `corr2d` được xác định ở trên. Trong hàm xây dựng `__init__`, chúng ta khai báo `weight` và `bias` là hai tham số model. Hàm tuyên truyền chuyển tiếp gọi hàm `corr2d` và thêm sự thiên vị.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Trong $h \times w$ sự phức tạp hoặc một hạt nhân phức tạp $h \times w$, chiều cao và chiều rộng của hạt nhân covolution lần lượt là h và w. Chúng tôi cũng đề cập đến một lớp phức tạp với một hạt nhân phức tạp $h \times w$ đơn giản là một lớp ghép $h \times w$.

7.2.3 Phát hiện cạnh đối tượng trong hình ảnh

Chúng ta hãy dành một chút thời gian để phân tích cú pháp một ứng dụng đơn giản của một lớp phức tạp: phát hiện cạnh của một đối tượng trong một hình ảnh bằng cách tìm vị trí của sự thay đổi điểm ảnh. Đầu tiên, chúng tôi xây dựng một "hình ảnh" của 6×8 pixel. Bốn cột giữa có màu đen (0) và phần còn lại có màu trắng (1).

```
X = torch.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

Tiếp theo, chúng ta xây dựng một hạt nhân `K` với chiều cao 1 và chiều rộng là 2. Khi chúng ta thực hiện thao tác tương quan chéo với đầu vào, nếu các phần tử liền kề theo chiều ngang giống nhau, đầu ra là 0. Nếu không, đầu ra không phải là không.

```
K = torch.tensor([[1.0, -1.0]])
```

Chúng tôi đã sẵn sàng để thực hiện các hoạt động tương quan chéo với các đối số `X` (đầu vào của chúng tôi) và `K` (hạt nhân của chúng tôi). Như bạn có thể thấy, chúng tôi phát hiện 1 cho cạnh từ trắng sang đen và -1 cho cạnh từ đen sang trắng. Tất cả các đầu ra khác đều có giá trị 0.

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

Bây giờ chúng ta có thể áp dụng hạt nhân vào hình ảnh chuyển tiếp. Đúng như dự đoán, nó biến mất. Hạt nhân `K` chỉ phát hiện các viền dọc.

```
corr2d(X.t(), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

7.2.4 Học một hạt nhân

Thiết kế một máy dò cạnh bằng sự khác biệt hữu hạn `[1, -1]` là gọn gàng nếu chúng ta biết đây chính xác là những gì chúng tôi đang tìm kiếm. Tuy nhiên, khi chúng ta nhìn vào các hạt nhân lớn hơn và xem xét các lớp phức tạp liên tiếp, có thể không thể xác định chính xác những gì mỗi bộ lọc nên làm thủ công.

Bây giờ chúng ta hãy xem liệu chúng ta có thể tìm hiểu hạt nhân tạo ra `Y` từ `X` bằng cách xem các cặp đầu vào-đầu ra chỉ. Đầu tiên chúng ta xây dựng một lớp phức tạp và khởi tạo hạt nhân của nó như một tensor ngẫu nhiên. Tiếp theo, trong mỗi lần lặp lại, chúng ta sẽ sử dụng lỗi bình phương để so sánh `Y` với đầu ra của lớp ghép. Sau đó chúng ta có thể tính toán gradient để cập nhật hạt nhân. Vì lợi ích của sự đơn giản, sau đây chúng ta sử dụng lớp tích hợp cho các lớp phức tạp hai chiều và bỏ qua sự thiên vị.

```
# Construct a two-dimensional convolutional layer with 1 output channel and a
# kernel of shape (1, 2). For the sake of simplicity, we ignore the bias here
conv2d = nn.Conv2d(1,1, kernel_size=(1, 2), bias=False)

# The two-dimensional convolutional layer uses four-dimensional input and
# output in the format of (example, channel, height, width), where the batch
# size (number of examples in the batch) and the number of channels are both 1
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
```

(continues on next page)

(continued from previous page)

```
    l.sum().backward()
    # Update the kernel
    conv2d.weight.data[:] -= lr * conv2d.weight.grad
    if (i + 1) % 2 == 0:
        print(f'epoch {i + 1}, loss {l.sum():.3f}')
```

```
epoch 2, loss 13.780
epoch 4, loss 4.541
epoch 6, loss 1.675
epoch 8, loss 0.655
epoch 10, loss 0.263
```

Lưu ý rằng lỗi đã giảm xuống một giá trị nhỏ sau 10 lần lặp lại. Bây giờ chúng ta sẽ hãy xem tensor kernel mà chúng ta đã học được.

```
conv2d.weight.data.reshape((1, 2))
```

```
tensor([[ 1.0415, -0.9363]])
```

Thật vậy, tensor kernel đã học được rất gần với tensor kernel `K` mà chúng tôi đã xác định trước đó.

7.2.5 Tương quan chéo và phức tạp

Nhớ lại quan sát của chúng tôi từ Section 7.1 về sự tương ứng giữa các hoạt động tương quan và phức tạp chéo. Ở đây chúng ta hãy tiếp tục xem xét các lớp phức tạp hai chiều. Điều gì sẽ xảy ra nếu các lớp như vậy thực hiện các hoạt động phức tạp nghiêm ngặt như được định nghĩa trong (7.1.6) thay vì tương quan chéo? Để có được đầu ra của thao tác * convolution* nghiêm ngặt, chúng ta chỉ cần lật tensor hạt nhân hai chiều theo chiều ngang và chiều dọc, sau đó thực hiện thao tác *cross-correlation* với tensor đầu vào.

Đáng chú ý là vì hạt nhân được học từ dữ liệu trong học sâu, các đầu ra của các lớp phức tạp vẫn không bị ảnh hưởng bất kể các lớp như vậy thực hiện các hoạt động phức tạp nghiêm ngặt hoặc các hoạt động tương quan chéo.

Để minh họa điều này, giả sử rằng một lớp phức hợp thực hiện *tương quan chéo * và học hạt nhân trong :numref:`fig_correlation`, được ký hiệu là ma trận :math:`mathbf{K}` ở đây. Giả sử rằng các điều kiện khác vẫn không thay đổi, khi lớp này thực hiện nghiêm ngặt *convolution* thay vào đó, hạt nhân đã học $\mathbf{K}'$ sẽ giống như $\mathbf{K}$ sau khi $\mathbf{K}'$ được lật cả hai chiều ngang và chiều dọc. Điều đó có nghĩa là, khi lớp phức tạp thực hiện nghiêm ngặt * độ lượng* cho đầu vào trong Fig. 7.2.1 và $\mathbf{K}'$, cùng một đầu ra trong Fig. 7.2.1 (tương quan chéo của đầu vào và $\mathbf{K}$) sẽ thu được.

Để phù hợp với thuật ngữ tiêu chuẩn với tài liệu học sâu, chúng ta sẽ tiếp tục đề cập đến hoạt động tương quan chéo như một sự phức tạp mặc dù, nói nghiêm ngặt, nó hơi khác nhau. Bên cạnh đó, chúng ta sử dụng thuật ngữ *element* để chỉ một mục nhập (hoặc component) của bất kỳ tensor đại diện cho một biểu diễn lớp hoặc một hạt nhân phức tạp.

7.2.6 Bản đồ tính năng và trường tiếp nhận

Như được mô tả trong Section 7.1.4, đầu ra lớp phức tạp trong Fig. 7.2.1 đôi khi được gọi là bản đồ *tính năng*, vì nó có thể được coi là biểu diễn (tính năng) đã học trong các kích thước không gian (ví dụ, chiều rộng và chiều cao) cho lớp tiếp theo. Trong CNN, đối với bất kỳ yếu tố x nào của một số lớp, trường tiếp thu* của nódề cập đến tất cả các yếu tố (từ tất cả các lớp trước) có thể ảnh hưởng đến việc tính toán x trong quá trình lan truyền về phía trước. Lưu ý rằng trường tiếp nhận có thể lớn hơn kích thước thực tế của đầu vào.

Chúng ta hãy tiếp tục sử dụng Fig. 7.2.1 để giải thích lĩnh vực tiếp nhận. Với hạt nhân phức tạp 2×2, trường tiếp nhận của phần tử đầu ra bóng mờ (có giá trị 19) là bốn phần tử trong phần bóng mờ của đầu vào. Bây giờ chúng ta hãy biểu thị đầu ra 2×2 là $\mathbf{Y}$ và xem xét một CNN sâu hơn với một lớp phức tạp 2×2 bổ sung lấy $\mathbf{Y}$ làm đầu vào của nó, xuất ra một phần tử duy nhất z. Trong trường hợp này, trường tiếp nhận z trên $\mathbf{Y}$ bao gồm tất cả bốn yếu tố của $\mathbf{Y}$, trong khi trường tiếp nhận trên đầu vào bao gồm tất cả chín yếu tố đầu vào. Do đó, khi bất kỳ yếu tố nào trong bản đồ tính năng cần một trường tiếp nhận lớn hơn để phát hiện các tính năng đầu vào trên một khu vực rộng hơn, chúng ta có thể xây dựng một mạng lưới sâu hơn.

7.2.7 Tóm tắt

- Tính toán cốt lõi của một lớp phức hợp hai chiều là một hoạt động tương quan chéo hai chiều. Ở dạng đơn giản nhất của nó, điều này thực hiện một hoạt động tương quan chéo trên dữ liệu đầu vào hai chiều và hạt nhân, và sau đó thêm một thiên vị.
- Chúng ta có thể thiết kế một hạt nhân để phát hiện các cạnh trong hình ảnh.
- Chúng ta có thể tìm hiểu các tham số của hạt nhân từ dữ liệu.
- Với hạt nhân học được từ dữ liệu, các đầu ra của các lớp phức tạp vẫn không bị ảnh hưởng bất kể các hoạt động được thực hiện của các lớp như vậy (hoặc là sự phức tạp nghiêm ngặt hoặc tương quan chéo).
- Khi bất kỳ yếu tố nào trong bản đồ tính năng cần một trường tiếp nhận lớn hơn để phát hiện các tính năng rộng hơn trên đầu vào, một mạng sâu hơn có thể được xem xét.

7.2.8 Bài tập

1. Xây dựng một hình ảnh `X` với các cạnh chéo.
    1. Điều gì xảy ra nếu bạn áp dụng hạt nhân `K` trong phần này cho nó?
    2. Điều gì xảy ra nếu bạn chuyển `X`?
    3. Điều gì xảy ra nếu bạn chuyển `K`?
2. Khi bạn cố gắng tự động tìm gradient cho lớp `Conv2D` mà chúng tôi đã tạo, bạn thấy loại thông báo lỗi nào?
3. Làm thế nào để bạn đại diện cho một hoạt động tương quan chéo như một phép nhân ma trận bằng cách thay đổi các hàng chục đầu vào và hạt nhân?
4. Thiết kế một số hạt nhân bằng tay.
    1. Hình thức của một hạt nhân cho đạo hàm thứ hai là gì?
    2. Hạt nhân cho một tích phân là gì?
    3. Kích thước tối thiểu của một hạt nhân để có được một đạo hàm của độ d là bao nhiêu?

Discussions[83]

7.3 Đệm và sải chân

Trong ví dụ trước của Fig. 7.2.1, đầu vào của chúng tôi có cả chiều cao và chiều rộng của 3 và hạt nhân phức tạp của chúng tôi có cả chiều cao và chiều rộng 2, mang lại một biểu diễn đầu ra với chiều 2×2. Như chúng ta tổng quát trong Section 7.2, giả sử rằng hình dạng đầu vào là $n_h \times n_w$ và hình dạng hạt nhân phức tạp là $k_h \times k_w$, sau đó hình dạng đầu ra sẽ là $(n_h - k_h + 1) \times (n_w - k_w + 1)$. Do đó, hình dạng đầu ra của lớp phức hợp được xác định bởi hình dạng của đầu vào và hình dạng của hạt nhân phức tạp.

Trong một số trường hợp, chúng tôi kết hợp các kỹ thuật, bao gồm padding và strided convolutions, ảnh hưởng đến kích thước của đầu ra. Là động lực, lưu ý rằng vì hạt nhân thường có chiều rộng và chiều cao lớn hơn 1, sau khi áp dụng nhiều phức tạp liên tiếp, chúng ta có xu hướng cuộn lên với các đầu ra nhỏ hơn đáng kể so với đầu vào của chúng tôi. Nếu chúng ta bắt đầu với một hình ảnh 240×240 pixel, 10 lớp 5×5 làm giảm hình ảnh xuống 200×200 pixel, cắt giảm \$30% \$ của hình ảnh và với nó xóa bỏ bất kỳ thông tin thú vị nào về ranh giới của hình ảnh gốc. *Padding* là công cụ phổ biến nhất để xử lý vấn đề này.

Trong các trường hợp khác, chúng ta có thể muốn giảm kích thước một cách đáng kể, ví dụ, nếu chúng ta thấy độ phân giải đầu vào ban đầu là khó sử dụng. *Sự phức tạp sóng* là một kỹ thuật phổ biến có thể giúp ích trong những trường hợp này.

7.3.1 Đệm

Như đã mô tả ở trên, một vấn đề khó khăn khi áp dụng các lớp phức tạp là chúng ta có xu hướng mất pixel trên chu vi của hình ảnh của chúng ta. Vì chúng ta thường sử dụng hạt nhân nhỏ, đối với bất kỳ sự phức tạp nhất định nào, chúng ta có thể chỉ mất một vài pixel, nhưng điều này có thể cộng lại khi chúng ta áp dụng nhiều lớp phức tạp liên tiếp. Một giải pháp đơn giản cho vấn đề này là thêm các pixel phụ xung quanh ranh giới của hình ảnh đầu vào của chúng ta, do đó làm tăng kích thước hiệu quả của hình ảnh. Thông thường, chúng tôi đặt các giá trị của các pixel bổ sung thành 0. Trong Fig. 7.3.1, chúng tôi pad một 3×3 đầu vào, tăng kích thước của nó lên 5×5. Đầu ra tương ứng sau đó tăng lên ma trận 4×4. Các phần bóng mờ là phần tử đầu ra đầu tiên cũng như các phần tử tensor đầu vào và hạt nhân được sử dụng cho tính toán đầu ra: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

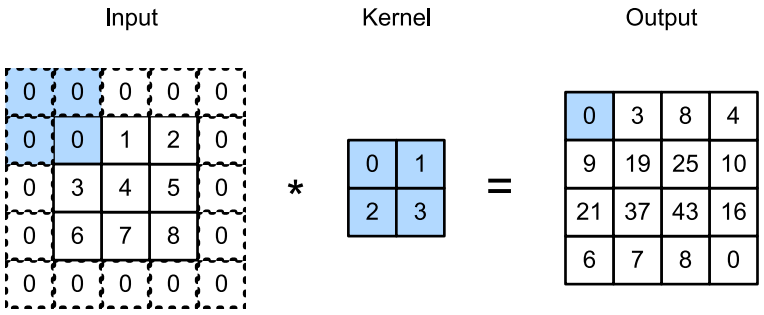


Fig. 7.3.1: Two-dimensional cross-correlation with padding.

Nói chung, nếu chúng ta thêm tổng cộng p_h hàng đệm (khoảng một nửa ở trên cùng và một nửa ở dưới cùng) và tổng cộng p_w cột đệm (khoảng một nửa ở bên trái và một nửa bên phải), hình dạng đầu ra sẽ là

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \tag{7.3.1}$$

[83] https://discuss.d2l.ai/t/66

Điều này có nghĩa là chiều cao và chiều rộng của đầu ra sẽ tăng lần lượt là p_h và p_w.

Trong nhiều trường hợp, chúng tôi sẽ muốn đặt $p_h = k_h - 1$ và $p_w = k_w - 1$ để cung cấp cho đầu vào và đầu ra cùng chiều cao và chiều rộng. Điều này sẽ giúp dự đoán hình dạng đầu ra của mỗi lớp dễ dàng hơn khi xây dựng mạng. Giả sử rằng k_h là lẻ ở đây, chúng tôi sẽ pad $p_h/2$ hàng trên cả hai mặt của chiều cao. Nếu k_h là chẵn, một khả năng là pad $\lceil p_h/2 \rceil$ hàng trên đầu vào và $\lfloor p_h/2 \rfloor$ hàng ở phía dưới. Chúng tôi sẽ pad cả hai mặt của chiều rộng theo cùng một cách.

CNN thường sử dụng hạt nhân phức tạp với các giá trị chiều cao và chiều rộng lẻ, chẳng hạn như 1, 3, 5 hoặc 7. Chọn kích thước hạt nhân lẻ có lợi ích là chúng ta có thể giữ nguyên chiều không gian trong khi đệm với cùng số hàng ở trên và dưới cùng, và cùng một số cột ở bên trái và bên phải.

Hơn nữa, thực hành này sử dụng hạt nhân lẻ và đệm để bảo tồn chính xác chiều hướng mang lại lợi ích văn thư. Đối với bất kỳ tensor hai chiều nào `X`, khi kích thước của hạt nhân là lẻ và số lượng hàng và cột đệm ở tất cả các mặt đều giống nhau, tạo ra một đầu ra có cùng chiều cao và chiều rộng với đầu vào, chúng ta biết rằng đầu ra `Y[i, j]` được tính bằng tương quan chéo của hạt nhân đầu vào và kết hợp với cửa sổ tập trung vào `X[i, j]`.

Trong ví dụ sau, chúng ta tạo ra một lớp ghép hai chiều với chiều cao và chiều rộng là 3 và áp dụng 1 pixel đệm trên tất cả các bên. Cho một đầu vào có chiều cao và chiều rộng 8, chúng ta thấy rằng chiều cao và chiều rộng của đầu ra cũng là 8.

```
import torch
from torch import nn


# We define a convenience function to calculate the convolutional layer. This
# function initializes the convolutional layer weights and performs
# corresponding dimensionality elevations and reductions on the input and
# output
def comp_conv2d(conv2d, X):
    # Here (1, 1) indicates that the batch size and the number of channels
    # are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: examples and
    # channels
    return Y.reshape(Y.shape[2:])
# Note that here 1 row or column is padded on either side, so a total of 2
# rows or columns are added
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

Khi chiều cao và chiều rộng của hạt nhân phức tạp khác nhau, chúng ta có thể làm cho đầu ra và đầu vào có cùng chiều cao và chiều rộng bằng cách thiết lập các số đệm khác nhau cho chiều cao và chiều rộng.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on either side of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2d(1, 1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

7.3.2 Sải chân

Khi tính toán tương quan chéo, chúng ta bắt đầu với cửa sổ covolution ở góc trên bên trái của tensor đầu vào, và sau đó trượt nó trên tất cả các vị trí cả xuống và bên phải. Trong các ví dụ trước, chúng ta mặc định trượt một phần tử tại một thời điểm. Tuy nhiên, đôi khi, cho hiệu quả tính toán hoặc vì chúng tôi muốn hạ mẫu, chúng tôi di chuyển cửa sổ của chúng tôi nhiều hơn một phần tử tại một thời điểm, bỏ qua các vị trí trung gian.

Chúng tôi đề cập đến số lượng hàng và cột đi qua trên mỗi slide là * stride*. Cho đến nay, chúng tôi đã sử dụng các bước tiến của 1, cả về chiều cao và chiều rộng. Đôi khi, chúng ta có thể muốn sử dụng một sải chân lớn hơn. Fig. 7.3.2 cho thấy một hoạt động tương quan chéo hai chiều với một sải chân 3 theo chiều dọc và 2 chiều ngang. Các phần bóng mờ là các phần tử đầu ra cũng như các phần tử tensor đầu vào và hạt nhân được sử dụng cho tính toán đầu ra: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$. Chúng ta có thể thấy rằng khi phần tử thứ hai của cột đầu tiên được xuất ra, cửa sổ covolution trượt xuống ba hàng. Cửa sổ phức tạp trượt hai cột sang phải khi phần tử thứ hai của hàng đầu tiên được xuất ra. Khi cửa sổ covolution tiếp tục trượt hai cột sang phải trên đầu vào, không có đầu ra vì phần tử đầu vào không thể điền vào cửa sổ (trừ khi chúng ta thêm một cột đệm khác).

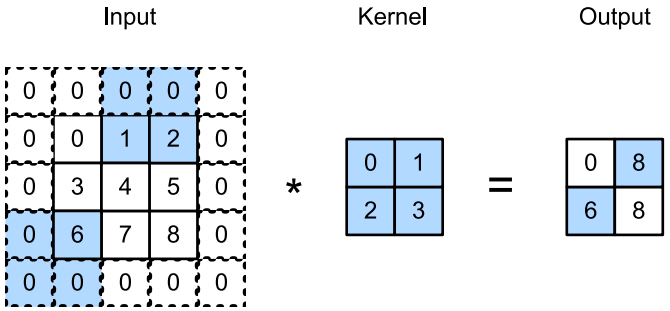


Fig. 7.3.2: Cross-correlation with strides of 3 and 2 for height and width, respectively.

Nói chung, khi sải chân cho chiều cao là s_h và sải chân cho chiều rộng là s_w, hình dạng đầu ra là

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \tag{7.3.2}$$

Nếu chúng ta đặt $p_h = k_h - 1$ và $p_w = k_w - 1$, thì hình dạng đầu ra sẽ được đơn giản hóa thành $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$. Tiến thêm một bước, nếu chiều cao và chiều rộng đầu vào được chia hết bởi các bước nhảy trên chiều cao và chiều rộng, thì hình dạng đầu ra sẽ là $(n_h/s_h) \times (n_w/s_w)$.

Dưới đây, chúng tôi đặt các bước tiến trên cả chiều cao và chiều rộng thành 2, do đó giảm một nửa chiều cao và chiều rộng đầu vào.

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

Tiếp theo, chúng ta sẽ xem xét một ví dụ phức tạp hơn.

```
conv2d = nn.Conv2d(1, 1, kernel_size=(3, 5), padding=(0, 1), stride=(3, 4))
comp_conv2d(conv2d, X).shape
```

```
torch.Size([2, 2])
```

Vì lợi ích của ngắn gọn, khi số đệm ở cả hai mặt của chiều cao và chiều rộng đầu vào lần lượt là p_h và p_w, chúng tôi gọi là đệm (p_h, p_w). Cụ thể, khi $p_h = p_w = p$, đệm là p. Khi các bước tiến trên chiều cao và chiều rộng là s_h và s_w, tương ứng, chúng tôi gọi sải chân (s_h, s_w). Cụ thể, khi $s_h = s_w = s$, sải chân là s. Theo mặc định, đệm là 0 và sải chân là 1. Trong thực tế, chúng ta hiếm khi sử dụng các bước tiến hoặc đệm không đồng nhất, tức là, chúng ta thường có $p_h = p_w$ và $s_h = s_w$.

7.3.3 Tóm tắt

- Đệm có thể làm tăng chiều cao và chiều rộng của đầu ra. Điều này thường được sử dụng để cung cấp cho đầu ra cùng chiều cao và chiều rộng như đầu vào.
- Sải chân có thể làm giảm độ phân giải của đầu ra, ví dụ giảm chiều cao và chiều rộng của đầu ra xuống chỉ $1/n$ chiều cao và chiều rộng của đầu vào (n là một số nguyên lớn hơn 1).
- Đệm và sải chân có thể được sử dụng để điều chỉnh kích thước của dữ liệu một cách hiệu quả.

7.3.4 Bài tập

1. Đối với ví dụ cuối cùng trong phần này, sử dụng toán học để tính hình dạng đầu ra để xem nó có phù hợp với kết quả thực nghiệm hay không.
2. Hãy thử các kết hợp đệm và sải chân khác trên các thí nghiệm trong phần này.
3. Đối với tín hiệu âm thanh, một sải chân của 2 tương ứng với những gì?
4. Những lợi ích tính toán của một sải chân lớn hơn 1 là gì?

Discussions[84]

7.4 Nhiều kênh đầu vào và nhiều đầu ra

Mặc dù chúng tôi đã mô tả nhiều kênh bao gồm mỗi hình ảnh (ví dụ: hình ảnh màu có các kênh RGB tiêu chuẩn để chỉ số lượng màu đỏ, xanh lá cây và xanh dương) và các lớp phức tạp cho nhiều kênh trong Section 7.1.4, cho đến bây giờ, chúng tôi đã đơn giản hóa tất cả các ví dụ số của mình bằng cách làm việc với chỉ một đầu vào và một kênh đầu ra duy nhất. Điều này đã cho phép chúng tôi nghĩ về các đầu vào của chúng tôi, hạt nhân phức tạp, và đầu ra mỗi cái là hàng chục hai chiều.

Khi chúng tôi thêm các kênh vào hỗn hợp, các đầu vào và biểu diễn ẩn của chúng tôi đều trở thành hàng chục ba chiều. Ví dụ, mỗi hình ảnh đầu vào RGB có hình dạng $3 \times h \times w$. Chúng tôi đề cập đến trục này, với kích thước 3, như kích thước * kênh*. Trong phần này, chúng ta sẽ xem xét sâu hơn về các hạt nhân phức tạp với nhiều kênh đầu vào và nhiều kênh đầu ra.

[84] https://discuss.d2l.ai/t/68

7.4.1 Nhiều kênh đầu vào

Khi dữ liệu đầu vào chứa nhiều kênh, chúng ta cần xây dựng một hạt nhân phức tạp có cùng số kênh đầu vào với dữ liệu đầu vào, để nó có thể thực hiện tương quan chéo với dữ liệu đầu vào. Giả sử rằng số kênh cho dữ liệu đầu vào là c_i, số kênh đầu vào của hạt nhân covolution cũng cần phải là c_i. Nếu hình dạng cửa sổ của hạt nhân phức tạp của chúng ta là $k_h \times k_w$, thì khi $c_i = 1$, chúng ta có thể nghĩ về hạt nhân phức tạp của chúng ta như chỉ là một tensor hai chiều của hình dạng $k_h \times k_w$.

Tuy nhiên, khi $c_i > 1$, chúng ta cần một hạt nhân chứa một tensor của hình dạng $k_h \times k_w$ cho *every* kênh đầu vào. Nối các số lượng c_i này lại với nhau mang lại một hạt nhân phức tạp của hình dạng $c_i \times k_h \times k_w$. Vì hạt nhân đầu vào và convolution mỗi kênh có c_i kênh, chúng ta có thể thực hiện một hoạt động tương quan chéo trên tensor hai chiều của đầu vào và tensor hai chiều của hạt nhân phức tạp cho mỗi kênh, thêm các kết quả c_i với nhau (tổng hợp qua các kênh) để mang lại một hai- tensor chiều. Đây là kết quả của mối tương quan chéo hai chiều giữa đầu vào đa kênh và hạt nhân phức tạp đa đầu vào kênh.

Trong Fig. 7.4.1, chúng tôi chứng minh một ví dụ về mối tương quan chéo hai chiều với hai kênh đầu vào. Các phần bóng mờ là phần tử đầu ra đầu tiên cũng như các phần tử tensor đầu vào và hạt nhân được sử dụng cho tính toán đầu ra: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

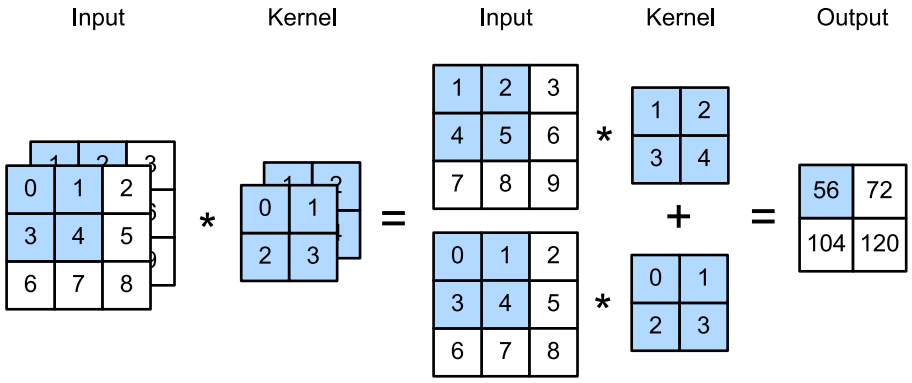


Fig. 7.4.1: Cross-correlation computation with 2 input channels.

Để đảm bảo rằng chúng ta thực sự hiểu những gì đang xảy ra ở đây, chúng ta có thể thực hiện các hoạt động tương quan chéo với nhiều kênh đầu vào chính mình. Lưu ý rằng tất cả những gì chúng tôi đang làm là thực hiện một hoạt động tương quan chéo trên mỗi kênh và sau đó thêm kết quả.

```
import torch
from d2l import torch as d2l
```

```
def corr2d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimension) of `X` and
    # `K`. Then, add them together
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

Chúng ta có thể xây dựng tensor đầu vào `X` và tensor kernel `K` tương ứng với các giá trị trong Fig. 7.4.1 đến xác nhận đầu ra của hoạt động tương quan chéo.

```
X = torch.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                  [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = torch.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])
```

(continues on next page)

(continued from previous page)

```
corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

7.4.2 Nhiều kênh đầu ra

Bất kể số lượng kênh đầu vào, cho đến nay chúng tôi luôn kết thúc với một kênh đầu ra. Tuy nhiên, như chúng ta đã thảo luận trong Section 7.1.4, hóa ra là điều cần thiết để có nhiều kênh ở mỗi lớp. Trong các kiến trúc mạng thần kinh phổ biến nhất, chúng tôi thực sự tăng kích thước kênh khi chúng tôi tăng cao hơn trong mạng thần kinh, thường là lấy mẫu xuống để giảm độ phân giải không gian cho độ sâu kênh * lớn hơn*. Trực giác, bạn có thể nghĩ về từng kênh là phản hồi một số bộ tính năng khác nhau. Thực tế phức tạp hơn một chút so với những giải thích ngây thơ nhất của trực giác này vì các đại diện không được học độc lập nhưng khá tối ưu hóa để cùng hữu ích. Vì vậy, có thể không phải là một kênh duy nhất học một máy dò cạnh mà là một số hướng trong không gian kênh tương ứng với việc phát hiện các cạnh.

Biểu thị bởi c_i và c_o số lượng các kênh đầu vào và đầu ra, tương ứng, và để k_h và k_w là chiều cao và chiều rộng của hạt nhân. Để có được một đầu ra với nhiều kênh, chúng ta có thể tạo một tensor hạt nhân của hình dạng $c_i \times k_h \times k_w$ cho * mỗi kênh đầu ra. Chúng tôi nối chúng trên kích thước kênh đầu ra, để hình dạng của hạt nhân phức tạp là $c_o \times c_i \times k_h \times k_w$. Trong các hoạt động tương quan chéo, kết quả trên mỗi kênh đầu ra được tính từ hạt nhân tích lũy tương ứng với kênh đầu ra đó và lấy đầu vào từ tất cả các kênh trong tensor đầu vào.

Chúng tôi thực hiện một hàm tương quan chéo để tính toán đầu ra của nhiều kênh như hình dưới đây.

```
def corr2d_multi_in_out(X, K):
    # Iterate through the 0th dimension of `K`, and each time, perform
    # cross-correlation operations with input `X`. All of the results are
    # stacked together
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```

Chúng tôi xây dựng một hạt nhân phức tạp với 3 kênh đầu ra bằng cách nối tensor kernel `K` với `K+1` (cộng với một cho mỗi phần tử trong `K`) và `K+2`.

```
K = torch.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

Dưới đây, chúng tôi thực hiện các thao tác tương quan chéo trên tensor đầu vào `X` với tensor kernel `K`. Bây giờ đầu ra chứa 3 kênh. Kết quả của kênh đầu tiên phù hợp với kết quả của tensor đầu vào trước `X` và kênh đa đầu vào, hạt nhân kênh đơn đầu ra.

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],
```

(continues on next page)

```
        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

7.4.3 1×1 Lớp phức tạp

Lúc đầu, một 1×1 convolution, tức là, $k_h = k_w = 1$, dường như không có ý nghĩa nhiều. Rốt cuộc, một sự covolution tương quan các pixel liền kề. Một sự phức tạp 1×1 rõ ràng là không. Tuy nhiên, chúng là những hoạt động phổ biến đôi khi được bao gồm trong các thiết kế của các mạng sâu phức tạp. Hãy để chúng tôi xem một số chi tiết những gì nó thực sự làm.

Bởi vì cửa sổ tối thiểu được sử dụng, sự phức tạp 1×1 mất khả năng của các lớp phức tạp lớn hơn để nhận ra các mẫu bao gồm các tương tác giữa các phần tử liền kề trong chiều cao và chiều rộng. Tính toán duy nhất của sự phức tạp 1×1 xảy ra trên kích thước kênh.

Fig. 7.4.2 cho thấy tính toán tương quan chéo bằng cách sử dụng hạt nhân phức tạp 1×1 với 3 kênh đầu vào và 2 kênh đầu ra. Lưu ý rằng các đầu vào và đầu ra có cùng chiều cao và chiều rộng. Mỗi phần tử trong đầu ra được bắt nguồn từ sự kết hợp tuyến tính của các phần tử * ở cùng một vị trí* trong hình ảnh đầu vào. Bạn có thể nghĩ về lớp ghép 1×1 là cấu thành một lớp kết nối hoàn toàn được áp dụng tại mọi vị trí pixel duy nhất để chuyển đổi các giá trị đầu vào tương ứng c_i thành các giá trị đầu ra c_o. Bởi vì đây vẫn là một lớp phức tạp, các trọng lượng được gắn trên vị trí pixel. Do đó lớp phức tạp 1×1 đòi hỏi $c_o \times c_i$ trọng lượng (cộng với sự thiên vị).

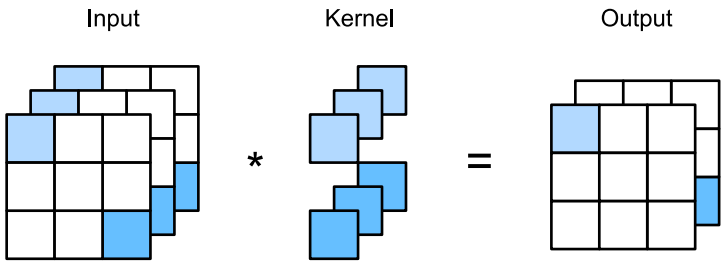


Fig. 7.4.2: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The input and output have the same height and width.

Hãy để chúng tôi kiểm tra xem điều này có hoạt động trong thực tế hay không: chúng tôi thực hiện một sự phức tạp 1×1 bằng cách sử dụng một lớp được kết nối hoàn toàn. Điều duy nhất là chúng ta cần thực hiện một số điều chỉnh đối với hình dạng dữ liệu trước và sau khi nhân ma trận.

```
def corr2d_multi_in_out_1x1(X, K):
    c_i, h, w = X.shape
    c_o = K.shape[0]
    X = X.reshape((c_i, h * w))
    K = K.reshape((c_o, c_i))
    # Matrix multiplication in the fully-connected layer
    Y = torch.matmul(K, X)
    return Y.reshape((c_o, h, w))
```

Khi thực hiện sự phức tạp 1×1, hàm trên tương đương với hàm tương quan chéo được triển khai trước đó `corr2d_multi_in_out`. Hãy để chúng tôi kiểm tra điều này với một số dữ liệu mẫu.

```
X = torch.normal(0, 1, (3, 3, 3))
K = torch.normal(0, 1, (2, 3, 1, 1))
```

```
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
assert float(torch.abs(Y1 - Y2).sum()) < 1e-6
```

7.4.4 Tóm tắt

- Nhiều kênh có thể được sử dụng để mở rộng các tham số mô hình của lớp phức tạp.
- Lớp kết nối 1×1 tương đương với lớp kết nối hoàn toàn, khi được áp dụng trên cơ sở trên mỗi pixel.
- Lớp phức tạp 1×1 thường được sử dụng để điều chỉnh số kênh giữa các lớp mạng và để kiểm soát độ phức tạp của mô hình.

7.4.5 Bài tập

1. Giả sử rằng chúng ta có hai hạt nhân phức tạp có kích thước k_1 và k_2, tương ứng (không có phi tuyến tính ở giữa).
    1. Chứng minh rằng kết quả của hoạt động có thể được thể hiện bằng một sự phức tạp duy nhất.
    2. Chiều của sự phức tạp đơn tương đương là gì?
    3. Cuộc trò chuyện có đúng không?
2. Giả sử một đầu vào của hình $c_i \times h \times w$ và một hạt nhân phức tạp của hình $c_o \times c_i \times k_h \times k_w$, đệm của (p_h, p_w), và sải chân của (s_h, s_w).
    1. Chi phí tính toán (nhân và bổ sung) cho việc tuyên truyền chuyển tiếp là bao nhiêu?
    2. Dấu chân bộ nhớ là gì?
    3. Dấu chân bộ nhớ cho tính toán ngược là gì?
    4. Chi phí tính toán cho việc truyền ngược là bao nhiêu?
3. Theo yếu tố nào số lượng tính toán tăng lên nếu chúng ta tăng gấp đôi số lượng kênh đầu vào c_i và số lượng kênh đầu ra c_o? Điều gì sẽ xảy ra nếu chúng ta tăng gấp đôi đệm?
4. Nếu chiều cao và chiều rộng của một hạt nhân phức tạp là $k_h = k_w = 1$, độ phức tạp tính toán của sự lan truyền về phía trước là gì?
5. Các biến `Y1` và `Y2` trong ví dụ cuối cùng của phần này có giống hệt nhau không? Tại sao?
6. Làm thế nào bạn sẽ thực hiện các phức tạp bằng cách sử dụng phép nhân ma trận khi cửa sổ phức tạp không phải là 1×1?

Discussions[85]

[85] https://discuss.d2l.ai/t/70

7.5 Pooling

Thông thường, khi chúng tôi xử lý hình ảnh, chúng tôi muốn giảm dần độ phân giải không gian của các biểu diễn ẩn của chúng tôi, tổng hợp thông tin để chúng ta đi lên cao hơn trong mạng, trường tiếp nhận càng lớn (trong đầu vào) mà mỗi nút ẩn nhạy cảm.

Thường thì nhiệm vụ cuối cùng của chúng tôi hỏi một số câu hỏi toàn cầu về hình ảnh, ví dụ: * nó có chứa một con mèo không? * Vì vậy, thông thường các đơn vị của lớp cuối cùng của chúng ta phải nhạy cảm với toàn bộ đầu vào. Bằng cách dần dần tổng hợp thông tin, mang lại bản đồ thô hơn và thô hơn, chúng tôi hoàn thành mục tiêu này cuối cùng là học một đại diện toàn cầu, đồng thời giữ tất cả các lợi thế của các lớp phức tạp ở các lớp xử lý trung gian.

Hơn nữa, khi phát hiện các tính năng cấp thấp hơn, chẳng hạn như các cạnh (như đã thảo luận trong Section 7.2), chúng ta thường muốn các đại diện của mình có phần bất biến với bản dịch. Ví dụ, nếu chúng ta chụp ảnh `X` với sự phân định sắc nét giữa đen và trắng và thay đổi toàn bộ hình ảnh theo một pixel sang phải, tức là `Z[i, j] = X[i, j + 1]`, thì đầu ra cho hình ảnh mới `Z` có thể rất khác nhau. Các cạnh sẽ thay đổi bởi một pixel. Trong thực tế, các vật thể hầu như không bao giờ xảy ra chính xác ở cùng một nơi. Trên thực tế, ngay cả với một chân máy và một đối tượng đứng yên, rung động của máy ảnh do chuyển động của màn trập có thể thay đổi mọi thứ bằng một pixel hoặc lâu hơn (máy ảnh cao cấp được tải với các tính năng đặc biệt để giải quyết vấn đề này).

Phần này giới thiệu các lớp * pooling*, phục vụ các mục đích kép của việc giảm thiểu độ nhạy của các lớp phức tạp đến vị trí và các biểu diễn lấy mẫu không gian.

7.5.1 Pooling tối đa và Pooling trung bình

Giống như các lớp phức tạp, các toán tử * bộ* bao gồm một cửa sổ hình dạng cố định được trượt trên tất cả các vùng trong đầu vào theo sải chân của nó, tính toán một đầu ra duy nhất cho mỗi vị trí đi qua bởi cửa sổ hình dạng cố định (đôi khi được gọi là cửa sổ * gộp lọc*). Tuy nhiên, không giống như tính toán tương quan chéo của các đầu vào và hạt nhân trong lớp phức tạp, lớp tổng hợp không chứa tham số (không có * kernel*). Thay vào đó, toán tử tổng hợp là xác định, thường tính toán giá trị tối đa hoặc giá trị trung bình của các phần tử trong cửa sổ tổng hợp. Các hoạt động này được gọi là * bể tối đa* (* tổng hợp tối đa* cho ngắn hạn) và * trung bình*, tương ứng.

Trong cả hai trường hợp, như với toán tử tương quan chéo, chúng ta có thể nghĩ về cửa sổ tổng hợp như bắt đầu từ phía trên bên trái của tensor đầu vào và trượt qua tensor đầu vào từ trái sang phải và trên xuống dưới. Tại mỗi vị trí mà cửa sổ pooling nhấn, nó tính toán giá trị lớn nhất hoặc trung bình của subtensor đầu vào trong cửa sổ, tùy thuộc vào việc tổng hợp tối đa hay trung bình được sử dụng.

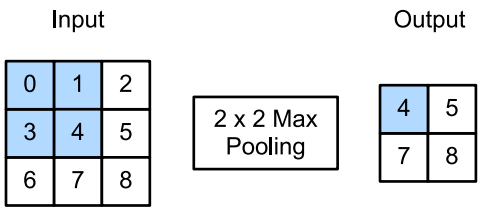


Fig. 7.5.1: Maximum pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

Tensor đầu ra trong Fig. 7.5.1 có chiều cao 2 và chiều rộng là 2. Bốn phần tử được bắt nguồn từ giá trị lớn

nhất trong mỗi cửa sổ pooling:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned} \tag{7.5.1}$$

Một lớp tổng hợp với hình dạng cửa sổ tổng hợp là $p \times q$ được gọi là một lớp tổng hợp $p \times q$. Các hoạt động pooling được gọi là $p \times q$ gộp.

Chúng ta hãy quay lại ví dụ phát hiện cạnh đối tượng được đề cập ở đầu phần này. Bây giờ chúng ta sẽ sử dụng đầu ra của lớp phức tạp làm đầu vào cho tổng hợp tối đa 2×2. Đặt đầu vào lớp phức tạp là `X` và đầu ra lớp pooling là `Y`. Các giá trị của `X[i, j]` và `X[i, j + 1]` có khác nhau hay không, hoặc `X[i, j + 1]` và `X[i, j + 2]` có khác nhau hay không, lớp tổng hợp luôn xuất ra `Y[i, j] = 1`. Có nghĩa là, sử dụng lớp tổng hợp tối đa 2×2, chúng ta vẫn có thể phát hiện xem mẫu được lớp phức tạp nhận ra không di chuyển không quá một phần tử về chiều cao hoặc chiều rộng.

Trong đoạn code dưới đây, chúng ta thực hiện sự lan truyền chuyển tiếp của lớp gộp trong hàm `pool2d`. Chức năng này tương tự như hàm `corr2d` trong Section 7.2. Tuy nhiên, ở đây chúng tôi không có hạt nhân, tính toán đầu ra là mức tối đa hoặc trung bình của từng vùng trong đầu vào.

```
import torch
from torch import nn
from d2l import torch as d2l
```

```
def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i: i + p_h, j: j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
    return Y
```

Chúng ta có thể xây dựng tensor đầu vào `X` trong Fig. 7.5.1 để xác nhận đầu ra của lớp tổng hợp tối đa hai chiều.

```
X = torch.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Ngoài ra, chúng tôi thử nghiệm với lớp tổng hợp trung bình.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

7.5.2 Padding và Stride

Như với các lớp phức tạp, các lớp gộp cũng có thể thay đổi hình dạng đầu ra. Và như trước đây, chúng ta có thể thay đổi hoạt động để đạt được hình dạng đầu ra mong muốn bằng cách đệm đầu vào và điều chỉnh sải chân. Chúng ta có thể chứng minh việc sử dụng đệm và bước tiến trong các lớp tổng hợp thông qua lớp tổng hợp tối đa hai chiều tích hợp từ khung học sâu. Đầu tiên chúng ta xây dựng một tensor đầu vào `X` có hình dạng có bốn chiều, trong đó số lượng ví dụ (kích thước lô) và số kênh đều là 1.

```
X = torch.arange(16, dtype=torch.float32).reshape((1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Theo mặc định, sải chân và cửa sổ tổng hợp trong ví dụ từ lớp tích hợp của framework có cùng hình dạng. Dưới đây, chúng ta sử dụng một cửa sổ tổng hợp hình dạng `(3, 3)`, vì vậy chúng ta có được một hình dạng sải chân của `(3, 3)` theo mặc định.

```
pool2d = nn.MaxPool2d(3)
pool2d(X)
```

```
tensor([[[[10.]]]])
```

Sải chân và đệm có thể được chỉ định thủ công.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Tất nhiên, chúng ta có thể chỉ định một cửa sổ tổng hợp hình chữ nhật tùy ý và chỉ định padding và sải chân cho chiều cao và chiều rộng, tương ứng.

```
pool2d = nn.MaxPool2d((2, 3), stride=(2, 3), padding=(0, 1))
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

7.5.3 Nhiều kênh

Khi xử lý dữ liệu đầu vào đa kênh, lớp tổng hợp nhóm mỗi kênh đầu vào riêng biệt, thay vì tổng hợp các đầu vào lên trên các kênh như trong một lớp phức tạp. Điều này có nghĩa là số kênh đầu ra cho lớp tổng hợp giống như số kênh đầu vào. Dưới đây, chúng tôi sẽ nối hàng chục `X` và `X + 1` trên kích thước kênh để xây dựng một đầu vào với 2 kênh.

```
X = torch.cat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

Như chúng ta có thể thấy, số lượng kênh đầu ra vẫn còn 2 sau khi tổng hợp.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

7.5.4 Tóm tắt

- Lấy các phần tử đầu vào trong cửa sổ tổng hợp, thao tác tổng hợp tối đa gán giá trị lớn nhất làm đầu ra và hoạt động tổng hợp trung bình gán giá trị trung bình làm đầu ra.
- Một trong những lợi ích chính của một lớp tổng hợp là làm giảm bớt độ nhạy quá mức của lớp phức tạp với vị trí.
- Chúng ta có thể chỉ định padding và sải chân cho layer pooling.
- Tổng hợp tối đa, kết hợp với một sải chân lớn hơn 1 có thể được sử dụng để giảm kích thước không gian (ví dụ, chiều rộng và chiều cao).
- Số kênh đầu ra của lớp tổng hợp giống như số kênh đầu vào.

7.5.5 Bài tập

1. Bạn có thể thực hiện tổng hợp trung bình như một trường hợp đặc biệt của một lớp phức tạp không? Nếu vậy, hãy làm điều đó.
2. Bạn có thể thực hiện pooling tối đa như một trường hợp đặc biệt của một lớp phức tạp? Nếu vậy, hãy làm điều đó.
3. Chi phí tính toán của lớp pooling là bao nhiêu? Giả sử rằng đầu vào của lớp pooling có kích thước $c \times h \times w$, cửa sổ pooling có hình dạng $p_h \times p_w$ với một padding là (p_h, p_w) và một sải chân là (s_h, s_w).
4. Tại sao bạn mong đợi tổng hợp tối đa và trung bình để làm việc khác nhau?
5. Chúng ta có cần một lớp tổng hợp tối thiểu riêng biệt không? Bạn có thể thay thế nó bằng một hoạt động khác không?
6. Có một hoạt động khác giữa tổng hợp trung bình và tối đa mà bạn có thể xem xét (gợi ý: nhớ lại softmax)? Tại sao nó có thể không được phổ biến như vậy?

Discussions[86]

7.6 Mạng thần kinh phức tạp (LeNet)

Bây giờ chúng tôi có tất cả các thành phần cần thiết để lắp ráp một CNN đầy đủ chức năng. Trong cuộc gặp gỡ trước đó của chúng tôi với dữ liệu hình ảnh, chúng tôi đã áp dụng mô hình hồi quy softmax (Section 4.6) và mô hình MLP (Section 5.2) cho hình ảnh quần áo trong bộ dữ liệu Fashion-MNIST. Để làm cho dữ liệu như vậy tuân theo hồi quy softmax và MLP s, trước tiên chúng ta làm phẳng mỗi hình ảnh từ một ma trận 28×28 thành một vector 784 chiều dài cố định, và sau đó xử lý chúng với các lớp được kết nối hoàn toàn. Bây giờ chúng ta có một tay cầm trên các lớp phức tạp, chúng ta có thể giữ lại cấu trúc không gian trong hình ảnh của chúng ta. Là một lợi ích bổ sung của việc thay thế các lớp được kết nối hoàn toàn bằng các lớp kết nối, chúng ta sẽ tận hưởng các mô hình phân tích hơn đòi hỏi ít tham số hơn nhiều.

Trong phần này, chúng tôi sẽ giới thiệu *LeNet*, trong số các CNN được xuất bản đầu tiên để thu hút sự chú ý rộng rãi về hiệu suất của nó trên các tác vụ thị giác máy tính. Mô hình được giới thiệu bởi (và đặt tên cho) Yann LeCun, sau đó là một nhà nghiên cứu tại AT&T Bell Labs, với mục đích nhận dạng chữ số viết tay trong hình ảnh (LeCun et al., 1998). Công trình này đại diện cho đỉnh cao của một thập kỷ nghiên cứu phát triển công nghệ. Năm 1989, LeCun công bố nghiên cứu đầu tiên để đào tạo thành công CNN thông qua truyền ngược.

Vào thời điểm đó LeNet đạt được kết quả xuất sắc phù hợp với hiệu suất của các máy vector hỗ trợ, sau đó là một cách tiếp cận thống trị trong việc học có giám sát. LeNet cuối cùng đã được điều chỉnh để nhận ra các chữ số để xử lý tiền gửi trong máy ATM. Cho đến ngày nay, một số máy ATM vẫn chạy mã mà Yann và đồng nghiệp Leon Bottou đã viết vào những năm 1990!

[86] https://discuss.d2l.ai/t/72

7.6.1 LeNet

Ở cấp độ cao, LeNet (LeNet-5) bao gồm hai phần: (i) một bộ mã hóa phức tạp bao gồm hai lớp phức tạp; và (ii) một khối dày đặc bao gồm ba lớp kết nối đầy đủ; Kiến trúc được tóm tắt trong Fig. 7.6.1.

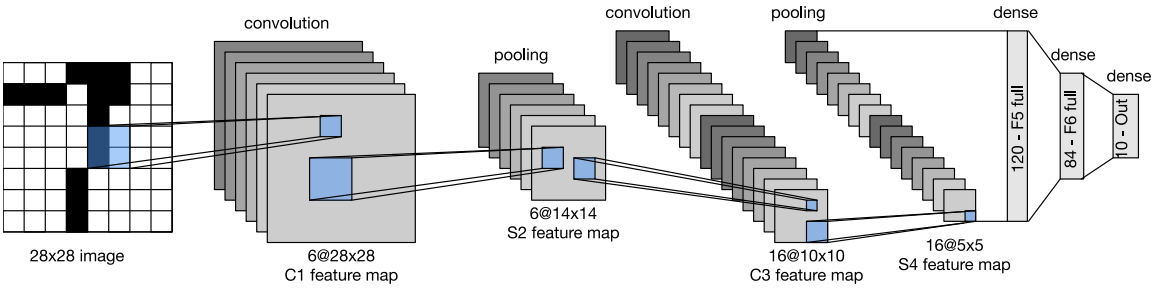


Fig. 7.6.1: Data flow in LeNet. The input is a handwritten digit, the output a probability over 10 possible outcomes.

Các đơn vị cơ bản trong mỗi khối phức tạp là một lớp phức tạp, một chức năng kích hoạt sigmoid, và một hoạt động tổng hợp trung bình tiếp theo. Lưu ý rằng trong khi Relus và max-pooling hoạt động tốt hơn, những khám phá này vẫn chưa được thực hiện trong những năm 1990. Mỗi lớp phức hợp sử dụng một hạt nhân 5×5 và một hàm kích hoạt sigmoid. Các lớp này ánh xạ các đầu vào sắp xếp không gian cho một số bản đồ tính năng hai chiều, thường làm tăng số lượng kênh. Lớp phức tạp đầu tiên có 6 kênh đầu ra, trong khi thứ hai có 16 kênh. Mỗi hoạt động tổng hợp 2×2 (sải chân 2) làm giảm kích thước bằng hệ số 4 thông qua lấy mẫu xuống không gian. Khối phức tạp phát ra một đầu ra với hình dạng được đưa ra bởi (kích thước lô, số kênh, chiều cao, chiều rộng).

Để truyền đầu ra từ khối phức tạp đến khối dày đặc, chúng ta phải làm phẳng từng ví dụ trong minibatch. Nói cách khác, chúng ta lấy đầu vào bốn chiều này và biến nó thành đầu vào hai chiều được mong đợi bởi các lớp được kết nối hoàn toàn: như một lời nhắc nhở, biểu diễn hai chiều mà chúng ta mong muốn sử dụng kích thước đầu tiên để lập chỉ mục các ví dụ trong minibatch và thứ hai để đưa ra biểu diễn vector phẳng of each mỗi example thí dụ. Khối dày đặc của LeNet có ba lớp kết nối hoàn toàn, tương ứng với 120, 84 và 10 đầu ra. Bởi vì chúng ta vẫn đang thực hiện phân loại, lớp đầu ra 10 chiều tương ứng với số lượng lớp đầu ra có thể.

Trong khi đi đến mức bạn thực sự hiểu những gì đang xảy ra bên trong LeNet có thể đã thực hiện một chút công việc, hy vọng đoạn mã sau đây sẽ thuyết phục bạn rằng việc thực hiện các mô hình như vậy với các khuôn khổ học sâu hiện đại rất đơn giản. Chúng ta chỉ cần khởi tạo một khối `Sequential` và chuỗi với nhau các lớp thích hợp.

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
```

(continues on next page)

```
    nn.Flatten(),
    nn.Linear(16 * 5 * 5, 120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Chúng tôi đã có một sự tự do nhỏ với mô hình ban đầu, loại bỏ kích hoạt Gaussian trong lớp cuối cùng. Ngoài ra, mạng này phù hợp với kiến trúc LeNet-5 ban đầu.

Bằng cách truyền một hình ảnh 28×28 một kênh (đen và trắng) qua mạng và in hình dạng đầu ra ở mỗi lớp, chúng ta có thể kiểm tra mô hình để đảm bảo rằng các hoạt động của nó phù hợp với những gì chúng ta mong đợi từ Fig. 7.6.2.

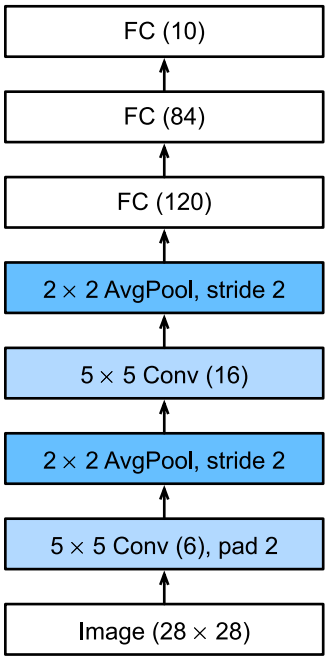


Fig. 7.6.2: Compressed notation for LeNet-5.

```
X = torch.rand(size=(1, 1, 28, 28), dtype=torch.float32)
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape: \t',X.shape)
```

```
Conv2d output shape:         torch.Size([1, 6, 28, 28])
Sigmoid output shape:        torch.Size([1, 6, 28, 28])
AvgPool2d output shape:      torch.Size([1, 6, 14, 14])
Conv2d output shape:         torch.Size([1, 16, 10, 10])
Sigmoid output shape:        torch.Size([1, 16, 10, 10])
AvgPool2d output shape:      torch.Size([1, 16, 5, 5])
Flatten output shape:        torch.Size([1, 400])
Linear output shape:         torch.Size([1, 120])
Sigmoid output shape:        torch.Size([1, 120])
Linear output shape:         torch.Size([1, 84])
Sigmoid output shape:        torch.Size([1, 84])
Linear output shape:         torch.Size([1, 10])
```

Lưu ý rằng chiều cao và chiều rộng của biểu diễn tại mỗi lớp trong suốt khối phức tạp bị giảm (so với lớp trước đó). Lớp kết hợp đầu tiên sử dụng 2 pixel đệm để bù đắp cho việc giảm chiều cao và chiều rộng mà nếu không sẽ là kết quả của việc sử dụng một hạt nhân 5×5. Ngược lại, lớp phức tạp thứ hai bỏ đệm, và do đó chiều cao và chiều rộng đều giảm 4 pixel. Khi chúng ta đi lên ngăn xếp các lớp, số lượng kênh tăng lớp trên lớp từ 1 trong đầu vào lên 6 sau lớp phức tạp đầu tiên và 16 sau lớp phức tạp thứ hai. Tuy nhiên, mỗi lớp tổng hợp giảm một nửa chiều cao và chiều rộng. Cuối cùng, mỗi lớp được kết nối hoàn toàn làm giảm chiều, cuối cùng phát ra một đầu ra có kích thước phù hợp với số lượng lớp.

7.6.2 Đào tạo

Bây giờ chúng tôi đã triển khai mô hình, chúng ta hãy chạy một thử nghiệm để xem LeNet giá vé trên Fashion-MNIST như thế nào.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

Trong khi CNN có ít tham số hơn, chúng vẫn có thể tốn kém hơn để tính toán so với MLP sâu tương tự vì mỗi tham số tham gia vào nhiều phép nhân hơn. Nếu bạn có quyền truy cập vào GPU, đây có thể là thời điểm tốt để đưa nó vào hành động để tăng tốc độ đào tạo.

Để đánh giá, chúng ta cần thực hiện một sửa đổi nhỏ đối với hàm `evaluate_accuracy` mà chúng tôi đã mô tả trong Section 4.6. Vì tập dữ liệu đầy đủ nằm trong bộ nhớ chính, chúng ta cần sao chép nó vào bộ nhớ GPU trước khi mô hình sử dụng GPU để tính toán với bộ dữ liệu.

```
def evaluate_accuracy_gpu(net, data_iter, device=None): #@save
    """Compute the accuracy for a model on a dataset using a GPU."""
    if isinstance(net, nn.Module):
        net.eval()  # Set the model to evaluation mode
        if not device:
            device = next(iter(net.parameters())).device
    # No. of correct predictions, no. of predictions
    metric = d2l.Accumulator(2)

    with torch.no_grad():
        for X, y in data_iter:
            if isinstance(X, list):
                # Required for BERT Fine-tuning (to be covered later)
                X = [x.to(device) for x in X]
            else:
                X = X.to(device)
            y = y.to(device)
            metric.add(d2l.accuracy(net(X), y), y.numel())
    return metric[0] / metric[1]
```

Chúng tôi cũng cần cập nhật chức năng đào tạo của chúng tôi để đối phó với GPU. Không giống như `train_epoch_ch3` được xác định trong Section 4.6, bây giờ chúng ta cần di chuyển từng minibatch dữ liệu đến thiết bị được chỉ định của chúng tôi (hy vọng là GPU) trước khi thực hiện tuyên truyền chuyển tiếp và lùi.

Chức năng huấn luyện `train_ch6` cũng tương tự như `train_ch3` được định nghĩa trong Section 4.6. Vì chúng tôi sẽ triển khai các mạng với nhiều lớp trong tương lai, chúng tôi sẽ chủ yếu dựa vào các API cấp cao. Hàm đào tạo sau đây giả định một mô hình được tạo từ API cấp cao làm đầu vào và được tối ưu hóa cho phù hợp. Chúng tôi khởi tạo các tham số mô hình trên thiết bị được chỉ định bởi đối số `device`, sử dụng khởi

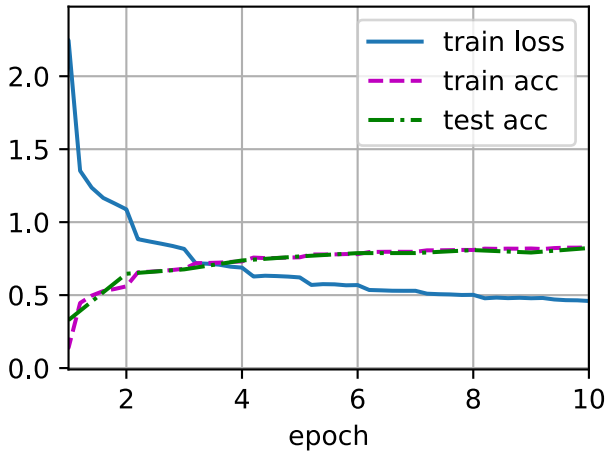
tạo Xavier như được giới thiệu trong Section 5.8.2. Cũng giống như với MLP, chức năng mất mát của chúng tôi là cross-entropy, và chúng tôi giảm thiểu nó thông qua minibatch stochastic gradient descent. Vì mỗi kỷ nguyên mất hàng chục giây để chạy, chúng tôi hình dung sự mất mát đào tạo thường xuyên hơn.

```
#@save
def train_ch6(net, train_iter, test_iter, num_epochs, lr, device):
    """Train a model with a GPU (defined in Chapter 6)."""
    def init_weights(m):
        if type(m) == nn.Linear or type(m) == nn.Conv2d:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

Bây giờ chúng ta hãy đào tạo và đánh giá mô hình LeNet-5.

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.460, train acc 0.827, test acc 0.823
82186.3 examples/sec on cuda:0
```



7.6.3 Tóm tắt

- CNN là một mạng sử dụng các lớp phức tạp.
- Trong một CNN, chúng tôi xen kẽ các phức tạp, phi tuyến tính, và (thường) các hoạt động tập hợp.
- Trong một CNN, các lớp phức hợp thường được sắp xếp sao cho chúng giảm dần độ phân giải không gian của các biểu diễn, đồng thời tăng số lượng kênh.
- Trong CNN truyền thống, các biểu diễn được mã hóa bởi các khối phức tạp được xử lý bởi một hoặc nhiều lớp kết nối đầy đủ trước khi phát ra đầu ra.
- LeNet được cho là triển khai thành công đầu tiên của một mạng như vậy.

7.6.4 Bài tập

1. Thay thế các pooling trung bình với tổng hợp tối đa. Điều gì xảy ra?
2. Cố gắng xây dựng một mạng phức tạp hơn dựa trên LeNet để cải thiện độ chính xác của nó.
    1. Điều chỉnh kích thước cửa sổ covolution.
    2. Điều chỉnh số lượng kênh đầu ra.
    3. Điều chỉnh chức năng kích hoạt (ví dụ: ReLU).
    4. Điều chỉnh số lượng lớp covolution.
    5. Điều chỉnh số lượng lớp được kết nối hoàn toàn.
    6. Điều chỉnh tỷ lệ học tập và các chi tiết đào tạo khác (ví dụ: khởi tạo và số kỷ nguyên.)
3. Hãy thử mạng cải tiến trên tập dữ liệu MNIST ban đầu.
4. Hiển thị các kích hoạt của lớp đầu tiên và thứ hai của LeNet cho các đầu vào khác nhau (ví dụ, áo len và áo khoác).

Discussions[87]

[87] https://discuss.d2l.ai/t/74

8 | Mạng thần kinh phức tạp hiện đại

Bây giờ chúng tôi hiểu những điều cơ bản của hệ thống dây điện với nhau CNN, chúng tôi sẽ đưa bạn qua một chuyến tham quan các kiến trúc CNN hiện đại. Trong chương này, mỗi phần tương ứng với một kiến trúc CNN quan trọng đó là tại một số điểm (hoặc hiện tại) mô hình cơ sở mà trên đó nhiều dự án nghiên cứu và hệ thống triển khai đã được xây dựng. Mỗi mạng này là một kiến trúc thống trị ngắn gọn và nhiều người là người chiến thắng hoặc á quân trong cuộc thi ImageNet, đã phục vụ như một phong vũ biểu tiến bộ về học tập được giám sát trong tầm nhìn máy tính kể từ năm 2010.

Các mô hình này bao gồm AlexNet, mạng quy mô lớn đầu tiên được triển khai để đánh bại các phương pháp tầm nhìn máy tính thông thường trên một thách thức tầm nhìn quy mô lớn; mạng VGG, sử dụng một số khối lặp lại của các yếu tố; mạng trong mạng (Nin) mà xoay quanh toàn bộ mạng thần kinh patch-wise qua đầu vào; GoogLeNet, sử dụng mạng có kết nối song song; mạng dư (ResNet), vẫn là kiến trúc ngoài kệ phổ biến nhất trong tầm nhìn máy tính; và các mạng kết nối mật độ (DenseNet), đắt tiền để tính toán nhưng đã đặt ra một số điểm chuẩn gần đây.

Mặc dù ý tưởng về mạng nơ-ron * deep* khá đơn giản (xếp chồng lên nhau một loạt các lớp), hiệu suất có thể thay đổi rất nhiều trên các kiến trúc và các lựa chọn siêu tham số. Các mạng thần kinh được mô tả trong chương này là sản phẩm của trực giác, một vài hiểu biết toán học, và rất nhiều thử nghiệm và sai. Chúng tôi trình bày các mô hình này theo thứ tự thời gian, một phần để truyền đạt ý thức về lịch sử để bạn có thể hình thành trực giác của riêng bạn về nơi lĩnh vực đang hướng đến và có lẽ phát triển kiến trúc của riêng bạn. Ví dụ, bình thường hóa hàng loạt và các kết nối còn lại được mô tả trong chương này đã đưa ra hai ý tưởng phổ biến để đào tạo và thiết kế các mô hình sâu.

8.1 Mạng thần kinh phức tạp sâu (AlexNet)

Mặc dù CNN nổi tiếng trong thị giác máy tính và cộng đồng học máy sau khi giới thiệu LeNet, họ đã không thống trị ngay lập tức lĩnh vực này. Mặc dù LeNet đạt được kết quả tốt trên các tập dữ liệu nhỏ ban đầu, nhưng hiệu suất và tính khả thi của việc đào tạo CNN trên các tập dữ liệu lớn hơn, thực tế hơn vẫn chưa được thiết lập. Trên thực tế, trong phần lớn thời gian can thiệp giữa đầu thập niên 1990 và kết quả đầu nguồn của năm 2012, các mạng thần kinh thường bị vượt qua bởi các phương pháp học máy khác, chẳng hạn như các máy vector hỗ trợ.

Đối với tầm nhìn máy tính, so sánh này có lẽ không công bằng. Đó là mặc dù các đầu vào cho các mạng tích hợp bao gồm các giá trị pixel thô hoặc được xử lý nhẹ (ví dụ: bằng cách định tâm), các học viên sẽ không bao giờ cung cấp các pixel thô vào các mô hình truyền thống. Thay vào đó, đường ống tầm nhìn máy tính điển hình bao gồm các đường ống trích xuất tính năng kỹ thuật thủ công. Thay vì * tìm hiểu các tính năng*, các tính năng là * crafted*. Hầu hết các tiến bộ đến từ việc có nhiều ý tưởng thông minh hơn cho các tính năng, và thuật toán học tập thường được chuyển xuống một suy nghĩ sau.

Mặc dù một số máy gia tốc mạng thần kinh đã có sẵn trong những năm 1990, nhưng chúng chưa đủ mạnh để tạo ra các CNN đa kênh sâu, đa lớp với một số lượng lớn các tham số. Hơn nữa, các bộ dữ liệu vẫn còn tương

đối nhỏ. Thêm vào những trở ngại này, các thủ thuật chính để đào tạo các mạng thần kinh bao gồm heuristics khởi tạo tham số, các biến thể thông minh của gốc gradient ngẫu nhiên, chức năng kích hoạt không squashing, và các kỹ thuật chính quy hóa hiệu quả vẫn còn thiếu.

Do đó, thay vì đào tạo hệ thống * end-to-end* (pixel để phân loại), các đường ống cổ điển trông như thế này hơn:

1. Lấy một bộ dữ liệu thú vị. Trong những ngày đầu, các bộ dữ liệu này yêu cầu các cảm biến đắt tiền (vào thời điểm đó, hình ảnh 1 megapixel là hiện đại).
2. Xử lý trước bộ dữ liệu với các tính năng thủ công dựa trên một số kiến thức về quang học, hình học, các công cụ phân tích khác và thỉnh thoảng trên những khám phá ngẫu nhiên của sinh viên tốt nghiệp may mắn.
3. Cung cấp dữ liệu thông qua một bộ trích xuất tính năng tiêu chuẩn như SIFT (biến đổi tính năng bất biến quy mô) (Lowe, 2004), SURF (tăng tốc các tính năng mạnh mẽ) (Bay et al., 2006) hoặc bất kỳ số lượng đường ống điều chỉnh bằng tay nào khác.
4. Dump các biểu diễn kết quả vào phân loại yêu thích của bạn, có thể là một mô hình tuyến tính hoặc phương pháp hạt nhân, để đào tạo một phân loại.

Nếu bạn nói chuyện với các nhà nghiên cứu machine learning, họ tin rằng machine learning vừa quan trọng vừa đẹp. Các lý thuyết thanh lịch đã chứng minh các tính chất của các nhà phân loại khác nhau. Lĩnh vực máy học rất phát triển mạnh, nghiêm ngặt và hữu ích. Tuy nhiên, nếu bạn nói chuyện với một nhà nghiên cứu thị giác máy tính, bạn sẽ nghe một câu chuyện rất khác. Sự thật bẩn thỉu của nhận dạng hình ảnh, họ sẽ nói với bạn, đó là các tính năng, không phải học thuật toán, đã thúc đẩy tiến bộ. Các nhà nghiên cứu thị giác máy tính tin một cách hợp lý rằng một bộ dữ liệu lớn hơn hoặc sạch hơn một chút hoặc một đường ống trích xuất tính năng được cải thiện một chút quan trọng hơn nhiều so với độ chính xác cuối cùng so với bất kỳ thuật toán học tập nào.

8.1.1 Đại diện học tập

Một cách khác để đúc tình trạng của các vấn đề là phần quan trọng nhất của đường ống là đại diện. Và cho đến năm 2012, đại diện đã được tính toán bằng máy móc. Trong thực tế, kỹ thuật một bộ mới của chức năng tính năng, cải thiện kết quả, và viết lên phương pháp là một thể loại giấy nổi bật. SIFT (Lowe, 2004), SURF (Bay et al., 2006), HOG (biểu đồ của gradient định hướng) (Dalal & Triggs, 2005), bags of visual words[88] và các máy chiết xuất tính năng tương tự cai trị roost.

Một nhóm các nhà nghiên cứu khác, bao gồm Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, và Juergen Schmidhuber, có những kế hoạch khác nhau. Họ tin rằng bản thân các tính năng nên được học. Hơn nữa, họ tin rằng để phức tạp một cách hợp lý, các tính năng phải được cấu tạo theo thứ bậc với nhiều lớp học chung, mỗi lớp có các tham số có thể học được. Trong trường hợp của một hình ảnh, các lớp thấp nhất có thể đến để phát hiện các cạnh, màu sắc và kết cấu. Thật vậy, Alex Krizhevsky, Ilya Sutkever và Geoff Hinton đã đề xuất một biến thể mới của CNN, *AlexNet*, đạt được hiệu suất xuất sắc trong thử thách ImageNet 2012. AlexNet được đặt theo tên Alex Krizhevsky, tác giả đầu tiên của bài phân loại ImageNet đột phá (Krizhevsky et al., 2012).

Điều thú vị là ở các lớp thấp nhất của mạng, mô hình đã học được tính năng trích xuất giống như một số bộ lọc truyền thống. Fig. 8.1.1 được sao chép từ giấy AlexNet (Krizhevsky et al., 2012) và mô tả mô tả hình ảnh cấp thấp hơn.

[88] https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision

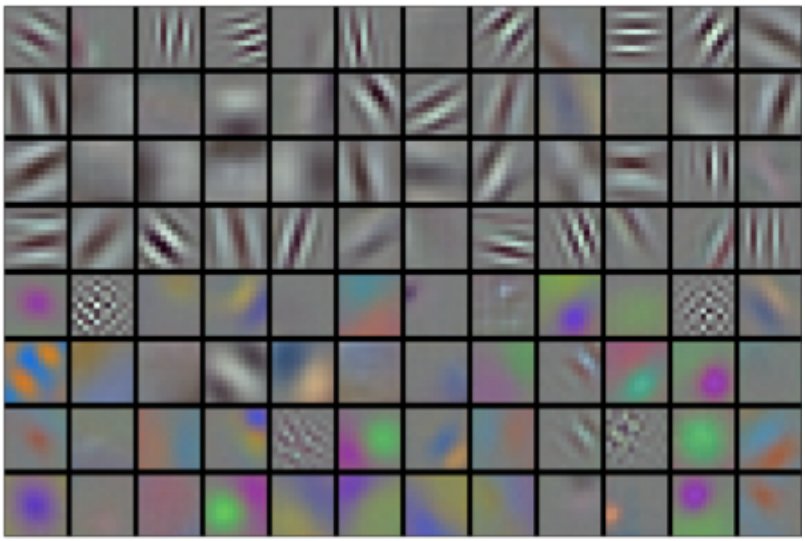


Fig. 8.1.1: Image filters learned by the first layer of AlexNet.

Các lớp cao hơn trong mạng có thể xây dựng dựa trên các biểu diễn này để đại diện cho các cấu trúc lớn hơn, như mắt, mũi, lưỡi cỏ, v.v. Thậm chí các lớp cao hơn có thể đại diện cho toàn bộ các đối tượng như người, máy bay, chó, hoặc frisbees. Cuối cùng, trạng thái ẩn cuối cùng học được một biểu diễn nhỏ gọn của hình ảnh tóm tắt nội dung của nó sao cho dữ liệu thuộc các danh mục khác nhau có thể dễ dàng tách ra.

Trong khi bước đột phá cuối cùng cho các CNN nhiều lớp đến vào năm 2012, một nhóm các nhà nghiên cứu cốt lõi đã cống hiến hết mình cho ý tưởng này, cố gắng tìm hiểu các đại diện phân cấp của dữ liệu trực quan trong nhiều năm. Bước đột phá cuối cùng vào năm 2012 có thể được quy cho hai yếu tố chính.

Thiếu thành phần: Dữ liệu

Các mô hình sâu với nhiều lớp đòi hỏi một lượng lớn dữ liệu để vào chế độ mà chúng vượt trội hơn đáng kể các phương pháp truyền thống dựa trên tối ưu hóa lồi (ví dụ, phương pháp tuyến tính và hạt nhân). Tuy nhiên, với khả năng lưu trữ hạn chế của máy tính, chi phí tương đối của các cảm biến, và ngân sách nghiên cứu tương đối chặt chẽ hơn trong những năm 1990, hầu hết các nghiên cứu đều dựa vào các tập dữ liệu nhỏ. Nhiều giấy tờ đề cập đến bộ sưu tập dữ liệu UCI, nhiều trong số đó chỉ chứa hàng trăm hoặc (một vài) hàng ngàn hình ảnh được chụp trong các cài đặt không tự nhiên với độ phân giải thấp.

Năm 2009, tập dữ liệu ImageNet được phát hành, thách thức các nhà nghiên cứu tìm hiểu các mô hình từ 1 triệu ví dụ, 1000 mỗi loại từ 1000 loại đối tượng riêng biệt. Các nhà nghiên cứu, dẫn đầu bởi Fei-Fei Li, người đã giới thiệu tập dữ liệu này đã tận dụng Google Image Search để lọc trước các bộ ứng cử viên lớn cho từng danh mục và sử dụng đường ống cộng đồng Amazon Mechanical Turk để xác nhận cho mỗi hình ảnh xem nó có thuộc về danh mục liên quan hay không. Quy mô này là chưa từng có. Cuộc cạnh tranh liên quan, được mệnh danh là ImageNet Challenge đã đẩy tầm nhìn máy tính và nghiên cứu máy học tiến lên phía trước, thách thức các nhà nghiên cứu để xác định mô hình nào hoạt động tốt nhất ở quy mô lớn hơn so với các học giả trước đây đã xem xét.

Thiếu thành phần: Phần cứng

Mô hình học sâu là những người tiêu dùng phàm ăn của chu kỳ tính toán. Đào tạo có thể mất hàng trăm kỷ nguyên, và mỗi lần lặp lại yêu cầu truyền dữ liệu qua nhiều lớp hoạt động đại số tuyến tính đắt tiền tính toán. Đây là một trong những lý do chính khiến vào những năm 1990 và đầu những năm 2000, các thuật toán đơn giản dựa trên các mục tiêu lồi được tối ưu hóa hiệu quả hơn được ưu tiên.

Các đơn vị xử lý đồ học (GPU) được chứng minh là một người thay đổi trò chơi in making chế tạo deep sâu learning học tập feasible khả thi. Những chip này từ lâu đã được phát triển để tăng tốc xử lý đồ họa để mang lại lợi ích cho các trò chơi máy tính. Đặc biệt, chúng đã được tối ưu hóa cho các sản phẩm ma thuật-vector 4×4 thông lượng cao, cần thiết cho nhiều tác vụ đồ họa máy tính. May mắn thay, toán học này rất giống với điều cần thiết để tính toán các lớp phức tạp. Khoảng thời gian đó, NVIDIA và ATI đã bắt đầu tối ưu hóa GPU cho các hoạt động điện toán chung, đi xa đến mức tiếp thị chúng như * GPU đa năng (GPGPU).

Để cung cấp một số trực giác, hãy xem xét các lõi của bộ vi xử lý hiện đại (CPU). Mỗi lõi là khá mạnh mẽ chạy ở tần số đồng hồ cao và thể thao bộ nhớ cache lớn (lên đến vài megabyte L3). Mỗi lõi rất phù hợp để thực hiện một loạt các hướng dẫn, với các dự báo nhánh, một đường ống sâu, và các chuông và còi khác cho phép nó chạy một loạt các chương trình. Tuy nhiên, sức mạnh rõ ràng này cũng là gót chân Achilles của nó: lõi mục đích chung rất tốn kém để xây dựng. Chúng yêu cầu nhiều khu vực chip, cấu trúc hỗ trợ tinh vi (giao diện bộ nhớ, logic bộ nhớ đệm giữa các lõi, kết nối tốc độ cao, v.v.) và chúng tương đối xấu ở bất kỳ tác vụ duy nhất nào. Máy tính xách tay hiện đại có tới 4 lõi và thậm chí các máy chủ cao cấp hiếm khi vượt quá 64 lõi, đơn giản vì nó không hiệu quả về chi phí.

Bằng cách so sánh, GPU bao gồm $100 \sim 1000$ các yếu tố xử lý nhỏ (các chi tiết khác nhau phần nào giữa NVIDIA, ATI, ARM và các nhà cung cấp chip khác), thường được nhóm thành các nhóm lớn hơn (NVIDIA gọi chúng là cong vênh). Trong khi mỗi lõi tương đối yếu, đôi khi thậm chí chạy ở tần số xung nhịp dưới 1GHz, đó là tổng số lõi như vậy làm cho các đơn đặt hàng GPU có độ lớn nhanh hơn CPU. Ví dụ, thế hệ Volta gần đây của NVIDIA cung cấp lên đến 120 TFlops mỗi chip cho các hướng dẫn chuyên ngành (và lên đến 24 TFlops cho các mục đích chung hơn), trong khi hiệu suất điểm nổi của CPU không vượt quá 1 TFlop cho đến nay. Lý do tại sao điều này là có thể thực sự khá đơn giản: đầu tiên, tiêu thụ điện năng có xu hướng tăng * quadratically* với tần số xung nhịp. Do đó, đối với ngân sách năng lượng của lõi CPU chạy nhanh hơn 4 lần (một số điển hình), bạn có thể sử dụng 16 lõi GPU ở tốc độ $1/4$, mang lại hiệu suất $16 \times 1/4 = 4$ lần. Hơn nữa, các lõi GPU đơn giản hơn nhiều (trên thực tế, trong một thời gian dài, chúng thậm chí không thể * để thực thi mã mục đích chung), điều này làm cho chúng tiết kiệm năng lượng hơn. Cuối cùng, nhiều hoạt động trong deep learning yêu cầu băng thông bộ nhớ cao. Một lần nữa, GPU tỏa sáng ở đây với các xe buýt có độ rộng ít nhất 10 lần so với nhiều CPU.

Quay lại năm 2012. Một bước đột phá lớn đến khi Alex Krizhevsky và Ilya Sutkever triển khai một CNN sâu có thể chạy trên phần cứng GPU. Họ nhận ra rằng các tắc nghẽn tính toán trong CNN s, sự phức tạp và nhân ma trận, là tất cả các hoạt động có thể được song song trong phần cứng. Sử dụng hai NVIDIA GTX 580s với 3GB bộ nhớ, họ đã thực hiện các kết hợp nhanh. Mã cuda-convnet[89] đủ tốt để trong nhiều năm, đó là tiêu chuẩn ngành và cung cấp cho vài năm đầu tiên của sự bùng nổ học tập sâu.

[89] https://code.google.com/archive/p/cuda-convnet/

8.1.2 AlexNet

AlexNet, sử dụng CNN 8 lớp, đã giành chiến thắng trong ImageNet Large Scale Visual Recognition Challenge 2012 bởi một biên độ lớn phi thường. Mạng này lần đầu tiên cho thấy rằng các tính năng thu được bằng cách học tập có thể vượt qua các tính năng được thiết kế thủ công, phá vỡ mô hình trước đó trong tầm nhìn máy tính.

Các kiến trúc của AlexNet và LeNet rất giống nhau, như Fig. 8.1.2 minh họa. Lưu ý rằng chúng tôi cung cấp một phiên bản hơi hợp lý của AlexNet loại bỏ một số điều kỳ lạ thiết kế cần thiết trong năm 2012 để làm cho mô hình phù hợp với hai GPU nhỏ.

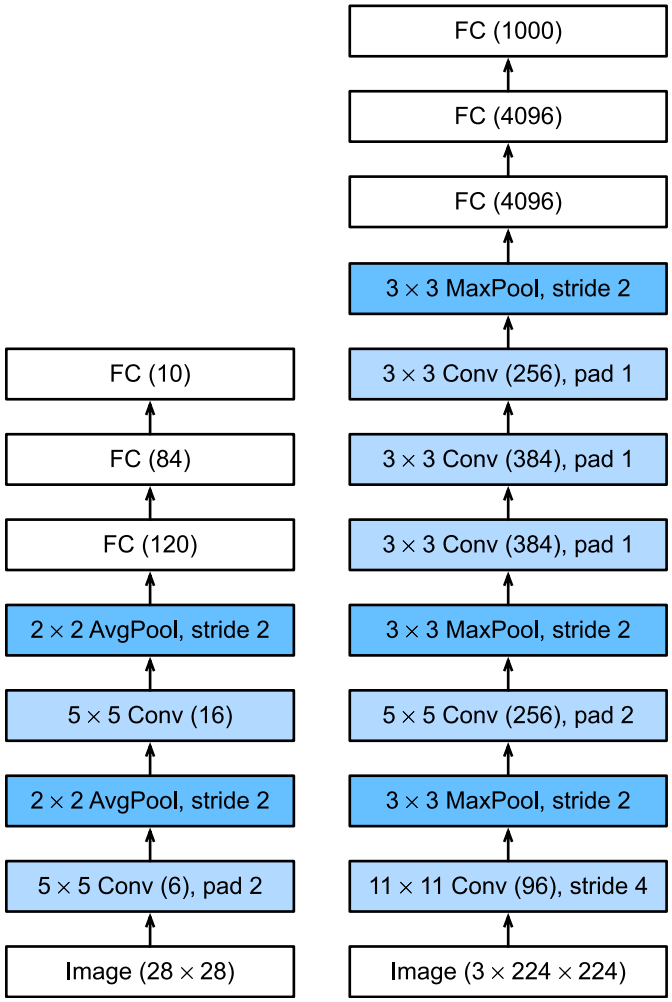


Fig. 8.1.2: From LeNet (left) to AlexNet (right).

Các triết lý thiết kế của AlexNet và LeNet rất giống nhau, nhưng cũng có sự khác biệt đáng kể. Đầu tiên, AlexNet sâu hơn nhiều so với LeNet5 tương đối nhỏ. AlexNet bao gồm tám lớp: năm lớp phức tạp, hai lớp ẩn được kết nối hoàn toàn và một lớp đầu ra được kết nối hoàn toàn. Thứ hai, AlexNet sử dụng ReLU thay vì sigmoid làm chức năng kích hoạt của nó. Hãy để chúng tôi đi sâu vào các chi tiết dưới đây.

Architecture

Trong lớp đầu tiên của AlexNet, hình dạng cửa sổ phức tạp là 11×11. Vì hầu hết các hình ảnh trong ImageNet cao hơn mười lần và rộng hơn hình ảnh MNIST, các đối tượng trong dữ liệu ImageNet có xu hướng chiếm nhiều pixel hơn. Do đó, một cửa sổ phức tạp lớn hơn là cần thiết để chụp đối tượng. Hình dạng cửa sổ phức tạp trong lớp thứ hai được giảm xuống còn 5×5, tiếp theo là 3×3. Ngoài ra, sau các lớp phức tạp thứ nhất, thứ hai và thứ năm, mạng thêm các lớp tổng hợp tối đa với hình dạng cửa sổ là 3×3 và một sải chân là 2. Hơn nữa, AlexNet có kênh phức tạp gấp mười lần so với LeNet.

Sau lớp phức tạp cuối cùng có hai lớp kết nối hoàn toàn với 4096 đầu ra. Hai lớp được kết nối hoàn toàn khổng lồ này tạo ra các thông số mô hình gần 1 GB. Do bộ nhớ hạn chế trong các GPU đầu tiên, AlexNet ban đầu đã sử dụng một thiết kế luồng dữ liệu kép, để mỗi trong hai GPU của họ có thể chịu trách nhiệm lưu trữ và tính toán chỉ một nửa mô hình của nó. May mắn thay, bộ nhớ GPU hiện nay tương đối phong phú, vì vậy chúng ta hiếm khi cần chia tay các mô hình trên GPU trong những ngày này (phiên bản của mô hình AlexNet của chúng tôi lệch khỏi giấy gốc ở khía cạnh này).

Chức năng kích hoạt

Bên cạnh đó, AlexNet đã thay đổi chức năng kích hoạt sigmoid thành chức năng kích hoạt ReLU đơn giản hơn. Một mặt, việc tính toán chức năng kích hoạt ReLU đơn giản hơn. Ví dụ, nó không có hoạt động số mũ được tìm thấy trong chức năng kích hoạt sigmoid. Mặt khác, chức năng kích hoạt ReLU giúp đào tạo mô hình dễ dàng hơn khi sử dụng các phương pháp khởi tạo tham số khác nhau. Điều này là do, khi đầu ra của chức năng kích hoạt sigmoid rất gần 0 hoặc 1, gradient của các vùng này gần như là 0, do đó sự lan truyền ngược không thể tiếp tục cập nhật một số tham số mô hình. Ngược lại, gradient của hàm kích hoạt ReLU trong khoảng dương luôn là 1. Do đó, nếu các tham số mô hình không được khởi tạo đúng cách, hàm sigmoid có thể thu được một gradient gần 0 trong khoảng dương, do đó mô hình không thể được đào tạo hiệu quả.

Kiểm soát công suất và tiền xử lý

AlexNet kiểm soát độ phức tạp mô hình của lớp kết nối hoàn toàn bằng cách bỏ học (Section 5.6), trong khi LeNet chỉ sử dụng trọng lượng phân rã. Để tăng cường dữ liệu hơn nữa, vòng đào tạo của AlexNet đã thêm rất nhiều nâng hình ảnh, chẳng hạn như lật, cắt và thay đổi màu sắc. Điều này làm cho mô hình mạnh mẽ hơn và kích thước mẫu lớn hơn làm giảm hiệu quả overfitting. Chúng tôi sẽ thảo luận về việc tăng dữ liệu chi tiết hơn trong Section 14.1.

```
import torch
from torch import nn
from d2l import torch as d2l

net = nn.Sequential(
    # Here, we use a larger 11 x 11 window to capture objects. At the same
    # time, we use a stride of 4 to greatly reduce the height and width of the
    # output. Here, the number of output channels is much larger than that in
    # LeNet
    nn.Conv2d(1, 96, kernel_size=11, stride=4, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    # Make the convolution window smaller, set padding to 2 for consistent
    # height and width across the input and output, and increase the number of
    # output channels
    nn.Conv2d(96, 256, kernel_size=5, padding=2), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
```

(continues on next page)

(continued from previous page)

```
    # Use three successive convolutional layers and a smaller convolution
    # window. Except for the final convolutional layer, the number of output
    # channels is further increased. Pooling layers are not used to reduce the
    # height and width of input after the first two convolutional layers
    nn.Conv2d(256, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 384, kernel_size=3, padding=1), nn.ReLU(),
    nn.Conv2d(384, 256, kernel_size=3, padding=1), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2),
    nn.Flatten(),
    # Here, the number of outputs of the fully-connected layer is several
    # times larger than that in LeNet. Use the dropout layer to mitigate
    # overfitting
    nn.Linear(6400, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    nn.Linear(4096, 4096), nn.ReLU(),
    nn.Dropout(p=0.5),
    # Output layer. Since we are using Fashion-MNIST, the number of classes is
    # 10, instead of 1000 as in the paper
    nn.Linear(4096, 10))
```

Chúng tôi xây dựng một ví dụ dữ liệu một kênh với cả chiều cao và chiều rộng là 224 để quan sát hình dạng đầu ra của mỗi lớp. Nó phù hợp với kiến trúc AlexNet vào năm Fig. 8.1.2.

```
X = torch.randn(1, 1, 224, 224)
for layer in net:
    X=layer(X)
    print(layer.__class__.__name__,'output shape:\t',X.shape)
```

```
Conv2d output shape:         torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Conv2d output shape:         torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:         torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:      torch.Size([1, 256, 5, 5])
Flatten output shape:        torch.Size([1, 6400])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

8.1.3 Đọc tập dữ liệu

Mặc dù AlexNet được đào tạo trên ImageNet trong giấy, chúng tôi sử dụng Fashion-MNIST ở đây vì đào tạo mô hình ImageNet để hội tụ có thể mất hàng giờ hoặc vài ngày ngay cả trên GPU hiện đại. Một trong những vấn đề với việc áp dụng AlexNet trực tiếp trên Fashion-MNIST là nó hình ảnh có độ phân giải thấp hơn (28×28 pixel) hơn ImageNet images. Để làm cho mọi thứ hoạt động, chúng tôi upsample chúng lên 224×224 (nói chung không phải là một thực hành thông minh, nhưng chúng tôi làm điều đó ở đây để trung thành với AlexNet kiến trúc). Chúng tôi thực hiện thay đổi kích thước này với đối số `resize` trong hàm `d2l.load_data_fashion_mnist`.

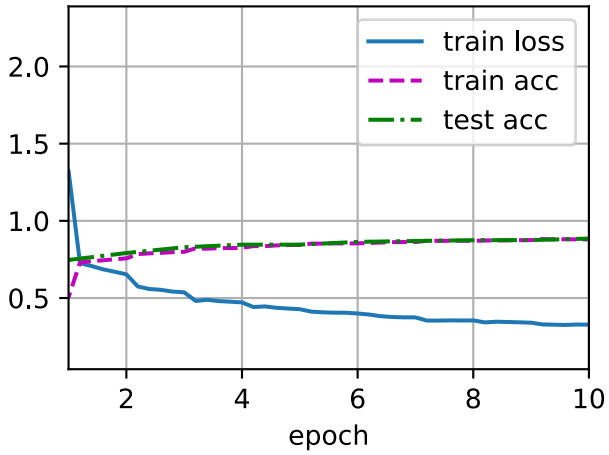
```
batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
```

8.1.4 Đào tạo

Bây giờ, chúng ta có thể bắt đầu đào tạo AlexNet. So với LeNet năm Section 7.6, thay đổi chính ở đây là việc sử dụng tốc độ học tập nhỏ hơn và đào tạo chậm hơn nhiều do mạng sâu hơn và rộng hơn, độ phân giải hình ảnh cao hơn và các phức tạp tốn kém hơn.

```
lr, num_epochs = 0.01, 10
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.328, train acc 0.880, test acc 0.885
4047.8 examples/sec on cuda:0
```



8.1.5 Tóm tắt

- AlexNet có cấu trúc tương tự như của LeNet, nhưng sử dụng nhiều lớp phức tạp hơn và một không gian tham số lớn hơn để phù hợp với tập dữ liệu ImageNet quy mô lớn.
- Ngày nay AlexNet đã bị vượt qua bởi các kiến trúc hiệu quả hơn nhiều nhưng đó là một bước quan trọng từ nông đến mạng sâu được sử dụng ngày nay.
- Mặc dù có vẻ như chỉ có một vài dòng nữa trong việc thực hiện AlexNet so với LeNet, cộng đồng học thuật phải mất nhiều năm để nắm lấy sự thay đổi khái niệm này và tận dụng kết quả thử nghiệm tuyệt

vời của nó. Điều này cũng là do thiếu các công cụ tính toán hiệu quả.

- Bỏ học, ReLU và tiền xử lý là những bước quan trọng khác trong việc đạt được hiệu suất tuyệt vời trong các tác vụ thị giác máy tính.

8.1.6 Bài tập

1. Hãy thử tăng số lượng thời đại. So với LeNet, kết quả khác nhau như thế nào? Tại sao?
2. AlexNet có thể quá phức tạp đối với tập dữ liệu Fashion-MNIST.
    1. Hãy thử đơn giản hóa mô hình để đào tạo nhanh hơn, đồng thời đảm bảo rằng độ chính xác không giảm đáng kể.
    2. Thiết kế một mô hình tốt hơn hoạt động trực tiếp trên 28×28 hình ảnh.
3. Sửa đổi kích thước lô, và quan sát những thay đổi về độ chính xác và bộ nhớ GPU.
4. Phân tích hiệu suất tính toán của AlexNet.
    1. Phần chiếm ưu thế cho dấu chân bộ nhớ của AlexNet là gì?
    2. Phần chi phối cho tính toán trong AlexNet là gì?
    3. Làm thế nào về băng thông bộ nhớ khi tính toán kết quả?
5. Áp dụng dropout và ReLU cho LeNet-5. Nó có cải thiện không? Làm thế nào về tiền xử lý?

Discussions[90]

8.2 Mạng sử dụng khối (VGG)

Trong khi AlexNet đưa ra bằng chứng thực nghiệm rằng CNN sâu có thể đạt được kết quả tốt, nó không cung cấp một mẫu chung để hướng dẫn các nhà nghiên cứu tiếp theo trong việc thiết kế các mạng mới. Trong các phần sau, chúng tôi sẽ giới thiệu một số khái niệm heuristic thường được sử dụng để thiết kế các mạng sâu.

Tiến bộ trong lĩnh vực này phản ánh rằng trong thiết kế chip nơi các kỹ sư đã đi từ đặt bóng bán dẫn đến các yếu tố logic đến các khối logic. Tương tự, thiết kế của các kiến trúc mạng thần kinh đã phát triển dần dần trừu tượng hơn, với các nhà nghiên cứu chuyển từ suy nghĩ về các tế bào thần kinh riêng lẻ sang toàn bộ các lớp, và bây giờ đến các khối, lặp lại các mô hình của các lớp.

Ý tưởng sử dụng các khối đầu tiên xuất hiện từ Visual Geometry Group[91] (VGG) tại Đại học Oxford, trong mạng *VGG* eponymously-name của họ. Thật dễ dàng để thực hiện các cấu trúc lặp đi lặp lại này trong code với bất kỳ khuôn khổ học sâu hiện đại nào bằng cách sử dụng các vòng lặp và chương trình con.

[90] https://discuss.d2l.ai/t/76
[91] http://www.robots.ox.ac.uk/~vgg/

8.2.1 VGG Blocks

Khối xây dựng cơ bản của CNN cổ điển là một chuỗi các như sau: (i) một lớp tích hợp với đệm để duy trì độ phân giải, (ii) một phi tuyến tính như một ReLU, (iii) một lớp tổng hợp như một lớp tổng hợp tối đa. Một khối VGG bao gồm một chuỗi các lớp phức tạp, tiếp theo là một lớp tổng hợp tối đa để lấy mẫu không gian. Trong giấy VGG gốc (Simonyan & Zisserman, 2014), các tác giả đã sử dụng các phức hợp với 3×3 hạt nhân với lớp đệm 1 (giữ chiều cao và chiều rộng) và 2×2 tổng hợp tối đa với sải chân là 2 (giảm một nửa độ phân giải sau mỗi khối). Trong đoạn code dưới đây, ta định nghĩa một hàm gọi là `vgg_block` để thực hiện một khối VGG.

Hàm này có ba đối số tương ứng với số lớp phức tạp `num_convs`, số kênh đầu vào `in_channels` và số kênh đầu ra `out_channels`.

```
import torch
from torch import nn
from d2l import torch as d2l


def vgg_block(num_convs, in_channels, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels,
                                kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

8.2.2 VGG mạng

Giống như AlexNet và LeNet, Mạng VGG có thể được phân chia thành hai phần: phần đầu tiên bao gồm chủ yếu là các lớp phức tạp và tập hợp và thứ hai bao gồm các lớp được kết nối hoàn toàn. Điều này được miêu tả năm Fig. 8.2.1.

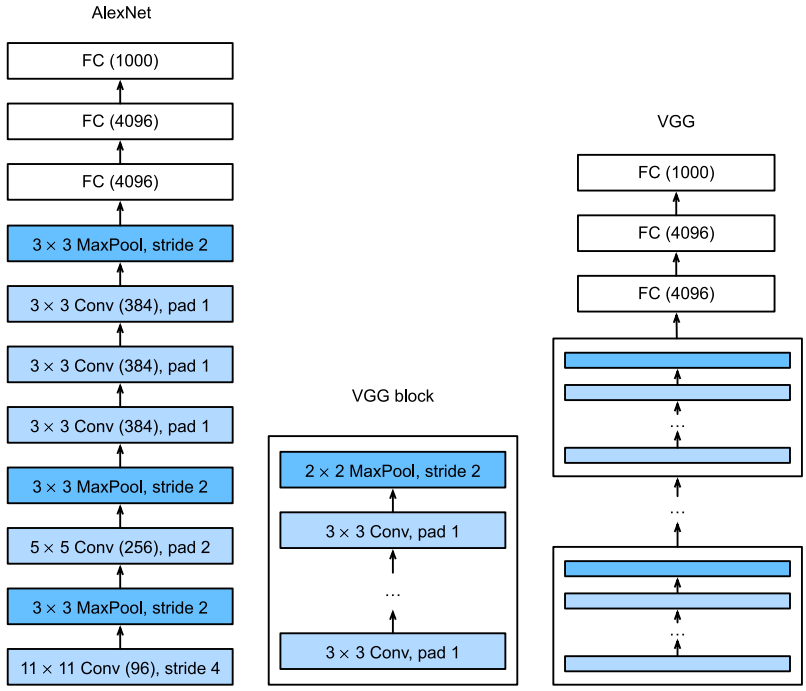


Fig. 8.2.1: From AlexNet to VGG that is designed from building blocks.

Phần tích tụ của mạng kết nối một số khối VGG từ Fig. 8.2.1 (cũng được định nghĩa trong hàm `vgg_block`) liên tiếp. Biến sau đây `conv_arch` bao gồm một danh sách các tuples (một cho mỗi khối), trong đó mỗi khối chứa hai giá trị: số lớp phức tạp và số kênh đầu ra, đó chính xác là các đối số cần thiết để gọi hàm `vgg_block`. Phần được kết nối hoàn toàn của mạng VGG giống hệt với phần được bao phủ trong AlexNet.

Mạng VGG ban đầu có 5 khối phức tạp, trong đó hai khối đầu tiên có một lớp ghép mỗi lớp và ba khối sau chứa hai lớp ghép mỗi lớp. Khối đầu tiên có 64 kênh đầu ra và mỗi khối tiếp theo tăng gấp đôi số kênh đầu ra, cho đến khi con số đó đạt 512. Vì mạng này sử dụng 8 lớp kết nối và 3 lớp kết nối hoàn toàn nên nó thường được gọi là VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

Mã sau đây thực hiện VGG-11. Đây là một vấn đề đơn giản của việc thực hiện một for-loop trên `conv_arch`.

```
def vgg(conv_arch):
    conv_blks = []
    in_channels = 1
    # The convolutional part
    for (num_convs, out_channels) in conv_arch:
        conv_blks.append(vgg_block(num_convs, in_channels, out_channels))
        in_channels = out_channels

    return nn.Sequential(
        *conv_blks, nn.Flatten(),
        # The fully-connected part
        nn.Linear(out_channels * 7 * 7, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 4096), nn.ReLU(), nn.Dropout(0.5),
        nn.Linear(4096, 10))

net = vgg(conv_arch)
```

Tiếp theo, chúng ta sẽ xây dựng một ví dụ dữ liệu đơn kênh với chiều cao và chiều rộng từ 224 đến quan sát hình dạng đầu ra của mỗi lớp.

```
X = torch.randn(size=(1, 1, 224, 224))
for blk in net:
    X = blk(X)
    print(blk.__class__.__name__,'output shape:\t',X.shape)
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

Như bạn có thể thấy, chúng ta giảm một nửa chiều cao và chiều rộng tại mỗi khối, cuối cùng đạt đến chiều cao và chiều rộng 7 trước khi làm phẳng các biểu diễn để xử lý bởi phần kết nối hoàn toàn của mạng.

8.2.3 Đào tạo

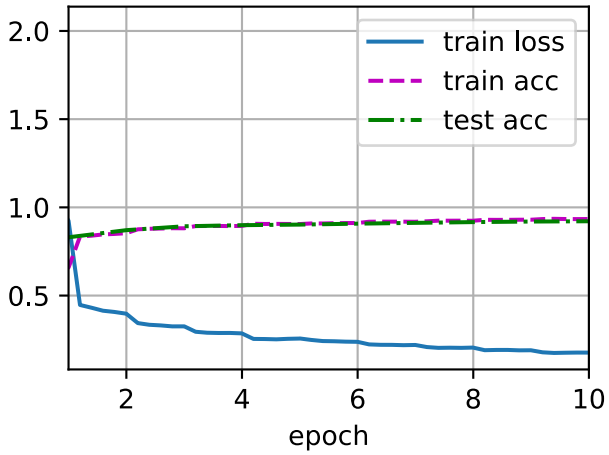
Vì VGG-11 nặng tính toán hơn AlexNet, chúng tôi xây dựng một mạng với số lượng kênh nhỏ hơn. Điều này là quá đủ để đào tạo về Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Ngoài việc sử dụng tốc độ học tập lớn hơn một chút, quá trình model training cũng tương tự như của AlexNet trong Section 8.1.

```
lr, num_epochs, batch_size = 0.05, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.177, train acc 0.934, test acc 0.921
2499.7 examples/sec on cuda:0
```



8.2.4 Tóm tắt

- VGG-11 xây dựng một mạng bằng cách sử dụng các khối phức tạp có thể tái sử dụng. Các mô hình VGG khác nhau có thể được xác định bởi sự khác biệt về số lượng lớp phức tạp và các kênh đầu ra trong mỗi khối.
- Việc sử dụng các khối dẫn đến các đại diện rất nhỏ gọn của định nghĩa mạng. Nó cho phép thiết kế hiệu quả các mạng phức tạp.
- Trong giấy VGG của họ, Simonyan và Ziserman đã thử nghiệm với nhiều kiến trúc khác nhau. Đặc biệt, họ phát hiện ra rằng một số lớp phức tạp sâu và hẹp (tức là 3×3) có hiệu quả hơn so với ít lớp phức tạp rộng hơn.

8.2.5 Bài tập

1. Khi in ra kích thước của các lớp chúng ta chỉ thấy 8 kết quả hơn là 11. Thông tin 3 lớp còn lại đã đi đâu?
2. So với AlexNet, VGG chậm hơn nhiều về mặt tính toán, và nó cũng cần nhiều bộ nhớ GPU hơn. Phân tích lý do cho việc này.
3. Hãy thử thay đổi chiều cao và chiều rộng của hình ảnh trong Fashion-MNIST từ 224 đến 96. Điều này có ảnh hưởng gì đến các thí nghiệm?
4. Tham khảo Bảng 1 trong giấy VGG (Simonyan & Zisserman, 2014) để xây dựng các mô hình phổ biến khác, chẳng hạn như VGG-16 hoặc VGG-19.

Discussions[92]

[92] https://discuss.d2l.ai/t/78

8.3 Mạng trong mạng (Nin)

LeNet, AlexNet và VGG đều chia sẻ một mẫu thiết kế chung: trích xuất các tính năng khai thác cấu trúc *không gian* thông qua một chuỗi các lớp phức tạp và tập hợp và sau đó xử lý các biểu diễn thông qua các lớp được kết nối hoàn toàn. Những cải tiến trên LeNet của AlexNet và VGG chủ yếu nằm ở cách các mạng sau này mở rộng và làm sâu sắc thêm hai mô-đun này. Ngoài ra, người ta có thể tưởng tượng sử dụng các lớp được kết nối hoàn toàn trước đó trong quá trình này. Tuy nhiên, việc sử dụng bất cẩn các lớp dày đặc có thể từ bỏ cấu trúc không gian của biểu diễn hoàn toàn, *mạng trong mạng* (* NiN*) khối cung cấp một giải pháp thay thế. Chúng được đề xuất dựa trên một cái nhìn sâu sắc rất đơn giản: sử dụng MLP trên các kênh cho mỗi pixel riêng biệt (Lin et al., 2013).

8.3.1 NiN Blocks

Nhớ lại rằng các đầu vào và đầu ra của các lớp phức tạp bao gồm các hàng chục bốn chiều với các trục tương ứng với ví dụ, kênh, chiều cao và chiều rộng. Cũng nhớ lại rằng các đầu vào và đầu ra của các lớp được kết nối hoàn toàn thường là các hàng chục hai chiều tương ứng với ví dụ và tính năng. Ý tưởng đằng sau Nin là áp dụng một lớp được kết nối hoàn toàn tại mỗi vị trí pixel (cho mỗi chiều cao và chiều rộng). Nếu chúng ta buộc trọng lượng trên mỗi vị trí không gian, chúng ta có thể nghĩ đây là một lớp ghép 1×1 (như được mô tả trong Section 7.4) hoặc như một lớp được kết nối hoàn toàn hoạt động độc lập trên mỗi vị trí pixel. Một cách khác để xem điều này là nghĩ về từng phần tử trong chiều không gian (chiều cao và chiều rộng) tương đương với một ví dụ và một kênh tương đương với một đối tượng.

Fig. 8.3.1 minh họa sự khác biệt về cấu trúc chính giữa VGG và Nin, và các khối của chúng. Khối Nin bao gồm một lớp phức tạp tiếp theo là hai lớp ghép 1×1 hoạt động như các lớp được kết nối hoàn toàn trên mỗi pixel với các kích hoạt ReLU. Hình dạng cửa sổ phức tạp của lớp đầu tiên thường được đặt bởi người dùng. Các hình dạng cửa sổ tiếp theo được cố định thành 1×1.

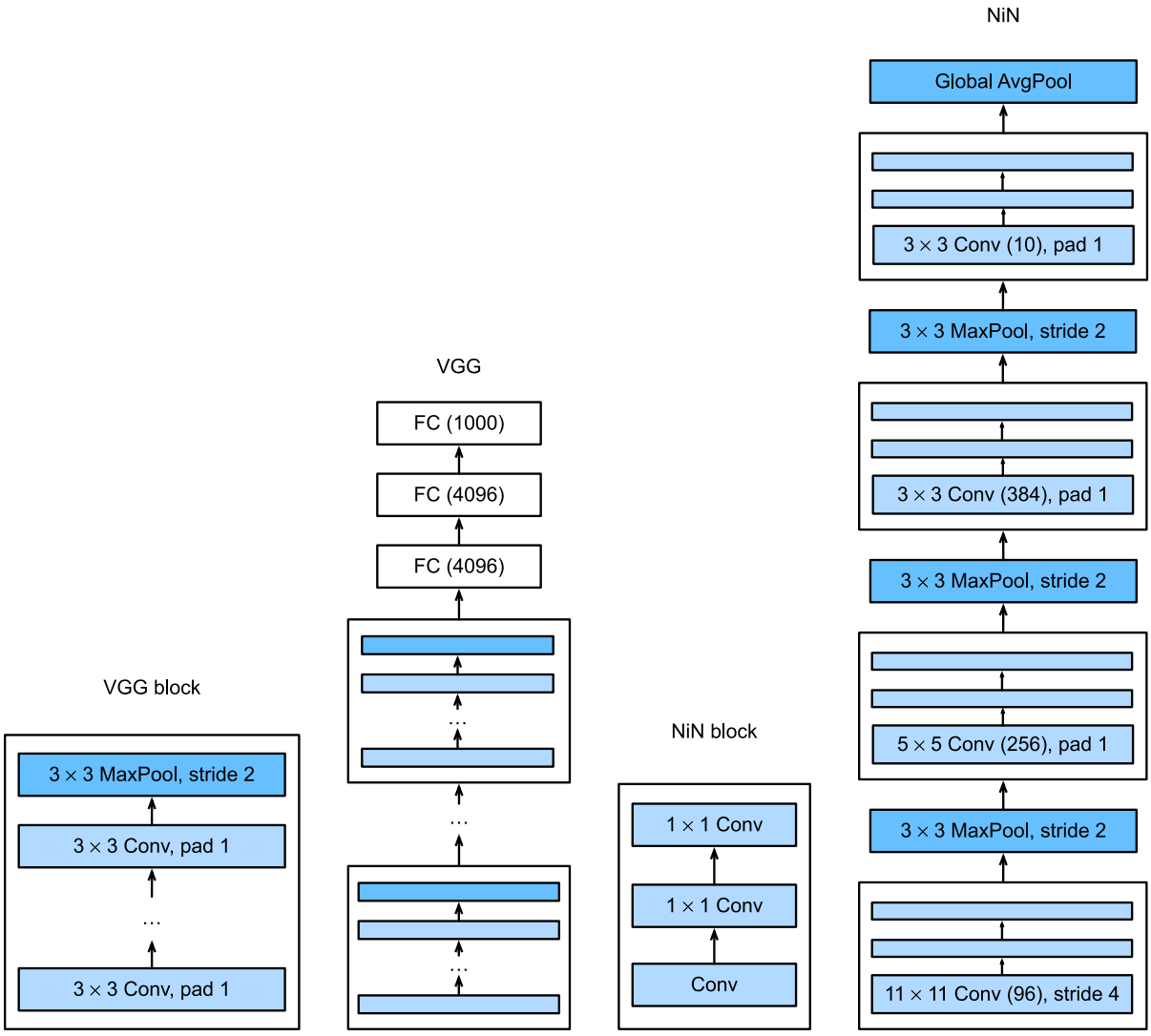


Fig. 8.3.1: Comparing architectures of VGG and NiN, and their blocks.

```
import torch
from torch import nn
from d2l import torch as d2l


def nin_block(in_channels, out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size, strides, padding),
        nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU(),
        nn.Conv2d(out_channels, out_channels, kernel_size=1), nn.ReLU())
```

8.3.2 Mô hình NiN

Mạng Nin ban đầu được đề xuất ngay sau AlexNet và rõ ràng rút ra một số cảm hứng. Nin sử dụng các lớp phức tạp với hình dạng cửa sổ 11×11, 5×5, và 3×3, và các số kênh đầu ra tương ứng giống như trong AlexNet. Mỗi khối Nin được theo sau bởi một lớp tổng hợp tối đa với một sải chân là 2 và một hình dạng cửa sổ là 3×3.

Một điểm khác biệt đáng kể giữa Nin và AlexNet là Nin tránh hoàn toàn các lớp được kết nối hoàn toàn. Thay vào đó, Nin sử dụng một khối Nin với một số kênh đầu ra bằng số lớp nhãn, tiếp theo là một lớp tổng hợp trung bình *global*, mang lại một vectơ của logits. Một ưu điểm của thiết kế của Nin là nó làm giảm đáng kể số lượng các thông số mô hình cần thiết. Tuy nhiên, trong thực tế, thiết kế này đôi khi đòi hỏi thời gian đào tạo mô hình tăng lên.

```
net = nn.Sequential(
    nin_block(1, 96, kernel_size=11, strides=4, padding=0),
    nn.MaxPool2d(3, stride=2),
    nin_block(96, 256, kernel_size=5, strides=1, padding=2),
    nn.MaxPool2d(3, stride=2),
    nin_block(256, 384, kernel_size=3, strides=1, padding=1),
    nn.MaxPool2d(3, stride=2),
    nn.Dropout(0.5),
    # There are 10 label classes
    nin_block(384, 10, kernel_size=3, strides=1, padding=1),
    nn.AdaptiveAvgPool2d((1, 1)),
    # Transform the four-dimensional output into two-dimensional output with a
    # shape of (batch size, 10)
    nn.Flatten())
```

Chúng ta tạo ra một ví dụ dữ liệu để xem hình dạng đầu ra của mỗi block.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

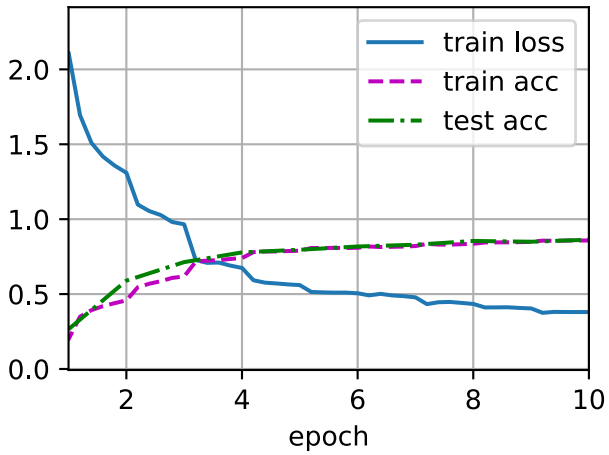
```
Sequential output shape:     torch.Size([1, 96, 54, 54])
MaxPool2d output shape:      torch.Size([1, 96, 26, 26])
Sequential output shape:     torch.Size([1, 256, 26, 26])
MaxPool2d output shape:      torch.Size([1, 256, 12, 12])
Sequential output shape:     torch.Size([1, 384, 12, 12])
MaxPool2d output shape:      torch.Size([1, 384, 5, 5])
Dropout output shape:        torch.Size([1, 384, 5, 5])
Sequential output shape:     torch.Size([1, 10, 5, 5])
AdaptiveAvgPool2d output shape:      torch.Size([1, 10, 1, 1])
Flatten output shape:        torch.Size([1, 10])
```

8.3.3 Đào tạo

Như trước đây chúng ta sử dụng Fashion-MNIST để đào tạo mô hình. Đào tạo của Nin tương tự như cho AlexNet và VGG.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.381, train acc 0.858, test acc 0.863
3160.4 examples/sec on cuda:0
```



8.3.4 Tóm tắt

- Nin sử dụng các khối bao gồm một lớp phức tạp và nhiều lớp ghép 1×1. Điều này có thể được sử dụng trong ngăn xếp phức tạp để cho phép tính phi tuyến trên mỗi pixel hơn.
- Nin loại bỏ các lớp được kết nối hoàn toàn và thay thế chúng bằng tổng hợp trung bình toàn cầu (tức là tổng hợp tất cả các vị trí) sau khi giảm số lượng kênh xuống số lượng đầu ra mong muốn (ví dụ: 10 cho Fashion-MNIST).
- Loại bỏ các lớp được kết nối hoàn toàn làm giảm quá mức. Nin có ít tham số hơn đáng kể.
- Thiết kế Nin ảnh hưởng đến nhiều thiết kế CNN tiếp theo.

8.3.5 Bài tập

1. Điều chỉnh các siêu tham số để cải thiện độ chính xác phân loại.
2. Tại sao có hai lớp ghép 1×1 trong khối Nin? Loại bỏ một trong số chúng, sau đó quan sát và phân tích các hiện tượng thử nghiệm.
3. Tính toán mức sử dụng tài nguyên cho Nin.
    1. Số lượng tham số là bao nhiêu?
    2. Số lượng tính toán là bao nhiêu?
    3. Dung lượng bộ nhớ cần thiết trong quá trình đào tạo là bao nhiêu?

4. Dung lượng bộ nhớ cần thiết trong quá trình dự đoán là bao nhiêu?

4. Những vấn đề có thể xảy ra với việc giảm đại diện $384 \times 5 \times 5$ thành đại diện $10 \times 5 \times 5$ trong một bước là gì?

Discussions[93]

8.4 Mạng có kết nối song song (GoogLeNet)

Năm 2014, *GoogLeNet* giành chiến thắng trong ImageNet Challenge, đề xuất một cấu trúc kết hợp các thế mạnh của Nin và mô hình của các khối lặp lại (Szegedy et al., 2015). Một trọng tâm của bài báo là giải quyết câu hỏi về hạt nhân có kích thước nào là tốt nhất. Rốt cuộc, các mạng phổ biến trước đây sử dụng các lựa chọn nhỏ tới 1×1 và lớn tới 11×11. Một cái nhìn sâu sắc trong bài báo này là đôi khi nó có thể thuận lợi để sử dụng một sự kết hợp của các hạt nhân có kích thước đa dạng. Trong phần này, chúng tôi sẽ giới thiệu GoogLeNet, trình bày một phiên bản đơn giản hóa một chút của mô hình gốc: chúng tôi bỏ qua một vài tính năng đặc biệt đã được thêm vào để ổn định đào tạo nhưng bây giờ không cần thiết với các thuật toán đào tạo tốt hơn có sẵn.

8.4.1 Chống** Bắt nối

Khối phức tạp cơ bản trong GoogLeNet được gọi là khối *Inception *, có thể được đặt tên do một trích dẫn từ bộ phim *Inception* ("Chúng ta cần phải đi sâu hơn"), đã đưa ra một meme virus.

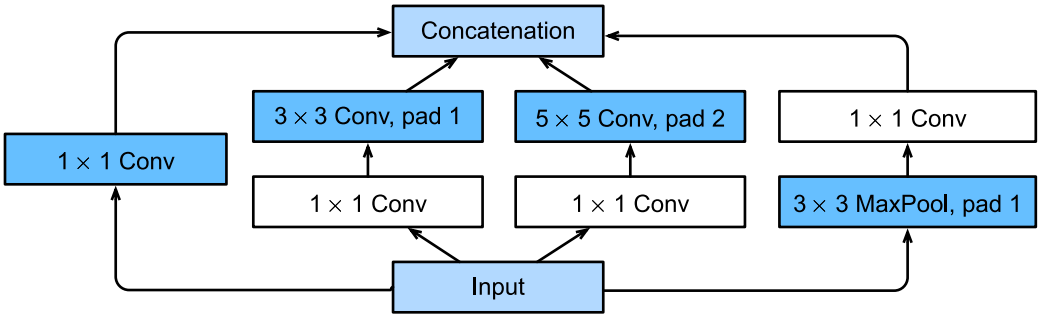


Fig. 8.4.1: Structure of the Inception block.

Như được mô tả trong Fig. 8.4.1, khối khởi đầu bao gồm bốn con đường song song. Ba đường dẫn đầu tiên sử dụng các lớp phức tạp với kích thước cửa sổ 1×1, 3×3 và 5×5 để trích xuất thông tin từ các kích thước không gian khác nhau. Hai đường dẫn giữa thực hiện sự phức tạp 1×1 trên đầu vào để giảm số lượng kênh, giảm độ phức tạp của mô hình. Đường dẫn thứ tư sử dụng một lớp tổng hợp tối đa 3×3, tiếp theo là một lớp ghép 1×1 để thay đổi số kênh. Bốn đường dẫn đều sử dụng đệm thích hợp để cung cấp cho đầu vào và đầu ra cùng chiều cao và chiều rộng. Cuối cùng, các đầu ra dọc theo mỗi đường dẫn được nối dọc theo kích thước kênh và bao gồm đầu ra của khối. Các siêu tham số được điều chỉnh chung của khối Inception là số kênh đầu ra trên mỗi lớp.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

(continues on next page)

[93] https://discuss.d2l.ai/t/80

(continued from previous page)

```
class Inception(nn.Module):
    # `c1`--`c4` are the number of output channels for each path
    def __init__(self, in_channels, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Path 1 is a single 1 x 1 convolutional layer
        self.p1_1 = nn.Conv2d(in_channels, c1, kernel_size=1)
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2d(in_channels, c2[0], kernel_size=1)
        self.p2_2 = nn.Conv2d(c2[0], c2[1], kernel_size=3, padding=1)
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2d(in_channels, c3[0], kernel_size=1)
        self.p3_2 = nn.Conv2d(c3[0], c3[1], kernel_size=5, padding=2)
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.p4_2 = nn.Conv2d(in_channels, c4, kernel_size=1)

    def forward(self, x):
        p1 = F.relu(self.p1_1(x))
        p2 = F.relu(self.p2_2(F.relu(self.p2_1(x))))
        p3 = F.relu(self.p3_2(F.relu(self.p3_1(x))))
        p4 = F.relu(self.p4_2(self.p4_1(x)))
        # Concatenate the outputs on the channel dimension
        return torch.cat((p1, p2, p3, p4), dim=1)
```

Để đạt được một số trực giác về lý do tại sao mạng này hoạt động rất tốt, hãy xem xét sự kết hợp của các bộ lọc. Họ khám phá hình ảnh trong một loạt các kích cỡ bộ lọc. Điều này có nghĩa là các chi tiết ở các phạm vi khác nhau có thể được nhận dạng hiệu quả bởi các bộ lọc có kích thước khác nhau. Đồng thời, chúng ta có thể phân bổ lượng tham số khác nhau cho các bộ lọc khác nhau.

8.4.2 Mô hình GoogLeNet

Như thể hiện trong Fig. 8.4.2, GoogLeNet sử dụng một chồng tổng cộng 9 khối khởi đầu và tổng hợp trung bình toàn cầu để tạo ra ước tính của nó. Tổng hợp tối đa giữa các khối khởi đầu làm giảm kích thước. mô-đun đầu tiên tương tự như AlexNet và LeNet. Ngăn xếp các khối được kế thừa từ VGG và tổng hợp trung bình toàn cầu tránh được một chồng các lớp được kết nối hoàn toàn ở cuối.

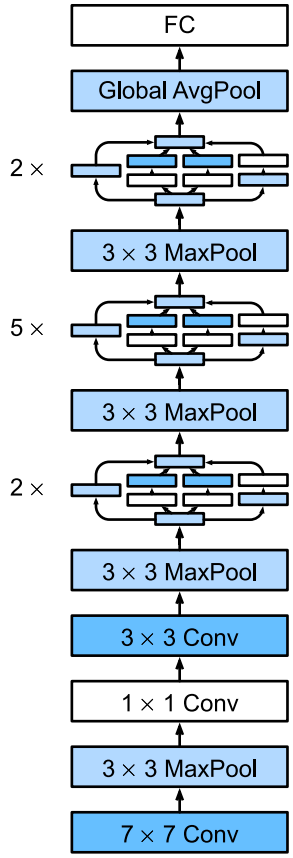


Fig. 8.4.2: The GoogLeNet architecture.

Bây giờ chúng ta có thể thực hiện GoogLeNet từng mảnh. mô-đun đầu tiên sử dụng lớp ghép 7×7 64 kênh.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

mô-đun thứ hai sử dụng hai lớp phức tạp: thứ nhất, một lớp ghép 1×1 64 kênh, sau đó là một lớp ghép 3×3 tăng gấp ba lần số kênh. Điều này tương ứng với đường dẫn thứ hai trong khối Inception.

```
b2 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=1),
                   nn.ReLU(),
                   nn.Conv2d(64, 192, kernel_size=3, padding=1),
                   nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

mô-đun thứ ba kết nối hai khối Inception hoàn chỉnh trong chuỗi. Số kênh đầu ra của khối Inception đầu tiên là $64+128+32+32 = 256$ và tỷ lệ số kênh đầu ra giữa bốn đường dẫn là $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. Các đường dẫn thứ hai và thứ ba đầu tiên làm giảm số lượng kênh đầu vào xuống $96/192 = 1/2$ và $16/192 = 1/12$, và sau đó kết nối lớp phức tạp thứ hai. Số lượng kênh đầu ra của khối Inception thứ hai được tăng lên $128 + 192 + 96 + 64 = 480$ và tỷ lệ số kênh đầu ra trong bốn đường dẫn là $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. Các đường dẫn thứ hai và thứ ba đầu tiên làm giảm số lượng kênh đầu vào xuống $128/256 = 1/2$ và $32/256 = 1/8$, tương ứng.

```
b3 = nn.Sequential(Inception(192, 64, (96, 128), (16, 32), 32),
                   Inception(256, 128, (128, 192), (32, 96), 64),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

mô-đun thứ tư phức tạp hơn. Nó kết nối năm khối Inception trong loạt, và họ có $192+208+48+64=512$, $160+224+64+64=512$, $128+256+64+64=512$, $112+288+64+64=528$, và $256+320+128+128=832$ kênh đầu ra, tương ứng. Số lượng kênh được gán cho các đường dẫn này tương tự như trong mô-đun thứ ba: đường dẫn thứ hai với lớp cuộn 3×3 xuất ra số lượng kênh lớn nhất, tiếp theo là đường dẫn đầu tiên chỉ có lớp ghép 1×1, đường dẫn thứ ba với lớp biên độ 5×5 và con đường thứ tư với lớp tổng hợp tối đa 3×3. Các đường dẫn thứ hai và thứ ba trước tiên sẽ giảm số lượng kênh theo tỷ lệ. Các tỷ lệ này hơi khác nhau trong các khối Inception khác nhau.

```
b4 = nn.Sequential(Inception(480, 192, (96, 208), (16, 48), 64),
                   Inception(512, 160, (112, 224), (24, 64), 64),
                   Inception(512, 128, (128, 256), (24, 64), 64),
                   Inception(512, 112, (144, 288), (32, 64), 64),
                   Inception(528, 256, (160, 320), (32, 128), 128),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

mô-đun thứ năm có hai khối Inception với các kênh đầu ra $256+320+128+128=832$ và $384+384+128+128=1024$. Số lượng kênh được gán cho mỗi đường dẫn giống như trong các mô-đun thứ ba và thứ tư, nhưng khác nhau về các giá trị cụ thể. Cần lưu ý rằng khối thứ năm được theo sau bởi lớp đầu ra. Khối này sử dụng lớp tổng hợp trung bình toàn cầu để thay đổi chiều cao và chiều rộng của mỗi kênh thành 1, giống như trong Nin. Cuối cùng, chúng ta biến đầu ra thành một mảng hai chiều theo sau là một lớp kết nối hoàn toàn có số lượng đầu ra là số lớp nhãn.

```
b5 = nn.Sequential(Inception(832, 256, (160, 320), (32, 128), 128),
                   Inception(832, 384, (192, 384), (48, 128), 128),
                   nn.AdaptiveAvgPool2d((1,1)),
                   nn.Flatten())

net = nn.Sequential(b1, b2, b3, b4, b5, nn.Linear(1024, 10))
```

Mô hình GoogLeNet phức tạp về mặt tính toán, vì vậy không dễ dàng để sửa đổi số lượng kênh như trong VGG. Để có thời gian đào tạo hợp lý về Fashion-MNIST, chúng tôi giảm chiều cao và chiều rộng đầu vào từ 224 xuống 96. Điều này đơn giản hóa việc tính toán. Những thay đổi về hình dạng của đầu ra giữa các mô-đun khác nhau được thể hiện dưới đây.

```
X = torch.rand(size=(1, 1, 96, 96))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

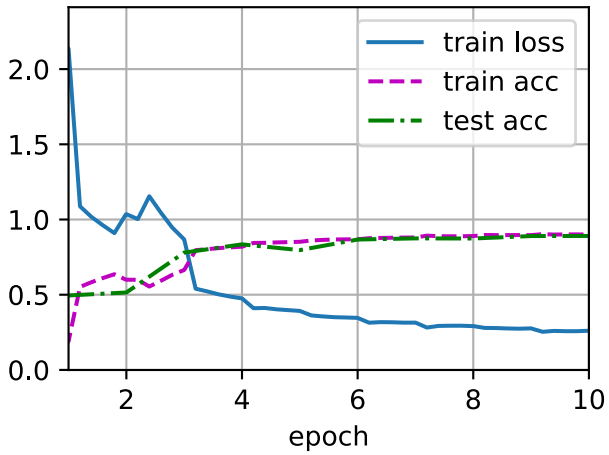
```
Sequential output shape:     torch.Size([1, 64, 24, 24])
Sequential output shape:     torch.Size([1, 192, 12, 12])
Sequential output shape:     torch.Size([1, 480, 6, 6])
Sequential output shape:     torch.Size([1, 832, 3, 3])
Sequential output shape:     torch.Size([1, 1024])
Linear output shape:         torch.Size([1, 10])
```

8.4.3 Đào tạo

Như trước đây, chúng tôi đào tạo mô hình của mình bằng cách sử dụng bộ dữ liệu Fashion-MNIST. Chúng tôi chuyển đổi nó thành độ phân giải 96×96 pixel trước khi gọi quy trình đào tạo.

```
lr, num_epochs, batch_size = 0.1, 10, 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.260, train acc 0.900, test acc 0.891
3459.1 examples/sec on cuda:0
```



8.4.4 Tóm tắt

- Khối Inception tương đương với một mạng con với bốn đường dẫn. Nó trích xuất thông tin song song thông qua các lớp phức tạp của các hình dạng cửa sổ khác nhau và các lớp tổng hợp tối đa. 1×1 sự covolutions làm giảm kích thước kênh ở mức mỗi pixel. Tổng hợp tối đa làm giảm độ phân giải.
- GoogLeNet kết nối nhiều khối Inception được thiết kế tốt với các lớp khác trong loạt. Tỷ lệ số kênh được gán trong khối Inception thu được thông qua một số lượng lớn các thí nghiệm trên tập dữ liệu ImageNet.
- GoogLeNet, cũng như các phiên bản thành công của nó, là một trong những mô hình hiệu quả nhất trên ImageNet, cung cấp độ chính xác thử nghiệm tương tự với độ phức tạp tính toán thấp hơn.

8.4.5 Bài tập

1. Có một số lần lặp lại của GoogLeNet. Cố gắng thực hiện và chạy chúng. Một số trong số họ bao gồm những điều sau đây:
    - Thêm một lớp chuẩn hóa hàng loạt (Ioffe & Szegedy, 2015), như được mô tả sau trong Section 8.5.
    - Thực hiện các điều chỉnh đối với khối Inception (Szegedy et al., 2016).
    - Sử dụng làm mịn nhãn cho mô hình regarization (Szegedy et al., 2016).
    - Bao gồm nó trong kết nối còn lại (Szegedy et al., 2017), như được mô tả sau trong Section 8.6.

2. Kích thước hình ảnh tối thiểu để GoogLeNet hoạt động là bao nhiêu?
3. So sánh kích thước tham số mô hình của AlexNet, VGG và Nin với GoogLeNet. Làm thế nào để hai kiến trúc mạng sau làm giảm đáng kể kích thước tham số mô hình?

Discussions[94]

8.5 Chuẩn hóa hàng loạt

Đào tạo mạng thần kinh sâu là khó khăn. Và khiến họ hội tụ trong một khoảng thời gian hợp lý có thể khó khăn. Trong phần này, chúng tôi mô tả * chuẩn hóa theo lô*, một kỹ thuật phổ biến và hiệu quả liên tục tăng tốc sự hội tụ của các mạng sâu (Ioffe & Szegedy, 2015). Cùng với các khối còn lại - được bao phủ sau này trong Section 8.6 — bình thường hóa hàng loạt đã giúp các học viên có thể thường xuyên đào tạo mạng với hơn 100 lớp.

8.5.1 Đào tạo mạng sâu

Để thúc đẩy bình thường hóa hàng loạt, chúng ta hãy xem xét một vài thách thức thực tế phát sinh khi đào tạo các mô hình học máy và mạng thần kinh nói riêng.

Đầu tiên, các lựa chọn liên quan đến tiền xử lý dữ liệu thường tạo ra sự khác biệt rất lớn trong kết quả cuối cùng. Nhớ lại ứng dụng MLP của chúng tôi để dự đoán giá nhà (Section 5.10). Bước đầu tiên của chúng tôi khi làm việc với dữ liệu thực là tiêu chuẩn hóa các tính năng đầu vào của chúng tôi cho mỗi tính năng có trung bình bằng 0 và phương sai của một. Trực giác, tiêu chuẩn hóa này chơi độc đáo với các trình tối ưu hóa của chúng tôi vì nó đặt các thông số* một ưu tiết* ở quy mô tương tự.

Thứ hai, đối với MLP hoặc CNN điển hình, khi chúng ta đào tạo, các biến (ví dụ, đầu ra chuyển đổi affine trong MLP) trong các lớp trung gian có thể lấy các giá trị với kích thước khác nhau rộng rãi: cả dọc theo các lớp từ đầu vào đến đầu ra, trên các đơn vị trong cùng một lớp và theo thời gian do các bản cập nhật của chúng tôi cho mô hình tham số. Các nhà phát minh của bình thường hóa hàng loạt đưa ra một cách không chính thức rằng sự trôi dạt này trong việc phân phối các biến như vậy có thể cản trở sự hội tụ của mạng. Bằng trực giác, chúng ta có thể phỏng đoán rằng nếu một lớp có giá trị biến gấp 100 lần so với một lớp khác, điều này có thể đòi hỏi phải điều chỉnh bù trong tỷ lệ học tập.

Thứ ba, các mạng sâu hơn rất phức tạp và dễ dàng có khả năng vượt trội. Điều này có nghĩa là sự chính quy hóa trở nên quan trọng hơn.

Chuẩn hóa hàng loạt được áp dụng cho các lớp riêng lẻ (tùy chọn, cho tất cả chúng) và hoạt động như sau: Trong mỗi lần lặp đào tạo, trước tiên chúng ta bình thường hóa các đầu vào (bình thường hóa hàng loạt) bằng cách trừ trung bình của chúng và chia cho độ lệch chuẩn của chúng, trong đó cả hai được ước tính dựa trên số liệu thống kê của minibatch hiện tại. Tiếp theo, chúng tôi áp dụng một hệ số tỷ lệ và bù tỷ lệ. Chính xác là do *bình thường hóa* dựa trên số liệu thống kê * lô* mà * chuẩn hóa lông* bắt nguồn tên của nó.

Lưu ý rằng nếu chúng tôi cố gắng áp dụng bình thường hóa hàng loạt với minibatches kích thước 1, chúng tôi sẽ không thể học bất cứ điều gì. Đó là bởi vì sau khi trừ các phương tiện, mỗi đơn vị ẩn sẽ lấy giá trị 0! Như bạn có thể đoán, vì chúng tôi đang dành cả một phần để bình thường hóa hàng loạt, với các minibatches đủ lớn, cách tiếp cận chứng minh hiệu quả và ổn định. Một takeaway ở đây là khi áp dụng bình thường hóa hàng loạt, sự lựa chọn kích thước lô có thể còn quan trọng hơn so với không bình thường hóa hàng loạt.

[94] https://discuss.d2l.ai/t/82

Chính thức, biểu thị bằng $\mathbf{x} \in \mathcal{B}$ một đầu vào để bình thường hóa hàng loạt (BN) đó là từ một minibatch $\mathcal{B}$, bình thường hóa hàng loạt biến đổi $\mathbf{x}$ theo biểu thức sau:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}. \tag{8.5.1}$$

Trong (8.5.1), $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ là trung bình mẫu và $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ là độ lệch chuẩn mẫu của minibatch $\mathcal{B}$. Sau khi áp dụng tiêu chuẩn hóa, minibatch kết quả có không trung bình và phương sai đơn vị. Bởi vì sự lựa chọn phương sai đơn vị (so với một số số phép thuật khác) là một lựa chọn tùy ý, chúng ta thường bao gồm elementwise *tham số quy mô* $\boldsymbol{\gamma}$ và *thay đổi* $\boldsymbol{\beta}$ that have the sametương tự shape hình dạng as $\mathbf{x}$. Lưu ý rằng $\boldsymbol{\gamma}$ và $\boldsymbol{\beta}$ là các thông số cần được học chung với các tham số mô hình khác.

Do đó, các cường độ biến đổi cho các lớp trung gian không thể phân kỳ trong quá trình đào tạo vì bình thường hóa hàng loạt tích cực tập trung và rescales chúng trở lại một trung bình và kích thước nhất định (thông qua $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ và $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$). Một phần của trực giác hoặc trí tuệ của học viên là bình thường hóa hàng loạt dường như cho phép tỷ lệ học tập tích cực hơn.

Chính thức, ta tính $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ và $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ trong (8.5.1) như sau:

$$\begin{aligned}
\hat{\boldsymbol{\mu}}_{\mathcal{B}} &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x}, \\
\hat{\boldsymbol{\sigma}}_{\mathcal{B}}^2 &= \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}})^2 + \epsilon.
\end{aligned} \tag{8.5.2}$$

Lưu ý rằng chúng ta thêm một hằng số nhỏ $\epsilon > 0$ vào ước tính phương sai để đảm bảo rằng chúng ta không bao giờ cố gắng chia bằng 0, ngay cả trong trường hợp ước tính phương sai thực nghiệm có thể biến mất. Các ước tính $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ và $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ chống lại vấn đề mở rộng quy mô bằng cách sử dụng các ước tính ồn ào về trung bình và phương sai. Bạn có thể nghĩ rằng tiếng ồn này nên là một vấn đề. Hóa ra, điều này thực sự có lợi.

Điều này hóa ra là một chủ đề định kỳ trong học sâu. Vì những lý do chưa được đặc trưng tốt về mặt lý thuyết, các nguồn tiếng ồn khác nhau trong tối ưu hóa thường dẫn đến đào tạo nhanh hơn và ít quá mức hơn: biến thể này dường như hoạt động như một hình thức chính quy hóa. Trong một số nghiên cứu sơ bộ, (Teye et al., 2018) và (Luo et al., 2018) liên quan đến các tính chất của bình thường hóa hàng loạt với các ưu tiên Bayesian và hình phạt tương ứng. Đặc biệt, điều này làm sáng tỏ câu đố tại sao bình thường hóa hàng loạt hoạt động tốt nhất cho kích thước minibatches vừa phải trong phạm vi $50 \sim 100$.

Sửa một mô hình được đào tạo, bạn có thể nghĩ rằng chúng tôi muốn sử dụng toàn bộ tập dữ liệu để ước tính trung bình và phương sai. Sau khi đào tạo hoàn tất, tại sao chúng ta lại muốn cùng một hình ảnh được phân loại khác nhau, tùy thuộc vào lô mà nó xảy ra cư trú? Trong quá trình đào tạo, tính toán chính xác như vậy là không khả thi vì các biến trung gian cho tất cả các ví dụ dữ liệu thay đổi mỗi khi chúng tôi cập nhật mô hình của chúng tôi. Tuy nhiên, một khi mô hình được đào tạo, chúng ta có thể tính toán các phương tiện và phương sai của các biến của mỗi lớp dựa trên toàn bộ tập dữ liệu. Thật vậy đây là thực hành tiêu chuẩn cho các mô hình sử dụng bình thường hóa hàng loạt và do đó các lớp bình thường hóa hàng loạt hoạt động khác nhau trong chế độ đào tạo* (bình thường hóa bằng thống kê minibatch) và trong chế độ dự đoán * (bình thường hóa theo thống kê tập dữ liệu).

Bây giờ chúng tôi đã sẵn sàng để xem cách bình thường hóa hàng loạt hoạt động trong thực tế.

8.5.2 Các lớp chuẩn hóa hàng loạt

Triển khai bình thường hóa hàng loạt cho các lớp được kết nối hoàn toàn và các lớp phức tạp hơi khác nhau. Chúng tôi thảo luận về cả hai trường hợp dưới đây. Nhớ lại rằng một sự khác biệt chính giữa chuẩn hóa hàng loạt và các lớp khác là vì việc chuẩn hóa hàng loạt hoạt động trên một minibatch đầy đủ tại một thời điểm, chúng ta không thể bỏ qua kích thước lô như chúng ta đã làm trước khi giới thiệu các layer khác.

Các lớp được kết nối hoàn toàn

Khi áp dụng bình thường hóa hàng loạt cho các lớp được kết nối hoàn toàn, giấy ban đầu sẽ chèn bình thường hóa hàng loạt sau khi chuyển đổi affine và trước chức năng kích hoạt phi tuyến (các ứng dụng sau này có thể chèn bình thường hóa hàng loạt ngay sau khi kích hoạt chức năng) (Ioffe & Szegedy, 2015). Biểu thị đầu vào cho lớp được kết nối hoàn toàn bằng $\mathbf{x}$, chuyển đổi affine bởi $\mathbf{W}\mathbf{x} + \mathbf{b}$ (với tham số trọng lượng $\mathbf{W}$ và tham số thiên vị $\mathbf{b}$) và chức năng kích hoạt bởi ϕ, chúng ta có thể thể hiện tính toán của đầu ra lớp được kích hoạt theo lô, được kết nối đầy đủ $\mathbf{h}$ như sau:

$$\mathbf{h} = \phi(\text{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{8.5.3}$$

Nhớ lại rằng trung bình và phương sai được tính toán trên minibatch * same* mà việc chuyển đổi được áp dụng.

Layers phức tạp

Tương tự, với các lớp phức tạp, chúng ta có thể áp dụng bình thường hóa hàng loạt sau khi kết hợp và trước hàm kích hoạt phi tuyến. Khi sự phức tạp có nhiều kênh đầu ra, chúng ta cần thực hiện bình thường hóa hàng loạt cho * mỗi* đầu ra của các kênh này và mỗi kênh có các thông số quy mô và thay đổi riêng, cả hai đều là vô hướng. Giả sử rằng minibatches của chúng tôi chứa m ví dụ và đối với mỗi kênh, đầu ra của sự kết hợp có chiều cao p và chiều rộng q. Đối với các lớp phức tạp, chúng tôi thực hiện mỗi đợt bình thường hóa trên $m \cdot p \cdot q$ các phần tử trên mỗi kênh đầu ra cùng một lúc. Do đó, chúng tôi thu thập các giá trị trên tất cả các vị trí không gian khi tính toán trung bình và phương sai và do đó áp dụng cùng một trung bình và phương sai trong một kênh nhất định để bình thường hóa giá trị tại mỗi vị trí không gian.

Chuẩn hóa hàng loạt trong dự đoán

Như chúng tôi đã đề cập trước đó, bình thường hóa hàng loạt thường hoạt động khác nhau trong chế độ đào tạo và chế độ dự đoán. Đầu tiên, tiếng ồn trong trung bình mẫu và phương sai mẫu phát sinh từ việc ước tính từng chiếc trên minibatches không còn mong muốn một khi chúng tôi đã đào tạo mô hình. Thứ hai, chúng ta có thể không có sự sang trọng của tính toán số liệu thống kê bình thường hóa mỗi lô. Ví dụ, chúng ta có thể cần áp dụng mô hình của mình để đưa ra một dự đoán tại một thời điểm.

Thông thường, sau khi đào tạo, chúng tôi sử dụng toàn bộ tập dữ liệu để tính toán các ước tính ổn định của số liệu thống kê biến và sau đó sửa chúng vào thời điểm dự đoán. Do đó, bình thường hóa hàng loạt hoạt động khác nhau trong quá trình đào tạo và tại thời điểm thử nghiệm. Nhớ lại rằng bỏ học cũng thể hiện đặc điểm này.

8.5.3 Implementation from Scratch

Dưới đây, chúng tôi thực hiện một lớp bình thường hóa hàng loạt với hàng chục từ đầu.

```
import torch
from torch import nn
from d2l import torch as d2l


def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use `is_grad_enabled` to determine whether the current mode is training
    # mode or prediction mode
    if not torch.is_grad_enabled():
        # If it is prediction mode, directly use the mean and variance
        # obtained by moving average
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:
            # When using a fully-connected layer, calculate the mean and
            # variance on the feature dimension
            mean = X.mean(dim=0)
            var = ((X - mean) ** 2).mean(dim=0)
        else:
            # When using a two-dimensional convolutional layer, calculate the
            # mean and variance on the channel dimension (axis=1). Here we
            # need to maintain the shape of `X`, so that the broadcasting
            # operation can be carried out later
            mean = X.mean(dim=(0, 2, 3), keepdim=True)
            var = ((X - mean) ** 2).mean(dim=(0, 2, 3), keepdim=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the mean and variance using moving average
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift
    return Y, moving_mean.data, moving_var.data
```

Bây giờ chúng ta có thể tạo một lớp `BatchNorm` thích hợp. Lớp của chúng tôi sẽ duy trì các thông số thích hợp cho quy mô `gamma` và thay đổi `beta`, cả hai sẽ được cập nhật trong quá trình đào tạo. Ngoài ra, lớp của chúng ta sẽ duy trì đường trung bình động của phương tiện và phương sai để sử dụng tiếp theo trong quá trình dự đoán mô hình.

Gạt các chi tiết thuật toán sang một bên, lưu ý mô hình thiết kế bên dưới việc thực hiện lớp của chúng ta. Thông thường, chúng tôi xác định toán học trong một hàm riêng biệt, nói `batch_norm`. Sau đó, chúng tôi tích hợp chức năng này vào một lớp tùy chỉnh, có mã chủ yếu đề cập đến các vấn đề kế toán, chẳng hạn như di chuyển dữ liệu sang bối cảnh thiết bị phù hợp, phân bổ và khởi tạo bất kỳ biến nào cần thiết, theo dõi các đường trung bình động (ở đây cho trung bình và phương sai), v.v. Mô hình này cho phép tách toán học sạch khỏi mã boilerplate. Cũng lưu ý rằng vì lợi ích của sự tiện lợi, chúng tôi đã không lo lắng về việc tự động suy ra hình dạng đầu vào ở đây, do đó chúng tôi cần chỉ định số lượng các tính năng trong suốt. Đừng lo lắng, các API chuẩn hóa hàng loạt cấp cao trong khuôn khổ học tập sâu sẽ quan tâm đến điều này cho chúng tôi và chúng tôi sẽ chứng minh rằng sau này.

```
class BatchNorm(nn.Module):
    # `num_features`: the number of outputs for a fully-connected layer
    # or the number of output channels for a convolutional layer. `num_dims`:
    # 2 for a fully-connected layer and 4 for a convolutional layer
    def __init__(self, num_features, num_dims):
        super().__init__()
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter (model parameters) are
        # initialized to 1 and 0, respectively
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # The variables that are not model parameters are initialized to 0␣
↪and 1
        self.moving_mean = torch.zeros(shape)
        self.moving_var = torch.ones(shape)

    def forward(self, X):
        # If `X` is not on the main memory, copy `moving_mean` and
        # `moving_var` to the device where `X` is located
        if self.moving_mean.device != X.device:
            self.moving_mean = self.moving_mean.to(X.device)
            self.moving_var = self.moving_var.to(X.device)
        # Save the updated `moving_mean` and `moving_var`
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

8.5.4 Áp dụng Bình thường hóa hàng loạt trong LeNet

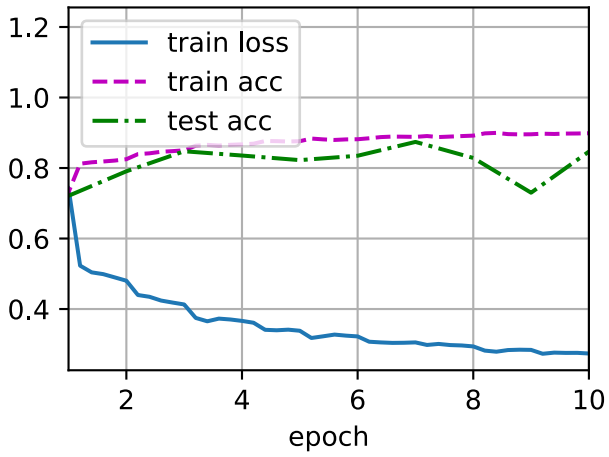
Để xem cách áp dụng `BatchNorm` trong bối cảnh, bên dưới chúng tôi áp dụng nó cho một mô hình LeNet truyền thống (Section 7.6). Nhớ lại rằng việc chuẩn hóa hàng loạt được áp dụng sau các lớp phức tạp hoặc các lớp được kết nối hoàn toàn nhưng trước các chức năng kích hoạt tương ứng.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), BatchNorm(6, num_dims=4), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), BatchNorm(16, num_dims=4), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(16*4*4, 120), BatchNorm(120, num_dims=2), nn.Sigmoid(),
    nn.Linear(120, 84), BatchNorm(84, num_dims=2), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Như trước đây, chúng tôi sẽ đào tạo mạng của chúng tôi trên dữ liệu Fashion-MNIST. Mã này hầu như giống hệt với điều đó khi chúng tôi lần đầu tiên đào tạo LeNet (Section 7.6). Sự khác biệt chính là tỷ lệ học tập lớn hơn.

```
lr, num_epochs, batch_size = 1.0, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.274, train acc 0.899, test acc 0.847
40075.7 examples/sec on cuda:0
```



Hãy để chúng tôi có một cái nhìn tại tham số quy mô `gamma` và tham số shift `beta` học được từ lớp chuẩn hóa hàng loạt đầu tiên.

```
net[1].gamma.reshape((-1,)), net[1].beta.reshape((-1,))
```

```
(tensor([3.3951, 1.3776, 3.8082, 2.5995, 1.6570, 2.8225], device='cuda:0',
        grad_fn=<ReshapeAliasBackward0>),
 tensor([-2.7676,  1.5291, -2.5730,  2.6386,  0.0966, -0.6415], device='cuda:0
↪',
        grad_fn=<ReshapeAliasBackward0>))
```

8.5.5 Thiết tập

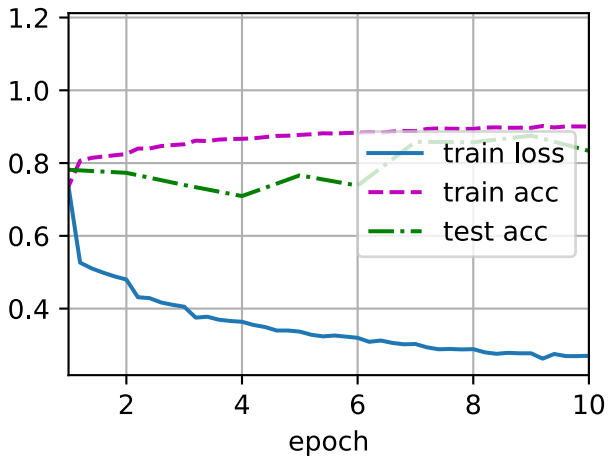
So với lớp `BatchNorm`, mà chúng ta vừa định nghĩa, chúng ta có thể sử dụng lớp `BatchNorm` được định nghĩa trong API cấp cao từ khung học sâu trực tiếp. Mã trông hầu như giống hệt với việc thực hiện của chúng tôi ở trên.

```
net = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5), nn.BatchNorm2d(6), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.Sigmoid(),
    nn.AvgPool2d(kernel_size=2, stride=2), nn.Flatten(),
    nn.Linear(256, 120), nn.BatchNorm1d(120), nn.Sigmoid(),
    nn.Linear(120, 84), nn.BatchNorm1d(84), nn.Sigmoid(),
    nn.Linear(84, 10))
```

Dưới đây, chúng ta sử dụng cùng một siêu tham số để đào tạo mô hình của chúng tôi. Lưu ý rằng như thường lệ, biến thể API cấp cao chạy nhanh hơn nhiều vì mã của nó đã được biên dịch thành C++ hoặc CDA trong khi triển khai tùy chỉnh của chúng tôi phải được giải thích bởi Python.

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.270, train acc 0.900, test acc 0.834
71223.6 examples/sec on cuda:0
```



8.5.6 Tranh cãi

Trực giác, bình thường hóa hàng loạt được cho là làm cho cảnh quan tối ưu hóa mượt mà hơn. Tuy nhiên, chúng ta phải cẩn thận để phân biệt giữa trực giác đầu cơ và giải thích thực sự cho các hiện tượng mà chúng ta quan sát khi đào tạo các mô hình sâu. Nhớ lại rằng chúng ta thậm chí không biết tại sao các mạng thần kinh sâu đơn giản hơn (MLP và CNN thông thường) khái quát tốt ngay từ đầu. Ngay cả khi bỏ học và phân rã trọng lượng, chúng vẫn linh hoạt đến mức khả năng khái quát hóa với dữ liệu vô hình của họ không thể được giải thích thông qua các đảm bảo tổng quát học hỏi thông thường.

Trong bài viết gốc đề xuất bình thường hóa hàng loạt, các tác giả, ngoài việc giới thiệu một công cụ mạnh mẽ và hữu ích, đã đưa ra một lời giải thích cho lý do tại sao nó hoạt động: bằng cách giảm sự thay đổi giao hợp nội bộ*. Có lẽ bởi * chuyển giao hợp nội bộ* các tác giả có nghĩa là một cái gì đó giống như trực giác thể hiện ở trên - khái niệm rằng sự phân bố các giá trị biến thay đổi trong quá trình đào tạo. Tuy nhiên, đã có hai vấn đề với lời giải thích này: i) Sự trôi dạt này rất khác so với * covariate shift*, khiến tên một tên sai. ii) Lời giải thích cung cấp một trực giác dưới quy định nhưng để lại câu hỏi * tại sao chính xác kỹ thuật này hoạt động* một câu hỏi mở muốn giải thích nghiêm ngặt . Trong suốt cuốn sách này, chúng tôi đặt mục tiêu truyền đạt những trực giác mà các học viên sử dụng để hướng dẫn sự phát triển của họ về các mạng thần kinh sâu. Tuy nhiên, chúng tôi tin rằng điều quan trọng là phải tách các trực giác hướng dẫn này khỏi thực tế khoa học đã được thiết lập. Cuối cùng, khi bạn nắm vững tài liệu này và bắt đầu viết các bài báo nghiên cứu của riêng bạn, bạn sẽ muốn rõ ràng để phân định giữa các tuyên bố kỹ thuật và hunches.

Sau sự thành công của việc bình thường hóa hàng loạt, lời giải thích của nó về * chuyển giao hợp nội bộ* đã nhiều lần nổi lên trong các cuộc tranh luận trong tài liệu kỹ thuật và diễn ngôn rộng hơn về cách trình bày nghiên cứu machine learning. Trong một bài phát biểu đáng nhớ được đưa ra trong khi chấp nhận Giải thưởng Thử nghiệm Thời gian tại hội nghị NeurIPS 2017, Ali Rahimi đã sử dụng dịch chuyển giao hợp nội bộ * làm tâm điểm trong một cuộc tranh luận giống như thực hành học sâu hiện đại với giả kim thuật. Sau đó, ví dụ đã được xem xét lại chi tiết trong một bài báo vị trí phác thảo xu hướng gây rắc rối trong máy học (Lipton & Steinhardt, 2018). Các tác giả khác đã đề xuất các giải thích thay thế cho sự thành công của việc bình thường hóa hàng loạt, một số người tuyên bố rằng thành công của bình thường hóa hàng loạt đến mặc dù thể hiện hành vi theo một số cách trái ngược với những người được tuyên bố trong bài báo gốc (Santurkar et al., 2018).

Chúng tôi lưu ý rằng sự thay đổi nội bộ* không xứng đáng với những lời chỉ trích hơn bất kỳ trong số hàng ngàn tuyên bố mơ hồ tương tự được đưa ra mỗi năm trong tài liệu học máy kỹ thuật. Có khả năng, sự cộng

hưởng của nó như là một tâm điểm của các cuộc tranh luận này nợ khả năng nhận biết rộng rãi của nó đối với đối tượng mục tiêu. Chuẩn hóa hàng loạt đã chứng minh một phương pháp không thể thiếu, áp dụng trong gần như tất cả các phân loại hình ảnh được triển khai, kiếm được bài báo giới thiệu kỹ thuật hàng chục ngàn trích dẫn.

8.5.7 Tóm tắt

- Trong quá trình đào tạo mô hình, bình thường hóa hàng loạt liên tục điều chỉnh đầu ra trung gian của mạng thần kinh bằng cách sử dụng độ lệch trung bình và chuẩn của minibatch, do đó các giá trị của đầu ra trung gian trong mỗi lớp trong suốt mạng thần kinh ổn định hơn.
- Các phương pháp bình thường hóa hàng loạt cho các lớp được kết nối hoàn toàn và các lớp phức tạp hơi khác nhau.
- Giống như một lớp bỏ học, các lớp chuẩn hóa hàng loạt có kết quả tính toán khác nhau trong chế độ đào tạo và chế độ dự đoán.
- Bình thường hóa hàng loạt có nhiều tác dụng phụ có lợi, chủ yếu là thường xuyên hóa. Mặt khác, động lực ban đầu của việc giảm sự thay đổi đồng biến nội bộ dường như không phải là một lời giải thích hợp lệ.

8.5.8 Bài tập

1. Chúng ta có thể loại bỏ tham số thiên vị khỏi lớp được kết nối hoàn toàn hoặc lớp phức tạp trước khi bình thường hóa hàng loạt không? Tại sao?
2. So sánh tỷ lệ học tập cho LeNet có và không có bình thường hóa hàng loạt.
    1. Vẽ sự gia tăng trong đào tạo và kiểm tra độ chính xác.
    2. Làm thế nào lớn bạn có thể làm cho tỷ lệ học tập?
3. Chúng ta có cần bình thường hóa hàng loạt trong mỗi lớp không? Thử nghiệm với nó?
4. Bạn có thể thay thế bỏ học bằng cách bình thường hóa hàng loạt không? Hành vi thay đổi như thế nào?
5. Khắc phục các thông số `beta` và `gamma`, và quan sát và phân tích kết quả.
6. Xem lại tài liệu trực tuyến cho `BatchNorm` từ API cấp cao để xem các ứng dụng khác để bình thường hóa hàng loạt.
7. Ý tưởng nghiên cứu: nghĩ về các biến đổi bình thường hóa khác mà bạn có thể áp dụng? Bạn có thể áp dụng biến đổi tích phân xác suất? Làm thế nào về một ước tính hiệp phương sai cấp đầy đủ?

Discussions[95]

[95] https://discuss.d2l.ai/t/84

8.6 Mạng dư (ResNet)

Khi chúng ta thiết kế các mạng ngày càng sâu hơn, bắt buộc phải hiểu cách thêm các lớp có thể làm tăng độ phức tạp và biểu cảm của mạng. Thậm chí quan trọng hơn nữa là khả năng thiết kế mạng mà việc thêm các lớp làm cho các mạng trở nên biểu cảm nghiêm ngặt hơn là chỉ khác biệt. Để đạt được một số tiến bộ, chúng ta cần một chút toán học.

8.6.1 Các lớp hàm

Xem xét $\mathcal{F}$, lớp hàm mà một kiến trúc mạng cụ thể (cùng với tốc độ học tập và các cài đặt siêu tham số khác) có thể đạt được. Đó là, đối với tất cả $f \in \mathcal{F}$ có tồn tại một số tập hợp các tham số (ví dụ, trọng lượng và thành kiến) có thể thu được thông qua đào tạo trên một tập dữ liệu phù hợp. Chúng ta hãy giả sử rằng f^* là chức năng "sự thật" mà chúng ta thực sự muốn tìm thấy. Nếu nó là trong $\mathcal{F}$, chúng tôi đang trong tình trạng tốt nhưng thông thường chúng tôi sẽ không hoàn toàn may mắn như vậy. Thay vào đó, chúng tôi sẽ cố gắng tìm một số $f^*_{\mathcal{F}}$ là đặt cược tốt nhất của chúng tôi trong vòng $\mathcal{F}$. Ví dụ, với một tập dữ liệu với các tính năng $\mathbf{X}$ và nhãn $\mathbf{y}$, chúng ta có thể thử tìm nó bằng cách giải quyết vấn đề tối ưu hóa sau:

$$f^*_{\mathcal{F}} \stackrel{\text{def}}{=} \operatorname*{argmin}_f L(\mathbf{X}, \mathbf{y}, f) \text{ subject to } f \in \mathcal{F}. \tag{8.6.1}$$

Nó chỉ là hợp lý để giả định rằng nếu chúng ta thiết kế một kiến trúc khác và mạnh mẽ hơn $\mathcal{F}'$ chúng ta nên đi đến một kết quả tốt hơn. Nói cách khác, chúng tôi hy vọng rằng $f^*_{\mathcal{F}'}$ là "tốt hơn" so với $f^*_{\mathcal{F}}$. Tuy nhiên, nếu $\mathcal{F} \not\subseteq \mathcal{F}'$ không có gì đảm bảo rằng điều này thậm chí sẽ xảy ra. Trong thực tế, $f^*_{\mathcal{F}'}$ cũng có thể tồi tệ hơn. Như minh họa bởi Fig. 8.6.1, đối với các lớp hàm không lồng nhau, một lớp hàm lớn hơn không phải lúc nào cũng di chuyển gần hơn với hàm "truth" f^*. Ví dụ, ở bên trái của Fig. 8.6.1, mặc dù $\mathcal{F}_3$ gần f^* hơn $\mathcal{F}_1$, $\mathcal{F}_6$ di chuyển đi và không có gì đảm bảo rằng việc tăng thêm độ phức tạp có thể làm giảm khoảng cách từ f^*. Với các lớp hàm lồng nhau trong đó $\mathcal{F}_1 \subseteq \ldots \subseteq \mathcal{F}_6$ ở bên phải của Fig. 8.6.1, chúng ta có thể tránh được vấn đề nói trên từ các lớp hàm không lồng nhau.

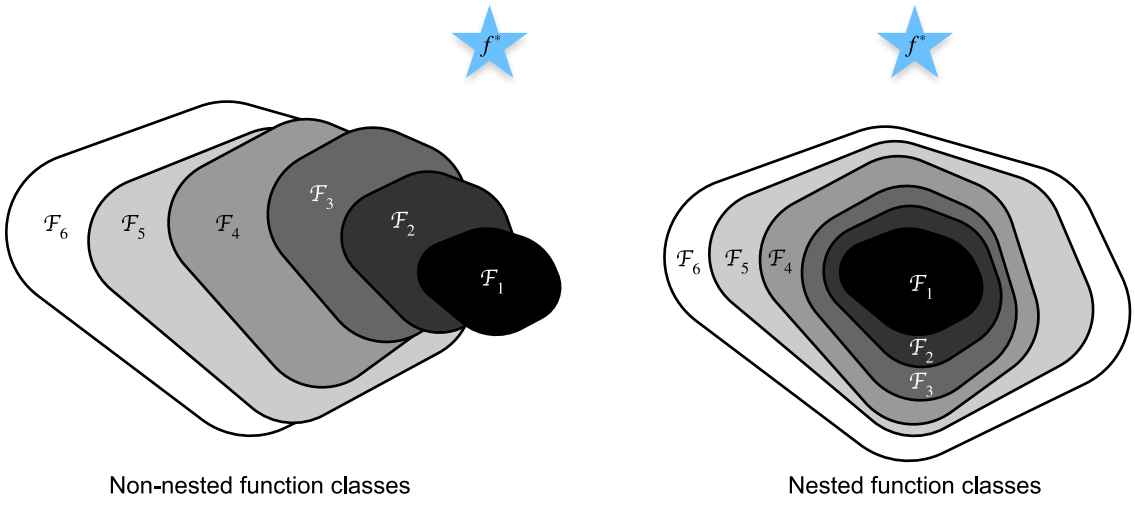


Fig. 8.6.1: For non-nested function classes, a larger (indicated by area) function class does not guarantee to get closer to the "truth" function (f^*). This does not happen in nested function classes.

Do đó, chỉ khi các lớp hàm lớn hơn chứa các lớp nhỏ hơn, chúng tôi đảm bảo rằng việc tăng chúng làm tăng nghiêm ngặt sức mạnh biểu cảm của mạng. Đối với các mạng thần kinh sâu, nếu chúng ta có thể đào tạo lớp mới được thêm vào một hàm nhận dạng $f(\mathbf{x}) = \mathbf{x}$, mô hình mới sẽ hiệu quả như mô hình ban đầu. Vì mô

hình mới có thể nhận được một giải pháp tốt hơn để phù hợp với tập dữ liệu đào tạo, lớp được thêm vào có thể giúp giảm lỗi đào tạo dễ dàng hơn.

Đây là câu hỏi mà Ông et al. xem xét khi làm việc trên các mô hình tầm nhìn máy tính rất sâu (He et al., 2016a). Trọng tâm của họ* mạng dư * (* ResNet*) được đề xuất là ý tưởng rằng mỗi lớp bổ sung nên dễ dàng chứa chức năng nhận dạng như một trong những yếu tố của nó. Những cân nhắc này khá sâu sắc nhưng chúng dẫn đến một giải pháp đơn giản đáng ngạc nhiên, một khối * dư*. Với nó, ResNet đã giành chiến thắng trong ImageNet Large Scale Visual Recognition Challenge vào năm 2015. Thiết kế có ảnh hưởng sâu sắc đến cách xây dựng các mạng thần kinh sâu.

8.6.2 Khối** Residual

Chúng ta hãy tập trung vào một phần cục bộ của mạng thần kinh, như được mô tả trong Fig. 8.6.2. Biểu thị đầu vào bằng $\mathbf{x}$. Chúng tôi giả định rằng bản đồ cơ bản mong muốn mà chúng tôi muốn có được bằng cách học là $f(\mathbf{x})$, được sử dụng làm đầu vào cho hàm kích hoạt ở trên cùng. Ở bên trái của Fig. 8.6.2, phần trong hộp dòng dotted-line phải trực tiếp tìm hiểu bản đồ $f(\mathbf{x})$. Ở bên phải, phần trong hộp dòng dotted-line cần tìm hiểu ánh xạ dư* $f(\mathbf{x}) - \mathbf{x}$, đó là cách khối còn lại lấy tên của nó. Nếu ánh xạ nhận dạng $f(\mathbf{x}) = \mathbf{x}$ là ánh xạ cơ bản mong muốn, ánh xạ còn lại sẽ dễ học hơn: chúng ta chỉ cần đẩy trọng lượng và thành kiến của lớp trọng lượng trên (ví dụ: lớp kết nối hoàn toàn và lớp kết nối) trong hộp dòng dotted-line về 0. Con số bên phải trong Fig. 8.6.2 minh họa khối * dư* của ResNet, trong đó đường rắn mang đầu vào lớp $\mathbf{x}$ đến toán tử bổ sung được gọi là kết nối dư* (hoặc kết nối phím tắt *). Với các khối còn lại, đầu vào có thể chuyển tiếp tuyên truyền nhanh hơn thông qua các kết nối còn lại trên các lớp.

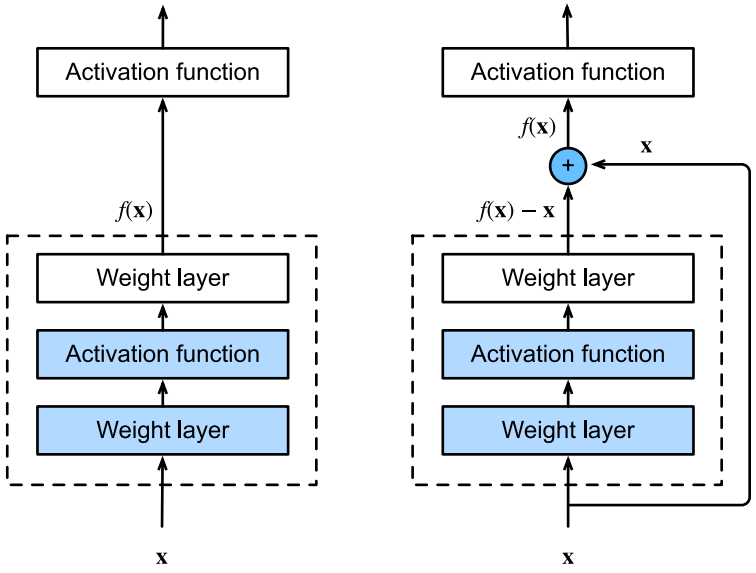


Fig. 8.6.2: A regular block (left) and a residual block (right).

ResNet tuân theo thiết kế lớp phức tạp 3×3 đầy đủ của VGG. Khối còn lại có hai lớp ghép 3×3 với cùng số kênh đầu ra. Mỗi lớp phức tạp được theo sau bởi một lớp chuẩn hóa hàng loạt và một chức năng kích hoạt ReLU. Sau đó, chúng tôi bỏ qua hai thao tác phức tạp này và thêm đầu vào trực tiếp trước chức năng kích hoạt ReLU cuối cùng. Kiểu thiết kế này đòi hỏi đầu ra của hai lớp ghép phải có cùng hình dạng với đầu vào, để chúng có thể được thêm vào lại với nhau. Nếu chúng ta muốn thay đổi số lượng kênh, chúng ta cần giới thiệu thêm một lớp ghép 1×1 để chuyển đổi đầu vào thành hình dạng mong muốn cho thao tác bổ sung. Hãy để chúng tôi có một cái nhìn tại mã dưới đây.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


class Residual(nn.Module):  #@save
    """The Residual block of ResNet."""
    def __init__(self, input_channels, num_channels,
                 use_1x1conv=False, strides=1):
        super().__init__()
        self.conv1 = nn.Conv2d(input_channels, num_channels,
                               kernel_size=3, padding=1, stride=strides)
        self.conv2 = nn.Conv2d(num_channels, num_channels,
                               kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2d(input_channels, num_channels,
                                   kernel_size=1, stride=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm2d(num_channels)
        self.bn2 = nn.BatchNorm2d(num_channels)

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        Y += X
        return F.relu(Y)
```

Mã này tạo ra hai loại mạng: một trong đó chúng tôi thêm đầu vào vào đầu ra trước khi áp dụng tính phi tuyến ReLU bất cứ khi nào `use_1x1conv=False` và một loại mà chúng tôi điều chỉnh các kênh và độ phân giải bằng cách ghép 1×1 trước khi thêm. Fig. 8.6.3 minh họa điều này:

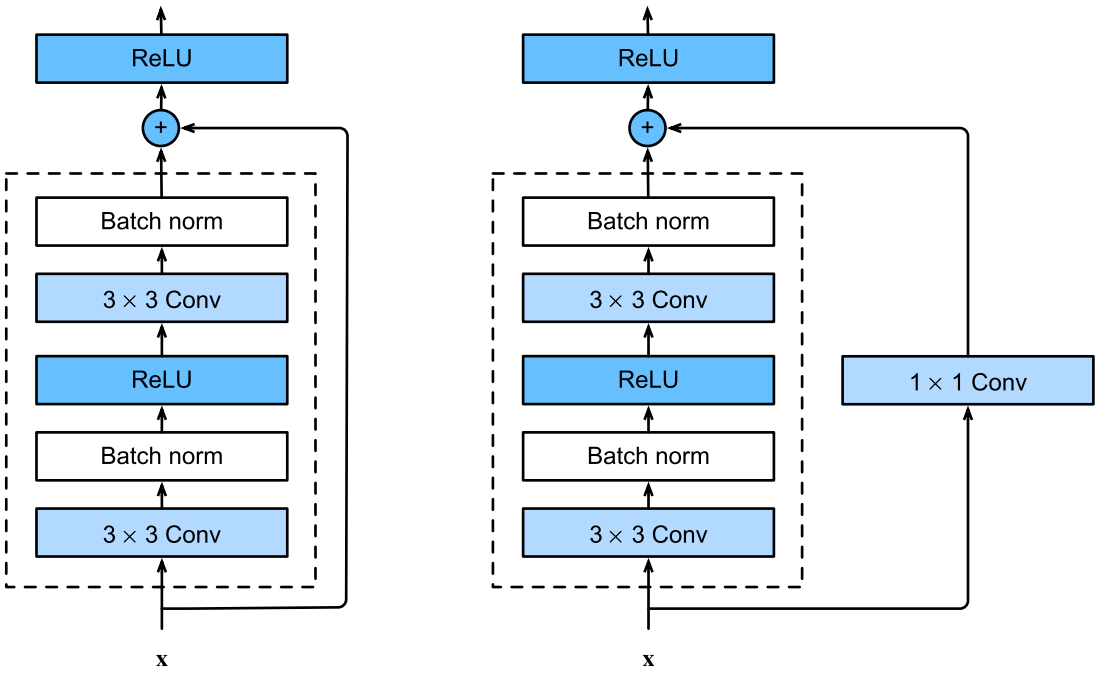


Fig. 8.6.3: ResNet block with and without 1×1 convolution.

Bây giờ chúng ta hãy nhìn vào một tình huống mà đầu vào và đầu ra có cùng hình dạng.

```
blk = Residual(3,3)
X = torch.rand(4, 3, 6, 6)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 3, 6, 6])
```

Chúng tôi cũng có tùy chọn để giảm một nửa chiều cao và chiều rộng đầu ra trong khi tăng số lượng kênh đầu ra.

```
blk = Residual(3,6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

8.6.3 Mô hình ResNet

Hai lớp ResNet đầu tiên giống như của GoogLeNet mà chúng tôi đã mô tả trước đây: lớp ghép 7×7 với 64 kênh đầu ra và bước tiến 2 được theo sau bởi lớp tổng hợp tối đa 3×3 với một sải chân là 2. Sự khác biệt là lớp chuẩn hóa hàng loạt được thêm vào sau mỗi lớp phức tạp trong ResNet.

```
b1 = nn.Sequential(nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
                   nn.BatchNorm2d(64), nn.ReLU(),
                   nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet sử dụng bốn mô-đun được tạo thành từ các khối Inception. Tuy nhiên, ResNet sử dụng bốn mô-đun được tạo thành từ các khối còn lại, mỗi khối sử dụng một số khối còn lại với cùng một số kênh đầu ra. Số lượng kênh trong mô-đun đầu tiên giống như số kênh đầu vào. Kể từ khi một lớp tổng hợp tối đa với một sải chân là 2 đã được sử dụng, nó không phải là cần thiết để giảm chiều cao và chiều rộng. Trong khối dư đầu tiên cho mỗi mô-đun tiếp theo, số lượng kênh được tăng gấp đôi so với mô-đun trước đó và chiều cao và chiều rộng giảm một nửa.

Bây giờ, chúng tôi thực hiện mô-đun này. Lưu ý rằng xử lý đặc biệt đã được thực hiện trên mô-đun đầu tiên.

```
def resnet_block(input_channels, num_channels, num_residuals,
                 first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(input_channels, num_channels,
                                use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels, num_channels))
    return blk
```

Sau đó, chúng tôi thêm tất cả các mô-đun vào ResNet. Ở đây, hai khối còn lại được sử dụng cho mỗi mô-đun.

```
b2 = nn.Sequential(*resnet_block(64, 64, 2, first_block=True))
b3 = nn.Sequential(*resnet_block(64, 128, 2))
b4 = nn.Sequential(*resnet_block(128, 256, 2))
b5 = nn.Sequential(*resnet_block(256, 512, 2))
```

Cuối cùng, giống như GoogLeNet, chúng ta thêm một lớp tổng hợp trung bình toàn cầu, tiếp theo là đầu ra lớp được kết nối hoàn toàn.

```
net = nn.Sequential(b1, b2, b3, b4, b5,
                    nn.AdaptiveAvgPool2d((1,1)),
                    nn.Flatten(), nn.Linear(512, 10))
```

Có 4 lớp phức tạp trong mỗi mô-đun (không bao gồm lớp ghép 1×1). Cùng với lớp kết nối 7×7 đầu tiên và lớp kết nối hoàn toàn cuối cùng, tổng cộng có 18 lớp. Do đó, mô hình này thường được gọi là ResNet-18. Bằng cách định cấu hình các số kênh và khối còn lại khác nhau trong mô-đun, chúng ta có thể tạo các mô hình ResNet khác nhau, chẳng hạn như ResNet-152 lớp sâu hơn 152. Mặc dù kiến trúc chính của ResNet tương tự như kiến trúc của GoogLeNet, cấu trúc của ResNet đơn giản và dễ sửa đổi hơn. Tất cả những yếu tố này đã dẫn đến việc sử dụng ResNet nhanh chóng và rộng rãi. Fig. 8.6.4 mô tả đầy đủ ResNet-18.

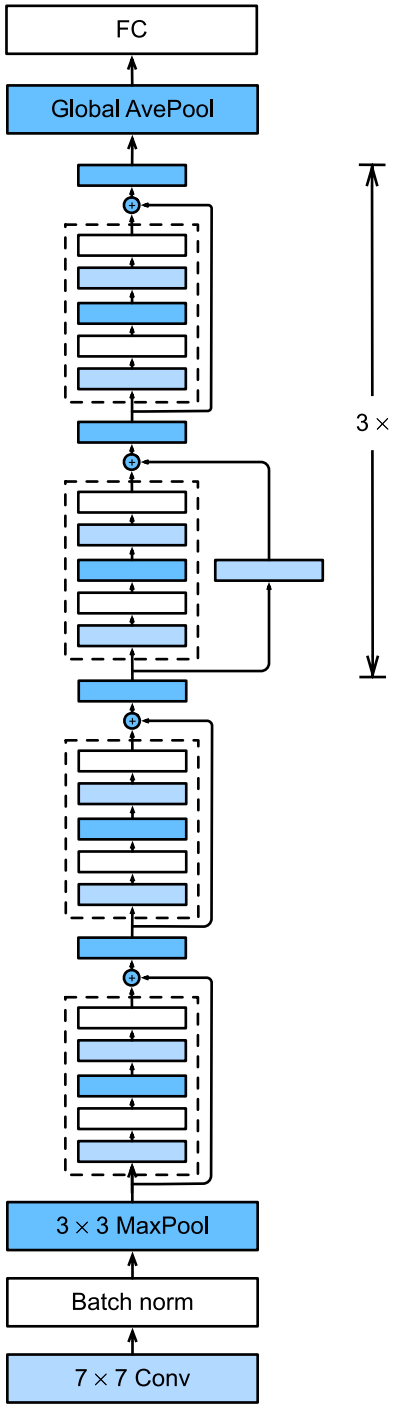


Fig. 8.6.4: The ResNet-18 architecture.

Trước khi đào tạo ResNet, chúng ta hãy quan sát cách hình dạng đầu vào thay đổi trên các mô-đun khác nhau trong ResNet. Như trong tất cả các kiến trúc trước đó, độ phân giải giảm trong khi số lượng kênh tăng lên cho đến khi điểm mà một lớp tổng hợp trung bình toàn cầu tổng hợp tất cả các tính năng.

```
X = torch.rand(size=(1, 1, 224, 224))
for layer in net:
    X = layer(X)
    print(layer.__class__.__name__,'output shape:\t', X.shape)
```

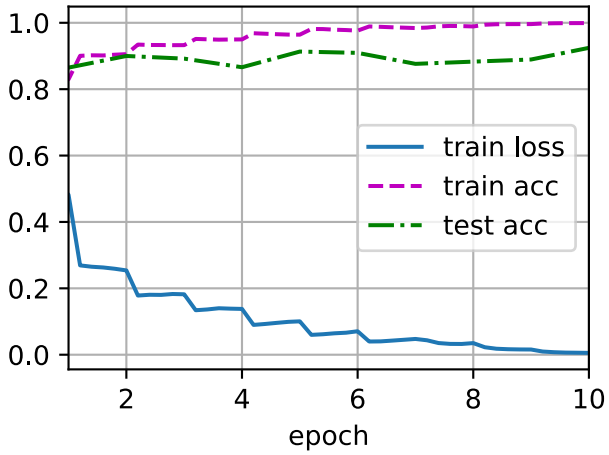
```
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 64, 56, 56])
Sequential output shape:     torch.Size([1, 128, 28, 28])
Sequential output shape:     torch.Size([1, 256, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
AdaptiveAvgPool2d output shape:      torch.Size([1, 512, 1, 1])
Flatten output shape:        torch.Size([1, 512])
Linear output shape:         torch.Size([1, 10])
```

8.6.4 Đào tạo

Chúng tôi đào tạo ResNet trên bộ dữ liệu Fashion-MNIST, giống như trước đây.

```
lr, num_epochs, batch_size = 0.05, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.006, train acc 0.999, test acc 0.924
4716.9 examples/sec on cuda:0
```



8.6.5 Tóm tắt

- Các lớp hàm lồng nhau là mong muốn. Học thêm một lớp trong các mạng thần kinh sâu như một chức năng nhận dạng (mặc dù đây là một trường hợp cực đoan) nên được thực hiện dễ dàng.
- Ánh xạ còn lại có thể tìm hiểu hàm nhận dạng dễ dàng hơn, chẳng hạn như đẩy các tham số trong lớp trọng lượng xuống 0.
- Chúng ta có thể đào tạo một mạng lưới thần kinh sâu hiệu quả bằng cách có các khối còn lại. Đầu vào có thể chuyển tiếp tuyên truyền nhanh hơn thông qua các kết nối còn lại trên các lớp.
- ResNet có ảnh hưởng lớn đến việc thiết kế các mạng thần kinh sâu tiếp theo, cả về tính chất phức tạp và tuần tự.

8.6.6 Bài tập

1. Sự khác biệt lớn giữa khối Inception trong Fig. 8.4.1 và khối còn lại là gì? Sau khi loại bỏ một số đường dẫn trong khối Inception, chúng liên quan đến nhau như thế nào?
2. Tham khảo Bảng 1 trong giấy ResNet (He et al., 2016a) để thực hiện các biến thể khác nhau.
3. Đối với các mạng sâu hơn, ResNet giới thiệu kiến trúc "nút cổ chai" để giảm độ phức tạp của mô hình. Cố gắng thực hiện nó.
4. Trong các phiên bản tiếp theo của ResNet, các tác giả đã thay đổi cấu trúc "phức tạp, bình thường hóa hàng loạt và kích hoạt" thành cấu trúc "bình thường hóa hàng loạt, kích hoạt và phức tạp". Tự cải thiện này. Xem Hình 1 trong (He et al., 2016b) để biết chi tiết.
5. Tại sao chúng ta không thể tăng độ phức tạp của hàm mà không bị ràng buộc, ngay cả khi các lớp hàm được lồng nhau?

Discussions[96]

8.7 Mạng kết nối mật độ (DenseNet)

ResNet thay đổi đáng kể quan điểm về cách tham số hóa các chức năng trong các mạng sâu. * DenseNet* (mạng phức tạp dày đặc) ở một mức độ nào đó là phần mở rộng hợp lý của (Huang et al., 2017) này. Để hiểu làm thế nào để đến nó, chúng ta hãy đi một đường vòng nhỏ đến toán học.

8.7.1 Từ ResNet đến DenseNet

Nhớ lại bản mở rộng Taylor cho các chức năng. Đối với điểm $x = 0$ nó có thể được viết là

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \dots. \tag{8.7.1}$$

Điểm mấu chốt là nó phân hủy một hàm thành các thuật ngữ thứ tự ngày càng cao hơn. Trong một tĩnh mạch tương tự, ResNet phân hủy các chức năng thành

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \tag{8.7.2}$$

Đó là, ResNet phân hủy f thành một thuật ngữ tuyến tính đơn giản và một thuật ngữ phi tuyến phức tạp hơn. Điều gì sẽ xảy ra nếu chúng ta muốn nắm bắt (không nhất thiết phải thêm) thông tin ngoài hai thuật ngữ? Một giải pháp là DenseNet (Huang et al., 2017).

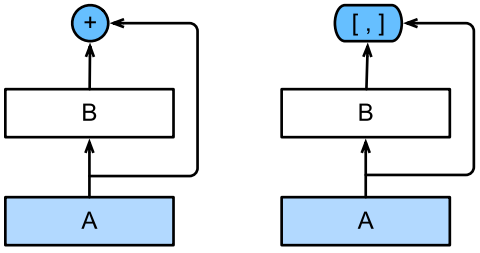


Fig. 8.7.1: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

[96] https://discuss.d2l.ai/t/86

Như thể hiện trong Fig. 8.7.1, sự khác biệt chính giữa ResNet và DenseNet là trong trường hợp đầu ra sau là * concatenated* (được biểu thị bởi $[,]$) thay vì được thêm vào. Do đó, chúng tôi thực hiện ánh xạ từ $\mathbf{x}$ đến các giá trị của nó sau khi áp dụng một chuỗi hàm ngày càng phức tạp:

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots] . \tag{8.7.3}$$

Cuối cùng, tất cả các chức năng này được kết hợp trong MLP để giảm số lượng tính năng một lần nữa. Về mặt thực hiện, điều này khá đơn giản: thay vì thêm các thuật ngữ, chúng tôi nối chúng. Tên DenseNet phát sinh từ thực tế là đồ thị phụ thuộc giữa các biến trở nên khá dày đặc. Lớp cuối cùng của một chuỗi như vậy được kết nối mật độ với tất cả các lớp trước đó. Các kết nối dày đặc được hiển thị trong Fig. 8.7.2.

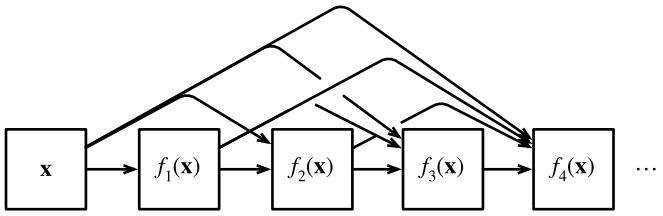


Fig. 8.7.2: Dense connections in DenseNet.

Các thành phần chính tạo ra DenseNet là các khối * dày đặc * và lớp chuyển thuyền*. Cái trước xác định cách các đầu vào và đầu ra được nối, trong khi cái sau kiểm soát số lượng kênh để nó không quá lớn.

8.7.2 Khối dày dàng

DenseNet sử dụng cấu trúc "chuẩn hóa hàng loạt, kích hoạt và phức tạp" được sửa đổi của ResNet (xem bài tập trong Section 8.6). Đầu tiên, chúng tôi thực hiện cấu trúc khối phức tạp này.

```
import torch
from torch import nn
from d2l import torch as d2l


def conv_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=3, padding=1))
```

Một khối * dày dàng* bao gồm nhiều khối phức tạp, mỗi khối sử dụng cùng một số kênh đầu ra. Tuy nhiên, trong quá trình lan truyền về phía trước, chúng tôi nối đầu vào và đầu ra của mỗi khối phức tạp trên kích thước kênh.

```
class DenseBlock(nn.Module):
    def __init__(self, num_convs, input_channels, num_channels):
        super(DenseBlock, self).__init__()
        layer = []
        for i in range(num_convs):
            layer.append(conv_block(
                num_channels * i + input_channels, num_channels))
        self.net = nn.Sequential(*layer)
```

(continues on next page)

(continued from previous page)

```
    def forward(self, X):
        for blk in self.net:
            Y = blk(X)
            # Concatenate the input and output of each block on the channel
            # dimension
            X = torch.cat((X, Y), dim=1)
        return X
```

Trong ví dụ sau, chúng ta define a `DenseBlock` instance với 2 khối phức tạp của 10 kênh đầu ra. Khi sử dụng đầu vào với 3 kênh, chúng tôi sẽ nhận được đầu ra với $3 + 2 \times 10 = 23$ kênh. Số lượng các kênh khối phức tạp kiểm soát sự tăng trưởng về số lượng kênh đầu ra so với số lượng kênh đầu vào. Đây cũng được gọi là * tốc độ tăng trưởng*.

```
blk = DenseBlock(2, 3, 10)
X = torch.randn(4, 3, 8, 8)
Y = blk(X)
Y.shape
```

```
torch.Size([4, 23, 8, 8])
```

8.7.3 Layers Transition

Vì mỗi khối dày đặc sẽ làm tăng số lượng kênh, việc thêm quá nhiều trong số chúng sẽ dẫn đến một mô hình quá phức tạp. Một lớp chuyển đổi* được sử dụng để kiểm soát độ phức tạp của mô hình. Nó làm giảm số lượng kênh bằng cách sử dụng lớp ghép 1×1 và giảm một nửa chiều cao và chiều rộng của lớp tổng hợp trung bình với một bước tiến là 2, làm giảm thêm độ phức tạp của mô hình.

```
def transition_block(input_channels, num_channels):
    return nn.Sequential(
        nn.BatchNorm2d(input_channels), nn.ReLU(),
        nn.Conv2d(input_channels, num_channels, kernel_size=1),
        nn.AvgPool2d(kernel_size=2, stride=2))
```

Áp dụng một lớp chuyển tiếp với 10 kênh đến đầu ra của khối dày đặc trong ví dụ trước. Điều này làm giảm số lượng kênh đầu ra xuống 10 và giảm một nửa chiều cao và chiều rộng.

```
blk = transition_block(23, 10)
blk(Y).shape
```

```
torch.Size([4, 10, 4, 4])
```

8.7.4 Mô hình DenseNet

Tiếp theo, chúng ta sẽ xây dựng một mô hình DenseNet. DenseNet lần đầu tiên sử dụng cùng một lớp tích hợp đơn và lớp tổng hợp tối đa như trong ResNet.

```
b1 = nn.Sequential(
    nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3),
    nn.BatchNorm2d(64), nn.ReLU(),
    nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

Sau đó, tương tự như bốn mô-đun được tạo thành từ các khối còn lại mà ResNet sử dụng, DenseNet sử dụng bốn khối dày đặc. Tương tự như ResNet, chúng ta có thể đặt số lớp phức tạp được sử dụng trong mỗi khối dày đặc. Ở đây, chúng tôi đặt nó thành 4, phù hợp với mô hình ResNet-18 trong Section 8.6. Hơn nữa, chúng tôi đặt số lượng kênh (tức là tốc độ tăng trưởng) cho các lớp phức tạp trong khối dày đặc thành 32, vì vậy 128 kênh sẽ được thêm vào mỗi khối dày đặc.

Trong ResNet, chiều cao và chiều rộng được giảm giữa mỗi mô-đun bởi một khối còn lại với một sải chân là 2. Ở đây, chúng ta sử dụng layer transition để giảm một nửa chiều cao và chiều rộng và giảm một nửa số kênh.

```
# `num_channels`: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]
blks = []
for i, num_convs in enumerate(num_convs_in_dense_blocks):
    blks.append(DenseBlock(num_convs, num_channels, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that halves the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        blks.append(transition_block(num_channels, num_channels // 2))
        num_channels = num_channels // 2
```

Tương tự như ResNet, một lớp tổng hợp toàn cầu và một lớp kết nối hoàn toàn được kết nối ở cuối để tạo ra đầu ra.

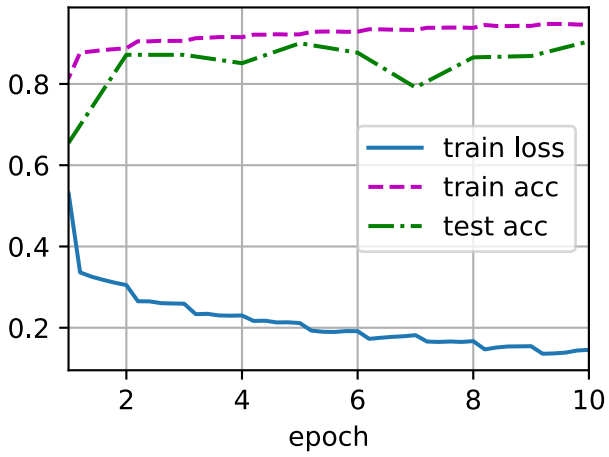
```
net = nn.Sequential(
    b1, *blks,
    nn.BatchNorm2d(num_channels), nn.ReLU(),
    nn.AdaptiveAvgPool2d((1, 1)),
    nn.Flatten(),
    nn.Linear(num_channels, 10))
```

8.7.5 Đào tạo

Vì chúng ta đang sử dụng mạng sâu hơn ở đây, trong phần này, chúng ta sẽ giảm chiều cao và chiều rộng đầu vào từ 224 xuống 96 để đơn giản hóa tính toán.

```
lr, num_epochs, batch_size = 0.1, 10, 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr, d2l.try_gpu())
```

```
loss 0.146, train acc 0.946, test acc 0.905
5558.7 examples/sec on cuda:0
```



8.7.6 Tóm tắt

- Về kết nối chéo lớp, không giống như ResNet, nơi đầu vào và đầu ra được thêm vào với nhau, DenseNet nối đầu vào và đầu ra trên kích thước kênh.
- Các thành phần chính sáng tác DenseNet là các khối dày đặc và các lớp chuyển tiếp.
- Chúng ta cần phải kiểm soát kích thước khi soạn mạng bằng cách thêm các lớp chuyển tiếp thu nhỏ lại số kênh.

8.7.7 Bài tập

1. Tại sao chúng ta sử dụng tổng hợp trung bình chứ không phải là tổng hợp tối đa trong layer chuyển tiếp?
2. Một trong những lợi thế được đề cập trong giấy DenseNet là các thông số mô hình của nó nhỏ hơn so với các thông số của ResNet. Tại sao lại là trường hợp này?
3. Một vấn đề mà DenseNet đã bị chỉ trích là mức tiêu thụ bộ nhớ cao.
    1. Đây có thực sự là trường hợp? Cố gắng thay đổi hình dạng đầu vào thành 224×224 để xem mức tiêu thụ bộ nhớ GPU thực tế.
    2. Bạn có thể nghĩ về một phương tiện thay thế để giảm tiêu thụ bộ nhớ? Làm thế nào bạn sẽ cần phải thay đổi khuôn khổ?
4. Thực hiện các phiên bản DenseNet khác nhau được trình bày trong Bảng 1 của giấy DenseNet (Huang et al., 2017).
5. Thiết kế mô hình dựa trên MLP bằng cách áp dụng ý tưởng DenseNet. Áp dụng nó cho nhiệm vụ dự đoán giá nhà ở trong Section 5.10.

Discussions[97]

[97] https://discuss.d2l.ai/t/88

9 | Mạng nơ-ron tái phát

Cho đến nay chúng tôi đã gặp hai loại dữ liệu: dữ liệu dạng bảng và dữ liệu hình ảnh. Đối với cái sau, chúng tôi đã thiết kế các lớp chuyên dụng để tận dụng sự đều đặn trong chúng. Nói cách khác, nếu chúng ta hoán vị các pixel trong một hình ảnh, sẽ khó khăn hơn nhiều để lý luận về nội dung của nó về một cái gì đó trông giống như nền của một mẫu thử nghiệm trong thời đại TV analog.

Quan trọng nhất, cho đến nay chúng tôi ngầm cho rằng dữ liệu của chúng tôi đều được rút ra từ một số phân phối và tất cả các ví dụ được phân phối độc lập và giống hệt nhau (i.i.d.). Thật không may, điều này không đúng đối với hầu hết dữ liệu. Ví dụ, các từ trong đoạn này được viết theo thứ tự, và sẽ khá khó để giải mã ý nghĩa của nó nếu chúng được hoán vị ngẫu nhiên. Tương tự như vậy, khung hình ảnh trong video, tín hiệu âm thanh trong cuộc trò chuyện và hành vi duyệt web trên một trang web, tất cả đều theo thứ tự tuần tự. Do đó, hợp lý khi cho rằng các mô hình chuyên dụng cho dữ liệu như vậy sẽ làm tốt hơn trong việc mô tả chúng.

Một vấn đề khác phát sinh từ thực tế là chúng ta có thể không chỉ nhận được một chuỗi như một đầu vào mà có thể được dự kiến sẽ tiếp tục trình tự. Ví dụ, nhiệm vụ có thể là tiếp tục loạt $2, 4, 6, 8, 10, \ldots$ Điều này khá phổ biến trong phân tích chuỗi thời gian, để dự đoán thị trường chứng khoán, đường cong sốt của bệnh nhân hoặc khả năng tăng tốc cần thiết cho một chiếc xe đua. Một lần nữa chúng tôi muốn có các mô hình có thể xử lý dữ liệu như vậy.

Nói tóm lại, trong khi CNN có thể xử lý hiệu quả thông tin không gian, * mạng thần kinh định kỳ* (RNN) được thiết kế để xử lý thông tin tuần tự tốt hơn. RNNs giới thiệu các biến trạng thái để lưu trữ thông tin quá khứ, cùng với các đầu vào hiện tại, để xác định các đầu ra hiện tại.

Nhiều ví dụ để sử dụng các mạng định kỳ dựa trên dữ liệu văn bản. Do đó, chúng tôi sẽ nhấn mạnh các mô hình ngôn ngữ trong chương này. Sau khi xem xét chính thức hơn về dữ liệu trình tự, chúng tôi giới thiệu các kỹ thuật thực tế để xử lý trước dữ liệu văn bản. Tiếp theo, chúng ta thảo luận về các khái niệm cơ bản về mô hình ngôn ngữ và sử dụng cuộc thảo luận này làm nguồn cảm hứng cho việc thiết kế RNNs. Cuối cùng, chúng tôi mô tả phương pháp tính toán gradient cho RNNs để khám phá các vấn đề có thể gặp phải khi đào tạo các mạng như vậy.

9.1 Mô hình trình tự

Hãy tưởng tượng rằng bạn đang xem phim trên Netflix. Là một người dùng Netflix tốt, bạn quyết định đánh giá từng bộ phim một cách tôn giáo. Rốt cuộc, một bộ phim hay là một bộ phim hay, và bạn muốn xem nhiều hơn trong số họ, phải không? Khi nó quay ra, mọi thứ không hoàn toàn đơn giản như vậy. Ý kiến của mọi người về phim có thể thay đổi khá đáng kể theo thời gian. Trên thực tế, các nhà tâm lý học thậm chí còn có tên cho một số hiệu ứng:

- Có *neo *, dựa trên ý kiến của người khác. Ví dụ, sau lễ trao giải Oscar, xếp hạng cho bộ phim tương ứng tăng lên, mặc dù nó vẫn là cùng một bộ phim. Hiệu ứng này tồn tại trong vài tháng cho đến khi giải thưởng bị lãng quên. Nó đã được chỉ ra rằng hiệu ứng nâng xếp hạng lên hơn nửa điểm (Wu et al., 2017).

- Có sự thích nghi * hedonic*, nơi con người nhanh chóng thích nghi để chấp nhận một tình huống cải thiện hoặc xấu đi như bình thường mới. Ví dụ, sau khi xem nhiều bộ phim hay, kỳ vọng rằng bộ phim tiếp theo cũng tốt hoặc tốt hơn là cao. Do đó, ngay cả một bộ phim trung bình cũng có thể được coi là xấu sau khi nhiều bộ phim tuyệt vời được xem.
- Có * thời gian*. Rất ít người xem thích xem một bộ phim ông già Noel vào tháng 8.
- Trong một số trường hợp, phim trở nên không được ưa chuộng do hành vi sai trái của đạo diễn hoặc diễn viên trong quá trình sản xuất.
- Một số bộ phim trở thành phim đình đám, bởi vì chúng gần như xấu về mặt hài hước. * Plan 9 từ Outer Space* và * Troll 2* đạt được mức độ nổi tiếng cao vì lý do này.

Nói tóm lại, xếp hạng phim là bất cứ điều gì ngoài văn phòng phẩm. Do đó, sử dụng động lực thời gian đã dẫn đến các đề xuất phim chính xác hơn (Koren, 2009). Tất nhiên, dữ liệu trình tự không chỉ là về xếp hạng phim. Sau đây cung cấp thêm hình minh họa.

- Nhiều người dùng có hành vi đặc biệt cao khi nói đến thời điểm họ mở ứng dụng. Ví dụ, các ứng dụng truyền thông xã hội phổ biến hơn nhiều sau giờ học với học sinh. Các ứng dụng giao dịch thị trường chứng khoán được sử dụng phổ biến hơn khi thị trường mở cửa.
- Dự đoán giá cổ phiếu ngày mai khó hơn nhiều so với việc điền vào khoảng trống cho giá cổ phiếu mà chúng tôi đã bỏ lỡ ngày hôm qua, mặc dù cả hai chỉ là vấn đề ước tính một số. Sau khi tất cả, tầm nhìn xa là khó khăn hơn nhiều so với nhận thức sau. Trong thống kê, trước đây (dự đoán vượt ra ngoài các quan sát đã biết) được gọi là * ngoại trị* trong khi sau (ước tính giữa các quan sát hiện có) được gọi là *nội suy*.
- Âm nhạc, lời nói, văn bản và video đều có tính chất tuần tự. Nếu chúng ta hoán vị họ, họ sẽ có ý nghĩa rất ít. Tiêu đề * chó cắn người đàn ông* ít đáng ngạc nhiên hơn nhiều so với * người đàn ông cắn chó*, mặc dù các từ giống hệt nhau.
- Động đất có mối tương quan mạnh mẽ, tức là, sau một trận động đất lớn, rất có thể có một số dư chấn nhỏ hơn, nhiều hơn nhiều so với không có trận động đất mạnh. Trên thực tế, động đất có tương quan về mặt không gian, tức là các dư chấn thường xảy ra trong khoảng thời gian ngắn và ở gần.
- Con người tương tác với nhau trong một bản chất tuần tự, như có thể thấy trong các trận đánh Twitter, mô hình khiêu vũ và các cuộc tranh luận.

9.1.1 Công cụ thống kê

Chúng ta cần các công cụ thống kê và kiến trúc mạng thần kinh sâu mới để xử lý dữ liệu trình tự. Để giữ cho mọi thứ đơn giản, chúng tôi sử dụng giá cổ phiếu (chỉ số FTSE 100) được minh họa trong Fig. 9.1.1 làm ví dụ.

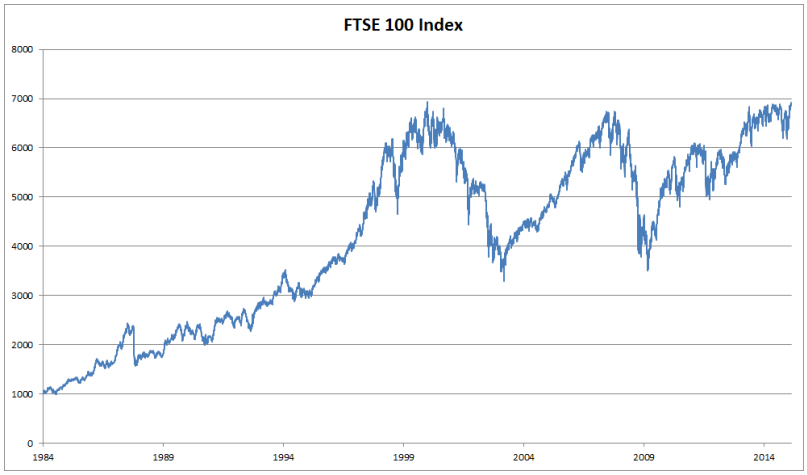


Fig. 9.1.1: FTSE 100 index over about 30 years.

Hãy để chúng tôi biểu thị giá bởi x_t, tức là, tại bước thời gian* $t \in \mathbb{Z}^+$ chúng tôi quan sát giá x_t. Lưu ý rằng đối với chuỗi trong văn bản này, t thường sẽ rời rạc và thay đổi theo số nguyên hoặc tập con của nó. Giả sử rằng một nhà giao dịch muốn làm tốt trên thị trường chứng khoán vào ngày t dự đoán x_t qua

$$x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1). \tag{9.1.1}$$

Mô hình Autoregressive

Để đạt được điều này, nhà giao dịch của chúng tôi có thể sử dụng mô hình hồi quy như mô hình mà chúng tôi đã đào tạo trong Section 4.3. Chỉ có một vấn đề lớn: số lượng đầu vào, $x_{t-1}, \ldots, x_1$ thay đổi, tùy thuộc vào t. Điều đó có nghĩa là, số lượng tăng lên theo lượng dữ liệu mà chúng ta gặp phải và chúng ta sẽ cần một xấp xỉ để làm cho tính toán này có thể truy xuất được. Phần lớn những gì sau trong chương này sẽ xoay quanh cách ước tính $P(x_t \mid x_{t-1}, \ldots, x_1)$ hiệu quả. Tóm lại, nó có hai chiến lược như sau.

Đầu tiên, giả sử rằng chuỗi $x_{t-1}, \ldots, x_1$ có khả năng khá dài là không thực sự cần thiết. Trong trường hợp này, chúng ta có thể tự hài lòng với một số khoảng thời gian có độ dài τ và chỉ sử dụng các quan sát $x_{t-1}, \ldots, x_{t-\tau}$. Lợi ích ngay lập tức là bây giờ số lượng lập luận luôn giống nhau, ít nhất là đối với $t > \tau$. Điều này cho phép chúng tôi đào tạo một mạng sâu như đã chỉ ra ở trên. Các mô hình như vậy sẽ được gọi là mô hình * autoregressive mẫu*, vì chúng khá thực hiện hồi quy trên chính họ.

Chiến lược thứ hai, thể hiện trong Fig. 9.1.2, là giữ một số bản tóm tắt h_t của các quan sát trong quá khứ, và đồng thời cập nhật h_t ngoài dự đoán $\hat{x}_t$. Điều này dẫn đến các mô hình ước tính x_t với $\hat{x}_t = P(x_t \mid h_t)$ và hơn nữa các bản cập nhật của mẫu $h_t = g(h_{t-1}, x_{t-1})$. Vì h_t không bao giờ được quan sát, các mô hình này còn được gọi là * mô hình tự động tiềm mật*.

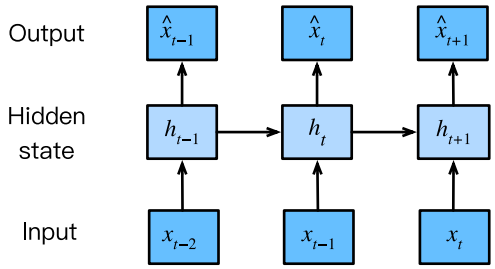


Fig. 9.1.2: A latent autoregressive model.

Cả hai trường hợp đều đặt ra câu hỏi rõ ràng về cách tạo dữ liệu đào tạo. Người ta thường sử dụng các quan sát lịch sử để dự đoán quan sát tiếp theo cho các quan sát đến ngay bây giờ. Rõ ràng chúng tôi không mong đợi thời gian để đứng yên. Tuy nhiên, một giả định phổ biến là trong khi các giá trị cụ thể của x_t có thể thay đổi, ít nhất động lực của dãy tự sẽ không. Điều này là hợp lý, vì động lực mới chỉ là thế, mới lạ và do đó không thể dự đoán được bằng cách sử dụng dữ liệu mà chúng ta có cho đến nay. Các nhà thống kê gọi động lực không thay đổi* đứng yên *. Bất kể những gì chúng tôi làm, do đó chúng tôi sẽ nhận được ước tính của toàn bộ chuỗi thông qua

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}, \ldots, x_1). \tag{9.1.2}$$

Lưu ý rằng những cân nhắc trên vẫn giữ nếu chúng ta đối phó với các đối tượng rời rạc, chẳng hạn như các từ, chứ không phải là số liên tục. Sự khác biệt duy nhất là trong một tình huống như vậy chúng ta cần sử dụng một phân loại chứ không phải là một mô hình hồi quy để ước tính $P(x_t \mid x_{t-1}, \ldots, x_1)$.

Mô hình Markov

Nhớ lại xấp xỉ rằng trong một mô hình autoregressive, chúng tôi chỉ sử dụng $x_{t-1}, \ldots, x_{t-\tau}$ thay vì $x_{t-1}, \ldots, x_1$ để ước tính x_t. Bất cứ khi nào xấp xỉ này là chính xác, chúng tôi nói rằng trình tự đáp ứng một điều kiện * Markov. *Đặc biệt, nếu :math:`tau = 1`, chúng tôi có mô hình Markov* đơn hàng đầu tiên* và $P(x)$ được đưa ra bởi

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{t-1}) \text{ where } P(x_1 \mid x_0) = P(x_1). \tag{9.1.3}$$

Các mô hình như vậy đặc biệt tốt đẹp bất cứ khi nào x_t giả định chỉ là một giá trị rời rạc, vì trong trường hợp này lập trình động có thể được sử dụng để tính toán các giá trị dọc theo chuỗi chính xác. Ví dụ, chúng ta có thể tính toán $P(x_{t+1} \mid x_{t-1})$ một cách hiệu quả:

$$\begin{aligned} P(x_{t+1} \mid x_{t-1}) &= \frac{\sum_{x_t} P(x_{t+1}, x_t, x_{t-1})}{P(x_{t-1})} \\ &= \frac{\sum_{x_t} P(x_{t+1} \mid x_t, x_{t-1}) P(x_t, x_{t-1})}{P(x_{t-1})} \\ &= \sum_{x_t} P(x_{t+1} \mid x_t) P(x_t \mid x_{t-1}) \end{aligned} \tag{9.1.4}$$

bằng cách sử dụng thực tế là chúng ta chỉ cần tính đến một lịch sử rất ngắn về các quan sát trong quá khứ: $P(x_{t+1} \mid x_t, x_{t-1}) = P(x_{t+1} \mid x_t)$. Đi sâu vào chi tiết về lập trình động là vượt quá phạm vi của phần này. Kiểm soát và củng cố các thuật toán học tập sử dụng rộng rãi các công cụ như vậy.

Quan hệ nhân quả

Về nguyên tắc, không có gì sai khi mở ra $P(x_1, \ldots, x_T)$ theo thứ tự ngược lại. Rốt cuộc, bằng cách điều hòa, chúng ta luôn có thể viết nó qua

$$P(x_1, \ldots, x_T) = \prod_{t=T}^{1} P(x_t \mid x_{t+1}, \ldots, x_T). \tag{9.1.5}$$

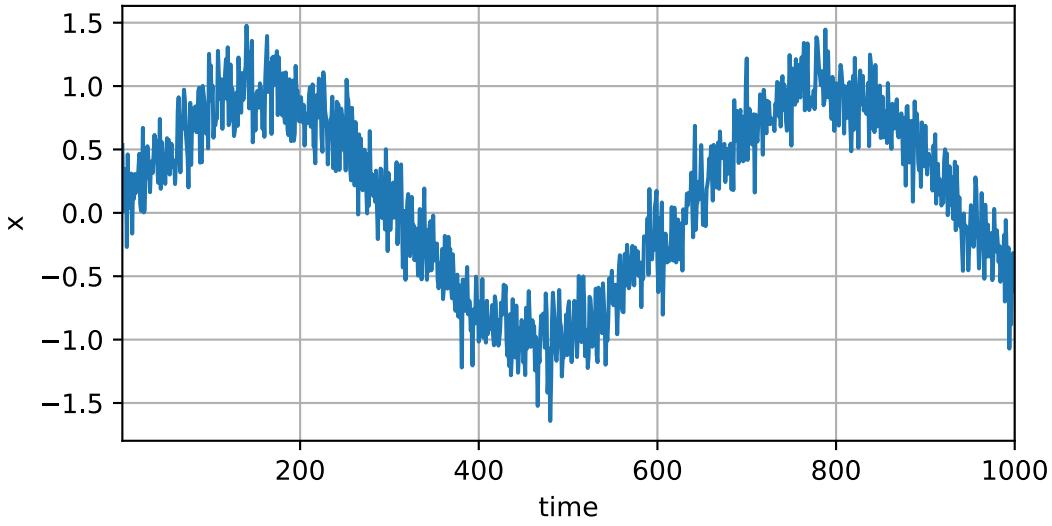
Trong thực tế, nếu chúng ta có một mô hình Markov, chúng ta cũng có thể có được một phân phối xác suất có điều kiện ngược lại. Tuy nhiên, trong nhiều trường hợp, tồn tại một hướng tự nhiên cho dữ liệu, cụ thể là đi về phía trước trong thời gian. Rõ ràng là các sự kiện trong tương lai không thể ảnh hưởng đến quá khứ. Do đó, nếu chúng ta thay đổi x_t, chúng ta có thể ảnh hưởng đến những gì xảy ra cho x_{t+1} trong tương lai nhưng không phải là cuộc trò chuyện. Đó là, nếu chúng ta thay đổi x_t, sự phân phối trong các sự kiện trong quá khứ sẽ không thay đổi. Do đó, nó phải được dễ dàng hơn để giải thích $P(x_{t+1} \mid x_t)$ hơn là $P(x_t \mid x_{t+1})$. Ví dụ, nó đã được chỉ ra rằng trong một số trường hợp, chúng ta có thể tìm thấy $x_{t+1} = f(x_t) + \epsilon$ đối với một số tiếng ồn phụ gia ϵ, trong khi cuộc trò chuyện không đúng (Hoyer et al., 2009). Đây là một tin tuyệt vời, vì nó thường là hướng về phía trước mà chúng tôi quan tâm đến việc ước tính. Cuốn sách của Peters et al. đã giải thích thêm về chủ đề này (Peters et al., 2017a). Chúng tôi hầu như không gãi bề mặt của nó.

9.1.2 Đào tạo

Sau khi xem xét rất nhiều công cụ thống kê, chúng ta hãy thử điều này trong thực tế. Chúng tôi bắt đầu bằng cách tạo ra một số dữ liệu. Để giữ cho mọi thứ đơn giản, chúng tôi tạo ra dữ liệu trình tự của chúng tôi bằng cách sử dụng một hàm sin với một số tiếng ồn phụ gia cho các bước thời gian $1, 2, \ldots, 1000$.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

```
T = 1000  # Generate a total of 1000 points
time = torch.arange(1, T + 1, dtype=torch.float32)
x = torch.sin(0.01 * time) + torch.normal(0, 0.2, (T,))
d2l.plot(time, [x], 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



Tiếp theo, chúng ta cần biến một chuỗi như vậy thành các tính năng và nhãn mà mô hình của chúng ta có thể đào tạo. Dựa trên kích thước nhúng τ, chúng tôi ánh xạ dữ liệu thành các cặp $y_t = x_t$ và $\mathbf{x}_t = [x_{t-\tau}, \ldots, x_{t-1}]$. Người đọc tinh xao có thể nhận thấy rằng điều này cho chúng ta τ ít ví dụ dữ liệu hơn, vì chúng tôi không có đủ lịch sử cho τ đầu tiên trong số chúng. Một sửa chữa đơn giản, đặc biệt nếu trình tự dài, là loại bỏ một vài thuật ngữ đó. Ngoài ra, chúng ta có thể pad chuỗi với số không. Ở đây chúng tôi chỉ sử dụng 600 cặp nhãn tính năng đầu tiên để đào tạo.

```
tau = 4
features = torch.zeros((T - tau, tau))
for i in range(tau):
    features[:, i] = x[i: T - tau + i]
labels = x[tau:].reshape((-1, 1))
```

```
batch_size, n_train = 16, 600
# Only the first `n_train` examples are used for training
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
```

Ở đây chúng ta giữ kiến trúc khá đơn giản: chỉ một MLP với hai lớp được kết nối hoàn toàn, kích hoạt ReLU và tổn thất bình phương.

```
# Function for initializing the weights of the network
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)

# A simple MLP
def get_net():
    net = nn.Sequential(nn.Linear(4, 10),
                        nn.ReLU(),
                        nn.Linear(10, 1))
    net.apply(init_weights)
    return net

# Note: `MSELoss` computes squared error without the 1/2 factor
loss = nn.MSELoss(reduction='none')
```

Bây giờ chúng tôi đã sẵn sàng để đào tạo mô hình. Mã dưới đây về cơ bản là giống hệt với vòng đào tạo trong các phần trước, chẳng hạn như Section 4.3. Do đó, chúng tôi sẽ không đi sâu vào nhiều chi tiết.

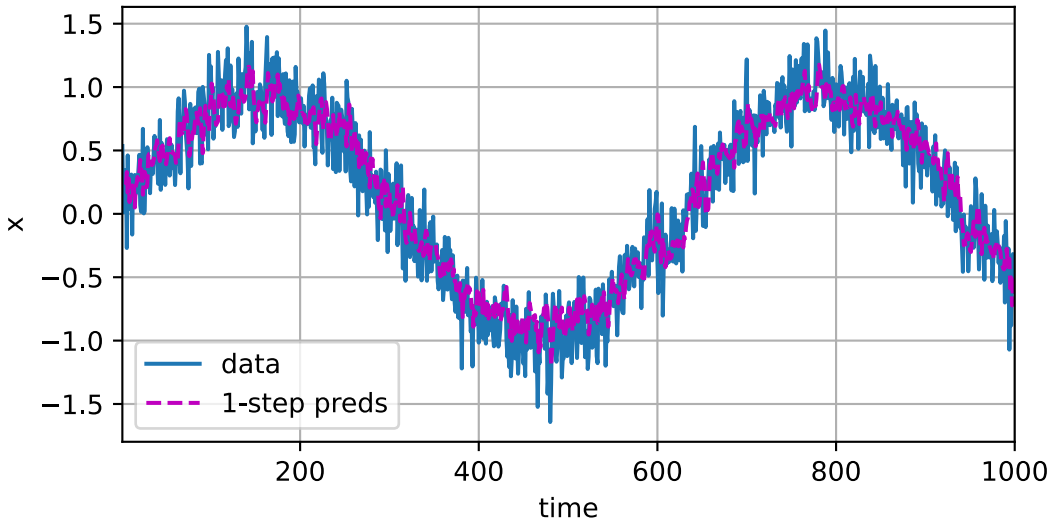
```
def train(net, train_iter, loss, epochs, lr):
    trainer = torch.optim.Adam(net.parameters(), lr)
    for epoch in range(epochs):
        for X, y in train_iter:
            trainer.zero_grad()
            l = loss(net(X), y)
            l.sum().backward()
            trainer.step()
        print(f'epoch {epoch + 1}, '
              f'loss: {d2l.evaluate_loss(net, train_iter, loss):f}')

net = get_net()
train(net, train_iter, loss, 5, 0.01)
```

```
epoch 1, loss: 0.062616
epoch 2, loss: 0.054403
epoch 3, loss: 0.054915
epoch 4, loss: 0.052986
epoch 5, loss: 0.053715
```

9.1.3 Prediction

Vì sự mất mát đào tạo là nhỏ, chúng tôi mong đợi mô hình của chúng tôi sẽ hoạt động tốt. Hãy để chúng tôi xem điều này có nghĩa là gì trong thực tế. Điều đầu tiên cần kiểm tra là mô hình có thể dự đoán điều gì xảy ra chỉ trong bước thời gian tới, cụ thể là dự đoán * một bước trước *.

```
onestep_preds = net(features)
d2l.plot([time, time[tau:]], [x.detach().numpy(), onestep_preds.detach().
↪numpy()], 'time',
         'x', legend=['data', '1-step preds'], xlim=[1, 1000], figsize=(6, 3))
```



Các dự đoán một bước trước trông đẹp, giống như chúng tôi mong đợi. Thậm chí vượt quá 604 (`n_train + tau`) quan sát các dự đoán vẫn trông đáng tin cậy. Tuy nhiên, chỉ có một vấn đề nhỏ đối với điều này: nếu chúng ta chỉ quan sát dữ liệu trình tự cho đến bước thời gian 604, chúng ta không thể hy vọng nhận được các đầu vào cho tất cả các dự đoán trước một bước trong tương lai. Thay vào đó, chúng ta cần phải làm việc theo cách của chúng tôi về phía trước một bước tại một thời điểm:

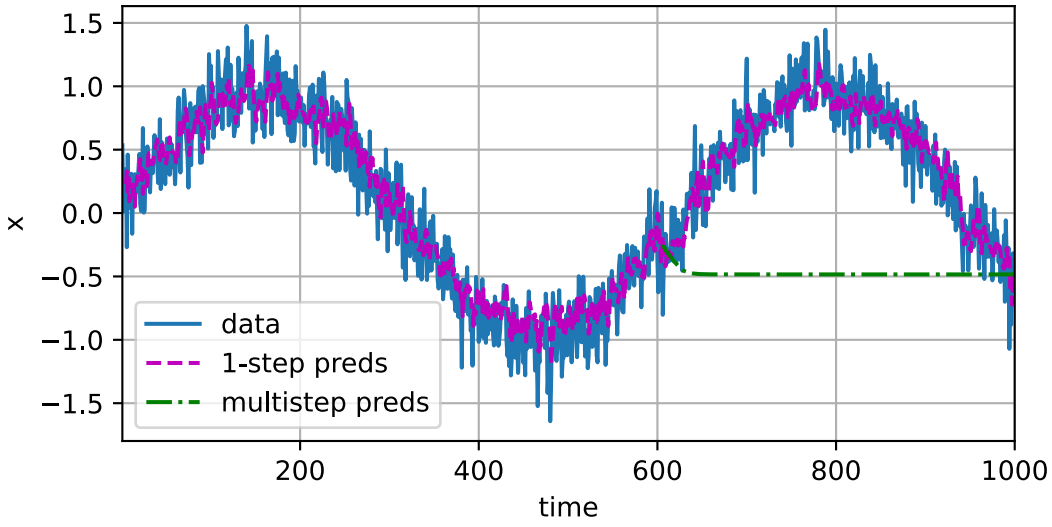
$$\begin{aligned}
\hat{x}_{605} &= f(x_{601}, x_{602}, x_{603}, x_{604}),\\
\hat{x}_{606} &= f(x_{602}, x_{603}, x_{604}, \hat{x}_{605}),\\
\hat{x}_{607} &= f(x_{603}, x_{604}, \hat{x}_{605}, \hat{x}_{606}),\\
\hat{x}_{608} &= f(x_{604}, \hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}),\\
\hat{x}_{609} &= f(\hat{x}_{605}, \hat{x}_{606}, \hat{x}_{607}, \hat{x}_{608}),\\
&\ldots
\end{aligned} \tag{9.1.6}$$

Nói chung, đối với một chuỗi quan sát lên đến x_t, sản lượng dự đoán của nó $\hat{x}_{t+k}$ tại bước thời điểm $t+k$ được gọi là k* -bước trước dự đoán *. Vì chúng tôi đã quan sát đến x_{604}, dự đoán k-bước trước của nó là

$\hat{x}_{604+k}$. Nói cách khác, chúng ta sẽ phải sử dụng dự đoán của riêng mình để đưa ra dự đoán nhiều bước trước. Hãy để chúng tôi xem điều này diễn ra tốt như thế nào.

```
multistep_preds = torch.zeros(T)
multistep_preds[: n_train + tau] = x[: n_train + tau]
for i in range(n_train + tau, T):
    multistep_preds[i] = net(
        multistep_preds[i - tau:i].reshape((1, -1)))
```

```
d2l.plot([time, time[tau:], time[n_train + tau:]],
         [x.detach().numpy(), onestep_preds.detach().numpy(),
          multistep_preds[n_train + tau:].detach().numpy()], 'time',
         'x', legend=['data', '1-step preds', 'multistep preds'],
         xlim=[1, 1000], figsize=(6, 3))
```



Như ví dụ trên cho thấy, đây là một thất bại ngoạn mục. Các dự đoán phân rã thành một hằng số khá nhanh sau một vài bước dự đoán. Tại sao thuật toán hoạt động rất kém? Điều này cuối cùng là do thực tế là các lỗi xây dựng. Chúng ta hãy nói rằng sau bước 1 chúng ta có một số lỗi $\epsilon_1 = \bar{\epsilon}$. Bây giờ *đầu vào* cho bước 2 bị ảnh hưởng bởi ϵ_1, do đó chúng tôi bị một số lỗi theo thứ tự $\epsilon_2 = \bar{\epsilon} + c\epsilon_1$ đối với một số hằng số c, v.v. Lỗi có thể phân kỳ khá nhanh từ các quan sát thực sự. Đây là một hiện tượng phổ biến. Ví dụ, dự báo thời tiết trong 24 giờ tới có xu hướng khá chính xác nhưng ngoài đó độ chính xác giảm nhanh chóng. Chúng tôi sẽ thảo luận về các phương pháp để cải thiện điều này trong suốt chương này và hơn thế nữa.

Hãy để chúng tôi xem xét kỹ hơn những khó khăn trong k-bước trước dự đoán bằng cách tính toán dự đoán trên toàn bộ chuỗi cho $k = 1, 4, 16, 64$.

```
max_steps = 64
```

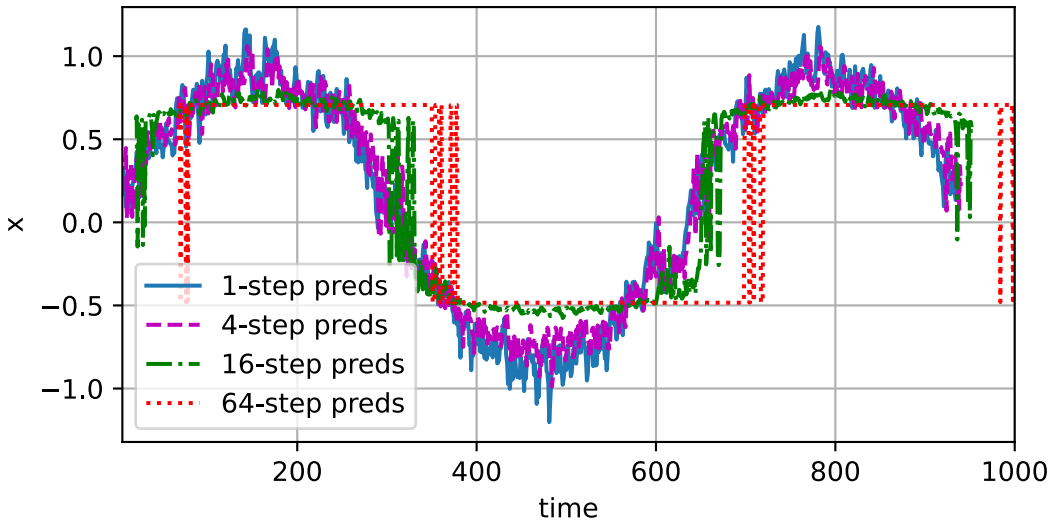
```
features = torch.zeros((T - tau - max_steps + 1, tau + max_steps))
# Column `i` (`i` < `tau`) are observations from `x` for time steps from
# `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau):
    features[:, i] = x[i: i + T - tau - max_steps + 1]
```

(continues on next page)

(continued from previous page)

```
# Column `i` (`i` >= `tau`) are the (`i - tau + 1`)-step-ahead predictions for
# time steps from `i + 1` to `i + T - tau - max_steps + 1`
for i in range(tau, tau + max_steps):
    features[:, i] = net(features[:, i - tau:i]).reshape(-1)
```

```
steps = (1, 4, 16, 64)
d2l.plot([time[tau + i - 1: T - max_steps + i] for i in steps],
         [features[:, (tau + i - 1)].detach().numpy() for i in steps], 'time',
↪ 'x',
         legend=[f'{i}-step preds' for i in steps], xlim=[5, 1000],
         figsize=(6, 3))
```



Điều này minh họa rõ ràng chất lượng dự đoán thay đổi như thế nào khi chúng ta cố gắng dự đoán thêm vào tương lai. Trong khi các dự đoán 4 bước trước vẫn trông tốt, nhưng bất cứ điều gì ngoài đó là gần như vô dụng.

9.1.4 Tóm tắt

- Có khá một sự khác biệt về khó khăn giữa nội suy và ngoại suy. Do đó, nếu bạn có một chuỗi, hãy luôn tôn trọng thứ tự thời gian của dữ liệu khi đào tạo, tức là, không bao giờ đào tạo về dữ liệu trong tương lai.
- Các mô hình trình tự yêu cầu các công cụ thống kê chuyên dụng để ước tính. Hai lựa chọn phổ biến là mô hình autoregressive và các mô hình autoregressive biến trễ.
- Đối với các mô hình nhân quả (ví dụ: thời gian đi về phía trước), ước tính hướng tiến thường dễ dàng hơn rất nhiều so với hướng ngược lại.
- Đối với một chuỗi quan sát đến thời gian bước t, đầu ra dự đoán của nó tại bước thời gian $t + k$ là k* -bước trước dự đoán *. Như chúng ta dự đoán thêm trong thời gian bằng cách tăng k, các lỗi tích lũy và chất lượng dự đoán suy giảm, thường đáng kể.

9.1.5 Bài tập

1. Cải thiện mô hình trong thí nghiệm của phần này.
    1. Kết hợp nhiều hơn 4 quan sát trong quá khứ? Bạn thực sự cần bao nhiêu?
    2. Bạn sẽ cần bao nhiêu quan sát trong quá khứ nếu không có tiếng ồn? Gợi ý: bạn có thể viết sin và cos dưới dạng phương trình vi phân.
    3. Bạn có thể kết hợp các quan sát cũ hơn trong khi vẫn giữ tổng số tính năng không đổi? Điều này có cải thiện độ chính xác không? Tại sao?
    4. Thay đổi kiến trúc mạng thần kinh và đánh giá hiệu suất.
2. Một nhà đầu tư muốn tìm một bảo mật tốt để mua. Anh ta nhìn vào sự trở lại trong quá khứ để quyết định cái nào có khả năng làm tốt. Điều gì có thể xảy ra sai với chiến lược này?
3. Liệu nhân quả cũng áp dụng cho văn bản? Đến mức nào?
4. Đưa ra một ví dụ cho khi một mô hình tự hồi quy tiềm ẩn có thể cần thiết để nắm bắt động của dữ liệu.

Discussions[98]

9.2 Xử lý sơ bộ văn bản

Chúng tôi đã xem xét và đánh giá các công cụ thống kê và các thách thức dự đoán đối với dữ liệu trình tự. Dữ liệu như vậy có thể có nhiều hình thức. Cụ thể, như chúng ta sẽ tập trung vào nhiều chương của cuốn sách, văn bản là một trong những ví dụ phổ biến nhất về dữ liệu trình tự. Ví dụ, một bài viết có thể được xem đơn giản là một chuỗi các từ, hoặc thậm chí là một chuỗi các ký tự. Để tạo điều kiện cho các thí nghiệm trong tương lai của chúng tôi với dữ liệu trình tự, chúng tôi sẽ dành phần này để giải thích các bước xử lý tiền xử lý phổ biến cho văn bản. Thông thường, các bước này là:

1. Tải văn bản dưới dạng chuỗi vào bộ nhớ.
2. Chia chuỗi thành mã thông báo (ví dụ: từ và ký tự).
3. Xây dựng một bảng từ vựng để ánh xạ các token chia thành các chỉ số số.
4. Chuyển đổi văn bản thành chuỗi các chỉ số số để chúng có thể được thao tác bởi các mô hình một cách dễ dàng.

```
import collections
import re
from d2l import torch as d2l
```

[98] https://discuss.d2l.ai/t/114

9.2.1 Đọc tập dữ liệu

Để bắt đầu, chúng tôi tải văn bản từ H G Wells' *The Time Machine*[99]. Đây là một cơ thể khá nhỏ chỉ hơn 30000 từ, nhưng với mục đích của những gì chúng tôi muốn minh họa điều này là tốt. Các bộ sưu tập tài liệu thực tế hơn chứa nhiều hàng tỷ từ. Hàm sau đọc tập dữ liệu vào danh sách các dòng văn bản, trong đó mỗi dòng là một chuỗi. Để đơn giản, ở đây chúng ta bỏ qua dấu câu và viết hoa.

```
#@save
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                '090b5e7e70c295757f55df93cb0a180b9691891a')

def read_time_machine():  #@save
    """Load the time machine dataset into a list of text lines."""
    with open(d2l.download('time_machine'), 'r') as f:
        lines = f.readlines()
    return [re.sub('[^A-Za-z]+', ' ', line).strip().lower() for line in lines]

lines = read_time_machine()
print(f'# text lines: {len(lines)}')
print(lines[0])
print(lines[10])
```

```
# text lines: 3221
the time machine by h g wells
twinkled and his usually pale face was flushed and animated the
```

9.2.2 Mã hóa

Hàm `tokenize` sau lấy một danh sách (`lines`) làm đầu vào, trong đó mỗi phần tử là một chuỗi văn bản (ví dụ, một dòng văn bản). Mỗi chuỗi văn bản được chia thành danh sách các tokens. Một * token* là đơn vị cơ bản trong văn bản. Cuối cùng, một danh sách các danh sách token được trả về, trong đó mỗi mã thông báo là một chuỗi.

```
def tokenize(lines, token='word'):  #@save
    """Split text lines into word or character tokens."""
    if token == 'word':
        return [line.split() for line in lines]
    elif token == 'char':
        return [list(line) for line in lines]
    else:
        print('ERROR: unknown token type: ' + token)

tokens = tokenize(lines)
for i in range(11):
    print(tokens[i])
```

```
['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
[]
[]
[]
```

(continues on next page)

[99] http://www.gutenberg.org/ebooks/35

(continued from previous page)

```
[]
['i']
[]
[]
['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be', 'convenient',
↪'to', 'speak', 'of', 'him']
['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his', 'grey',
↪'eyes', 'shone', 'and']
['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed', 'and',
↪ 'animated', 'the']
```

9.2.3 Từ vựng

Loại chuỗi của mã thông báo bất tiện khi được sử dụng bởi các mô hình, lấy đầu vào số. Bây giờ chúng ta hãy xây dựng một từ điển, thường được gọi là *vocabulary* là tốt, để ánh xạ chuỗi mã thông báo thành các chỉ số số bắt đầu từ 0. Để làm như vậy, trước tiên chúng ta đếm các mã thông báo duy nhất trong tất cả các tài liệu từ bộ đào tạo, cụ thể là một * corpus*, sau đó gán một chỉ số số cho mỗi mã thông báo duy nhất theo tần số của nó. Hiếm khi xuất hiện thẻ thường được loại bỏ để giảm sự phức tạp. Bất kỳ mã thông báo nào không tồn tại trong corpus hoặc đã được gỡ bỏ đều được ánh xạ thành một mã thông báo không xác định đặc biệt “”. Chúng tôi tùy chọn thêm một danh sách các token dành riêng, chẳng hạn như “” cho padding, “” để trình bày phần đầu cho một chuỗi, và “” cho phần cuối của một chuỗi.

```
class Vocab:  #@save
    """Vocabulary for text."""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                   reverse=True)
        # The index for the unknown token is 0
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]
```

(continues on next page)

(continued from previous page)

```
    def to_tokens(self, indices):
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]

    @property
    def unk(self):  # Index for the unknown token
        return 0

    @property
    def token_freqs(self):  # Index for the unknown token
        return self._token_freqs

def count_corpus(tokens):  #@save
    """Count token frequencies."""
    # Here `tokens` is a 1D list or 2D list
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # Flatten a list of token lists into a list of tokens
        tokens = [token for line in tokens for token in line]
    return collections.Counter(tokens)
```

Chúng tôi xây dựng một từ vừ bằng cách sử dụng tập dữ liệu máy thời gian làm cơ thể. Sau đó, chúng tôi in vài mã thông báo thường xuyên đầu tiên với các chỉ số của chúng.

```
vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[:10])
```

```
[('<unk>', 0), ('the', 1), ('i', 2), ('and', 3), ('of', 4), ('a', 5), ('to',␣
↪6), ('was', 7), ('in', 8), ('that', 9)]
```

Bây giờ chúng ta có thể chuyển đổi mỗi dòng văn bản thành một danh sách các chỉ số.

```
for i in [0, 10]:
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])
```

```
words: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
indices: [1, 19, 50, 40, 2183, 2184, 400]
words: ['twinkled', 'and', 'his', 'usually', 'pale', 'face', 'was', 'flushed',
↪ 'and', 'animated', 'the']
indices: [2186, 3, 25, 1044, 362, 113, 7, 1421, 3, 1045, 1]
```

9.2.4 Đặt tất cả mọi thứ lại với nhau

Sử dụng các hàm trên, chúng ta đóng gói mọi thứ vào hàm `load_corpus_time_machine`, trả về `corpus`, danh sách các chỉ số token và `vocab`, từ vựng của cơ thể máy thời gian. Các sửa đổi chúng tôi đã làm ở đây là: (i) chúng tôi mã hóa văn bản thành các ký tự, không phải từ ngữ, để đơn giản hóa việc đào tạo trong các phần sau; (ii) `corpus` là một danh sách duy nhất, không phải là danh sách các danh sách mã thông báo, vì mỗi dòng văn bản trong tập dữ liệu máy thời gian không nhất thiết phải là một câu hoặc một đoạn văn.

```
def load_corpus_time_machine(max_tokens=-1):  #@save
    """Return token indices and the vocabulary of the time machine dataset."""
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    # Since each text line in the time machine dataset is not necessarily a
    # sentence or a paragraph, flatten all the text lines into a single list
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)
```

```
(170580, 28)
```

9.2.5 Tóm tắt

- Văn bản là một hình thức quan trọng của dữ liệu trình tự.
- Để xử lý trước văn bản, chúng ta thường chia văn bản thành token, xây dựng một từ vựng để ánh xạ chuỗi token thành các chỉ số số, và chuyển đổi dữ liệu văn bản thành các chỉ số token cho các mô hình thao tác.

9.2.6 Bài tập

1. Token ization là một bước tiền xử lý chính. Nó thay đổi cho các ngôn ngữ khác nhau. Cố gắng tìm ba phương pháp thường được sử dụng khác để mã hóa văn bản.
2. Trong thí nghiệm của phần này, mã hóa văn bản thành các từ và thay đổi các đối số `min_freq` của trường hợp `Vocab`. Điều này ảnh hưởng đến kích thước từ vựng như thế nào?

Discussions[100]

[100] https://discuss.d2l.ai/t/115

9.3 Mô hình ngôn ngữ và bộ dữ liệu

Trong Section 9.2, chúng ta thấy cách ánh xạ dữ liệu văn bản thành mã thông báo, trong đó các mã thông báo này có thể được xem như một chuỗi các quan sát rời rạc, chẳng hạn như từ hoặc ký tự. Giả sử rằng các thẻ trong một chuỗi văn bản chiều dài T lần lượt $x_1, x_2, \ldots, x_T$. Sau đó, trong chuỗi văn bản, x_t $(1 \leq t \leq T)$ có thể được coi là quan sát hoặc nhãn tại bước thời gian t. Với một chuỗi văn bản như vậy, mục tiêu của mô hình ngôn ngữ* là ước tính xác suất chung của chuỗi

$$P(x_1, x_2, \ldots, x_T). \tag{9.3.1}$$

Mô hình ngôn ngữ cực kỳ hữu ích. Ví dụ, một mô hình ngôn ngữ lý tưởng sẽ có thể tự tạo văn bản tự nhiên, chỉ cần vẽ một mã thông báo tại một thời điểm $x_t \sim P(x_t \mid x_{t-1}, \ldots, x_1)$. Không giống như con khỉ sử dụng một máy đánh chữ, tất cả các văn bản nổi lên từ một mô hình như vậy sẽ truyền như ngôn ngữ tự nhiên, ví dụ, văn bản tiếng Anh. Hơn nữa, nó sẽ là đủ để tạo ra một hộp thoại có ý nghĩa, chỉ bằng cách điều chỉnh văn bản trên các đoạn hộp thoại trước đó. Rõ ràng chúng ta vẫn còn rất xa so với việc thiết kế một hệ thống như vậy, vì nó sẽ cần phải * hiểu văn bản chứ không phải chỉ tạo ra nội dung hợp lý ngữ pháp.

Tuy nhiên, các mô hình ngôn ngữ có dịch vụ tuyệt vời ngay cả ở dạng hạn chế của chúng. Ví dụ, các cụm từ "để nhận ra lời nói" và "để phá hỏng một bãi biển đẹp" nghe rất giống nhau. Điều này có thể gây ra sự mơ hồ trong nhận dạng giọng nói, dễ dàng giải quyết thông qua một mô hình ngôn ngữ từ chối bản dịch thứ hai là kỳ lạ. Tương tự như vậy, trong một thuật toán tóm tắt tài liệu, đáng để biết rằng "chó cắn người đàn ông" thường xuyên hơn nhiều so với "người đàn ông cắn chó", hoặc "Tôi muốn ăn bà" là một tuyên bố khá đáng lo ngại, trong khi "Tôi muốn ăn, bà" lành tính hơn nhiều.

9.3.1 Học một mô hình ngôn ngữ

Câu hỏi rõ ràng là làm thế nào chúng ta nên mô hình hóa một tài liệu, hoặc thậm chí là một chuỗi các mã thông báo. Giả sử rằng chúng ta mã hóa dữ liệu văn bản ở cấp độ từ. Chúng tôi có thể truy tìm phân tích mà chúng tôi áp dụng cho các mô hình trình tự trong Section 9.1. Hãy để chúng tôi bắt đầu bằng cách áp dụng các quy tắc xác suất cơ bản:

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}). \tag{9.3.2}$$

Ví dụ, xác suất của một chuỗi văn bản chứa bốn từ sẽ được đưa ra là:

$$P(\text{deep}, \text{learning}, \text{is}, \text{fun}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}). \tag{9.3.3}$$

Để tính toán mô hình ngôn ngữ, chúng ta cần tính xác suất của các từ và xác suất có điều kiện của một từ được đưa ra vài từ trước đó. Xác suất như vậy về cơ bản là các tham số mô hình ngôn ngữ.

Ở đây, chúng tôi giả định rằng tập dữ liệu đào tạo là một cơ thể văn bản lớn, chẳng hạn như tất cả các mục Wikipedia, Project Gutenberg[101] và tất cả văn bản được đăng trên Web. Xác suất của các từ có thể được tính từ tần số từ tương đối của một từ nhất định trong tập dữ liệu đào tạo. Ví dụ, ước tính $\hat{P}(\text{deep})$ có thể được tính là xác suất của bất kỳ câu nào bắt đầu bằng từ "sâu". Một cách tiếp cận ít chính xác hơn một chút sẽ là đếm tất cả các lần xuất hiện của từ "sâu" và chia nó cho tổng số từ trong corpus. Điều này hoạt động khá tốt, đặc biệt là đối với các từ thường xuyên. Moving Di chuyển on, we could attempt cố gắng to estimate ước tính

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})}, \tag{9.3.4}$$

[101] https://en.wikipedia.org/wiki/Project_Gutenberg

trong đó $n(x)$ và $n(x, x')$ là số lần xuất hiện của singletons và các cặp từ liên tiếp, tương ứng. Thật không may, ước tính xác suất của một cặp từ có phần khó khăn hơn, vì sự xuất hiện của "học sâu" ít thường xuyên hơn rất nhiều. Đặc biệt, đối với một số kết hợp từ bất thường, có thể khó tìm đủ lần xuất hiện để có được ước tính chính xác. Mọi thứ thay đổi cho tồi tệ hơn cho các kết hợp ba từ và hơn thế nữa. Sẽ có nhiều kết hợp ba từ hợp lý mà chúng ta có thể sẽ không thấy trong tập dữ liệu của mình. Trừ khi chúng tôi cung cấp một số giải pháp để gán các kết hợp từ như vậy không đếm, chúng tôi sẽ không thể sử dụng chúng trong một mô hình ngôn ngữ. Nếu tập dữ liệu nhỏ hoặc nếu các từ rất hiếm, chúng ta có thể không tìm thấy ngay cả một trong số chúng.

Một chiến lược phổ biến là thực hiện một số hình thức của * Laplace smooth*. Giải pháp là thêm một hằng số nhỏ cho tất cả các số đếm. Biểu thị bằng n tổng số từ trong bộ đào tạo và m số từ duy nhất. Giải pháp này giúp với singletons, ví dụ, thông qua

$$\begin{aligned} \hat{P}(x) &= \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\ \hat{P}(x' \mid x) &= \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\ \hat{P}(x'' \mid x, x') &= \frac{n(x, x', x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}. \end{aligned} \tag{9.3.5}$$

Ở đây ϵ_1, ϵ_2, và ϵ_3 là siêu tham số. Lấy ϵ_1 làm ví dụ: khi $\epsilon_1 = 0$, không áp dụng làm mịn; khi ϵ_1 tiếp cận vô cực dương, $\hat{P}(x)$ tiếp cận xác suất thống nhất $1/m$. Trên đây là một biến thể khá nguyên thủy của những kỹ thuật khác có thể thực hiện (Wood et al., 2011).

Thật không may, các mô hình như thế này trở nên khó sử dụng khá nhanh vì những lý do sau. Đầu tiên, chúng ta cần lưu trữ tất cả các số lượng. Thứ hai, điều này hoàn toàn bỏ qua ý nghĩa của các từ. Ví dụ, "mèo" và "mèo" sẽ xảy ra trong các bối cảnh liên quan. Khá khó để điều chỉnh các mô hình như vậy thành các bối cảnh bổ sung, trong khi, các mô hình ngôn ngữ dựa trên học tập sâu rất phù hợp để tính đến điều này. Cuối cùng, chuỗi từ dài gần như chắc chắn là mới lạ, do đó một mô hình chỉ đơn giản là đếm tần suất của các chuỗi từ đã thấy trước đó bị ràng buộc để thực hiện kém ở đó.

9.3.2 Mô hình Markov và n-gram

Trước khi thảo luận về các giải pháp liên quan đến học sâu, chúng ta cần thêm một số thuật ngữ và khái niệm. Nhớ lại cuộc thảo luận của chúng tôi về các mô hình Markov trong Section 9.1. Hãy để chúng tôi áp dụng điều này cho mô hình ngôn ngữ. Một phân phối trên chuỗi thỏa mãn thuộc tính Markov của thứ tự đầu tiên nếu $P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$. Đơn hàng cao hơn tương ứng với các phụ thuộc dài hơn. Điều này dẫn đến một số xấp xỉ mà chúng ta có thể áp dụng cho mô hình một chuỗi:

$$\begin{aligned} P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3), \\ P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3). \end{aligned} \tag{9.3.6}$$

Các công thức xác suất liên quan đến một, hai và ba biến thường được gọi là mô hình *unigram*, *bigram* và * trigram*, tương ứng. Sau đây, chúng ta sẽ học cách thiết kế các mô hình tốt hơn.

9.3.3 Natural Language Thống kê

Hãy để chúng tôi xem làm thế nào điều này hoạt động trên dữ liệu thực. Chúng tôi xây dựng một từ vựng dựa trên tập dữ liệu cỗ máy thời gian như được giới thiệu trong Section 9.2 và in top 10 từ thường xuyên nhất.

```
import random
import torch
from d2l import torch as d2l
```

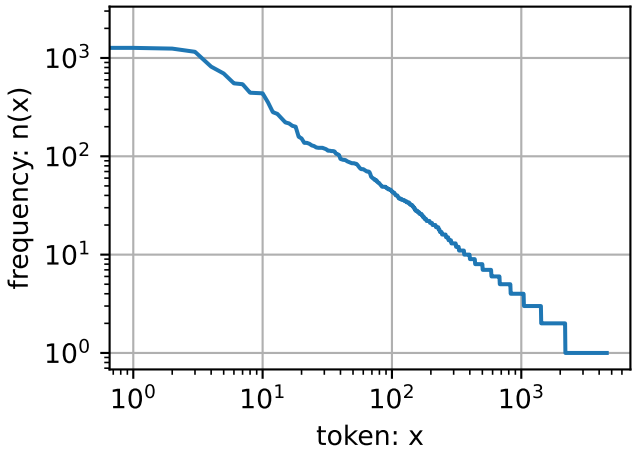
```
tokens = d2l.tokenize(d2l.read_time_machine())
# Since each text line is not necessarily a sentence or a paragraph, we
# concatenate all text lines
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
vocab.token_freqs[:10]
```

```
Downloading ../data/timemachine.txt from http://d2l-data.s3-accelerate.
↪amazonaws.com/timemachine.txt...
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

Như chúng ta có thể thấy, những từ phổ biến nhất là thực sự khá nhàm chán khi nhìn vào. Chúng thường được gọi là * stop words* và do đó lọc ra. Tuy nhiên, chúng vẫn mang ý nghĩa và chúng tôi vẫn sẽ sử dụng chúng. Bên cạnh đó, khá rõ ràng rằng tần số từ phân rã khá nhanh. Từ 10^{th} thường xuyên nhất là ít hơn $1/5$ phổ biến như từ phổ biến nhất. Để có được một ý tưởng tốt hơn, chúng tôi vẽ hình của tần số từ .

```
freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
```



Chúng tôi đang ở trên một cái gì đó khá cơ bản ở đây: tần số từ phân rã nhanh chóng theo một cách rõ ràng. Sau khi xử lý một vài từ đầu tiên là ngoại lệ, tất cả các từ còn lại gần như theo một đường thẳng trên một âm mưu log-log. Điều này có nghĩa là các từ thỏa mãn định luật *Zipf*, trong đó tuyên bố rằng tần số n_i của từ i^{th} thường xuyên nhất là:

$$n_i \propto \frac{1}{i^\alpha}, \tag{9.3.7}$$

which mà is equivalent tương đương to

$$\log n_i = -\alpha \log i + c, \tag{9.3.8}$$

trong đó α là số mũ đặc trưng cho sự phân bố và c là một hằng số. Điều này sẽ cho chúng ta tạm dừng nếu chúng ta muốn mô hình hóa các từ bằng cách đếm số liệu thống kê và làm mịn. Rốt cuộc, chúng ta sẽ đánh giá quá cao đáng kể tần số của đuôi, còn được gọi là những từ không thường xuyên. Nhưng những gì về các kết hợp từ khác, chẳng hạn như bigrams, trigrams, và hơn thế nữa? Chúng ta hãy xem liệu tần số bigram có hoạt động theo cách tương tự như tần số unigram hay không.

```
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
bigram_vocab.token_freqs[:10]
```

```
[(('of', 'the'), 309),
 (('in', 'the'), 169),
 (('i', 'had'), 130),
 (('i', 'was'), 112),
 (('and', 'the'), 109),
 (('the', 'time'), 102),
 (('it', 'was'), 99),
 (('to', 'the'), 85),
 (('as', 'i'), 78),
 (('of', 'a'), 73)]
```

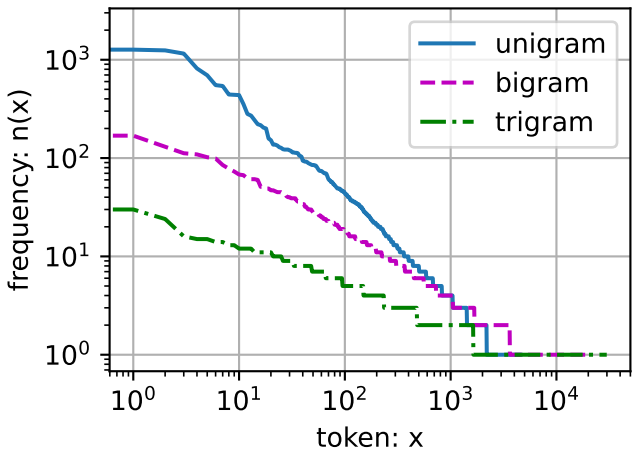
Một điều đáng chú ý ở đây. Trong số mười cặp từ thường xuyên nhất, chín cặp được cấu tạo từ cả hai từ dừng và chỉ có một cặp có liên quan đến cuốn sách thực tế—"thời gian". Hơn nữa, chúng ta hãy xem liệu tần số trigram có hoạt động theo cùng một cách hay không.

```
trigram_tokens = [triple for triple in zip(
    corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
trigram_vocab.token_freqs[:10]
```

```
[(('the', 'time', 'traveller'), 59),
 (('the', 'time', 'machine'), 30),
 (('the', 'medical', 'man'), 24),
 (('it', 'seemed', 'to'), 16),
 (('it', 'was', 'a'), 15),
 (('here', 'and', 'there'), 15),
 (('seemed', 'to', 'me'), 14),
 (('i', 'did', 'not'), 14),
 (('i', 'saw', 'the'), 13),
 (('i', 'began', 'to'), 13)]
```

Cuối cùng, chúng ta hãy hình dung tần số token trong số ba mô hình này: unigrams, bigram và trigram.

```
bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```



Con số này khá thú vị vì một số lý do. Đầu tiên, ngoài các từ unigram, chuỗi các từ cũng dường như tuân theo định luật Zipf, mặc dù với số mũ nhỏ hơn α trong (9.3.7), tùy thuộc vào độ dài trình tự. Thứ hai, số lượng n-gram khác biệt không phải là lớn. Điều này cho chúng ta hy vọng rằng có khá nhiều cấu trúc trong ngôn ngữ. Thứ ba, nhiều n-gram xảy ra rất hiếm khi, khiến Laplace làm mịn khá không phù hợp với mô hình ngôn ngữ. Thay vào đó, chúng tôi sẽ sử dụng các mô hình dựa trên học tập sâu.

9.3.4 Đọc dữ liệu trình tự dài

Vì dữ liệu trình tự là theo bản chất tuần tự của chúng, chúng ta cần giải quyết vấn đề xử lý nó. Chúng tôi đã làm như vậy một cách khá đặc biệt trong Section 9.1. Khi chuỗi quá lâu để được xử lý bởi các mô hình cùng một lúc, chúng tôi có thể muốn chia các chuỗi như vậy để đọc. Bây giờ chúng ta hãy mô tả các chiến lược chung. Trước khi giới thiệu mô hình, chúng ta hãy giả định rằng chúng ta sẽ sử dụng mạng thần kinh để đào tạo một mô hình ngôn ngữ, trong đó mạng xử lý một loạt các chuỗi nhỏ với độ dài được xác định trước, giả sử các bước thời gian n, tại một thời điểm. Bây giờ câu hỏi là làm thế nào để đọc minibatches của các tính năng và nhãn một cách ngẫu nhiên.

Để bắt đầu, vì một chuỗi văn bản có thể dài tùy ý, chẳng hạn như toàn bộ cuốn sách * Máy thời gian*, chúng ta có thể phân vùng một chuỗi dài như vậy thành các dãy con với cùng một số bước thời gian. Khi đào tạo mạng thần kinh của chúng tôi, một minibatch của các dãy con như vậy sẽ được đưa vào mô hình. Giả sử rằng mạng xử lý một dãy con n bước thời gian tại một thời điểm. Fig. 9.3.1 cho thấy tất cả các cách khác nhau để có được dãy con từ một chuỗi văn bản gốc, trong đó $n = 5$ và một mã thông báo tại mỗi bước thời gian tương ứng với một ký tự. Lưu ý rằng chúng ta có khá tự do vì chúng ta có thể chọn một phần bù tùy ý cho biết vị trí ban đầu.

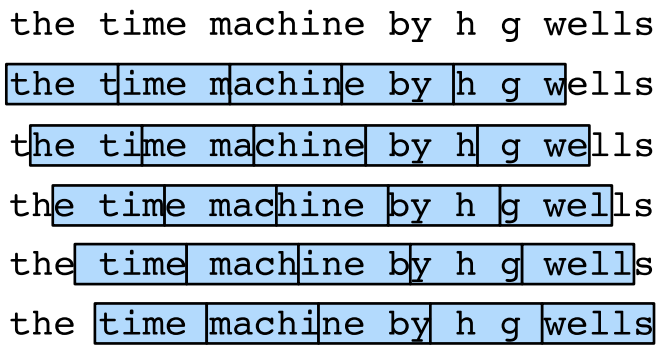


Fig. 9.3.1: Different offsets lead to different subsequences when splitting up text.

Do đó, chúng ta nên chọn cái nào từ Fig. 9.3.1? Trong thực tế, tất cả chúng đều tốt như nhau. Tuy nhiên, nếu chúng ta chỉ chọn một bù đắp, có phạm vi bảo hiểm hạn chế của tất cả các dãy con có thể để đào tạo mạng của chúng tôi. Do đó, chúng ta có thể bắt đầu với một offset ngẫu nhiên để phân vùng một chuỗi để có được cả *coverage* và *randomness*. Sau đây, chúng tôi mô tả cách thực hiện điều này cho cả hai *lấy mẫu ngẫu nhiên* và * phân vùng tuần tự* chiến lược.

Lấy mẫu ngẫu nhiên

Trong lấy mẫu ngẫu nhiên, mỗi ví dụ là một dãy con được chụp tùy tiện trên chuỗi dài ban đầu Các dãy con từ hai minibatches ngẫu nhiên liền kề trong quá trình lặp lại không nhất thiết liền kề trên chuỗi gốc. Đối với mô hình hóa ngôn ngữ, mục tiêu là dự đoán token tiếp theo dựa trên những mã thông báo mà chúng ta đã thấy cho đến nay, do đó các nhãn là chuỗi ban đầu, được dịch chuyển bởi một mã thông báo.

Mã sau ngẫu nhiên tạo ra một minibatch từ dữ liệu mỗi lần. Ở đây, đối số `batch_size` chỉ định số ví dụ dãy con trong mỗi minibatch và `num_steps` là số bước thời gian được xác định trước trong mỗi dãy con.

```
def seq_data_iter_random(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using random sampling."""
    # Start with a random offset (inclusive of `num_steps - 1`) to partition a
```

(continues on next page)

```
    # sequence
    corpus = corpus[random.randint(0, num_steps - 1):]
    # Subtract 1 since we need to account for labels
    num_subseqs = (len(corpus) - 1) // num_steps
    # The starting indices for subsequences of length `num_steps`
    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))
    # In random sampling, the subsequences from two adjacent random
    # minibatches during iteration are not necessarily adjacent on the
    # original sequence
    random.shuffle(initial_indices)

    def data(pos):
        # Return a sequence of length `num_steps` starting from `pos`
        return corpus[pos: pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Here, `initial_indices` contains randomized starting indices for
        # subsequences
        initial_indices_per_batch = initial_indices[i: i + batch_size]
        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]
        yield torch.tensor(X), torch.tensor(Y)
```

Hãy để chúng tôi tự tạo một chuỗi từ 0 đến 34. Chúng tôi giả định rằng kích thước lô và số bước thời gian là 2 và 5, tương ứng. Điều này có nghĩa là chúng ta có thể tạo ra $\lfloor (35 - 1)/5 \rfloor = 6$ các cặp con nhãn tính năng. Với kích thước minibatch là 2, chúng tôi chỉ nhận được 3 minibatches.

```
my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[10, 11, 12, 13, 14],
        [ 5,  6,  7,  8,  9]])
Y: tensor([[11, 12, 13, 14, 15],
        [ 6,  7,  8,  9, 10]])
X:  tensor([[20, 21, 22, 23, 24],
        [15, 16, 17, 18, 19]])
Y: tensor([[21, 22, 23, 24, 25],
        [16, 17, 18, 19, 20]])
X:  tensor([[25, 26, 27, 28, 29],
        [ 0,  1,  2,  3,  4]])
Y: tensor([[26, 27, 28, 29, 30],
        [ 1,  2,  3,  4,  5]])
```

Phân vùng tuần tự

Ngoài việc lấy mẫu ngẫu nhiên của chuỗi gốc, chúng tôi cũng có thể đảm bảo rằng các dãy con từ hai mini-batches liền kề trong quá trình lặp lại liền kề trên chuỗi gốc. Chiến lược này bảo tồn thứ tự phân chia dãy khi lặp qua minibatches, do đó được gọi là tuần tự phân vùng.

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):  #@save
    """Generate a minibatch of subsequences using sequential partitioning."""
    # Start with a random offset to partition a sequence
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = torch.tensor(corpus[offset: offset + num_tokens])
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num_tokens])
    Xs, Ys = Xs.reshape(batch_size, -1), Ys.reshape(batch_size, -1)
    num_batches = Xs.shape[1] // num_steps
    for i in range(0, num_steps * num_batches, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

Sử dụng cùng một cài đặt, chúng ta hãy tính năng in `X` và nhãn `Y` cho mỗi minibatch của dãy con được đọc bởi phân vùng tuần tự. Lưu ý rằng các dãy con từ hai minibatches liền kề trong khi lặp lại thực sự liền kề trên dãy ban đầu.

```
for X, Y in seq_data_iter_sequential(my_seq, batch_size=2, num_steps=5):
    print('X: ', X, '\nY:', Y)
```

```
X:  tensor([[ 2,  3,  4,  5,  6],
        [18, 19, 20, 21, 22]])
Y: tensor([[ 3,  4,  5,  6,  7],
        [19, 20, 21, 22, 23]])
X:  tensor([[ 7,  8,  9, 10, 11],
        [23, 24, 25, 26, 27]])
Y: tensor([[ 8,  9, 10, 11, 12],
        [24, 25, 26, 27, 28]])
X:  tensor([[12, 13, 14, 15, 16],
        [28, 29, 30, 31, 32]])
Y: tensor([[13, 14, 15, 16, 17],
        [29, 30, 31, 32, 33]])
```

Bây giờ chúng ta bọc hai hàm lấy mẫu ở trên vào một lớp để chúng ta có thể sử dụng nó như một bộ lặp dữ liệu sau này.

```
class SeqDataLoader:  #@save
    """An iterator to load sequence data."""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
        if use_random_iter:
            self.data_iter_fn = d2l.seq_data_iter_random
        else:
            self.data_iter_fn = d2l.seq_data_iter_sequential
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps
```

(continues on next page)

(continued from previous page)

```
    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Cuối cùng, chúng ta định nghĩa một hàm `load_data_time_machine` trả về cả bộ lặp dữ liệu và từ vựng , vì vậy chúng ta có thể sử dụng nó tương tự như các hàm khác với tiền tố `load_data`, chẳng hạn như `d2l.load_data_fashion_mnist` được định nghĩa trong Section 4.5.

```
def load_data_time_machine(batch_size, num_steps,  #@save
                           use_random_iter=False, max_tokens=10000):
    """Return the iterator and the vocabulary of the time machine dataset."""
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

9.3.5 Tóm tắt

- Mô hình ngôn ngữ là chìa khóa để xử lý ngôn ngữ tự nhiên.
- n-gram cung cấp một mô hình thuận tiện để xử lý các chuỗi dài bằng cách cắt ngắn sự phụ thuộc.
- Chuỗi dài bị vấn đề mà chúng xảy ra rất hiếm khi hoặc không bao giờ.
- Định luật Zipf chi phối từ phân phối cho không chỉ unigrams mà còn là n-gram khác.
- Có rất nhiều cấu trúc nhưng không đủ tần số để đối phó với các kết hợp từ không thường xuyên một cách hiệu quả thông qua Laplace làm mịn.
- Các lựa chọn chính để đọc các chuỗi dài là lấy mẫu ngẫu nhiên và phân vùng tuần tự. Cái sau có thể đảm bảo rằng các dãy con từ hai minibatches liền kề trong quá trình lặp lại liền kề trên chuỗi gốc.

9.3.6 Bài tập

1. Giả sử có $100,000$ từ trong tập dữ liệu đào tạo. Bốn gram cần lưu trữ bao nhiêu tần số từ và tần số liền kề nhiều từ?
2. Làm thế nào bạn sẽ mô hình một cuộc đối thoại?
3. Ước tính số mũ của định luật Zipf cho unigrams, bigram, và trigram.
4. Bạn có thể nghĩ đến những phương pháp nào khác để đọc dữ liệu trình tự dài?
5. Hãy xem xét bù ngẫu nhiên mà chúng ta sử dụng để đọc các chuỗi dài.
    1. Tại sao nó là một ý tưởng tốt để có một bù đắp ngẫu nhiên?
    2. Liệu nó có thực sự dẫn đến một phân phối hoàn toàn thống nhất trên các chuỗi trên tài liệu?
    3. Bạn sẽ phải làm gì để làm cho mọi thứ trở nên thống nhất hơn?
6. Nếu chúng ta muốn một ví dụ trình tự là một câu hoàn chỉnh, điều này giới thiệu loại vấn đề nào trong việc lấy mẫu minibatch? Làm thế nào chúng ta có thể khắc phục sự cố?

Discussions[102]

[102] https://discuss.d2l.ai/t/118

9.4 Mạng nơ-ron tái phát

Trong Section 9.3 chúng tôi đã giới thiệu các mô hình n-gram, trong đó xác suất có điều kiện của từ x_t tại bước thời gian t chỉ phụ thuộc vào $n-1$ từ trước. Nếu chúng ta muốn kết hợp hiệu ứng có thể có của các từ sớm hơn bước thời gian $t-(n-1)$ trên x_t, chúng ta cần tăng n. Tuy nhiên, số lượng tham số mô hình cũng sẽ tăng theo cấp số nhân với nó, vì chúng ta cần lưu trữ $|\mathcal{V}|^n$ số cho một bộ từ vựng $\mathcal{V}$. Do đó, thay vì mô hình hóa $P(x_t \mid x_{t-1}, \ldots, x_{t-n+1})$, tốt hơn là sử dụng một mô hình biến tiềm ẩn:

$$P(x_t \mid x_{t-1}, \ldots, x_1) \approx P(x_t \mid h_{t-1}), \tag{9.4.1}$$

trong đó h_{t-1} là trạng thái * ẩn* (còn được gọi là biến ẩn) lưu trữ thông tin trình tự lên đến bước thời gian $t-1$. Nói chung, trạng thái ẩn bất cứ lúc nào bước t có thể được tính toán dựa trên cả đầu vào hiện tại x_t và trạng thái ẩn trước đó h_{t-1}:

$$h_t = f(x_t, h_{t-1}). \tag{9.4.2}$$

Đối với một chức năng đủ mạnh f trong (9.4.2), mô hình biến tiềm ẩn không phải là một xấp xỉ. Rốt cuộc, h_t có thể chỉ đơn giản là lưu trữ tất cả dữ liệu mà nó đã quan sát cho đến nay. Tuy nhiên, nó có khả năng có thể làm cho cả tính toán và lưu trữ đắt tiền.

Nhớ lại rằng chúng ta đã thảo luận về các lớp ẩn với các đơn vị ẩn trong Chapter 5. Đáng chú ý là các lớp ẩn và trạng thái ẩn đề cập đến hai khái niệm rất khác nhau. Các lớp ẩn, như giải thích, các lớp được ẩn khỏi chế độ xem trên đường dẫn từ đầu vào đến đầu ra. Các trạng thái ẩn đang nói về mặt kỹ thuật * đầu vào* cho bất cứ điều gì chúng ta làm ở một bước nhất định và chúng chỉ có thể được tính toán bằng cách xem dữ liệu ở các bước thời gian trước.

Mạng thần kinh định kỳ (RNN) là các mạng thần kinh với các trạng thái ẩn. Trước khi giới thiệu mô hình RNN, lần đầu tiên chúng tôi xem lại mô hình MLP được giới thiệu trong Section 5.1.

9.4.1 Mạng thần kinh không có trạng thái ẩn

Chúng ta hãy nhìn vào một MLP với một lớp ẩn duy nhất. Hãy để chức năng kích hoạt của lớp ẩn là ϕ. Với một minibatch các ví dụ $\mathbf{X} \in \mathbb{R}^{n \times d}$ với kích thước lô n và d đầu vào, đầu ra $\mathbf{H} \in \mathbb{R}^{n \times h}$ của lớp ẩn được tính như

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \tag{9.4.3}$$

Trong (9.4.3), chúng ta có tham số trọng lượng $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, tham số thiên vị $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ và số lượng đơn vị ẩn h, cho lớp ẩn. Như vậy, phát sóng (xem Section 3.1.3) được áp dụng trong quá trình tổng kết. Tiếp theo, biến ẩn $\mathbf{H}$ được sử dụng làm đầu vào của lớp đầu ra. Lớp đầu ra được đưa ra bởi

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{9.4.4}$$

trong đó $\mathbf{O} \in \mathbb{R}^{n \times q}$ là biến đầu ra, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ là tham số trọng lượng và $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ là tham số thiên vị của lớp đầu ra. Nếu đó là một bài toán phân loại, chúng ta có thể sử dụng softmax($\mathbf{O}$) để tính toán phân phối xác suất của các loại đầu ra.

Điều này hoàn toàn tương tự như bài toán hồi quy mà chúng tôi đã giải quyết trước đây trong Section 9.1, do đó chúng tôi bỏ qua các chi tiết. Đủ để nói rằng chúng ta có thể chọn các cặp nhãn tính năng một cách ngẫu nhiên và tìm hiểu các thông số của mạng của chúng tôi thông qua sự phân biệt tự động và gốc gradient ngẫu nhiên.

9.4.2 Mạng thần kinh định kỳ với các trạng thái ẩn

Các vấn đề hoàn toàn khác nhau khi chúng ta có các trạng thái ẩn. Chúng ta hãy nhìn vào cấu trúc một số chi tiết hơn.

Giả sử rằng chúng ta có một minibatch của đầu vào $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ tại bước thời gian t. Nói cách khác, đối với một minibatch gồm n ví dụ chuỗi, mỗi hàng $\mathbf{X}_t$ tương ứng với một ví dụ tại bước thời điểm t từ chuỗi. Tiếp theo, biểu thị bằng $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ biến ẩn của bước thời gian t. Không giống như MLP, ở đây chúng ta lưu biến ẩn $\mathbf{H}_{t-1}$ từ bước thời gian trước và giới thiệu một tham số trọng lượng mới $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ để mô tả cách sử dụng biến ẩn của bước thời gian trước đó trong bước thời gian hiện tại. Cụ thể, việc tính toán biến ẩn của bước thời gian hiện tại được xác định bởi đầu vào của bước thời gian hiện tại cùng với biến ẩn của bước thời gian trước đó:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \tag{9.4.5}$$

So với (9.4.3), (9.4.5) bổ sung thêm một thuật ngữ $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ và do đó khởi tạo (9.4.2). Từ mối quan hệ giữa các biến ẩn $\mathbf{H}_t$ và $\mathbf{H}_{t-1}$ của các bước thời gian liền kề, chúng ta biết rằng các biến này đã thu thập và giữ lại thông tin lịch sử của trình tự lên đến bước thời gian hiện tại của chúng, giống như trạng thái hoặc bộ nhớ của bước thời gian hiện tại của mạng thần kinh. Do đó, một biến ẩn như vậy được gọi là trạng thái * hidden. *Vì trạng thái ẩn sử dụng cùng một định nghĩa của bước thời gian trước đó trong bước thời gian hiện tại, việc tính toán :eq:`rnn_h_with_state` là* đệ quy*. Do đó, các mạng thần kinh với các trạng thái ẩn dựa trên tính toán tái phát được đặt tên *mạng thần kinh định kỳ*. Các lớp thực hiện tính toán (9.4.5) trong RNN s được gọi là * lớp* lặp đi lặp lại*.

Có nhiều cách khác nhau để xây dựng RNNs. RNNs với trạng thái ẩn được xác định bởi (9.4.5) là rất phổ biến. Đối với bước thời gian t, đầu ra của lớp đầu ra tương tự như tính toán trong MLP:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{9.4.6}$$

Các thông số của RNN bao gồm các trọng lượng $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, và thiên vị $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ của lớp ẩn, cùng với trọng lượng $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ và thiên vị $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ của lớp đầu ra. Điều đáng nói là ngay cả ở các bước thời gian khác nhau, RNN s luôn sử dụng các tham số mô hình này. Do đó, chi phí tham số hóa của một RNN không tăng lên khi số bước thời gian tăng lên.

Fig. 9.4.1 minh họa logic tính toán của một RNN tại ba bước thời gian liền kề. Bất cứ lúc nào bước t, tính toán trạng thái ẩn có thể được coi là: (i) nối đầu vào $\mathbf{X}_t$ ở bước thời gian hiện tại t và trạng thái ẩn $\mathbf{H}_{t-1}$ ở bước thời gian trước $t-1$; (ii) cho kết quả nối vào một lớp kết nối đầy đủ với sự kích hoạt chức năng ϕ. Đầu ra của một lớp được kết nối hoàn toàn như vậy là trạng thái ẩn $\mathbf{H}_t$ của bước thời gian hiện tại t. Trong trường hợp này, các thông số mô hình là nối $\mathbf{W}_{xh}$ và $\mathbf{W}_{hh}$, và một sự thiên vị của $\mathbf{b}_h$, tất cả từ (9.4.5). Trạng thái ẩn của bước thời gian hiện tại t, $\mathbf{H}_t$, sẽ tham gia vào việc tính toán trạng thái ẩn $\mathbf{H}_{t+1}$ của bước thời gian tiếp theo $t+1$. Hơn nữa, $\mathbf{H}_t$ cũng sẽ được đưa vào lớp đầu ra được kết nối hoàn toàn để tính toán đầu ra $\mathbf{O}_t$ của bước thời gian hiện tại t.

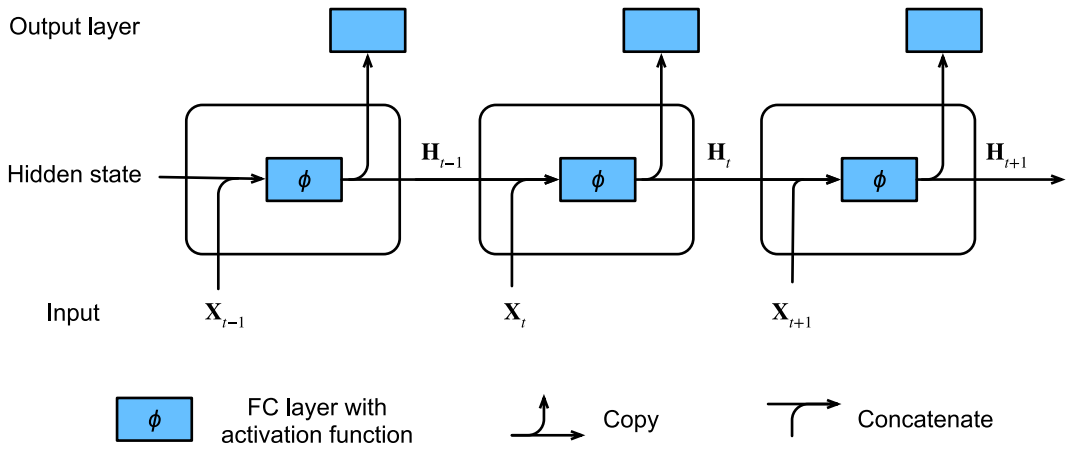


Fig. 9.4.1: An RNN with a hidden state.

Chúng tôi vừa đề cập rằng việc tính $\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh}$ cho trạng thái ẩn tương đương với phép nhân ma trận của nối $\mathbf{X}_t$ và $\mathbf{H}_{t-1}$ và nối $\mathbf{H}_{t-1}$ và nối $\mathbf{W}_{xh}$ và $\mathbf{W}_{hh}$. Mặc dù điều này có thể được chứng minh trong toán học, sau đây chúng ta chỉ sử dụng một đoạn mã đơn giản để hiển thị điều này. Để bắt đầu, chúng tôi xác định ma trận `X`, `W_xh`, `H` và `W_hh`, có hình dạng tương ứng (3, 1), (1, 4), (3, 4) và (4, 4). Nhân `X` với `W_xh`, và `H` `W_hh`, tương ứng, và sau đó thêm hai nhân này, chúng tôi có được một ma trận hình dạng (3, 4).

```
import torch
from d2l import torch as d2l
```

```
X, W_xh = torch.normal(0, 1, (3, 1)), torch.normal(0, 1, (1, 4))
H, W_hh = torch.normal(0, 1, (3, 4)), torch.normal(0, 1, (4, 4))
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

```
tensor([[ 0.0795,  4.0775,  0.3635, -3.0606],
        [-1.0515, -3.1537,  1.4391,  4.5820],
        [-2.4584, -0.9088,  3.2624,  2.7681]])
```

Bây giờ chúng ta nối các ma trận `X` và `H` dọc theo các cột (trục 1), và ma trận `W_xh` và `W_hh` dọc theo hàng (trục 0). Hai kết nối này dẫn đến ma trận của hình dạng (3, 5) và hình dạng (5, 4), tương ứng. Nhân hai ma trận nối này, chúng ta có được ma trận đầu ra giống nhau của hình dạng (3, 4) như trên.

```
torch.matmul(torch.cat((X, H), 1), torch.cat((W_xh, W_hh), 0))
```

```
tensor([[ 0.0795,  4.0775,  0.3635, -3.0606],
        [-1.0515, -3.1537,  1.4391,  4.5820],
        [-2.4584, -0.9088,  3.2624,  2.7681]])
```

9.4.3 Mô hình ngôn ngữ cấp ký tự dựa trên RNN

Nhớ lại rằng đối với mô hình hóa ngôn ngữ trong Section 9.3, chúng tôi đặt mục tiêu dự đoán token tiếp theo dựa trên các mã thông báo hiện tại và quá khứ, do đó chúng tôi chuyển chuỗi gốc bằng một mã thông báo làm nhãn. Bengio et al. lần đầu tiên đề xuất sử dụng mạng thần kinh để mô hình hóa ngôn ngữ [Bengio.Ducharme.Vincent.ea.2003]. Sau đây, chúng tôi minh họa cách RNN có thể được sử dụng để xây dựng một mô hình ngôn ngữ. Hãy để kích thước minibatch là một, và trình tự của văn bản là "máy". Để đơn giản hóa việc đào tạo trong các phần tiếp theo, chúng tôi mã hóa văn bản thành các ký tự thay vì từ và xem xét mô hình ngôn ngữ cấp ký tự*. Fig. 9.4.2 thể hiện cách dự đoán ký tự tiếp theo dựa trên các ký tự hiện tại và trước đó thông qua RNN cho mô hình ngôn ngữ cấp ký tự.

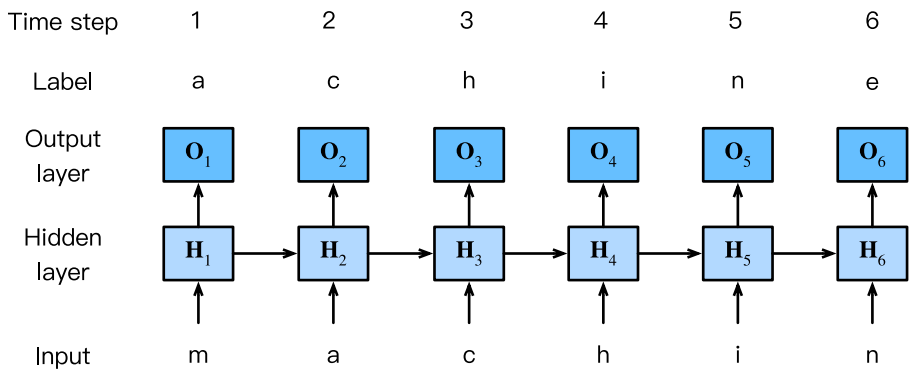


Fig. 9.4.2: A character-level language model based on the RNN. The input and label sequences are "machin" and "achine", respectively.

Trong quá trình đào tạo, chúng tôi chạy một thao tác softmax trên đầu ra từ lớp đầu ra cho mỗi bước thời gian, sau đó sử dụng tổn thất chéo entropy để tính toán lỗi giữa đầu ra mô hình và nhãn. Do tính toán tái phát của trạng thái ẩn trong lớp ẩn, đầu ra của bước thời gian 3 trong Fig. 9.4.2, $\mathbf{O}_3$, được xác định bởi chuỗi văn bản "m", "a", và "c". Vì ký tự tiếp theo của dãy trong dữ liệu đào tạo là "h", việc mất thời gian bước 3 sẽ phụ thuộc vào phân bố xác suất của ký tự tiếp theo được tạo ra dựa trên dãy tính năng "m", "a", "c" và nhãn "h" của bước thời gian này.

Trong thực tế, mỗi mã thông báo được đại diện bởi một vector d chiều, và chúng tôi sử dụng kích thước lô $n > 1$. Do đó, đầu vào $\mathbf{X}_t$ tại bước thời gian t sẽ là ma trận $n \times d$, giống hệt với những gì chúng ta đã thảo luận trong Section 9.4.2.

9.4.4 Bối rối

Cuối cùng, chúng ta hãy thảo luận về cách đo lường chất lượng mô hình ngôn ngữ, sẽ được sử dụng để đánh giá các mô hình dựa trên RN của chúng tôi trong các phần tiếp theo. Một cách là kiểm tra văn bản đáng ngạc nhiên như thế nào. Một mô hình ngôn ngữ tốt có thể dự đoán với các mã thông báo có độ chính xác cao mà những gì chúng ta sẽ thấy tiếp theo. Hãy xem xét các liên tục sau của cụm từ "Trời mưa", như được đề xuất bởi các mô hình ngôn ngữ khác nhau:

1. "Trời đang mưa bên ngoài"
2. "Trời đang mưa cây chuối"
3. "It is raining mưa; kcj pwepoiut"

Về chất lượng, ví dụ 1 rõ ràng là tốt nhất. Các từ là hợp lý và hợp lý mạch lạc. Mặc dù nó có thể không hoàn toàn phản ánh chính xác từ nào theo ngữ nghĩa ("ở San Francisco" và "vào mùa đông" sẽ là phần mở rộng hoàn toàn hợp lý), mô hình có thể nắm bắt loại từ nào theo sau. Ví dụ 2 tồi tệ hơn đáng kể bằng cách tạo ra một phần mở rộng vô nghĩa. Tuy nhiên, ít nhất mô hình đã học cách đánh vần các từ và một mức độ tương quan giữa các từ. Cuối cùng, ví dụ 3 chỉ ra một mô hình được đào tạo kém không phù hợp với dữ liệu đúng cách.

Chúng ta có thể đo lường chất lượng của mô hình bằng cách tính toán khả năng của trình tự. Thật không may đây là một con số khó hiểu và khó so sánh. Rốt cuộc, các trình tự ngắn hơn có nhiều khả năng xảy ra hơn nhiều so với các trình tự dài hơn, do đó đánh giá mô hình trên magnum opus của Tolstoy *Chiến tranh và Hòa bình* chắc chắn sẽ tạo ra một khả năng nhỏ hơn nhiều so với, nói, trên tiểu thuyết của Saint-Exupery* The Little Prince*. Những gì còn thiếu là tương đương với mức trung bình.

Lý thuyết thông tin có ích ở đây. Chúng tôi đã xác định entropy, ngạc nhiên và cross-entropy khi chúng tôi giới thiệu hồi quy softmax (Section 4.4.7) và nhiều lý thuyết thông tin được thảo luận trong online appendix on information theory[103]. Nếu chúng ta muốn nén văn bản, chúng ta có thể hỏi về việc dự đoán token tiếp theo cho bộ mã thông báo hiện tại. Một mô hình ngôn ngữ tốt hơn sẽ cho phép chúng ta dự đoán token tiếp theo chính xác hơn. Do đó, nó sẽ cho phép chúng ta chi tiêu ít bit hơn trong việc nén chuỗi. Vì vậy, chúng ta có thể đo lường nó bằng cách mất chéo entropy trung bình trên tất cả các mã thông báo n của một chuỗi:

$$\frac{1}{n}\sum_{t=1}^{n} -\log P(x_t \mid x_{t-1}, \ldots, x_1), \tag{9.4.7}$$

trong đó P được đưa ra bởi một mô hình ngôn ngữ và x_t là mã thông báo thực tế quan sát tại bước thời gian t từ chuỗi. Điều này làm cho hiệu suất trên các tài liệu có độ dài khác nhau tương đương. Vì lý do lịch sử, các nhà khoa học trong xử lý ngôn ngữ tự nhiên thích sử dụng một số lượng gọi là *perplexity*. Tóm lại, nó là hàm mũ của (9.4.7):

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{t-1}, \ldots, x_1)\right). \tag{9.4.8}$$

Sự bối rối có thể được hiểu rõ nhất là trung bình hài hòa của số lượng lựa chọn thực sự mà chúng ta có khi quyết định token nào sẽ chọn tiếp theo. Chúng ta hãy xem xét một số trường hợp:

- Trong trường hợp tốt nhất, mô hình luôn ước tính hoàn hảo xác suất của mã thông báo nhãn là 1. Trong trường hợp này, sự bối rối của mô hình là 1.
- Trong trường hợp xấu nhất, mô hình luôn dự đoán xác suất của token nhãn là 0. Trong tình huống này, sự bối rối là vô cùng tích cực.
- Tại đường cơ sở, mô hình dự đoán phân phối thống nhất trên tất cả các mã thông báo có sẵn của từ vựng. Trong trường hợp này, sự bối rối bằng số lượng mã thông báo duy nhất của từ vựng. Trên thực tế, nếu chúng ta lưu trữ trình tự mà không cần bất kỳ nén nào, đây sẽ là điều tốt nhất chúng ta có thể làm để mã hóa nó. Do đó, điều này cung cấp một ràng buộc trên không tầm thường mà bất kỳ mô hình hữu ích nào cũng phải đánh bại.

Trong các phần sau, chúng tôi sẽ triển khai RNNcho các mô hình ngôn ngữ cấp ký tự và sử dụng sự bối rối để đánh giá các mô hình như vậy.

[103] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/information-theory.html

9.4.5 Tóm tắt

- Một mạng thần kinh sử dụng tính toán định kỳ cho các trạng thái ẩn được gọi là mạng thần kinh tái phát (RNN).
- Trạng thái ẩn của một RNN có thể nắm bắt thông tin lịch sử của chuỗi lên đến bước thời gian hiện tại.
- Số lượng tham số mô hình RNN không tăng lên khi số bước thời gian tăng lên.
- Chúng ta có thể tạo mô hình ngôn ngữ cấp ký tự bằng cách sử dụng RNN.
- Chúng ta có thể sử dụng sự bối rối để đánh giá chất lượng của các mô hình ngôn ngữ.

9.4.6 Bài tập

1. Nếu chúng ta sử dụng một RNN để dự đoán ký tự tiếp theo trong một chuỗi văn bản, kích thước cần thiết cho bất kỳ đầu ra nào là gì?
2. Tại sao RNN s có thể thể hiện xác suất có điều kiện của một mã thông báo tại một bước thời gian dựa trên tất cả các token trước đó trong chuỗi văn bản?
3. Điều gì xảy ra với gradient nếu bạn backpropagate thông qua một chuỗi dài?
4. Một số vấn đề liên quan đến mô hình ngôn ngữ được mô tả trong phần này là gì?

Discussions[104]

9.5 Thực hiện các mạng nơ-ron tái phát từ đầu

Trong phần này, chúng tôi sẽ triển khai RNN từ đầu cho mô hình ngôn ngữ cấp ký tự, theo mô tả của chúng tôi trong Section 9.4. Một mô hình như vậy sẽ được đào tạo trên H Gwells' * The Time Machine*. Như trước đây, chúng ta bắt đầu bằng cách đọc tập dữ liệu đầu tiên, được giới thiệu trong Section 9.3.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

[104] https://discuss.d2l.ai/t/1050

9.5.1 Mã hóa Một-Hot

Nhớ lại rằng mỗi mã thông báo được biểu diễn dưới dạng chỉ số số trong `train_iter`. Việc cung cấp các chỉ số này trực tiếp vào mạng thần kinh có thể khiến bạn khó học. Chúng ta thường đại diện cho mỗi token như một vector tính năng biểu cảm hơn. Đại diện dễ nhất được gọi là * mã hóa một nóng*, được giới thiệu trong Section 4.4.1.

Tóm lại, chúng ta ánh xạ mỗi chỉ mục đến một vector đơn vị khác nhau: giả sử rằng số lượng token khác nhau trong từ vựng là N (`len(vocab)`) và các chỉ số token dao động từ 0 đến $N-1$. Nếu chỉ số của một mã thông báo là số nguyên i, thì ta tạo một vectơ của tất cả 0s với độ dài N và đặt phần tử ở vị trí i thành 1. Vector này là vectơ một nóng của mã thông báo gốc. Các vectơ một nóng với chỉ số 0 và 2 được hiển thị bên dưới.

```
F.one_hot(torch.tensor([0, 2]), len(vocab))
```

```
tensor([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0,
         0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0,
         0, 0, 0, 0]])
```

Hình dạng của minibatch mà chúng tôi lấy mẫu mỗi lần là (kích thước lô, số bước thời gian). Hàm `one_hot` biến một minibatch như vậy thành một tensor ba chiều với chiều cuối cùng bằng với kích thước từ vựng (`len(vocab)`). Chúng ta thường chuyển đổi đầu vào để chúng ta sẽ có được một đầu ra của hình dạng (số bước thời gian, kích thước lô, kích thước từ vựng). Điều này sẽ cho phép chúng tôi vòng lặp thuận tiện hơn thông qua kích thước ngoài cùng để cập nhật các trạng thái ẩn của một minibatch, từng bước từng bước.

```
X = torch.arange(10).reshape((2, 5))
F.one_hot(X.T, 28).shape
```

```
torch.Size([5, 2, 28])
```

9.5.2 Khởi tạo các tham số mô hình

Tiếp theo, chúng ta khởi tạo các tham số mô hình cho mô hình RNN. Số lượng các đơn vị ẩn `num_hiddens` là một siêu tham số có thể điều chỉnh. Khi đào tạo mô hình ngôn ngữ, các đầu vào và đầu ra là từ cùng một từ vựng. Do đó, chúng có cùng chiều, tương đương với kích thước từ vựng.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
```

(continues on next page)

```
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

9.5.3 Mô hình RNN

Để định nghĩa một mô hình RNN, trước tiên chúng ta cần một hàm `init_rnn_state` để trả về trạng thái ẩn lúc khởi hóa. Nó trả về một tensor chứa đầy 0 và với một hình dạng của (batch size, số đơn vị ẩn). Sử dụng các tuples làm cho nó dễ dàng hơn để xử lý các tình huống mà trạng thái ẩn chứa nhiều biến, mà chúng ta sẽ gặp phải trong các phần sau.

```
def init_rnn_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

Hàm `rnn` sau định nghĩa cách tính trạng thái ẩn và đầu ra tại một bước thời gian Lưu ý rằng mô hình RNN vòng qua chiều ngoài cùng của `inputs` để nó cập nhật các trạng thái ẩn `H` của một minibatch, từng bước từng bước thời gian. Bên cạnh đó, chức năng kích hoạt ở đây sử dụng chức năng tanh. Như được mô tả trong Section 5.1, giá trị trung bình của hàm tanh là 0, khi các phần tử được phân bố đồng đều trên các số thực.

```
def rnn(inputs, state, params):
    # Here `inputs` shape: (`num_steps`, `batch_size`, `vocab_size`)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    # Shape of `X`: (`batch_size`, `vocab_size`)
    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Với tất cả các hàm cần thiết được định nghĩa, tiếp theo chúng ta create a class để bọc các hàm này và lưu trữ các tham số cho một mô hình RNN được thực hiện từ đầu.

```
class RNNModelScratch: #@save
    """A RNN Model implemented from scratch."""
    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state, self.forward_fn = init_state, forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return self.init_state(batch_size, self.num_hiddens, device)
```

Hãy để chúng tôi kiểm tra xem các đầu ra có hình dạng chính xác, ví dụ, để đảm bảo rằng chiều của trạng thái ẩn vẫn không thay đổi.

```
num_hiddens = 512
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
state = net.begin_state(X.shape[0], d2l.try_gpu())
Y, new_state = net(X.to(d2l.try_gpu()), state)
Y.shape, len(new_state), new_state[0].shape
```

```
(torch.Size([10, 28]), 1, torch.Size([2, 512]))
```

Chúng ta có thể thấy rằng hình dạng đầu ra là (số bước thời gian $\times$ kích thước lô, kích thước từ vựng), trong khi hình dạng trạng thái ẩn vẫn giữ nguyên, tức là, (kích thước lô, số đơn vị ẩn).

9.5.4 Prediction

Hãy để chúng tôi đầu tiên xác định hàm dự đoán để tạo ra các ký tự mới sau `prefix` người dùng cung cấp, đó là một chuỗi chứa một số ký tự. Khi lặp qua các ký tự bắt đầu này trong `prefix`, chúng tôi tiếp tục chuyển trạng thái ẩn sang bước thời gian tiếp theo mà không tạo ra bất kỳ đầu ra nào. Đây được gọi là khoảng thời gian *warm-up*, trong đó mô hình tự cập nhật (ví dụ: cập nhật trạng thái ẩn) nhưng không đưa ra dự đoán. Sau thời gian khởi động, trạng thái ẩn thường tốt hơn giá trị khởi tạo của nó ở đầu. Vì vậy, chúng tôi tạo ra các nhân vật dự đoán và phát ra chúng.

```
def predict_ch8(prefix, num_preds, net, vocab, device):  #@save
    """Generate new characters following the `prefix`."""
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]
    get_input = lambda: torch.tensor([outputs[-1]], device=device).reshape((1,
↪ 1))
    for y in prefix[1:]:  # Warm-up period
        _, state = net(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_preds):  # Predict `num_preds` steps
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Bây giờ chúng ta có thể kiểm tra hàm `predict_ch8`. Chúng tôi chỉ định tiền tố là `time traveller` và có nó tạo ra 10 ký tự bổ sung. Cho rằng chúng tôi chưa đào tạo mạng, nó sẽ tạo ra những dự đoán vô nghĩa.

```
predict_ch8('time traveller ', 10, net, vocab, d2l.try_gpu())
```

```
'time traveller mkpbqhnnnn'
```

9.5.5 Clipping Gradient

Đối với một chuỗi chiều dài T, chúng tôi tính toán độ dốc trên các bước thời gian T này trong một lần lặp lại, dẫn đến một chuỗi các sản phẩm ma trận có chiều dài $\mathcal{O}(T)$ trong quá trình truyền ngược. Như đã đề cập trong Section 5.8, nó có thể dẫn đến sự bất ổn số, ví dụ, các gradient có thể phát nổ hoặc biến mất, khi T lớn. Do đó, các mô hình RNN thường cần trợ giúp thêm để ổn định việc đào tạo.

Nói chung, khi giải quyết vấn đề tối ưu hóa, chúng tôi thực hiện các bước cập nhật cho tham số mô hình, nói ở dạng vector $\mathbf{x}$, theo hướng gradient âm $\mathbf{g}$ trên một minibatch. Ví dụ: với $\eta > 0$ là tốc độ học tập, trong một lần lặp lại, chúng tôi cập nhật $\mathbf{x}$ là $\mathbf{x} - \eta\mathbf{g}$. Chúng ta hãy giả định thêm rằng chức năng khách quan f được cư xử tốt, giả sử, * Lipschitz liên tục* với hằng số L. Điều đó có nghĩa là, đối với bất kỳ $\mathbf{x}$ và $\mathbf{y}$ nào chúng tôi có

$$|f(\mathbf{x}) - f(\mathbf{y})| \leq L\|\mathbf{x} - \mathbf{y}\|. \tag{9.5.1}$$

Trong trường hợp này, chúng ta có thể giả định một cách an toàn rằng nếu chúng ta cập nhật vector tham số bởi $\eta\mathbf{g}$, thì

$$|f(\mathbf{x}) - f(\mathbf{x} - \eta\mathbf{g})| \leq L\eta\|\mathbf{g}\|, \tag{9.5.2}$$

điều đó có nghĩa là chúng ta sẽ không quan sát một sự thay đổi hơn $L\eta\|\mathbf{g}\|$. Đây vừa là một lời nguyền vừa là một phước lành. Về phía lời nguyền, nó giới hạn tốc độ tiến bộ; trong khi về phía phước lành, nó giới hạn mức độ mà mọi thứ có thể đi sai nếu chúng ta di chuyển sai hướng.

Đôi khi các gradient có thể khá lớn và thuật toán tối ưu hóa có thể không hội tụ. Chúng tôi có thể giải quyết điều này bằng cách giảm tỷ lệ học tập η. Nhưng nếu chúng ta chỉ * hiếm có * có được gradient lớn thì sao? Trong trường hợp này một cách tiếp cận như vậy có thể xuất hiện hoàn toàn không chính đáng. Một lựa chọn phổ biến là cắt gradient $\mathbf{g}$ bằng cách chiếu chúng trở lại một quả bóng có bán kính nhất định, giả sử θ qua

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{9.5.3}$$

Bằng cách đó, chúng tôi biết rằng định mức gradient không bao giờ vượt quá θ và gradient được cập nhật hoàn toàn phù hợp với hướng ban đầu là $\mathbf{g}$. Nó cũng có tác dụng phụ mong muốn của việc hạn chế ảnh hưởng bất kỳ minibatch nhất định nào (và trong đó bất kỳ mẫu nhất định nào) có thể tác dụng lên vectơ tham số. Điều này ban cho một mức độ mạnh mẽ nhất định cho mô hình. Gradient clipping cung cấp một sửa chữa nhanh chóng cho sự bùng nổ gradient. Mặc dù nó không hoàn toàn giải quyết vấn đề, nhưng nó là một trong nhiều kỹ thuật để giảm bớt nó.

Dưới đây chúng ta định nghĩa một hàm để cắt các gradient của một mô hình được triển khai từ đầu hoặc một mô hình được xây dựng bởi các API cấp cao. Cũng lưu ý rằng chúng tôi tính toán định mức gradient trên tất cả các tham số mô hình.

```
def grad_clipping(net, theta):  #@save
    """Clip the gradient."""
    if isinstance(net, nn.Module):
        params = [p for p in net.parameters() if p.requires_grad]
    else:
        params = net.params
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

9.5.6 Đào tạo

Trước khi đào tạo mô hình, chúng ta hãy xác định một hàm để đào tạo mô hình trong một kỷ chức. Nó khác với cách chúng tôi đào tạo mô hình Section 4.6 ở ba nơi:

1. Các phương pháp lấy mẫu khác nhau cho dữ liệu tuần tự (lấy mẫu ngẫu nhiên và phân vùng tuần tự) sẽ dẫn đến sự khác biệt trong việc khởi tạo các trạng thái ẩn.
2. Chúng tôi kẹp các gradient trước khi cập nhật các tham số mô hình. Điều này đảm bảo rằng mô hình không phân kỳ ngay cả khi gradient thổi lên tại một số điểm trong quá trình đào tạo.
3. Chúng tôi sử dụng sự bối rối để đánh giá mô hình. Như đã thảo luận trong Section 9.4.4, điều này đảm bảo rằng các chuỗi có độ dài khác nhau có thể so sánh được.

Cụ thể, khi phân vùng tuần tự được sử dụng, chúng ta chỉ khởi tạo trạng thái ẩn ở đầu mỗi kỷ nguyên. Vì ví dụ dãy con i^{th} trong minibatch tiếp theo liền kề với ví dụ dãy thứ tự i^{th} hiện tại, trạng thái ẩn ở cuối minibatch hiện tại sẽ được sử dụng để khởi tạo trạng thái ẩn ở đầu minibatch tiếp theo. Bằng cách này, thông tin lịch sử của dãy được lưu trữ trong trạng thái ẩn có thể chảy qua các dãy tiếp giáp bên trong một kỷ nguyên. Tuy nhiên, việc tính toán trạng thái ẩn tại bất kỳ điểm nào phụ thuộc vào tất cả các minibatches trước đó trong cùng một kỷ nguyên, làm phức tạp tính toán gradient. Để giảm chi phí tính toán, chúng tôi tách gradient trước khi xử lý bất kỳ minibatch nào để tính toán gradient của trạng thái ẩn luôn bị giới hạn ở các bước thời gian trong một minibatch.

Khi sử dụng lấy mẫu ngẫu nhiên, chúng ta cần khởi tạo lại trạng thái ẩn cho mỗi lần lặp lại vì mỗi ví dụ được lấy mẫu với một vị trí ngẫu nhiên. Tương tự như hàm `train_epoch_ch3` trong Section 4.6, `updater` là một hàm chung để cập nhật các tham số mô hình. Nó có thể là chức năng `d2l.sgd` được triển khai từ đầu hoặc chức năng tối ưu hóa tích hợp trong một khuôn khổ học sâu.

```
#@save
def train_epoch_ch8(net, train_iter, loss, updater, device, use_random_iter):
    """Train a net within one epoch (defined in Chapter 8)."""
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
    for X, Y in train_iter:
        if state is None or use_random_iter:
            # Initialize `state` when either it is the first iteration or
            # using random sampling
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                # `state` is a tensor for `nn.GRU`
                state.detach_()
            else:
                # `state` is a tuple of tensors for `nn.LSTM` and
                # for our custom scratch implementation
                for s in state:
                    s.detach_()
        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)
        l = loss(y_hat, y.long()).mean()
        if isinstance(updater, torch.optim.Optimizer):
            updater.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            updater.step()
```

(continues on next page)

```
        else:
            l.backward()
            grad_clipping(net, 1)
            # Since the `mean` function has been invoked
            updater(batch_size=1)
        metric.add(l * y.numel(), y.numel())
    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

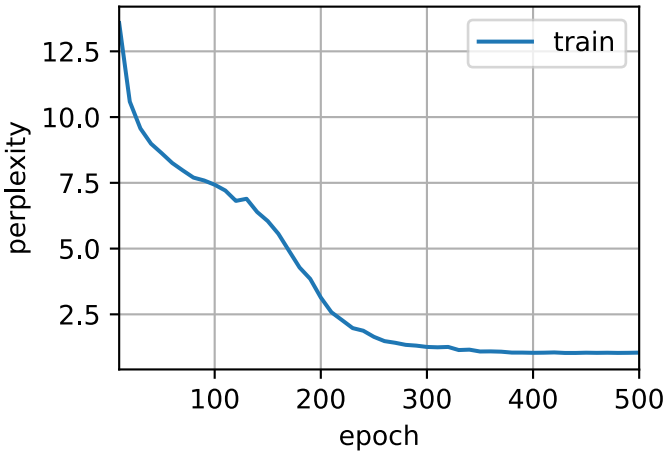
Chức năng đào tạo hỗ trợ một mô hình RNN được thực hiện từ đầu hoặc sử dụng APIs cấp cao.

```
#@save
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """Train a model (defined in Chapter 8)."""
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[10, num_epochs])
    # Initialize
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: d2l.sgd(net.params, lr, batch_size)
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)
    # Train and predict
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(
            net, train_iter, loss, updater, device, use_random_iter)
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))
            animator.add(epoch + 1, [ppl])
    print(f'perplexity {ppl:.1f}, {speed:.1f} tokens/sec on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

Bây giờ chúng ta có thể đào tạo mô hình RNN Vì chúng ta chỉ sử dụng 10000 mã thông báo trong tập dữ liệu, mô hình cần nhiều kỷ nguyên hơn để hội tụ tốt hơn.

```
num_epochs, lr = 500, 1
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu())
```

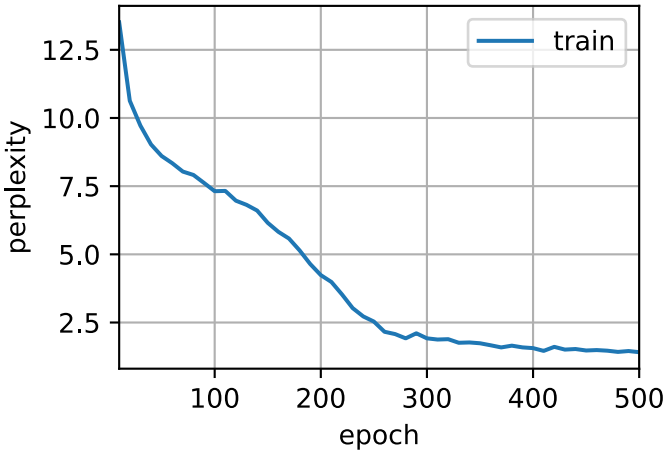
```
perplexity 1.0, 75622.3 tokens/sec on cuda:0
time traveller with a slight accession ofcheerfulness really thi
traveller with a slight accession ofcheerfulness really thi
```



Cuối cùng, chúng ta hãy kiểm tra kết quả của việc sử dụng phương pháp lấy mẫu ngẫu nhiên.

```
net = RNNModelScratch(len(vocab), num_hiddens, d2l.try_gpu(), get_params,
                      init_rnn_state, rnn)
train_ch8(net, train_iter, vocab, lr, num_epochs, d2l.try_gpu(),
          use_random_iter=True)
```

```
perplexity 1.4, 78034.7 tokens/sec on cuda:0
time traveller smiledcrination is to be accepted is an absolutel
traveller after the pauserequired for the proper assimilati
```



Trong khi thực hiện mô hình RNN trên từ đầu là hướng dẫn, nó không thuận tiện. Trong phần tiếp theo, chúng ta sẽ thấy cách cải thiện mô hình RNN, chẳng hạn như cách thực hiện dễ dàng hơn và làm cho nó chạy nhanh hơn.

9.5.7 Tóm tắt

- Chúng ta có thể đào tạo mô hình ngôn ngữ cấp ký tự dựa trên RNN để tạo văn bản theo tiền tố văn bản do người dùng cung cấp.
- Một mô hình ngôn ngữ RNN đơn giản bao gồm mã hóa đầu vào, mô hình RNN và tạo ra đầu ra.
- Các mô hình RNN cần khởi tạo trạng thái để đào tạo, mặc dù lấy mẫu ngẫu nhiên và phân vùng tuần tự sử dụng các cách khác nhau.
- Khi sử dụng phân vùng tuần tự, chúng ta cần tách gradient để giảm chi phí tính toán.
- Thời gian khởi động cho phép một mô hình tự cập nhật (ví dụ: có được trạng thái ẩn tốt hơn giá trị khởi tạo của nó) trước khi đưa ra bất kỳ dự đoán nào.
- Gradient clipping ngăn chặn sự bùng nổ gradient, nhưng nó không thể sửa chữa độ dốc biến mất.

9.5.8 Bài tập

1. Cho thấy rằng mã hóa một nóng tương đương với việc chọn một nhúng khác nhau cho mỗi đối tượng.
2. Điều chỉnh các siêu tham số (ví dụ: số kỷ nguyên, số lượng đơn vị ẩn, số bước thời gian trong một minibatch và tốc độ học tập) để cải thiện sự bối rối.
    - Làm thế nào thấp bạn có thể đi?
    - Thay thế mã hóa một nóng bằng các embeddings có thể học được. Điều này có dẫn đến hiệu suất tốt hơn?
    - Nó sẽ hoạt động tốt như thế nào trên các cuốn sách khác của H Gwells, ví dụ, *The War of the Worlds*[105]?
3. Sửa đổi chức năng dự đoán như sử dụng lấy mẫu thay vì chọn ký tự tiếp theo có khả năng cao nhất.
    - Điều gì xảy ra?
    - Thiên vị mô hình hướng tới các đầu ra có khả năng cao hơn, ví dụ, bằng cách lấy mẫu từ $q(x_t \mid x_{t-1}, \ldots, x_1) \propto P(x_t \mid x_{t-1}, \ldots, x_1)^\alpha$ cho $\alpha > 1$.
4. Chạy mã trong phần này mà không cần cắt gradient. Điều gì xảy ra?
5. Thay đổi phân vùng tuần tự để nó không tách các trạng thái ẩn khỏi biểu đồ tính toán. Thời gian chạy có thay đổi không? Làm thế nào về sự bối rối?
6. Thay thế chức năng kích hoạt được sử dụng trong phần này bằng ReLU và lặp lại các thí nghiệm trong phần này. Chúng ta vẫn cần cắt gradient? Tại sao?

Discussions[106]

[105] http://www.gutenberg.org/ebooks/36
[106] https://discuss.d2l.ai/t/486

9.6 Thực hiện ngắn gọn các mạng nơ-ron tái phát

Trong khi Section 9.5 được hướng dẫn để xem RNN được thực hiện như thế nào, điều này không thuận tiện hay nhanh chóng. Phần này sẽ chỉ ra cách triển khai cùng một mô hình ngôn ngữ hiệu quả hơn bằng cách sử dụng các hàm được cung cấp bởi các API cấp cao của một khuôn khổ học sâu. Chúng tôi bắt đầu như trước bằng cách đọc tập dữ liệu máy thời gian.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

9.6.1 Định nghĩa mẫu

API cấp cao cung cấp triển khai các mạng thần kinh định kỳ. Chúng tôi xây dựng lớp mạng thần kinh tái phát `rnn_layer` với một lớp ẩn duy nhất và 256 đơn vị ẩn. Trên thực tế, chúng tôi thậm chí còn chưa thảo luận về ý nghĩa của việc có nhiều lớp — điều này sẽ xảy ra trong Section 10.3. Hiện tại, đủ để nói rằng nhiều lớp chỉ đơn giản là lên đến đầu ra của một lớp RNN được sử dụng làm đầu vào cho lớp tiếp theo của RNN.

```
num_hiddens = 256
rnn_layer = nn.RNN(len(vocab), num_hiddens)
```

Chúng tôi sử dụng một tensor để khởi tạo trạng thái ẩn, có hình dạng là (số lớp ẩn, kích thước lô, số đơn vị ẩn).

```
state = torch.zeros((1, batch_size, num_hiddens))
state.shape
```

```
torch.Size([1, 32, 256])
```

Với trạng thái ẩn và đầu vào, chúng ta có thể tính toán đầu ra với trạng thái ẩn cập nhật. Cần nhấn mạnh rằng "đầu ra" (`Y`) của `rnn_layer` không * không* liên quan đến tính toán các lớp đầu ra: nó đề cập đến trạng thái ẩn ở bước thời gian * mỗi* và chúng có thể được sử dụng làm đầu vào cho lớp đầu ra tiếp theo.

```
X = torch.rand(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, state_new.shape
```

```
(torch.Size([35, 32, 256]), torch.Size([1, 32, 256]))
```

Tương tự như Section 9.5, chúng tôi định nghĩa một lớp `RNNModel` cho một mô hình RNN hoàn chỉnh Lưu ý rằng `rnn_layer` chỉ chứa các lớp tái phát ẩn, chúng ta cần tạo một lớp đầu ra riêng biệt.

```
#@save
class RNNModel(nn.Module):
    """The RNN model."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.rnn.hidden_size
        # If the RNN is bidirectional (to be introduced later),
        # `num_directions` should be 2, else it should be 1.
        if not self.rnn.bidirectional:
            self.num_directions = 1
            self.linear = nn.Linear(self.num_hiddens, self.vocab_size)
        else:
            self.num_directions = 2
            self.linear = nn.Linear(self.num_hiddens * 2, self.vocab_size)

    def forward(self, inputs, state):
        X = F.one_hot(inputs.T.long(), self.vocab_size)
        X = X.to(torch.float32)
        Y, state = self.rnn(X, state)
        # The fully connected layer will first change the shape of `Y` to
        # (`num_steps` * `batch_size`, `num_hiddens`). Its output shape is
        # (`num_steps` * `batch_size`, `vocab_size`).
        output = self.linear(Y.reshape((-1, Y.shape[-1])))
        return output, state

    def begin_state(self, device, batch_size=1):
        if not isinstance(self.rnn, nn.LSTM):
            # `nn.GRU` takes a tensor as hidden state
            return  torch.zeros((self.num_directions * self.rnn.num_layers,
                                 batch_size, self.num_hiddens),
                                device=device)
        else:
            # `nn.LSTM` takes a tuple of hidden states
            return (torch.zeros((
                self.num_directions * self.rnn.num_layers,
                batch_size, self.num_hiddens), device=device),
                    torch.zeros((
                        self.num_directions * self.rnn.num_layers,
                        batch_size, self.num_hiddens), device=device))
```

9.6.2 Đào tạo và dự đoán

Trước khi đào tạo mô hình, chúng ta hãy đưa ra dự đoán với một mô hình có trọng lượng ngẫu nhiên.

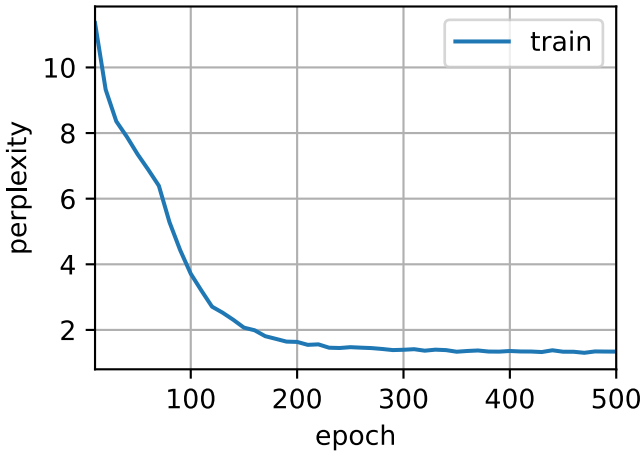
```
device = d2l.try_gpu()
net = RNNModel(rnn_layer, vocab_size=len(vocab))
net = net.to(device)
d2l.predict_ch8('time traveller', 10, net, vocab, device)
```

```
'time travellerxxxxxxxxxx'
```

Như là khá rõ ràng, mô hình này hoàn toàn không hoạt động. Tiếp theo, chúng tôi gọi `train_ch8` với các siêu tham số tương tự được xác định trong Section 9.5 và đào tạo mô hình của chúng tôi với APIs** cấp cao.

```
num_epochs, lr = 500, 1
d2l.train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.3, 286577.9 tokens/sec on cuda:0
time traveller proceeded anyong the eread must uply helo bree th
travelleryou can show black is white by argumed thave back
```



So với phần cuối, mô hình này đạt được sự bối rối tương đương, mặc dù trong một khoảng thời gian ngắn hơn, do mã được tối ưu hóa hơn bởi các API cấp cao của khung học sâu.

9.6.3 Tóm tắt

- API cấp cao của khung học sâu cung cấp một triển khai của lớp RNN.
- Lớp RNN của API cấp cao trả về một đầu ra và trạng thái ẩn cập nhật, trong đó đầu ra không liên quan đến tính toán lớp đầu ra.
- Sử dụng API cấp cao dẫn đến đào tạo RNN nhanh hơn so với sử dụng triển khai từ đầu.

9.6.4 Bài tập

1. Bạn có thể làm cho mô hình RNN overfit bằng cách sử dụng các API cấp cao không?
2. Điều gì sẽ xảy ra nếu bạn tăng số lượng các lớp ẩn trong mô hình RNN? Bạn có thể làm cho mô hình hoạt động?
3. Thực hiện mô hình autoregressive của Section 9.1 bằng cách sử dụng một RNN.

Discussions[107]

[107] https://discuss.d2l.ai/t/1053

9.7 Backpropagation qua thời gian

Cho đến nay chúng tôi đã nhiều lần ám chỉ những thứ như *bùng nổ độ dạng*, *biến mất độ dạng*, và sự cần thiết phải *tách gradient* cho RNNs. Ví dụ, trong Section 9.5, chúng tôi đã gọi hàm `detach` trên trình tự. Không ai trong số này thực sự được giải thích đầy đủ, vì lợi ích của việc có thể xây dựng một mô hình một cách nhanh chóng và xem nó hoạt động như thế nào. Trong phần này, chúng ta sẽ nghiên cứu sâu hơn một chút về các chi tiết về truyền ngược cho các mô hình trình tự và tại sao (và cách thức) toán học hoạt động.

Chúng tôi gặp phải một số hiệu ứng của vụ nổ gradient khi lần đầu tiên chúng tôi triển khai RNN s (Section 9.5). Đặc biệt, nếu bạn giải quyết các bài tập, bạn sẽ thấy rằng cắt gradient là rất quan trọng để đảm bảo sự hội tụ thích hợp. Để hiểu rõ hơn về vấn đề này, phần này sẽ xem xét cách tính toán độ dốc cho các mô hình trình tự. Lưu ý rằng không có gì về mặt khái niệm mới trong cách thức hoạt động của nó. Rốt cuộc, chúng ta vẫn chỉ đơn thuần áp dụng quy tắc chuỗi để tính toán độ dốc. Tuy nhiên, nó là giá trị trong khi xem xét backpropagation (Section 5.7) một lần nữa.

Chúng tôi đã mô tả về phía trước và lạc hậu tuyên truyền và đồ thị tính toán trong MLP s trong Section 5.7. Chuyển tiếp tuyên truyền trong một RNN là tương đối đơn giản. *Backpropagation thông qua thời giới* thực sự là một cụ thể ứng dụng của backpropagation trong RNNs (Werbos, 1990). Nó đòi hỏi chúng ta phải mở rộng biểu đồ tính toán của một RNN một bước một lần tại một thời điểm để có được các phụ thuộc giữa các biến mô hình và tham số. Sau đó, dựa trên quy tắc chuỗi, chúng tôi áp dụng backpropagation để tính toán và lưu trữ gradient. Vì các trình tự có thể khá dài, sự phụ thuộc có thể khá dài. Ví dụ, đối với một chuỗi 1000 ký tự, token đầu tiên có khả năng có thể có ảnh hưởng đáng kể đến mã thông báo ở vị trí cuối cùng. Điều này không thực sự khả thi về mặt tính toán (mất quá nhiều thời gian và đòi hỏi quá nhiều bộ nhớ) và nó đòi hỏi hơn 1000 sản phẩm ma trận trước khi chúng tôi sẽ đến gradient rất khó nắm bắt đó. Đây là một quá trình đầy sự không chắc chắn tính toán và thống kê. Trong phần sau đây, chúng tôi sẽ làm sáng tỏ những gì xảy ra và cách giải quyết vấn đề này trong thực tế.

9.7.1 Phân tích Gradient trong RNNs

Chúng tôi bắt đầu với một mô hình đơn giản hóa về cách thức hoạt động của một RNN. Mô hình này bỏ qua chi tiết về các chi tiết cụ thể của trạng thái ẩn và cách nó được cập nhật. Ký hiệu toán học ở đây không phân biệt rõ ràng vô hướng, vectơ, và ma trận như trước đây. Những chi tiết này không quan trọng đối với phân tích và sẽ chỉ phục vụ để làm lộn xộn ký hiệu trong tiểu mục này.

Trong mô hình đơn giản hóa này, chúng tôi biểu thị h_t là trạng thái ẩn, x_t làm đầu vào và o_t là đầu ra tại bước thời gian t. Nhớ lại các cuộc thảo luận của chúng tôi trong Section 9.4.2 rằng đầu vào và trạng thái ẩn có thể được nối để được nhân với một biến trọng lượng trong lớp ẩn. Do đó, chúng tôi sử dụng w_h và w_o để chỉ ra trọng lượng của lớp ẩn và lớp đầu ra, tương ứng. Kết quả là, các trạng thái ẩn và đầu ra tại mỗi thời điểm các bước có thể được giải thích là

$$\begin{aligned} h_t &= f(x_t, h_{t-1}, w_h), \\ o_t &= g(h_t, w_o), \end{aligned} \tag{9.7.1}$$

trong đó f và g là sự biến đổi của lớp ẩn và lớp đầu ra, tương ứng. Do đó, chúng ta có một chuỗi các giá trị $\{\ldots, (x_{t-1}, h_{t-1}, o_{t-1}), (x_t, h_t, o_t), \ldots\}$ phụ thuộc vào nhau thông qua tính toán lặp lại. Việc tuyên truyền về phía trước là khá đơn giản. Tất cả những gì chúng ta cần là vòng lặp qua (x_t, h_t, o_t) ba lần một bước tại một thời điểm. Sự khác biệt giữa đầu ra o_t và nhãn mong muốn y_t sau đó được đánh giá bởi một chức năng khách quan trên tất cả các bước thời gian T như

$$L(x_1, \ldots, x_T, y_1, \ldots, y_T, w_h, w_o) = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t). \tag{9.7.2}$$

Đối với truyền ngược, các vấn đề phức tạp hơn một chút, đặc biệt là khi chúng ta tính toán các gradient liên quan đến các tham số w_h của hàm khách quan L. Để được cụ thể, theo quy tắc chuỗi,

$$\begin{aligned}\frac{\partial L}{\partial w_h} &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial w_h} \\ &= \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial g(h_t, w_o)}{\partial h_t}\frac{\partial h_t}{\partial w_h}.\end{aligned} \tag{9.7.3}$$

Yếu tố thứ nhất và thứ hai của sản phẩm trong (9.7.3) rất dễ tính toán. Yếu tố thứ ba $\partial h_t/\partial w_h$ là nơi mọi thứ trở nên khó khăn, vì chúng ta cần tính toán lại hiệu ứng của tham số w_h trên h_t. Theo tính toán tái phát vào năm (9.7.1), h_t phụ thuộc vào cả h_{t-1} và w_h, trong đó tính toán h_{t-1} cũng phụ thuộc vào w_h. Do đó, sử dụng sản lượng quy tắc chuỗi

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial w_h}. \tag{9.7.4}$$

Để lấy được gradient trên, giả sử rằng chúng ta có ba chuỗi $\{a_t\}, \{b_t\}, \{c_t\}$ đáp ứng $a_0 = 0$ và $a_t = b_t + c_t a_{t-1}$ cho $t = 1, 2, \ldots$. Sau đó, đối với $t \geq 1$, thật dễ dàng để hiển thị

$$a_t = b_t + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} c_j\right) b_i. \tag{9.7.5}$$

Bằng cách thay thế a_t, b_t, và c_t theo

$$\begin{aligned}a_t &= \frac{\partial h_t}{\partial w_h}, \\ b_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h}, \\ c_t &= \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}},\end{aligned} \tag{9.7.6}$$

tính toán gradient trong (9.7.4) thỏa mãn $a_t = b_t + c_t a_{t-1}$. Do đó, mỗi (9.7.5), chúng ta có thể loại bỏ tính toán tái phát trong (9.7.4) với

$$\frac{\partial h_t}{\partial w_h} = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial f(x_j, h_{j-1}, w_h)}{\partial h_{j-1}}\right)\frac{\partial f(x_i, h_{i-1}, w_h)}{\partial w_h}. \tag{9.7.7}$$

Mặc dù chúng ta có thể sử dụng quy tắc chuỗi để tính toán $\partial h_t/\partial w_h$ đệ quy, chuỗi này có thể nhận được rất dài bất cứ khi nào t lớn. Hãy để chúng tôi thảo luận về một số chiến lược để đối phó với vấn đề này.

Full tính toán

Rõ ràng, chúng ta chỉ có thể tính toán toàn bộ tổng trong (9.7.7). Tuy nhiên, điều này rất chậm và độ dốc có thể nổ tung, vì những thay đổi tinh tế trong điều kiện ban đầu có khả năng ảnh hưởng đến kết quả rất nhiều. Đó là, chúng ta có thể thấy những thứ tương tự như hiệu ứng butterfly trong đó những thay đổi tối thiểu trong điều kiện ban đầu dẫn đến những thay đổi không cân xứng trong kết quả. Điều này thực sự là khá không mong muốn về mô hình mà chúng tôi muốn ước tính. Rốt cuộc, chúng tôi đang tìm kiếm những người dự đoán mạnh mẽ mà khái quát hóa tốt. Do đó chiến lược này hầu như không bao giờ được sử dụng trong thực tế.

Cắt ngắn thời gian bước

Ngoài ra, chúng ta có thể cắt ngắn tổng trong (9.7.7) sau τ bước. Đây là những gì chúng ta đã thảo luận cho đến nay, chẳng hạn như khi chúng ta tách các gradient trong Section 9.5. Điều này dẫn đến một *xấp xỉ * của gradient thật, đơn giản bằng cách chấm dứt tổng tại $\partial h_{t-\tau}/\partial w_h$. Trong thực tế, điều này hoạt động khá tốt. Nó là những gì thường được gọi là backpropgation cắt ngắn qua thời gian (Jaeger, 2002). Một trong những hậu quả của việc này là mô hình tập trung chủ yếu vào ảnh hưởng ngắn hạn chứ không phải là hậu quả lâu dài. Điều này thực sự là * mong muốn*, vì nó thiên vị ước tính đối với các mô hình đơn giản và ổn định hơn.

Ngẫu nhiên Cắt ngắn

Cuối cùng, chúng ta có thể thay thế $\partial h_t/\partial w_h$ bằng một biến ngẫu nhiên đó là chính xác trong kỳ vọng nhưng cắt ngắn chuỗi. Điều này đạt được bằng cách sử dụng một chuỗi ξ_t với $0 \leq \pi_t \leq 1$ được xác định trước, trong đó $P(\xi_t = 0) = 1 - \pi_t$ và $P(\xi_t = \pi_t^{-1}) = \pi_t$, do đó $E[\xi_t] = 1$. Chúng tôi sử dụng điều này để thay thế gradient $\partial h_t/\partial w_h$ trong (9.7.4) bằng

$$z_t = \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial w_h} + \xi_t \frac{\partial f(x_t, h_{t-1}, w_h)}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_h}. \tag{9.7.8}$$

Nó theo định nghĩa của ξ_t rằng $E[z_t] = \partial h_t/\partial w_h$. Bất cứ khi nào $\xi_t = 0$ tính toán tái phát chấm dứt tại thời điểm đó bước t. Điều này dẫn đến một tổng trọng số của các chuỗi có độ dài khác nhau trong đó các chuỗi dài rất hiếm nhưng quá nặng một cách thích hợp. Ý tưởng này được đề xuất bởi Tallec và Ollivier (Tallec & Ollivier, 2017).

So sánh chiến lược

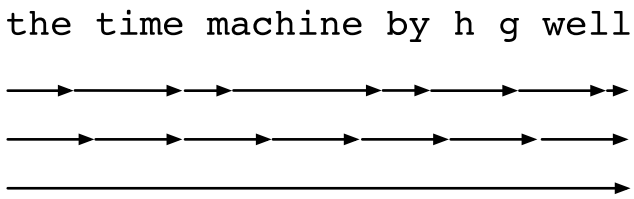


Fig. 9.7.1: Comparing strategies for computing gradients in RNNs. From top to bottom: randomized truncation, regular truncation, and full computation.

Fig. 9.7.1 minh họa ba chiến lược khi phân tích vài ký tự đầu tiên của *The Time Machine* cuốn sách sử dụng backpropagation qua thời gian cho RNNs:

- Hàng đầu tiên là sự cắt ngắn ngẫu nhiên phân vùng văn bản thành các phân đoạn có độ dài khác nhau.
- Hàng thứ hai là sự cắt ngắn thông thường phá vỡ văn bản thành các dãy con có cùng độ dài. Đây là những gì chúng tôi đã làm trong các thí nghiệm RNN.
- Hàng thứ ba là sự lan truyền ngược đầy đủ thông qua thời gian dẫn đến một biểu thức không khả thi về mặt tính toán.

Thật không may, trong khi hấp dẫn về lý thuyết, cắt ngắn ngẫu nhiên không hoạt động tốt hơn nhiều so với cắt ngắn thông thường, rất có thể là do một số yếu tố. Đầu tiên, hiệu quả của một quan sát sau một số bước lan truyền ngược vào quá khứ là khá đủ để nắm bắt các phụ thuộc trong thực tế. Thứ hai, phương sai tăng chống lại thực tế là gradient chính xác hơn với nhiều bước hơn. Thứ ba, chúng tôi thực sự * muốn* mô hình chỉ có

một phạm vi tương tác ngắn. Do đó, thường xuyên cắt ngắn backpropagation thông qua thời gian có một hiệu ứng thường xuyên nhẹ có thể được mong muốn.

9.7.2 Backpropagation qua thời gian chi tiết

Sau khi thảo luận về nguyên tắc chung, chúng ta hãy thảo luận về tuyên truyền ngược qua thời gian một cách chi tiết. Khác với phân tích trong Section 9.7.1, sau đây chúng ta sẽ chỉ ra cách tính độ dốc của hàm mục tiêu đối với tất cả các tham số mô hình bị phân hủy. Để giữ cho mọi thứ đơn giản, chúng ta xem xét một RNN không có tham số thiên vị, có chức năng kích hoạt trong lớp ẩn sử dụng ánh xạ danh tính ($\phi(x) = x$). Đối với bước thời gian t, hãy để đầu vào ví dụ duy nhất và nhãn lần lượt là $\mathbf{x}_t \in \mathbb{R}^d$ và y_t. Trạng thái ẩn $\mathbf{h}_t \in \mathbb{R}^h$ và đầu ra $\mathbf{o}_t \in \mathbb{R}^q$ được tính là

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1},\\ \mathbf{o}_t &= \mathbf{W}_{qh}\mathbf{h}_t,\end{aligned} \tag{9.7.9}$$

trong đó $\mathbf{W}_{hx} \in \mathbb{R}^{h\times d}$, $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, và $\mathbf{W}_{qh} \in \mathbb{R}^{q\times h}$ là các thông số trọng lượng. Biểu thị bởi $l(\mathbf{o}_t, y_t)$ sự mất mát tại bước thời gian t. Chức năng mục tiêu của chúng tôi, sự mất mát trên T bước thời gian từ đầu chuỗi là do đó

$$L = \frac{1}{T}\sum_{t=1}^{T} l(\mathbf{o}_t, y_t). \tag{9.7.10}$$

Để hình dung các phụ thuộc giữa các biến mô hình và tham số trong quá trình tính toán RNN, chúng ta có thể vẽ một biểu đồ tính toán cho mô hình, như thể hiện trong Fig. 9.7.2. Ví dụ, tính toán các trạng thái ẩn của bước thời gian 3, $\mathbf{h}_3$, phụ thuộc vào các thông số mô hình $\mathbf{W}_{hx}$ và $\mathbf{W}_{hh}$, trạng thái ẩn của bước thời gian cuối cùng $\mathbf{h}_2$ và đầu vào của bước thời gian hiện tại $\mathbf{x}_3$.

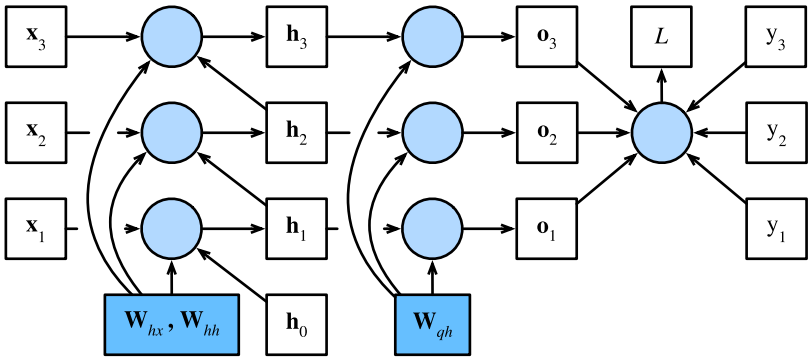


Fig. 9.7.2: Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

Như vừa đề cập, các thông số mô hình trong Fig. 9.7.2 là $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$ và $\mathbf{W}_{qh}$. Nói chung, đào tạo mô hình này yêu cầu tính toán gradient đối với các thông số này $\partial L/\partial \mathbf{W}_{hx}$, $\partial L/\partial \mathbf{W}_{hh}$ và $\partial L/\partial \mathbf{W}_{qh}$. Theo các phụ thuộc trong Fig. 9.7.2, chúng ta có thể đi qua theo hướng ngược lại của các mũi tên để tính toán và lưu trữ các gradient lần lượt. Để thể hiện linh hoạt phép nhân của ma trận, vectơ và vô hướng của các hình dạng khác nhau trong quy tắc chuỗi, chúng ta tiếp tục sử dụng toán tử prod như được mô tả trong Section 5.7.

Trước hết, việc phân biệt chức năng khách quan đối với đầu ra mô hình bất cứ lúc nào t là khá đơn giản:

$$\frac{\partial L}{\partial \mathbf{o}_t} = \frac{\partial l(\mathbf{o}_t, y_t)}{T \cdot \partial \mathbf{o}_t} \in \mathbb{R}^q. \tag{9.7.11}$$

Bây giờ, chúng ta có thể tính toán gradient của hàm mục tiêu đối với tham số $\mathbf{W}_{qh}$ trong lớp đầu ra: $\partial L/\partial \mathbf{W}_{qh} \in \mathbb{R}^{q\times h}$. Dựa trên Fig. 9.7.2, hàm khách quan L phụ thuộc vào $\mathbf{W}_{qh}$ qua $\mathbf{o}_1, \ldots, \mathbf{o}_T$. Sử dụng sản lượng quy tắc chuỗi

$$\frac{\partial L}{\partial \mathbf{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{o}_t} \mathbf{h}_t^\top, \tag{9.7.12}$$

trong đó $\partial L/\partial \mathbf{o}_t$ được đưa ra bởi (9.7.11).

Tiếp theo, như thể hiện trong Fig. 9.7.2, tại bước thời gian cuối cùng T các chức năng khách quan L phụ thuộc vào trạng thái ẩn $\mathbf{h}_T$ chỉ qua $\mathbf{o}_T$. Do đó, chúng ta có thể dễ dàng tìm thấy gradient $\partial L/\partial \mathbf{h}_T \in \mathbb{R}^h$ bằng quy tắc chuỗi:

$$\frac{\partial L}{\partial \mathbf{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_T}, \frac{\partial \mathbf{o}_T}{\partial \mathbf{h}_T}\right) = \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_T}. \tag{9.7.13}$$

Nó trở nên phức tạp hơn cho bất kỳ bước thời gian $t < T$, nơi chức năng khách quan L phụ thuộc vào $\mathbf{h}_t$ qua $\mathbf{h}_{t+1}$ và $\mathbf{o}_t$. Theo quy tắc chuỗi, gradient của trạng thái ẩn $\partial L/\partial \mathbf{h}_t \in \mathbb{R}^h$ bất cứ lúc nào bước $t < T$ có thể được tính toán một cách lặp lại như:

$$\frac{\partial L}{\partial \mathbf{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_{t+1}}, \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \mathbf{o}_t}, \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t}\right) = \mathbf{W}_{hh}^\top \frac{\partial L}{\partial \mathbf{h}_{t+1}} + \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_t}. \tag{9.7.14}$$

Để phân tích, mở rộng tính toán định kỳ cho bất kỳ bước thời gian $1 \leq t \leq T$ cho

$$\frac{\partial L}{\partial \mathbf{h}_t} = \sum_{i=t}^{T} \left(\mathbf{W}_{hh}^\top\right)^{T-i} \mathbf{W}_{qh}^\top \frac{\partial L}{\partial \mathbf{o}_{T+t-i}}. \tag{9.7.15}$$

Chúng ta có thể thấy từ (9.7.15) rằng ví dụ tuyến tính đơn giản này đã thể hiện một số vấn đề chính của các mô hình chuỗi dài: nó liên quan đến sức mạnh có khả năng rất lớn của $\mathbf{W}_{hh}^\top$. Trong đó, eigenvalues nhỏ hơn 1 biến mất và eigenvalues lớn hơn 1 phân kỳ. Điều này không ổn định về số lượng, biểu hiện dưới dạng biến mất và bùng nổ gradient. Một cách để giải quyết vấn đề này là cắt ngắn các bước thời gian ở kích thước thuận tiện về mặt tính toán như đã thảo luận trong Section 9.7.1. Trong thực tế, sự cắt ngắn này được thực hiện bằng cách tách gradient sau một số bước thời gian nhất định. Sau đó, chúng ta sẽ thấy các mô hình trình tự phức tạp hơn như bộ nhớ ngắn hạn dài có thể làm giảm bớt điều này hơn nữa.

Cuối cùng, Fig. 9.7.2 cho thấy hàm khách quan L phụ thuộc vào các tham số mô hình $\mathbf{W}_{hx}$ và $\mathbf{W}_{hh}$ trong lớp ẩn thông qua các trạng thái ẩn $\mathbf{h}_1, \ldots, \mathbf{h}_T$. Để tính toán gradient liên quan đến các tham số như vậy $\partial L/\partial \mathbf{W}_{hx} \in \mathbb{R}^{h\times d}$ và $\partial L/\partial \mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$, chúng tôi áp dụng quy tắc chuỗi cung cấp

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{W}_{hx}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hx}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{x}_t^\top,\\ \frac{\partial L}{\partial \mathbf{W}_{hh}} &= \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \mathbf{h}_t}, \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{h}_t} \mathbf{h}_{t-1}^\top,\end{aligned} \tag{9.7.16}$$

trong đó $\partial L/\partial \mathbf{h}_t$ được tính toán lặp đi lặp lại bởi (9.7.13) và (9.7.14) là đại lượng chính ảnh hưởng đến sự ổn định số.

Kể từ khi tuyên truyền ngược qua thời gian là việc áp dụng tuyên truyền ngược trong RNN s, như chúng tôi đã giải thích trong Section 5.7, đào tạo RNN s xen kẽ tuyên truyền về phía trước với truyền ngược qua thời gian. Bên cạnh đó, backpropagation thông qua thời gian tính toán và lưu trữ các gradient trên lần lượt. Cụ thể, các giá trị trung gian được lưu trữ được tái sử dụng để tránh tính toán trùng lặp, chẳng hạn như lưu trữ $\partial L/\partial \mathbf{h}_t$ được sử dụng trong tính toán cả $\partial L/\partial \mathbf{W}_{hx}$ và $\partial L/\partial \mathbf{W}_{hh}$.

9.7.3 Tóm tắt

- Backpropagation qua thời gian chỉ đơn thuần là một ứng dụng của backpropagation cho các mô hình trình tự với một trạng thái ẩn.
- Cắt ngắn là cần thiết để thuận tiện tính toán và ổn định số, chẳng hạn như cắt ngắn thường xuyên và cắt ngắn ngẫu nhiên.
- Sức mạnh cao của ma trận có thể dẫn đến sự khác biệt hoặc biến mất eigenvalues. Điều này thể hiện chính nó dưới dạng bùng nổ hoặc biến mất gradient.
- Để tính toán hiệu quả, các giá trị trung gian được lưu trữ trong quá trình truyền ngược qua thời gian.

9.7.4 Bài tập

1. Giả sử rằng chúng ta có một ma trận đối xứng $\mathbf{M} \in \mathbb{R}^{n \times n}$ với eigenvalues λ_i có eigenvectors tương ứng là $\mathbf{v}_i$ $(i = 1, \ldots, n)$. Không mất tính tổng quát, giả định rằng chúng được đặt hàng theo thứ tự $|\lambda_i| \geq |\lambda_{i+1}|$.
2. Cho thấy $\mathbf{M}^k$ có eigenvalues λ_i^k.
3. Chứng minh rằng đối với một vector ngẫu nhiên $\mathbf{x} \in \mathbb{R}^n$, với xác suất cao $\mathbf{M}^k\mathbf{x}$ sẽ rất phù hợp với eigenvector $\mathbf{v}_1$ của $\mathbf{M}$. Chính thức hóa tuyên bố này.
4. Kết quả trên có ý nghĩa gì đối với gradient trong RNNs?
5. Bên cạnh việc cắt gradient, bạn có thể nghĩ về bất kỳ phương pháp nào khác để đối phó với sự bùng nổ gradient trong các mạng thần kinh tái phát không?

Discussions[108]

[108] https://discuss.d2l.ai/t/334

10 | Mạng thần kinh tái phát hiện đại

Chúng tôi đã giới thiệu những điều cơ bản về RNNs, có thể xử lý dữ liệu trình tự tốt hơn. Để trình diễn, chúng tôi đã triển khai các mô hình ngôn ngữ dựa trên RNN trên dữ liệu văn bản. Tuy nhiên, những kỹ thuật như vậy có thể không đủ cho các học viên khi họ phải đối mặt với một loạt các vấn đề học tập chuỗi ngày nay.

Ví dụ, một vấn đề đáng chú ý trong thực tế là sự bất ổn số của RNNs. Mặc dù chúng tôi đã áp dụng các thủ thuật triển khai như cắt gradient, vấn đề này có thể được giảm bớt hơn nữa với các thiết kế phức tạp hơn của các mô hình trình tự. Cụ thể, RNN có cổng phổ biến hơn nhiều trong thực tế. Chúng tôi sẽ bắt đầu bằng cách giới thiệu hai trong số các mạng được sử dụng rộng rãi như vậy, cụ thể là các đơn vị định kỳ *ged* (Grus) và * bộ nhớ ngắn hạn dài* (LSTM). Hơn nữa, chúng tôi sẽ mở rộng kiến trúc RNN với một layer ẩn một chiều duy nhất đã được thảo luận cho đến nay. Chúng tôi sẽ mô tả các kiến trúc sâu với nhiều lớp ẩn, và thảo luận về thiết kế hai chiều với cả tính toán tái phát về phía trước và ngược. Những mở rộng như vậy thường được áp dụng trong các mạng tái phát hiện đại. Khi giải thích các biến thể RNN này, chúng tôi tiếp tục xem xét vấn đề mô hình hóa ngôn ngữ tương tự được giới thiệu trong Chapter 9.

Trên thực tế, mô hình ngôn ngữ chỉ tiết lộ một phần nhỏ trong số những gì trình tự học có khả năng. Trong một loạt các vấn đề học tập trình tự, chẳng hạn như nhận dạng giọng nói tự động, văn bản sang giọng nói và dịch máy, cả đầu vào và đầu ra đều là các chuỗi có độ dài tùy ý. Để giải thích làm thế nào để phù hợp với loại dữ liệu này, chúng tôi sẽ lấy dịch máy làm ví dụ, và giới thiệu kiến trúc bộ mã hóa-giải mã dựa trên RNNs và tìm kiếm chùm tia để tạo chuỗi.

10.1 Các đơn vị định kỳ cổng (GRU)

Trong Section 9.7, chúng tôi đã thảo luận về cách tính độ dốc trong RNNs. Đặc biệt, chúng tôi thấy rằng các sản phẩm dài của ma trận có thể dẫn đến biến mất hoặc bùng nổ gradient. Chúng ta hãy suy nghĩ ngắn gọn về những bất thường gradient như vậy có nghĩa là gì trong thực tế:

- Chúng ta có thể gặp phải một tình huống mà một quan sát sớm là rất quan trọng để dự đoán tất cả các quan sát trong tương lai. Hãy xem xét trường hợp hơi contrived trong đó quan sát đầu tiên chứa một tổng kiểm tra và mục tiêu là để phân biệt xem tổng kiểm tra có đúng ở cuối chuỗi hay không. Trong trường hợp này, ảnh hưởng của mã thông báo đầu tiên là rất quan trọng. Chúng tôi muốn có một số cơ chế để lưu trữ thông tin ban đầu quan trọng trong một tế bào bộ nhớ*. Không có cơ chế như vậy, chúng ta sẽ phải gán một gradient rất lớn cho quan sát này, vì nó ảnh hưởng đến tất cả các quan sát tiếp theo.
- Chúng ta có thể gặp phải những tình huống mà một số token không có quan sát thích hợp. Ví dụ, khi phân tích một trang web có thể có mã HTML phụ trợ không liên quan với mục đích đánh giá tình cảm được truyền tải trên trang. Chúng tôi muốn có một số cơ chế để * bỏ qua* token như vậy trong đại diện trạng thái tiềm ẩn.
- Chúng ta có thể gặp phải các tình huống trong đó có một sự phá vỡ logic giữa các phần của một chuỗi. Ví dụ, có thể có sự chuyển đổi giữa các chương trong một cuốn sách hoặc chuyển đổi giữa một con gấu

và thị trường tăng giá cho chứng khoán. Trong trường hợp này, thật tuyệt khi có một phương tiện * đặt lại* đại diện trạng thái nội bộ của chúng tôi.

Một số phương pháp đã được đề xuất để giải quyết vấn đề này. Một trong những sớm nhất là bộ nhớ ngắn hạn dài (Hochreiter & Schmidhuber, 1997) mà chúng ta sẽ thảo luận trong Section 10.2. Đơn vị định kỳ gated (GRU) (Cho et al., 2014a) là một biến thể sắp xếp hợp lý hơn một chút thường cung cấp hiệu suất tương đương và nhanh hơn đáng kể để tính toán (Chung et al., 2014). Do sự đơn giản của nó, chúng ta hãy bắt đầu với GRU.

10.1.1 Nhà nước ẩn Gated

Sự khác biệt chính giữa vanilla RNNs và Grus là hỗ trợ sau này gating của trạng thái ẩn. Điều này có nghĩa là chúng ta có các cơ chế dành riêng cho khi một trạng thái ẩn nên được * cập nhật* và cả khi nó nên được * reset*. Những cơ chế này được học và chúng giải quyết các mối quan tâm được liệt kê ở trên. Ví dụ, nếu token đầu tiên có tầm quan trọng lớn, chúng ta sẽ học cách không cập nhật trạng thái ẩn sau lần quan sát đầu tiên. Tương tự như vậy, chúng ta sẽ học cách bỏ qua các quan sát tạm thời không liên quan. Cuối cùng, chúng ta sẽ học cách đặt lại trạng thái tiềm ẩn bất cứ khi nào cần thiết. Chúng tôi thảo luận chi tiết về điều này dưới đây.

Cổng đặt lại và cổng cập nhật

Điều đầu tiên chúng ta cần giới thiệu là cổng *reset * và cổng cập nhật*. Chúng tôi kỹ sư chúng trở thành vectơ với các mục trong $(0, 1)$ sao cho chúng tôi có thể thực hiện các kết hợp lồi. Ví dụ, một cổng đặt lại sẽ cho phép chúng ta kiểm soát số lượng trạng thái trước đó mà chúng ta vẫn có thể muốn nhớ. Tương tự như vậy, một cổng cập nhật sẽ cho phép chúng ta kiểm soát bao nhiêu trạng thái mới chỉ là một bản sao của trạng thái cũ.

Chúng tôi bắt đầu bằng kỹ thuật các cổng này. Fig. 10.1.1 minh họa các đầu vào cho cả cổng đặt lại và cập nhật trong GRU, với đầu vào của bước thời gian hiện tại và trạng thái ẩn của bước thời gian trước đó. Các đầu ra của hai cổng được đưa ra bởi hai lớp được kết nối hoàn toàn với chức năng kích hoạt sigmoid.

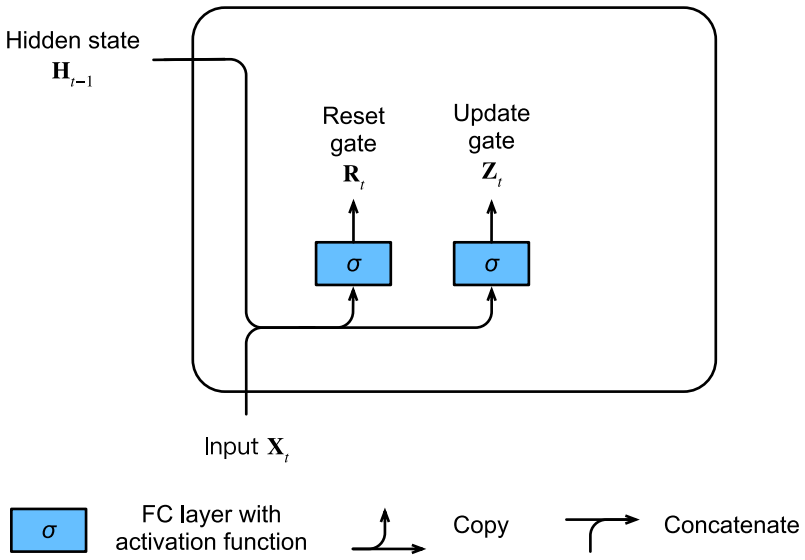


Fig. 10.1.1: Computing the reset gate and the update gate in a GRU model.

Về mặt toán học, cho một bước thời gian nhất định t, giả sử rằng đầu vào là một minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (số ví dụ: n, số lượng đầu vào: d) và trạng thái ẩn của bước thời gian trước đó là $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ (số đơn vị ẩn: h).

Sau đó, cổng đặt lại $\mathbf{R}_t \in \mathbb{R}^{n \times h}$ và cổng cập nhật $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$ được tính như sau:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r),\\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z),\end{aligned} \tag{10.1.1}$$

trong đó $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$ và $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$ là các thông số trọng lượng và $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$ là thiên vị. Lưu ý rằng phát sóng (xem Section 3.1.3) được kích hoạt trong quá trình tổng kết. Chúng tôi sử dụng các hàm sigmoid (như được giới thiệu trong Section 5.1) để chuyển đổi các giá trị đầu vào thành khoảng $(0, 1)$.

Ứng viên Hidden State

Tiếp theo, chúng ta hãy tích hợp cổng đặt lại $\mathbf{R}_t$ với cơ chế cập nhật trạng thái tiềm ẩn thông thường trong (9.4.5). Nó dẫn đến những điều sau đây *trạng thái ẩn ứng cử viên * $\tilde{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ tại bước thời gian t:

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h), \tag{10.1.2}$$

trong đó $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ và $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ là các thông số trọng lượng, $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ là thiên vị và ký hiệu $\odot$ là nhà điều hành sản phẩm Hadamard (elementwise). Ở đây chúng ta sử dụng một phi tuyến dưới dạng tanh để đảm bảo rằng các giá trị trong trạng thái ẩn ứng viên vẫn còn trong khoảng $(-1, 1)$.

Kết quả là *ứng cử viên* vì chúng ta vẫn cần kết hợp hành động của cổng cập nhật. So sánh với (9.4.5), bây giờ ảnh hưởng của các trạng thái trước đó có thể được giảm với phép nhân elementwise $\mathbf{R}_t$ và $\mathbf{H}_{t-1}$ trong (10.1.2). Bất cứ khi nào các mục trong cổng đặt lại $\mathbf{R}_t$ gần 1, chúng tôi phục hồi một RNN vani như trong (9.4.5). Đối với tất cả các mục của cổng đặt lại $\mathbf{R}_t$ gần 0, trạng thái ẩn ứng cử viên là kết quả của MLP với $\mathbf{X}_t$ làm đầu vào. Do đó, bất kỳ trạng thái ẩn đã tồn tại trước nào là * đặt lại* thành mặc định.

Fig. 10.1.2 minh họa dòng tính toán sau khi áp dụng cổng đặt lại.

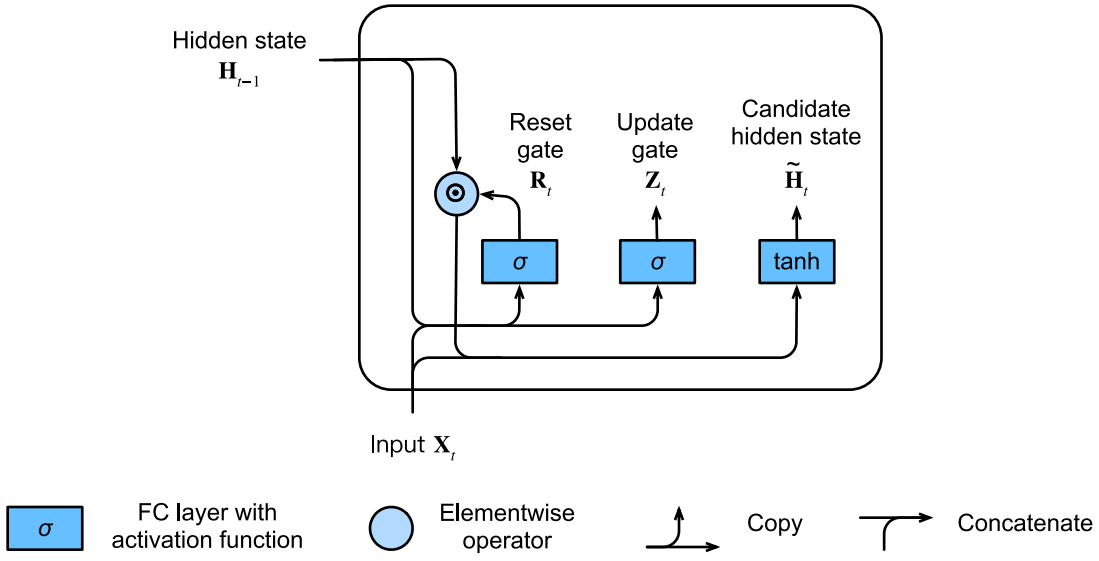


Fig. 10.1.2: Computing the candidate hidden state in a GRU model.

Cuối cùng, chúng ta cần kết hợp hiệu ứng của cổng cập nhật $\mathbf{Z}_t$. Điều này xác định mức độ mà trạng thái ẩn mới $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ chỉ là trạng thái cũ $\mathbf{H}_{t-1}$ và bằng cách sử dụng nhà nước ứng cử viên mới $\tilde{\mathbf{H}}_t$. Cổng cập nhật $\mathbf{Z}_t$ có thể được sử dụng cho mục đích này, đơn giản bằng cách dùng kết hợp lồi elementwise giữa cả $\mathbf{H}_{t-1}$ và $\tilde{\mathbf{H}}_t$. Điều này dẫn đến phương trình cập nhật cuối cùng cho GRU:

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \tag{10.1.3}$$

Bất cứ khi nào cổng cập nhật $\mathbf{Z}_t$ gần 1, chúng tôi chỉ đơn giản là giữ lại trạng thái cũ. Trong trường hợp này, thông tin từ $\mathbf{X}_t$ về cơ bản bị bỏ qua, có hiệu quả bỏ qua bước thời gian t trong chuỗi phụ thuộc. Ngược lại, bất cứ khi nào $\mathbf{Z}_t$ gần 0, trạng thái tiềm ẩn mới $\mathbf{H}_t$ tiếp cận trạng thái tiềm ẩn ứng cử viên $\tilde{\mathbf{H}}_t$. Những thiết kế này có thể giúp chúng ta đối phó với vấn đề gradient biến mất trong RNNs và nắm bắt các phụ thuộc tốt hơn cho các chuỗi với khoảng cách bước thời gian lớn. Ví dụ, nếu cổng cập nhật đã gần 1 cho tất cả các bước thời gian của toàn bộ dãy con, trạng thái ẩn cũ tại bước thời gian bắt đầu của nó sẽ dễ dàng được giữ lại và truyền đến cuối của nó, bất kể độ dài của dãy con.

Fig. 10.1.3 minh họa luồng tính toán sau khi cổng cập nhật đang hoạt động.

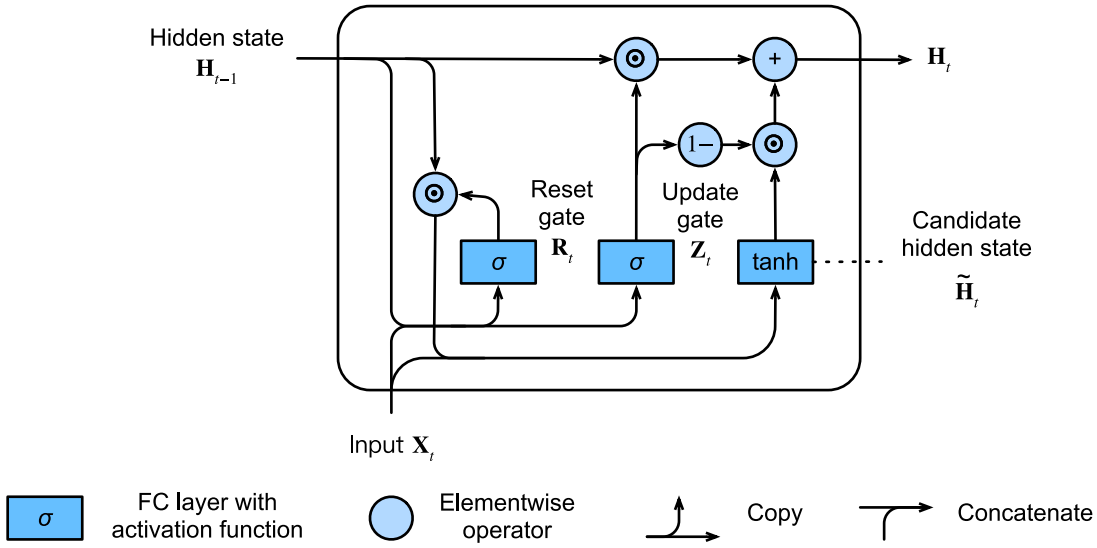


Fig. 10.1.3: Computing the hidden state in a GRU model.

Tóm lại, Grus có hai đặc điểm phân biệt sau:

- Đặt lại cổng giúp nắm bắt các phụ thuộc ngắn hạn trong chuỗi.
- Cập nhật cổng giúp nắm bắt các phụ thuộc dài hạn trong chuỗi.

10.1.2 Thực hiện từ đầu

Để hiểu rõ hơn về mô hình GRU, chúng ta hãy thực hiện nó từ đầu. Chúng tôi bắt đầu bằng cách đọc tập dữ liệu máy thời gian mà chúng tôi đã sử dụng trong Section 9.5. Mã để đọc tập dữ liệu được đưa ra dưới đây.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

Bước tiếp theo là khởi tạo các tham số mô hình. Chúng tôi vẽ các trọng lượng từ phân phối Gaussian với độ lệch chuẩn là 0,01 và đặt thiên vị thành 0. Siêu tham số `num_hiddens` xác định số đơn vị ẩn. Chúng tôi khởi tạo tất cả các trọng lượng và thành kiến liên quan đến cổng cập nhật, cổng đặt lại, trạng thái ẩn ứng cử viên và lớp đầu ra.

```
def get_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xz, W_hz, b_z = three()  # Update gate parameters
    W_xr, W_hr, b_r = three()  # Reset gate parameters
    W_xh, W_hh, b_h = three()  # Candidate hidden state parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Xác định mô hình

Bây giờ chúng ta sẽ định nghĩa chức năng khởi tạo trạng thái ẩn `init_gru_state`. Cũng giống như hàm `init_rnn_state` được định nghĩa trong Section 9.5, hàm này trả về một tensor với một hình dạng (kích thước lô, số đơn vị ẩn) có giá trị là tất cả các số không.

```
def init_gru_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device), )
```

Bây giờ chúng tôi đã sẵn sàng để xác định mô hình GRU. Cấu trúc của nó giống như của ô RNN cơ bản, ngoại trừ các phương trình cập nhật phức tạp hơn.

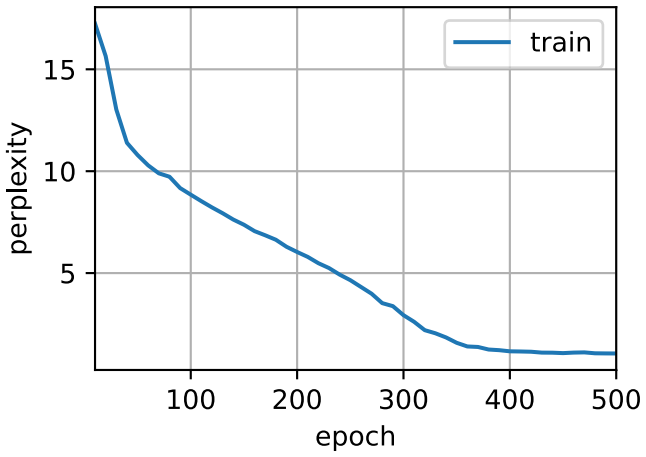
```
def gru(inputs, state, params):
    W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        Z = torch.sigmoid((X @ W_xz) + (H @ W_hz) + b_z)
        R = torch.sigmoid((X @ W_xr) + (H @ W_hr) + b_r)
        H_tilda = torch.tanh((X @ W_xh) + ((R * H) @ W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = H @ W_hq + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H,)
```

Đào tạo và dự đoán

Đào tạo và dự đoán hoạt động theo cách chính xác giống như trong Section 9.5. Sau khi đào tạo, chúng tôi in ra sự bối rối trên bộ đào tạo và trình tự dự đoán theo các tiền tố được cung cấp "khách du lịch thời gian" và "khách du lịch", tương ứng.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_params,
                            init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 26525.7 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```

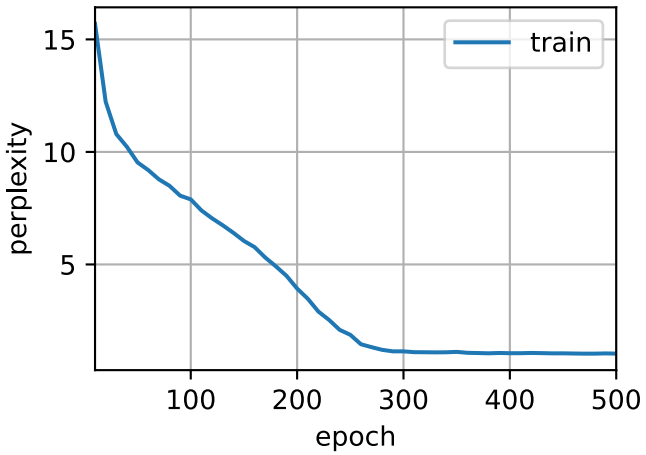


10.1.3 Thiết tập

Trong các API cấp cao, chúng ta có thể trực tiếp khởi tạo mô hình GPU. Điều này đóng gói tất cả các chi tiết cấu hình mà chúng tôi đã thực hiện rõ ràng ở trên. Mã này nhanh hơn đáng kể vì nó sử dụng các toán tử được biên dịch hơn là Python cho nhiều chi tiết mà chúng tôi đã nêu ra trước đó.

```
num_inputs = vocab_size
gru_layer = nn.GRU(num_inputs, num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 323876.1 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



10.1.4 Tóm tắt

- Các RNN có cổng có thể nắm bắt các phụ thuộc tốt hơn cho các trình tự với khoảng cách bước thời gian lớn.
- Đặt lại cổng giúp nắm bắt các phụ thuộc ngắn hạn trong chuỗi.
- Cập nhật cổng giúp nắm bắt các phụ thuộc dài hạn trong chuỗi.
- Grus chứa RNN cơ bản làm trường hợp cực đoan của chúng bất cứ khi nào cổng đặt lại được bật. Họ cũng có thể bỏ qua dãy con bằng cách bật cổng cập nhật.

10.1.5 Bài tập

1. Giả sử rằng chúng ta chỉ muốn sử dụng đầu vào tại bước thời gian t' để dự đoán đầu ra tại bước thời gian $t > t'$. Các giá trị tốt nhất cho các thiết lập lại và cập nhật cổng cho mỗi bước thời gian là gì?
2. Điều chỉnh các siêu tham số và phân tích ảnh hưởng của chúng đối với thời gian chạy, bối rối và trình tự đầu ra.
3. So sánh thời gian chạy, bối rối và các chuỗi đầu ra cho `rnn.RNN` và `rnn.GRU` triển khai với nhau.
4. Điều gì xảy ra nếu bạn chỉ triển khai các phần của GRU, ví dụ, chỉ với một cổng đặt lại hoặc chỉ một cổng cập nhật?

Discussions[109]

10.2 Bộ nhớ ngắn hạn dài (LSTM)

Thách thức giải quyết việc bảo tồn thông tin dài hạn và bỏ qua đầu vào ngắn hạn trong các mô hình biến tiềm ẩn đã tồn tại trong một thời gian dài. Một trong những cách tiếp cận sớm nhất để giải quyết vấn đề này là bộ nhớ ngắn hạn dài (LSTM) (Hochreiter & Schmidhuber, 1997). Nó chia sẻ nhiều tài sản của GRU. Điều thú vị là LSTMs có thiết kế phức tạp hơn một chút so với Grus nhưng trước Grus gần hai thập kỷ.

10.2.1 Gated Memory Cell

Được cho là thiết kế của LSTM được lấy cảm hứng từ cổng logic của máy tính. LSTM giới thiệu một tế bào *memory cell* (hoặc *cell* nói ngắn) có hình dạng giống như trạng thái ẩn (một số văn học coi ô nhớ là một loại đặc biệt của trạng thái ẩn), được thiết kế để ghi lại thông tin bổ sung. Để kiểm soát tế bào bộ nhớ, chúng ta cần một số cổng. Một cổng là cần thiết để đọc các mục từ ô. Chúng tôi sẽ đề cập đến điều này là *cổng đầu ra*. Một cổng thứ hai là cần thiết để quyết định khi nào nên đọc dữ liệu vào ô. Chúng tôi gọi đây là cổng đầu vào **. Cuối cùng, chúng ta cần một cơ chế để thiết lập lại nội dung của ô, được chi phối bởi một cổng * quên *. Động lực cho một thiết kế như vậy giống như của Grus, cụ thể là có thể quyết định khi nào cần nhớ và khi nào bỏ qua các đầu vào ở trạng thái ẩn thông qua một cơ chế chuyên dụng. Hãy để chúng tôi xem làm thế nào điều này hoạt động trong thực tế.

Cổng đầu vào, Cổng quên và Cổng đầu ra

Cũng giống như trong Grus, việc cấp dữ liệu vào cổng LSTM là đầu vào ở bước thời gian hiện tại và trạng thái ẩn của bước thời gian trước đó, như minh họa trong Fig. 10.2.1. Chúng được xử lý bởi ba lớp được kết nối hoàn toàn với chức năng kích hoạt sigmoid để tính toán các giá trị của đầu vào, quê. và cổng đầu ra. Kết quả là, các giá trị của ba cổng nằm trong khoảng $(0, 1)$.

[109] https://discuss.d2l.ai/t/1056

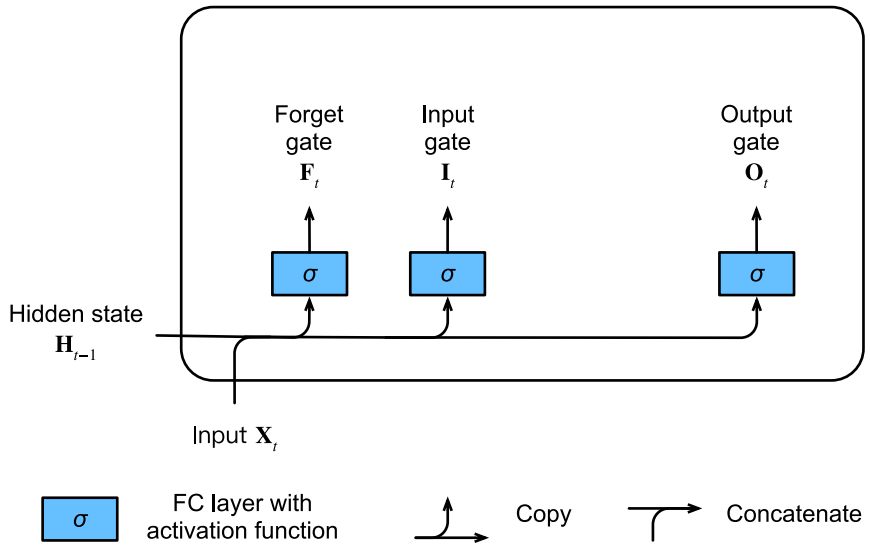


Fig. 10.2.1: Computing the input gate, the forget gate, and the output gate in an LSTM model.

Về mặt toán học, giả sử rằng có h đơn vị ẩn, kích thước lô là n và số lượng đầu vào là d. Do đó, đầu vào là $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ và trạng thái ẩn của bước thời gian trước là $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$. Tương ứng, các cổng tại bước thời điểm t được định nghĩa như sau: cổng đầu vào là $\mathbf{I}_t \in \mathbb{R}^{n \times h}$, cổng quên là $\mathbf{F}_t \in \mathbb{R}^{n \times h}$ và cổng đầu ra là $\mathbf{O}_t \in \mathbb{R}^{n \times h}$. Chúng được tính như sau:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i),\\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f),\\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned} \tag{10.2.1}$$

trong đó $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$ và $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h \times h}$ là các thông số trọng lượng và $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$ là các thông số thiên vị.

Tế bào bộ nhớ ứng cử viên

Tiếp theo chúng tôi thiết kế các tế bào bộ nhớ. Vì chúng tôi chưa chỉ định hành động của các cổng khác nhau, trước tiên chúng tôi giới thiệu tế bào bộ nhớ *ứng cử viên $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$. Tính toán của nó tương tự như của ba cổng được mô tả ở trên, nhưng sử dụng hàm tanh với phạm vi giá trị cho $(-1, 1)$ làm hàm kích hoạt. Điều này dẫn đến phương trình sau tại bước thời điểm t:

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c), \tag{10.2.2}$$

trong đó $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ và $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ là các thông số trọng lượng và $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ là một tham số thiên vị.

Một minh họa nhanh chóng của tế bào bộ nhớ ứng cử viên được thể hiện trong Fig. 10.2.2.

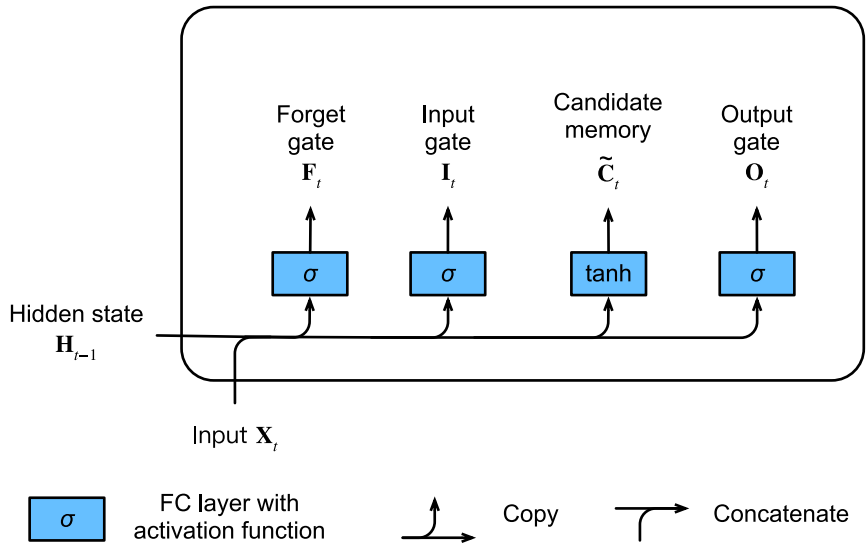


Fig. 10.2.2: Computing the candidate memory cell in an LSTM model.

Bộ nhớ Cell

Trong Grus, chúng ta có một cơ chế để chi phối đầu vào và quên (hoặc bỏ qua). Tương tự, trong LSTMs, chúng tôi có hai cổng chuyên dụng cho các mục đích như vậy: cổng đầu vào $\mathbf{I}_t$ chi phối số lượng chúng tôi tính đến dữ liệu mới thông qua $\tilde{\mathbf{C}}_t$ và cổng quên $\mathbf{F}_t$ giải quyết bao nhiêu nội dung tế bào bộ nhớ cũ $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ chúng tôi giữ lại. Sử dụng cùng một thủ thuật nhân theo chiều ngang như trước đây, chúng tôi đến phương trình cập nhật sau:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \tag{10.2.3}$$

Nếu cổng quên luôn xấp xỉ 1 và cổng đầu vào luôn xấp xỉ 0, các ô bộ nhớ quá khứ $\mathbf{C}_{t-1}$ sẽ được lưu theo thời gian và được chuyển sang bước thời gian hiện tại. Thiết kế này được giới thiệu để giảm bớt vấn đề gradient biến mất và để nắm bắt tốt hơn các phụ thuộc tầm xa trong chuỗi.

Do đó, chúng tôi đến sơ đồ dòng chảy trong Fig. 10.2.3.

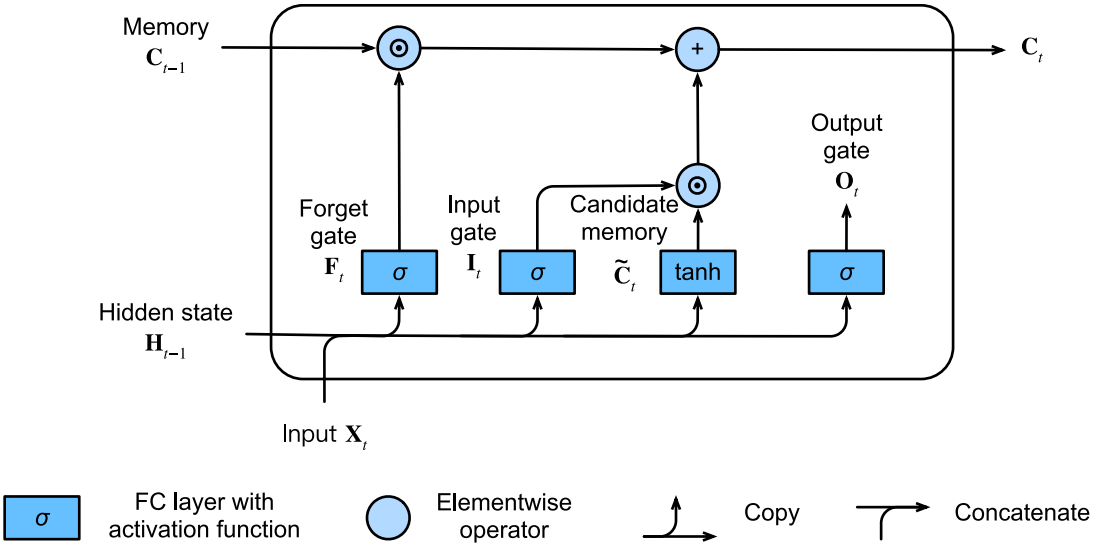


Fig. 10.2.3: Computing the memory cell in an LSTM model.

Nhà nước ẩn

Cuối cùng, chúng ta cần xác định cách tính trạng thái ẩn $\mathbf{H}_t \in \mathbb{R}^{n \times h}$. Đây là nơi cổng đầu ra phát huy tác dụng. Trong LSTM, nó chỉ đơn giản là một phiên bản cổng của tanh của tế bào bộ nhớ. Điều này đảm bảo rằng các giá trị của $\mathbf{H}_t$ luôn nằm trong khoảng $(-1, 1)$.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \tag{10.2.4}$$

Bất cứ khi nào cổng đầu ra xấp xỉ 1, chúng tôi có hiệu quả truyền tất cả thông tin bộ nhớ đến bộ dự đoán, trong khi đối với cổng đầu ra gần 0, chúng tôi chỉ giữ lại tất cả thông tin trong ô nhớ và không thực hiện xử lý thêm.

Fig. 10.2.4 có một minh họa đồ họa của luồng dữ liệu.

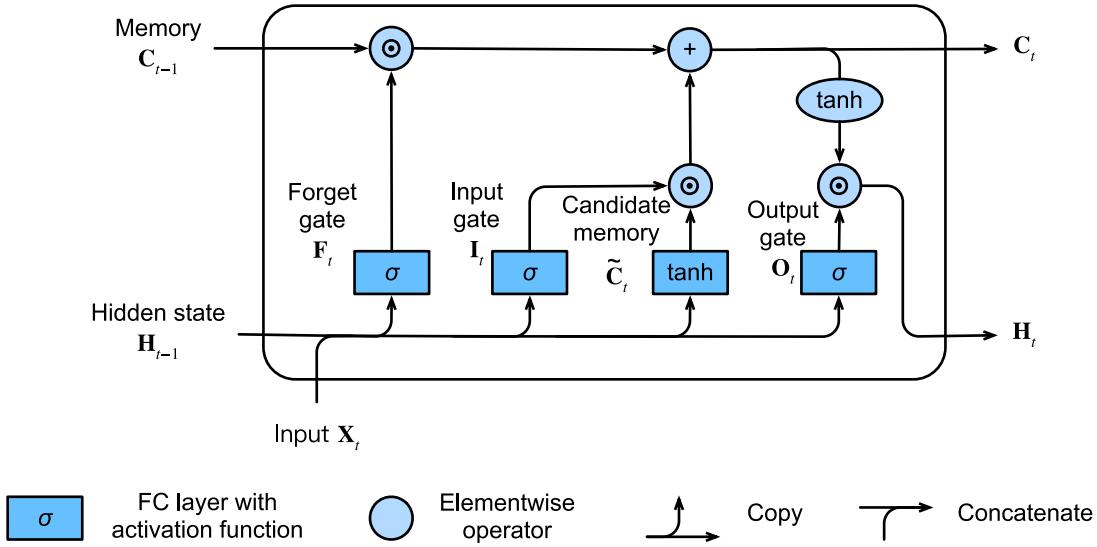


Fig. 10.2.4: Computing the hidden state in an LSTM model.

10.2.2 Thực hiện từ đầu

Bây giờ chúng ta hãy thực hiện một LSTM từ đầu. Giống như các thí nghiệm trong Section 9.5, lần đầu tiên chúng ta tải tập dữ liệu máy thời gian.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Initializing Model Parameters

Tiếp theo chúng ta cần xác định và khởi tạo các tham số mô hình. Như trước đây, siêu tham số `num_hiddens` xác định số đơn vị ẩn. Chúng tôi khởi tạo các trọng lượng sau một phân phối Gaussian với 0,01 độ lệch chuẩn, và chúng tôi đặt các thành kiến là 0.

```
def get_lstm_params(vocab_size, num_hiddens, device):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device)*0.01

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    W_xi, W_hi, b_i = three()  # Input gate parameters
    W_xf, W_hf, b_f = three()  # Forget gate parameters
    W_xo, W_ho, b_o = three()  # Output gate parameters
    W_xc, W_hc, b_c = three()  # Candidate memory cell parameters
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
    # Attach gradients
    params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
              b_c, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)
    return params
```

Xác định mô hình

Trong the initialization function, trạng thái ẩn của LSTM cần trả về một ô bộ nhớ *additional* với giá trị 0 và hình dạng (kích thước lô, số đơn vị ẩn). Do đó chúng tôi nhận được sự khởi tạo trạng thái sau đây.

```
def init_lstm_state(batch_size, num_hiddens, device):
    return (torch.zeros((batch_size, num_hiddens), device=device),
            torch.zeros((batch_size, num_hiddens), device=device))
```

mô hình thực tế được định nghĩa giống như những gì chúng ta đã thảo luận trước đây: cung cấp ba cổng và một tế bào bộ nhớ phụ trợ. Lưu ý rằng chỉ có trạng thái ẩn được chuyển đến lớp đầu ra. Các tế bào bộ nhớ $\mathbf{C}_t$ không trực tiếp tham gia vào việc tính toán đầu ra.

```
def lstm(inputs, state, params):
    [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
     W_hq, b_q] = params
    (H, C) = state
    outputs = []
    for X in inputs:
        I = torch.sigmoid((X @ W_xi) + (H @ W_hi) + b_i)
        F = torch.sigmoid((X @ W_xf) + (H @ W_hf) + b_f)
        O = torch.sigmoid((X @ W_xo) + (H @ W_ho) + b_o)
```

(continues on next page)

(continued from previous page)

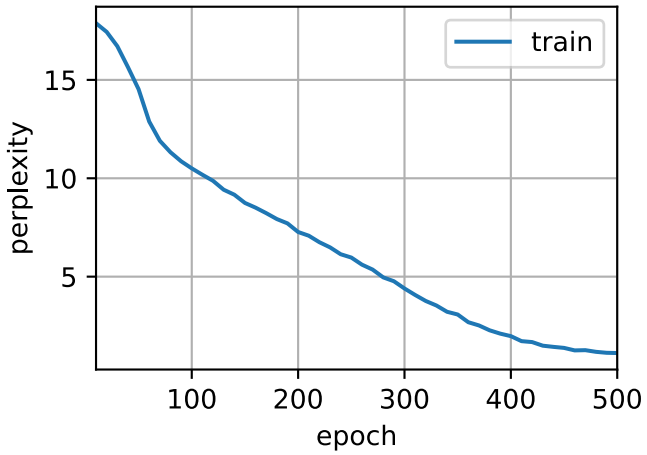
```
        C_tilda = torch.tanh((X @ W_xc) + (H @ W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * torch.tanh(C)
        Y = (H @ W_hq) + b_q
        outputs.append(Y)
    return torch.cat(outputs, dim=0), (H, C)
```

Đào tạo và Dự đoán

Chúng ta hãy đào tạo một LSTM giống như những gì chúng tôi đã làm trong Section 10.1, bằng cách khởi tạo lớp `RNNModelScratch` như được giới thiệu trong Section 9.5.

```
vocab_size, num_hiddens, device = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, device, get_lstm_params,
                            init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 21930.5 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
traveller a forescereopeer three dimensions and the latter
```

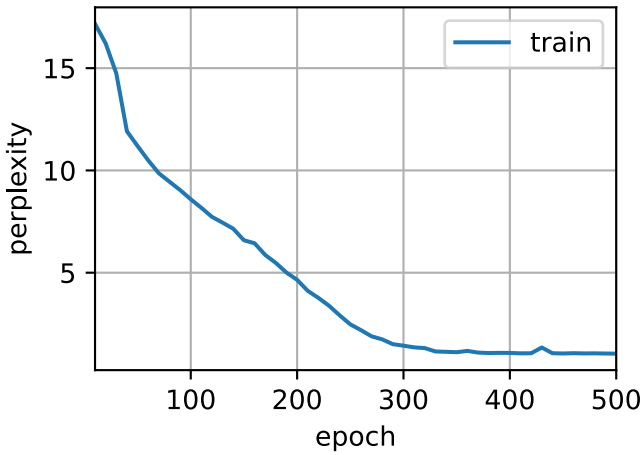


10.2.3 Thiết tập

Sử dụng API cấp cao, chúng ta có thể khởi tạo trực tiếp một mô hình `LSTM`. Điều này đóng gói tất cả các chi tiết cấu hình mà chúng tôi đã thực hiện rõ ràng ở trên. Mã này nhanh hơn đáng kể vì nó sử dụng các toán tử được biên dịch hơn là Python cho nhiều chi tiết mà chúng tôi đã nêu chi tiết trước đây.

```
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 311960.7 tokens/sec on cuda:0
time traveller with a slight accession ofcheerfulness really thi
travelleryou can show black is white by argument said filby
```



LSTMs là mô hình autoregressive biến tiềm ẩn nguyên mẫu với điều khiển trạng thái không tầm thường. Nhiều biến thể của chúng đã được đề xuất trong những năm qua, ví dụ, nhiều lớp, kết nối còn lại, các loại chính quy hóa khác nhau. Tuy nhiên, việc đào tạo LSTMs và các mô hình trình tự khác (chẳng hạn như Grus) khá tốn kém do sự phụ thuộc tầm xa của trình tự. Sau đó chúng ta sẽ gặp phải các mô hình thay thế như máy biến áp có thể được sử dụng trong một số trường hợp.

10.2.4 Tóm tắt

- LSTMs có ba loại cổng: cổng đầu vào, cổng quên và cổng đầu ra kiểm soát luồng thông tin.
- Đầu ra lớp ẩn của LSTM bao gồm trạng thái ẩn và ô nhớ. Chỉ có trạng thái ẩn được truyền vào lớp đầu ra. Tế bào bộ nhớ hoàn toàn bên trong.
- LSTMs có thể làm giảm bớt độ dốc biến mất và bùng nổ.

10.2.5 Bài tập

1. Điều chỉnh các siêu tham số và phân tích ảnh hưởng của chúng đối với thời gian chạy, bối rối và trình tự đầu ra.
2. Làm thế nào bạn sẽ cần phải thay đổi mô hình để tạo ra các từ thích hợp như trái ngược với chuỗi các ký tự?
3. So sánh chi phí tính toán cho Grus, LSTMs và RNN thông thường cho một chiều ẩn nhất định. Đặc biệt chú ý đến chi phí đào tạo và suy luận.
4. Vì ô bộ nhớ ứng cử viên đảm bảo rằng phạm vi giá trị nằm trong khoảng -1 và 1 bằng cách sử dụng hàm tanh, tại sao trạng thái ẩn cần sử dụng lại chức năng tanh để đảm bảo rằng phạm vi giá trị đầu ra nằm trong khoảng từ -1 và 1?
5. Triển khai một mô hình LSTM cho dự đoán chuỗi thời gian hơn là dự đoán chuỗi ký tự.

Discussions[110]

10.3 Mạng thần kinh tái phát sâu

Cho đến nay, chúng ta chỉ thảo luận về RNNs với một layer ẩn một chiều duy nhất. Trong đó hình thức chức năng cụ thể của cách các biến và quan sát tiềm ẩn tương tác là khá tùy ý. Đây không phải là một vấn đề lớn miễn là chúng ta có đủ linh hoạt để mô hình hóa các loại tương tác khác nhau. Tuy nhiên, với một lớp duy nhất, điều này có thể khá khó khăn. Trong trường hợp của các mô hình tuyến tính, chúng tôi đã khắc phục vấn đề này bằng cách thêm nhiều lớp hơn. Trong RNNs, điều này phức tạp hơn một chút, vì trước tiên chúng ta cần quyết định cách thức và ở đâu để thêm tính phi tuyến thêm.

Trên thực tế, chúng ta có thể xếp chồng nhiều lớp RNNlên nhau. Điều này dẫn đến một cơ chế linh hoạt, do sự kết hợp của một số lớp đơn giản. Đặc biệt, dữ liệu có thể có liên quan ở các cấp độ khác nhau của ngăn xếp. Ví dụ: chúng tôi có thể muốn giữ dữ liệu cấp cao về các điều kiện thị trường tài chính (thị trường gấu hoặc thị trường tăng giá) có sẵn, trong khi ở mức thấp hơn, chúng tôi chỉ ghi lại động thái thời gian ngắn hạn.

Ngoài tất cả các cuộc thảo luận trừu tượng ở trên, có lẽ dễ dàng nhất để hiểu được gia đình của các mô hình mà chúng tôi quan tâm bằng cách xem xét Fig. 10.3.1. Nó mô tả một RNN sâu với L lớp ẩn. Mỗi trạng thái ẩn liên tục được truyền cho cả bước thời gian tiếp theo của lớp hiện tại và bước thời gian hiện tại của lớp tiếp theo.

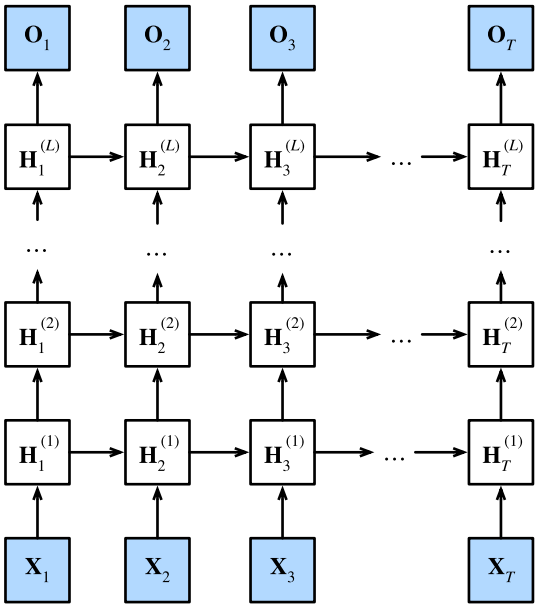


Fig. 10.3.1: Architecture of a deep RNN.

[110] https://discuss.d2l.ai/t/1057

10.3.1 Phụ thuộc chức năng

Chúng ta có thể chính thức hóa các phụ thuộc chức năng trong kiến trúc sâu của L các lớp ẩn được mô tả trong Fig. 10.3.1. Cuộc thảo luận sau đây của chúng tôi tập trung chủ yếu vào mô hình vanilla RNN, nhưng nó cũng áp dụng cho các mô hình trình tự khác.

Giả sử rằng chúng ta có một đầu vào minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (số ví dụ: n, số lượng đầu vào trong mỗi ví dụ: d) tại bước thời gian t. Đồng thời bước, hãy để trạng thái ẩn của lớp ẩn l^{th} ($l = 1, \ldots, L$) là $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (số đơn vị ẩn: h) và biến lớp đầu ra là $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (số lượng đầu ra: q). Đặt $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, trạng thái ẩn của lớp ẩn l^{th} sử dụng hàm kích hoạt ϕ_l được thể hiện như sau:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}), \tag{10.3.1}$$

trong đó trọng lượng $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ và $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, cùng với sự thiên vị $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, là các thông số mô hình của lớp ẩn l^{th}.

Cuối cùng, việc tính toán lớp đầu ra chỉ dựa trên trạng thái ẩn của lớp ẩn cuối cùng L^{th}:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q, \tag{10.3.2}$$

trong đó trọng lượng $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ và thiên vị $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ là các thông số mô hình của lớp đầu ra.

Cũng giống như với MLP, số lượng lớp ẩn L và số đơn vị ẩn h là các siêu tham số. Nói cách khác, chúng có thể được điều chỉnh hoặc chỉ định bởi chúng tôi. Ngoài ra, chúng ta có thể dễ dàng có được một RNN có cổng sâu bằng cách thay thế tính toán trạng thái ẩn trong (10.3.1) với nó từ GRU hoặc LSTM.

10.3.2 Thực hiện ngắn gọn

May mắn thay, nhiều chi tiết hậu cần cần thiết để triển khai nhiều lớp của một RNN có sẵn trong các API cấp cao. Để giữ cho mọi thứ đơn giản, chúng tôi chỉ minh họa việc thực hiện bằng cách sử dụng các chức năng tích hợp như vậy. Hãy để chúng tôi lấy một mô hình LSTM làm ví dụ. Mã này rất giống với mã chúng tôi đã sử dụng trước đây trong Section 10.2. Trên thực tế, sự khác biệt duy nhất là chúng ta chỉ định số lớp một cách rõ ràng hơn là chọn mặc định của một lớp duy nhất. Như thường lệ, chúng ta bắt đầu bằng cách tải tập dữ liệu.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Các quyết định kiến trúc như chọn siêu tham số rất giống với các quyết định của Section 10.2. Chúng tôi chọn cùng một số đầu vào và đầu ra như chúng tôi có mã thông báo riêng biệt, tức là `vocab_size`. Số lượng đơn vị ẩn vẫn là 256. Sự khác biệt duy nhất là chúng ta bây giờ chọn một số không tầm thường của các lớp ẩn bằng cách xác định giá trị của `num_layers`.

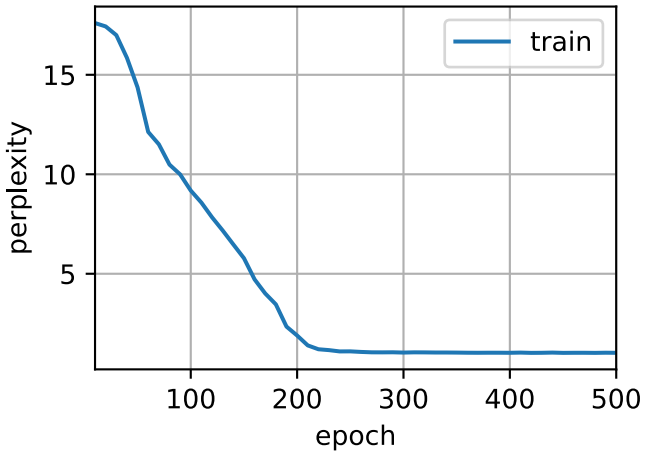
```
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
device = d2l.try_gpu()
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
```

10.3.3 Đào tạo và Dự đoán

Kể từ bây giờ chúng tôi khởi tạo hai lớp với mô hình LSTM, kiến trúc khá phức tạp hơn này làm chậm quá trình đào tạo đáng kể.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.0, 191605.7 tokens/sec on cuda:0
time travelleryou can show black is white by argument said filby
travelleryou can show black is white by argument said filby
```



10.3.4 Tóm tắt

- Trong RNNsâu, thông tin trạng thái ẩn được chuyển sang bước thời gian tiếp theo của lớp hiện tại và bước thời gian hiện tại của lớp tiếp theo.
- Có tồn tại nhiều hương vị khác nhau của RNNsâu, chẳng hạn như LSTMs, Grus, hoặc RNN vani. Thuận tiện, các mô hình này đều có sẵn như là một phần của API cấp cao của các khuôn khổ deep learning.
- Khởi tạo các mô hình đòi hỏi sự chăm sóc. Nhìn chung, RNN sâu đòi hỏi lượng công việc đáng kể (chẳng hạn như tốc độ học tập và cắt) để đảm bảo sự hội tụ thích hợp.

10.3.5 Bài tập

1. Cố gắng thực hiện RNN hai lớp từ đầu bằng cách sử dụng triển khai lớp duy nhất mà chúng tôi đã thảo luận trong Section 9.5.
2. Thay thế LSTM bằng GRU và so sánh độ chính xác và tốc độ đào tạo.
3. Tăng dữ liệu đào tạo để bao gồm nhiều cuốn sách. Làm thế nào thấp bạn có thể đi trên quy mô bối rối?
4. Bạn có muốn kết hợp các nguồn của các tác giả khác nhau khi mô hình hóa văn bản? Tại sao đây là một ý tưởng tốt? Điều gì có thể đi sai?

Discussions[111]

[111] https://discuss.d2l.ai/t/1058

10.4 Mạng nơ-ron định kỳ hai chiều

Trong học trình tự, cho đến nay chúng tôi giả định rằng mục tiêu của chúng tôi là mô hình hóa đầu ra tiếp theo cho những gì chúng ta đã thấy cho đến nay, ví dụ, trong bối cảnh của một chuỗi thời gian hoặc trong bối cảnh của một mô hình ngôn ngữ. Mặc dù đây là một kịch bản điển hình, nhưng nó không phải là kịch bản duy nhất chúng ta có thể gặp phải. Để minh họa vấn đề, hãy xem xét ba nhiệm vụ sau đây để điền vào chỗ trống trong một chuỗi văn bản:

- Tôi là ____.
- Tôi ____ đói.
- Tôi ____ đói, và tôi có thể ăn nửa con lợn.

Tùy thuộc vào lượng thông tin có sẵn, chúng tôi có thể điền vào các khoảng trống với các từ rất khác nhau như "hạnh phúc", "không" và "rất". Rõ ràng phần cuối của cụm từ (nếu có) truyền tải thông tin quan trọng về việc chọn từ nào. Một mô hình trình tự không có khả năng tận dụng điều này sẽ thực hiện kém trên các nhiệm vụ liên quan. Ví dụ, để làm tốt trong việc nhận dạng thực thể được đặt tên (ví dụ: để nhận ra liệu "Green" có đề cập đến "Mr. Green" hay màu) bối cảnh tầm xa cũng quan trọng không kém. Để có được một số nguồn cảm hứng để giải quyết vấn đề, chúng ta hãy đi đường bộ đến các mô hình đồ họa xác suất.

10.4.1 Lập trình động trong các mô hình Markov ẩn

Tiểu mục này phục vụ để minh họa bài toán lập trình động. Các chi tiết kỹ thuật cụ thể không quan trọng để hiểu các mô hình học sâu nhưng chúng giúp thúc đẩy lý do tại sao người ta có thể sử dụng deep learning và tại sao người ta có thể chọn kiến trúc cụ thể.

Nếu chúng ta muốn giải quyết vấn đề bằng cách sử dụng mô hình đồ họa xác suất, chúng ta có thể ví dụ thiết kế một mô hình biến tiềm ẩn như sau. Bất cứ lúc nào bước t, chúng tôi giả định rằng có tồn tại một số biến tiềm ẩn h_t chi phối phát thải quan sát của chúng tôi x_t thông qua $P(x_t \mid h_t)$. Hơn nữa, bất kỳ quá trình chuyển đổi $h_t \to h_{t+1}$ được đưa ra bởi một số xác suất chuyển đổi trạng thái $P(h_{t+1} \mid h_t)$. Mô hình đồ họa xác suất này sau đó là mô hình Markov * ẩn* như trong Fig. 10.4.1.

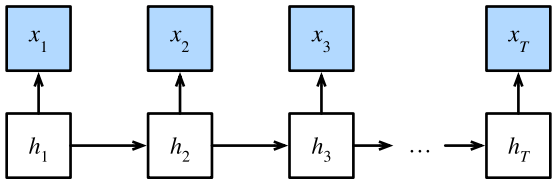


Fig. 10.4.1: A hidden Markov model.

Do đó, đối với một chuỗi T quan sát chúng ta có sự phân bố xác suất chung sau đây trên các trạng thái quan sát và ẩn:

$$P(x_1, \ldots, x_T, h_1, \ldots, h_T) = \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t), \text{ where } P(h_1 \mid h_0) = P(h_1). \tag{10.4.1}$$

Bây giờ giả định rằng chúng ta quan sát tất cả x_i ngoại trừ một số x_j và đó là mục tiêu của chúng tôi để tính toán $P(x_j \mid x_{-j})$, trong đó $x_{-j} = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_T)$. Vì không có biến tiềm ẩn trong $P(x_j \mid x_{-j})$, chúng tôi xem xét tổng hợp tất cả các kết hợp có thể có của các lựa chọn cho $h_1, \ldots, h_T$. Trong trường hợp bất kỳ h_i nào cũng có thể đảm nhận k các giá trị riêng biệt (một số trạng thái hữu hạn), điều này có nghĩa là

chúng ta cần tổng hợp hơn k^T điều khoản — thường là nhiệm vụ không thể! May mắn thay, có một giải pháp thanh lịch cho việc này: * lập trình động *.

Để xem nó hoạt động như thế nào, hãy xem xét tổng hợp các biến tiềm ẩn $h_1, \ldots, h_T$ lần lượt. Theo (10.4.1), sản lượng này:

$$\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_2, \ldots, h_T} \underbrace{\left[\sum_{h_1} P(h_1) P(x_1 \mid h_1) P(h_2 \mid h_1)\right]}_{\pi_2(h_2) \stackrel{\text{def}}{=}} P(x_2 \mid h_2) \prod_{t=3}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \sum_{h_3, \ldots, h_T} \underbrace{\left[\sum_{h_2} \pi_2(h_2) P(x_2 \mid h_2) P(h_3 \mid h_2)\right]}_{\pi_3(h_3) \stackrel{\text{def}}{=}} P(x_3 \mid h_3) \prod_{t=4}^{T} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \\
&= \ldots \\
&= \sum_{h_T} \pi_T(h_T) P(x_T \mid h_T).
\end{aligned} \tag{10.4.2}$$

Nói chung chúng ta có đệ quy* forward* như

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t) P(x_t \mid h_t) P(h_{t+1} \mid h_t). \tag{10.4.3}$$

Đệ quy được khởi tạo là $\pi_1(h_1) = P(h_1)$. Trong thuật ngữ trừu tượng, điều này có thể được viết là $\pi_{t+1} = f(\pi_t, x_t)$, trong đó f là một số hàm có thể học được. Điều này trông rất giống phương trình cập nhật trong các mô hình biến tiềm ẩn mà chúng ta đã thảo luận cho đến nay trong bối cảnh RNN!

Hoàn toàn tương tự như đệ quy về phía trước, chúng ta cũng có thể tổng hợp cùng một tập hợp các biến tiềm

ẩn với đệ quy ngược. Điều này mang lại:

$$\begin{aligned}
&P(x_1, \ldots, x_T) \\
&= \sum_{h_1, \ldots, h_T} P(x_1, \ldots, x_T, h_1, \ldots, h_T) \\
&= \sum_{h_1, \ldots, h_T} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot P(h_T \mid h_{T-1}) P(x_T \mid h_T) \\
&= \sum_{h_1, \ldots, h_{T-1}} \prod_{t=1}^{T-1} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_T} P(h_T \mid h_{T-1}) P(x_T \mid h_T)\right]}_{\rho_{T-1}(h_{T-1}) \stackrel{\text{def}}{=}} \\
&= \sum_{h_1, \ldots, h_{T-2}} \prod_{t=1}^{T-2} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \cdot \underbrace{\left[\sum_{h_{T-1}} P(h_{T-1} \mid h_{T-2}) P(x_{T-1} \mid h_{T-1}) \rho_{T-1}(h_{T-1})\right]}_{\rho_{T-2}(h_{T-2}) \stackrel{\text{def}}{=}} \\
&= \ldots \\
&= \sum_{h_1} P(h_1) P(x_1 \mid h_1) \rho_1(h_1).
\end{aligned} \tag{10.4.4}$$

Do đó, chúng ta có thể viết các * backward recursion* như

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} P(h_t \mid h_{t-1}) P(x_t \mid h_t) \rho_t(h_t), \tag{10.4.5}$$

với khởi tạo $\rho_T(h_T) = 1$. Cả đệ quy tiến và lùi đều cho phép chúng ta tổng hợp hơn T các biến tiềm ẩn trong thời gian $\mathcal{O}(kT)$ (tuyến tính) trên tất cả các giá trị của $(h_1, \ldots, h_T)$ chứ không phải trong thời gian hàm mũ. Đây là một trong những lợi ích lớn của suy luận xác suất với các mô hình đồ họa. Nó cũng là một trường hợp rất đặc biệt của một thông điệp chung vượt qua thuật toán (Aji & McEliece, 2000). Kết hợp cả đệ quy tiến và lùi, chúng ta có thể tính toán

$$P(x_j \mid x_{-j}) \propto \sum_{h_j} \pi_j(h_j) \rho_j(h_j) P(x_j \mid h_j). \tag{10.4.6}$$

Lưu ý rằng trong thuật ngữ trừu tượng đệ quy ngược có thể được viết là $\rho_{t-1} = g(\rho_t, x_t)$, trong đó g là một hàm có thể học được. Một lần nữa, điều này trông rất giống như một phương trình cập nhật, chỉ chạy ngược không giống như những gì chúng ta đã thấy cho đến nay trong RNNs. Thật vậy, các mô hình Markov ẩn được hưởng lợi từ việc biết dữ liệu trong tương lai khi có sẵn. Các nhà khoa học xử lý tín hiệu phân biệt giữa hai trường hợp biết và không biết các quan sát trong tương lai như nội suy v.s. ngoại suy. Xem chương giới thiệu của cuốn sách về thuật toán Monte Carlo tuần tự để biết thêm chi tiết (Doucet et al., 2001).

10.4.2 Mô hình hai chiều

Nếu chúng ta muốn có một cơ chế trong RNN cung cấp khả năng nhìn trước tương đương như trong các mô hình Markov ẩn, chúng ta cần sửa đổi thiết kế RNN mà chúng ta đã thấy cho đến nay. May mắn thay, điều này là dễ dàng về mặt khái niệm. Thay vì chạy RNN chỉ ở chế độ chuyển tiếp bắt đầu từ token đầu tiên, chúng ta bắt đầu một mã thông báo khác từ token cuối cùng chạy từ sau ra trước. *Hai chiều RNNs* thêm một lớp ẩn truyền thông tin theo hướng ngược để xử lý thông tin đó linh hoạt hơn. Fig. 10.4.2 minh họa kiến trúc của RNN hai chiều với một lớp ẩn duy nhất.

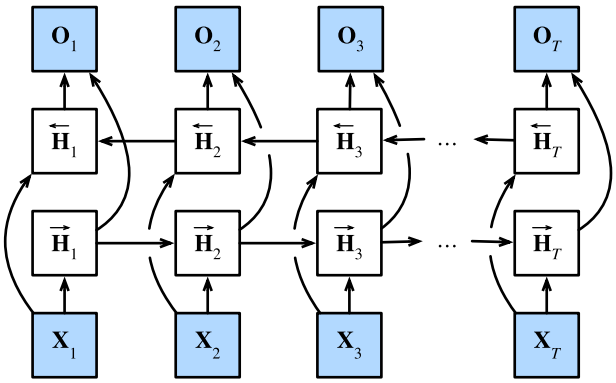


Fig. 10.4.2: Architecture of a bidirectional RNN.

Trên thực tế, điều này không quá khác nhau với các đệ quy về phía trước và lạc hậu trong việc lập trình động của các mô hình Markov ẩn. Sự khác biệt chính là trong trường hợp trước các phương trình này có ý nghĩa thống kê cụ thể. Bây giờ chúng không có những giải thích dễ tiếp cận như vậy và chúng ta chỉ có thể coi chúng như là các chức năng chung và có thể học được. Quá trình chuyển đổi này mô tả nhiều nguyên tắc hướng dẫn thiết kế các mạng sâu hiện đại: đầu tiên, sử dụng loại phụ thuộc chức năng của các mô hình thống kê cổ điển, và sau đó tham số hóa chúng trong một hình thức chung chung.

Definition

Các RNN hai chiều được giới thiệu bởi (Schuster & Paliwal, 1997). Đối với một cuộc thảo luận chi tiết về các kiến trúc khác nhau xem thêm bài báo (Graves & Schmidhuber, 2005). Chúng ta hãy nhìn vào các chi tiết cụ thể của một mạng như vậy.

Đối với bất kỳ bước thời gian t, đưa ra một đầu vào minibatch $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (số ví dụ: n, số đầu vào trong mỗi ví dụ: d) và để cho chức năng kích hoạt lớp ẩn là ϕ. Trong kiến trúc hai chiều, chúng ta giả định rằng các trạng thái ẩn về phía trước và lạc hậu cho bước thời gian này là $\overrightarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ và $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$, tương ứng, trong đó h là số đơn vị ẩn. Các bản cập nhật trạng thái ẩn về phía trước và lạc hậu như sau:

$$\begin{aligned}\overrightarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}),\\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}),\end{aligned} \tag{10.4.7}$$

trong đó trọng lượng $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$, and $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$, và thiên vị $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$ and $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ là tất cả các thông số mô hình.

Tiếp theo, chúng tôi nối các trạng thái ẩn về phía trước và lạc hậu $\overrightarrow{\mathbf{H}}_t$ và $\overleftarrow{\mathbf{H}}_t$ để có được trạng thái ẩn $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$ được đưa vào lớp đầu ra. Trong các RNN hai chiều sâu với nhiều lớp ẩn, thông tin như vậy được truyền dưới dạng * đầu vào* sang lớp hai chiều tiếp theo. Cuối cùng, lớp đầu ra tính toán đầu ra $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (số lượng đầu ra: q):

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \tag{10.4.8}$$

Ở đây, ma trận trọng lượng $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$ và thiên vị $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ là các tham số mô hình của lớp đầu ra. Trên thực tế, hai hướng có thể có số lượng khác nhau của các đơn vị ẩn.

Chi phí tính toán và các ứng dụng

Một trong những tính năng chính của RNN hai chiều là thông tin từ cả hai đầu của chuỗi được sử dụng để ước tính đầu ra. Đó là, chúng tôi sử dụng thông tin từ cả quan sát trong tương lai và trong quá khứ để dự đoán thông tin hiện tại. Trong trường hợp dự đoán token tiếp theo, đây không hoàn toàn là những gì chúng ta muốn. Rốt cuộc, chúng ta không có sự xa xỉ khi biết mã thông báo tiếp theo khi dự đoán mã tiếp theo. Do đó, nếu chúng ta sử dụng RNN hai chiều một cách ngây thơ, chúng ta sẽ không có được độ chính xác rất tốt: trong quá trình đào tạo, chúng ta có dữ liệu trong quá khứ và tương lai để ước tính hiện tại. Trong thời gian thử nghiệm, chúng tôi chỉ có dữ liệu trong quá khứ và do đó độ chính xác kém. Chúng tôi sẽ minh họa điều này trong một thí nghiệm dưới đây.

Để thêm xúc phạm đến chấn thương, RNN hai chiều cũng cực kỳ chậm. Những lý do chính cho điều này là sự lan truyền về phía trước đòi hỏi cả đệ quy về phía trước và ngược trong các lớp hai chiều và sự lan truyền ngược phụ thuộc vào kết quả của sự lan truyền về phía trước. Do đó, gradient sẽ có một chuỗi phụ thuộc rất dài.

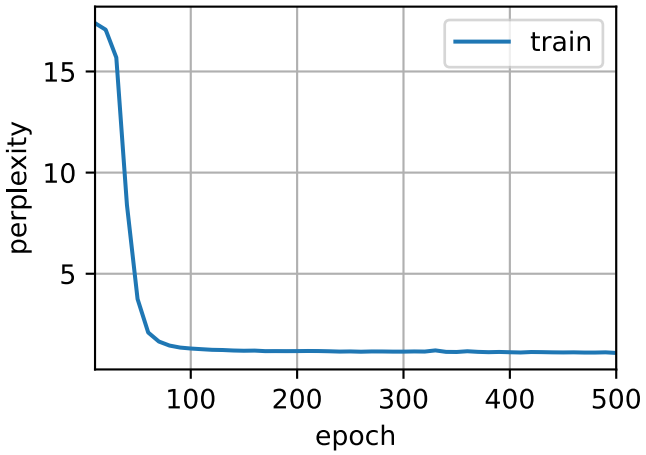
Trong thực tế các lớp hai chiều được sử dụng rất ít và chỉ cho một tập hợp các ứng dụng hẹp, chẳng hạn như điền vào các từ bị thiếu, mã thông báo chú thích (ví dụ, để nhận dạng thực thể được đặt tên) và mã hóa chuỗi bán buôn như một bước trong một đường ống xử lý trình tự (ví dụ, cho dịch máy). Trong Section 15.7 và Section 16.2, chúng tôi sẽ giới thiệu cách sử dụng RNN hai chiều để mã hóa chuỗi văn bản.

10.4.3 Đào tạo một RNN hai chiều cho một ứng dụng sai

Nếu chúng ta bỏ qua tất cả lời khuyên liên quan đến thực tế là RNN hai chiều sử dụng dữ liệu trong quá khứ và tương lai và chỉ cần áp dụng nó cho các mô hình ngôn ngữ, chúng tôi sẽ nhận được ước tính với sự bối rối chấp nhận được. Tuy nhiên, khả năng của mô hình dự đoán token trong tương lai bị tổn hại nghiêm trọng như thí nghiệm dưới đây minh họa. Mặc dù bối rối hợp lý, nó chỉ tạo ra sự lúng túng ngay cả sau nhiều lần lặp lại. Chúng tôi bao gồm mã dưới đây như một ví dụ cảnh báo chống lại việc sử dụng chúng trong bối cảnh sai.

```
import torch
from torch import nn
from d2l import torch as d2l

# Load data
batch_size, num_steps, device = 32, 35, d2l.try_gpu()
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the bidirectional LSTM model by setting `bidirectional=True`
vocab_size, num_hiddens, num_layers = len(vocab), 256, 2
num_inputs = vocab_size
lstm_layer = nn.LSTM(num_inputs, num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
model = model.to(device)
# Train the model
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, device)
```

```
perplexity 1.1, 122518.9 tokens/sec on cuda:0
time travellererererererererererererererererererererererererererererer
travellererererererererererererererererererererererererererererer
```



Đầu ra rõ ràng là không đạt yêu cầu vì những lý do được mô tả ở trên. Để thảo luận về việc sử dụng RNN hai chiều hiệu quả hơn, vui lòng xem ứng dụng phân tích tâm lý trong Section 16.2.

10.4.4 Tóm tắt

- Trong RNN hai chiều, trạng thái ẩn cho mỗi bước thời gian được xác định đồng thời bởi dữ liệu trước và sau bước thời gian hiện tại.
- RNN hai chiều mang một sự tương đồng nổi bật với thuật toán về phía trước về phía sau trong các mô hình đồ họa xác suất.
- Các RNN hai chiều chủ yếu hữu ích cho mã hóa trình tự và ước lượng các quan sát được đưa ra bối cảnh hai chiều.
- RNNhai chiều rất tốn kém để đào tạo do chuỗi gradient dài.

10.4.5 Bài tập

1. Nếu các hướng khác nhau sử dụng một số đơn vị ẩn khác nhau, hình dạng của $\mathbf{H}_t$ sẽ thay đổi như thế nào?
2. Thiết kế một RNN hai chiều với nhiều lớp ẩn.
3. Polysemy phổ biến trong các ngôn ngữ tự nhiên. Ví dụ, từ "ngân hàng" có ý nghĩa khác nhau trong bối cảnh "tôi đã đến ngân hàng để gửi tiền mặt" và "tôi đã đến ngân hàng để ngồi xuống". Làm thế nào chúng ta có thể thiết kế một mô hình mạng thần kinh như vậy mà đưa ra một chuỗi ngữ cảnh và một từ, một biểu diễn vector của từ trong ngữ cảnh sẽ được trả về? Loại kiến trúc thần kinh nào được ưa thích để xử lý polysemy?

Discussions[112]

[112] https://discuss.d2l.ai/t/1059

10.5 Dịch máy và bộ dữ liệu

Chúng tôi đã sử dụng RNNs để thiết kế các mô hình ngôn ngữ, đó là chìa khóa để xử lý ngôn ngữ tự nhiên. Một điểm chuẩn hàng đầu khác là * dịch máy*, một miền vấn đề trung tâm cho các mô hình * chuyển đổi chuỗi lự* chuyển đổi chuỗi đầu vào thành chuỗi đầu ra. Đóng một vai trò quan trọng trong các ứng dụng AI hiện đại khác nhau, các mô hình chuyển tải chuỗi sẽ tạo thành trọng tâm của phần còn lại của chương này và Chapter 11. Để kết thúc này, phần này giới thiệu vấn đề dịch máy và tập dữ liệu của nó sẽ được sử dụng sau này.

Dịch máy đề cập đến dịch tự động của một chuỗi từ ngôn ngữ này sang ngôn ngữ khác. Trên thực tế, lĩnh vực này có thể có niên đại từ những năm 1940 ngay sau khi máy tính kỹ thuật số được phát minh, đặc biệt là bằng cách xem xét việc sử dụng máy tính để nứt mã ngôn ngữ trong Thế chiến II. Trong nhiều thập kỷ, các phương pháp thống kê đã chiếm ưu thế trong lĩnh vực này (Brown et al., 1988, 1990) trước khi sự gia tăng của học tập từ đầu đến cuối bằng cách sử dụng mạng thần kinh. Cái sau thường được gọi là *dịch máy nơ-ron* to distinguish phân biệt itself chinh no from *dịch máy thống thống* liên quan đến việc phân tích thống kê trong các thành phần như mô hình dịch thuật và mô hình ngôn ngữ.

Nhấn mạnh việc học từ đầu đến cuối, cuốn sách này sẽ tập trung vào các phương pháp dịch máy thần kinh. Khác với vấn đề mô hình ngôn ngữ của chúng tôi trong Section 9.3 có corpus là trong một ngôn ngữ duy nhất, các tập dữ liệu dịch máy bao gồm các cặp chuỗi văn bản trong ngôn ngữ nguồn và ngôn ngữ đích, tương ứng. Do đó, thay vì sử dụng lại thói quen tiền xử lý để mô hình hóa ngôn ngữ, chúng ta cần một cách khác để xử lý trước các bộ dữ liệu dịch máy. Trong phần sau, chúng tôi chỉ ra cách tải dữ liệu được xử lý trước vào minibatches để đào tạo.

```
import os
import torch
from d2l import torch as d2l
```

10.5.1 Tải xuống và xử lý sơ bộ dữ liệu

Để bắt đầu, chúng tôi tải xuống một tập dữ liệu tiếng Anh-Pháp bao gồm bilingual sentence pairs from the Tatoeba Project[113]. Mỗi dòng trong tập dữ liệu là một cặp phân cách tab của một chuỗi văn bản tiếng Anh và chuỗi văn bản tiếng Pháp được dịch. Lưu ý rằng mỗi chuỗi văn bản có thể chỉ là một câu hoặc một đoạn văn của nhiều câu. Trong vấn đề dịch máy này, nơi tiếng Anh được dịch sang tiếng Pháp, tiếng Anh là ngôn ngữ nguồn * và tiếng Pháp là ngôn ngữ mục tiêu *.

```
#@save
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                           '94646ad1522d915e7b0f9296181140edcf86a4f5')

#@save
def read_data_nmt():
    """Load the English-French dataset."""
    data_dir = d2l.download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
```

[113] http://www.manythings.org/anki/

```
Go. Va !
Hi. Salut !
Run!        Cours !
Run!        Courez !
Who?        Qui ?
Wow!        Ça alors !
```

Sau khi tải xuống tập dữ liệu, chúng tôi tiến hành một số bước xử lý cho dữ liệu văn bản thô. Ví dụ, chúng ta thay thế không gian không phá vỡ bằng dấu cách, chuyển đổi chữ hoa thành chữ thường và chèn khoảng cách giữa các từ và dấu chấm câu.

```
#@save
def preprocess_nmt(text):
    """Preprocess the English-French dataset."""
    def no_space(char, prev_char):
        return char in set(',.!?') and prev_char != ' '

    # Replace non-breaking space with space, and convert uppercase letters to
    # lowercase ones
    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    # Insert space between words and punctuation marks
    out = [' ' + char if i > 0 and no_space(char, text[i - 1]) else char
           for i, char in enumerate(text)]
    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
```

```
go .        va !
hi .        salut !
run !       cours !
run !       courez !
who ?       qui ?
wow !       ça alors !
```

10.5.2 Tokenization

Khác với mã hóa cấp ký tự trong Section 9.3, đối với dịch máy, chúng tôi thích mã hóa cấp từ ở đây (các mô hình hiện đại có thể sử dụng các kỹ thuật mã hóa tiên tiến hơn). Hàm `tokenize_nmt` sau mã hóa các cặp chuỗi văn bản `num_examples` đầu tiên, trong đó mỗi token là một từ hoặc dấu chấm câu. Hàm này trả về hai danh sách các danh sách token: `source` và `target`. Cụ thể, `source[i]` là danh sách các mã thông báo từ chuỗi văn bản i^{th} trong ngôn ngữ nguồn (tiếng Anh ở đây) và `target[i]` là trong ngôn ngữ đích (tiếng Pháp ở đây).

```
#@save
def tokenize_nmt(text, num_examples=None):
    """Tokenize the English-French dataset."""
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
```

(continues on next page)

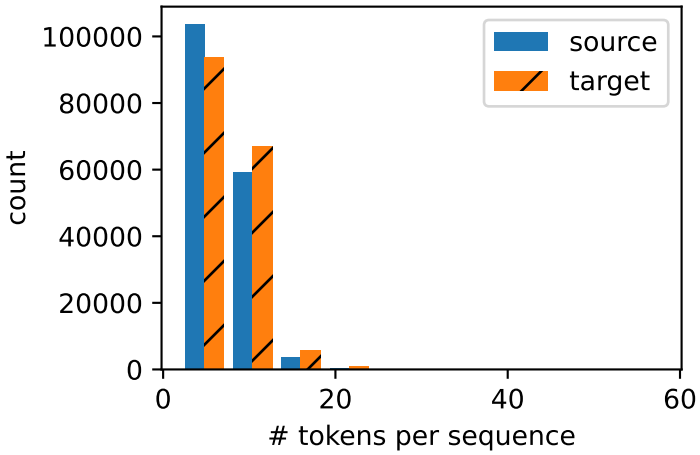
(continued from previous page)

```
        parts = line.split('\t')
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[:6], target[:6]
```

```
([['go', '.'],
  ['hi', '.'],
  ['run', '!'],
  ['run', '!'],
  ['who', '?'],
  ['wow', '!']],
 [['va', '!'],
  ['salut', '!'],
  ['cours', '!'],
  ['courez', '!'],
  ['qui', '?'],
  ['ça', 'alors', '!']])
```

Hãy để chúng tôi vẽ biểu đồ của số lượng mã thông báo trên mỗi chuỗi văn bản. Trong tập dữ liệu tiếng Anh-Pháp đơn giản này, hầu hết các chuỗi văn bản có ít hơn 20 mã thông báo.

```
#@save
def show_list_len_pair_hist(legend, xlabel, ylabel, xlist, ylist):
    """Plot the histogram for list length pairs."""
    d2l.set_figsize()
    _, _, patches = d2l.plt.hist(
        [[len(l) for l in xlist], [len(l) for l in ylist]])
    d2l.plt.xlabel(xlabel)
    d2l.plt.ylabel(ylabel)
    for patch in patches[1].patches:
        patch.set_hatch('/')
    d2l.plt.legend(legend)

show_list_len_pair_hist(['source', 'target'], '# tokens per sequence',
                        'count', source, target);
```



10.5.3 Từ vừ

Vì tập dữ liệu dịch máy bao gồm các cặp ngôn ngữ, chúng ta có thể xây dựng hai từ vựng cho cả ngôn ngữ nguồn và ngôn ngữ đích riêng biệt. Với tokenization cấp từ, kích thước từ vựng sẽ lớn hơn đáng kể so với việc sử dụng tokenization cấp ký tự. Để giảm bớt điều này, ở đây chúng tôi xử lý các token không thường xuyên xuất hiện ít hơn 2 lần so với cùng một mã thông báo chưa biết (" "). Bên cạnh đó, chúng tôi chỉ định các token đặc biệt bổ sung như cho các chuỗi padding (" ") với cùng độ dài trong minibatches và để đánh dấu đầu (" ") hoặc kết thúc (" ") của chuỗi. Các token đặc biệt như vậy thường được sử dụng trong các nhiệm vụ xử lý ngôn ngữ tự nhiên.

```
src_vocab = d2l.Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)
```

```
10012
```

10.5.4 Đọc tập dữ liệu

Nhớ lại rằng trong mô hình ngôn ngữ mỗi chuỗi ví dụ, hoặc là một đoạn của một câu hoặc một khoảng trên nhiều câu, có chiều dài cố định. Điều này được chỉ định bởi đối số `num_steps` (số bước thời gian hoặc mã thông báo) trong Section 9.3. Trong dịch máy, mỗi ví dụ là một cặp chuỗi văn bản nguồn và đích, trong đó mỗi chuỗi văn bản có thể có độ dài khác nhau.

Đối với hiệu quả tính toán, chúng ta vẫn có thể xử lý một minibatch các chuỗi văn bản cùng một lúc bằng cách *cutcation* và *padding*. Giả sử rằng mọi chuỗi trong cùng một minibatch nên có cùng chiều dài `num_steps`. Nếu chuỗi văn bản có ít hơn `num_steps` token, chúng ta sẽ tiếp tục nối mã thông báo "" đặc biệt vào cuối cho đến khi độ dài của nó đạt đến `num_steps`. Nếu không, chúng tôi sẽ cắt ngắn chuỗi văn bản bằng cách chỉ lấy mã thông báo `num_steps` đầu tiên của nó và loại bỏ phần còn lại. Bằng cách này, mỗi chuỗi văn bản sẽ có cùng độ dài để được tải trong các minibatches có cùng hình dạng.

Chức năng `truncate_pad` sau đây cắt ngắn hoặc miếng đệm chuỗi văn bản như mô tả trước đó.

```
#@save
def truncate_pad(line, num_steps, padding_token):
    """Truncate or pad sequences."""
```

(continues on next page)

(continued from previous page)

```
    if len(line) > num_steps:
        return line[:num_steps]  # Truncate
    return line + [padding_token] * (num_steps - len(line))  # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])
```

```
[47, 4, 1, 1, 1, 1, 1, 1, 1, 1]
```

Bây giờ chúng ta định nghĩa một hàm để transform text sequences thành minibatches để đào tạo. Chúng tôi thêm token “” đặc biệt vào cuối mỗi chuỗi để chỉ ra kết thúc của chuỗi. Khi một mô hình dự đoán bằng cách tạo ra một token chuỗi sau token, việc tạo ra token “” có thể gợi ý rằng chuỗi đầu ra đã hoàn tất. Bên cạnh đó, chúng tôi cũng ghi lại độ dài của mỗi chuỗi văn bản không bao gồm các token padding. Thông tin này sẽ cần thiết bởi một số mô hình mà chúng tôi sẽ đề cập sau.

```
#@save
def build_array_nmt(lines, vocab, num_steps):
    """Transform text sequences of machine translation into minibatches."""
    lines = [vocab[l] for l in lines]
    lines = [l + [vocab['<eos>']] for l in lines]
    array = torch.tensor([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)
    return array, valid_len
```

10.5.5 Putting All Things Together

Cuối cùng, chúng ta định nghĩa hàm `load_data_nmt` để trả về bộ lặp dữ liệu, cùng với từ vựng cho cả ngôn ngữ nguồn và ngôn ngữ đích.

```
#@save
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """Return the iterator and the vocabularies of the translation dataset."""
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=2,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return data_iter, src_vocab, tgt_vocab
```

Hãy để chúng tôi đọc minibatch đầu tiên từ dữ liệu tiếng Anh-Pháp.

```
train_iter, src_vocab, tgt_vocab = load_data_nmt(batch_size=2, num_steps=8)
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', X.type(torch.int32))
    print('valid lengths for X:', X_valid_len)
```

(continues on next page)

(continued from previous page)

```
    print('Y:', Y.type(torch.int32))
    print('valid lengths for Y:', Y_valid_len)
    break
```

```
X: tensor([[53,  8,  4,  3,  1,  1,  1,  1],
        [83, 15,  4,  3,  1,  1,  1,  1]], dtype=torch.int32)
valid lengths for X: tensor([4, 4])
Y: tensor([[192,   5,   3,   1,   1,   1,   1,   1],
        [  0,   5,   3,   1,   1,   1,   1,   1]], dtype=torch.int32)
valid lengths for Y: tensor([3, 3])
```

10.5.6 Tóm tắt

- Dịch máy đề cập đến bản dịch tự động của một chuỗi từ ngôn ngữ này sang ngôn ngữ khác.
- Sử dụng tokenization cấp từ, kích thước từ vựng sẽ lớn hơn đáng kể so với việc sử dụng tokenization cấp ký tự. Để giảm bớt điều này, chúng ta có thể coi các token không thường xuyên như cùng một mã thông báo không xác định.
- Chúng ta có thể cắt ngắn và pad chuỗi văn bản để tất cả chúng sẽ có cùng độ dài để được tải trong minibatches.

10.5.7 Bài tập

1. Hãy thử các giá trị khác nhau của đối số `num_examples` trong hàm `load_data_nmt`. Điều này ảnh hưởng đến kích thước từ vựng của ngôn ngữ nguồn và ngôn ngữ đích như thế nào?
2. Văn bản trong một số ngôn ngữ như tiếng Trung và tiếng Nhật không có chỉ số ranh giới từ (ví dụ: khoảng cách). Token ization cấp từ vẫn là một ý tưởng hay cho những trường hợp như vậy? Tại sao hoặc tại sao không?

Discussions[114]

10.6 Kiến trúc mã hóa-Decoder

Như chúng ta đã thảo luận trong Section 10.5, dịch máy là một tên miền vấn đề lớn cho các mô hình chuyển tải chuỗi, có đầu vào và đầu ra đều là chuỗi độ dài biến đổi. Để xử lý loại đầu vào và đầu ra này, chúng ta có thể thiết kế một kiến trúc với hai thành phần chính. Thành phần đầu tiên là bộ mã hóa*: nó lấy một chuỗi có độ dài thay đổi làm đầu vào và biến nó thành trạng thái có hình dạng cố định. Thành phần thứ hai là một *decoder*: nó ánh xạ trạng thái mã hóa của một hình dạng cố định đến một chuỗi có chiều dài biến đổi. Đây được gọi là kiến trúc *encoder-decoder*, được mô tả trong Fig. 10.6.1.

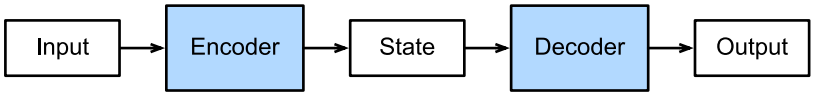


Fig. 10.6.1: The encoder-decoder architecture.

[114] https://discuss.d2l.ai/t/1060

Hãy để chúng tôi lấy bản dịch máy từ tiếng Anh sang tiếng Pháp làm ví dụ. Cho một chuỗi đầu vào bằng tiếng Anh: “They”, “are”, “watching”,” . “, kiến trúc bộ mã hóa giải mã này đầu tiên mã hóa đầu vào có độ dài biến đổi thành một trạng thái, sau đó giải mã trạng thái để tạo ra token chuỗi dịch bằng token làm đầu ra: “Ils”, “regardent”, “.”. Vì kiến trúc bộ mã hóa-giải mã tạo thành nền tảng của các mô hình chuyển đổi chuỗi khác nhau trong các phần tiếp theo, phần này sẽ chuyển đổi kiến trúc này thành một giao diện sẽ được triển khai sau này.

10.6.1 Bộ mã hóa

Trong giao diện bộ mã hóa, chúng tôi chỉ xác định rằng bộ mã hóa lấy các chuỗi có độ dài thay đổi làm đầu vào `X`. Việc triển khai sẽ được cung cấp bởi bất kỳ mô hình nào kế thừa lớp cơ sở `Encoder` này.

```
from torch import nn


#@save
class Encoder(nn.Module):
    """The base encoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        raise NotImplementedError
```

10.6.2 Decoder

Trong giao diện giải mã sau, chúng tôi thêm một chức năng `init_state` bổ sung để chuyển đổi đầu ra bộ mã hóa (`enc_outputs`) vào trạng thái mã hóa. Lưu ý rằng bước này có thể cần thêm các đầu vào như độ dài hợp lệ của đầu vào, được giải thích trong Section 10.5.4. Để tạo ra một mã thông báo chuỗi có độ dài biến đổi theo mã thông báo, mỗi khi bộ giải mã có thể ánh xạ một đầu vào (ví dụ, mã thông báo được tạo ra ở bước thời gian trước) và trạng thái mã hóa thành một mã thông báo đầu ra ở bước thời gian hiện tại.

```
#@save
class Decoder(nn.Module):
    """The base decoder interface for the encoder-decoder architecture."""
    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
        raise NotImplementedError

    def forward(self, X, state):
        raise NotImplementedError
```

10.6.3 Đút bộ mã hóa và bộ giải mã cùng nhau

Cuối cùng, kiến trúc bộ mã hóa-giải mã chứa cả bộ mã hóa và bộ giải mã, với các đối số bổ sung tùy chọn. Trong tuyên truyền chuyển tiếp, đầu ra của bộ mã hóa được sử dụng để tạo ra trạng thái được mã hóa và trạng thái này sẽ được bộ giải mã sử dụng thêm như một trong những đầu vào của nó.

```
#@save
class EncoderDecoder(nn.Module):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        enc_outputs = self.encoder(enc_X, *args)
        dec_state = self.decoder.init_state(enc_outputs, *args)
        return self.decoder(dec_X, dec_state)
```

Thuật ngữ "trạng thái" trong kiến trúc bộ mã hóa-giải mã có lẽ đã truyền cảm hứng cho bạn thực hiện kiến trúc này bằng cách sử dụng các mạng thần kinh với các trạng thái. Trong phần tiếp theo, chúng ta sẽ thấy cách áp dụng RNNs để thiết kế các mô hình chuyển tải chuỗi dựa trên kiến trúc bộ mã hóa-giải mã này.

10.6.4 Tóm tắt

- Kiến trúc bộ mã hóa-giải mã có thể xử lý các đầu vào và đầu ra vừa là chuỗi độ dài biến đổi, do đó phù hợp với các vấn đề chuyển tải trình tự như dịch máy.
- Bộ mã hóa lấy một chuỗi chiều dài biến đổi làm đầu vào và biến nó thành trạng thái có hình dạng cố định.
- Bộ giải mã ánh xạ trạng thái được mã hóa của một hình dạng cố định đến một chuỗi có độ dài thay đổi.

10.6.5 Bài tập

1. Giả sử rằng chúng ta sử dụng mạng nơ-ron để thực hiện kiến trúc bộ mã hóa-giải mã. Bộ mã hóa và bộ giải mã có phải là cùng một loại mạng thần kinh không?
2. Bên cạnh dịch máy, bạn có thể nghĩ đến một ứng dụng khác mà kiến trúc bộ mã hóa-giải mã có thể được áp dụng không?

Discussions[115]

[115] https://discuss.d2l.ai/t/1061

10.7 Trình tự để học trình tự

Như chúng ta đã thấy trong Section 10.5, trong dịch máy cả đầu vào và đầu ra là một chuỗi chiều dài biến đổi. Để giải quyết loại vấn đề này, chúng tôi đã thiết kế một kiến trúc bộ mã hóa giải mã chung trong Section 10.6. Trong phần này, chúng tôi sẽ sử dụng hai RNNđể thiết kế bộ mã hóa và bộ giải mã của kiến trúc này và áp dụng nó vào trình tự * trình tự để học tập cho dịch máy (Sutskever et al., 2014; Cho et al., 2014b).

Theo nguyên tắc thiết kế của kiến trúc bộ mã hóa-giải mã, bộ mã hóa RNN có thể lấy một chuỗi độ dài thay đổi làm đầu vào và biến nó thành trạng thái ẩn hình dạng cố định. Nói cách khác, thông tin của chuỗi đầu vào (nguồn) là *mã hóa* ở trạng thái ẩn của bộ mã hóa RNN. Để tạo ra mã thông báo chuỗi đầu ra theo mã thông báo, một bộ giải mã RNN riêng biệt có thể dự đoán token tiếp theo dựa trên những mã thông báo đã được nhìn thấy (chẳng hạn như trong mô hình ngôn ngữ) hoặc được tạo ra, cùng với thông tin được mã hóa của chuỗi đầu vào. Fig. 10.7.1 minh họa cách sử dụng hai RNNđể trình tự learning học tập in machine máy móc translation dịch thuật.

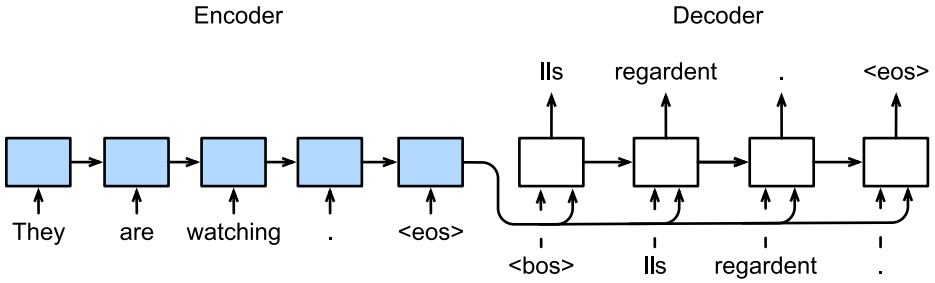


Fig. 10.7.1: Sequence to sequence learning with an RNN encoder and an RNN decoder.

Năm Fig. 10.7.1, mã thông báo “” đặc biệt đánh dấu sự kết thúc của chuỗi. Mô hình có thể ngừng đưa ra dự đoán sau khi mã thông báo này được tạo ra. Tại bước thời gian ban đầu của bộ giải mã RNN, có hai quyết định thiết kế đặc biệt. Đầu tiên, mã thông báo bắt đầu chuỗi “” đặc biệt là một đầu vào. Thứ hai, trạng thái ẩn cuối cùng của bộ mã hóa RNN được sử dụng để bắt đầu trạng thái ẩn của bộ giải mã. Trong các thiết kế như (Sutskever et al., 2014), đây chính xác là cách thông tin chuỗi đầu vào được mã hóa được đưa vào bộ giải mã để tạo ra chuỗi đầu ra (mục tiêu). Trong một số thiết kế khác như (Cho et al., 2014b), trạng thái ẩn cuối cùng của bộ mã hóa cũng được đưa vào bộ giải mã như một phần của các đầu vào tại từng bước như thể hiện trong Fig. 10.7.1. Tương tự như việc đào tạo các mô hình ngôn ngữ trong Section 9.3, chúng ta có thể cho phép các nhãn là chuỗi đầu ra ban đầu, được dịch chuyển bởi một mã thông báo: ““, “Ils”, “regardent”, “.” → “Ils”, “regardent”,” . “,””.

Sau đây, chúng tôi sẽ giải thích thiết kế của Fig. 10.7.1 chi tiết hơn. Chúng tôi sẽ đào tạo mô hình này để dịch máy trên tập dữ liệu tiếng Anh-Pháp như được giới thiệu trong Section 10.5.

```
import collections
import math
import torch
from torch import nn
from d2l import torch as d2l
```

10.7.1 Bộ mã hóa

Về mặt kỹ thuật, bộ mã hóa biến đổi một chuỗi đầu vào có độ dài biến thành một biến đổi ngữ cảnh hình dạng cố định* **c** và mã hóa thông tin trình tự đầu vào trong biến ngữ cảnh này. Như được mô tả trong Fig. 10.7.1, chúng ta có thể sử dụng RNN để thiết kế bộ mã hóa.

Chúng ta hãy xem xét một ví dụ trình tự (kích thước lô: 1). Giả sử chuỗi đầu vào là $x_1, \ldots, x_T$, sao cho x_t là mã thông báo t^{th} trong chuỗi văn bản đầu vào. Tại bước thời gian t, RNN biến đổi vector tính năng đầu vào $\mathbf{x}_t$ cho x_t và trạng thái ẩn $\mathbf{h}_{t-1}$ từ bước thời gian trước vào trạng thái ẩn hiện tại $\mathbf{h}_t$. Ta có thể sử dụng một hàm f để thể hiện sự biến đổi của lớp tái phát của RNN:

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \tag{10.7.1}$$

Nói chung, bộ mã hóa biến đổi các trạng thái ẩn ở tất cả các bước thời gian thành biến ngữ cảnh thông qua một chức năng tùy chỉnh q:

$$\mathbf{c} = q(\mathbf{h}_1, \ldots, \mathbf{h}_T). \tag{10.7.2}$$

Ví dụ, khi chọn $q(\mathbf{h}_1, \ldots, \mathbf{h}_T) = \mathbf{h}_T$ chẳng hạn như trong Fig. 10.7.1, biến ngữ cảnh chỉ là trạng thái ẩn $\mathbf{h}_T$ của chuỗi đầu vào ở bước thời gian cuối cùng.

Cho đến nay chúng ta đã sử dụng RNN một chiều để thiết kế bộ mã hóa, trong đó trạng thái ẩn chỉ phụ thuộc vào dãy con đầu vào tại và trước bước thời gian của trạng thái ẩn. Chúng tôi cũng có thể xây dựng bộ mã hóa bằng cách sử dụng RNN hai chiều. Trong trường hợp này, trạng thái ẩn phụ thuộc vào dãy con trước và sau bước thời gian (bao gồm cả đầu vào ở bước thời gian hiện tại), mã hóa thông tin của toàn bộ chuỗi.

Bây giờ chúng ta hãy thực hiện bộ mã hóa RNN. Lưu ý rằng chúng ta sử dụng một lớp * nhúng * để lấy vectơ tính năng cho mỗi mã thông báo trong chuỗi đầu vào. Trọng lượng của một lớp nhúng là một ma trận có số hàng bằng với kích thước của từ vựng đầu vào (`vocab_size`) và số cột bằng với kích thước của vectơ đối tượng (`embed_size`). Đối với bất kỳ chỉ số mã thông báo đầu vào i, lớp nhúng lấy hàng i^{th} (bắt đầu từ 0) của ma trận trọng lượng để trả về vectơ tính năng của nó. Bên cạnh đó, ở đây chúng tôi chọn một GRU nhiều lớp để thực hiện bộ mã hóa.

```
#@save
class Seq2SeqEncoder(d2l.Encoder):
    """The RNN encoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        # Embedding layer
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
                          dropout=dropout)

    def forward(self, X, *args):
        # The output `X` shape: (`batch_size`, `num_steps`, `embed_size`)
        X = self.embedding(X)
        # In RNN models, the first axis corresponds to time steps
        X = X.permute(1, 0, 2)
        # When state is not mentioned, it defaults to zeros
        output, state = self.rnn(X)
        # `output` shape: (`num_steps`, `batch_size`, `num_hiddens`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

Các biến trả về của các lớp tái phát đã được giải thích trong Section 9.6. Chúng ta hãy vẫn sử dụng một ví dụ cụ thể để minh họa việc triển khai bộ mã hóa ở trên. Dưới đây chúng tôi khởi tạo một bộ mã hóa GRU hai lớp có số lượng đơn vị ẩn là 16. Với một minibatch các đầu vào chuỗi `X` (kích thước lô: 4, số bước thời gian: 7), các trạng thái ẩn của lớp cuối cùng ở tất cả các bước thời gian (`output` trả về bởi các lớp lặp lại của bộ mã hóa) là một tensor của hình dạng (số bước thời gian, kích thước lô, số đơn vị ẩn).

```
encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)
output, state = encoder(X)
output.shape
```

```
torch.Size([7, 4, 16])
```

Vì GRU được sử dụng ở đây, hình dạng của các trạng thái ẩn nhiều lớp ở bước thời gian cuối cùng là (số lớp ẩn, kích thước lô, số đơn vị ẩn). Nếu một LSTM được sử dụng, thông tin tế bào bộ nhớ cũng sẽ được chứa trong `state`.

```
state.shape
```

```
torch.Size([2, 4, 16])
```

10.7.2 Decoder

Như chúng tôi vừa đề cập, biến ngữ cảnh $\mathbf{c}$ của đầu ra của bộ mã hóa mã hóa toàn bộ chuỗi đầu vào $x_1, \ldots, x_T$. Với chuỗi đầu ra $y_1, y_2, \ldots, y_{T'}$ từ tập dữ liệu đào tạo, cho mỗi bước t' thời gian (biểu tượng khác với bước thời gian t của chuỗi đầu vào hoặc bộ mã hóa), xác suất của đầu ra bộ giải mã $y_{t'}$ có điều kiện trên dãy đầu ra trước $y_1, \ldots, y_{t'-1}$ và biến ngữ cảnh $\mathbf{c}$, tức là, $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$.

Để mô hình hóa xác suất có điều kiện này trên các chuỗi, chúng ta có thể sử dụng một RNN khác làm bộ giải mã. Bất cứ lúc nào bước t' trên chuỗi đầu ra, RNN lấy đầu ra $y_{t'-1}$ từ bước thời gian trước và biến ngữ cảnh $\mathbf{c}$ làm đầu vào của nó, sau đó biến đổi chúng và trạng thái ẩn trước đó $\mathbf{s}_{t'-1}$ thành trạng thái ẩn $\mathbf{s}_{t'}$ ở bước thời gian hiện tại. Kết quả là, chúng ta có thể sử dụng một hàm g để thể hiện sự biến đổi của lớp ẩn của bộ giải mã:

$$\mathbf{s}_{t'} = g(y_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \tag{10.7.3}$$

Sau khi có được trạng thái ẩn của bộ giải mã, chúng ta có thể sử dụng một lớp đầu ra và hoạt động softmax để tính toán phân phối xác suất có điều kiện $P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$ cho đầu ra tại bước thời điểm t'.

Sau Fig. 10.7.1, khi triển khai bộ giải mã như sau, chúng tôi trực tiếp sử dụng trạng thái ẩn ở bước thời gian cuối cùng của bộ mã hóa để khởi tạo trạng thái ẩn của bộ giải mã. Điều này đòi hỏi bộ mã hóa RNN và bộ giải mã RNN có cùng số lớp và các đơn vị ẩn. Để kết hợp thêm thông tin chuỗi đầu vào được mã hóa, biến ngữ cảnh được nối với đầu vào bộ giải mã ở tất cả các bước thời gian. Để dự đoán phân phối xác suất của mã thông báo đầu ra, một lớp kết nối hoàn toàn được sử dụng để biến đổi trạng thái ẩn ở lớp cuối cùng của bộ giải mã RNN.

```
class Seq2SeqDecoder(d2l.Decoder):
    """The RNN decoder for sequence to sequence learning."""
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
```

(continues on next page)

(continued from previous page)

```
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens, num_layers,
                          dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        # The output `X` shape: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        # Broadcast `context` so it has the same `num_steps` as `X`
        context = state[-1].repeat(X.shape[0], 1, 1)
        X_and_context = torch.cat((X, context), 2)
        output, state = self.rnn(X_and_context, state)
        output = self.dense(output).permute(1, 0, 2)
        # `output` shape: (`batch_size`, `num_steps`, `vocab_size`)
        # `state` shape: (`num_layers`, `batch_size`, `num_hiddens`)
        return output, state
```

Để minh họa bộ giải mã đã thực hiện, dưới đây chúng tôi khởi tạo nó với các siêu tham số tương tự từ bộ mã hóa đã nói ở trên. Như chúng ta có thể thấy, hình dạng đầu ra của bộ giải mã trở thành (kích thước lô, số bước thời gian, kích thước từ vựng), trong đó kích thước cuối cùng của tensor lưu trữ phân phối mã thông báo dự đoán.

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
decoder.eval()
state = decoder.init_state(encoder(X))
output, state = decoder(X, state)
output.shape, state.shape
```

```
(torch.Size([4, 7, 10]), torch.Size([2, 4, 16]))
```

Tóm lại, các lớp trong mô hình bộ mã hóa-giải mã RNN ở trên được minh họa trong Fig. 10.7.2.

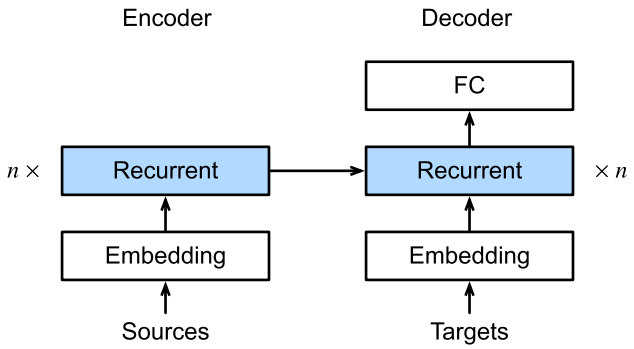


Fig. 10.7.2: Layers in an RNN encoder-decoder model.

10.7.3 Chức năng mất

Tại mỗi bước thời gian, bộ giải mã dự đoán phân phối xác suất cho các mã thông báo đầu ra. Tương tự như mô hình hóa ngôn ngữ, chúng ta có thể áp dụng softmax để có được sự phân bố và tính toán tổn thất chéo entropy để tối ưu hóa. Nhớ lại Section 10.5 rằng các thẻ đệm đặc biệt được gắn vào cuối chuỗi để các chuỗi có độ dài khác nhau có thể được tải hiệu quả trong các minibatches có cùng hình dạng. Tuy nhiên, dự đoán của các mã thông báo đệm nên được loại trừ khỏi các tính toán mất mát.

Để kết thúc này, chúng ta có thể sử dụng hàm `sequence_mask` sau để mask các mục không liên quan với giá trị bằng 0 vì vậy sau này nhân của bất kỳ dự đoán không liên quan nào với 0 bằng 0. Ví dụ: nếu độ dài hợp lệ của hai chuỗi không bao gồm thẻ đệm là một và hai, tương ứng, các mục còn lại sau cái đầu tiên và hai mục đầu tiên được xóa thành số không.

```
#@save
def sequence_mask(X, valid_len, value=0):
    """Mask irrelevant entries in sequences."""
    maxlen = X.size(1)
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X

X = torch.tensor([[1, 2, 3], [4, 5, 6]])
sequence_mask(X, torch.tensor([1, 2]))
```

```
tensor([[1, 0, 0],
        [4, 5, 0]])
```

Chúng tôi cũng có thể che dấu tất cả các mục trên vài trục cuối cùng. Nếu bạn thích, bạn thậm chí có thể chỉ định để thay thế các mục đó bằng giá trị không phải bằng không.

```
X = torch.ones(2, 3, 4)
sequence_mask(X, torch.tensor([1, 2]), value=-1)
```

```
tensor([[[ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.],
         [-1., -1., -1., -1.]],

        [[ 1.,  1.,  1.,  1.],
         [ 1.,  1.,  1.,  1.],
         [-1., -1., -1., -1.]]])
```

Bây giờ chúng ta có thể mở rộng lỗ liên entropy softmax để cho phép che dấu các dự đoán không liên quan Ban đầu, mặt nạ cho tất cả các token dự đoán được đặt thành một. Khi độ dài hợp lệ được đưa ra, mặt nạ tương ứng với bất kỳ mã thông báo padding nào sẽ bị xóa về 0. Cuối cùng, khoản lỗ cho tất cả các mã thông báo sẽ được mặt nạ tăng gấp bội để lọc ra những dự đoán không liên quan của các mã thông báo đệm trong khoản lỗ.

```
#@save
class MaskedSoftmaxCELoss(nn.CrossEntropyLoss):
    """The softmax cross-entropy loss with masks."""
    # `pred` shape: (`batch_size`, `num_steps`, `vocab_size`)
    # `label` shape: (`batch_size`, `num_steps`)
```

(continues on next page)

(continued from previous page)

```
    # `valid_len` shape: (`batch_size`,)
    def forward(self, pred, label, valid_len):
        weights = torch.ones_like(label)
        weights = sequence_mask(weights, valid_len)
        self.reduction='none'
        unweighted_loss = super(MaskedSoftmaxCELoss, self).forward(
            pred.permute(0, 2, 1), label)
        weighted_loss = (unweighted_loss * weights).mean(dim=1)
        return weighted_loss
```

Đối với a sanity check, chúng ta có thể tạo ra ba chuỗi giống hệt nhau. Sau đó, chúng ta có thể chỉ định rằng độ dài hợp lệ của các chuỗi này là 4, 2 và 0, tương ứng. Do đó, sự mất mát của chuỗi đầu tiên phải lớn gấp đôi so với chuỗi thứ hai, trong khi chuỗi thứ ba phải có tổn thất bằng không.

```
loss = MaskedSoftmaxCELoss()
loss(torch.ones(3, 4, 10), torch.ones((3, 4), dtype=torch.long),
     torch.tensor([4, 2, 0]))
```

```
tensor([2.3026, 1.1513, 0.0000])
```

10.7.4 Đào tạo

Trong vòng đào tạo sau, chúng tôi nối mã thông báo bắt đầu chuỗi đặc biệt và chuỗi đầu ra ban đầu không bao gồm token cuối cùng làm đầu vào cho bộ giải mã, như thể hiện trong Fig. 10.7.1. Điều này được gọi là *giáo viên forcing* vì chuỗi đầu ra ban đầu (nhãn mã thông báo) được đưa vào bộ giải mã. Ngoài ra, chúng ta cũng có thể cung cấp token *predicted* từ bước thời gian trước làm đầu vào hiện tại cho bộ giải mã.

```
#@save
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device):
    """Train a model for sequence to sequence."""
    def xavier_init_weights(m):
        if type(m) == nn.Linear:
            nn.init.xavier_uniform_(m.weight)
        if type(m) == nn.GRU:
            for param in m._flat_weights_names:
                if "weight" in param:
                    nn.init.xavier_uniform_(m._parameters[param])
    net.apply(xavier_init_weights)
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    loss = MaskedSoftmaxCELoss()
    net.train()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2)  # Sum of training loss, no. of tokens
        for batch in data_iter:
            optimizer.zero_grad()
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
```

(continues on next page)

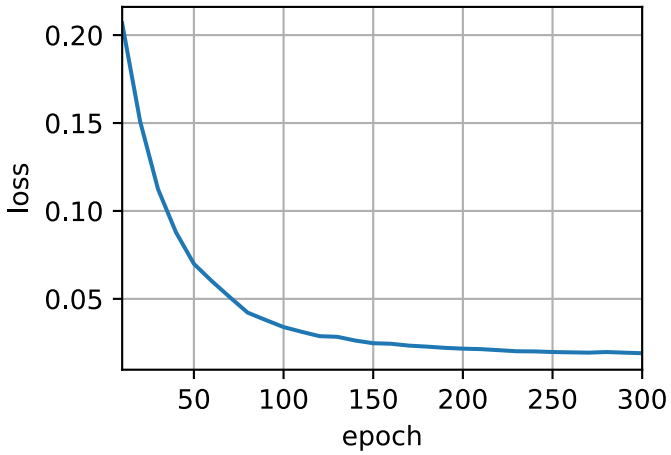
(continued from previous page)

```
                                device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1)  # Teacher forcing
            Y_hat, _ = net(X, dec_input, X_valid_len)
            l = loss(Y_hat, Y, Y_valid_len)
            l.sum().backward()  # Make the loss scalar for `backward`
            d2l.grad_clipping(net, 1)
            num_tokens = Y_valid_len.sum()
            optimizer.step()
            with torch.no_grad():
                metric.add(l.sum(), num_tokens)
        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, {metric[1] / timer.stop():.1f} '
          f'tokens/sec on {str(device)}')
```

Bây giờ chúng ta có thể tạo và đào tạo một mô hình bộ mã hóa-giải mã RNN cho trình tự để học trình tự trên bộ dữ liệu dịch máy.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 300, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.019, 11708.3 tokens/sec on cuda:0
```



10.7.5 Prediction

Để dự đoán mã thông báo chuỗi đầu ra theo mã thông báo, tại mỗi bước thời gian giải mã, token dự đoán từ bước thời gian trước đó được đưa vào bộ giải mã như một đầu vào. Tương tự như đào tạo, tại bước thời gian ban đầu, mã thông báo bắt đầu trình tự (" ") được đưa vào bộ giải mã. Quá trình dự đoán này được minh họa trong Fig. 10.7.3. Khi mã thông báo end-of-sequence (" ") được dự đoán, dự đoán của chuỗi đầu ra đã hoàn tất.

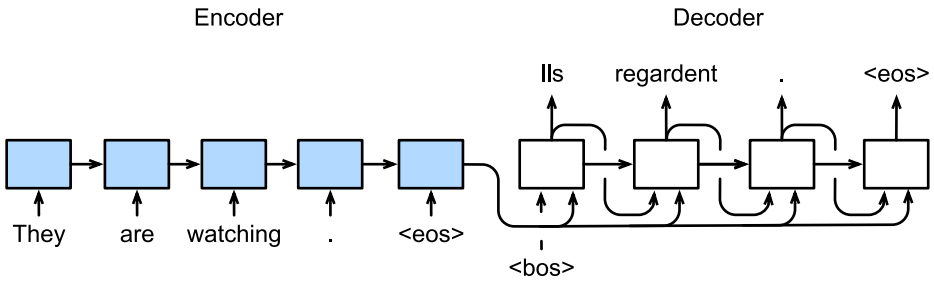


Fig. 10.7.3: Predicting the output sequence token by token using an RNN encoder-decoder.

Chúng tôi sẽ giới thiệu các chiến lược khác nhau để tạo chuỗi trong Section 10.8.

```
#@save
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab, num_steps,
                    device, save_attention_weights=False):
    """Predict for sequence to sequence."""
    # Set `net` to eval mode for inference
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    # Add the batch axis
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)
    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)
    # Add the batch axis
    dec_X = torch.unsqueeze(torch.tensor(
        [tgt_vocab['<bos>']], dtype=torch.long, device=device), dim=0)
    output_seq, attention_weight_seq = [], []
    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        # We use the token with the highest prediction likelihood as the input
        # of the decoder at the next time step
        dec_X = Y.argmax(dim=2)
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()
        # Save attention weights (to be covered later)
        if save_attention_weights:
            attention_weight_seq.append(net.decoder.attention_weights)
        # Once the end-of-sequence token is predicted, the generation of the
        # output sequence is complete
        if pred == tgt_vocab['<eos>']:
            break
```

(continues on next page)

```
        output_seq.append(pred)
    return ' '.join(tgt_vocab.to_tokens(output_seq)), attention_weight_seq
```

10.7.6 Đánh giá các chuỗi dự đoán

Chúng ta có thể đánh giá một chuỗi dự đoán bằng cách so sánh nó với chuỗi nhãn (sự thật mặt đất). BLEU (Nghiên cứu đánh giá song ngữ), mặc dù ban đầu được đề xuất để đánh giá kết quả dịch máy (Papineni et al., 2002), đã được sử dụng rộng rãi trong việc đo lường chất lượng trình tự đầu ra cho các ứng dụng khác nhau. Về nguyên tắc, đối với bất kỳ n-gram nào trong trình tự dự đoán, BLEU đánh giá liệu n-gram này có xuất hiện trong trình tự nhãn hay không.

Biểu thị bằng p_n độ chính xác của n-gram, là tỷ lệ của số lượng phù hợp n-gram trong các chuỗi dự đoán và nhãn với số n-gram trong trình tự dự đoán. Để giải thích, đưa ra một chuỗi nhãn A, B, C, D, E, F, và một chuỗi dự đoán A, B, B, C, D, chúng tôi có $p_1 = 4/5$, $p_1 = 4/5$, $p_2 = 3/4$ 293624, và $p_4 = 0$. Bên cạnh đó, hãy để $\text{len}_{\text{label}}$ và len_{pred} là số lượng thẻ trong chuỗi nhãn và trình tự dự đoán, tương ứng. Sau đó, BLEU được định nghĩa là

$$\exp\left(\min\left(0, 1 - \frac{\text{len}_{\text{label}}}{\text{len}_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n}, \tag{10.7.4}$$

trong đó k là n-gram dài nhất để phù hợp.

Dựa trên định nghĩa của BLEU năm (10.7.4), bất cứ khi nào trình tự dự đoán giống như chuỗi nhãn, BLEU là 1. Hơn nữa, vì kết hợp lâu hơn n-gram khó khăn hơn, BLEU gán trọng lượng lớn hơn cho độ chính xác n-gram dài hơn. Cụ thể, khi p_n được cố định, $p_n^{1/2^n}$ tăng lên khi n phát triển (giấy gốc sử dụng $p_n^{1/n}$). Hơn nữa, vì dự đoán các chuỗi ngắn hơn có xu hướng đạt được giá trị p_n cao hơn, hệ số trước thuật ngữ nhân trong (10.7.4) phạt các chuỗi dự đoán ngắn hơn. Ví dụ, khi $k = 2$, với chuỗi nhãn A, B, C, D, E, F và trình tự dự đoán A, B, mặc dù $p_1 = p_2 = 1$, hệ số hình phạt $\exp(1 - 6/2) \approx 0.14$ làm giảm BLEU.

Chúng tôi thực hiện biện pháp BLEU như sau.

```
def bleu(pred_seq, label_seq, k):  #@save
    """Compute the BLEU."""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)
    score = math.exp(min(0, 1 - len_label / len_pred))
    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)
        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1
        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1
        score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
    return score
```

Cuối cùng, chúng tôi sử dụng bộ giải mã RNN được đào tạo để dịch một vài câu tiếng Anh sang tiếng Pháp và tính toán BLEU của kết quả.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, attention_weight_seq = predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device)
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => attendez un homme ., bleu 0.000
i'm home . => je suis faible ., bleu 0.512
```

10.7.7 Tóm tắt

- Theo thiết kế của kiến trúc bộ mã hóa-giải mã, chúng ta có thể sử dụng hai RNNđể thiết kế một mô hình cho trình tự để học trình tự.
- Khi triển khai bộ mã hóa và bộ giải mã, chúng ta có thể sử dụng RNN nhiều lớp.
- Chúng ta có thể sử dụng mặt nạ để lọc ra các tính toán không liên quan, chẳng hạn như khi tính toán tổn thất.
- Trong đào tạo bộ mã hóa-giải mã, giáo viên buộc phương pháp tiếp cận cung cấp các chuỗi đầu ra ban đầu (trái ngược với dự đoán) vào bộ giải mã.
- BLEU là một thước đo phổ biến để đánh giá trình tự đầu ra bằng cách khớp n-gram giữa trình tự dự đoán và chuỗi nhãn.

10.7.8 Bài tập

1. Bạn có thể điều chỉnh các siêu tham số để cải thiện kết quả dịch thuật không?
2. Chạy lại thí nghiệm mà không cần sử dụng mặt nạ trong tính toán mất mát. Bạn quan sát kết quả gì? Tại sao?
3. Nếu bộ mã hóa và bộ giải mã khác nhau về số lượng lớp hoặc số lượng đơn vị ẩn, làm thế nào chúng ta có thể khởi tạo trạng thái ẩn của bộ giải mã?
4. Trong đào tạo, thay thế giáo viên buộc bằng việc cho ăn dự đoán ở bước thời gian trước vào bộ giải mã. Làm thế nào để điều này ảnh hưởng đến hiệu suất?
5. Chạy lại thí nghiệm bằng cách thay thế GRU bằng LSTM.
6. Có cách nào khác để thiết kế lớp đầu ra của bộ giải mã không?

Discussions[116]

[116] https://discuss.d2l.ai/t/1062

10.8 Tìm kiếm chùm

Năm Section 10.7, chúng tôi dự đoán mã thông báo chuỗi đầu ra bằng token cho đến khi token “” end-of-sequence đặc biệt được dự đoán. Trong phần này, chúng ta sẽ bắt đầu bằng cách chính thức hóa chiến lược tìm kiếm* tham lam này* và khám phá các vấn đề với nó, sau đó so sánh chiến lược này với các lựa chọn thay thế khác: *tìm kiếm đầy đủ* và * chùm tìm kiếm*.

Trước khi giới thiệu chính thức về tìm kiếm tham lam, chúng ta hãy chính thức hóa bài toán tìm kiếm bằng cách sử dụng cùng một ký hiệu toán học từ Section 10.7. Bất cứ lúc nào bước t', xác suất của đầu ra bộ giải mã $y_{t'}$ có điều kiện trên dãy thứ tự đầu ra $y_1, \ldots, y_{t'-1}$ trước t' và biến ngữ cảnh $\mathbf{c}$ mã hóa thông tin của chuỗi đầu vào. Để định lượng chi phí tính toán, biểu thị bằng $\mathcal{Y}$ (nó chứa “”) từ vựng đầu ra. Vì vậy, tính cardinality $|\mathcal{Y}|$ của bộ từ vựng này là kích thước từ vựng. Chúng ta cũng chỉ định số lượng thẻ tối đa của một chuỗi đầu ra là T'. Kết quả là, mục tiêu của chúng tôi là tìm kiếm một đầu ra lý tưởng từ tất cả các chuỗi đầu ra có thể có $\mathcal{O}(|\mathcal{Y}|^{T'})$. Tất nhiên, đối với tất cả các chuỗi đầu ra này, các phần bao gồm và sau “” sẽ bị loại bỏ trong đầu ra thực tế.

10.8.1 Tìm kiếm tham lam

Đầu tiên, chúng ta hãy xem xét một chiến lược đơn giản: * tìm kiếm tham lam*. Chiến lược này đã được sử dụng để dự đoán các trình tự trong Section 10.7. Trong tìm kiếm tham lam, bất cứ lúc nào bước t' của chuỗi đầu ra, chúng tôi tìm kiếm mã thông báo có xác suất có điều kiện cao nhất từ $\mathcal{Y}$, tức là,

$$y_{t'} = \operatorname*{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{10.8.1}$$

như đầu ra. Khi “” được xuất ra hoặc chuỗi đầu ra đã đạt đến độ dài tối đa T', chuỗi đầu ra được hoàn thành.

Vì vậy, những gì có thể đi sai với tìm kiếm tham lam? Trên thực tế, trình tự tối ưu * phải là chuỗi đầu ra với $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c})$ tối đa, là xác suất có điều kiện tạo ra một chuỗi đầu ra dựa trên chuỗi đầu vào. Thật không may, không có gì đảm bảo rằng trình tự tối ưu sẽ thu được bằng cách tìm kiếm tham lam.

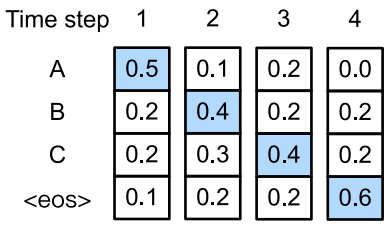


Fig. 10.8.1: At each time step, greedy search selects the token with the highest conditional probability.

Hãy để chúng tôi minh họa nó với một ví dụ. Giả sử có bốn mã thông báo “A”, “B”, “C”, và “” trong từ điển đầu ra. Năm Fig. 10.8.1, bốn số dưới mỗi bước thời gian đại diện cho xác suất có điều kiện tạo ra “A”, “B”, “C”, và “” tại bước thời điểm đó, tương ứng. Tại mỗi bước thời gian, tìm kiếm tham lam chọn mã thông báo có xác suất có điều kiện cao nhất. Do đó, chuỗi đầu ra “A”, “B”, “C”, và “” sẽ được dự đoán vào năm Fig. 10.8.1. Xác suất có điều kiện của chuỗi đầu ra này là $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$.

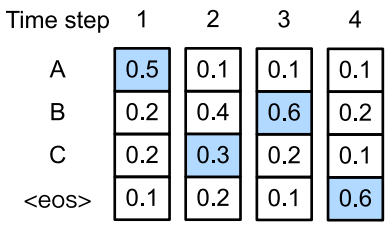


Fig. 10.8.2: The four numbers under each time step represent the conditional probabilities of generating “A”, “B”, “C”, and “<eos>” at that time step. At time step 2, the token “C”, which has the second highest conditional probability, is selected.

Tiếp theo, chúng ta hãy xem xét một ví dụ khác trong Fig. 10.8.2. Không giống như trong Fig. 10.8.1, tại thời điểm bước 2 chúng tôi chọn token “C” trong Fig. 10.8.2, có xác suất điều kiện cao nhất * giây*. Vì các dãy con đầu ra tại thời điểm bước 1 và 2, trên đó bước thời gian 3 dựa trên, đã thay đổi từ “A” và “B” trong Fig. 10.8.1 thành “A” và “C” trong Fig. 10.8.2, xác suất có điều kiện của mỗi mã thông báo tại thời điểm bước 3 cũng đã thay đổi trong Fig. 10.8.2. Giả sử rằng chúng ta chọn token “B” tại bước thời điểm 3. Bây giờ bước thời gian 4 là điều kiện trên dãy con đầu ra ở ba bước thời gian đầu tiên “A”, “C”, và “B”, khác với “A”, “B”, và “C” trong Fig. 10.8.1. Do đó, xác suất có điều kiện tạo ra từng token tại bước thời điểm 4 trong Fig. 10.8.2 cũng khác với xác suất có điều kiện trong Fig. 10.8.1. Kết quả là xác suất có điều kiện của chuỗi đầu ra “A”, “C”, “B” và “” trong Fig. 10.8.2 là $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, lớn hơn so với tìm kiếm tham lam vào năm Fig. 10.8.1. Trong ví dụ này, chuỗi đầu ra “A”, “B”, “C”, và “” thu được bằng cách tìm kiếm tham lam không phải là một chuỗi tối ưu.

10.8.2 Tìm kiếm đầy đủ

Nếu mục tiêu là để có được trình tự tối ưu, chúng tôi có thể xem xét sử dụng * tìm kiếm đầy đủ*: liệt kê toàn bộ tất cả các chuỗi đầu ra có thể có với xác suất có điều kiện của chúng, sau đó xuất ra một trong những xác suất có điều kiện cao nhất.

Mặc dù chúng ta có thể sử dụng tìm kiếm đầy đủ để có được trình tự tối ưu, nhưng chi phí tính toán của nó $\mathcal{O}(|\mathcal{Y}|^{T'})$ có thể là quá cao. Ví dụ: khi $|\mathcal{Y}| = 10000$ và $T' = 10$, chúng ta sẽ cần đánh giá trình tự $10000^{10} = 10^{40}$. Đây là bên cạnh không thể! Mặt khác, chi phí tính toán của tìm kiếm tham lam là $\mathcal{O}(|\mathcal{Y}|\,T')$: nó thường nhỏ hơn đáng kể so với tìm kiếm đầy đủ. Ví dụ, khi $|\mathcal{Y}| = 10000$ và $T' = 10$, chúng ta chỉ cần đánh giá $10000 \times 10 = 10^5$ trình tự.

10.8.3 Tìm kiếm chùm

Các quyết định về chiến lược tìm kiếm trình tự nằm trên một phổ, với những câu hỏi dễ dàng ở một trong hai cực đoan. Điều gì sẽ xảy ra nếu chỉ chính xác quan trọng? Rõ ràng, tìm kiếm đầy đủ. Điều gì sẽ xảy ra nếu chỉ có chi phí tính toán quan trọng? Rõ ràng, tìm kiếm tham lam. Một ứng dụng trong thế giới thực thường đặt ra một câu hỏi phức tạp, ở đâu đó giữa hai thái cực đó.

Chùm tìm kiếm là một phiên bản cải tiến của tìm kiếm tham lam. Nó có một siêu tham số có tên là kích thước chùm tia *, k. Vào thời điểm bước 1, chúng tôi chọn k token có xác suất có điều kiện cao nhất. Mỗi người trong số họ sẽ là mã thông báo đầu tiên của k trình tự đầu ra ứng cử viên, tương ứng. Tại mỗi bước thời gian tiếp theo, dựa trên trình tự đầu ra ứng cử viên k ở bước thời gian trước, chúng tôi tiếp tục chọn trình tự đầu ra ứng cử viên k với xác suất có điều kiện cao nhất từ $k\,|\mathcal{Y}|$ các lựa chọn có thể.

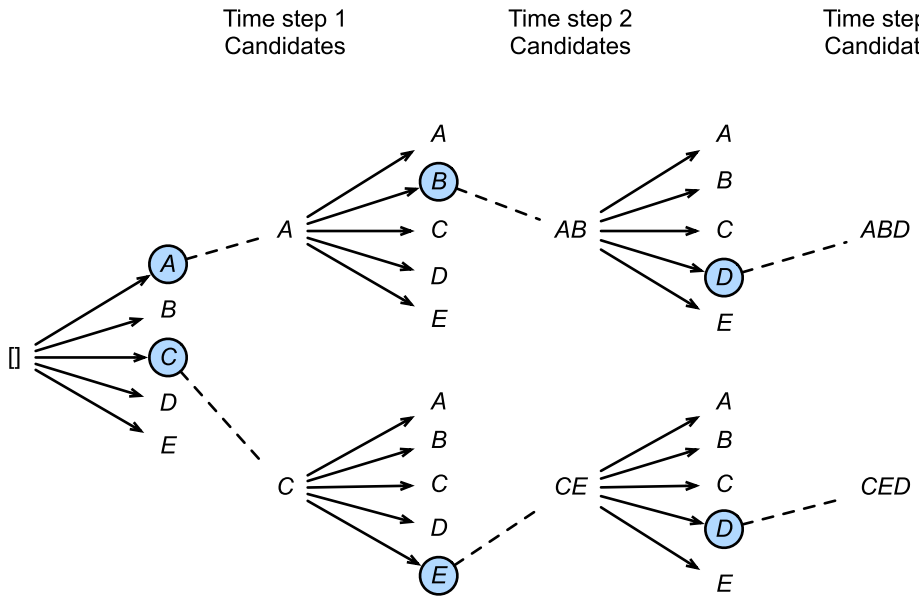


Fig. 10.8.3: The process of beam search (beam size: 2, maximum length of an output sequence: 3). The candidate output sequences are A, C, AB, CE, ABD, and CED.

Fig. 10.8.3 thể hiện quá trình tìm kiếm chùm tia với một ví dụ. Giả sử rằng từ vựng đầu ra chỉ chứa năm phần tử: $\mathcal{Y} = \{A, B, C, D, E\}$, trong đó một trong số chúng là "". Hãy để kích thước chùm là 2 và chiều dài tối đa của một chuỗi đầu ra là 3. Vào thời điểm bước 1, giả sử rằng các mã thông báo có xác suất có điều kiện cao nhất $P(y_1 \mid \mathbf{c})$ là A và C. Tại thời điểm bước 2, cho tất cả $y_2 \in \mathcal{Y}$, chúng tôi tính toán

$$\begin{aligned}P(A, y_2 \mid \mathbf{c}) &= P(A \mid \mathbf{c})P(y_2 \mid A, \mathbf{c}),\\ P(C, y_2 \mid \mathbf{c}) &= P(C \mid \mathbf{c})P(y_2 \mid C, \mathbf{c}),\end{aligned} \tag{10.8.2}$$

và chọn hai giá trị lớn nhất trong số mười giá trị này, nói $P(A, B \mid \mathbf{c})$ và $P(C, E \mid \mathbf{c})$. Sau đó, tại thời điểm bước 3, cho tất cả $y_3 \in \mathcal{Y}$, chúng tôi tính

$$\begin{aligned}P(A, B, y_3 \mid \mathbf{c}) &= P(A, B \mid \mathbf{c})P(y_3 \mid A, B, \mathbf{c}),\\ P(C, E, y_3 \mid \mathbf{c}) &= P(C, E \mid \mathbf{c})P(y_3 \mid C, E, \mathbf{c}),\end{aligned} \tag{10.8.3}$$

và chọn hai lớn nhất trong số mười giá trị này, nói $P(A, B, D \mid \mathbf{c})$ và $P(C, E, D \mid \mathbf{c})$. Kết quả là, chúng tôi nhận được sáu trình tự đầu ra ứng cử viên: (i) A; (ii) C; (iii) A, A, A, A, A 3617, D; và (vi) C, E, D.

Cuối cùng, chúng tôi có được tập hợp các chuỗi đầu ra ứng viên cuối cùng dựa trên sáu chuỗi này (ví dụ, loại bỏ các phần bao gồm và sau ""). Sau đó, chúng ta chọn dãy có điểm cao nhất sau làm chuỗi đầu ra:

$$\frac{1}{L^\alpha} \log P(y_1, \ldots, y_L \mid \mathbf{c}) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log P(y_{t'} \mid y_1, \ldots, y_{t'-1}, \mathbf{c}), \tag{10.8.4}$$

trong đó L là độ dài của trình tự ứng cử viên cuối cùng và α thường được đặt thành 0,75. Vì một chuỗi dài hơn có nhiều thuật ngữ logarit hơn trong tổng kết (10.8.4), thuật ngữ L^α trong mẫu số phạt các chuỗi dài.

Chi phí tính toán của tìm kiếm chùm tia là $\mathcal{O}(k\,|\mathcal{Y}|\,T')$. Kết quả này nằm giữa tìm kiếm tham lam và tìm kiếm đầy đủ. Trên thực tế, tìm kiếm tham lam có thể được coi là một loại tìm kiếm chùm đặc biệt với kích thước chùm tia là 1. Với sự lựa chọn linh hoạt về kích thước chùm tia, tìm kiếm chùm tia cung cấp sự cân bằng giữa độ chính xác so với chi phí tính toán.

10.8.4 Tóm tắt

- Các chiến lược tìm kiếm trình tự bao gồm tìm kiếm tham lam, tìm kiếm đầy đủ và tìm kiếm chùm tia.
- Tìm kiếm chùm tia cung cấp sự cân bằng giữa độ chính xác so với chi phí tính toán thông qua sự lựa chọn linh hoạt của kích thước chùm tia.

10.8.5 Bài tập

1. Chúng ta có thể coi tìm kiếm toàn diện như một loại tìm kiếm chùm đặc biệt không? Tại sao hoặc tại sao không?
2. Áp dụng tìm kiếm chùm tia trong vấn đề dịch máy trong Section 10.7. Kích thước chùm tia ảnh hưởng đến kết quả dịch thuật và tốc độ dự đoán như thế nào?
3. Chúng tôi đã sử dụng mô hình hóa ngôn ngữ để tạo văn bản sau tiền tố do người dùng cung cấp trong Section 9.5. Nó sử dụng loại chiến lược tìm kiếm nào? Bạn có thể cải thiện nó?

Discussions[117]

[117] https://discuss.d2l.ai/t/338

11 | Cơ chế chú ý

Dây thần kinh thị giác của hệ thống thị giác của linh trưởng nhận được đầu vào cảm giác lớn, vượt xa những gì não có thể xử lý hoàn toàn. May mắn thay, không phải tất cả các kích thích đều được tạo ra bằng nhau. Sự tập trung và tập trung ý thức đã cho phép các loài linh trưởng chú ý trực tiếp đến các đối tượng quan tâm, chẳng hạn như săn mồi và động vật ăn thịt, trong môi trường thị giác phức tạp. Khả năng chú ý đến chỉ một phần nhỏ của thông tin có ý nghĩa tiến hóa, cho phép con người sống và thành công.

Các nhà khoa học đã nghiên cứu sự chú ý trong lĩnh vực khoa học thần kinh nhận thức từ thế kỷ 19. Trong chương này, chúng ta sẽ bắt đầu bằng cách xem xét một khuôn khổ phổ biến giải thích cách sự chú ý được triển khai trong một cảnh trực quan. Lấy cảm hứng từ các tín hiệu chú ý trong khuôn khổ này, chúng tôi sẽ thiết kế các mô hình tận dụng các tín hiệu chú ý như vậy. Đáng chú ý là hồi quy hạt nhân Nadaraya-Waston vào năm 1964 là một minh chứng đơn giản về học máy với cơ chế chú ý*.

Tiếp theo, chúng tôi sẽ tiếp tục giới thiệu các chức năng chú ý đã được sử dụng rộng rãi trong thiết kế các mô hình chú ý trong học sâu. Cụ thể, chúng tôi sẽ chỉ ra cách sử dụng các chức năng này để thiết kế sự chú ý * Bahdanau*, một mô hình chú ý đột phá trong học sâu có thể sắp xếp hai chiều và có thể khác biệt.

Cuối cùng, được trang bị gần đây hơn *sự chú ý nhiều đầu* và *self-attention* thiết kế, chúng tôi sẽ mô tả kiến trúc * transformer* chỉ dựa trên cơ chế chú ý. Kể từ khi đề xuất của họ vào năm 2017, các máy biến áp đã phổ biến trong các ứng dụng học sâu hiện đại, chẳng hạn như trong các lĩnh vực ngôn ngữ, tầm nhìn, lời nói và học tập củng cố.

11.1 Chú ý tín hiệu

Cảm ơn bạn đã quan tâm đến cuốn sách này. Chú ý là một nguồn tài nguyên khan hiếm: tại thời điểm bạn đang đọc cuốn sách này và bỏ qua phần còn lại. Do đó, tương tự như tiền, sự chú ý của bạn đang được trả bằng chi phí cơ hội. Để đảm bảo rằng sự đầu tư của bạn chú ý ngay bây giờ là đáng giá, chúng tôi đã rất có động lực để chú ý cẩn thận để sản xuất một cuốn sách hay. Sự chú ý là nền tảng trong vòm của cuộc sống và giữ chìa khóa cho bất kỳ tác phẩm ngoại lệ nào.

Kể từ khi kinh tế học nghiên cứu việc phân bổ các nguồn lực khan hiếm, chúng ta đang ở trong thời đại của nền kinh tế chú ý, nơi sự chú ý của con người được coi là một hàng hóa hạn chế, có giá trị và khan hiếm có thể được trao đổi. Nhiều mô hình kinh doanh đã được phát triển để tận dụng nó. Trên các dịch vụ phát nhạc hoặc video, chúng tôi chú ý đến quảng cáo của họ hoặc trả tiền để ẩn chúng. Để tăng trưởng trong thế giới trò chơi trực tuyến, chúng tôi hoặc chú ý tham gia vào các trận chiến, thu hút các game thủ mới hoặc trả tiền để ngay lập tức trở nên mạnh mẽ. Không có gì đến miễn phí.

Nói chung, thông tin trong môi trường của chúng ta không khan hiếm, chú ý là. Khi kiểm tra cảnh thị giác, dây thần kinh thị giác của chúng ta nhận được thông tin theo thứ tự 10^8 bit mỗi giây, vượt xa những gì bộ não của chúng ta có thể xử lý hoàn toàn. May mắn thay, tổ tiên của chúng ta đã học được từ kinh nghiệm (còn được gọi là dữ liệu) rằng * không phải tất cả các đầu vào cảm giác được tạo ra bằng nhau*. Trong suốt lịch sử loài người, khả năng chỉ đạo sự chú ý đến một phần nhỏ thông tin quan tâm đã cho phép bộ não của chúng ta

phân bổ tài nguyên thông minh hơn để tồn tại, phát triển và xã hội hóa, chẳng hạn như phát hiện kẻ săn mồi, săn mồi và bạn tình.

11.1.1 Các tín hiệu chú ý trong sinh học

Để giải thích cách sự chú ý của chúng ta được triển khai trong thế giới thị giác, một khung hai thành phần đã xuất hiện và phổ biến. Ý tưởng này có từ William James vào những năm 1890, người được coi là "cha đẻ của tâm lý học Mỹ" [James.2007]. Trong khuôn khổ này, các đối tượng có chọn lọc hướng sự chú ý của sự chú ý bằng cách sử dụng cả hai tín hiệu *phi ý dị* và *tín hiệu ý thích*.

Các cue không có ý định dựa trên sự nổi bật và dễ thấy của các vật thể trong môi trường. Hãy tưởng tượng có năm đối tượng trước mặt bạn: một tờ báo, một bài báo nghiên cứu, một tách cà phê, một cuốn sổ và một cuốn sách như năm Fig. 11.1.1. Trong khi tất cả các sản phẩm giấy được in màu đen và trắng, cốc cà phê có màu đỏ. Nói cách khác, cà phê này thực sự nổi bật và dễ thấy trong môi trường thị giác này, tự động và vô tình thu hút sự chú ý. Vì vậy, bạn mang fovea (trung tâm của điểm vàng nơi thị lực cao nhất) lên cà phê như thể hiện trong Fig. 11.1.1.

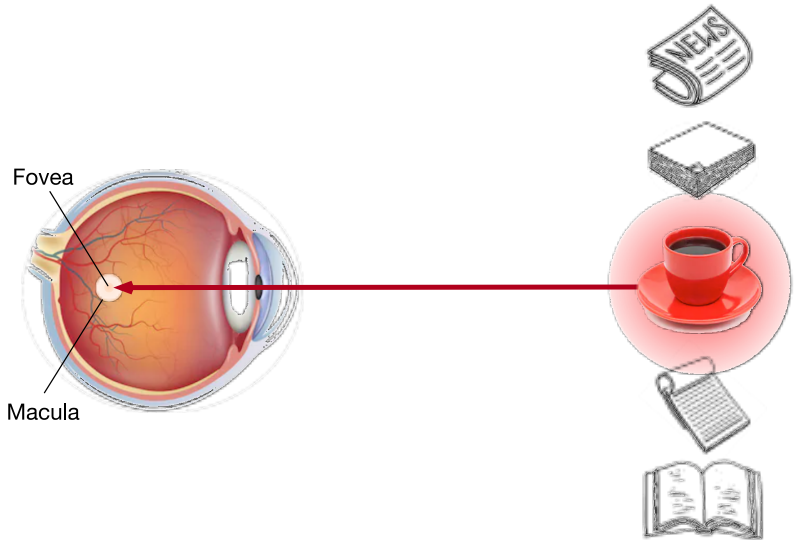


Fig. 11.1.1: Using the nonvolitional cue based on saliency (red cup, non-paper), attention is involuntarily directed to the coffee.

Sau khi uống cà phê, bạn trở nên caffein và muốn đọc một cuốn sách. Vì vậy, bạn quay đầu, tập trung lại đôi mắt của bạn, và nhìn vào cuốn sách như mô tả trong Fig. 11.1.2. Khác với trường hợp trong Fig. 11.1.1 nơi cà phê thiên vị bạn hướng tới lựa chọn dựa trên sự nổi bật, trong trường hợp phụ thuộc vào nhiệm vụ này, bạn chọn cuốn sách dưới sự kiểm soát nhận thức và ý chí. Sử dụng cue ý chí dựa trên các tiêu chí lựa chọn biến, hình thức chú ý này có chủ ý hơn. Nó cũng mạnh hơn với nỗ lực tự nguyện của đối tượng.

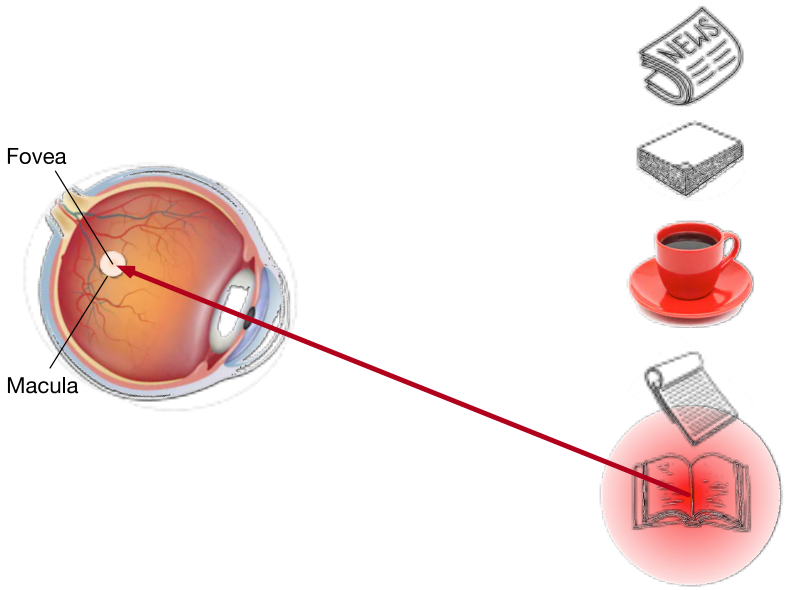


Fig. 11.1.2: Using the volitional cue (want to read a book) that is task-dependent, attention is directed to the book under volitional control.

11.1.2 Truy vấn, Khóa và Giá trị

Lấy cảm hứng từ các tín hiệu chú ý không có ý định và ý chí giải thích việc triển khai chú ý, trong phần sau, chúng tôi sẽ mô tả một khuôn khổ để thiết kế các cơ chế chú ý bằng cách kết hợp hai tín hiệu chú ý này.

Để bắt đầu, hãy xem xét trường hợp đơn giản hơn, nơi chỉ có các tín hiệu phi nghĩa có sẵn. Để lựa chọn thiên vị so với các đầu vào cảm giác, chúng ta có thể chỉ cần sử dụng một lớp được kết nối đầy đủ được tham số hóa hoặc thậm chí là tổng hợp tối đa hoặc trung bình không tham số hóa.

Do đó, điều đặt ra các cơ chế chú ý khác biệt với các lớp hoặc các lớp kết nối hoàn toàn đó là sự bao gồm các tín hiệu ý chí. Trong bối cảnh của các cơ chế chú ý, chúng tôi đề cập đến các tín hiệu ý chí là * queries*. Với bất kỳ truy vấn nào, lựa chọn thiên vị cơ chế chú ý so với các đầu vào cảm giác (ví dụ: biểu diễn tính năng trung gian) thông qua * chú ý cùng*. Những đầu vào cảm giác này được gọi là *giá trị* trong bối cảnh các cơ chế chú ý. Nói chung hơn, mọi giá trị được ghép nối với một * key*, có thể được nghĩ đến gợi ý phi nghĩa của đầu vào cảm giác đó. Như được hiển thị trong Fig. 11.1.3, chúng ta có thể thiết kế tập hợp sự chú ý để truy vấn đã cho (cue ý chí) có thể tương tác với các phím (tín hiệu phi nghĩa), hướng dẫn lựa chọn thiên vị so với các giá trị (đầu vào cảm giác).

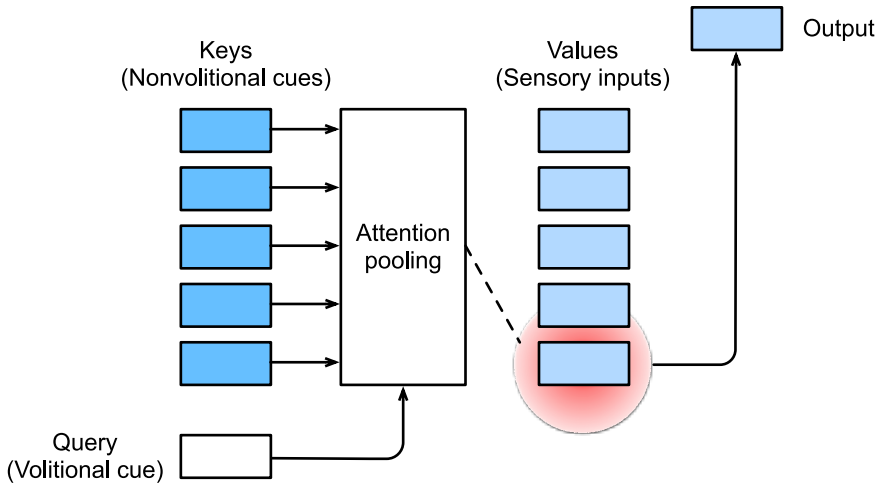


Fig. 11.1.3: Attention mechanisms bias selection over values (sensory inputs) via attention pooling, which incorporates queries (volitional cues) and keys (nonvolitional cues).

Lưu ý rằng có nhiều lựa chọn thay thế cho việc thiết kế các cơ chế chú ý. Ví dụ, chúng ta có thể thiết kế một mô hình chú ý không phân biệt có thể được đào tạo bằng phương pháp học tăng cường [Mnih.Heess.Graves.ea.2014]. Với sự thống trị của khuôn khổ trong Fig. 11.1.3, các mô hình trong khuôn khổ này sẽ là trung tâm của sự chú ý của chúng tôi trong chương này.

11.1.3 Hình dung của sự chú ý

Tổng hợp trung bình có thể được coi là trung bình trọng số của đầu vào, trong đó trọng lượng là đồng nhất. Trong thực tế, sự chú ý tập hợp các giá trị sử dụng trung bình trọng số, trong đó trọng lượng được tính toán giữa truy vấn đã cho và các khóa khác nhau.

```
import torch
from d2l import torch as d2l
```

Để hình dung trọng lượng chú ý, chúng tôi xác định hàm `show_heatmaps`. Đầu vào của nó `matrices` có hình dạng (số hàng để hiển thị, số cột để hiển thị, số truy vấn, số phím).

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
                  cmap='Reds'):
    """Show heatmaps of matrices."""
    d2l.use_svg_display()
    num_rows, num_cols = matrices.shape[0], matrices.shape[1]
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
                                 sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            pcm = ax.imshow(matrix.detach().numpy(), cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
            if j == 0:
                ax.set_ylabel(ylabel)
            if titles:
```

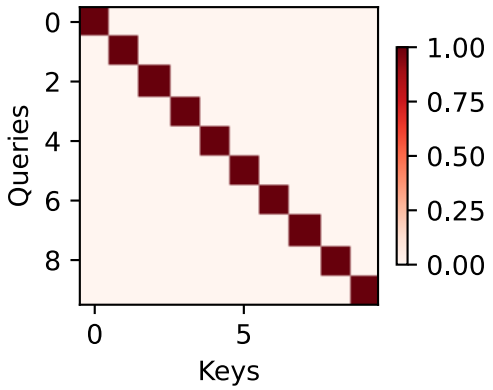
(continues on next page)

(continued from previous page)

```
                ax.set_title(titles[j])
    fig.colorbar(pcm, ax=axes, shrink=0.6);
```

Để trình diễn, chúng tôi xem xét một trường hợp đơn giản trong đó trọng lượng chú ý chỉ là một khi truy vấn và khóa giống nhau; nếu không thì nó bằng 0.

```
attention_weights = torch.eye(10).reshape((1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')
```



Trong các phần tiếp theo, chúng ta thường sẽ gọi hàm này để hình dung trọng lượng chú ý.

11.1.4 Tóm tắt

- Sự chú ý của con người là một nguồn tài nguyên hạn chế, có giá trị và khan hiếm.
- Đối tượng có chọn lọc trực tiếp sự chú ý bằng cách sử dụng cả tín hiệu phi nghĩa và ý chí. Cái trước dựa trên sự nổi bật và cái sau phụ thuộc vào nhiệm vụ.
- Các cơ chế chú ý khác với các lớp được kết nối hoàn toàn hoặc các lớp tổng hợp do bao gồm các tín hiệu ý chí.
- Các cơ chế chú ý lựa chọn thiên vị so với các giá trị (đầu vào cảm giác) thông qua tập hợp sự chú ý, kết hợp các truy vấn (tín hiệu ý chí) và các phím (tín hiệu phi ý chí). Các phím và giá trị được ghép nối.
- Chúng ta có thể hình dung trọng lượng chú ý giữa các truy vấn và các phím.

11.1.5 Bài tập

1. Điều gì có thể là cue ý chí khi giải mã một mã thông báo chuỗi bằng mã thông báo trong dịch máy? Các tín hiệu phi nghĩa và đầu vào cảm giác là gì?
2. Tạo ngẫu nhiên ma trận 10×10 và sử dụng thao tác softmax để đảm bảo mỗi hàng là một phân phối xác suất hợp lệ. Hình dung trọng lượng chú ý đầu ra.

Discussions[118]

[118] https://discuss.d2l.ai/t/1592

11.2 Chú ý Pooling: Hồi quy hạt nhân Nadaraya-Watson

Bây giờ bạn đã biết các thành phần chính của các cơ chế chú ý theo khuôn khổ trong Fig. 11.1.3. Để tái lập lại, các tương tác giữa các truy vấn (tín hiệu ý chí) và các phím (tín hiệu không có ý định) dẫn đến *chú ý cùng*. Sự chú ý tập hợp chọn lọc các giá trị (đầu vào cảm giác) để tạo ra đầu ra. Trong phần này, chúng tôi sẽ mô tả chi tiết hơn để cung cấp cho bạn cái nhìn cấp cao về cách các cơ chế chú ý hoạt động trong thực tế. Cụ thể, mô hình hồi quy hạt nhân Nadaraya-Watson đề xuất năm 1964 là một ví dụ đơn giản nhưng đầy đủ để chứng minh máy học với các cơ chế chú ý.

```
import torch
from torch import nn
from d2l import torch as d2l
```

11.2.1 Tạo ra bộ dữ liệu

Để giữ cho mọi thứ đơn giản, chúng ta hãy xem xét vấn đề hồi quy sau: đưa ra một tập dữ liệu của cặp đầu vào-đầu ra $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, làm thế nào để tìm hiểu f để dự đoán đầu ra $\hat{y} = f(x)$ cho bất kỳ đầu vào mới x?

Ở đây chúng ta tạo ra một bộ dữ liệu nhân tạo theo hàm phi tuyến sau với thuật ngữ nhiễu ϵ:

$$y_i = 2\sin(x_i) + x_i^{0.8} + \epsilon, \tag{11.2.1}$$

trong đó ϵ tuân theo một phân phối bình thường với 0 trung bình và độ lệch chuẩn 0,5. Cả 50 ví dụ đào tạo và 50 ví dụ thử nghiệm được tạo ra. Để hình dung rõ hơn mô hình chú ý sau này, các đầu vào đào tạo được sắp xếp.

```
n_train = 50  # No. of training examples
x_train, _ = torch.sort(torch.rand(n_train) * 5)   # Training inputs
```

```
def f(x):
    return 2 * torch.sin(x) + x**0.8

y_train = f(x_train) + torch.normal(0.0, 0.5, (n_train,))  # Training outputs
x_test = torch.arange(0, 5, 0.1)  # Testing examples
y_truth = f(x_test)  # Ground-truth outputs for the testing examples
n_test = len(x_test)  # No. of testing examples
n_test
```

```
50
```

Hàm sau vẽ tất cả các ví dụ đào tạo (được biểu diễn bằng các vòng tròn), hàm tạo dữ liệu đất-chân lý `f` không có thuật ngữ nhiễu (được dán nhãn bởi "Truth"), và hàm dự đoán đã học (được dán nhãn bởi "Pred").

```
def plot_kernel_reg(y_hat):
    d2l.plot(x_test, [y_truth, y_hat], 'x', 'y', legend=['Truth', 'Pred'],
             xlim=[0, 5], ylim=[-1, 5])
    d2l.plt.plot(x_train, y_train, 'o', alpha=0.5);
```

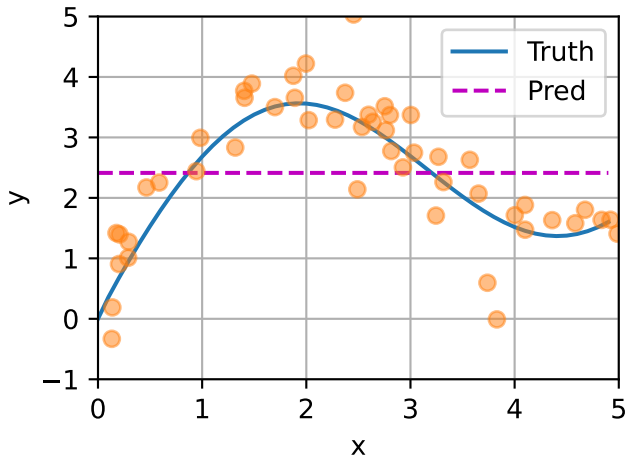
11.2.2 Pooling trung bình

Chúng ta bắt đầu với sự ước tính "ngu ngốc nhất" của thế giới cho vấn đề hồi quy này: sử dụng tổng hợp trung bình đến trung bình trên tất cả các đầu ra đào tạo:

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} y_i, \tag{11.2.2}$$

được vẽ dưới đây. Như chúng ta có thể thấy, ước tính này thực sự không quá thông minh.

```
y_hat = torch.repeat_interleave(y_train.mean(), n_test)
plot_kernel_reg(y_hat)
```



11.2.3 Không tham số Chú ý Pooling

Rõ ràng, tổng hợp trung bình bỏ qua các đầu vào x_i. Một ý tưởng tốt hơn đã được Nadaraya [Nadaraya.1964] và Watson [Watson.1964] đề xuất để cân nhắc các đầu ra y_i theo vị trí đầu vào của chúng:

$$f(x) = \sum_{i=1}^{n} \frac{K(x - x_i)}{\sum_{j=1}^{n} K(x - x_j)} y_i, \tag{11.2.3}$$

trong đó K là một * kernel. *Các ước tính trong :eq:`eq_nadaraya-watson` được gọi là * Nadaraya-Watson kernel regression*. Ở đây chúng tôi sẽ không đi sâu vào chi tiết của hạt nhân. Nhớ lại khuôn khổ của các cơ chế chú ý trong :numref:`fig_qkv`. Từ quan điểm của sự chú ý, chúng ta có thể viết lại :eq:`eq_nadaraya-watson` dưới dạng tổng quát hơn* chú ý cùng*:

$$f(x) = \sum_{i=1}^{n} \alpha(x, x_i) y_i, \tag{11.2.4}$$

trong đó x là truy vấn và (x_i, y_i) là cặp giá trị khóa-giá trị. So sánh (11.2.4) và (11.2.2), sự chú ý tập hợp ở đây là trung bình có trọng số của các giá trị y_i. Trọng lượng chú ý*$\alpha(x, x_i)$ trong (11.2.4) được gán cho giá trị tương ứng y_i dựa trên sự tương tác giữa truy vấn x và khóa x_i được mô hình hóa bởi α. Đối với bất kỳ truy vấn nào, trọng lượng chú ý của nó đối với tất cả các cặp khóa-giá trị là một phân phối xác suất hợp lệ: chúng không âm và tổng hợp lên đến một.

Để đạt được trực giác của sự chú ý, chỉ cần xem xét một * Gaussian kernel* được định nghĩa là

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp(-\frac{u^2}{2}). \tag{11.2.5}$$

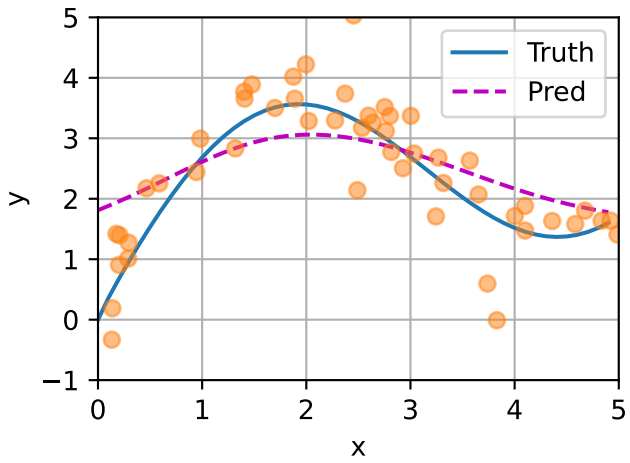
Cắm hạt nhân Gaussian vào (11.2.4) và (11.2.3) cho

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}(x - x_i)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}(x - x_j)^2\right)} y_i \\ &= \sum_{i=1}^{n} \text{softmax}\left(-\frac{1}{2}(x - x_i)^2\right) y_i. \end{aligned} \tag{11.2.6}$$

Trong (11.2.6), một khóa x_i gần với truy vấn đã cho x sẽ nhận được *chú ý hơn* thông qua trọng lượng chú ý * lớn hơn* được gán cho giá trị tương ứng của khóa y_i.

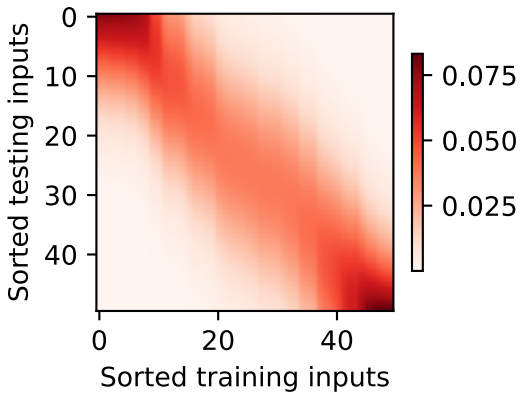
Đáng chú ý, hồi quy hạt nhân Nadaraya-Watson là một mô hình không tham số; do đó (11.2.6) là một ví dụ về *không tham số attention pooling*. Sau đây, chúng tôi vẽ dự đoán dựa trên mô hình chú ý không tham số này. Đường dự đoán là trơn tru và gần với sự thật mặt đất hơn so với sự thật được tạo ra bởi tổng hợp trung bình.

```
# Shape of `X_repeat`: (`n_test`, `n_train`), where each row contains the
# same testing inputs (i.e., same queries)
X_repeat = x_test.repeat_interleave(n_train).reshape((-1, n_train))
# Note that `x_train` contains the keys. Shape of `attention_weights`:
# (`n_test`, `n_train`), where each row contains attention weights to be
# assigned among the values (`y_train`) given each query
attention_weights = nn.functional.softmax(-(X_repeat - x_train)**2 / 2, dim=1)
# Each element of `y_hat` is weighted average of values, where weights are
# attention weights
y_hat = torch.matmul(attention_weights, y_train)
plot_kernel_reg(y_hat)
```



Bây giờ chúng ta hãy nhìn vào trọng lượng chú ý. Ở đây thử nghiệm đầu vào là các truy vấn trong khi đầu vào đào tạo là chìa khóa. Vì cả hai đầu vào được sắp xếp, chúng ta có thể thấy rằng cặp khóa truy vấn càng gần, trọng lượng chú ý cao hơn nằm trong sự chú ý.

```
d2l.show_heatmaps(attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



11.2.4 ** Tham số chú ý Pooling**

Hồi quy hạt nhân Nadaraya-Watson không tham số được hưởng lợi ích *nhất quán *: cung cấp đủ dữ liệu mô hình này hội tụ với giải pháp tối ưu. Tuy nhiên, chúng ta có thể dễ dàng tích hợp các thông số có thể học được vào tập hợp sự chú ý.

Ví dụ, hơi khác so với (11.2.6), trong khoảng cách sau giữa truy vấn x và khóa x_i được nhân với một tham số có thể học được w:

$$\begin{aligned} f(x) &= \sum_{i=1}^{n} \alpha(x, x_i) y_i \\ &= \sum_{i=1}^{n} \frac{\exp\left(-\frac{1}{2}((x - x_i)w)^2\right)}{\sum_{j=1}^{n} \exp\left(-\frac{1}{2}((x - x_j)w)^2\right)} y_i \\ &= \sum_{i=1}^{n} \text{softmax}\left(-\frac{1}{2}((x - x_i)w)^2\right) y_i. \end{aligned} \tag{11.2.7}$$

Trong phần còn lại của phần này, chúng tôi sẽ đào tạo mô hình này bằng cách học tham số của sự chú ý trong (11.2.7).

Phép nhân ma trận hàng loạt

Để tính toán sự chú ý hiệu quả hơn cho các minibatches, chúng ta có thể tận dụng các tiện ích nhân ma trận hàng loạt được cung cấp bởi các framework deep learning.

Giả sử rằng minibatch đầu tiên chứa n ma trận $\mathbf{X}_1, \ldots, \mathbf{X}_n$ của hình dạng $a \times b$, và minibatch thứ hai chứa n ma trận $\mathbf{Y}_1, \ldots, \mathbf{Y}_n$ hình dạng $b \times c$. Phép nhân ma trận lô của chúng dẫn đến ma trận n $\mathbf{X}_1\mathbf{Y}_1, \ldots, \mathbf{X}_n\mathbf{Y}_n$ của hình dạng $a \times c$. Do đó, cho hai hàng chục hình dạng (n, a, b) và (n, b, c), hình dạng của sản lượng nhân ma trận lô của chúng là (n, a, c) .

```
X = torch.ones((2, 1, 4))
Y = torch.ones((2, 4, 6))
torch.bmm(X, Y).shape
```

```
torch.Size([2, 1, 6])
```

Trong bối cảnh của các cơ chế chú ý, chúng ta có thể sử dụng phép nhân ma trận minibatch để tính toán trung bình có trọng số của các giá trị trong một minibatch.

```
weights = torch.ones((2, 10)) * 0.1
values = torch.arange(20.0).reshape((2, 10))
torch.bmm(weights.unsqueeze(1), values.unsqueeze(-1))
```

```
tensor([[[ 4.5000]],

        [[14.5000]]])
```

Xác định mô hình

Sử dụng phép nhân ma trận minibatch, bên dưới chúng ta xác định phiên bản tham số của hồi quy hạt nhân Nadaraya-Watson dựa trên parametric attention pooling trong (11.2.7).

```
class NWKernelRegression(nn.Module):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.w = nn.Parameter(torch.rand((1,), requires_grad=True))

    def forward(self, queries, keys, values):
        # Shape of the output `queries` and `attention_weights`:
        # (no. of queries, no. of key-value pairs)
        queries = queries.repeat_interleave(keys.shape[1]).reshape((-1, keys.
↪shape[1]))
        self.attention_weights = nn.functional.softmax(
            -((queries - keys) * self.w)**2 / 2, dim=1)
        # Shape of `values`: (no. of queries, no. of key-value pairs)
        return torch.bmm(self.attention_weights.unsqueeze(1),
                         values.unsqueeze(-1)).reshape(-1)
```

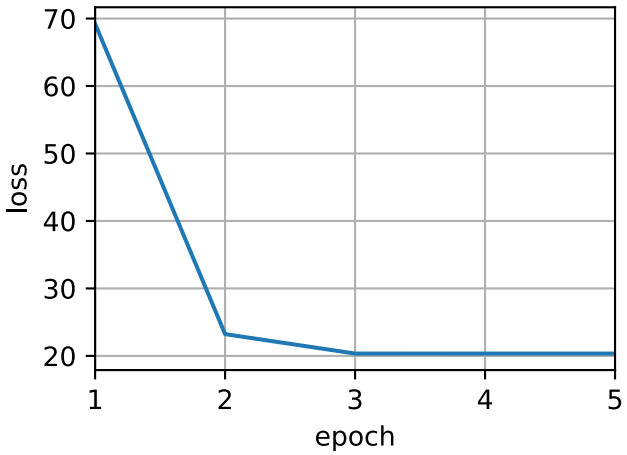
Đào tạo

Sau đây, chúng tôi chuyển đổi tập dữ liệu đào tạo thành các khóa và giá trị để đào tạo mô hình chú ý. Trong tập hợp sự chú ý tham số, bất kỳ đầu vào đào tạo nào lấy các cặp giá trị khóa từ tất cả các ví dụ đào tạo ngoại trừ chính nó để dự đoán đầu ra của nó.

```
# Shape of `X_tile`: (`n_train`, `n_train`), where each column contains the
# same training inputs
X_tile = x_train.repeat((n_train, 1))
# Shape of `Y_tile`: (`n_train`, `n_train`), where each column contains the
# same training outputs
Y_tile = y_train.repeat((n_train, 1))
# Shape of `keys`: ('n_train', 'n_train' - 1)
keys = X_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -
↪1))
# Shape of `values`: ('n_train', 'n_train' - 1)
values = Y_tile[(1 - torch.eye(n_train)).type(torch.bool)].reshape((n_train, -
↪1))
```

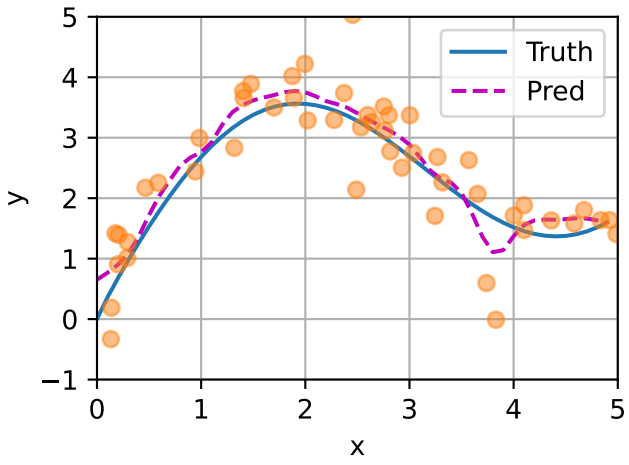
Sử dụng sự mất mát bình phương và gốc gradient ngẫu nhiên, chúng tôi đào tạo mô hình chú ý tham số.

```
net = NWKernelRegression()
loss = nn.MSELoss(reduction='none')
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
animator = d2l.Animator(xlabel='epoch', ylabel='loss', xlim=[1, 5])

for epoch in range(5):
    trainer.zero_grad()
    l = loss(net(x_train, keys, values), y_train)
    l.sum().backward()
    trainer.step()
    print(f'epoch {epoch + 1}, loss {float(l.sum()):.6f}')
    animator.add(epoch + 1, float(l.sum()))
```



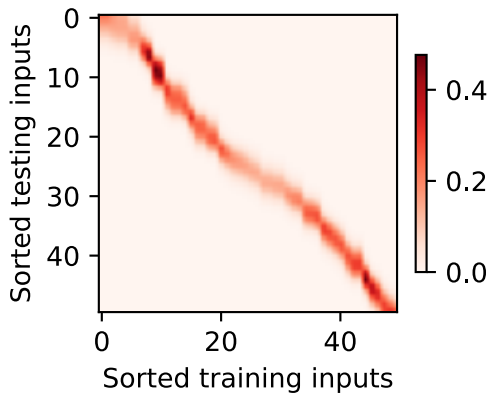
Sau khi đào tạo mô hình chú ý tham số, chúng ta có thể vẽ giá của nó. Cố gắng để phù hợp với tập dữ liệu đào tạo với tiếng ồn, dòng dự đoán là ít trơn tru hơn so với đối tác không tham số của nó đã được vẽ trước đó.

```
# Shape of `keys`: (`n_test`, `n_train`), where each column contains the same
# training inputs (i.e., same keys)
keys = x_train.repeat((n_test, 1))
# Shape of `value`: (`n_test`, `n_train`)
values = y_train.repeat((n_test, 1))
y_hat = net(x_test, keys, values).unsqueeze(1).detach()
plot_kernel_reg(y_hat)
```



So sánh với sự chú ý không tham số, khu vực có trọng lượng chú ý lớn trở nên sắc bén hơn trong cài đặt có thể học được và tham số.

```
d2l.show_heatmaps(net.attention_weights.unsqueeze(0).unsqueeze(0),
                  xlabel='Sorted training inputs',
                  ylabel='Sorted testing inputs')
```



11.2.5 Tóm tắt

- Hồi quy hạt nhân Nadaraya-Watson là một ví dụ về máy học với các cơ chế chú ý.
- Sự chú ý của hồi quy hạt nhân Nadaraya-Watson là một trung bình trọng số của các đầu ra đào tạo. Từ góc độ chú ý, trọng lượng chú ý được gán cho một giá trị dựa trên một hàm của truy vấn và khóa được ghép nối với giá trị.
- Chú ý pooling có thể là một trong hai không tham số hoặc tham số.

11.2.6 Bài tập

1. Tăng số lượng các ví dụ đào tạo. Bạn có thể học hồi quy hạt nhân Nadaraya-Watson không tham số tốt hơn?
2. Giá trị của w đã học được của chúng tôi trong thí nghiệm tập hợp chú ý tham số là gì? Tại sao nó làm cho vùng có trọng số sắc nét hơn khi hình dung trọng lượng chú ý?
3. Làm thế nào chúng ta có thể thêm các siêu tham số vào hồi quy hạt nhân Nadaraya-Watson không tham số để dự đoán tốt hơn?
4. Thiết kế một tập hợp chú ý tham số khác cho hồi quy hạt nhân của phần này. Đào tạo mô hình mới này và hình dung trọng lượng chú ý của nó.

Discussions[119]

11.3 Chức năng chấm điểm chú ý

Trong Section 11.2, chúng tôi đã sử dụng một hạt nhân Gaussian để mô hình hóa tương tác giữa các truy vấn và khóa. Xử lý số mũ của hạt nhân Gaussian trong (11.2.6) như một chức năng ghi điểm *chú ý* (hoặc chức năng *ghi điểm*), kết quả của hàm này về cơ bản đã được đưa vào một hoạt động softmax. Kết quả là, chúng tôi thu được một phân phối xác suất (trọng lượng chú ý) so với các giá trị được ghép nối với các phím. Cuối cùng, đầu ra của sự chú ý tập hợp chỉ đơn giản là một tổng trọng số của các giá trị dựa trên các trọng lượng chú ý này.

Ở cấp độ cao, chúng ta có thể sử dụng thuật toán trên để khởi tạo khuôn khổ của các cơ chế chú ý trong Fig. 11.1.3. Biểu thị một chức năng chấm điểm chú ý bởi a, Fig. 11.3.1 minh họa cách đầu ra của sự chú ý pooling có thể được tính như một tổng trọng số của các giá trị. Vì trọng lượng chú ý là một phân phối xác suất, tổng trọng số về cơ bản là một trung bình có trọng số.

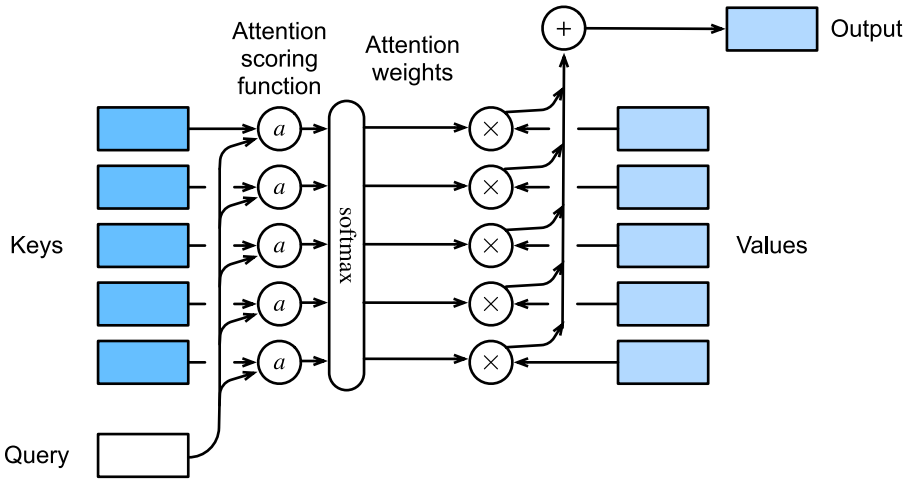


Fig. 11.3.1: Computing the output of attention pooling as a weighted average of values.

Về mặt toán học, giả sử rằng chúng ta có một truy vấn $\mathbf{q} \in \mathbb{R}^q$ và m cặp giá trị khóa $(\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)$, trong đó bất kỳ $\mathbf{k}_i \in \mathbb{R}^k$ và $\mathbf{v}_i \in \mathbb{R}^v$ bất kỳ $\mathbf{v}_i \in \mathbb{R}^v$. Sự chú ý tập hợp f được khởi tạo như một tổng trọng số

[119] https://discuss.d2l.ai/t/1599

của các giá trị:

$$f(\mathbf{q}, (\mathbf{k}_1, \mathbf{v}_1), \ldots, (\mathbf{k}_m, \mathbf{v}_m)) = \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i \in \mathbb{R}^v, \tag{11.3.1}$$

trong đó trọng lượng chú ý (vô hướng) cho truy vấn $\mathbf{q}$ và khóa $\mathbf{k}_i$ được tính bằng hoạt động softmax của một chức năng ghi điểm chú ý a ánh xạ hai vectơ đến vô hướng:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_{j=1}^{m} \exp(a(\mathbf{q}, \mathbf{k}_j))} \in \mathbb{R}. \tag{11.3.2}$$

Như chúng ta có thể thấy, các lựa chọn khác nhau của chức năng ghi điểm chú ý a dẫn đến các hành vi khác nhau của sự chú ý chung. Trong phần này, chúng tôi giới thiệu hai chức năng tính điểm phổ biến mà chúng tôi sẽ sử dụng để phát triển các cơ chế chú ý phức tạp hơn sau này.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

11.3.1 Masked Softmax Operation

Như chúng tôi vừa đề cập, một hoạt động softmax được sử dụng để tạo ra một phân phối xác suất như trọng lượng chú ý. Trong một số trường hợp, không phải tất cả các giá trị nên được đưa vào tập hợp chú ý. Ví dụ, để xử lý minibatch hiệu quả trong Section 10.5, một số chuỗi văn bản được đệm với các mã thông báo đặc biệt không mang ý nghĩa. Để thu hút sự chú ý chỉ có các token có ý nghĩa làm giá trị, chúng ta có thể chỉ định độ dài chuỗi hợp lệ (về số lượng token) để lọc ra những mã thông báo vượt quá phạm vi quy định này khi tính toán softmax. Bằng cách này, chúng ta có thể thực hiện một operation*masked softmax* như vậy trong hàm `masked_softmax` sau, trong đó bất kỳ giá trị nào vượt quá độ dài hợp lệ được che dấu là zero.

```
#@save
def masked_softmax(X, valid_lens):
    """Perform softmax operation by masking elements on the last axis."""
    # `X`: 3D tensor, `valid_lens`: 1D or 2D tensor
    if valid_lens is None:
        return nn.functional.softmax(X, dim=-1)
    else:
        shape = X.shape
        if valid_lens.dim() == 1:
            valid_lens = torch.repeat_interleave(valid_lens, shape[1])
        else:
            valid_lens = valid_lens.reshape(-1)
        # On the last axis, replace masked elements with a very large negative
        # value, whose exponentiation outputs 0
        X = d2l.sequence_mask(X.reshape(-1, shape[-1]), valid_lens,
                              value=-1e6)
        return nn.functional.softmax(X.reshape(shape), dim=-1)
```

Để chứng minh cách thức hoạt động của hàm này, hãy xem xét một minibatch gồm hai ví dụ ma trận 2×4, trong đó độ dài hợp lệ cho hai ví dụ này là hai và ba, tương ứng. Kết quả của hoạt động softmax đeo mặt nạ, các giá trị vượt quá độ dài hợp lệ đều được che dấu là 0.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([2, 3]))
```

```
tensor([[[0.3713, 0.6287, 0.0000, 0.0000],
         [0.3659, 0.6341, 0.0000, 0.0000]],

        [[0.2812, 0.2965, 0.4223, 0.0000],
         [0.4899, 0.2254, 0.2847, 0.0000]]])
```

Tương tự, chúng ta cũng có thể sử dụng tensor hai chiều để chỉ định độ dài hợp lệ cho mỗi hàng trong mỗi ví dụ ma trận.

```
masked_softmax(torch.rand(2, 2, 4), torch.tensor([[1, 3], [2, 4]]))
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],
         [0.3866, 0.3205, 0.2930, 0.0000]],

        [[0.4323, 0.5677, 0.0000, 0.0000],
         [0.2959, 0.1760, 0.3077, 0.2204]]])
```

11.3.2 Phụ gia chú ý

Nói chung, khi truy vấn và phím là vectơ có độ dài khác nhau, chúng ta có thể sử dụng sự chú ý phụ gia làm chức năng chấm điểm. Đưa ra một truy vấn $\mathbf{q} \in \mathbb{R}^q$ và một khóa $\mathbf{k} \in \mathbb{R}^k$, chức năng chấm điểm * phụ thuật*

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R}, \tag{11.3.3}$$

trong đó các thông số có thể học được $\mathbf{W}_q \in \mathbb{R}^{h\times q}$, $\mathbf{W}_k \in \mathbb{R}^{h\times k}$ và $\mathbf{w}_v \in \mathbb{R}^h$. Tương đương với (11.3.3), truy vấn và khóa được nối và đưa vào một MLP với một lớp ẩn duy nhất có số đơn vị ẩn là h, một siêu tham số. Bằng cách sử dụng tanh làm chức năng kích hoạt và vô hiệu hóa các thuật ngữ thiên vị, chúng tôi thực hiện sự chú ý phụ gia trong những điều sau đây.

```
#@save
class AdditiveAttention(nn.Module):
    """Additive attention."""
    def __init__(self, key_size, query_size, num_hiddens, dropout, **kwargs):
        super(AdditiveAttention, self).__init__(**kwargs)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=False)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=False)
        self.w_v = nn.Linear(num_hiddens, 1, bias=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, queries, keys, values, valid_lens):
        queries, keys = self.W_q(queries), self.W_k(keys)
        # After dimension expansion, shape of `queries`: (`batch_size`, no. of
        # queries, 1, `num_hiddens`) and shape of `keys`: (`batch_size`, 1,
        # no. of key-value pairs, `num_hiddens`). Sum them up with
        # broadcasting
        features = queries.unsqueeze(2) + keys.unsqueeze(1)
        features = torch.tanh(features)
        # There is only one output of `self.w_v`, so we remove the last
        # one-dimensional entry from the shape. Shape of `scores`:
```

(continues on next page)

(continued from previous page)

```
        # (`batch_size`, no. of queries, no. of key-value pairs)
        scores = self.w_v(features).squeeze(-1)
        self.attention_weights = masked_softmax(scores, valid_lens)
        # Shape of `values`: (`batch_size`, no. of key-value pairs, value
        # dimension)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

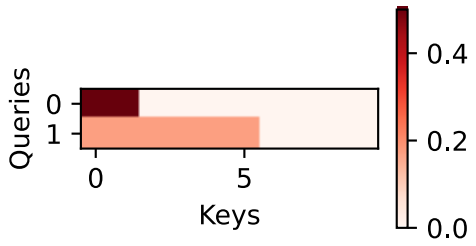
Hãy để chúng tôi chứng minh lớp `AdditiveAttention` ở trên với một ví dụ đồ chơi, trong đó các hình dạng (kích thước lô, số bước hoặc độ dài chuỗi trong thẻ, kích thước tính năng) của truy vấn, khóa và giá trị là (2, 1, 20), (2, 2), và (2), (2, 10, 4), tương ứng. Đầu ra tập hợp chú ý có hình dạng (kích thước lô, số bước cho truy vấn, kích thước tính năng cho các giá trị).

```
queries, keys = torch.normal(0, 1, (2, 1, 20)), torch.ones((2, 10, 2))
# The two value matrices in the `values` minibatch are identical
values = torch.arange(40, dtype=torch.float32).reshape(1, 10, 4).repeat(
    2, 1, 1)
valid_lens = torch.tensor([2, 6])

attention = AdditiveAttention(key_size=2, query_size=20, num_hiddens=8,
                              dropout=0.1)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]], grad_fn=<BmmBackward0>)
```

Mặc dù sự chú ý phụ gia chứa các tham số có thể học được, vì mỗi khóa đều giống nhau trong ví dụ này, trọng lượng chú ý đều đồng nhất, được xác định bởi độ dài hợp lệ được chỉ định.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



11.3.3 Quy mô Dot-Product Attention

Một thiết kế hiệu quả tính toán hơn cho chức năng tính điểm có thể chỉ đơn giản là chấm sản phẩm. Tuy nhiên, hoạt động của sản phẩm chấm đòi hỏi cả truy vấn và khóa đều có cùng độ dài vector, giả sử d. Giả sử rằng tất cả các yếu tố của truy vấn và khóa là các biến ngẫu nhiên độc lập với không trung bình và phương sai đơn vị. Tích chấm của cả hai vectơ có trung bình bằng 0 và phương sai là d. Để đảm bảo rằng phương sai của sản phẩm chấm vẫn là một bất kể chiều dài vectơ, chức năng chấm điểm chú ý * thu nhỏ sản phẩm điểm

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}/\sqrt{d} \tag{11.3.4}$$

chia sản phẩm chấm cho $\sqrt{d}$. Trong thực tế, chúng ta thường nghĩ về minibatches cho hiệu quả, chẳng hạn như tính toán sự chú ý cho n truy vấn và m cặp giá trị khóa, trong đó các truy vấn và khóa có chiều dài d và các giá trị có chiều dài v. Sự chú ý của sản phẩm điểm được chia tỷ lệ của truy vấn $\mathbf{Q} \in \mathbb{R}^{n\times d}$, phím $\mathbf{K} \in \mathbb{R}^{m\times d}$ và các giá trị $\mathbf{V} \in \mathbb{R}^{m\times v}$ là

$$\text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}. \tag{11.3.5}$$

Trong việc thực hiện sau đây của sự chú ý của sản phẩm chấm thu nhỏ, chúng tôi sử dụng dropout để điều chỉnh mô hình.

```
#@save
class DotProductAttention(nn.Module):
    """Scaled dot product attention."""
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # Shape of `queries`: (`batch_size`, no. of queries, `d`)
    # Shape of `keys`: (`batch_size`, no. of key-value pairs, `d`)
    # Shape of `values`: (`batch_size`, no. of key-value pairs, value
    # dimension)
    # Shape of `valid_lens`: (`batch_size`,) or (`batch_size`, no. of queries)
    def forward(self, queries, keys, values, valid_lens=None):
        d = queries.shape[-1]
        # Set `transpose_b=True` to swap the last two dimensions of `keys`
        scores = torch.bmm(queries, keys.transpose(1,2)) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, valid_lens)
        return torch.bmm(self.dropout(self.attention_weights), values)
```

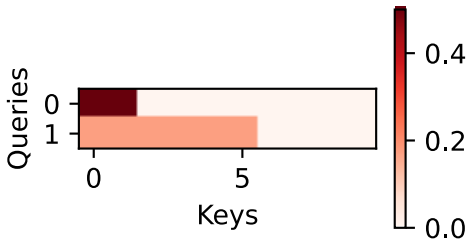
Để chứng minh lớp `DotProductAttention` ở trên, chúng tôi sử dụng cùng một phím, giá trị và độ dài hợp lệ từ ví dụ đồ chơi trước đó để chú ý phụ gia. Đối với hoạt động sản phẩm chấm, chúng tôi làm cho kích thước tính năng của các truy vấn giống như kích thước của các phím.

```
queries = torch.normal(0, 1, (2, 1, 2))
attention = DotProductAttention(dropout=0.5)
attention.eval()
attention(queries, keys, values, valid_lens)
```

```
tensor([[[ 2.0000,  3.0000,  4.0000,  5.0000]],

        [[10.0000, 11.0000, 12.0000, 13.0000]]])
```

Giống như trong trình diễn chú ý phụ gia, vì `keys` chứa cùng một yếu tố không thể phân biệt bằng bất kỳ truy vấn nào, trọng lượng chú ý thống nhất thu được.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Queries')
```



11.3.4 Tóm tắt

- Chúng ta có thể tính toán đầu ra của sự chú ý tập hợp như một trung bình trọng số của các giá trị, trong đó các lựa chọn khác nhau của chức năng chấm điểm chú ý dẫn đến các hành vi khác nhau của sự chú ý.
- Khi truy vấn và phím là vectơ có độ dài khác nhau, chúng ta có thể sử dụng chức năng chấm điểm chú ý phụ gia. Khi chúng giống nhau, chức năng ghi điểm chú ý dot-sản phẩm thu nhỏ sẽ hiệu quả hơn về mặt tính toán.

11.3.5 Bài tập

1. Sửa đổi các phím trong ví dụ đồ chơi và hình dung trọng lượng chú ý. Do phụ gia chú ý và thu nhỏ sự chú ý của sản phẩm điểm vẫn xuất ra cùng một trọng lượng chú ý? Tại sao hoặc tại sao không?
2. Chỉ sử dụng phép nhân ma trận, bạn có thể thiết kế một chức năng tính điểm mới cho các truy vấn và các phím có độ dài vector khác nhau không?
3. Khi truy vấn và phím có cùng chiều dài vector, tổng kết vectơ có thiết kế tốt hơn so với sản phẩm chấm cho chức năng chấm điểm không? Tại sao hoặc tại sao không?

Discussions[120]

11.4 Bahdanau Chú ý

Chúng tôi đã nghiên cứu vấn đề dịch máy trong Section 10.7, nơi chúng tôi thiết kế một kiến trúc bộ mã hóa-giải mã dựa trên hai RNNs cho trình tự để học trình tự. Cụ thể, bộ mã hóa RNN biến một chuỗi có độ dài biến đổi thành một biến ngữ cảnh hình dạng cố định, sau đó bộ giải mã RNN tạo ra mã thông báo chuỗi đầu ra (mục tiêu) theo mã thông báo dựa trên các token được tạo ra và biến ngữ cảnh. Tuy nhiên, mặc dù không phải tất cả các mã thông báo đầu vào (nguồn) đều hữu ích cho việc giải mã một mã thông báo nhất định, biến ngữ cảnh *same* mã hóa toàn bộ chuỗi đầu vào vẫn được sử dụng ở mỗi bước giải mã.

Trong một thách thức riêng biệt nhưng liên quan về thế hệ chữ viết tay cho một chuỗi văn bản nhất định, Graves đã thiết kế một mô hình chú ý khác biệt để căn chỉnh các ký tự văn bản với dấu vết bút dài hơn nhiều, trong đó căn chỉnh chỉ di chuyển theo một hướng [Graves.2013]. Lấy cảm hứng từ ý tưởng học cách căn chỉnh, Bahdanau et al. đề xuất một mô hình chú ý khác biệt mà không có giới hạn liên kết một chiều nghiêm trọng (Bahdanau et al., 2014). Khi dự đoán một mã thông báo, nếu không phải tất cả các token đầu vào đều có liên

[120] https://discuss.d2l.ai/t/1064

quan, mô hình sẽ căn chỉnh (hoặc tham dự) chỉ với các phần của chuỗi đầu vào có liên quan đến dự đoán hiện tại. Điều này đạt được bằng cách coi biến ngữ cảnh như một đầu ra của sự chú ý pooling.

11.4.1 Mô hình

Khi mô tả sự chú ý của Bahdanau cho bộ giải mã RNN bên dưới, chúng tôi sẽ làm theo cùng một ký hiệu trong Section 10.7. Mô hình dựa trên sự chú ý mới giống như trong Section 10.7 ngoại trừ biến ngữ cảnh $\mathbf{c}$ trong (10.7.3) được thay thế bằng $\mathbf{c}_{t'}$ tại bất kỳ bước thời gian giải mã nào t'. Giả sử có T token trong chuỗi đầu vào, biến ngữ cảnh tại bước thời gian giải mã t' là đầu ra của sự chú ý pooling:

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{s}_{t'-1}, \mathbf{h}_t)\mathbf{h}_t, \tag{11.4.1}$$

trong đó bộ giải mã ẩn trạng thái $\mathbf{s}_{t'-1}$ tại thời điểm bước $t'-1$ là truy vấn và các trạng thái ẩn mã hóa $\mathbf{h}_t$ là cả các phím và giá trị, và trọng lượng chú ý α được tính như trong (11.3.2) bằng cách sử dụng chức năng chấm điểm chú ý phụ gia được xác định bởi (11.3.3).

Hơi khác so với kiến trúc mã hóa giải mã RNN vani trong Fig. 10.7.2, kiến trúc tương tự với sự chú ý Bahdanau được mô tả trong Fig. 11.4.1.

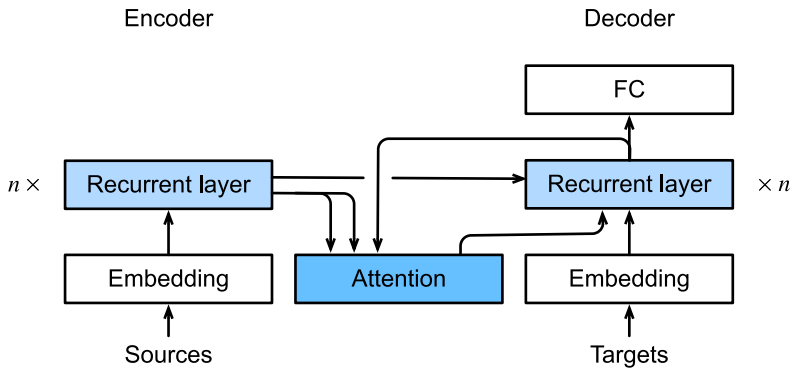


Fig. 11.4.1: Layers in an RNN encoder-decoder model with Bahdanau attention.

```
import torch
from torch import nn
from d2l import torch as d2l
```

11.4.2 Xác định bộ giải mã với sự chú ý

Để thực hiện bộ giải mã mã RNN với sự chú ý của Bahdanau, chúng ta chỉ cần xác định lại bộ giải mã. Để hình dung các trọng lượng chú ý đã học được thuận tiện hơn, lớp `AttentionDecoder` sau định nghĩa giao diện cơ bản cho bộ giải mã với cơ chế chú ý.

```
#@save
class AttentionDecoder(d2l.Decoder):
    """The base attention-based decoder interface."""
    def __init__(self, **kwargs):
        super(AttentionDecoder, self).__init__(**kwargs)
```

(continues on next page)

(continued from previous page)

```
    @property
    def attention_weights(self):
        raise NotImplementedError
```

Bây giờ chúng ta hãy triển khai bộ giải mã RNN với sự chú ý Bahdanau trong lớp `Seq2SeqAttentionDecoder` sau. Trạng thái của bộ giải mã được khởi tạo với (i) các trạng thái ẩn lớp cuối của bộ mã hóa ở tất cả các bước thời gian (như các phím và giá trị của sự chú ý); (ii) bộ mã hóa tất cả các lớp ẩn trạng thái ở bước thời gian cuối cùng (để khởi tạo trạng thái ẩn của bộ giải mã); và (iii) bộ mã hóa độ dài hợp lệ (để loại trừ các thẻ đệm trong tập hợp chú ý). Tại mỗi bước thời gian giải mã, trạng thái ẩn lớp cuối của bộ giải mã ở bước thời gian trước đó được sử dụng làm truy vấn của sự chú ý. Kết quả là, cả đầu ra chú ý và nhúng đầu vào được nối làm đầu vào của bộ giải mã RNN.

```
class Seq2SeqAttentionDecoder(AttentionDecoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)
        self.attention = d2l.AdditiveAttention(
            num_hiddens, num_hiddens, num_hiddens, dropout)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = nn.GRU(
            embed_size + num_hiddens, num_hiddens, num_layers,
            dropout=dropout)
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        # Shape of `outputs`: (`num_steps`, `batch_size`, `num_hiddens`).
        # Shape of `hidden_state[0]`: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        outputs, hidden_state = enc_outputs
        return (outputs.permute(1, 0, 2), hidden_state, enc_valid_lens)

    def forward(self, X, state):
        # Shape of `enc_outputs`: (`batch_size`, `num_steps`, `num_hiddens`).
        # Shape of `hidden_state[0]`: (`num_layers`, `batch_size`,
        # `num_hiddens`)
        enc_outputs, hidden_state, enc_valid_lens = state
        # Shape of the output `X`: (`num_steps`, `batch_size`, `embed_size`)
        X = self.embedding(X).permute(1, 0, 2)
        outputs, self._attention_weights = [], []
        for x in X:
            # Shape of `query`: (`batch_size`, 1, `num_hiddens`)
            query = torch.unsqueeze(hidden_state[-1], dim=1)
            # Shape of `context`: (`batch_size`, 1, `num_hiddens`)
            context = self.attention(
                query, enc_outputs, enc_outputs, enc_valid_lens)
            # Concatenate on the feature dimension
            x = torch.cat((context, torch.unsqueeze(x, dim=1)), dim=-1)
            # Reshape `x` as (1, `batch_size`, `embed_size` + `num_hiddens`)
            out, hidden_state = self.rnn(x.permute(1, 0, 2), hidden_state)
            outputs.append(out)
            self._attention_weights.append(self.attention.attention_weights)
        # After fully-connected layer transformation, shape of `outputs`:
        # (`num_steps`, `batch_size`, `vocab_size`)
```

(continues on next page)

(continued from previous page)

```
        outputs = self.dense(torch.cat(outputs, dim=0))
        return outputs.permute(1, 0, 2), [enc_outputs, hidden_state,
                                          enc_valid_lens]

    @property
    def attention_weights(self):
        return self._attention_weights
```

Sau đây, chúng tôi test the implemented decoder với Bahdanau chú ý sử dụng một minibatch gồm 4 chuỗi đầu vào của 7 bước thời gian.

```
encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                             num_layers=2)
encoder.eval()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8, num_hiddens=16,
                                  num_layers=2)
decoder.eval()
X = torch.zeros((4, 7), dtype=torch.long)  # (`batch_size`, `num_steps`)
state = decoder.init_state(encoder(X), None)
output, state = decoder(X, state)
output.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape
```

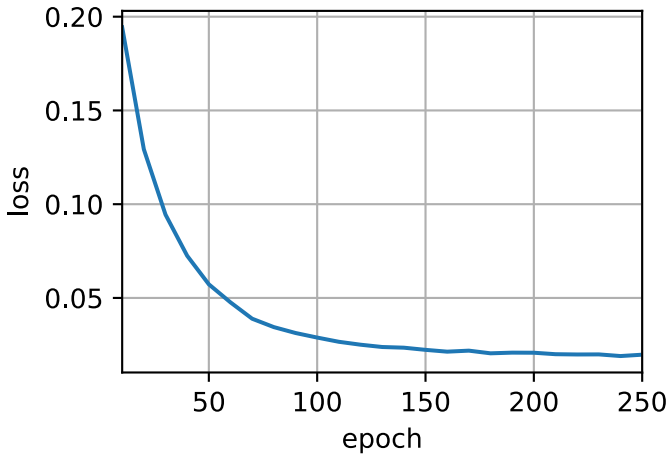
```
(torch.Size([4, 7, 10]), 3, torch.Size([4, 7, 16]), 2, torch.Size([4, 16]))
```

11.4.3 Đào tạo

Tương tự như Section 10.7.4, ở đây chúng tôi chỉ định hyperparemeters, khởi tạo bộ mã hóa và bộ giải mã với sự chú ý của Bahdanau và đào tạo mô hình này để dịch máy. Do cơ chế chú ý mới được thêm vào, việc đào tạo này chậm hơn nhiều so với năm Section 10.7.4 mà không có cơ chế chú ý.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.1
batch_size, num_steps = 64, 10
lr, num_epochs, device = 0.005, 250, d2l.try_gpu()

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = d2l.Seq2SeqEncoder(
    len(src_vocab), embed_size, num_hiddens, num_layers, dropout)
decoder = Seq2SeqAttentionDecoder(
    len(tgt_vocab), embed_size, num_hiddens, num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.020, 5580.3 tokens/sec on cuda:0
```



Sau khi mô hình được đào tạo, chúng tôi sử dụng nó để dịch một vài câu tiếng Anh sang tiếng Pháp và tính điểm BLEU của họ.

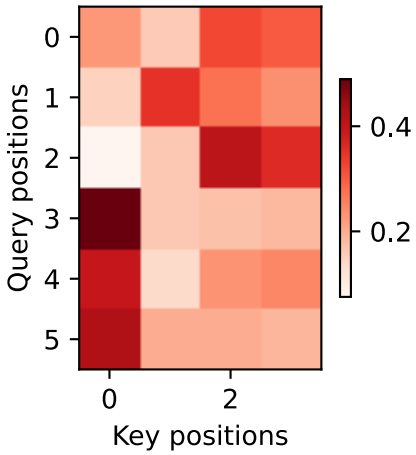
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est riche .,  bleu 0.658
i'm home . => je suis chez moi .,  bleu 1.000
```

```
attention_weights = torch.cat([step[0][0][0] for step in dec_attention_weight_
↪seq], 0).reshape((
    1, 1, -1, num_steps))
```

Bằng cách visualizing the attention weights khi dịch câu tiếng Anh cuối cùng, chúng ta có thể thấy rằng mỗi truy vấn gán trọng lượng không thống nhất trên các cặp key-value. Nó cho thấy ở mỗi bước giải mã, các phần khác nhau của chuỗi đầu vào được tổng hợp một cách chọn lọc trong tập hợp sự chú ý.

```
# Plus one to include the end-of-sequence token
d2l.show_heatmaps(
    attention_weights[:, :, :, :len(engs[-1].split()) + 1].cpu(),
    xlabel='Key positions', ylabel='Query positions')
```



11.4.4 Tóm tắt

- Khi dự đoán một mã thông báo, nếu không phải tất cả các mã thông báo đầu vào đều có liên quan, bộ giải mã RNN với sự chú ý Bahdanau tập hợp chọn lọc các phần khác nhau của chuỗi đầu vào. Điều này đạt được bằng cách coi biến ngữ cảnh như một đầu ra của sự chú ý phụ gia tập hợp.
- Trong bộ giải mã RNN, sự chú ý Bahdanau xử lý trạng thái ẩn bộ giải mã ở bước thời gian trước như truy vấn và các trạng thái ẩn mã hóa ở tất cả các bước thời gian như cả các phím và giá trị.

11.4.5 Bài tập

1. Thay thế GRU bằng LSTM trong thí nghiệm.
2. Sửa đổi thí nghiệm để thay thế chức năng chấm điểm sự chú ý phụ gia bằng sản phẩm điểm thu nhỏ. Làm thế nào để nó ảnh hưởng đến hiệu quả đào tạo?

Discussions[121]

11.5 Chú ý nhiều đầu

Trong thực tế, với cùng một tập hợp các truy vấn, khóa và giá trị, chúng tôi có thể muốn mô hình của chúng tôi kết hợp kiến thức từ các hành vi khác nhau của cùng một cơ chế chú ý, chẳng hạn như nắm bắt các phụ thuộc của các phạm vi khác nhau (ví dụ: phạm vi ngắn hơn so với phạm vi dài hơn) trong một chuỗi. Do đó, có thể có lợi khi cho phép cơ chế chú ý của chúng ta cùng sử dụng các không gian con đại diện khác nhau của các truy vấn, khóa và giá trị.

Để kết thúc này, thay vì thực hiện một tập hợp chú ý duy nhất, các truy vấn, khóa và giá trị có thể được chuyển đổi với h các phép chiếu tuyến tính học độc lập. Sau đó, các truy vấn, khóa và giá trị dự kiến h này được đưa vào sự chú ý chung song song. Cuối cùng, h đầu ra tập hợp chú ý được nối và chuyển đổi với một phép chiếu tuyến tính khác đã học để tạo ra đầu ra cuối cùng. Thiết kế này được gọi là * sự chú ý đa đầu*, trong đó mỗi đầu ra tập hợp chú ý h là đầu * (Vaswani et al., 2017). Sử dụng các lớp được kết nối hoàn toàn để thực hiện các biến đổi tuyến tính có thể học được, Fig. 11.5.1 mô tả sự chú ý nhiều đầu.

[121] https://discuss.d2l.ai/t/1065

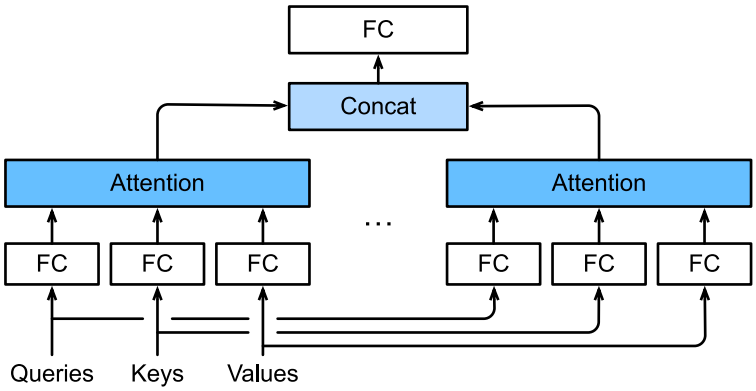


Fig. 11.5.1: Multi-head attention, where multiple heads are concatenated then linearly transformed.

11.5.1 Mô hình

Trước khi cung cấp việc thực hiện sự chú ý nhiều đầu, chúng ta hãy chính thức hóa mô hình này một cách toán học. Đưa ra một truy vấn $\mathbf{q} \in \mathbb{R}^{d_q}$, một khóa $\mathbf{k} \in \mathbb{R}^{d_k}$, và một giá trị $\mathbf{v} \in \mathbb{R}^{d_v}$, mỗi đầu chú ý $\mathbf{h}_i$ $(i = 1, \ldots, h)$ được tính là

$$\mathbf{h}_i = f(\mathbf{W}_i^{(q)}\mathbf{q}, \mathbf{W}_i^{(k)}\mathbf{k}, \mathbf{W}_i^{(v)}\mathbf{v}) \in \mathbb{R}^{p_v}, \tag{11.5.1}$$

trong đó các thông số có thể học được $\mathbf{W}_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$, $\mathbf{W}_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$ và $\mathbf{W}_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$, và f là sự chú ý tập hợp, chẳng hạn như sự chú ý của phụ gia và sự chú ý của sản phẩm điểm thu nhỏ trong Section 11.3. Đầu ra chú ý nhiều đầu là một chuyển đổi tuyến tính khác thông qua các tham số có thể học được $\mathbf{W}_o \in \mathbb{R}^{p_o \times hp_v}$ của nối h đầu:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{h}_1 \\ \vdots \\ \mathbf{h}_h \end{bmatrix} \in \mathbb{R}^{p_o}. \tag{11.5.2}$$

Dựa trên thiết kế này, mỗi đầu có thể tham dự các phần khác nhau của đầu vào. Các chức năng phức tạp hơn mức trung bình có trọng số đơn giản có thể được thể hiện.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

11.5.2 Thực hiện

Trong triển khai của chúng tôi, chúng tôi chọn sự chú ý của sản phẩm điểm thu nhỏ cho mỗi đầu của sự chú ý nhiều đầu. Để tránh tăng trưởng đáng kể chi phí tính toán và chi phí tham số hóa, chúng tôi đặt $p_q = p_k = p_v = p_o/h$. Lưu ý rằng các đầu h có thể được tính toán song song nếu chúng ta đặt số lượng đầu ra của các biến đổi tuyến tính cho truy vấn, khóa và giá trị thành $p_q h = p_k h = p_v h = p_o$. Trong việc triển khai sau đây, p_o được chỉ định thông qua đối số `num_hiddens`.

```
#@save
class MultiHeadAttention(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    """Multi-head attention."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, valid_lens):
        # Shape of `queries`, `keys`, or `values`:
        # (`batch_size`, no. of queries or key-value pairs, `num_hiddens`)
        # Shape of `valid_lens`:
        # (`batch_size`,) or (`batch_size`, no. of queries)
        # After transposing, shape of output `queries`, `keys`, or `values`:
        # (`batch_size` * `num_heads`, no. of queries or key-value pairs,
        # `num_hiddens` / `num_heads`)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            # On axis 0, copy the first item (scalar or vector) for
            # `num_heads` times, then copy the next item, and so on
            valid_lens = torch.repeat_interleave(
                valid_lens, repeats=self.num_heads, dim=0)

        # Shape of `output`: (`batch_size` * `num_heads`, no. of queries,
        # `num_hiddens` / `num_heads`)
        output = self.attention(queries, keys, values, valid_lens)

        # Shape of `output_concat`:
        # (`batch_size`, no. of queries, `num_hiddens`)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

Để cho phép tính toán song song của nhiều đầu, lớp `MultiHeadAttention` trên sử dụng hai hàm chuyển vị như định nghĩa bên dưới. Cụ thể, chức năng `transpose_output` đảo ngược hoạt động của hàm `transpose_qkv`.

```
#@save
def transpose_qkv(X, num_heads):
    """Transposition for parallel computation of multiple attention heads."""
    # Shape of input `X`:
    # (`batch_size`, no. of queries or key-value pairs, `num_hiddens`).
    # Shape of output `X`:
    # (`batch_size`, no. of queries or key-value pairs, `num_heads`,
    # `num_hiddens` / `num_heads`)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # Shape of output `X`:
    # (`batch_size`, `num_heads`, no. of queries or key-value pairs,
```

(continues on next page)

```
    # `num_hiddens` / `num_heads`)
    X = X.permute(0, 2, 1, 3)

    # Shape of `output`:
    # (`batch_size` * `num_heads`, no. of queries or key-value pairs,
    # `num_hiddens` / `num_heads`)
    return X.reshape(-1, X.shape[2], X.shape[3])


#@save
def transpose_output(X, num_heads):
    """Reverse the operation of `transpose_qkv`."""
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Hãy để chúng tôi test our implemented `MultiHeadAttention` class sử dụng một ví dụ đồ chơi trong đó các phím và giá trị giống nhau. Kết quả là, hình dạng của đầu ra chú ý nhiều đầu là (`batch_size`, `num_queries`, `num_hiddens`).

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                               num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, num_kvpairs, valid_lens = 2, 4, 6, torch.tensor([3,␣
↪2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))
attention(X, Y, Y, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

11.5.3 Tóm tắt

- Sự chú ý nhiều đầu kết hợp kiến thức về cùng một tập hợp sự chú ý thông qua các không gian con đại diện khác nhau của các truy vấn, khóa và giá trị.
- Để tính toán nhiều đầu của sự chú ý nhiều đầu song song, cần phải thao tác tensor thích hợp.

11.5.4 Bài tập

1. Hình dung trọng lượng chú ý của nhiều đầu trong thí nghiệm này.
2. Giả sử rằng chúng ta có một mô hình được đào tạo dựa trên sự chú ý nhiều đầu và chúng ta muốn cắt tỉa những đầu chú ý ít quan trọng nhất để tăng tốc độ dự đoán. Làm thế nào chúng ta có thể thiết kế các thí nghiệm để đo lường tầm quan trọng của một đầu chú ý?

Discussions[122]

11.6 Tự ghi nhận và mã hóa vị trí

In deep learning, we often use CNNs or RNNs to encode a sequence. Now with attention mechanisms, imagine that we feed a sequence of tokens into attention pooling so that the same set of tokens act as queries, keys, and values. Specifically, each query attends to all the key-value pairs and generates one attention output. Since the queries, keys, and values come from the same place, this performs In this section, we will discuss sequence encoding using self-attention, including using additional information for the sequence order.

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

11.6.1 Self-Attention

Given a sequence of input tokens $\mathbf{x}_1, \ldots, \mathbf{x}_n$ where any $\mathbf{x}_i \in \mathbb{R}^d$ $(1 \leq i \leq n)$, its self-attention outputs a sequence of the same length $\mathbf{y}_1, \ldots, \mathbf{y}_n$, where

$$\mathbf{y}_i = f(\mathbf{x}_i, (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)) \in \mathbb{R}^d \tag{11.6.1}$$

according to the definition of attention pooling f in (11.2.4). Using multi-head attention, the following code snippet computes the self-attention of a tensor with shape (batch size, number of time steps or sequence length in tokens, d). The output tensor has the same shape.

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
                                   num_hiddens, num_heads, 0.5)
attention.eval()
```

[122] https://discuss.d2l.ai/t/1635

```
MultiHeadAttention(
  (attention): DotProductAttention(
    (dropout): Dropout(p=0.5, inplace=False)
  )
  (W_q): Linear(in_features=100, out_features=100, bias=False)
  (W_k): Linear(in_features=100, out_features=100, bias=False)
  (W_v): Linear(in_features=100, out_features=100, bias=False)
  (W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
attention(X, X, X, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

11.6.2 So sánh CNN s, RNNs, và Self-Attention

Chúng ta hãy so sánh các kiến trúc để lập bản đồ một chuỗi các mã thông báo n với một chuỗi khác có độ dài bằng nhau, trong đó mỗi mã thông báo đầu vào hoặc đầu ra được thể hiện bằng một vector d chiều. Cụ thể, chúng tôi sẽ xem xét CNN, RNN và sự tự chú ý. Chúng tôi sẽ so sánh độ phức tạp tính toán, các hoạt động tuần tự và độ dài đường dẫn tối đa của chúng. Lưu ý rằng các phép toán tuần tự ngăn chặn tính toán song song, trong khi một đường dẫn ngắn hơn giữa bất kỳ tổ hợp các vị trí trình tự làm cho việc tìm hiểu các phụ thuộc tầm xa trong chuỗi [Hochreiter.Bengio.Frasconi.ea.2001] dễ dàng hơn.

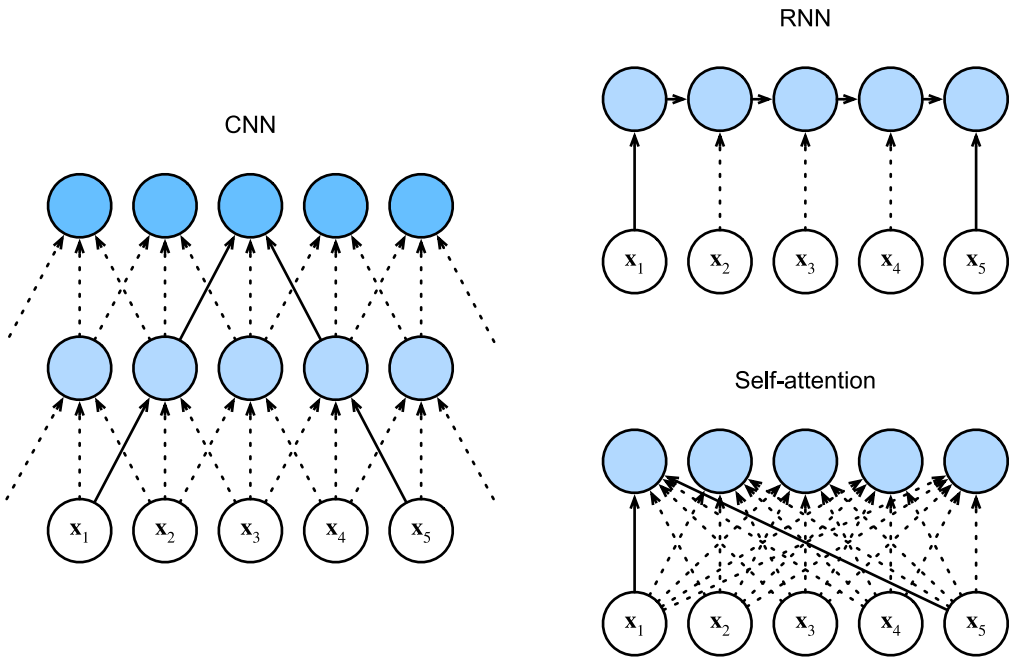


Fig. 11.6.1: Comparing CNN (padding tokens are omitted), RNN, and self-attention architectures.

Hãy xem xét một lớp phức tạp có kích thước hạt nhân là k. Chúng tôi sẽ cung cấp thêm chi tiết về xử lý trình tự sử dụng CNN trong các chương sau. Hiện tại, chúng ta chỉ cần biết rằng vì độ dài chuỗi là n, số lượng kênh

đầu vào và đầu ra đều là d, độ phức tạp tính toán của lớp tích hợp là $\mathcal{O}(knd^2)$. Như Fig. 11.6.1 cho thấy, CNN có thứ bậc nên có $\mathcal{O}(1)$ phép toán tuần tự và độ dài đường dẫn tối đa là $\mathcal{O}(n/k)$. Ví dụ, $\mathbf{x}_1$ và $\mathbf{x}_5$ nằm trong trường tiếp nhận của CNN hai lớp với kích thước hạt nhân 3 trong Fig. 11.6.1.

Khi cập nhật trạng thái ẩn của RNNs, phép nhân của ma trận trọng lượng $d \times d$ và trạng thái ẩn d chiều có độ phức tạp tính toán là $\mathcal{O}(d^2)$. Do độ dài trình tự là n, độ phức tạp tính toán của lớp tái phát là $\mathcal{O}(nd^2)$. Theo Fig. 11.6.1, có $\mathcal{O}(n)$ phép toán tuần tự không thể song song và độ dài đường tối đa cũng là $\mathcal{O}(n)$.

Trong sự tự chú ý, các truy vấn, khóa và giá trị đều là $n \times d$ ma trận. Hãy xem xét sự chú ý của sản phẩm điểm thu nhỏ trong (11.3.5), trong đó ma trận $n \times d$ được nhân với ma trận $d \times n$, sau đó ma trận $n \times n$ đầu ra được nhân với ma trận $n \times d$. Kết quả là, sự tự chú ý có độ phức tạp tính toán $\mathcal{O}(n^2d)$. Như chúng ta có thể thấy trong Fig. 11.6.1, mỗi token được kết nối trực tiếp với bất kỳ mã thông báo nào khác thông qua sự tự chú ý. Do đó, tính toán có thể song song với $\mathcal{O}(1)$ phép toán tuần tự và độ dài đường tối đa cũng là $\mathcal{O}(1)$.

Nói chung, cả CNN và sự tự chú ý đều được hưởng tính toán song song và sự tự chú ý có độ dài đường tối đa ngắn nhất. Tuy nhiên, độ phức tạp tính toán bậc hai đối với độ dài trình tự làm cho sự tự chú ý rất chậm đối với các trình tự rất dài.

11.6.3 Mã hóa vị trí

Unlike RNNs that recurrently process tokens of a sequence one by one, self-attention ditches sequential operations in favor of parallel computation. To use the sequence order information, we can inject absolute or relative positional information by adding *positional encoding* to the input representations. Positional encodings can be either learned or fixed. In the following, we describe a fixed positional encoding based on sine and cosine functions (Vaswani et al., 2017).

Suppose that the input representation $\mathbf{X} \in \mathbb{R}^{n \times d}$ contains the d-dimensional embeddings for n tokens of a sequence. The positional encoding outputs $\mathbf{X} + \mathbf{P}$ using a positional embedding matrix $\mathbf{P} \in \mathbb{R}^{n \times d}$ of the same shape, whose element on the i^{th} row and the $(2j)^{\text{th}}$ or the $(2j+1)^{\text{th}}$ column is

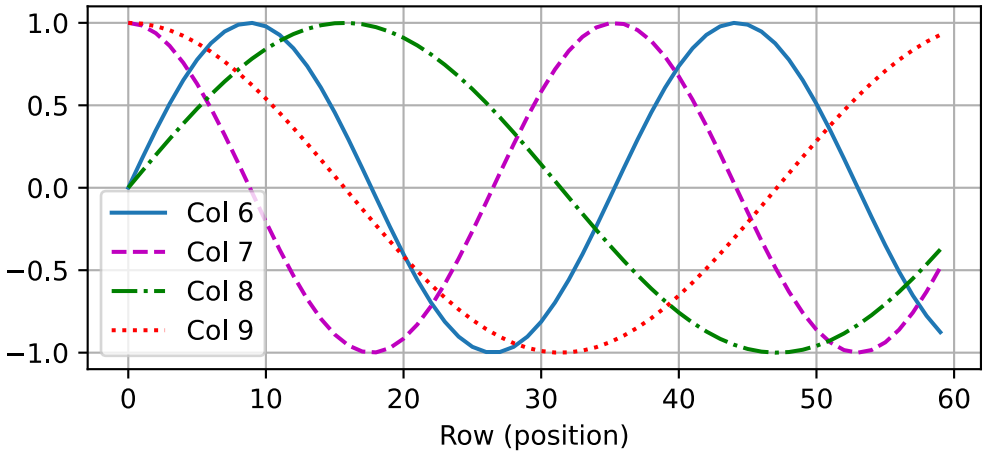
$$\begin{aligned} p_{i,2j} &= \sin\left(\frac{i}{10000^{2j/d}}\right), \\ p_{i,2j+1} &= \cos\left(\frac{i}{10000^{2j/d}}\right). \end{aligned} \tag{11.6.2}$$

At first glance, this trigonometric-function design looks weird. Before explanations of this design, let us first implement it in the following `PositionalEncoding` class.

```
#@save
class PositionalEncoding(nn.Module):
    """Positional encoding."""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # Create a long enough `P`
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
            -1, 1) / torch.pow(10000, torch.arange(
            0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

In the positional embedding matrix **P**, rows correspond to positions within a sequence and columns represent different positional encoding dimensions. In the example below, we can see that the 6^{th} and the 7^{th} columns of the positional embedding matrix have a higher frequency than the 8^{th} and the 9^{th} columns. The offset between the 6^{th} and the 7^{th} (same for the 8^{th} and the 9^{th}) columns is due to the alternation of sine and cosine functions.

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
         figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
```



Absolute Positional Information

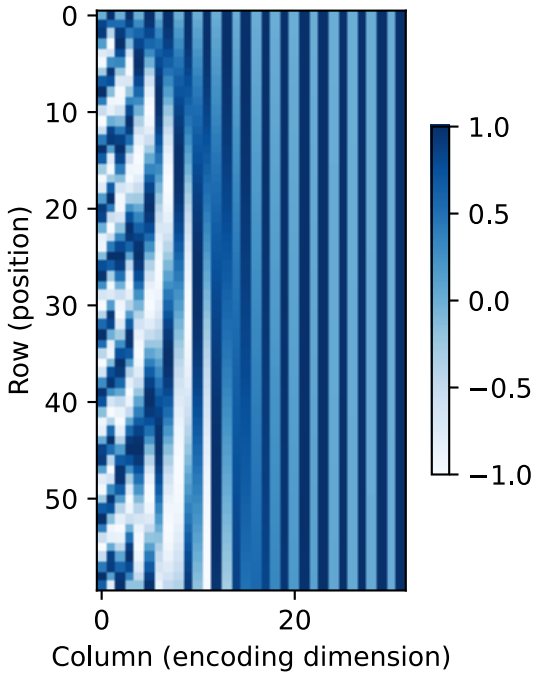
Để xem tần số giảm đơn điệu dọc theo kích thước mã hóa liên quan đến thông tin vị trí tuyệt đối như thế nào, chúng ta hãy in ra đại diện nhị phân của $0, 1, \ldots, 7$. Như chúng ta có thể thấy, bit thấp nhất, bit thấp thứ hai và bit thấp nhất thứ ba thay thế trên mỗi số, mỗi hai số và mỗi bốn số, tương ứng.

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

Trong biểu diễn nhị phân, một bit cao hơn có tần số thấp hơn một bit thấp hơn. Tương tự, như đã thể hiện trong bản đồ nhiệt bên dưới, mã hóa vị trí làm giảm tần số dọc theo kích thước mã hóa bằng cách sử dụng các hàm lượng giác. Vì các đầu ra là số float, các biểu diễn liên tục như vậy có hiệu quả không gian hơn so với biểu diễn nhị phân.

```
P = P[0, :, :].unsqueeze(0).unsqueeze(0)
d2l.show_heatmaps(P, xlabel='Column (encoding dimension)',
                  ylabel='Row (position)', figsize=(3.5, 4), cmap='Blues')
```



Relative Positional Information

Besides capturing absolute positional information, the above positional encoding also allows a model to easily learn to attend by relative positions. This is because for any fixed position offset δ, the positional encoding at position $i + \delta$ can be represented by a linear projection of that at position i.

This projection can be explained mathematically. Denoting $\omega_j = 1/10000^{2j/d}$, any pair of $(p_{i,2j}, p_{i,2j+1})$ in (11.6.2) can be linearly projected to $(p_{i+\delta,2j}, p_{i+\delta,2j+1})$ for any fixed offset δ:

$$\begin{aligned}
&\begin{bmatrix} \cos(\delta\omega_j) & \sin(\delta\omega_j) \\ -\sin(\delta\omega_j) & \cos(\delta\omega_j) \end{bmatrix} \begin{bmatrix} p_{i,2j} \\ p_{i,2j+1} \end{bmatrix} \\
&= \begin{bmatrix} \cos(\delta\omega_j)\sin(i\omega_j) + \sin(\delta\omega_j)\cos(i\omega_j) \\ -\sin(\delta\omega_j)\sin(i\omega_j) + \cos(\delta\omega_j)\cos(i\omega_j) \end{bmatrix} \\
&= \begin{bmatrix} \sin\left((i+\delta)\omega_j\right) \\ \cos\left((i+\delta)\omega_j\right) \end{bmatrix} \\
&= \begin{bmatrix} p_{i+\delta,2j} \\ p_{i+\delta,2j+1} \end{bmatrix},
\end{aligned} \tag{11.6.3}$$

where the 2×2 projection matrix does not depend on any position index i.

11.6.4 Tóm tắt

- Trong sự tự chú ý, các truy vấn, khóa và giá trị đều đến từ cùng một nơi.
- Cả CNN và sự tự chú ý đều được hưởng tính toán song song và sự tự chú ý có độ dài đường tối đa ngắn nhất. Tuy nhiên, độ phức tạp tính toán bậc hai đối với độ dài trình tự làm cho sự tự chú ý rất chậm đối với các trình tự rất dài.
- Để sử dụng thông tin thứ tự trình tự, chúng ta có thể tiêm thông tin vị trí tuyệt đối hoặc tương đối bằng cách thêm mã hóa vị trí vào biểu diễn đầu vào.

11.6.5 Bài tập

1. Giả sử rằng chúng ta thiết kế một kiến trúc sâu để đại diện cho một chuỗi bằng cách xếp chồng các lớp tự chú ý với mã hóa vị trí. Những gì có thể là vấn đề?
2. Bạn có thể thiết kế một phương pháp mã hóa vị trí có thể học được không?

Discussions[123]

11.7 Máy biến áp

Chúng tôi đã so sánh CNN, RNN và sự tự chú ý trong Section 11.6.2. Đáng chú ý, sự tự chú ý thích cả tính toán song song và độ dài đường tối đa ngắn nhất. Do đó về mặt tự nhiên, nó là hấp dẫn để thiết kế kiến trúc sâu sắc bằng cách sử dụng sự tự chú ý. Không giống như các mô hình tự chú ý trước đó vẫn dựa vào RNN để biểu diễn đầu vào [Cheng.Dong.Lapata.2016][Lin.Feng.Santos.ea.2017][Paulus.Xiong.Socher.2017], mô hình máy biến áp chỉ dựa trên các cơ chế chú ý mà không có bất kỳ lớp phức tạp hoặc tái phát nào (Vaswani et al., 2017). Mặc dù ban đầu được đề xuất cho trình tự để học trình tự về dữ liệu văn bản, các máy biến áp đã phổ biến trong một loạt các ứng dụng học sâu hiện đại, chẳng hạn như trong các lĩnh vực ngôn ngữ, tầm nhìn, lời nói và học tập củng cố.

11.7.1 Mô hình

Là một ví dụ của kiến trúc bộ mã hóa-giải mã, kiến trúc tổng thể của máy biến áp được trình bày trong Fig. 11.7.1. Như chúng ta có thể thấy, máy biến áp bao gồm một bộ mã hóa và bộ giải mã. Khác với sự chú ý của Bahdanau cho trình tự để học trình tự trong Fig. 11.4.1, các nhúng chuỗi đầu vào (nguồn) và đầu ra (mục tiêu) được thêm vào với mã hóa vị trí trước khi được đưa vào bộ mã hóa và bộ giải mã ngăn xếp các mô-đun dựa trên sự tự chú ý.

[123] https://discuss.d2l.ai/t/1652

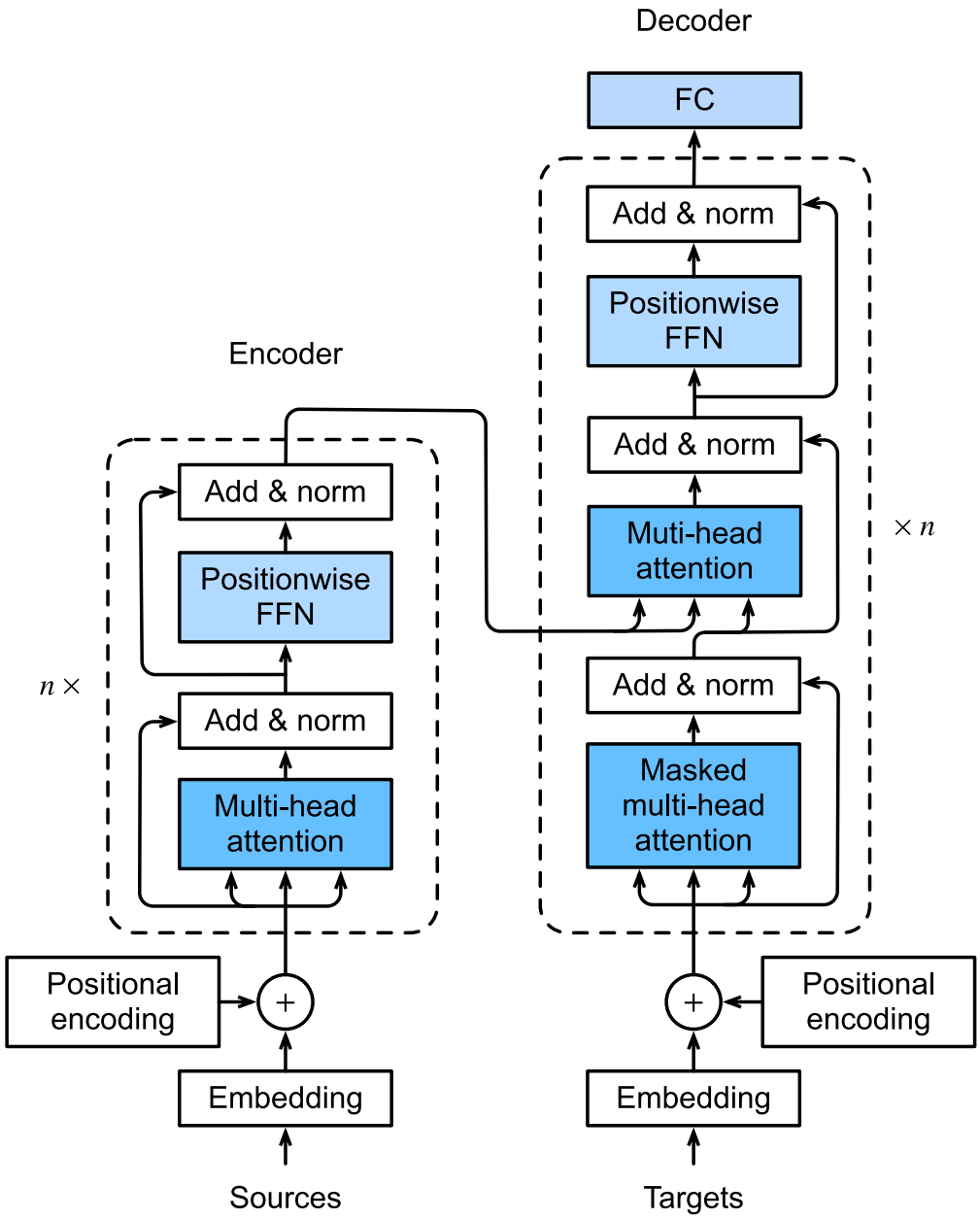


Fig. 11.7.1: The transformer architecture.

Bây giờ chúng tôi cung cấp một cái nhìn tổng quan về kiến trúc biến áp trong Fig. 11.7.1. Ở mức độ cao, bộ mã hóa biến áp là một chồng của nhiều lớp giống hệt nhau, trong đó mỗi lớp có hai lớp con (hoặc được ký hiệu là sublayer). Đầu tiên là một tập hợp tự chú ý nhiều đầu và thứ hai là một mạng lưới chuyển tiếp nguồn cấp dữ liệu theo định vị. Cụ thể, trong bộ mã hóa tự chú ý, truy vấn, phím và giá trị đều từ đầu ra của lớp mã hóa trước đó. Lấy cảm hứng từ thiết kế ResNet trong Section 8.6, một kết nối còn lại được sử dụng xung quanh cả hai lớp con. Trong máy biến áp, đối với bất kỳ đầu vào $\mathbf{x} \in \mathbb{R}^d$ nào ở bất kỳ vị trí nào của chuỗi, chúng tôi yêu cầu $\text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ để kết nối còn lại $\mathbf{x} + \text{sublayer}(\mathbf{x}) \in \mathbb{R}^d$ là khả thi. Bổ sung này từ kết nối còn lại ngay lập tức được theo sau bởi chuẩn hóa lớp [Ba.Kiros.Hinton.2016]. Kết quả là, bộ mã hóa biến áp xuất ra một biểu diễn vector d chiều cho mỗi vị trí của chuỗi đầu vào.

Bộ giải mã biến áp cũng là một chồng của nhiều lớp giống hệt nhau với các kết nối còn lại và chuẩn hóa lớp. Bên cạnh hai sublayers được mô tả trong bộ mã hóa, bộ giải mã chèn một sublayer thứ ba, được gọi là sự chú

ý của bộ mã hóa-giải mã, giữa hai bộ giải mã này. Trong sự chú ý của bộ mã hóa giải mã, các truy vấn là từ đầu ra của lớp giải mã trước đó và các phím và giá trị là từ đầu ra bộ mã hóa biến áp. Trong bộ giải mã tự chú ý, truy vấn, khóa và giá trị đều từ đầu ra của lớp giải mã trước đó. Tuy nhiên, mỗi vị trí trong bộ giải mã chỉ được phép tham dự tất cả các vị trí trong bộ giải mã cho đến vị trí đó. Chú ý *masked* này bảo tồn thuộc tính tự động hồi quy, đảm bảo rằng dự đoán chỉ phụ thuộc vào các token đầu ra đã được tạo ra.

Chúng tôi đã mô tả và thực hiện sự chú ý nhiều đầu dựa trên các sản phẩm dot-thu nhỏ trong Section 11.5 và mã hóa vị trí trong Section 11.6.3. Sau đây, chúng tôi sẽ thực hiện phần còn lại của mô hình máy biến áp.

```
import math
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

11.7.2 Các mạng thức ăn chuyển tiếp vị trí

Mạng chuyển tiếp nguồn cấp dữ liệu định vị biến đổi biểu diễn ở tất cả các vị trí trình tự bằng cách sử dụng cùng một MLP. Đây là lý do tại sao chúng tôi gọi nó là *positionwise*. Trong phần thực hiện dưới đây, đầu vào `X` với hình dạng (kích thước lô, số bước thời gian hoặc độ dài chuỗi trong thẻ, số đơn vị ẩn hoặc kích thước tính năng) sẽ được chuyển đổi bởi một MLP hai lớp thành một tensor đầu ra của hình dạng (kích thước lô, số bước thời gian, `ffn_num_outputs`).

```
#@save
class PositionWiseFFN(nn.Module):
    """Positionwise feed-forward network."""
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
                 **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

Ví dụ sau đây cho thấy chiều trong cùng của một tensor thay đổi thành số lượng đầu ra trong mạng chuyển tiếp theo vị trí. Vì cùng một MLP biến đổi ở tất cả các vị trí, khi các đầu vào ở tất cả các vị trí này giống nhau, đầu ra của chúng cũng giống hệt nhau.

```
ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
ffn(torch.ones((2, 3, 4)))[0]
```

```
tensor([[ 0.3415, -0.6896, -0.0887, -0.0173, -0.1739,  0.2835,  0.6384,  0.
↪3857],
        [ 0.3415, -0.6896, -0.0887, -0.0173, -0.1739,  0.2835,  0.6384,  0.
↪3857],
        [ 0.3415, -0.6896, -0.0887, -0.0173, -0.1739,  0.2835,  0.6384,  0.
↪3857]],
       grad_fn=<SelectBackward0>)
```

11.7.3 Kết nối còn lại và chuẩn hóa lớp

Bây giờ chúng ta hãy tập trung vào thành phần "add & norm" trong Fig. 11.7.1. Như chúng tôi đã mô tả ở đầu phần này, đây là một kết nối còn lại ngay lập tức theo sau là chuẩn hóa lớp. Cả hai đều là chìa khóa cho các kiến trúc sâu hiệu quả.

Trong Section 8.5, chúng tôi đã giải thích cách bình thường hóa hàng loạt lại và rescales trên các ví dụ trong một minibatch. Chuẩn hóa lớp giống như bình thường hóa hàng loạt ngoại trừ việc trước đây bình thường hóa trên kích thước tính năng. Mặc dù các ứng dụng phổ biến của nó trong tầm nhìn máy tính, việc bình thường hóa hàng loạt thường kém hiệu quả hơn so với bình thường hóa lớp trong các tác vụ xử lý ngôn ngữ tự nhiên, có đầu vào thường là chuỗi độ dài thay đổi.

Đoạn mã sau đây so sánh bình thường hóa trên các kích thước khác nhau theo chuẩn hóa lớp và chuẩn hóa hàng lạnh.

```
ln = nn.LayerNorm(2)
bn = nn.BatchNorm1d(2)
X = torch.tensor([[1, 2], [2, 3]], dtype=torch.float32)
# Compute mean and variance from `X` in the training mode
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([[-1.0000,  1.0000],
        [-1.0000,  1.0000]], grad_fn=<NativeLayerNormBackward0>)
batch norm: tensor([[-1.0000, -1.0000],
        [ 1.0000,  1.0000]], grad_fn=<NativeBatchNormBackward0>)
```

Bây giờ chúng ta có thể thực hiện lớp `AddNorm` sử dụng một kết nối còn lại theo sau là chuẩn hóa lớp . Dropout cũng được áp dụng để thường xuyên hóa.

```
#@save
class AddNorm(nn.Module):
    """Residual connection followed by layer normalization."""
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

Kết nối còn lại yêu cầu hai đầu vào có cùng hình dạng sao cho tensor đầu ra cũng có hình dạng giống nhau sau khi hoạt động bổ sung.

```
add_norm = AddNorm([3, 4], 0.5) # Normalized_shape is input.size()[1:]
add_norm.eval()
add_norm(torch.ones((2, 3, 4)), torch.ones((2, 3, 4))).shape
```

```
torch.Size([2, 3, 4])
```

11.7.4 Bộ mã hóa

Với tất cả các thành phần thiết yếu để lắp ráp bộ mã hóa biến áp, chúng ta hãy bắt đầu bằng cách thực hiện một lớp duy nhất trong bộ mã hóa. Lớp `EncoderBlock` sau chứa hai lớp con: nhiều đầu tự chú ý và các mạng chuyển tiếp theo định vị, trong đó một kết nối còn lại tiếp theo là chuẩn hóa lớp được sử dụng xung quanh cả hai lớp con.

```
#@save
class EncoderBlock(nn.Module):
    """Transformer encoder block."""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout,
            use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
            ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

Như chúng ta có thể thấy, bất kỳ lớp nào trong bộ mã hóa biến áp không thay đổi hình dạng của đầu vào của nó.

```
X = torch.ones((2, 100, 24))
valid_lens = torch.tensor([3, 2])
encoder_blk = EncoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5)
encoder_blk.eval()
encoder_blk(X, valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

Trong thực hiện transformer encoder sau đây, chúng tôi xếp chồng `num_layers` phiên bản của `EncoderBlock` lớp trên. Vì chúng ta sử dụng mã hóa vị trí cố định có giá trị luôn nằm trong khoảng từ -1 đến 1, chúng ta nhân các giá trị của các nhúng đầu vào có thể học được bằng căn bậc hai của kích thước nhúng để giải phóng trước khi tổng hợp nhúng đầu vào và mã hóa vị trí.

```
#@save
class TransformerEncoder(d2l.Encoder):
    """Transformer encoder."""
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
```

(continues on next page)

(continued from previous page)

```
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

Dưới đây chúng tôi chỉ định các siêu tham số để tạo một bộ mã hóa biến áp hai lớp. Hình dạng của đầu ra bộ mã hóa biến áp là (kích thước lô, số bước thời gian, `num_hiddens`).

```
encoder = TransformerEncoder(
    200, 24, 24, 24, 24, [100, 24], 24, 48, 8, 2, 0.5)
encoder.eval()
encoder(torch.ones((2, 100), dtype=torch.long), valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

11.7.5 Bộ giải mã

Như thể hiện trong Fig. 11.7.1, bộ giải mã biến áp bao gồm nhiều lớp giống hệt nhau. Mỗi lớp được triển khai trong lớp `DecoderBlock` sau, chứa ba lớp con: bộ giải mã tự chú ý, chú ý bộ mã hóa-giải mã, và các mạng chuyển tiếp nguồn cấp dữ liệu định vị. Các lớp con này sử dụng một kết nối còn lại xung quanh chúng theo sau là chuẩn hóa lớp.

Như chúng ta đã mô tả trước đó trong phần này, trong bộ giải mã đa đầu được đeo mặt nạ tự chú ý (con đầu tiên), truy vấn, khóa và giá trị đều đến từ đầu ra của lớp giải mã trước đó. Khi đào tạo các mô hình trình tự theo trình tự, mã thông báo ở tất cả các vị trí (bước thời gian) của chuỗi đầu ra được biết đến. Tuy nhiên, trong quá trình dự đoán chuỗi đầu ra được tạo ra token bằng mã thông báo; do đó, tại bất kỳ bước thời gian giải mã nào chỉ có các token được tạo ra có thể được sử dụng trong bộ giải mã tự chú ý. Để duy trì hồi quy tự động trong bộ giải mã, sự tự chú ý đeo mặt nạ của nó chỉ định `dec_valid_lens` để bất kỳ truy vấn nào chỉ tham quan đến tất cả các vị trí trong bộ giải mã cho đến vị trí truy vấn.

```
class DecoderBlock(nn.Module):
    # The `i`-th block in the decoder
    def __init__(self, key_size, query_size, value_size, num_hiddens,
                 norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
```

(continues on next page)

(continued from previous page)

```
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
                                   num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # During training, all the tokens of any output sequence are processed
        # at the same time, so `state[2][self.i]` is `None` as initialized.
        # When decoding any output sequence token by token during prediction,
        # `state[2][self.i]` contains representations of the decoded output at
        # the `i`-th block up to the current time step
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if self.training:
            batch_size, num_steps, _ = X.shape
            # Shape of `dec_valid_lens`: (`batch_size`, `num_steps`), where
            # every row is [1, 2, ..., `num_steps`]
            dec_valid_lens = torch.arange(
                1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None

        # Self-attention
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
        Y = self.addnorm1(X, X2)
        # Encoder-decoder attention. Shape of `enc_outputs`:
        # (`batch_size`, `num_steps`, `num_hiddens`)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

Để tạo điều kiện cho các hoạt động sản phẩm điểm thu nhỏ trong bộ giải mã mã hóa chú ý và các hoạt động bổ sung trong các kết nối còn lại, kích thước tính năng (`num_hiddens`) của bộ giải mã giống như của bộ mã hóa.

```
decoder_blk = DecoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5, 0)
decoder_blk.eval()
X = torch.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
decoder_blk(X, state)[0].shape
```

```
torch.Size([2, 100, 24])
```

Bây giờ chúng ta xây dựng toàn bộ bộ giải mã transformer bao gồm `num_layers` trường hợp của `DecoderBlock`. Cuối cùng, một lớp kết nối hoàn toàn tính toán dự đoán cho tất cả các token đầu ra có thể có `vocab_size`. Cả hai trọng lượng tự chú ý của bộ giải mã và trọng lượng chú ý bộ giải mã hóa được lưu trữ

để trực quan hóa sau này.

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                DecoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, enc_valid_lens, *args):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]

    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # Decoder self-attention weights
            self._attention_weights[0][
                i] = blk.attention1.attention.attention_weights
            # Encoder-decoder attention weights
            self._attention_weights[1][
                i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

11.7.6 Đào tạo

Hãy để chúng tôi khởi tạo một mô hình bộ mã hóa-giải mã bằng cách làm theo kiến trúc biến áp. Ở đây chúng tôi chỉ định rằng cả bộ mã hóa biến áp và bộ giải mã biến áp có 2 lớp sử dụng sự chú ý 4 đầu. Tương tự như Section 10.7.4, chúng tôi đào tạo mô hình biến áp cho trình tự để học trình tự trên tập dữ liệu dịch máy tiếng Anh-Pháp.

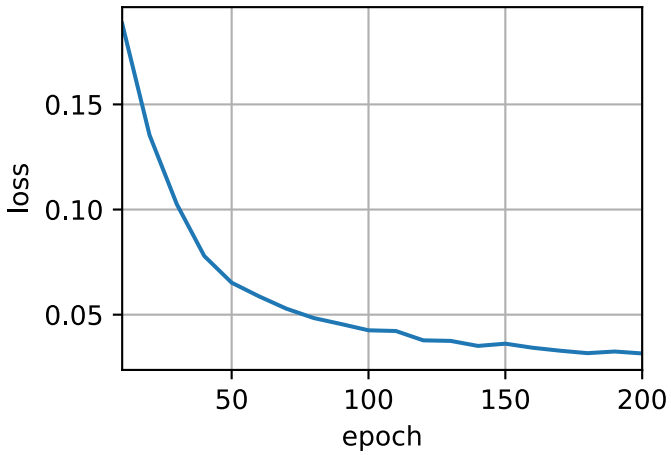
```
num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.1, 64, 10
lr, num_epochs, device = 0.005, 200, d2l.try_gpu()
ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
norm_shape = [32]

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
```

(continues on next page)

(continued from previous page)

```
encoder = TransformerEncoder(
    len(src_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
decoder = TransformerDecoder(
    len(tgt_vocab), key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    num_layers, dropout)
net = d2l.EncoderDecoder(encoder, decoder)
d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.032, 5927.0 tokens/sec on cuda:0
```



Sau khi đào tạo, chúng tôi sử dụng mô hình biến áp để dịch một vài câu tiếng Anh sang tiếng Pháp và tính điểm BLEU của họ.

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est <unk> .,  bleu 0.658
i'm home . => je suis chez moi .,  bleu 1.000
```

Hãy để chúng tôi hình dung trọng lượng chú ý của máy biến áp khi dịch câu tiếng Anh cuối cùng sang tiếng Pháp. Hình dạng của các quả cân tự chú ý của bộ mã hóa là (số lớp mã hóa, số lượng đầu chú ý, `num_steps` hoặc số truy vấn, `num_steps` hoặc số cặp giá trị khóa).

```
enc_attention_weights = torch.cat(net.encoder.attention_weights, 0).
↪reshape((num_layers, num_heads,
```

(continues on next page)

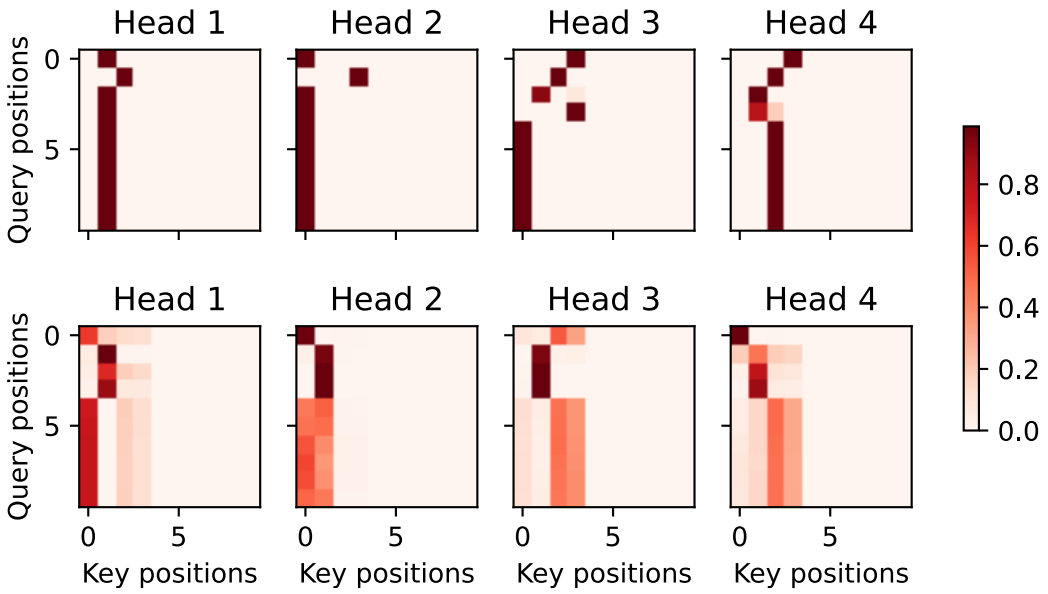
(continued from previous page)

```
    -1, num_steps))
enc_attention_weights.shape
```

```
torch.Size([2, 4, 10, 10])
```

Trong bộ mã hóa tự chú ý, cả truy vấn và phím đều đến từ cùng một chuỗi đầu vào. Vì thẻ đệm không mang ý nghĩa, với độ dài hợp lệ được chỉ định của chuỗi đầu vào, không có truy vấn nào tham dự các vị trí của thẻ đệm. Sau đây, hai lớp trọng lượng chú ý nhiều đầu được trình bày từng hàng. Mỗi đầu độc lập tham dự dựa trên một không gian con đại diện riêng biệt của các truy vấn, khóa và giá trị.

```
d2l.show_heatmaps(
    enc_attention_weights.cpu(), xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



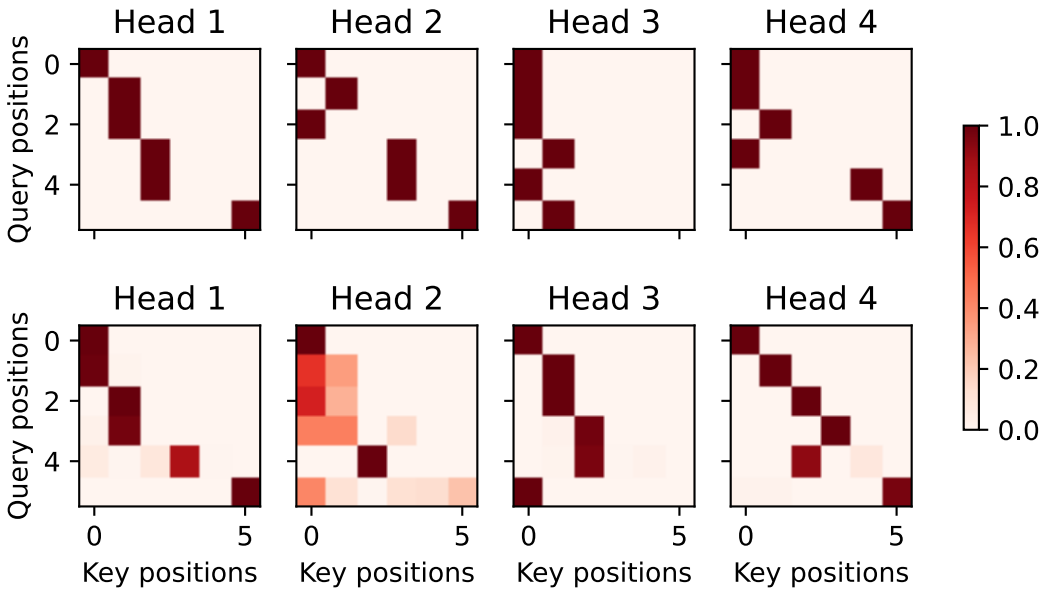
Để hình dung cả trọng lượng tự chú ý của bộ giải mã và trọng lượng chú ý bộ giải mã hóa, chúng ta cần nhiều thao tác dữ liệu Ví dụ, chúng tôi lấp đầy trọng lượng chú ý đeo mặt nạ bằng 0. Lưu ý rằng trọng lượng tự chú ý của bộ giải mã và trọng lượng chú ý bộ giải mã hóa cả hai đều có cùng một truy vấn: mã thông báo bắt đầu theo sau là mã thông báo đầu ra.

```
dec_attention_weights_2d = [head[0].tolist()
                            for step in dec_attention_weight_seq
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = torch.tensor(
    pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = dec_attention_weights_filled.reshape((-1, 2, num_
↪layers, num_heads, num_steps))
dec_self_attention_weights, dec_inter_attention_weights = \
    dec_attention_weights.permute(1, 2, 3, 0, 4)
dec_self_attention_weights.shape, dec_inter_attention_weights.shape
```

```
(torch.Size([2, 4, 6, 10]), torch.Size([2, 4, 6, 10]))
```

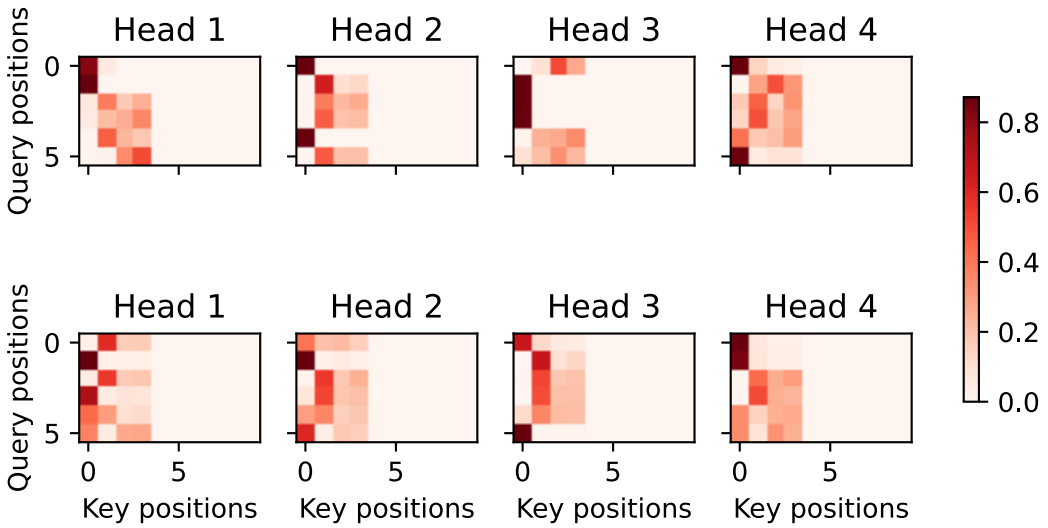
Do thuộc tính tự động hồi quy của bộ giải mã tự chú ý, không có truy vấn nào tham dự các cặp khóa-giá trị sau vị trí truy vấn.

```
# Plus one to include the beginning-of-sequence token
d2l.show_heatmaps(
    dec_self_attention_weights[:, :, :, :len(translation.split()) + 1],
    xlabel='Key positions', ylabel='Query positions',
    titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```



Tương tự như trường hợp trong bộ mã hóa tự chú ý, thông qua độ dài hợp lệ được chỉ định của chuỗi đầu vào, không có truy vấn từ chuỗi đầu ra tham quan đến các thẻ đệm đó từ chuỗi đầu vào.

```
d2l.show_heatmaps(
    dec_inter_attention_weights, xlabel='Key positions',
    ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
    figsize=(7, 3.5))
```



Mặc dù kiến trúc biến áp ban đầu được đề xuất để học theo trình tự, vì chúng ta sẽ khám phá sau trong cuốn sách, bộ mã hóa biến áp hoặc bộ giải mã biến áp thường được sử dụng riêng cho các nhiệm vụ học sâu khác nhau.

11.7.7 Tóm tắt

- Máy biến áp là một thể hiện của kiến trúc bộ mã hóa-giải mã, mặc dù bộ mã hóa hoặc bộ giải mã có thể được sử dụng riêng lẻ trong thực tế.
- Trong máy biến áp, tự chú ý nhiều đầu được sử dụng để biểu diễn trình tự đầu vào và chuỗi đầu ra, mặc dù bộ giải mã phải bảo toàn thuộc tính hồi quy tự động thông qua một phiên bản đeo mặt nạ.
- Cả hai kết nối còn lại và bình thường hóa lớp trong máy biến áp đều quan trọng để đào tạo một mô hình rất sâu.
- Mạng chuyển tiếp nguồn cấp dữ liệu định vị trong mô hình máy biến áp biến đổi biểu diễn ở tất cả các vị trí trình tự bằng cách sử dụng cùng một MLP.

11.7.8 Bài tập

1. Đào tạo một máy biến áp sâu hơn trong các thí nghiệm. Làm thế nào để nó ảnh hưởng đến tốc độ đào tạo và hiệu suất dịch thuật?
2. Có phải là một ý tưởng tốt để thay thế sự chú ý của sản phẩm điểm thu nhỏ bằng sự chú ý phụ gia trong máy biến áp? Tại sao?
3. Đối với mô hình hóa ngôn ngữ, chúng ta có nên sử dụng bộ mã hóa biến áp, bộ giải mã hoặc cả hai? Làm thế nào để thiết kế phương pháp này?
4. Điều gì có thể là thách thức đối với máy biến áp nếu chuỗi đầu vào rất dài? Tại sao?
5. Làm thế nào để cải thiện tính toán và hiệu quả bộ nhớ của máy biến áp? Hint: you may refer to the survey paper by Tay et al. [Tay.Dehghani.Bahri.ea.2020].
6. Làm thế nào chúng ta có thể thiết kế các mô hình dựa trên máy biến áp cho các tác vụ phân loại hình ảnh mà không cần sử dụng CNN? Hint: you may refer to the vision transformer [Dosovitskiy.Beyer.Kolesnikov.ea.2021].

Discussions[124]

[124] https://discuss.d2l.ai/t/1066

12 | Thuật toán tối ưu hóa

Nếu bạn đọc cuốn sách theo thứ tự cho đến thời điểm này, bạn đã sử dụng một số thuật toán tối ưu hóa để đào tạo các mô hình học sâu. Chúng là những công cụ cho phép chúng tôi tiếp tục cập nhật các tham số mô hình và giảm thiểu giá trị của chức năng mất mát, như được đánh giá trên bộ đào tạo. Thật vậy, bất kỳ ai nội dung với việc xử lý tối ưu hóa như một thiết bị hộp đen để giảm thiểu các chức năng khách quan trong một cài đặt đơn giản cũng có thể nội dung chính mình với kiến thức rằng tồn tại một loạt các câu thần chú của một thủ tục như vậy (với tên như "SGD" và "Adam").

Tuy nhiên, để làm tốt, một số kiến thức sâu hơn là bắt buộc. Thuật toán tối ưu hóa rất quan trọng đối với việc học sâu. Một mặt, đào tạo một mô hình học sâu phức tạp có thể mất hàng giờ, ngày hoặc thậm chí vài tuần. Hiệu suất của thuật toán tối ưu hóa ảnh hưởng trực tiếp đến hiệu quả đào tạo của mô hình. Mặt khác, việc hiểu các nguyên tắc của các thuật toán tối ưu hóa khác nhau và vai trò của các siêu tham số của chúng sẽ cho phép chúng ta điều chỉnh các siêu tham số theo cách nhằm mục tiêu để cải thiện hiệu suất của các mô hình học sâu.

Trong chương này, chúng tôi khám phá các thuật toán tối ưu hóa học sâu phổ biến trong chuyên sâu. Hầu như tất cả các vấn đề tối ưu hóa phát sinh trong học sâu đều là * nonconvex*. Tuy nhiên, việc thiết kế và phân tích các thuật toán trong bối cảnh các bài toán * lồi đã được chứng minh là rất hướng dẫn. Chính vì lý do đó mà chương này bao gồm một mồi về tối ưu hóa lồi và bằng chứng cho một thuật toán gốc gradient ngẫu nhiên rất đơn giản trên một chức năng khách quan lồi.

12.1 Tối ưu hóa và học sâu

Trong phần này, chúng ta sẽ thảo luận về mối quan hệ giữa tối ưu hóa và học sâu cũng như những thách thức của việc sử dụng tối ưu hóa trong học sâu. Đối với một vấn đề học sâu, chúng ta thường sẽ xác định hàm *loss* trước tiên. Khi chúng tôi có chức năng mất mát, chúng ta có thể sử dụng thuật toán tối ưu hóa để cố gắng giảm thiểu tổn thất. Trong tối ưu hóa, một hàm mất thường được gọi là chức năng mục tiêu * của bài toán tối ưu hóa. Theo truyền thống và quy ước hầu hết các thuật toán tối ưu hóa có liên quan đến *giảm thiểu *. Nếu chúng ta cần tối đa hóa một mục tiêu thì có một giải pháp đơn giản: chỉ cần lật dấu hiệu trên mục tiêu.

12.1.1 Mục tiêu tối ưu hóa

Mặc dù tối ưu hóa cung cấp một cách để giảm thiểu chức năng mất mát cho học sâu, nhưng về bản chất, các mục tiêu tối ưu hóa và học sâu về cơ bản là khác nhau. Cái trước chủ yếu quan tâm đến việc giảm thiểu một mục tiêu trong khi sau này liên quan đến việc tìm kiếm một mô hình phù hợp, cho một lượng dữ liệu hữu hạn. Năm Section 5.4, chúng tôi đã thảo luận chi tiết về sự khác biệt giữa hai mục tiêu này. Ví dụ, lỗi đào tạo và lỗi tổng quát thường khác nhau: vì chức năng khách quan của thuật toán tối ưu hóa thường là hàm mất dựa trên tập dữ liệu đào tạo, mục tiêu tối ưu hóa là giảm lỗi đào tạo. Tuy nhiên, mục tiêu của học sâu (hoặc rộng hơn là suy luận thống kê) là giảm sai số tổng quát hóa. Để thực hiện sau này, chúng ta cần chú ý đến overfitting ngoài việc sử dụng thuật toán tối ưu hóa để giảm lỗi đào tạo.

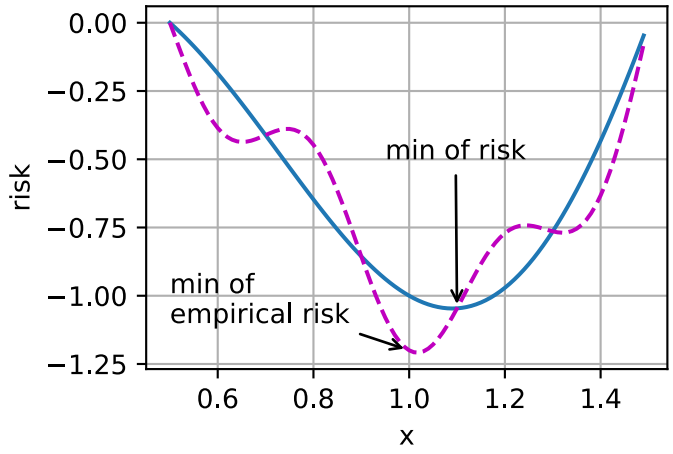
```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

Để minh họa các mục tiêu khác nhau đã nói ở trên, chúng ta hãy xem xét rủi ro thực nghiệm và rủi ro. Như được mô tả trong Section 5.9.3, rủi ro thực nghiệm là một tổn thất trung bình trên tập dữ liệu đào tạo trong khi rủi ro là sự mất mát dự kiến trên toàn bộ dân số dữ liệu. Dưới đây chúng tôi xác định hai chức năng: hàm rủi ro `f` và hàm rủi ro thực nghiệm `g`. Giả sử rằng chúng ta chỉ có một số lượng hữu hạn của dữ liệu đào tạo. Kết quả là, ở đây `g` ít mịn hơn `f`.

```
def f(x):
    return x * torch.cos(np.pi * x)

def g(x):
    return f(x) + 0.2 * torch.cos(5 * np.pi * x)
```

Biểu đồ dưới đây minh họa rằng mức tối thiểu của rủi ro thực nghiệm trên một tập dữ liệu đào tạo có thể ở một vị trí khác với mức tối thiểu của rủi ro (lỗi tổng quát).

```
def annotate(text, xy, xytext):  #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = torch.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('min of\nempirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('min of risk', (1.1, -1.05), (0.95, -0.5))
```



12.1.2 Tối ưu hóa thách thức trong Deep Learning

Trong chương này, chúng ta sẽ tập trung cụ thể vào hiệu suất của các thuật toán tối ưu hóa trong việc giảm thiểu hàm khách quan, chứ không phải là lỗi tổng quát hóa của mô hình. Trong Section 4.1, chúng tôi phân biệt giữa các giải pháp phân tích và các giải pháp số trong các vấn đề tối ưu hóa. Trong học sâu, hầu hết các chức năng khách quan đều phức tạp và không có giải pháp phân tích. Thay vào đó, chúng ta phải sử dụng các thuật toán tối ưu hóa số. Các thuật toán tối ưu hóa trong chương này đều thuộc thể loại này.

Có rất nhiều thách thức trong tối ưu hóa học tập sâu. Một số trong những điều đáng kinh ngạc nhất là minima địa phương, điểm yên ngựa, và độ dốc biến mất. Hãy để chúng tôi có một cái nhìn tại họ.

Minima địa phương

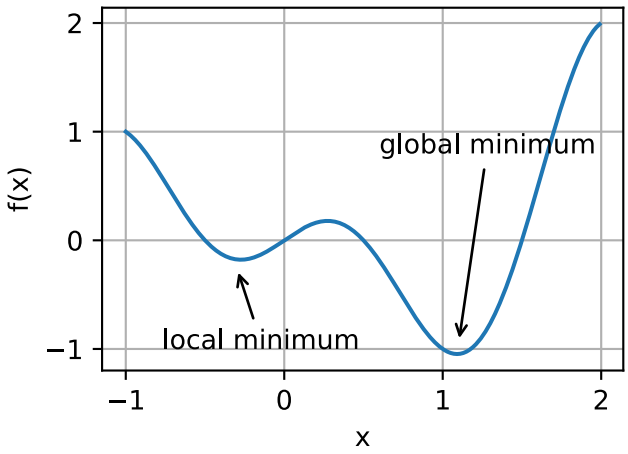
Đối với bất kỳ hàm khách quan $f(x)$, nếu giá trị của $f(x)$ tại x nhỏ hơn giá trị của $f(x)$ tại bất kỳ điểm nào khác trong vùng lân cận x, thì $f(x)$ có thể là mức tối thiểu địa phương. Nếu giá trị của $f(x)$ tại x là mức tối thiểu của hàm khách quan trên toàn bộ miền, thì $f(x)$ là mức tối thiểu toàn cầu.

Ví dụ, cho chức năng

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \tag{12.1.1}$$

chúng ta có thể xấp xỉ tối thiểu địa phương và tối thiểu toàn cầu của chức năng này.

```
x = torch.arange(-1.0, 2.0, 0.01)
d2l.plot(x, [f(x), ], 'x', 'f(x)')
annotate('local minimum', (-0.3, -0.25), (-0.77, -1.0))
annotate('global minimum', (1.1, -0.95), (0.6, 0.8))
```

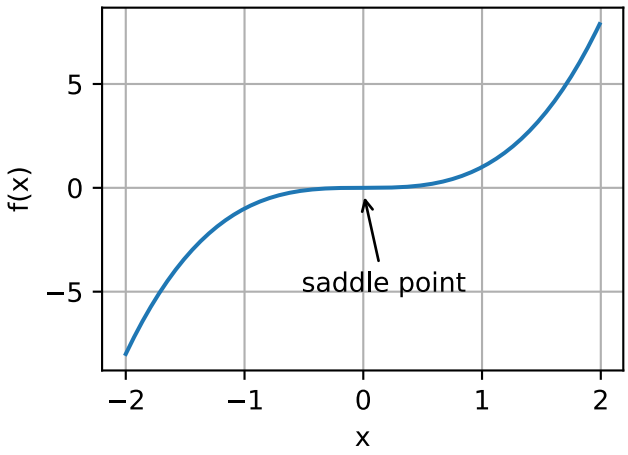


Chức năng khách quan của các mô hình học sâu thường có nhiều optima địa phương. Khi giải pháp số của một bài toán tối ưu hóa ở gần tối ưu cục bộ, giải pháp số thu được bằng cách lặp cuối cùng chỉ có thể giảm thiểu hàm mục tiêu *locally*, thay vì * globally*, khi gradient của các giải pháp của hàm khách quan tiếp cận hoặc trở thành số không. Chỉ một số mức độ tiếng ồn có thể đánh bật tham số ra khỏi mức tối thiểu cục bộ. Trên thực tế, đây là một trong những đặc tính có lợi của dòng gradient ngẫu nhiên minibatch, nơi sự thay đổi tự nhiên của gradient trên minibatches có thể loại bỏ các thông số từ minima cục bộ.

Điểm yên

Bên cạnh minima địa phương, điểm yên là một lý do khác để gradient biến mất. Một điểm yên * là bất kỳ vị trí nào mà tất cả các gradient của một hàm biến mất nhưng không phải là một toàn cầu cũng không phải là một mức tối thiểu địa phương. Hãy xem xét chức năng $f(x) = x^3$. Dẫn xuất đầu tiên và thứ hai của nó biến mất cho $x = 0$. Tối ưu hóa có thể bị đình trệ vào thời điểm này, mặc dù nó không phải là mức tối thiểu.

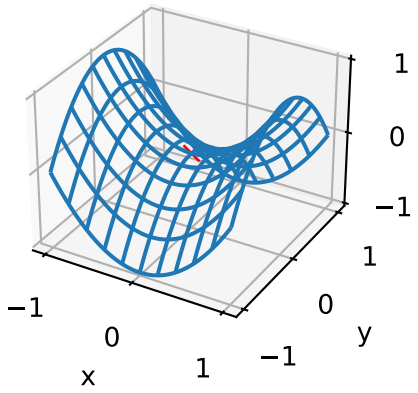
```
x = torch.arange(-2.0, 2.0, 0.01)
d2l.plot(x, [x**3], 'x', 'f(x)')
annotate('saddle point', (0, -0.2), (-0.52, -5.0))
```



Các điểm yên ở kích thước cao hơn thậm chí còn ngấm ngầm hơn, như ví dụ dưới đây cho thấy. Hãy xem xét chức năng $f(x, y) = x^2 - y^2$. Nó có điểm yên của nó ở $(0, 0)$. Đây là mức tối đa đối với y và mức tối thiểu đối với x. Hơn nữa, nó * trông giống như một yên ngựa, đó là nơi tài sản toán học này có tên của nó.

```
x, y = torch.meshgrid(
    torch.linspace(-1.0, 1.0, 101), torch.linspace(-1.0, 1.0, 101))
z = x**2 - y**2

ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot([0], [0], [0], 'rx')
ticks = [-1, 0, 1]
d2l.plt.xticks(ticks)
d2l.plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

```
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Chúng ta giả định rằng đầu vào của một hàm là một vectơ k chiều và đầu ra của nó là vô hướng, do đó ma trận Hessian của nó sẽ có k eigenvalues (tham khảo online appendix on eigendecompositions[125]). Giải pháp của hàm có thể là một mức tối thiểu cục bộ, tối đa cục bộ, hoặc một điểm yên tại một vị trí mà gradient hàm bằng 0:

- Khi eigenvalues của ma trận Hessian của hàm tại vị trí zero-gradient đều dương, chúng ta có một mức tối thiểu cục bộ cho hàm.
- Khi eigenvalues của ma trận Hessian của hàm tại vị trí zero-gradient đều âm, chúng ta có một cực đại cục bộ cho hàm.
- Khi eigenvalues của ma trận Hessian của hàm tại vị trí zero-gradient là âm và dương, chúng ta có một điểm yên cho hàm.

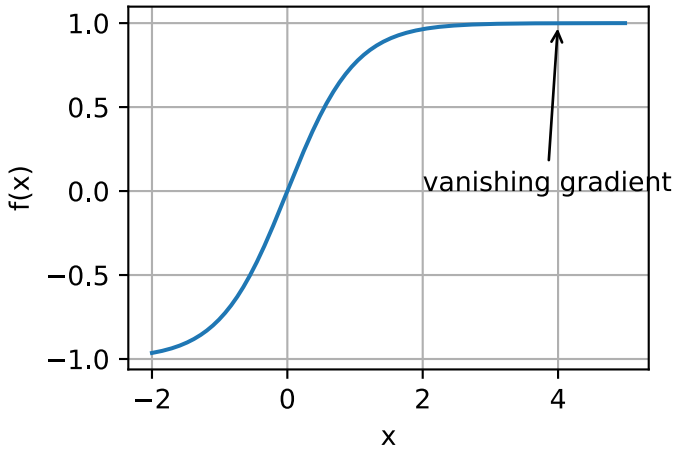
Đối với các vấn đề chiều cao, khả năng ít nhất * một số* của eigenvalues là âm là khá cao. Điều này làm cho các điểm yên có nhiều khả năng hơn minima địa phương. Chúng tôi sẽ thảo luận một số ngoại lệ đối với tình huống này trong phần tiếp theo khi giới thiệu lồi. Nói tóm lại, các hàm lồi là những hàm mà các eigenvalues của Hessian không bao giờ là âm. Đáng buồn thay, mặc dù, hầu hết các vấn đề học tập sâu không rơi vào thể loại này. Tuy nhiên, nó là một công cụ tuyệt vời để nghiên cứu các thuật toán tối ưu hóa.

Biến mất Gradient

Có lẽ vấn đề ngấm ngầm nhất để gặp phải là gradient biến mất. Nhớ lại các chức năng kích hoạt được sử dụng phổ biến của chúng tôi và các dẫn xuất của chúng trong Section 5.1.2. Ví dụ, giả sử rằng chúng tôi muốn giảm thiểu chức năng $f(x) = \tanh(x)$ và chúng tôi tình cờ bắt đầu tại $x = 4$. Như chúng ta có thể thấy, gradient của f gần với nil. Cụ thể hơn, $f'(x) = 1 - \tanh^2(x)$ và do đó $f'(4) = 0.0013$. Do đó, tối ưu hóa sẽ bị kẹt trong một thời gian dài trước khi chúng tôi tiến bộ. Đây hóa ra là một trong những lý do khiến việc đào tạo các mô hình học sâu khá khó khăn trước khi giới thiệu chức năng kích hoạt ReLU.

```
x = torch.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [torch.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```

[125] https://d2l.ai/chapter_appendix-mathematics-for-deep-learning/eigendecomposition.html



Như chúng ta đã thấy, tối ưu hóa cho học sâu là đầy thách thức. May mắn thay, tồn tại một loạt các thuật toán mạnh mẽ hoạt động tốt và dễ sử dụng ngay cả đối với người mới bắt đầu. Hơn nữa, nó không phải là thực sự cần thiết để tìm *giải pháp tốt nhất*. Optima địa phương hoặc thậm chí các giải pháp gần đúng của chúng vẫn còn rất hữu ích.

12.1.3 Tóm tắt

- Giảm thiểu lỗi đào tạo không * không* đảm bảo rằng chúng tôi tìm thấy bộ thông số tốt nhất để giảm thiểu lỗi tổng quát hóa.
- Các vấn đề tối ưu hóa có thể có nhiều minima cục bộ.
- Vấn đề có thể có nhiều điểm yên hơn, như nói chung các vấn đề không lồi.
- Độ dốc biến mất có thể gây ra tối ưu hóa để gian hàng. Thông thường một reparameterization của vấn đề giúp. Khởi tạo tốt các thông số cũng có thể có lợi.

12.1.4 Bài tập

1. Hãy xem xét một MLP đơn giản với một lớp ẩn duy nhất của, ví dụ, d kích thước trong lớp ẩn và một đầu ra duy nhất. Cho thấy rằng đối với bất kỳ mức tối thiểu địa phương có ít nhất $ d! $ các giải pháp tương đương hành xử giống hệt nhau.
2. Giả sử rằng chúng ta có một ma trận ngẫu nhiên đối xứng $\mathbf{M}$ nơi các mục $M_{ij} = M_{ji}$ mỗi được rút ra từ một số phân phối xác suất p_{ij}. Hơn nữa giả định rằng $p_{ij}(x) = p_{ij}(-x)$, tức là, rằng phân phối là đối xứng (xem ví dụ, (Wigner, 1958) để biết chi tiết).
    1. Chứng minh rằng sự phân bố trên eigenvalues cũng là đối xứng. Đó là, đối với bất kỳ eigenvector $\mathbf{v}$ xác suất mà giá trị eigenvalue liên quan λ thỏa mãn $P(\lambda > 0) = P(\lambda < 0)$.
    2. Tại sao *không* ở trên ngụ ý $P(\lambda > 0) = 0.5$?
3. Bạn có thể nghĩ đến những thách thức nào khác liên quan đến tối ưu hóa học tập sâu?
4. Giả sử rằng bạn muốn cân bằng một quả bóng (thực) trên một (thực) yên xe.
    1. Tại sao điều này khó?
    2. Bạn có thể khai thác hiệu ứng này cũng cho các thuật toán tối ưu hóa?

Discussions[126]

12.2 Độ lồi

Lồi đóng một vai trò quan trọng trong việc thiết kế các thuật toán tối ưu hóa. Điều này phần lớn là do thực tế là nó dễ dàng hơn nhiều để phân tích và kiểm tra các thuật toán trong một bối cảnh như vậy. Nói cách khác, nếu thuật toán thực hiện kém ngay cả trong cài đặt lồi, thông thường chúng ta không nên hy vọng sẽ thấy kết quả tuyệt vời khác. Hơn nữa, mặc dù các vấn đề tối ưu hóa trong học sâu nói chung là không lồi, chúng thường thể hiện một số tính chất của các vấn đề lồi gần minima cục bộ. Điều này có thể dẫn đến các biến thể tối ưu hóa mới thú vị như (Izmailov et al., 2018).

```
%matplotlib inline
import numpy as np
import torch
from mpl_toolkits import mplot3d
from d2l import torch as d2l
```

12.2.1 Định nghĩa

Trước khi phân tích lồi, chúng ta cần xác định * bộ lồi * và * hàm lồi *. Chúng dẫn đến các công cụ toán học thường được áp dụng cho học máy.

Bộ lồi

Bộ là cơ sở của sự lồi lên. Nói một cách đơn giản, một bộ $\mathcal{X}$ trong một không gian vector là * lồi * nếu đối với bất kỳ $a, b \in \mathcal{X}$ đoạn đường nối a và b cũng là trong $\mathcal{X}$. Trong thuật ngữ toán học, điều này có nghĩa là đối với tất cả $\lambda \in [0, 1]$ chúng tôi có

$$\lambda a + (1 - \lambda) b \in \mathcal{X} \text{ whenever } a, b \in \mathcal{X}. \tag{12.2.1}$$

Điều này nghe có vẻ hơi trừu tượng. Hãy xem xét Fig. 12.2.1. Tập đầu tiên không lồi vì tồn tại các đoạn đường không chứa trong đó. Hai bộ còn lại không gặp vấn đề như vậy.

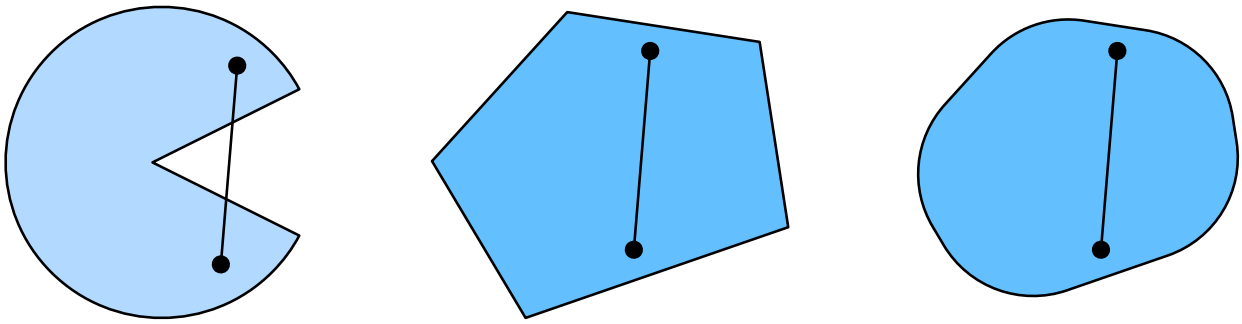


Fig. 12.2.1: The first set is nonconvex and the other two are convex.

Định nghĩa của riêng họ không đặc biệt hữu ích trừ khi bạn có thể làm điều gì đó với họ. Trong trường hợp này, chúng ta có thể xem xét các nút giao như thể hiện trong Fig. 12.2.2. Giả sử rằng $\mathcal{X}$ và $\mathcal{Y}$ là các bộ lồi.

[126] https://discuss.d2l.ai/t/487

Sau đó $\mathcal{X} \cap \mathcal{Y}$ cũng lồi. Để xem điều này, hãy xem xét bất kỳ $a, b \in \mathcal{X} \cap \mathcal{Y}$. Vì $\mathcal{X}$ và $\mathcal{Y}$ là lồi, các đoạn đường nối a và b được chứa trong cả $\mathcal{X}$ và $\mathcal{Y}$. Cho rằng, chúng cũng cần phải được chứa trong $\mathcal{X} \cap \mathcal{Y}$, do đó chứng minh định lý của chúng ta.

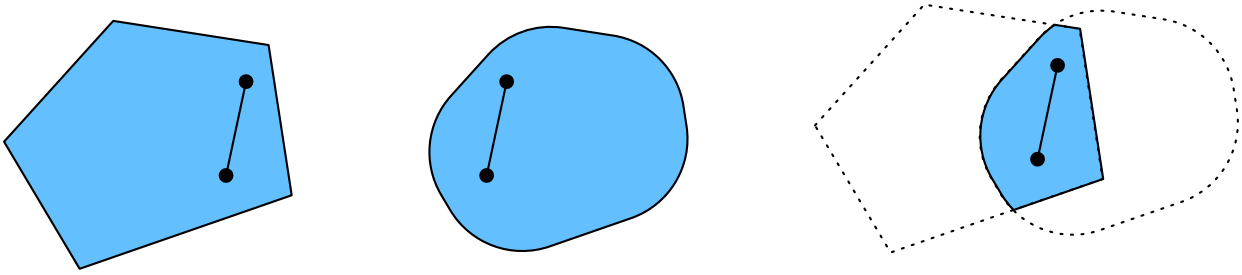


Fig. 12.2.2: The intersection between two convex sets is convex.

Chúng ta có thể củng cố kết quả này với ít nỗ lực: cho bộ lồi $\mathcal{X}_i$, giao lộ của chúng $\cap_i \mathcal{X}_i$ là lồi. Để thấy rằng cuộc trò chuyện là không đúng sự thật, hãy xem xét hai bộ tách rời $\mathcal{X} \cap \mathcal{Y} = \emptyset$. Bây giờ chọn $a \in \mathcal{X}$ và $b \in \mathcal{Y}$. Đoạn đường trong Fig. 12.2.3 kết nối a và b cần chứa một số phần không phải trong $\mathcal{X}$ cũng không phải trong $\mathcal{Y}$, vì chúng tôi giả định rằng $\mathcal{X} \cap \mathcal{Y} = \emptyset$. Do đó đoạn đường cũng không nằm trong $\mathcal{X} \cup \mathcal{Y}$, do đó chứng minh rằng trong các công đoàn chung của bộ lồi không cần phải lồi.

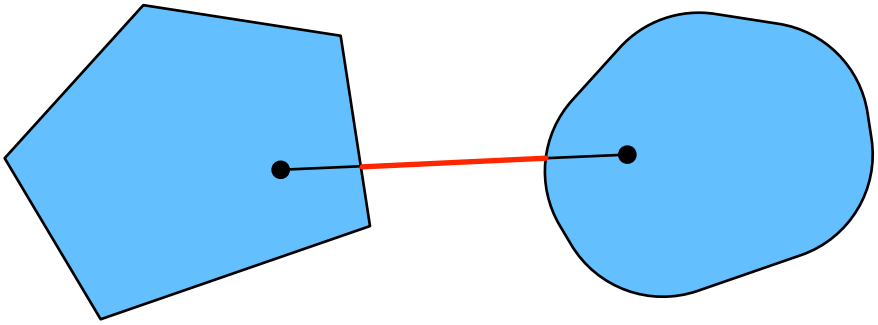


Fig. 12.2.3: The union of two convex sets need not be convex.

Thông thường các vấn đề trong học sâu được xác định trên các bộ lồi. Ví dụ, $\mathbb{R}^d$, tập hợp các vectơ d chiều của số thực, là một tập lồi (sau tất cả, đường giữa hai điểm bất kỳ trong $\mathbb{R}^d$ vẫn còn trong $\mathbb{R}^d$). Trong một số trường hợp, chúng tôi làm việc với các biến có độ dài giới hạn, chẳng hạn như các quả bóng bán kính r theo định nghĩa bởi $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\| \leq r\}$.

Hàm lồi

Bây giờ chúng ta có bộ lồi, chúng ta có thể giới thiệu các chức năng lồi * f. Cho một bộ lồi $\mathcal{X}$, một chức năng $f : \mathcal{X} \to \mathbb{R}$ là * lồi * nếu cho tất cả $x, x' \in \mathcal{X}$ và cho tất cả $\lambda \in [0, 1]$ chúng tôi có

$$\lambda f(x) + (1 - \lambda) f(x') \geq f(\lambda x + (1 - \lambda) x'). \tag{12.2.2}$$

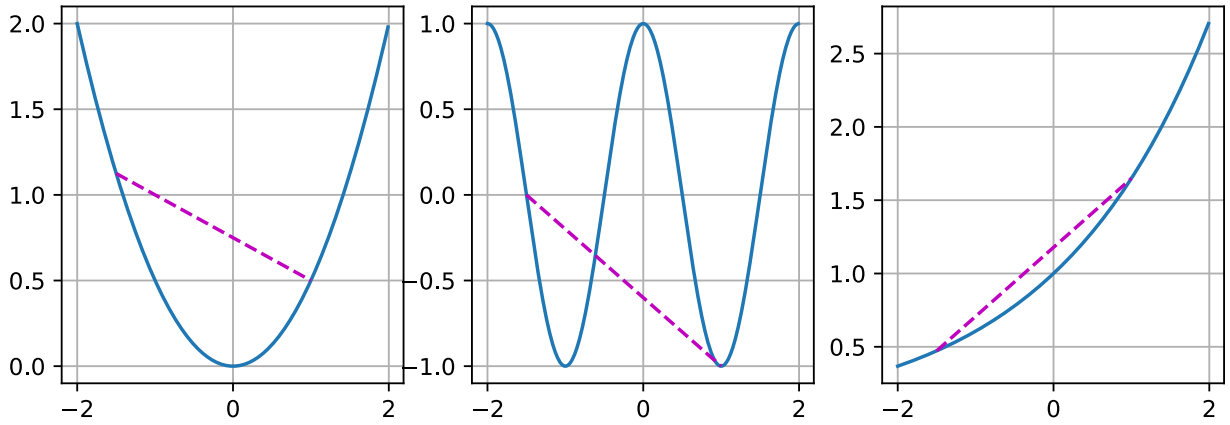
Để minh họa điều này, chúng ta hãy vẽ một vài chức năng và kiểm tra cái nào đáp ứng yêu cầu. Dưới đây chúng tôi xác định một vài hàm, cả lồi và không lồi.

```
f = lambda x: 0.5 * x**2  # Convex
g = lambda x: torch.cos(np.pi * x)  # Nonconvex
```

(continues on next page)

(continued from previous page)

```
h = lambda x: torch.exp(0.5 * x)  # Convex

x, segment = torch.arange(-2, 2, 0.01), torch.tensor([-1.5, 1])
d2l.use_svg_display()
_, axes = d2l.plt.subplots(1, 3, figsize=(9, 3))
for ax, func in zip(axes, [f, g, h]):
    d2l.plot([x, segment], [func(x), func(segment)], axes=ax)
```



Đúng như dự đoán, hàm cosin là *nonconvex*, trong khi parabol và hàm mũ là. Lưu ý rằng yêu cầu $\mathcal{X}$ là một bộ lồi là cần thiết để điều kiện có ý nghĩa. Nếu không, kết quả của $f(\lambda x + (1-\lambda)x')$ có thể không được xác định rõ.

Sự bất bình đẳng của Jensen

Với hàm lồi f, một trong những công cụ toán học hữu ích nhất là sự bất bình đẳng của *Jensen*. Nó lên tới một khái quát hóa định nghĩa về độ lồi:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ and } E_X[f(X)] \geq f\left(E_X[X]\right), \tag{12.2.3}$$

trong đó α_i là số thực không âm sao cho $\sum_i \alpha_i = 1$ và X là một biến ngẫu nhiên. Nói cách khác, kỳ vọng của một hàm lồi không kém hàm lồi của một kỳ vọng, trong đó hàm sau thường là một biểu thức đơn giản hơn. Để chứng minh sự bất bình đẳng đầu tiên, chúng tôi liên tục áp dụng định nghĩa lồi cho một thuật ngữ trong tổng tại một thời điểm.

Một trong những ứng dụng phổ biến của bất bình đẳng của Jensen là ràng buộc một biểu thức phức tạp hơn bằng một biểu thức đơn giản hơn. Ví dụ, ứng dụng của nó có thể liên quan đến khả năng đăng nhập của các biến ngẫu nhiên được quan sát một phần. Đó là, chúng tôi sử dụng

$$E_{Y \sim P(Y)}[-\log P(X \mid Y)] \geq -\log P(X), \tag{12.2.4}$$

kể từ $\int P(Y)P(X \mid Y)dY = P(X)$. Điều này có thể được sử dụng trong các phương pháp biến thể. Ở đây Y thường là biến ngẫu nhiên không quan sát được, $P(Y)$ là đoán tốt nhất về cách nó có thể được phân phối và $P(X)$ là phân phối với Y tích hợp ra. Ví dụ, trong phân nhóm Y có thể là nhãn cụm và $P(X \mid Y)$ là mô hình tạo khi áp dụng nhãn cụm.

12.2.2 Thuộc tính

Hàm lồi có nhiều tính chất hữu ích. Chúng tôi mô tả một vài cái được sử dụng phổ biến dưới đây.

Minima địa phương là Minima toàn cầu

Đầu tiên và quan trọng nhất, minima cục bộ của các hàm lồi cũng là minima toàn cầu. Chúng ta có thể chứng minh điều đó bằng mâu thuẫn như sau.

Hãy xem xét một hàm lồi f được xác định trên một bộ lồi $\mathcal{X}$. Giả sử rằng $x^* \in \mathcal{X}$ là mức tối thiểu cục bộ: tồn tại một giá trị dương nhỏ p sao cho $x \in \mathcal{X}$ thỏa mãn $0 < |x - x^*| \leq p$ chúng ta có $f(x^*) < f(x)$.

Giả sử rằng mức tối thiểu địa phương x^* không phải là minum um toàn cầu của f: có tồn tại $x' \in \mathcal{X}$ mà $f(x') < f(x^*)$. Cũng tồn tại $\lambda \in [0, 1)$ như $\lambda = 1 - \frac{p}{|x^* - x'|}$ sao cho $0 < |\lambda x^* + (1 - \lambda)x' - x^*| \leq p$.

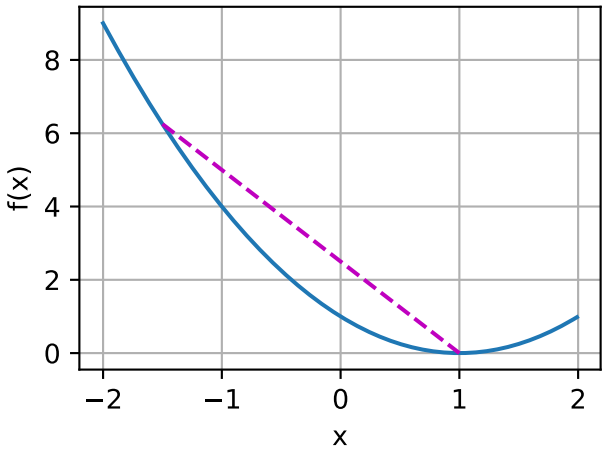
Tuy nhiên, theo định nghĩa của các hàm lồi, ta có

$$\begin{aligned} f(\lambda x^* + (1 - \lambda)x') &\leq \lambda f(x^*) + (1 - \lambda) f(x') \\ &< \lambda f(x^*) + (1 - \lambda) f(x^*) \\ &= f(x^*), \end{aligned} \tag{12.2.5}$$

mâu thuẫn với tuyên bố của chúng tôi rằng x^* là mức tối thiểu địa phương. Do đó, không tồn tại $x' \in \mathcal{X}$ mà $f(x') < f(x^*)$. Mức tối thiểu địa phương x^* cũng là mức tối thiểu toàn cầu.

Ví dụ, hàm lồi $f(x) = (x - 1)^2$ có mức tối thiểu cục bộ là $x = 1$, đây cũng là mức tối thiểu toàn cầu.

```
f = lambda x: (x - 1) ** 2
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```



Thực tế là minima cục bộ cho các chức năng lồi cũng là minima toàn cầu rất thuận tiện. Điều đó có nghĩa là nếu chúng ta giảm thiểu các chức năng, chúng ta không thể "gặp khó khăn". Tuy nhiên, lưu ý rằng điều này không có nghĩa là không thể có nhiều hơn một mức tối thiểu toàn cầu hoặc thậm chí có thể tồn tại. Ví dụ, hàm $f(x) = \max(|x| - 1, 0)$ đạt được giá trị tối thiểu của nó trong khoảng $[-1, 1]$. Ngược lại, chức năng $f(x) = \exp(x)$ không đạt được giá trị tối thiểu trên $\mathbb{R}$: đối với $x \to -\infty$ nó asymptotes đến 0, nhưng không có x mà $f(x) = 0$.

Các bộ hàm lồi dưới đây là lồi

Chúng tôi có thể thuận tiện xác định các bộ lồi thông qua * dưới bộ* của hàm lồi. Cụ thể, đưa ra một hàm lồi f được xác định trên một bộ lồi $\mathcal{X}$, bất kỳ thiết lập dưới đây

$$\mathcal{S}_b := \{x | x \in \mathcal{X} \text{ and } f(x) \leq b\} \tag{12.2.6}$$

là lồi.

Hãy để chúng tôi chứng minh điều này một cách nhanh chóng. Nhớ lại rằng đối với bất kỳ $x, x' \in \mathcal{S}_b$ chúng ta cần phải cho thấy rằng $\lambda x + (1 - \lambda)x' \in \mathcal{S}_b$ miễn là $\lambda \in [0, 1]$. Kể từ $f(x) \leq b$ và $f(x') \leq b$, theo định nghĩa về độ lồi chúng ta có

$$f(\lambda x + (1 - \lambda)x') \leq \lambda f(x) + (1 - \lambda) f(x') \leq b. \tag{12.2.7}$$

Lồi và dẫn xuất thứ hai

Bất cứ khi nào đạo hàm thứ hai của một hàm $f : \mathbb{R}^n \to \mathbb{R}$ tồn tại thì rất dễ dàng để kiểm tra xem f có lồi hay không. Tất cả những gì chúng ta cần làm là kiểm tra xem Hessian của f là semidefinite dương: $\nabla^2 f \succeq 0$, tức là, biểu thị ma trận Hessian $\nabla^2 f$ bởi $\mathbf{H}$, $\mathbf{x}^\top \mathbf{H} \mathbf{x} \geq 0$ cho tất cả $\mathbf{x} \in \mathbb{R}^n$. Ví dụ, hàm $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{x}\|^2$ là lồi từ $\nabla^2 f = \mathbf{1}$, tức là, Hessian của nó là một ma trận nhận dạng.

Về mặt chính thức, một hàm một chiều có thể phân biệt hai lần $f : \mathbb{R} \to \mathbb{R}$ là lồi nếu và chỉ khi đạo hàm thứ hai của nó $f'' \geq 0$. Đối với bất kỳ chức năng đa chiều hai lần khác biệt $f : \mathbb{R}^n \to \mathbb{R}$, nó là lồi nếu và chỉ khi Hessian $\nabla^2 f \succeq 0$ của nó.

Đầu tiên, chúng ta cần chứng minh trường hợp một chiều. Để thấy rằng lồi của f ngụ ý $f'' \geq 0$ chúng tôi sử dụng thực tế là

$$\frac{1}{2} f(x + \epsilon) + \frac{1}{2} f(x - \epsilon) \geq f\left(\frac{x + \epsilon}{2} + \frac{x - \epsilon}{2}\right) = f(x). \tag{12.2.8}$$

Vì đạo hàm thứ hai được đưa ra bởi giới hạn trên sự khác biệt hữu hạn nó sau đó

$$f''(x) = \lim_{\epsilon \to 0} \frac{f(x + \epsilon) + f(x - \epsilon) - 2f(x)}{\epsilon^2} \geq 0. \tag{12.2.9}$$

Để thấy rằng $f'' \geq 0$ ngụ ý rằng f là lồi, chúng tôi sử dụng thực tế là $f'' \geq 0$ ngụ ý rằng f' là một chức năng không giảm đơn điệu. Hãy để $a < x < b$ được ba điểm trong $\mathbb{R}$, nơi $x = (1 - \lambda)a + \lambda b$ và $\lambda \in (0, 1)$. Theo định lý giá trị trung bình, có tồn tại $\alpha \in [a, x]$ và $\beta \in [x, b]$ sao cho

$$f'(\alpha) = \frac{f(x) - f(a)}{x - a} \text{ and } f'(\beta) = \frac{f(b) - f(x)}{b - x}. \tag{12.2.10}$$

Bởi monotonicity $f'(\beta) \geq f'(\alpha)$, do đó

$$\frac{x - a}{b - a} f(b) + \frac{b - x}{b - a} f(a) \geq f(x). \tag{12.2.11}$$

Kể từ khi $x = (1 - \lambda)a + \lambda b$, chúng tôi có

$$\lambda f(b) + (1 - \lambda) f(a) \geq f((1 - \lambda)a + \lambda b), \tag{12.2.12}$$

thy như vậy proving chứng minh lồi.

Thứ hai, chúng ta cần một đề tài trước khi chứng minh trường hợp đa chiều: $f : \mathbb{R}^n \to \mathbb{R}$ là lồi nếu và chỉ khi cho tất cả $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$

$$g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y}) \text{ where } z \in [0,1] \tag{12.2.13}$$

là lồi.

Để chứng minh rằng độ lồi của f ngụ ý rằng g là lồi, chúng ta có thể chỉ ra rằng đối với tất cả $a, b, \lambda \in [0,1]$ (do đó $0 \leq \lambda a + (1-\lambda)b \leq 1$)

$$\begin{aligned}
&g(\lambda a + (1-\lambda)b)\\
=&f\left((\lambda a + (1-\lambda)b)\,\mathbf{x} + (1-\lambda a - (1-\lambda)b)\,\mathbf{y}\right)\\
=&f\left(\lambda\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda)\left(b\mathbf{x} + (1-b)\mathbf{y}\right)\right)\\
\leq&\lambda f\left(a\mathbf{x} + (1-a)\mathbf{y}\right) + (1-\lambda) f\left(b\mathbf{x} + (1-b)\mathbf{y}\right)\\
=&\lambda g(a) + (1-\lambda) g(b).
\end{aligned} \tag{12.2.14}$$

Để chứng minh cuộc trò chuyện, chúng ta có thể chỉ ra rằng cho tất cả $\lambda \in [0,1]$

$$\begin{aligned}
&f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y})\\
=&g(\lambda \cdot 1 + (1-\lambda)\cdot 0)\\
\leq&\lambda g(1) + (1-\lambda) g(0)\\
=&\lambda f(\mathbf{x}) + (1-\lambda) g(\mathbf{y}).
\end{aligned} \tag{12.2.15}$$

Cuối cùng, sử dụng lemma ở trên và kết quả của trường hợp một chiều, trường hợp đa chiều có thể được chứng minh như sau. Một chức năng đa chiều $f : \mathbb{R}^n \to \mathbb{R}$ là lồi nếu và chỉ khi cho tất cả $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ $g(z) \stackrel{\text{def}}{=} f(z\mathbf{x} + (1-z)\mathbf{y})$, trong đó $z \in [0,1]$, là lồi. Theo trường hợp một chiều, điều này giữ nếu và chỉ khi $g'' = (\mathbf{x}-\mathbf{y})^\top \mathbf{H}(\mathbf{x}-\mathbf{y}) \geq 0$ ($\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f$) cho tất cả $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, tương đương với $\mathbf{H} \succeq 0$ trên mỗi định nghĩa của ma trận bán xác định dương.

12.2.3 Ràng buộc

Một trong những tính chất tốt đẹp của tối ưu hóa lồi là nó cho phép chúng ta xử lý các ràng buộc một cách hiệu quả. Đó là, nó cho phép chúng tôi giải quyết các vấn đề tối ưu hóa hạn chế* của biểu mẫu:

$$\begin{aligned}
&\underset{\mathbf{x}}{\text{minimize}}\ f(\mathbf{x})\\
&\text{subject to } c_i(\mathbf{x}) \leq 0 \text{ for all } i \in \{1, \ldots, n\},
\end{aligned} \tag{12.2.16}$$

trong đó f là mục tiêu và các chức năng c_i là các hàm hạn chế. Để xem điều này xem xét những gì trường hợp $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$. Trong trường hợp này, các thông số $\mathbf{x}$ được hạn chế vào quả bóng đơn vị. Nếu một ràng buộc thứ hai là $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$, thì điều này tương ứng với tất cả $\mathbf{x}$ nằm trên một nửa không gian. Thỏa mãn cả hai ràng buộc đồng thời số tiền để lựa chọn một lát của một quả bóng.

Lagrangian

Nói chung, việc giải quyết một vấn đề tối ưu hóa bị hạn chế là khó khăn. Một cách để giải quyết nó bắt nguồn từ vật lý với một trực giác khá đơn giản. Hãy tưởng tượng một quả bóng bên trong một hộp. Quả bóng sẽ lăn đến nơi thấp nhất và lực hấp dẫn sẽ được cân bằng với các lực mà các cạnh của hộp có thể áp đặt lên quả bóng. Nói tóm lại, gradient của hàm khách quan (tức là trọng lực) sẽ được bù đắp bởi gradient của hàm ràng buộc (quả bóng cần phải ở lại bên trong hộp nhờ đức hạnh của các bức tường "đẩy lùi"). Lưu ý rằng một số

hạn chế có thể không hoạt động: các bức tường không được chạm vào bởi quả bóng sẽ không thể tác dụng bất kỳ lực nào lên quả bóng.

Bỏ qua nguồn gốc của * Lagrangian* L, lý luận trên có thể được thể hiện thông qua bài toán tối ưu hóa điểm yên sau:

$$L(\mathbf{x}, \alpha_1, \ldots, \alpha_n) = f(\mathbf{x}) + \sum_{i=1}^{n} \alpha_i c_i(\mathbf{x}) \text{ where } \alpha_i \geq 0. \tag{12.2.17}$$

Ở đây các biến α_i ($i = 1, \ldots, n$) là cái gọi là hệ số nhân * Lagrange* đảm bảo rằng các ràng buộc được thực thi đúng cách. Chúng được chọn vừa đủ lớn để đảm bảo rằng $c_i(\mathbf{x}) \leq 0$ cho tất cả i. Ví dụ, đối với bất kỳ $\mathbf{x}$ nơi $c_i(\mathbf{x}) < 0$ một cách tự nhiên, chúng tôi sẽ chọn $\alpha_i = 0$. Hơn nữa, đây là một vấn đề tối ưu hóa điểm yên ngựa mà người ta muốn * tối đa hóa* L đối với tất cả α_i và đồng thời * giảm xỉnh* nó đối với $\mathbf{x}$. Có một cơ thể văn học phong phú giải thích làm thế nào để đến chức năng $L(\mathbf{x}, \alpha_1, \ldots, \alpha_n)$. Đối với mục đích của chúng tôi, nó là đủ để biết rằng điểm yên của L là nơi mà vấn đề tối ưu hóa hạn chế ban đầu được giải quyết một cách tối ưu.

Hình phạt

Một cách để đáp ứng các vấn đề tối ưu hóa hạn chế ít nhất * gần đúng * là thích ứng với Lagrangian L. Thay vì thỏa mãn $c_i(\mathbf{x}) \leq 0$, chúng tôi chỉ cần thêm $\alpha_i c_i(\mathbf{x})$ vào hàm khách quan $f(x)$. Điều này đảm bảo rằng các ràng buộc sẽ không bị vi phạm quá nặng.

Trong thực tế, chúng tôi đã sử dụng thủ thuật này tất cả cùng. Xem xét phân rã cân nặng năm Section 5.5. Trong đó, chúng tôi thêm $\frac{\lambda}{2}\|\mathbf{w}\|^2$ vào chức năng khách quan để đảm bảo rằng $\mathbf{w}$ không phát triển quá lớn. Từ quan điểm tối ưu hóa bị hạn chế, chúng ta có thể thấy rằng điều này sẽ đảm bảo rằng $\|\mathbf{w}\|^2 - r^2 \leq 0$ cho một số bán kính r. Điều chỉnh giá trị của λ cho phép chúng tôi thay đổi kích thước của $\mathbf{w}$.

Nói chung, thêm hình phạt là một cách tốt để đảm bảo sự hài lòng hạn chế gần đúng. Trong thực tế, điều này hóa ra mạnh mẽ hơn nhiều so với sự hài lòng chính xác. Hơn nữa, đối với các vấn đề không lồi, nhiều thuộc tính làm cho cách tiếp cận chính xác trở nên hấp dẫn trong trường hợp lồi (ví dụ, tối ưu) không còn giữ được nữa.

Dự báo

Một chiến lược thay thế để thỏa mãn các ràng buộc là dự đoán. Một lần nữa, chúng tôi đã gặp chúng trước đây, ví dụ, khi xử lý cắt gradient trong Section 9.5. Ở đó chúng tôi đảm bảo rằng một gradient có chiều dài giới hạn bởi θ qua

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min(1, \theta/\|\mathbf{g}\|). \tag{12.2.18}$$

Điều này hóa ra là một * dự áo* của $\mathbf{g}$ lên quả bóng bán kính θ. Nói chung hơn, một phép chiếu trên một bộ lồi $\mathcal{X}$ được định nghĩa là

$$\text{Proj}_{\mathcal{X}}(\mathbf{x}) = \underset{\mathbf{x}' \in \mathcal{X}}{\text{argmin}} \|\mathbf{x} - \mathbf{x}'\|, \tag{12.2.19}$$

đó là điểm gần nhất trong $\mathcal{X}$ đến $\mathbf{x}$.

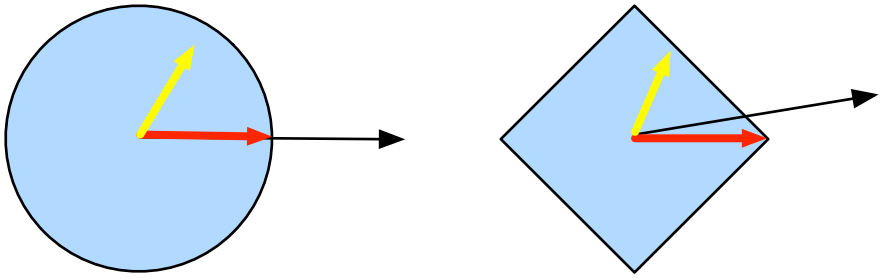


Fig. 12.2.4: Convex Projections.

Định nghĩa toán học của các dự báo có vẻ hơi trừu tượng. Fig. 12.2.4 giải thích nó một chút rõ ràng hơn. Trong đó chúng ta có hai bộ lồi, một vòng tròn và một viên kim cương. Các điểm bên trong cả hai bộ (màu vàng) vẫn không thay đổi trong quá trình chiếu. Các điểm bên ngoài cả hai bộ (màu đen) được chiếu vào các điểm bên trong các bộ (màu đỏ) là tủ quần áo đến các điểm ban đầu (màu đen). Trong khi đối với L_2 quả bóng này để lại hướng không thay đổi, điều này không cần phải là trường hợp nói chung, như có thể thấy trong trường hợp của kim cương.

Một trong những công dụng cho các phép chiếu lồi là tính toán các vectơ trọng lượng thưa thớt. Trong trường hợp này, chúng tôi dự án vectơ trọng lượng lên một quả bóng L_1, là một phiên bản tổng quát của vỏ kim cương trong Fig. 12.2.4.

12.2.4 Tóm tắt

Trong bối cảnh học sâu, mục đích chính của các hàm lồi là thúc đẩy các thuật toán tối ưu hóa và giúp chúng ta hiểu chi tiết chúng. Trong phần sau đây, chúng ta sẽ thấy độ dốc và gốc gradient ngẫu nhiên có thể được bắt nguồn cho phù hợp như thế nào.

- Giao điểm của bộ lồi lồi là lồi. Đoàn thể thì không.
- Kỳ vọng của một hàm lồi không kém hàm lồi của một kỳ vọng (bất bình đẳng của Jensen).
- Một hàm phân biệt hai lần là lồi nếu và chỉ khi Hessian của nó (một ma trận của các dẫn xuất thứ hai) là semidefinite dương.
- Ràng buộc lồi có thể được thêm vào thông qua Lagrangian. Trong thực tế, chúng ta có thể chỉ cần thêm chúng với một hình phạt cho chức năng khách quan.
- Các hình chiếu ánh xạ đến các điểm trong bộ lồi gần nhất với các điểm ban đầu.

12.2.5 Bài tập

1. Giả sử rằng chúng ta muốn xác minh độ lồi của một tập hợp bằng cách vẽ tất cả các đường giữa các điểm trong tập hợp và kiểm tra xem các dòng có chứa hay không.
    1. Chứng minh rằng nó là đủ để chỉ kiểm tra các điểm trên ranh giới.
    2. Chứng minh rằng nó là đủ để chỉ kiểm tra các đỉnh của tập hợp.
2. Biểu thị bằng $\mathcal{B}_p[r] \stackrel{\text{def}}{=} \{\mathbf{x}|\mathbf{x} \in \mathbb{R}^d \text{ and } \|\mathbf{x}\|_p \leq r\}$ bóng bán kính r bằng cách sử dụng p-chuẩn. Chứng minh rằng $\mathcal{B}_p[r]$ là lồi cho tất cả $p \geq 1$.
3. Cho hàm lồi f và g, cho thấy $\max(f, g)$ cũng lồi. Chứng minh rằng $\min(f, g)$ không lồi.

4. Chứng minh rằng việc bình thường hóa chức năng softmax là lồi. Cụ thể hơn chứng minh sự lồi của $f(x) = \log \sum_i \exp(x_i)$.
5. Chứng minh rằng không gian con tuyến tính, tức là, $\mathcal{X} = \{\mathbf{x}|\mathbf{W}\mathbf{x} = \mathbf{b}\}$, là các bộ lồi.
6. Chứng minh rằng trong trường hợp không gian con tuyến tính với $\mathbf{b} = \mathbf{0}$ phép chiếu $\text{Proj}_{\mathcal{X}}$ có thể được viết là $\mathbf{M}\mathbf{x}$ cho một số ma trận $\mathbf{M}$.
7. Cho thấy rằng đối với các hàm lồi hai lần khác biệt f chúng ta có thể viết $f(x+\epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x+\xi)$ cho một số $\xi \in [0, \epsilon]$.
8. Đưa ra một vector $\mathbf{w} \in \mathbb{R}^d$ với $\|\mathbf{w}\|_1 > 1$ tính toán phép chiếu trên quả bóng đơn vị L_1.
    1. Như một bước trung gian viết ra mục tiêu bị phạt $\|\mathbf{w} - \mathbf{w}'\|^2 + \lambda\|\mathbf{w}'\|_1$ và tính toán các giải pháp cho một $\lambda > 0$ nhất định.
    2. Bạn có thể tìm thấy giá trị "đúng" của λ mà không có nhiều thử và sai không?
9. Cho một bộ lồi $\mathcal{X}$ và hai vectơ $\mathbf{x}$ và $\mathbf{y}$, chứng minh rằng các dự báo không bao giờ tăng khoảng cách, tức là $\|\mathbf{x} - \mathbf{y}\| \geq \|\text{Proj}_{\mathcal{X}}(\mathbf{x}) - \text{Proj}_{\mathcal{X}}(\mathbf{y})\|$.

Discussions[127]

12.3 Gradient Descent

Trong phần này, chúng ta sẽ giới thiệu các khái niệm cơ bản bên dưới * gradient descent*. Mặc dù nó hiếm khi được sử dụng trực tiếp trong học sâu, một sự hiểu biết về gradient descent là chìa khóa để hiểu các thuật toán gốc gradient ngẫu nhiên. Ví dụ, vấn đề tối ưu hóa có thể phân kỳ do tốc độ học tập quá lớn. Hiện tượng này đã có thể được nhìn thấy trong gradient gốc. Tương tự như vậy, preconditioning là một kỹ thuật phổ biến trong chuyển đổi gradient và mang đến các thuật toán tiên tiến hơn. Hãy để chúng tôi bắt đầu với một trường hợp đặc biệt đơn giản.

12.3.1 Một chiều Gradient Descent

Gradient gốc trong một chiều là một ví dụ tuyệt vời để giải thích lý do tại sao thuật toán gốc gradient có thể làm giảm giá trị của hàm khách quan. Xem xét một số chức năng có giá trị thực liên tục khác biệt $f : \mathbb{R} \to \mathbb{R}$. Sử dụng bản mở rộng Taylor, chúng tôi có được

$$f(x+\epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \tag{12.3.1}$$

Đó là, trong xấp xỉ thứ tự thứ nhất $f(x+\epsilon)$ được đưa ra bởi giá trị hàm $f(x)$ và đạo hàm đầu tiên $f'(x)$ tại x. Nó không phải là không hợp lý để giả định rằng đối với nhỏ ϵ di chuyển theo hướng của gradient âm sẽ giảm f. Để giữ cho mọi thứ đơn giản, chúng tôi chọn một kích thước bước cố định $\eta > 0$ và chọn $\epsilon = -\eta f'(x)$. Cắm điều này vào bản mở rộng Taylor ở trên chúng tôi nhận được

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \tag{12.3.2}$$

Nếu đạo hàm $f'(x) \neq 0$ không biến mất chúng ta tiến bộ kể từ $\eta f'^2(x) > 0$. Hơn nữa, chúng tôi luôn có thể chọn η đủ nhỏ để các điều khoản bậc cao hơn trở nên không liên quan. Do đó chúng tôi đến

$$f(x - \eta f'(x)) \lessapprox f(x). \tag{12.3.3}$$

[127] https://discuss.d2l.ai/t/350

Điều này có nghĩa rằng, nếu chúng ta sử dụng

$$x \leftarrow x - \eta f'(x) \tag{12.3.4}$$

để lặp lại x, giá trị của hàm $f(x)$ có thể giảm. Do đó, trong gradient descent đầu tiên chúng ta chọn một giá trị ban đầu x và một hằng số $\eta > 0$ và sau đó sử dụng chúng để liên tục lặp lại x cho đến khi đạt đến điều kiện dừng, ví dụ, khi độ lớn của gradient $|f'(x)|$ là đủ nhỏ hoặc số lần lặp đã đạt đến một số nhất định giá trị.

Để đơn giản, chúng tôi chọn hàm mục tiêu $f(x) = x^2$ để minh họa cách thực hiện gradient descent. Mặc dù chúng ta biết rằng $x = 0$ là giải pháp để giảm thiểu $f(x)$, chúng tôi vẫn sử dụng chức năng đơn giản này để quan sát cách x thay đổi.

```
%matplotlib inline
import numpy as np
import torch
from d2l import torch as d2l
```

```
def f(x):  # Objective function
    return x ** 2

def f_grad(x):  # Gradient (derivative) of the objective function
    return 2 * x
```

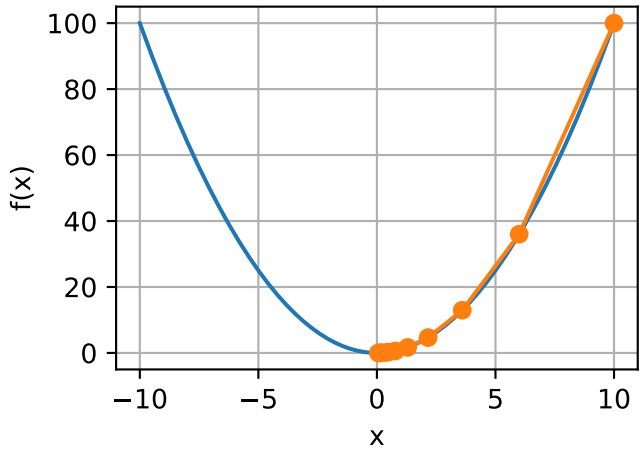
Tiếp theo, chúng tôi sử dụng $x = 10$ làm giá trị ban đầu và giả sử $\eta = 0.2$. Sử dụng gradient descent để lặp lại x trong 10 lần chúng ta có thể thấy rằng, cuối cùng, giá trị của x tiếp cận giải pháp tối ưu.

```
def gd(eta, f_grad):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'epoch 10, x: {x:f}')
    return results

results = gd(0.2, f_grad)
```

```
epoch 10, x: 0.060466
```

Tiến trình tối ưu hóa hơn x có thể được vẽ như sau.

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = torch.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [[f(x) for x in f_line], [
        f(x) for x in results]], 'x', 'f(x)', fmts=['-', '-o'])

show_trace(results, f)
```

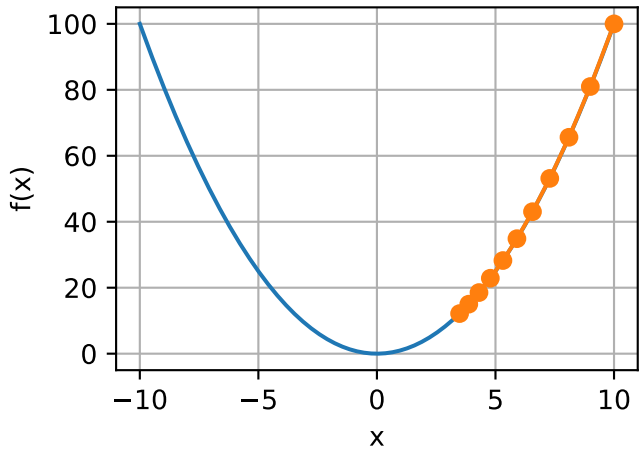


Tỷ lệ học tập

Tỷ lệ học tập η có thể được thiết lập bởi nhà thiết kế thuật toán. Nếu chúng ta sử dụng tốc độ học tập quá nhỏ, nó sẽ khiến x cập nhật rất chậm, đòi hỏi nhiều lần lặp lại hơn để có được giải pháp tốt hơn. Để hiển thị những gì xảy ra trong trường hợp như vậy, hãy xem xét tiến trình trong cùng một vấn đề tối ưu hóa cho $\eta = 0.05$. Như chúng ta có thể thấy, ngay cả sau 10 bước, chúng ta vẫn còn rất xa giải pháp tối ưu.

```
show_trace(gd(0.05, f_grad), f)
```

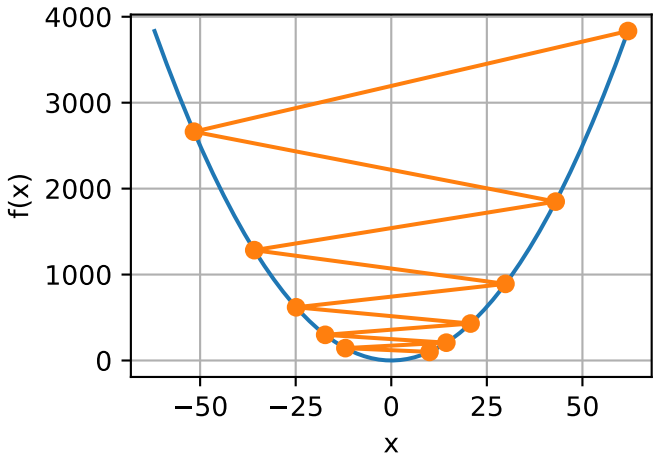
```
epoch 10, x: 3.486784
```



Ngược lại, nếu chúng ta sử dụng tỷ lệ học tập quá cao, $|\eta f'(x)|$ có thể quá lớn đối với công thức mở rộng Taylor bậc nhất. Đó là, thuật ngữ $\mathcal{O}(\eta^2 f'^2(x))$ trong (12.3.2) có thể trở nên quan trọng. Trong trường hợp này, chúng tôi không thể đảm bảo rằng việc lặp lại x sẽ có thể hạ giá trị $f(x)$. Ví dụ, khi chúng ta đặt tỷ lệ học tập thành $\eta = 1.1$, x vượt qua giải pháp tối ưu $x = 0$ và dần dần phân kỳ.

```
show_trace(gd(1.1, f_grad), f)
```

```
epoch 10, x: 61.917364
```

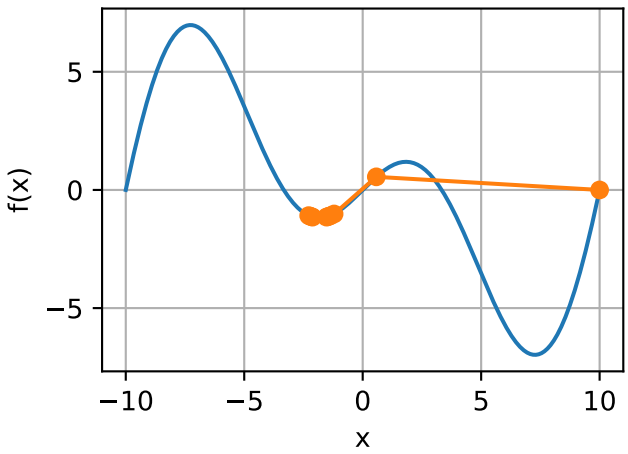


Minima địa phương

Để minh họa những gì xảy ra cho các chức năng không lồi xem xét trường hợp của $f(x) = x \cdot \cos(cx)$ cho một số hằng số c. Chức năng này có vô hạn nhiều minima cục bộ. Tùy thuộc vào sự lựa chọn của chúng tôi về tỷ lệ học tập và tùy thuộc vào mức độ điều kiện của vấn đề, chúng tôi có thể kết thúc với một trong nhiều giải pháp. Ví dụ dưới đây minh họa mức độ học tập cao (không thực tế) sẽ dẫn đến mức tối thiểu địa phương kém.

```
c = torch.tensor(0.15 * np.pi)

def f(x):  # Objective function
    return x * torch.cos(c * x)

def f_grad(x):  # Gradient of the objective function
    return torch.cos(c * x) - c * x * torch.sin(c * x)

show_trace(gd(2, f_grad), f)
```

```
epoch 10, x: -1.528166
```



12.3.2 Đa biến Gradient Descent

Bây giờ chúng ta có một trực giác tốt hơn về trường hợp thống nhất, chúng ta hãy xem xét tình huống mà $\mathbf{x} = [x_1, x_2, \ldots, x_d]^\top$. Đó là, hàm khách quan $f : \mathbb{R}^d \to \mathbb{R}$ ánh xạ vectơ thành vô hướng. Tương ứng gradient của nó là đa biến, quá. Nó là một vectơ gồm d dẫn xuất từng phần:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \ldots, \frac{\partial f(\mathbf{x})}{\partial x_d}\right]^\top. \tag{12.3.5}$$

Mỗi phần tử phái sinh một phần $\partial f(\mathbf{x})/\partial x_i$ trong gradient cho biết tốc độ thay đổi của f tại $\mathbf{x}$ đối với đầu vào x_i. Như trước đây trong trường hợp thống nhất, chúng ta có thể sử dụng xấp xỉ Taylor tương ứng cho các chức năng đa biến để có được một số ý tưởng về những gì chúng ta nên làm. Đặc biệt, chúng tôi có điều đó

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\boldsymbol{\epsilon}\|^2). \tag{12.3.6}$$

Nói cách khác, lên đến các thuật ngữ bậc hai trong $\boldsymbol{\epsilon}$ hướng xuống dốc nhất được đưa ra bởi gradient âm $-\nabla f(\mathbf{x})$. Chọn một tỷ lệ học tập phù hợp $\eta > 0$ mang lại thuật toán gốc gradient nguyên mẫu:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \tag{12.3.7}$$

Để xem cách thuật toán hoạt động trong thực tế chúng ta hãy xây dựng một hàm khách quan $f(\mathbf{x}) = x_1^2 + 2x_2^2$ với một vector hai chiều $\mathbf{x} = [x_1, x_2]^\top$ như đầu vào và vô hướng như đầu ra. Gradient được đưa ra bởi $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$. Chúng ta sẽ quan sát quỹ đạo của $\mathbf{x}$ bằng cách giảm gradient từ vị trí ban đầu $[-5, -2]$.

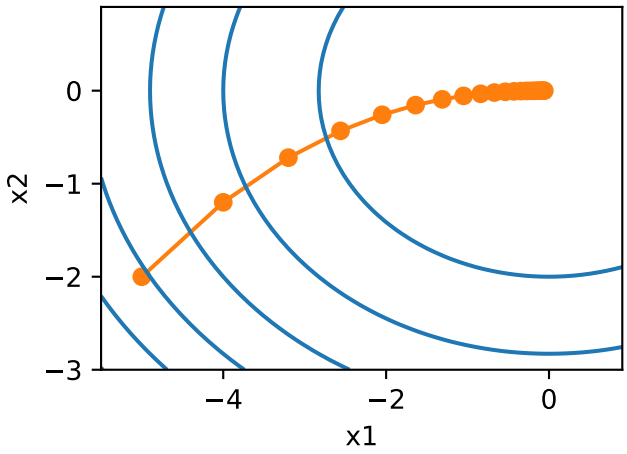
Để bắt đầu, chúng ta cần thêm hai chức năng trợ giúp. Đầu tiên sử dụng chức năng cập nhật và áp dụng nó 20 lần cho giá trị ban đầu. Người trợ giúp thứ hai hình dung quỹ đạo của $\mathbf{x}$.

```
def train_2d(trainer, steps=20, f_grad=None):  #@save
    """Optimize a 2D objective function with a customized trainer."""
    # `s1` and `s2` are internal state variables that will be used later
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        if f_grad:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad)
        else:
            x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    print(f'epoch {i + 1}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results

def show_trace_2d(f, results):  #@save
    """Show the trace of 2D variables during optimization."""
    d2l.set_figsize()
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = torch.meshgrid(torch.arange(-5.5, 1.0, 0.1),
                            torch.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

Tiếp theo, chúng ta quan sát quỹ đạo của biến tối ưu hóa $\mathbf{x}$ cho tỷ lệ học tập $\eta = 0.1$. Chúng ta có thể thấy rằng sau 20 bước, giá trị của $\mathbf{x}$ đạt đến mức tối thiểu của nó ở mức $[0, 0]$. Tiến bộ khá cư xử tốt mặc dù khá chậm.

```
def f_2d(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_2d_grad(x1, x2):  # Gradient of the objective function
    return (2 * x1, 4 * x2)

def gd_2d(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)

eta = 0.1
show_trace_2d(f_2d, train_2d(gd_2d, f_grad=f_2d_grad))
```

```
epoch 20, x1: -0.057646, x2: -0.000073
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



12.3.3 Phương pháp thích ứng

Như chúng ta có thể thấy trong Section 12.3.1, nhận được tỷ lệ học tập η "vừa phải" là khó khăn. Nếu chúng ta chọn nó quá nhỏ, chúng ta sẽ tiến bộ rất ít. Nếu chúng ta chọn nó quá lớn, giải pháp dao động và trong trường hợp xấu nhất nó thậm chí có thể phân kỳ. Điều gì sẽ xảy ra nếu chúng ta có thể xác định η tự động hoặc thoát khỏi việc phải chọn một tỷ lệ học tập ở tất cả? Các phương thức thứ hai không chỉ nhìn vào giá trị và gradient của hàm mục tiêu mà còn ở độ cong * của nócó thể giúp ích trong trường hợp này. Mặc dù các phương pháp này không thể áp dụng trực tiếp vào deep learning do chi phí tính toán, chúng cung cấp trực giác hữu ích về cách thiết kế các thuật toán tối ưu hóa nâng cao bắt chước nhiều thuộc tính mong muốn của các thuật toán được nêu dưới đây.

Phương pháp Newton

Xem xét việc mở rộng Taylor của một số chức năng $f : \mathbb{R}^d \to \mathbb{R}$ không cần phải dừng lại sau nhiệm kỳ đầu tiên. Trong thực tế, chúng ta có thể viết nó như

$$f(\mathbf{x} + \boldsymbol{\epsilon}) = f(\mathbf{x}) + \boldsymbol{\epsilon}^\top \nabla f(\mathbf{x}) + \frac{1}{2}\boldsymbol{\epsilon}^\top \nabla^2 f(\mathbf{x})\boldsymbol{\epsilon} + \mathcal{O}(\|\boldsymbol{\epsilon}\|^3). \tag{12.3.8}$$

Để tránh ký hiệu rườm rà, chúng tôi định nghĩa $\mathbf{H} \stackrel{\text{def}}{=} \nabla^2 f(\mathbf{x})$ là Hessian của f, là ma trận $d \times d$. Đối với d nhỏ và các vấn đề đơn giản $\mathbf{H}$ rất dễ tính toán. Đối với các mạng thần kinh sâu, mặt khác, $\mathbf{H}$ có thể rất lớn, do chi phí lưu trữ $\mathcal{O}(d^2)$ mục. Hơn nữa nó có thể quá tốn kém để tính toán thông qua backpropagation. Bây giờ chúng ta hãy bỏ qua những cân nhắc như vậy và nhìn vào thuật toán nào chúng ta sẽ nhận được.

Rốt cuộc, tối thiểu f thỏa mãn $\nabla f = 0$. Tuân theo các quy tắc giải tích trong Section 3.4.3, bằng cách dùng các dẫn xuất của (12.3.8) liên quan đến $\boldsymbol{\epsilon}$ và bỏ qua các điều khoản bậc cao hơn, chúng tôi đến

$$\nabla f(\mathbf{x}) + \mathbf{H}\boldsymbol{\epsilon} = 0 \text{ and hence } \boldsymbol{\epsilon} = -\mathbf{H}^{-1}\nabla f(\mathbf{x}). \tag{12.3.9}$$

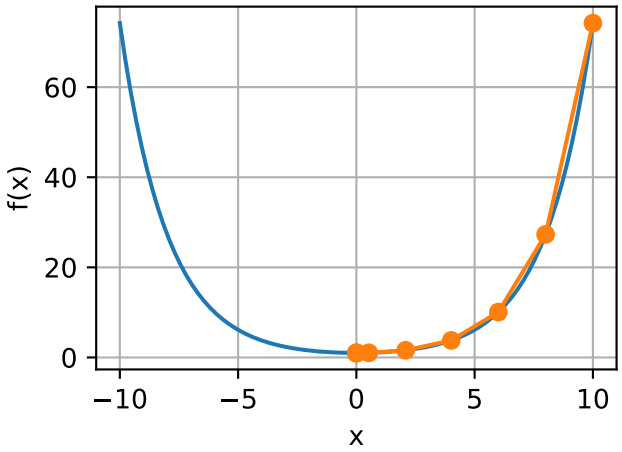
Đó là, chúng ta cần đảo ngược Hessian $\mathbf{H}$ như một phần của vấn đề tối ưu hóa.

Như một ví dụ đơn giản, đối với $f(x) = \frac{1}{2}x^2$, chúng tôi có $\nabla f(x) = x$ và $\mathbf{H} = 1$. Do đó đối với bất kỳ x chúng tôi có được $\epsilon = -x$. Nói cách khác, bước * single* là đủ để hội tụ hoàn hảo mà không cần bất kỳ điều chỉnh nào! Than ôi, chúng tôi đã có một chút may mắn ở đây: bản mở rộng Taylor là chính xác kể từ $f(x + \epsilon) = \frac{1}{2}x^2 + \epsilon x + \frac{1}{2}\epsilon^2$.

Hãy để chúng tôi xem những gì xảy ra trong các vấn đề khác. Với hàm cosin hyperbol lồi $f(x) = \cosh(cx)$ cho một số hằng số c, chúng ta có thể thấy rằng mức tối thiểu toàn cầu tại $x = 0$ đạt được sau một vài lần lặp lại.

```
c = torch.tensor(0.5)

def f(x):  # Objective function
    return torch.cosh(c * x)

def f_grad(x):  # Gradient of the objective function
    return c * torch.sinh(c * x)

def f_hess(x):  # Hessian of the objective function
    return c**2 * torch.cosh(c * x)

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results

show_trace(newton(), f)
```

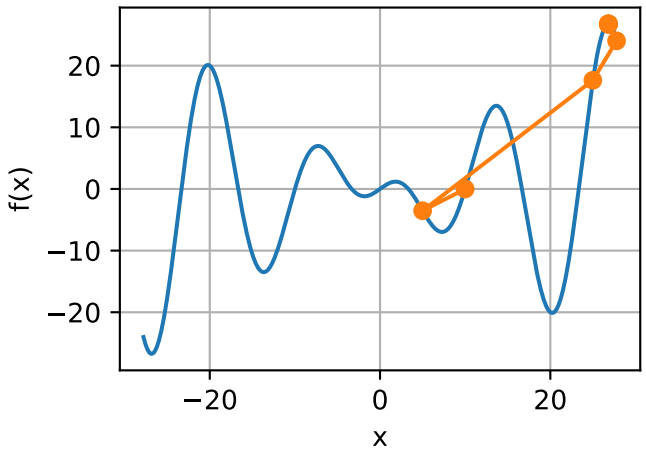
```
epoch 10, x: tensor(0.)
```



Bây giờ chúng ta hãy xem xét một hàm *nonconvex*, chẳng hạn như $f(x) = x\cos(cx)$ cho một số hằng số c. Rốt cuộc, lưu ý rằng trong phương pháp của Newton, chúng ta sẽ chia cho Hessian. Điều này có nghĩa là nếu đạo hàm thứ hai là *tiêu cực* chúng ta có thể đi theo hướng * tăng* giá trị của f. Đó là một lỗ hổng gây tử vong của thuật toán. Hãy để chúng tôi xem những gì xảy ra trong thực tế.

```
c = torch.tensor(0.15 * np.pi)

def f(x):  # Objective function
    return x * torch.cos(c * x)

def f_grad(x):  # Gradient of the objective function
    return torch.cos(c * x) - c * x * torch.sin(c * x)

def f_hess(x):  # Hessian of the objective function
    return - 2 * c * torch.sin(c * x) - x * c**2 * torch.cos(c * x)

show_trace(newton(), f)
```

```
epoch 10, x: tensor(26.8341)
```

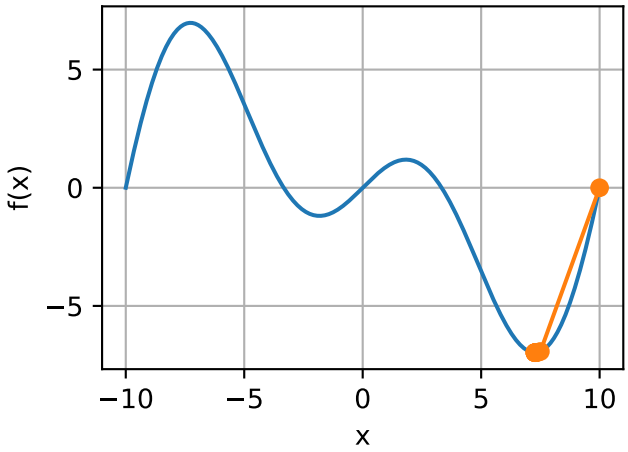


Điều này đã đi sai một cách ngoạn mục. Làm thế nào chúng ta có thể sửa nó? Một cách sẽ là "sửa chữa" Hessian bằng cách lấy giá trị tuyệt đối của nó thay thế. Một chiến lược khác là mang lại tốc độ học tập. Điều

này dường như đánh bại mục đích, nhưng không hoàn toàn. Có thông tin thứ hai cho phép chúng ta thận trọng bất cứ khi nào độ cong lớn và thực hiện các bước lâu hơn bất cứ khi nào chức năng khách quan phẳng hơn. Hãy để chúng tôi xem cách điều này hoạt động với tốc độ học tập nhỏ hơn một chút, nói $\eta = 0.5$. Như chúng ta có thể thấy, chúng ta có một thuật toán khá hiệu quả.

```
show_trace(newton(0.5), f)
```

```
epoch 10, x: tensor(7.2699)
```



Phân tích hội tụ

Chúng tôi chỉ phân tích tốc độ hội tụ của phương pháp Newton đối với một số hàm khách quan lồi và ba lần khác biệt f, trong đó đạo hàm thứ hai là nonzero, tức là $f'' > 0$. Bằng chứng đa biến là một phần mở rộng đơn giản của đối số một chiều bên dưới và bỏ qua vì nó không giúp chúng ta nhiều về trực giác.

Biểu thị bằng $x^{(k)}$ giá trị của x tại lần lặp k^{th} và để $e^{(k)} \stackrel{\text{def}}{=} x^{(k)} - x^*$ là khoảng cách từ tối ưu tại lần lặp k^{th}. Bằng cách mở rộng Taylor, chúng tôi có rằng điều kiện $f'(x^*) = 0$ có thể được viết là

$$0 = f'(x^{(k)} - e^{(k)}) = f'(x^{(k)}) - e^{(k)} f''(x^{(k)}) + \frac{1}{2}(e^{(k)})^2 f'''(\xi^{(k)}), \tag{12.3.10}$$

which mà holds giữ for some $\xi^{(k)} \in [x^{(k)} - e^{(k)}, x^{(k)}]$. Chia việc mở rộng trên cho sản lượng $f''(x^{(k)})$

$$e^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} = \frac{1}{2}(e^{(k)})^2 \frac{f'''(\xi^{(k)})}{f''(x^{(k)})}. \tag{12.3.11}$$

Nhớ lại rằng chúng tôi có bản cập nhật $x^{(k+1)} = x^{(k)} - f'(x^{(k)})/f''(x^{(k)})$. Cắm vào phương trình cập nhật này và lấy giá trị tuyệt đối của cả hai bên, chúng ta có

$$\left|e^{(k+1)}\right| = \frac{1}{2}(e^{(k)})^2 \frac{\left|f'''(\xi^{(k)})\right|}{f''(x^{(k)})}. \tag{12.3.12}$$

Do đó, bất cứ khi nào chúng tôi đang ở trong một khu vực có giới hạn $\left|f'''(\xi^{(k)})\right| / (2f''(x^{(k)})) \leq c$, chúng tôi có một lỗi giảm bốn lần

$$\left|e^{(k+1)}\right| \leq c(e^{(k)})^2. \tag{12.3.13}$$

Ngoài ra, các nhà nghiên cứu tối ưu hóa gọi sự hội tụ * tuyến tính* này, trong khi một điều kiện như $\left|e^{(k+1)}\right| \leq \alpha \left|e^{(k)}\right|$ sẽ được gọi là tốc độ hội tụ * không đổi*. Lưu ý rằng phân tích này đi kèm với một số cảnh báo. Đầu tiên, chúng tôi không thực sự có nhiều sự đảm bảo khi chúng tôi sẽ đạt được khu vực hội tụ nhanh chóng. Thay vào đó, chúng ta chỉ biết rằng một khi chúng ta đạt được nó, sự hội tụ sẽ rất nhanh chóng. Thứ hai, phân tích này đòi hỏi f được cư xử tốt đến các dẫn xuất bậc cao hơn. Nó đi xuống để đảm bảo rằng f không có bất kỳ thuộc tính "đáng ngạc nhiên" nào về cách nó có thể thay đổi giá trị của nó.

Điều hòa trước

Khá không ngạc nhiên khi tính toán và lưu trữ Hessian đầy đủ là rất tốn kém. Do đó, nó là mong muốn để tìm lựa chọn thay thế. Một cách để cải thiện vấn đề là * điều kiện tiên chuẩn*. Nó tránh tính toán toàn bộ Hessian nhưng chỉ tính toán các mục nhập *diagonal*. Điều này dẫn đến cập nhật các thuật toán của biểu mẫu

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \mathrm{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}). \tag{12.3.14}$$

Mặc dù điều này không hoàn toàn tốt bằng phương pháp Newton đầy đủ, nhưng nó vẫn tốt hơn nhiều so với việc không sử dụng nó. Để xem lý do tại sao đây có thể là một ý tưởng tốt, hãy xem xét một tình huống mà một biến biểu thị chiều cao tính bằng milimét và cái kia biểu thị chiều cao tính bằng km. Giả sử rằng đối với cả hai quy mô tự nhiên đều tính bằng mét, chúng ta có một sự không phù hợp khủng khiếp trong các tham số hóa. May mắn thay, sử dụng preconditioning loại bỏ điều này. Điều hòa trước hiệu quả với số lượng gốc gradient để lựa chọn một tốc độ học tập khác nhau cho mỗi biến (tọa độ của vector $\mathbf{x}$). Như chúng ta sẽ thấy sau này, điều hòa trước thúc đẩy một số sự đổi mới trong các thuật toán tối ưu hóa dòng dốc ngẫu nhiên.

Gradient Descent với tìm kiếm dòng

Một trong những vấn đề chính trong gradient gốc là chúng ta có thể vượt qua mục tiêu hoặc không đủ tiến bộ. Một sửa chữa đơn giản cho vấn đề là sử dụng tìm kiếm dòng kết hợp với gradient gốc. Đó là, chúng tôi sử dụng hướng được đưa ra bởi $\nabla f(\mathbf{x})$ và sau đó thực hiện tìm kiếm nhị phân như tỷ lệ học tập η giảm thiểu $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$.

Thuật toán này hội tụ nhanh chóng (để phân tích và chứng minh xem ví dụ, (Boyd & Vandenberghe, 2004)). Tuy nhiên, với mục đích học sâu, điều này không hoàn toàn khả thi, vì mỗi bước của tìm kiếm dòng sẽ yêu cầu chúng ta đánh giá chức năng khách quan trên toàn bộ tập dữ liệu. Đây là cách quá tốn kém để thực hiện.

12.3.4 Tóm tắt

- Tỷ lệ học vấn đề. Quá lớn và chúng tôi phân kỳ, quá nhỏ và chúng tôi không đạt được tiến bộ.
- Gradient gốc có thể bị mắc kẹt trong minima địa phương.
- Ở kích thước cao, việc điều chỉnh tốc độ học tập rất phức tạp.
- Điều hòa trước có thể giúp điều chỉnh quy mô.
- Phương pháp của Newton nhanh hơn rất nhiều khi nó đã bắt đầu hoạt động bình thường trong các bài toán lồi.
- Cẩn thận với việc sử dụng phương pháp Newton mà không có bất kỳ điều chỉnh nào đối với các bài toán không lồi.

12.3.5 Bài tập

1. Thử nghiệm với các tốc độ học tập khác nhau và chức năng khách quan để chuyển đổi độ dốc.
2. Thực hiện tìm kiếm dòng để giảm thiểu một hàm lồi trong khoảng $[a, b]$.
    1. Bạn có cần các dẫn xuất cho tìm kiếm nhị phân, tức là, để quyết định chọn $[a, (a + b)/2]$ hoặc $[(a + b)/2, b]$.
    2. Tốc độ hội tụ cho thuật toán nhanh như thế nào?
    3. Thực hiện thuật toán và áp dụng nó để giảm thiểu $\log(\exp(x) + \exp(-2x - 3))$.
3. Thiết kế một chức năng khách quan được xác định trên $\mathbb{R}^2$ trong đó chuyển đổi độ dốc cực kỳ chậm. Gợi ý: quy mô tọa độ khác nhau khác nhau.
4. Thực hiện phiên bản nhẹ của phương pháp Newton bằng cách sử dụng điều hòa trước:
    1. Sử dụng đường chéo Hessian như preconditioner.
    2. Sử dụng các giá trị tuyệt đối của giá trị đó chứ không phải là giá trị thực tế (có thể ký).
    3. Áp dụng điều này cho vấn đề trên.
5. Áp dụng thuật toán trên cho một số hàm khách quan (lồi hay không). Điều gì xảy ra nếu bạn xoay tọa độ 45 độ?

Discussions[128]

12.4 Stochastic Gradient Descent

Tuy nhiên, trong các chương trước, chúng tôi tiếp tục sử dụng gốc gradient ngẫu nhiên trong quy trình đào tạo của chúng tôi, tuy nhiên, mà không giải thích lý do tại sao nó hoạt động. Để làm sáng tỏ nó, chúng tôi chỉ mô tả các nguyên tắc cơ bản của gradient gốc trong Section 12.3. Trong phần này, chúng tôi tiếp tục thảo luận *xuống dốc ngẫu nhiên* chi tiết hơn.

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

12.4.1 Stochastic Gradient Những Thông Tin Cập Nhập

Trong học sâu, chức năng khách quan thường là trung bình của các hàm mất cho mỗi ví dụ trong tập dữ liệu đào tạo. Với một tập dữ liệu đào tạo n ví dụ, chúng tôi giả định rằng $f_i(\mathbf{x})$ là hàm mất liên quan đến ví dụ đào tạo của chỉ số i, trong đó $\mathbf{x}$ là vectơ tham số. Sau đó, chúng tôi đến chức năng mục tiêu

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} f_i(\mathbf{x}). \tag{12.4.1}$$

Gradient của hàm khách quan tại $\mathbf{x}$ được tính là

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{12.4.2}$$

[128] https://discuss.d2l.ai/t/351

Nếu sử dụng gradient descent, chi phí tính toán cho mỗi lần lặp biến độc lập là $\mathcal{O}(n)$, phát triển tuyến tính với n. Do đó, khi tập dữ liệu đào tạo lớn hơn, chi phí của gradient descent cho mỗi lần lặp sẽ cao hơn.

Stochastic gradient descent (SGD) giảm chi phí tính toán tại mỗi lần lặp lại. Tại mỗi lần lặp lại của dòng dốc ngẫu nhiên, chúng tôi lấy mẫu thống nhất một chỉ số $i \in \{1, \ldots, n\}$ cho các ví dụ dữ liệu một cách ngẫu nhiên và tính toán gradient $\nabla f_i(\mathbf{x})$ để cập nhật $\mathbf{x}$:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}), \tag{12.4.3}$$

trong đó η là tỷ lệ học tập. Chúng ta có thể thấy rằng chi phí tính toán cho mỗi lần lặp giảm từ $\mathcal{O}(n)$ của gradient gốc xuống hằng số $\mathcal{O}(1)$. Hơn nữa, chúng tôi muốn nhấn mạnh rằng gradient ngẫu nhiên $\nabla f_i(\mathbf{x})$ là một ước tính không thiên vị của gradient đầy đủ $\nabla f(\mathbf{x})$ bởi vì

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{12.4.4}$$

Điều này có nghĩa là, trung bình, gradient stochastic là một ước tính tốt của gradient.

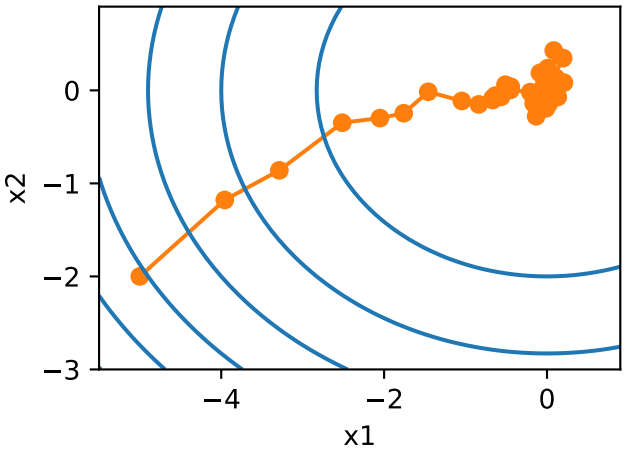
Bây giờ, chúng ta sẽ so sánh nó với gradient descent bằng cách thêm nhiễu ngẫu nhiên với trung bình 0 và phương sai 1 với gradient để mô phỏng một gradient gốc ngẫu nhiên.

```
def f(x1, x2):  # Objective function
    return x1 ** 2 + 2 * x2 ** 2

def f_grad(x1, x2):  # Gradient of the objective function
    return 2 * x1, 4 * x2
```

```
def sgd(x1, x2, s1, s2, f_grad):
    g1, g2 = f_grad(x1, x2)
    # Simulate noisy gradient
    g1 += torch.normal(0.0, 1, (1,))
    g2 += torch.normal(0.0, 1, (1,))
    eta_t = eta * lr()
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0)
```

```
def constant_lr():
    return 1

eta = 0.1
lr = constant_lr  # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: 0.100386, x2: 0.117807
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪numpy/core/shape_base.py:65: VisibleDeprecationWarning: Creating an ndarray␣
↪from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-
↪or ndarrays with different lengths or shapes) is deprecated. If you meant␣
↪to do this, you must specify 'dtype=object' when creating the ndarray.
  ary = asanyarray(ary)
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Như chúng ta có thể thấy, quỹ đạo của các biến trong gốc gradient ngẫu nhiên ồn hơn nhiều so với quỹ đạo mà chúng ta quan sát thấy trong gradient gốc trong Section 12.3. Điều này là do tính chất ngẫu nhiên của gradient. Đó là, ngay cả khi chúng tôi đến gần mức tối thiểu, chúng tôi vẫn phải chịu sự không chắc chắn được tiêm bởi gradient tức thời qua $\eta \nabla f_i(\mathbf{x})$. Ngay cả sau 50 bước chất lượng vẫn không tốt như vậy. Thậm chí tệ hơn, nó sẽ không cải thiện sau các bước bổ sung (chúng tôi khuyến khích bạn thử nghiệm với một số lượng lớn hơn các bước để xác nhận điều này). Điều này để lại cho chúng ta sự thay thế duy nhất: thay đổi tỷ lệ học tập η. Tuy nhiên, nếu chúng ta chọn điều này quá nhỏ, ban đầu chúng ta sẽ không đạt được bất kỳ tiến bộ có ý nghĩa nào. Mặt khác, nếu chúng ta chọn nó quá lớn, chúng ta sẽ không nhận được một giải pháp tốt, như đã thấy ở trên. Cách duy nhất để giải quyết các mục tiêu mâu thuẫn này là giảm tỷ lệ học tập * năng lượng* khi tối ưu hóa tiến triển.

Đây cũng là lý do để thêm một hàm tốc độ học tập `lr` vào hàm bước `sgd`. Trong ví dụ trên bất kỳ chức năng nào để lập kế hoạch tỷ lệ học tập nằm không hoạt động khi chúng ta đặt hàm `lr` liên quan là hằng số.

12.4.2 Tốc độ học động

Thay thế η bằng tốc độ học tập phụ thuộc vào thời gian $\eta(t)$ thêm vào sự phức tạp của việc kiểm soát sự hội tụ của một thuật toán tối ưu hóa. Đặc biệt, chúng ta cần tìm ra cách nhanh chóng η sẽ phân rã. Nếu quá nhanh, chúng tôi sẽ ngừng tối ưu hóa sớm. Nếu chúng ta giảm nó quá chậm, chúng ta lãng phí quá nhiều thời gian để tối ưu hóa. Sau đây là một vài chiến lược cơ bản được sử dụng để điều chỉnh η theo thời gian (chúng tôi sẽ thảo luận về các chiến lược nâng cao hơn sau):

$$\begin{aligned} \eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{piecewise constant} \\ \eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{exponential decay} \\ \eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{polynomial decay} \end{aligned} \tag{12.4.5}$$

Trong kịch bản *piecewise constant* đầu tiên, chúng tôi giảm tỷ lệ học tập, ví dụ, bất cứ khi nào tiến bộ trong các quầy hàng tối ưu hóa. Đây là một chiến lược phổ biến để đào tạo các mạng sâu. Ngoài ra, chúng ta có thể giảm nó mạnh mẽ hơn nhiều bởi sự phân rã theo cấp số nhân *. Thật không may điều này thường dẫn đến dừng sớm trước khi thuật toán đã hội tụ. Một lựa chọn phổ biến là * phân rã đa thứ* với $\alpha = 0.5$. Trong trường hợp tối ưu hóa lồi, có một số bằng chứng cho thấy tỷ lệ này được cư xử tốt.

Chúng ta hãy xem sự phân rã theo cấp số nhân trông như thế nào trong thực tế.

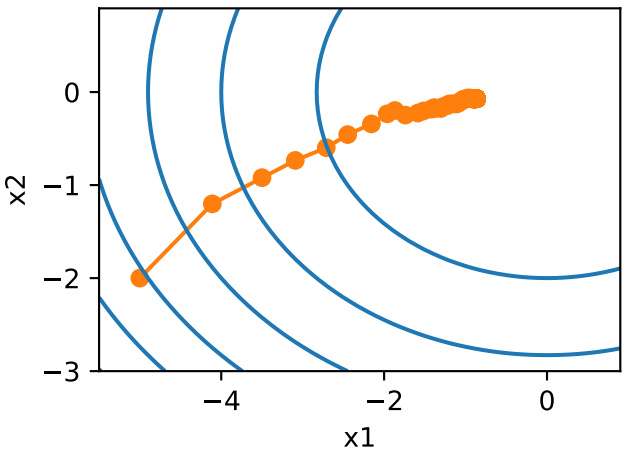
```
def exponential_lr():
    # Global variable that is defined outside this function and updated inside
```

(continues on next page)

(continued from previous page)

```
    global t
    t += 1
    return math.exp(-0.1 * t)

t = 1
lr = exponential_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000, f_grad=f_grad))
```

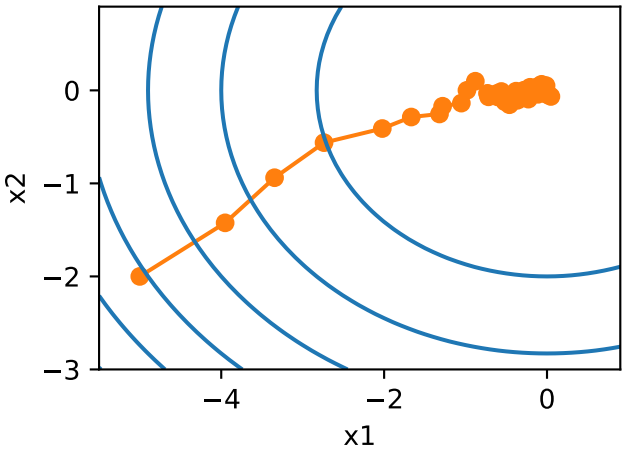
```
epoch 1000, x1: -0.866443, x2: -0.070251
```



Đúng như dự đoán, phương sai trong các thông số được giảm đáng kể. Tuy nhiên, điều này đến với chi phí không hội tụ với giải pháp tối ưu $\mathbf{x} = (0, 0)$. Ngay cả sau 1000 bước lặp lại chúng ta vẫn còn rất xa giải pháp tối ưu. Thật vậy, thuật toán không hội tụ ở tất cả. Mặt khác, nếu chúng ta sử dụng phân rã đa thức trong đó tốc độ học tập phân rã với căn bậc hai nghịch đảo của số bước, sự hội tụ sẽ tốt hơn chỉ sau 50 bước.

```
def polynomial_lr():
    # Global variable that is defined outside this function and updated inside
    global t
    t += 1
    return (1 + 0.1 * t) ** (-0.5)

t = 1
lr = polynomial_lr
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50, f_grad=f_grad))
```

```
epoch 50, x1: -0.011478, x2: 0.051227
```



Tồn tại nhiều lựa chọn hơn cho cách thiết lập tỷ lệ học tập. Ví dụ, chúng ta có thể bắt đầu với một tốc độ nhỏ, sau đó nhanh chóng tăng lên và sau đó giảm nó một lần nữa, mặc dù chậm hơn. Chúng tôi thậm chí có thể thay thế giữa tỷ lệ học tập nhỏ hơn và lớn hơn. Có tồn tại một loạt các lịch trình như vậy. Bây giờ chúng ta hãy tập trung vào lịch trình học tập mà có thể phân tích lý thuyết toàn diện, tức là về tốc độ học tập trong một môi trường lồi. Đối với các bài toán phi lồi nói chung, rất khó để có được những đảm bảo hội tụ có ý nghĩa, vì nói chung giảm thiểu các bài toán phi lồi phi tuyến là NP cứng. Đối với một cuộc khảo sát xem ví dụ, xuất sắc bài giảng ghi chú[129] của Tibshirani 2015.

12.4.3 Phân tích hội tụ cho mục tiêu lồi

Phân tích hội tụ sau đây của gốc gradient ngẫu nhiên cho các chức năng mục tiêu lồi là tùy chọn và chủ yếu phục vụ để truyền đạt nhiều trực giác hơn về vấn đề. Chúng tôi giới hạn bản thân mình với một trong những bằng chứng đơn giản nhất (Nesterov & Vial, 2000). Các kỹ thuật chứng minh tiên tiến hơn đáng kể tồn tại, ví dụ, bất cứ khi nào chức năng mục tiêu được cư xử đặc biệt tốt.

Giả sử hàm khách quan $f(\boldsymbol{\xi}, \mathbf{x})$ là lồi trong $\mathbf{x}$ cho tất cả $\boldsymbol{\xi}$. Cụ thể hơn, chúng tôi xem xét bản cập nhật gradient gốc stochastic:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}), \tag{12.4.6}$$

trong đó $f(\boldsymbol{\xi}_t, \mathbf{x})$ là chức năng khách quan đối với ví dụ đào tạo $\boldsymbol{\xi}_t$ rút ra từ một số phân phối ở bước t và $\mathbf{x}$ là tham số mô hình. Biểu thị bởi

$$R(\mathbf{x}) = E_{\boldsymbol{\xi}}[f(\boldsymbol{\xi}, \mathbf{x})] \tag{12.4.7}$$

rủi ro dự kiến và đến R^* mức tối thiểu của nó đối với $\mathbf{x}$. Cuối cùng cho phép $\mathbf{x}^*$ là minimizer (chúng tôi giả định rằng nó tồn tại trong miền nơi $\mathbf{x}$ được xác định). Trong trường hợp này, chúng ta có thể theo dõi khoảng cách giữa tham số hiện tại $\mathbf{x}_t$ tại thời điểm t và bộ giảm thiểu rủi ro $\mathbf{x}^*$ và xem liệu nó có cải thiện theo thời gian hay không:

$$\begin{aligned} &\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \eta_t \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) - \mathbf{x}^*\|^2 \\ =&\|\mathbf{x}_t - \mathbf{x}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 - 2\eta_t \left\langle \mathbf{x}_t - \mathbf{x}^*, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}) \right\rangle. \end{aligned} \tag{12.4.8}$$

[129] https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf

Chúng tôi giả định rằng định mức L_2 của gradient ngẫu nhiên $\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})$ được giới hạn bởi một số hằng số L, do đó chúng tôi có

$$\eta_t^2 \|\partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x})\|^2 \leq \eta_t^2 L^2. \tag{12.4.9}$$

Chúng tôi chủ yếu quan tâm đến khoảng cách giữa $\mathbf{x}_t$ và $\mathbf{x}^*$ thay đổi như thế nào *theo mong đợi*. Trên thực tế, đối với bất kỳ trình tự cụ thể nào của các bước, khoảng cách có thể tăng lên, tùy thuộc vào bất kỳ $\boldsymbol{\xi}_t$ nào chúng ta gặp phải. Do đó chúng ta cần phải ràng buộc các sản phẩm dot. Kể từ khi đối với bất kỳ chức năng lồi f nó giữ rằng $f(\mathbf{y}) \geq f(\mathbf{x}) + \langle f'(\mathbf{x}), \mathbf{y} - \mathbf{x} \rangle$ cho tất cả $\mathbf{x}$ và $\mathbf{y}$, bởi lồi chúng tôi có

$$f(\boldsymbol{\xi}_t, \mathbf{x}^*) \geq f(\boldsymbol{\xi}_t, \mathbf{x}_t) + \langle \mathbf{x}^* - \mathbf{x}_t, \partial_{\mathbf{x}} f(\boldsymbol{\xi}_t, \mathbf{x}_t) \rangle \,. \tag{12.4.10}$$

Cắm cả hai bất đẳng thức (12.4.9) và (12.4.10) vào (12.4.8) chúng tôi có được một ràng buộc về khoảng cách giữa các tham số tại thời điểm $t+1$ như sau:

$$\|\mathbf{x}_t - \mathbf{x}^*\|^2 - \|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2 \geq 2\eta_t (f(\boldsymbol{\xi}_t, \mathbf{x}_t) - f(\boldsymbol{\xi}_t, \mathbf{x}^*)) - \eta_t^2 L^2. \tag{12.4.11}$$

Điều này có nghĩa là chúng tôi đạt được tiến bộ miễn là sự khác biệt giữa tổn thất hiện tại và tổn thất tối ưu lớn hơn $\eta_t L^2/2$. Vì sự khác biệt này được ràng buộc để hội tụ với 0 nên sau đó tỷ lệ học tập η_t cũng cần phải * biến mất *.

Tiếp theo, chúng tôi có kỳ vọng hơn (12.4.11). This yields năng suất

$$E\left[\|\mathbf{x}_t - \mathbf{x}^*\|^2\right] - E\left[\|\mathbf{x}_{t+1} - \mathbf{x}^*\|^2\right] \geq 2\eta_t [E[R(\mathbf{x}_t)] - R^*] - \eta_t^2 L^2. \tag{12.4.12}$$

Bước cuối cùng liên quan đến việc tổng hợp các bất bình đẳng cho $t \in \{1, \ldots, T\}$. Kể từ khi tổng kính thiên văn và bằng cách giảm thuật ngữ thấp hơn, chúng tôi thu được

$$\|\mathbf{x}_1 - \mathbf{x}^*\|^2 \geq 2 \left(\sum_{t=1}^{T} \eta_t \right) [E[R(\mathbf{x}_t)] - R^*] - L^2 \sum_{t=1}^{T} \eta_t^2. \tag{12.4.13}$$

Lưu ý rằng chúng tôi đã khai thác rằng $\mathbf{x}_1$ được đưa ra và do đó kỳ vọng có thể được giảm xuống. Định nghĩa cuối

$$\bar{\mathbf{x}} \stackrel{\text{def}}{=} \frac{\sum_{t=1}^{T} \eta_t \mathbf{x}_t}{\sum_{t=1}^{T} \eta_t}. \tag{12.4.14}$$

Kể từ

$$E\left(\frac{\sum_{t=1}^{T} \eta_t R(\mathbf{x}_t)}{\sum_{t=1}^{T} \eta_t} \right) = \frac{\sum_{t=1}^{T} \eta_t E[R(\mathbf{x}_t)]}{\sum_{t=1}^{T} \eta_t} = E[R(\mathbf{x}_t)], \tag{12.4.15}$$

bởi sự bất bình đẳng của Jensen (thiết lập $i = t$, $\alpha_i = \eta_t / \sum_{t=1}^{T} \eta_t$ trong (12.2.3)) và độ lồi của R nó theo sau đó $E[R(\mathbf{x}_t)] \geq E[R(\bar{\mathbf{x}})]$, do đó

$$\sum_{t=1}^{T} \eta_t E[R(\mathbf{x}_t)] \geq \sum_{t=1}^{T} \eta_t E\left[R(\bar{\mathbf{x}})\right]. \tag{12.4.16}$$

Cắm này vào bất bình đẳng (12.4.13) mang lại sự ràng buộc

$$[E[\bar{\mathbf{x}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^{T} \eta_t^2}{2 \sum_{t=1}^{T} \eta_t}, \tag{12.4.17}$$

trong đó $r^2 \stackrel{\text{def}}{=} \|\mathbf{x}_1 - \mathbf{x}^*\|^2$ là một ràng buộc về khoảng cách giữa sự lựa chọn ban đầu của các tham số và kết quả cuối cùng. Nói tóm lại, tốc độ hội tụ phụ thuộc vào cách định mức của gradient ngẫu nhiên được giới hạn (L) và cách tối ưu giá trị tham số ban đầu là bao xa (r). Lưu ý rằng sự ràng buộc là về $\bar{\mathbf{x}}$ chứ không phải là $\mathbf{x}_T$. Đây là trường hợp vì $\bar{\mathbf{x}}$ là một phiên bản được làm mịn của đường dẫn tối ưu hóa. Bất cứ khi nào r, L, và T được biết đến, chúng tôi có thể chọn tỷ lệ học tập $\eta = r/(L\sqrt{T})$. Điều này mang lại như trên ràng buộc $rL/\sqrt{T}$. Đó là, chúng tôi hội tụ với tỷ lệ $\mathcal{O}(1/\sqrt{T})$ đến giải pháp tối ưu.

12.4.4 Gradient Stochastic và mẫu hữu hạn

Cho đến nay chúng tôi đã chơi một chút nhanh và lỏng lẻo khi nói về gốc gradient ngẫu nhiên. Chúng tôi xác định rằng chúng tôi vẽ các trường hợp x_i, thường với nhãn y_i từ một số phân phối $p(x, y)$ và chúng tôi sử dụng điều này để cập nhật các tham số mô hình theo một cách nào đó. Đặc biệt, đối với một kích thước mẫu hữu hạn, chúng tôi chỉ đơn giản lập luận rằng phân phối rời rạc $p(x, y) = \frac{1}{n} \sum_{i=1}^{n} \delta_{x_i}(x)\delta_{y_i}(y)$ cho một số chức năng δ_{x_i} và δ_{y_i} cho phép chúng tôi thực hiện chuyển đổi gradient ngẫu nhiên trên nó.

Tuy nhiên, đây không thực sự là những gì chúng tôi đã làm. Trong các ví dụ đồ chơi trong phần hiện tại, chúng tôi chỉ cần thêm tiếng ồn vào một gradient không ngẫu nhiên khác, tức là, chúng tôi giả vờ có cặp (x_i, y_i). Nó chỉ ra rằng điều này là hợp lý ở đây (xem các bài tập để thảo luận chi tiết). Rắc rối hơn là trong tất cả các cuộc thảo luận trước đó, chúng tôi rõ ràng đã không làm điều này. Thay vào đó, chúng tôi lặp lại tất cả các phiên bản * chính xác một lần*. Để xem lý do tại sao điều này là thích hợp hơn, hãy xem xét cuộc trò chuyện, cụ thể là chúng tôi đang lấy mẫu n quan sát từ phân phối rời rạc * với thay thế*. Xác suất chọn một phần tử i ngẫu nhiên là $1/n$. Do đó để chọn nó * ít nhất* một lần là

$$P(\text{choose } i) = 1 - P(\text{omit } i) = 1 - (1 - 1/n)^n \approx 1 - e^{-1} \approx 0.63. \tag{12.4.18}$$

Một lý do tương tự cho thấy xác suất chọn một số mẫu (tức là ví dụ đào tạo) * chính xác một lần* được đưa ra bởi

$$\binom{n}{1}\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} = \frac{n}{n-1}\left(1 - \frac{1}{n}\right)^{n} \approx e^{-1} \approx 0.37. \tag{12.4.19}$$

Điều này dẫn đến sự gia tăng phương sai và giảm hiệu quả dữ liệu so với lấy mẫu * mà không thay thế*. Do đó, trong thực tế, chúng tôi thực hiện sau này (và đây là lựa chọn mặc định trong suốt cuốn sách này). Lưu ý cuối cùng rằng lặp đi lặp lại qua tập dữ liệu đào tạo đi qua nó theo thứ tự ngẫu nhiên *khác nhau*.

12.4.5 Tóm tắt

- Đối với các vấn đề lồi, chúng ta có thể chứng minh rằng đối với một sự lựa chọn rộng của tỷ lệ học tập stochastic gradient gốc sẽ hội tụ với giải pháp tối ưu.
- Đối với học sâu, điều này thường không phải là trường hợp. Tuy nhiên, việc phân tích các bài toán lồi cho chúng ta cái nhìn sâu sắc hữu ích về cách tiếp cận tối ưu hóa, cụ thể là giảm tốc độ học tập dần dần, mặc dù không quá nhanh.
- Vấn đề xảy ra khi tỷ lệ học tập quá nhỏ hoặc quá lớn. Trong thực tế, một tỷ lệ học tập phù hợp thường chỉ được tìm thấy sau nhiều thí nghiệm.
- Khi có nhiều ví dụ hơn trong tập dữ liệu đào tạo, chi phí nhiều hơn để tính toán mỗi lần lặp cho gradient descent, do đó, stochastic gradient descent được ưa thích trong những trường hợp này.
- Đảm bảo tối ưu cho dòng dốc ngẫu nhiên nói chung không có sẵn trong các trường hợp không lồi vì số lượng minima cục bộ yêu cầu kiểm tra cũng có thể là cấp số nhân.

12.4.6 Bài tập

1. Thử nghiệm với các lịch trình tỷ lệ học tập khác nhau cho gốc gradient ngẫu nhiên và với các số lần lặp khác nhau. Đặc biệt, vẽ khoảng cách từ giải pháp tối ưu $(0, 0)$ như một hàm của số lần lặp lại.
2. Chứng minh rằng đối với hàm $f(x_1, x_2) = x_1^2 + 2x_2^2$ thêm nhiễu bình thường vào gradient tương đương với việc giảm thiểu hàm mất $f(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$ trong đó $\mathbf{x}$ được rút ra từ một phân phối bình thường.
3. So sánh sự hội tụ của gốc gradient ngẫu nhiên khi bạn lấy mẫu từ $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ với sự thay thế và khi bạn lấy mẫu mà không cần thay thế.
4. Làm thế nào bạn sẽ thay đổi các stochastic gradient descent solver nếu một số gradient (hoặc đúng hơn là một số phối hợp liên quan đến nó) đã luôn lớn hơn tất cả các gradient khác?
5. Giả sử rằng $f(x) = x^2(1 + \sin x)$. f có bao nhiêu minima địa phương? Bạn có thể thay đổi f theo cách để giảm thiểu nó, người ta cần đánh giá tất cả các minima địa phương không?

Discussions[130]

12.5 Minibatch Stochastic Gradient Descent

Cho đến nay chúng ta đã gặp phải hai thái cực trong cách tiếp cận để gradient dựa learning: Section 12.3 sử dụng bộ dữ liệu đầy đủ để tính toán gradient và để cập nhật các tham số, một lần vượt qua tại một thời điểm. Ngược lại Section 12.4 xử lý một quan sát tại một thời điểm để đạt được tiến bộ. Mỗi người trong số họ có nhược điểm riêng. Gradient Descent không đặc biệt* hiệu quả dữ liệu* bất cứ khi nào dữ liệu rất giống nhau. Stochastic Gradient Descent không đặc biệt* hiệu quả tính tế* vì CPU và GPU không thể khai thác toàn bộ sức mạnh của vectơ hóa. Điều này cho thấy rằng có thể có một phương tiện hạnh phúc, và trên thực tế, đó là những gì chúng tôi đã sử dụng cho đến nay trong các ví dụ chúng tôi đã thảo luận.

12.5.1 Vector hóa và bộ nhớ cache

Trọng tâm của quyết định sử dụng minibatches là hiệu quả tính toán. Điều này dễ hiểu nhất khi xem xét song song với nhiều GPU và nhiều máy chủ. Trong trường hợp này, chúng ta cần gửi ít nhất một hình ảnh cho mỗi GPU. Với 8 GPU trên mỗi máy chủ và 16 máy chủ, chúng tôi đã có kích thước minibatch là 128.

Mọi thứ tinh tế hơn một chút khi nói đến GPU đơn lẻ hoặc thậm chí CPU. Các thiết bị này có nhiều loại bộ nhớ, thường là nhiều loại đơn vị tính toán và hạn chế băng thông khác nhau giữa chúng. Ví dụ, CPU có một số lượng nhỏ các thanh ghi và sau đó là L1, L2 và trong một số trường hợp thậm chí bộ nhớ cache L3 (được chia sẻ giữa các lõi bộ xử lý khác nhau). Các bộ nhớ đệm này có kích thước và độ trễ ngày càng tăng (đồng thời chúng giảm băng thông). Nó đủ để nói, bộ xử lý có khả năng thực hiện nhiều hoạt động hơn so với những gì giao diện bộ nhớ chính có thể cung cấp.

- CPU 2GHz với 16 lõi và vectorization AVX-512 có thể xử lý lên đến $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$ byte mỗi giây. Khả năng của GPU dễ dàng vượt quá con số này theo hệ số 100. Mặt khác, một bộ xử lý máy chủ tầm trung có thể không có nhiều hơn 100 Gb/s băng thông, tức là, ít hơn một phần mười những gì sẽ được yêu cầu để giữ cho bộ xử lý ăn. Để làm cho vấn đề tồi tệ hơn, không phải tất cả truy cập bộ nhớ được tạo ra bằng nhau: đầu tiên, giao diện bộ nhớ thường rộng 64 bit hoặc rộng hơn (ví dụ, trên GPU lên đến 384 bit), do đó đọc một byte duy nhất phải chịu chi phí truy cập rộng hơn nhiều.

[130] https://discuss.d2l.ai/t/497

- Có chi phí đáng kể cho truy cập đầu tiên trong khi truy cập tuần tự là tương đối rẻ (điều này thường được gọi là một lần đọc liên tục). Có rất nhiều điều cần lưu ý, chẳng hạn như bộ nhớ đệm khi chúng ta có nhiều ổ cắm, chiplet và các cấu trúc khác. Một cuộc thảo luận chi tiết về điều này nằm ngoài phạm vi của phần này. Xem ví dụ, Wikipedia article[131] này để có một cuộc thảo luận chuyên sâu hơn.

Cách để giảm bớt những hạn chế này là sử dụng một hệ thống phân cấp của bộ nhớ cache CPU thực sự đủ nhanh để cung cấp cho bộ xử lý dữ liệu. Đây là * động lực là* đằng sau việc phân mẻ trong học sâu. Để giữ cho vấn đề đơn giản, hãy xem xét phép nhân ma trận ma trận, nói $\mathbf{A} = \mathbf{BC}$. Chúng tôi có một số tùy chọn để tính toán $\mathbf{A}$. Ví dụ, chúng tôi có thể thử như sau:

1. Chúng tôi có thể tính toán $\mathbf{A}_{ij} = \mathbf{B}_{i,:}\mathbf{C}_{:,j}^\top$, tức là, chúng tôi có thể tính toán nó elementwise bằng phương tiện của các sản phẩm chấm.
2. Chúng ta có thể tính toán $\mathbf{A}_{:,j} = \mathbf{BC}_{:,j}^\top$, tức là, chúng ta có thể tính toán nó một cột tại một thời điểm. Tương tự như vậy chúng ta có thể tính $\mathbf{A}$ một hàng $\mathbf{A}_{i,:}$ tại một thời điểm.
3. We could simply đơn giản compute tính toán $\mathbf{A} = \mathbf{BC}$.
4. Chúng ta có thể phá vỡ $\mathbf{B}$ và $\mathbf{C}$ thành các ma trận khối nhỏ hơn và tính toán $\mathbf{A}$ một khối tại một thời điểm.

Nếu chúng ta làm theo tùy chọn đầu tiên, chúng ta sẽ cần sao chép một hàng và một vectơ cột vào CPU mỗi khi chúng ta muốn tính một phần tử $\mathbf{A}_{ij}$. Thậm chí tệ hơn, do thực tế là các yếu tố ma trận được căn chỉnh tuần tự, do đó, chúng ta được yêu cầu truy cập nhiều vị trí tách rời cho một trong hai vectơ khi chúng ta đọc chúng từ bộ nhớ. Tùy chọn thứ hai thuận lợi hơn nhiều. Trong đó, chúng ta có thể giữ vector cột $\mathbf{C}_{:,j}$ trong bộ nhớ cache CPU trong khi chúng tôi tiếp tục đi qua B. Điều này giảm một nửa yêu cầu băng thông bộ nhớ với truy cập nhanh hơn tương ứng. Tất nhiên, tùy chọn 3 là mong muốn nhất. Thật không may, hầu hết các ma trận có thể không hoàn toàn phù hợp với bộ nhớ cache (đây là những gì chúng ta đang thảo luận sau khi tất cả). Tuy nhiên, tùy chọn 4 cung cấp một giải pháp thay thế thực tế hữu ích: chúng ta có thể di chuyển các khối ma trận vào bộ nhớ cache và nhân chúng cục bộ. Tối ưu hóa thư viện chăm sóc này cho chúng tôi. Chúng ta hãy xem các hoạt động này hiệu quả như thế nào trong thực tế.

Ngoài hiệu quả tính toán, chi phí được giới thiệu bởi Python và bởi chính khuôn khổ học tập sâu là đáng kể. Nhớ lại rằng mỗi lần chúng ta thực hiện một lệnh, trình thông dịch Python sẽ gửi một lệnh đến công cụ MXNet cần chèn nó vào biểu đồ tính toán và xử lý nó trong quá trình lên lịch. Chi phí như vậy có thể khá bất lợi. Nói tóm lại, rất nên sử dụng vector hóa (và ma trận) bất cứ khi nào có thể.

```
%matplotlib inline
import numpy as np
import torch
from torch import nn
from d2l import torch as d2l

timer = d2l.Timer()
A = torch.zeros(256, 256)
B = torch.randn(256, 256)
C = torch.randn(256, 256)
```

Nhiệm vụ yếu tố khôn ngoan chỉ đơn giản là lặp lại trên tất cả các hàng và cột của $\mathbf{B}$ và $\mathbf{C}$ tương ứng để gán giá trị cho $\mathbf{A}$.

```
# Compute A = BC one element at a time
timer.start()
```

(continues on next page)

[131] https://en.wikipedia.org/wiki/Cache_hierarchy

(continued from previous page)

```
for i in range(256):
    for j in range(256):
        A[i, j] = torch.dot(B[i, :], C[:, j])
timer.stop()
```

```
1.056152582168579
```

Một chiến lược nhanh hơn là thực hiện gán cột khôn ngoan.

```
# Compute A = BC one column at a time
timer.start()
for j in range(256):
    A[:, j] = torch.mv(B, C[:, j])
timer.stop()
```

```
0.009429693222045898
```

Cuối cùng, cách hiệu quả nhất là thực hiện toàn bộ hoạt động trong một khối. Hãy để chúng tôi xem tốc độ tương ứng của các hoạt động là bao nhiêu.

```
# Compute A = BC in one go
timer.start()
A = torch.mm(B, C)
timer.stop()

# Multiply and add count as separate operations (fused in practice)
gigaflops = [2/i for i in timer.times]
print(f'performance in Gigaflops: element {gigaflops[0]:.3f}, '
      f'column {gigaflops[1]:.3f}, full {gigaflops[2]:.3f}')
```

```
performance in Gigaflops: element 1.894, column 212.096, full 3315.655
```

12.5.2 Minibatches

Trong quá khứ, chúng tôi đã cho rằng chúng tôi sẽ đọc * minibatches* của dữ liệu chứ không phải là quan sát duy nhất để cập nhật các tham số. Bây giờ chúng tôi đưa ra một lời biện minh ngắn gọn cho nó. Xử lý các quan sát đơn yêu cầu chúng ta thực hiện nhiều phép nhân ma thuật-vector (hoặc thậm chí là vector-vector) đơn lẻ, khá tốn kém và phát sinh một chi phí đáng kể thay mặt cho khuôn khổ học sâu cơ bản. Điều này áp dụng cả để đánh giá một mạng khi áp dụng cho dữ liệu (thường được gọi là suy luận) và khi tính toán gradient để cập nhật các tham số. Đó là, điều này áp dụng bất cứ khi nào chúng tôi thực hiện $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$ ở đâu

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \tag{12.5.1}$$

Chúng ta có thể tăng hiệu quả * tính tị* của thao tác này bằng cách áp dụng nó vào một loạt các quan sát tại một thời điểm. Đó là, chúng tôi thay thế gradient $\mathbf{g}_t$ trong một quan sát duy nhất bằng một trong một lô nhỏ

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \tag{12.5.2}$$

Chúng ta hãy xem điều này làm gì với các thuộc tính thống kê của $\mathbf{g}_t$: vì cả $\mathbf{x}_t$ và tất cả các yếu tố của minibatch $\mathcal{B}_t$ được vẽ đồng đều ngẫu nhiên từ bộ đào tạo, kỳ vọng của gradient vẫn không thay đổi. Mặt khác, phương sai được giảm đáng kể. Kể từ khi gradient minibatch bao gồm $b := |\mathcal{B}_t|$ gradient độc lập đang được trung bình, độ lệch chuẩn của nó được giảm bởi một hệ số $b^{-\frac{1}{2}}$. Bản thân nó, điều này là một điều tốt, vì nó có nghĩa là các bản cập nhật được liên kết đáng tin cậy hơn với gradient đầy đủ.

Ngây thơ điều này sẽ chỉ ra rằng việc lựa chọn một minibatch lớn $\mathcal{B}_t$ sẽ là mong muốn phổ biến. Than ôi, sau một số điểm, việc giảm thêm độ lệch chuẩn là tối thiểu khi so sánh với sự gia tăng tuyến tính trong chi phí tính toán. Trong thực tế, chúng tôi chọn một minibatch đủ lớn để mang lại hiệu quả tính toán tốt trong khi vẫn phù hợp với bộ nhớ của GPU. Để minh họa tiết kiệm chúng ta hãy xem xét một số mã. Trong đó chúng ta thực hiện cùng một phép nhân ma trận ma trận, nhưng lần này chia thành "minibatches" của 64 cột tại một thời điểm.

```
timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = torch.mm(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {2 / timer.times[3]:.3f}')
```

```
performance in Gigaflops: block 1455.092
```

Như chúng ta có thể thấy, tính toán trên minibatch về cơ bản là hiệu quả như trên ma trận đầy đủ. Một lời thận trọng là theo thứ tự. Trong Section 8.5, chúng tôi đã sử dụng một loại chính quy hóa phụ thuộc nhiều vào số lượng phương sai trong một minibatch. Khi chúng ta tăng sau này, phương sai giảm và cùng với nó là lợi ích của việc phun tiếng ồn do bình thường hóa hàng loạt. Xem ví dụ, (Ioffe, 2017) để biết chi tiết về cách giải thích và tính toán các điều khoản thích hợp.

12.5.3 Đọc tập dữ liệu

Chúng ta hãy xem cách minibatches được tạo ra hiệu quả từ dữ liệu. Sau đây chúng tôi sử dụng một tập dữ liệu do NASA phát triển để kiểm tra cánh noise from different aircraft[132] để so sánh các thuật toán tối ưu hóa này. Để thuận tiện, chúng tôi chỉ sử dụng các ví dụ $1,500$ đầu tiên. Dữ liệu được làm trắng để xử lý trước, tức là, chúng tôi loại bỏ trung bình và giải thích phương sai thành 1 cho mỗi tọa độ.

```
#@save
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')

#@save
def get_data_ch11(batch_size=10, n=1500):
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    data_iter = d2l.load_array((data[:n, :-1], data[:n, -1]),
                               batch_size, is_train=True)
    return data_iter, data.shape[1]-1
```

[132] https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

12.5.4 Thực hiện từ đầu

Nhớ lại việc thực hiện giảm dần dần ngẫu nhiên minibatch từ Section 4.2. Trong phần sau đây, chúng tôi cung cấp một thực hiện tổng quát hơn một chút. Để thuận tiện, nó có chữ ký cuộc gọi giống như các thuật toán tối ưu hóa khác được giới thiệu sau trong chương này. Cụ thể, chúng tôi thêm đầu vào trạng thái `states` và đặt siêu tham số trong từ điển `hyperparams`. Ngoài ra, chúng tôi sẽ trung bình mất mỗi ví dụ minibatch trong chức năng đào tạo, do đó gradient trong thuật toán tối ưu hóa không cần phải chia cho kích thước lô.

```
def sgd(params, states, hyperparams):
    for p in params:
        p.data.sub_(hyperparams['lr'] * p.grad)
        p.grad.data.zero_()
```

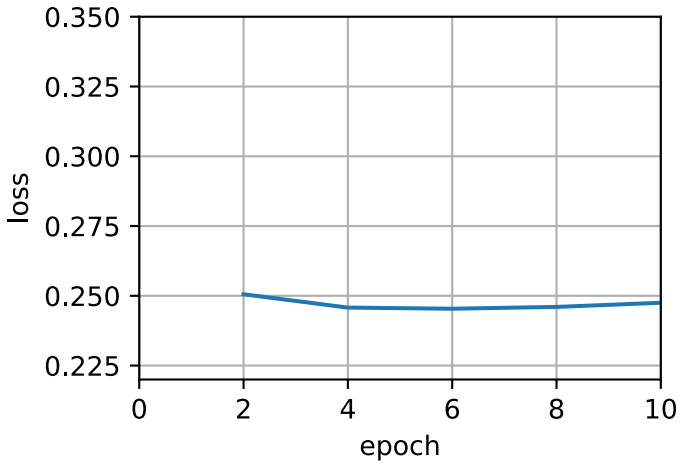
Tiếp theo, chúng tôi thực hiện một chức năng đào tạo chung để tạo điều kiện cho việc sử dụng các thuật toán tối ưu hóa khác được giới thiệu sau này trong chương này. Nó khởi tạo một mô hình hồi quy tuyến tính và có thể được sử dụng để đào tạo mô hình với minibatch stochastic gradient gốc và các thuật toán khác được giới thiệu sau đó.

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # Initialization
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     requires_grad=True)
    b = torch.zeros((1), requires_grad=True)
    net, loss = lambda X: d2l.linreg(X, w, b), d2l.squared_loss
    # Train
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            l = loss(net(X), y).mean()
            l.backward()
            trainer_fn([w, b], states, hyperparams)
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss),))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
    return timer.cumsum(), animator.Y[0]
```

Hãy để chúng tôi xem tối ưu hóa tiến hành như thế nào để giảm gradient hàng loạt. Điều này có thể đạt được bằng cách đặt kích thước minibatch thành 1500 (tức là tổng số ví dụ). Kết quả là các thông số mô hình chỉ được cập nhật một lần cho mỗi kỷ nguyên. Có rất ít tiến bộ. Trong thực tế, sau 6 bước tiến trình quầy hàng.

```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_data_ch11(batch_size)
    return train_ch11(
        sgd, None, {'lr': lr}, data_iter, feature_dim, num_epochs)

gd_res = train_sgd(1, 1500, 10)
```

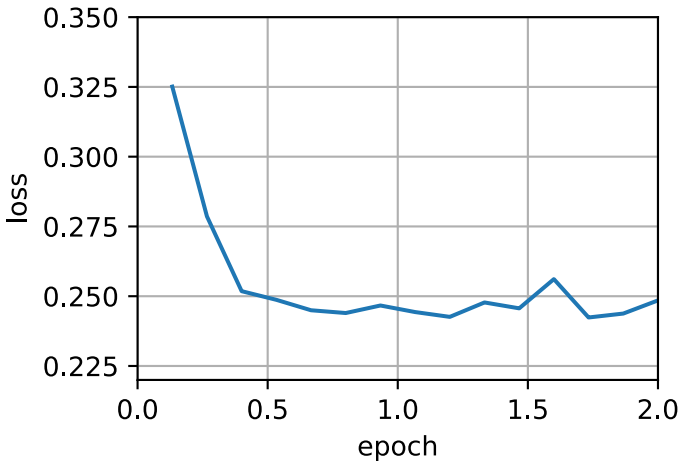
```
loss: 0.248, 0.036 sec/epoch
```



Khi kích thước lô bằng 1, chúng ta sử dụng stochastic gradient descent để tối ưu hóa. Để đơn giản thực hiện, chúng tôi đã chọn một tỷ lệ học tập liên tục (mặc dù nhỏ). Trong dòng dốc ngẫu nhiên, các tham số mô hình được cập nhật bất cứ khi nào một ví dụ được xử lý. Trong trường hợp của chúng tôi, điều này lên tới 1500 bản cập nhật mỗi kỷ nguyên. Như chúng ta có thể thấy, sự suy giảm giá trị của hàm mục tiêu chậm lại sau một kỷ nguyên. Mặc dù cả hai quy trình đã xử lý 1500 ví dụ trong một kỷ nguyên, dòng dốc ngẫu nhiên tiêu thụ nhiều thời gian hơn so với gradient gốc trong thí nghiệm của chúng tôi. Điều này là do stochastic gradient gốc cập nhật các thông số thường xuyên hơn và vì nó kém hiệu quả hơn để xử lý các quan sát duy nhất tại một thời điểm.

```
sgd_res = train_sgd(0.005, 1)
```

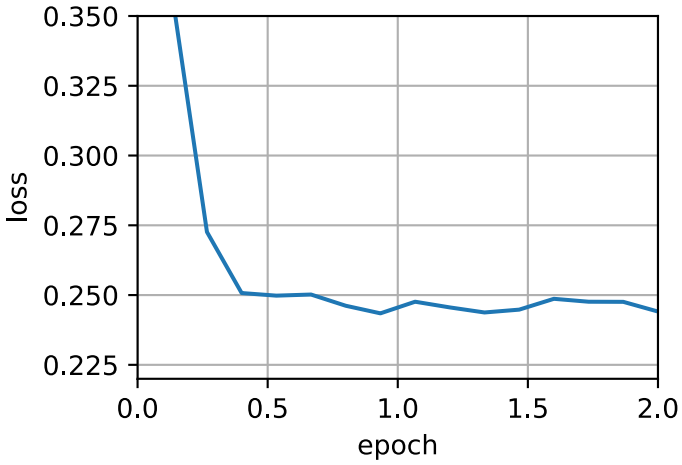
```
loss: 0.248, 0.076 sec/epoch
```



Cuối cùng, khi kích thước lô bằng 100, chúng ta sử dụng minibatch stochastic gradient descent để tối ưu hóa. Thời gian cần thiết cho mỗi kỷ nguyên ngắn hơn thời gian cần thiết cho dòng gradient stochastic và thời gian để giảm gradient hàng loạt.

```
mini1_res = train_sgd(.4, 100)
```

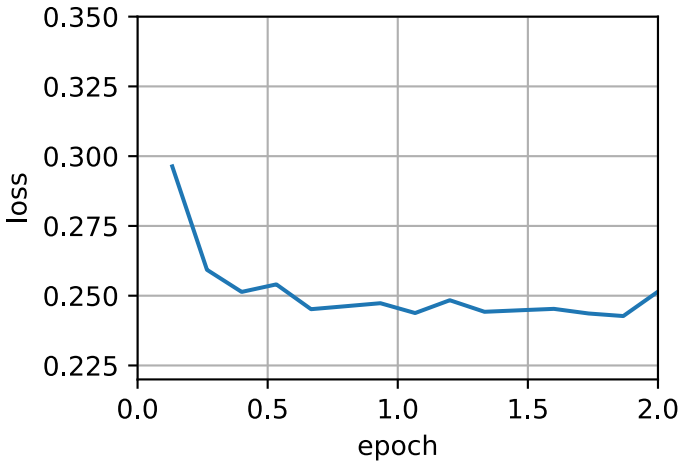
```
loss: 0.244, 0.003 sec/epoch
```



Giảm kích thước lô xuống còn 10, thời gian cho mỗi kỷ nguyên tăng vì khối lượng công việc cho mỗi lô kém hiệu quả hơn để thực hiện.

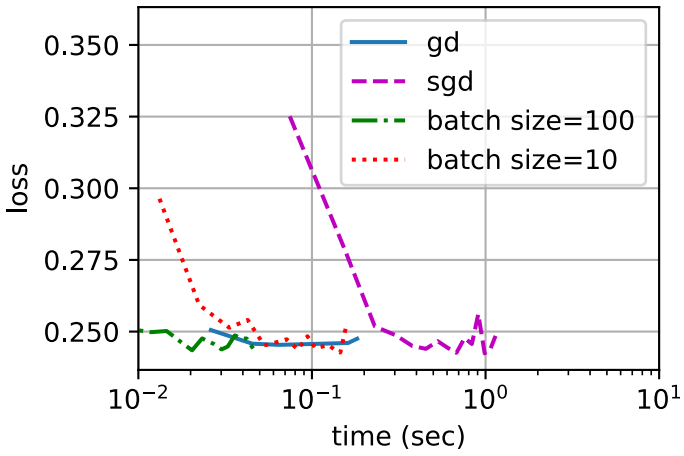
```
mini2_res = train_sgd(.05, 10)
```

```
loss: 0.251, 0.010 sec/epoch
```



Bây giờ chúng ta có thể so sánh thời gian so với mất mát cho bốn thí nghiệm trước đó. Như có thể thấy, mặc dù stochastic gradient descent hội tụ nhanh hơn GD về số lượng ví dụ được xử lý, nó sử dụng nhiều thời gian hơn để đạt được tổn thất tương tự so với GD vì tính toán ví dụ gradient bằng ví dụ không hiệu quả như vậy. Minibatch stochastic gradient gốc có thể đánh đổi tốc độ hội tụ và hiệu quả tính toán. Kích thước minibatch là 10 hiệu quả hơn so với dòng gradient stochastic; kích thước minibatch 100 thậm chí vượt trội hơn GD về thời gian chạy.

```
d2l.set_figsize([6, 3])
d2l.plot(*list(map(list, zip(gd_res, sgd_res, mini1_res, mini2_res))),
         'time (sec)', 'loss', xlim=[1e-2, 10],
         legend=['gd', 'sgd', 'batch size=100', 'batch size=10'])
d2l.plt.gca().set_xscale('log')
```



12.5.5 Thực hiện ngắn gọn

Trong Gluon, chúng ta có thể sử dụng lớp `Trainer` để gọi các thuật toán tối ưu hóa. Điều này được sử dụng để thực hiện một chức năng đào tạo chung. Chúng tôi sẽ sử dụng điều này trong suốt chương hiện tại.

```
#@save
def train_concise_ch11(trainer_fn, hyperparams, data_iter, num_epochs=4):
    # Initialization
    net = nn.Sequential(nn.Linear(5, 1))
    def init_weights(m):
        if type(m) == nn.Linear:
            torch.nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)

    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = nn.MSELoss(reduction='none')
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs], ylim=[0.22, 0.35])
    n, timer = 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            optimizer.zero_grad()
            out = net(X)
            y = y.reshape(out.shape)
            l = loss(out, y)
            l.mean().backward()
            optimizer.step()
            n += X.shape[0]
            if n % 200 == 0:
                timer.stop()
                # `MSELoss` computes squared error without the 1/2 factor
```

(continues on next page)

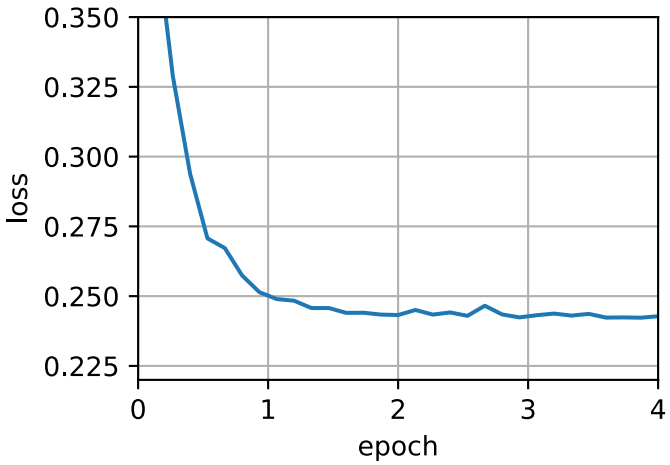
(continued from previous page)

```
                animator.add(n/X.shape[0]/len(data_iter),
                             (d2l.evaluate_loss(net, data_iter, loss) / 2,))
                timer.start()
    print(f'loss: {animator.Y[0][-1]:.3f}, {timer.avg():.3f} sec/epoch')
```

Sử dụng Gluon để lặp lại thí nghiệm cuối cùng cho thấy hành vi giống hệt nhau.

```
data_iter, _ = get_data_ch11(10)
trainer = torch.optim.SGD
train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.243, 0.010 sec/epoch
```



12.5.6 Tóm tắt

- Vectorization làm cho mã hiệu quả hơn do giảm chi phí phát sinh từ khung học sâu và do địa phương bộ nhớ tốt hơn và bộ nhớ đệm trên CPU và GPU.
- Có một sự đánh đổi giữa hiệu quả thống kê phát sinh từ gốc gradient ngẫu nhiên và hiệu quả tính toán phát sinh từ việc xử lý các lô dữ liệu lớn tại một thời điểm.
- Minibatch stochastic gradient descent cung cấp tốt nhất của cả hai thế giới: tính toán và hiệu quả thống kê.
- Trong minibatch stochastic gradient descent, chúng tôi xử lý các lô dữ liệu thu được bằng một hoán vị ngẫu nhiên của dữ liệu đào tạo (tức là, mỗi quan sát chỉ được xử lý một lần cho mỗi kỷ nguyên, mặc dù theo thứ tự ngẫu nhiên).
- Đó là khuyến khích để phân rã tỷ lệ học tập trong quá trình đào tạo.
- Nói chung, minibatch stochastic gradient gốc nhanh hơn so với stochastic gradient descent và gradient descent để hội tụ đến một rủi ro nhỏ hơn, khi được đo về thời gian đồng hồ.

12.5.7 Bài tập

1. Sửa đổi kích thước lô và tỷ lệ học tập và quan sát tốc độ suy giảm đối với giá trị của hàm khách quan và thời gian tiêu thụ trong mỗi kỷ nguyên.
2. Đọc tài liệu MXNet và sử dụng chức năng `Trainer` lớp `set_learning_rate` để giảm tốc độ học tập của gradient ngẫu nhiên minibatch xuống 1/10 giá trị trước đó của nó sau mỗi kỷ nguyên.
3. So sánh minibatch stochastic gradient descent với một biến thể mà thực sự * mẫu với thay thế* từ bộ đào tạo. Điều gì xảy ra?
4. Một vị thần ác sao chép tập dữ liệu của bạn mà không nói với bạn (tức là, mỗi quan sát xảy ra hai lần và tập dữ liệu của bạn phát triển lên gấp đôi kích thước ban đầu của nó, nhưng không ai nói với bạn). Làm thế nào để hành vi của stochastic gradient gốc, minibatch stochastic gradient descent và của gradient gốc thay đổi?

Discussions[133]

12.6 Đà

Trong Section 12.4, chúng tôi đã xem xét những gì xảy ra khi thực hiện gốc gradient ngẫu nhiên, tức là khi thực hiện tối ưu hóa, nơi chỉ có một biến thể ồn ào của gradient có sẵn. Đặc biệt, chúng tôi nhận thấy rằng đối với độ dốc ồn ào, chúng ta cần phải thận trọng hơn khi chọn tốc độ học tập khi đối mặt với tiếng ồn. Nếu chúng ta giảm nó quá nhanh, hội tụ quầy hàng. Nếu chúng ta quá khoan dung, chúng ta không hội tụ thành một giải pháp đủ tốt vì tiếng ồn tiếp tục khiến chúng ta tránh xa sự tối ưu.

12.6.1 Khái niệm cơ bản

Trong phần này, chúng ta sẽ khám phá các thuật toán tối ưu hóa hiệu quả hơn, đặc biệt là đối với một số loại vấn đề tối ưu hóa phổ biến trong thực tế.

Đường trung bình bị rò rỉ

Phần trước đã thấy chúng tôi thảo luận về minibatch SGD như một phương tiện để tăng tốc tính toán. Nó cũng có tác dụng phụ tốt đẹp mà độ dốc trung bình làm giảm lượng phương sai. Các minibatch stochastic gradient descent có thể được tính bằng cách:

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \tag{12.6.1}$$

Để giữ cho ký hiệu đơn giản, ở đây chúng tôi đã sử dụng $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$ làm gốc gradient ngẫu nhiên cho mẫu i bằng cách sử dụng các trọng lượng được cập nhật tại thời điểm $t-1$. Sẽ thật tuyệt nếu chúng ta có thể hưởng lợi từ ảnh hưởng của việc giảm phương sai ngay cả ngoài độ dốc trung bình trên một minibatch. Một lựa chọn để thực hiện nhiệm vụ này là thay thế tính toán gradient bằng một "trung bình rò rỉ":

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \tag{12.6.2}$$

for some $\beta \in (0, 1)$. Điều này thay thế hiệu quả gradient tức thời bằng một gradient được trung bình trên nhiều độ dốc * past*. **v** được gọi là *momentum*. Nó tích lũy gradient trong quá khứ tương tự như cách một

[133] https://discuss.d2l.ai/t/1068

quả bóng nặng lăn xuống cảnh quan chức năng mục tiêu tích hợp trên các lực trong quá khứ. Để xem những gì đang xảy ra chi tiết hơn, chúng ta hãy mở rộng $\mathbf{v}_t$ đệ quy vào

$$\mathbf{v}_t = \beta^2\mathbf{v}_{t-2} + \beta\mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \ldots, = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \tag{12.6.3}$$

β lớn lên tới mức trung bình tầm xa, trong khi β nhỏ chỉ có một sự điều chỉnh nhẹ so với phương pháp gradient. Sự thay thế gradient mới không còn trỏ vào hướng dốc nhất trên một trường hợp cụ thể nữa mà theo hướng trung bình có trọng số của các gradient trong quá khứ. Điều này cho phép chúng tôi nhận ra hầu hết các lợi ích của việc trung bình trong một lô mà không có chi phí thực sự tính toán độ dốc trên đó. Chúng tôi sẽ xem lại quy trình trung bình này chi tiết hơn sau.

Lý luận trên hình thành cơ sở cho những gì bây giờ được gọi là phương pháp gradient *accelerated*, chẳng hạn như gradient với đà. Họ tận hưởng lợi ích bổ sung là hiệu quả hơn nhiều trong trường hợp vấn đề tối ưu hóa là không điều kiện (tức là, nơi có một số hướng mà sự tiến bộ chậm hơn nhiều so với những người khác, giống như một hẻm núi hẹp). Hơn nữa, chúng cho phép chúng ta trung bình trên các gradient tiếp theo để có được hướng xuống ổn định hơn. Thật vậy, khía cạnh của khả năng tăng tốc ngay cả đối với các vấn đề lồi không ồn là một trong những lý do chính khiến động lượng hoạt động và tại sao nó hoạt động tốt như vậy.

Như người ta mong đợi, do động lực hiệu quả của nó là một môn học được nghiên cứu tốt trong tối ưu hóa cho học sâu và hơn thế nữa. Xem ví dụ, đẹp bài viết expository[134] by (Goh, 2017) để phân tích chuyên sâu và hoạt hình tương tác. Nó được đề xuất bởi (Polyak, 1964). (Nesterov, 2018) có một cuộc thảo luận lý thuyết chi tiết trong bối cảnh tối ưu hóa lồi. Đà trong học sâu đã được biết đến là có lợi trong một thời gian dài. Xem ví dụ, cuộc thảo luận của (Sutskever et al., 2013) để biết chi tiết.

Một vấn đề Ill-conditioned

Để hiểu rõ hơn về các thuộc tính hình học của phương pháp động lượng, chúng tôi xem lại gradient descent, mặc dù với một chức năng khách quan ít dễ chịu hơn đáng kể. Nhớ lại rằng trong Section 12.3, chúng tôi đã sử dụng $f(\mathbf{x}) = x_1^2 + 2x_2^2$, tức là, một mục tiêu ellipsoid biến dạng vừa phải. Chúng tôi làm biến dạng chức năng này hơn nữa bằng cách kéo dài nó theo hướng x_1 thông qua

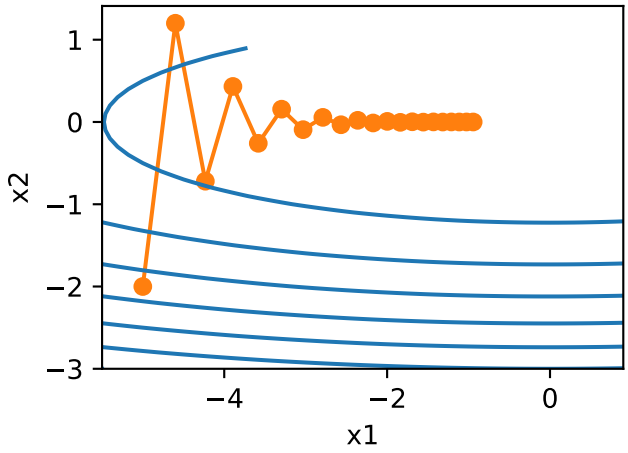
$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{12.6.4}$$

Như trước f có mức tối thiểu là $(0, 0)$. Chức năng này là * rất* phẳng theo hướng x_1. Chúng ta hãy xem những gì sẽ xảy ra khi chúng ta thực hiện gradient gốc như trước đây trên chức năng mới này. Chúng tôi chọn một tỷ lệ học tập là 0.4.

```
%matplotlib inline
import torch
from d2l import torch as d2l

eta = 0.4
def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2
def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

[134] https://distill.pub/2017/momentum/

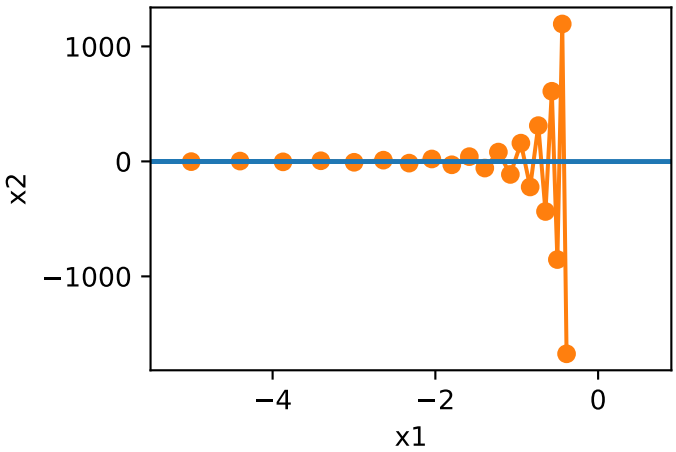
```
epoch 20, x1: -0.943467, x2: -0.000073
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Bằng cách xây dựng, gradient theo hướng x_2 là * nhiều* cao hơn và thay đổi nhanh hơn nhiều so với theo chiều ngang x_1. Do đó, chúng tôi đang bị mắc kẹt giữa hai lựa chọn không mong muốn: nếu chúng tôi chọn một tốc độ học tập nhỏ, chúng tôi đảm bảo rằng giải pháp không phân kỳ theo hướng x_2 nhưng chúng tôi đang yên tâm với sự hội tụ chậm theo hướng x_1. Ngược lại, với tốc độ học tập lớn, chúng tôi tiến bộ nhanh chóng theo hướng x_1 nhưng phân kỳ trong x_2. Ví dụ dưới đây minh họa những gì xảy ra ngay cả sau khi tăng nhẹ tỷ lệ học tập từ 0.4 lên 0.6. Sự hội tụ theo hướng x_1 được cải thiện nhưng chất lượng giải pháp tổng thể tồi tệ hơn nhiều.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

```
epoch 20, x1: -0.387814, x2: -1673.365109
```



Phương pháp Momentum

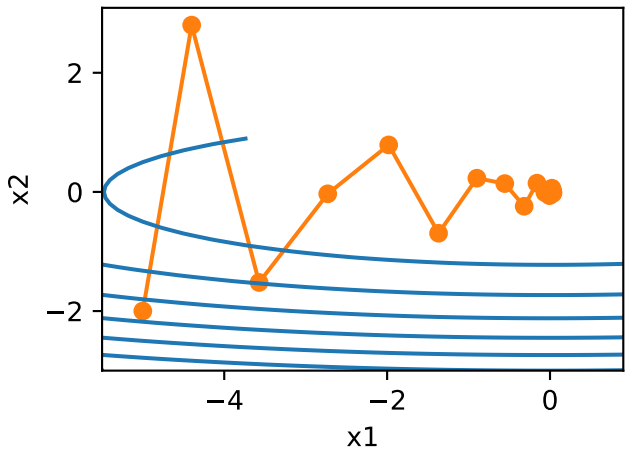
Phương pháp động lượng cho phép chúng ta giải quyết vấn đề gốc gradient được mô tả ở trên. Nhìn vào dấu vết tối ưu hóa ở trên, chúng ta có thể cảm thấy rằng độ dốc trung bình trong quá khứ sẽ hoạt động tốt. Rốt cuộc, theo hướng x_1, điều này sẽ tổng hợp các gradient được căn chỉnh tốt, do đó làm tăng khoảng cách chúng ta bao gồm mỗi bước. Ngược lại, theo hướng x_2 nơi gradient dao động, một gradient tổng hợp sẽ làm giảm kích thước bước do dao động hủy bỏ lẫn nhau ra. Sử dụng $\mathbf{v}_t$ thay vì gradient $\mathbf{g}_t$ mang lại các phương trình cập nhật sau:

$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1}, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t. \end{aligned} \tag{12.6.5}$$

Lưu ý rằng đối với $\beta = 0$, chúng tôi phục hồi độ dốc thường xuyên. Trước khi nghiên cứu sâu hơn vào các thuộc tính toán học, chúng ta hãy nhìn nhanh cách thuật toán hoạt động trong thực tế.

```
def momentum_2d(x1, x2, v1, v2):
    v1 = beta * v1 + 0.2 * x1
    v2 = beta * v2 + 4 * x2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

eta, beta = 0.6, 0.5
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

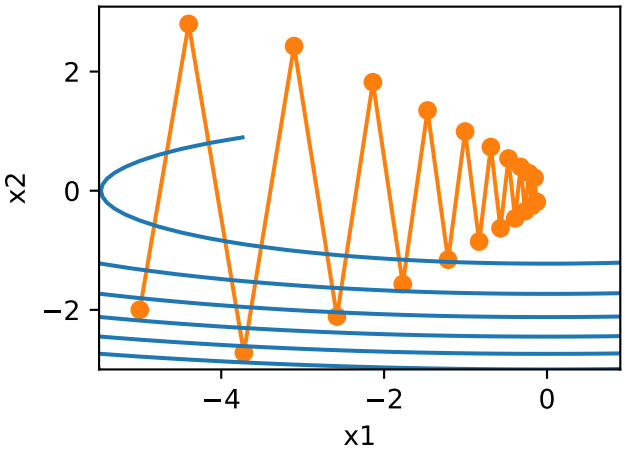
```
epoch 20, x1: 0.007188, x2: 0.002553
```



Như chúng ta có thể thấy, ngay cả với cùng tốc độ học tập mà chúng ta đã sử dụng trước đây, đà vẫn hội tụ tốt. Hãy để chúng tôi xem những gì xảy ra khi chúng ta giảm tham số động lượng. Giảm một nửa nó xuống $\beta = 0.25$ dẫn đến một quỹ đạo hầu như không hội tụ ở tất cả. Tuy nhiên, nó tốt hơn rất nhiều so với không có đà (khi giải pháp phân kỳ).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1: -0.126340, x2: -0.186632
```

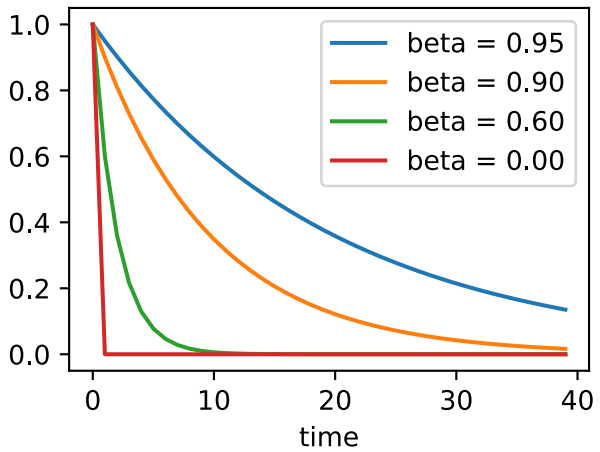


Lưu ý rằng chúng ta có thể kết hợp động lượng với gốc gradient ngẫu nhiên và đặc biệt, minibatch stochastic gradient descent. Thay đổi duy nhất là trong trường hợp đó, chúng tôi thay thế gradient $\mathbf{g}_{t,t-1}$ bằng $\mathbf{g}_t$. Cuối cùng, để thuận tiện, chúng tôi khởi tạo $\mathbf{v}_0 = 0$ tại thời điểm $t = 0$. Chúng ta hãy nhìn vào những gì rò rỉ trung bình thực sự làm cho các bản cập nhật.

Trọng lượng mẫu hiệu quả

Nhớ lại rằng $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}$. Trong giới hạn, các điều khoản thêm lên đến $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$. Nói cách khác, thay vì thực hiện một bước có kích thước η trong gradient gốc hoặc gốc gradient ngẫu nhiên, chúng tôi thực hiện một bước có kích thước $\frac{\eta}{1-\beta}$ trong khi đồng thời, đối phó với một hướng gốc có khả năng tốt hơn nhiều. Đây là hai lợi ích trong một. Để minh họa cách thức hoạt động của trọng số cho các lựa chọn khác nhau của β hãy xem xét sơ đồ dưới đây.

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



12.6.2 Các thí nghiệm thực tế

Chúng ta hãy xem động lượng hoạt động như thế nào trong thực tế, tức là, khi được sử dụng trong bối cảnh của một trình tối ưu hóa thích hợp. Đối với điều này, chúng ta cần một triển khai có thể mở rộng hơn một chút.

Thực hiện từ đầu

So với (minibatch) stochastic gradient gốc phương pháp động lượng cần duy trì một tập hợp các biến phụ trợ, tức là vận tốc. Nó có hình dạng tương tự như gradient (và các biến của bài toán tối ưu hóa). Trong việc thực hiện dưới đây, chúng tôi gọi các biến này là `states`.

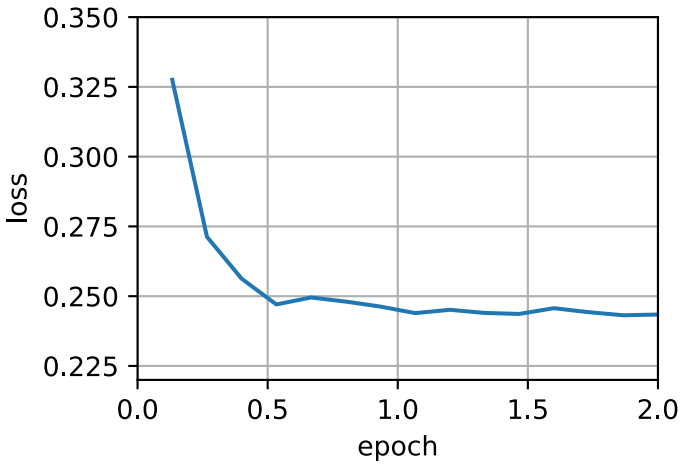
```
def init_momentum_states(feature_dim):
    v_w = torch.zeros((feature_dim, 1))
    v_b = torch.zeros(1)
    return (v_w, v_b)
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.data.zero_()
```

Hãy để chúng tôi xem làm thế nào điều này hoạt động trong thực tế.

```
def train_momentum(lr, momentum, num_epochs=2):
    d2l.train_ch11(sgd_momentum, init_momentum_states(feature_dim),
                   {'lr': lr, 'momentum': momentum}, data_iter,
                   feature_dim, num_epochs)

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
train_momentum(0.02, 0.5)
```

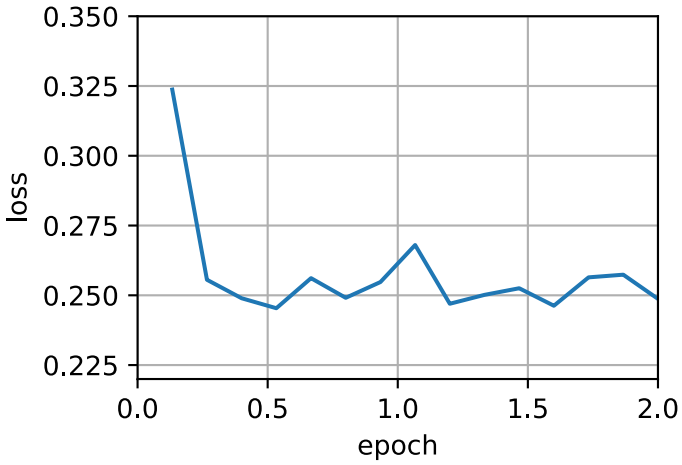
```
loss: 0.243, 0.013 sec/epoch
```



Khi chúng ta tăng siêu tham số động lượng `momentum` lên 0,9, nó sẽ lên tới kích thước mẫu hiệu quả lớn hơn đáng kể là $\frac{1}{1-0.9} = 10$. Chúng tôi giảm tỷ lệ học tập một chút xuống 0.01 để kiểm soát các vấn đề.

```
train_momentum(0.01, 0.9)
```

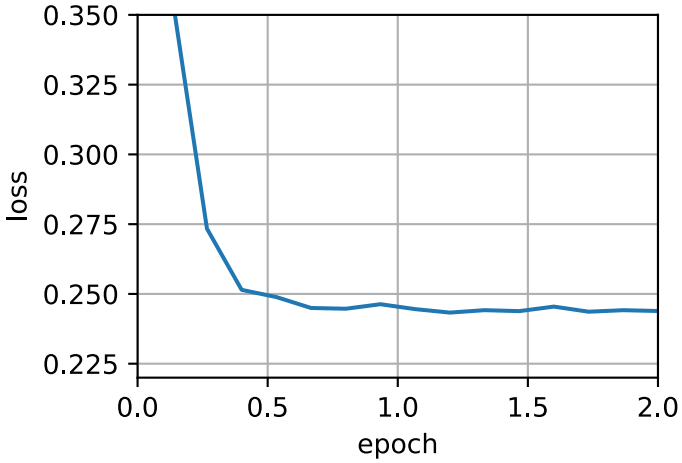
```
loss: 0.249, 0.012 sec/epoch
```



Giảm tỷ lệ học tập tiếp tục giải quyết bất kỳ vấn đề nào về các vấn đề tối ưu hóa không trơn tru. Đặt nó thành 0.005 mang lại các đặc tính hội tụ tốt.

```
train_momentum(0.005, 0.9)
```

```
loss: 0.244, 0.012 sec/epoch
```

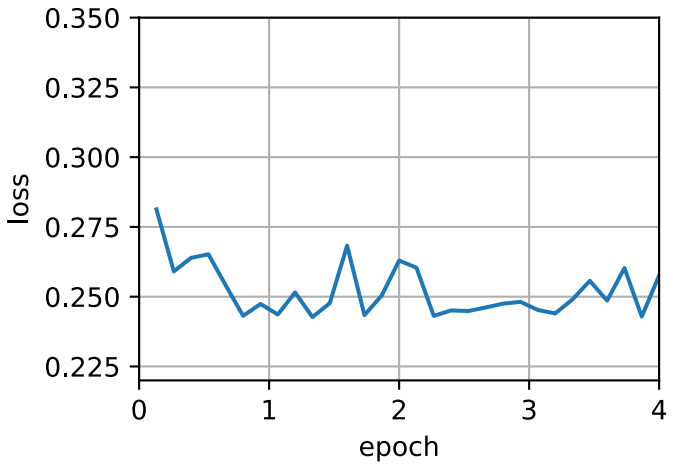


Thực hiện ngắn gọn

Có rất ít việc phải làm trong Gluon kể từ khi bộ giải `sgd` tiêu chuẩn đã có động lực tích hợp. Thiết lập các tham số phù hợp mang lại một quỹ đạo rất giống nhau.

```
trainer = torch.optim.SGD
d2l.train_concise_ch11(trainer, {'lr': 0.005, 'momentum': 0.9}, data_iter)
```

```
loss: 0.258, 0.012 sec/epoch
```



12.6.3 Phân tích lý thuyết

Cho đến nay, ví dụ 2D của $f(x) = 0.1x_1^2 + 2x_2^2$ dường như khá contrived. Bây giờ chúng ta sẽ thấy rằng điều này thực sự khá đại diện cho các loại vấn đề mà người ta có thể gặp phải, ít nhất là trong trường hợp giảm thiểu các hàm khách quan bậc hai lồi.

Hàm lồi bậc hai

Xem xét các chức năng

$$h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \tag{12.6.6}$$

Đây là một hàm bậc hai chung. Đối với ma trận xác định dương $\mathbf{Q} \succ 0$, tức là, đối với ma trận có giá trị eigenvalues dương, điều này có bộ giảm thiểu ở $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$ với giá trị tối thiểu $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Do đó chúng tôi có thể viết lại h như

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \tag{12.6.7}$$

Gradient được đưa ra bởi $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$. Đó là, nó được đưa ra bởi khoảng cách giữa $\mathbf{x}$ và bộ thu nhỏ, nhân với $\mathbf{Q}$. Do đó, động lượng cũng là sự kết hợp tuyến tính của các thuật ngữ $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$.

Kể từ $\mathbf{Q}$ là xác định dương, nó có thể được phân hủy thành hệ thống eigencủa nó thông qua $\mathbf{Q} = \mathbf{O}^\top \mathbf{\Lambda} \mathbf{O}$ cho một ma trận trực giao (xoay) $\mathbf{O}$ và một ma trận chéo $\mathbf{\Lambda}$ của eigenvalues dương. Điều này cho phép chúng ta thực hiện thay đổi các biến từ $\mathbf{x}$ thành $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ để có được một biểu thức đơn giản hóa nhiều:

$$h(\mathbf{z}) = \frac{1}{2}\mathbf{z}^\top \mathbf{\Lambda} \mathbf{z} + b'. \tag{12.6.8}$$

đây $b' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$. Vì $\mathbf{O}$ chỉ là một ma trận trực giao, điều này không làm xáo trộn độ dốc một cách có ý nghĩa. Thể hiện trong điều khoản của $\mathbf{z}$ gradient gốc trở thành

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \mathbf{\Lambda}\mathbf{z}_{t-1} = (\mathbf{I} - \mathbf{\Lambda})\mathbf{z}_{t-1}. \tag{12.6.9}$$

Thực tế quan trọng trong biểu thức này là gradient descent * không trộn xúc* giữa các eigenspace khác nhau. Đó là, khi được thể hiện theo hệ thống eigensystem của $\mathbf{Q}$, vấn đề tối ưu hóa tiến hành theo cách phối hợp khôn ngoan. Điều này cũng giữ cho đà.

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \mathbf{\Lambda}\mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta \left(\beta \mathbf{v}_{t-1} + \mathbf{\Lambda}\mathbf{z}_{t-1}\right) \\ &= (\mathbf{I} - \eta\mathbf{\Lambda})\mathbf{z}_{t-1} - \eta\beta\mathbf{v}_{t-1}. \end{aligned} \tag{12.6.10}$$

Khi làm điều này chúng ta chỉ chứng minh định lý sau: Gradient Descent có và không có động lượng cho một hàm bậc hai lồi phân hủy thành tối ưu hóa phối hợp khôn ngoan theo hướng của eigenvectors của ma trận bậc hai.

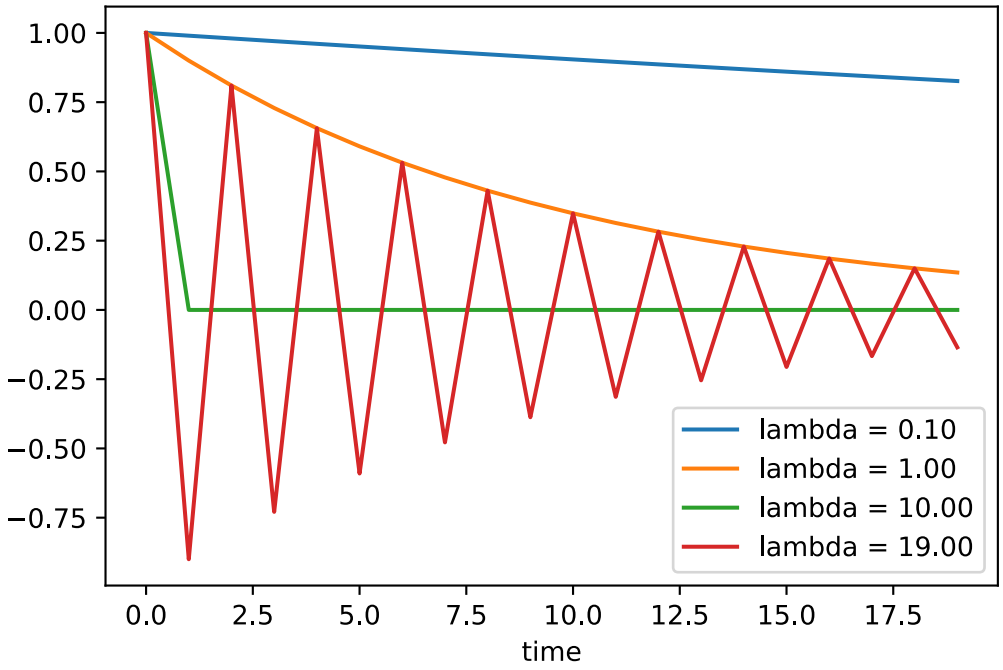
Hàm vô hướng

Với kết quả trên chúng ta hãy xem những gì sẽ xảy ra khi chúng ta giảm thiểu chức năng $f(x) = \frac{\lambda}{2}x^2$. Đối với gradient descent chúng tôi có

$$x_{t+1} = x_t - \eta\lambda x_t = (1 - \eta\lambda)x_t. \tag{12.6.11}$$

Bất cứ khi nào $|1 - \eta\lambda| < 1$ tối ưu hóa này hội tụ với tốc độ theo cấp số nhân kể từ sau t bước, chúng tôi có $x_t = (1 - \eta\lambda)^t x_0$. Điều này cho thấy tốc độ hội tụ được cải thiện ban đầu như thế nào khi chúng ta tăng tỷ lệ học tập η cho đến $\eta\lambda = 1$. Ngoài ra, mọi thứ phân kỳ và cho $\eta\lambda > 2$ vấn đề tối ưu hóa phân kỳ.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = torch.arange(20).detach().numpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```



Để phân tích sự hội tụ trong trường hợp động lượng, chúng ta bắt đầu bằng cách viết lại phương trình cập nhật theo hai vô hướng: một cho x và một cho động lượng v. Điều này mang lại:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1 - \eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \tag{12.6.12}$$

Chúng tôi đã sử dụng $\mathbf{R}$ để biểu thị 2×2 điều chỉnh hành vi hội tụ. Sau t bước sự lựa chọn ban đầu $[v_0, x_0]$ trở thành $\mathbf{R}(\beta, \eta, \lambda)^t[v_0, x_0]$. Do đó, tùy thuộc vào giá trị eigenvalues của $\mathbf{R}$ để xác định tốc độ hội tụ. Xem Distill post[135] of (Goh, 2017) cho một hình ảnh động tuyệt vời và (Flammarion & Bach, 2015) để phân tích chi tiết. Người ta có thể cho thấy đà $0 < \eta\lambda < 2 + 2\beta$ hội tụ. Đây là một phạm vi lớn hơn của các thông số khả thi khi so sánh với $0 < \eta\lambda < 2$ cho gradient gốc. Nó cũng cho thấy rằng nói chung các giá trị lớn của β là mong muốn. Thêm chi tiết yêu cầu một số lượng hợp lý của chi tiết kỹ thuật và chúng tôi đề nghị người đọc quan tâm tham khảo ý kiến các ấn phẩm gốc.

12.6.4 Tóm tắt

- Momentum thay thế gradient với một trung bình bị rò rỉ trên gradient trong quá khứ. Điều này tăng tốc sự hội tụ đáng kể.
- Đó là mong muốn cho cả hai gốc gradient không có tiếng ồn và (ồn ào) stochastic gradient gốc.
- Momentum ngăn chặn sự trì trệ của quá trình tối ưu hóa có nhiều khả năng xảy ra cho gốc gradient ngẫu nhiên.
- Số gradient hiệu quả được đưa ra bởi $\frac{1}{1-\beta}$ do giảm trọng lượng theo cấp số của dữ liệu trong quá khứ.
- Trong trường hợp các bài toán bậc hai lồi, điều này có thể được phân tích một cách rõ ràng một cách chi tiết.
- Việc thực hiện khá đơn giản nhưng nó đòi hỏi chúng ta phải lưu trữ một vector trạng thái bổ sung (đà **v**).

[135] https://distill.pub/2017/momentum/

12.6.5 Bài tập

1. Sử dụng các kết hợp khác của các siêu tham số động lượng và tỷ lệ học tập và quan sát và phân tích các kết quả thử nghiệm khác nhau.
2. Hãy thử GD và đà cho một vấn đề bậc hai nơi bạn có nhiều eigenvalues, tức là, $f(x) = \frac{1}{2}\sum_i \lambda_i x_i^2$, ví dụ, $\lambda_i = 2^{-i}$. Vẽ cách các giá trị của x giảm cho khởi tạo $x_i = 1$.
3. Lấy giá trị tối thiểu và minimizer cho $h(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$.
4. Điều gì thay đổi khi chúng ta thực hiện chuyển đổi ngẫu nhiên gradient với đà? Điều gì xảy ra khi chúng ta sử dụng minibatch stochastic gradient descent với đà? Thử nghiệm với các thông số?

Discussions[136]

12.7 Adagrad

Chúng ta hãy bắt đầu bằng cách xem xét các vấn đề học tập với các tính năng xảy ra không thường xuyên.

12.7.1 Tính năng thưa thớt và tỷ lệ học tập

Hãy tưởng tượng rằng chúng ta đang đào tạo một mô hình ngôn ngữ. Để có được độ chính xác tốt, chúng tôi thường muốn giảm tỷ lệ học tập khi chúng tôi tiếp tục đào tạo, thường ở tốc độ $\mathcal{O}(t^{-\frac{1}{2}})$ hoặc chậm hơn. Bây giờ hãy xem xét một đào tạo mô hình về các tính năng thưa thớt, tức là, các tính năng chỉ xảy ra không thường xuyên. Điều này là phổ biến cho ngôn ngữ tự nhiên, ví dụ, nó là rất ít khả năng rằng chúng ta sẽ thấy từ * preconditioning* so với *learning*. Tuy nhiên, nó cũng phổ biến ở các lĩnh vực khác như quảng cáo tính toán và lọc hợp tác được cá nhân hóa. Rốt cuộc, có rất nhiều điều chỉ được quan tâm đối với một số lượng nhỏ người.

Các tham số liên quan đến các tính năng không thường xuyên chỉ nhận được cập nhật có ý nghĩa bất cứ khi nào các tính năng này xảy ra. Với tốc độ học tập giảm, chúng ta có thể kết thúc trong một tình huống mà các thông số cho các tính năng chung hội tụ khá nhanh đến các giá trị tối ưu của chúng, trong khi đối với các tính năng không thường xuyên, chúng ta vẫn thiếu quan sát chúng đủ thường xuyên trước khi các giá trị tối ưu của chúng có thể được xác định. Nói cách khác, tỷ lệ học tập giảm quá chậm đối với các tính năng thường xuyên hoặc quá nhanh đối với những người không thường xuyên.

Một hack có thể để khắc phục vấn đề này sẽ là đếm số lần chúng ta thấy một tính năng cụ thể và sử dụng điều này làm đồng hồ để điều chỉnh tỷ lệ học tập. Đó là, thay vì chọn một tỷ lệ học tập của mẫu $\eta = \frac{\eta_0}{\sqrt{t+c}}$ chúng ta có thể sử dụng $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$. Ở đây $s(i,t)$ đếm số lượng nonzeros cho tính năng i mà chúng tôi đã quan sát đến thời gian t. Điều này thực sự khá dễ thực hiện tại không có chi phí có ý nghĩa. Tuy nhiên, nó thất bại bất cứ khi nào chúng ta không hoàn toàn có độ thưa thớt mà chỉ là dữ liệu mà các gradient thường rất nhỏ và chỉ hiếm khi lớn. Rốt cuộc, không rõ nơi người ta sẽ vẽ đường giữa một cái gì đó đủ điều kiện là một tính năng quan sát hay không.

Adagrad bởi (Duchi et al., 2011) giải quyết điều này bằng cách thay thế bộ đếm khá thô $s(i,t)$ bằng tổng hợp các ô vuông của gradient quan sát trước đó. Đặc biệt, nó sử dụng $s(i,t+1) = s(i,t) + \left(\partial_i f(\mathbf{x})\right)^2$ như một phương tiện để điều chỉnh tốc độ học tập. Điều này có hai lợi ích: đầu tiên, chúng ta không còn cần phải quyết định chỉ khi một gradient đủ lớn. Thứ hai, nó tự động quy mô với độ lớn của gradient. Các tọa độ thường xuyên tương ứng với các gradient lớn được thu nhỏ đáng kể, trong khi các tọa độ khác có độ dốc nhỏ sẽ được điều trị nhẹ nhàng hơn nhiều. Trong thực tế, điều này dẫn đến một thủ tục tối ưu hóa rất hiệu quả cho quảng

[136] https://discuss.d2l.ai/t/1070

cáo tính toán và các vấn đề liên quan. Nhưng điều này che giấu một số lợi ích bổ sung vốn có trong Adagrad được hiểu rõ nhất trong bối cảnh điều kiện tiên quyết.

12.7.2 Điều hòa trước

Vấn đề tối ưu hóa lồi là tốt cho việc phân tích các đặc tính của thuật toán. Rốt cuộc, đối với hầu hết các vấn đề không lồi, rất khó để lấy được những đảm bảo lý thuyết có ý nghĩa, nhưng * trực tuyến* và * cái nhìn sâu thắng* thường tiếp tục. Chúng ta hãy nhìn vào vấn đề giảm thiểu $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$.

Như chúng ta đã thấy trong Section 12.6, có thể viết lại vấn đề này về khả năng phân hủy $\mathbf{Q} = \mathbf{U}^\top \mathbf{\Lambda}\mathbf{U}$ để đến một vấn đề đơn giản hóa nhiều trong đó mỗi tọa độ có thể được giải quyết riêng lẻ:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \mathbf{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \tag{12.7.1}$$

Ở đây chúng tôi đã sử dụng $\mathbf{x} = \mathbf{U}\mathbf{x}$ và do đó $\mathbf{c} = \mathbf{U}\mathbf{c}$. Vấn đề sửa đổi có như minimizer của nó $\bar{\mathbf{x}} = -\mathbf{\Lambda}^{-1}\bar{\mathbf{c}}$ và giá trị tối thiểu $-\frac{1}{2}\bar{\mathbf{c}}^\top \mathbf{\Lambda}^{-1}\bar{\mathbf{c}} + b$. Điều này dễ tính toán hơn nhiều vì $\mathbf{\Lambda}$ là một ma trận chéo chứa các giá trị eigenvalues của $\mathbf{Q}$.

Nếu chúng ta làm hỏng $\mathbf{c}$ một chút, chúng tôi hy vọng sẽ chỉ tìm thấy những thay đổi nhỏ trong bộ giảm thiểu f. Thật không may đây không phải là trường hợp. Trong khi những thay đổi nhỏ trong $\mathbf{c}$ dẫn đến những thay đổi nhỏ như nhau trong $\bar{\mathbf{c}}$, đây không phải là trường hợp cho minimizer f (và $\bar{f}$ tương ứng). Bất cứ khi nào giá trị eigenvalues $\mathbf{\Lambda}_i$ lớn, chúng ta sẽ chỉ thấy những thay đổi nhỏ trong $\bar{x}_i$ và tối thiểu là $\bar{f}$. Ngược lại, đối với những thay đổi nhỏ $\mathbf{\Lambda}_i$ trong $\bar{x}_i$ có thể là kịch tính. Tỷ lệ giữa eigenvalue lớn nhất và nhỏ nhất được gọi là số điều kiện của một bài toán tối ưu hóa.

$$\kappa = \frac{\mathbf{\Lambda}_1}{\mathbf{\Lambda}_d}. \tag{12.7.2}$$

Nếu điều kiện số κ lớn, rất khó để giải quyết vấn đề tối ưu hóa một cách chính xác. Chúng ta cần đảm bảo rằng chúng ta cẩn thận trong việc có được một phạm vi động lớn các giá trị đúng. Phân tích của chúng tôi dẫn đến một câu hỏi rõ ràng, mặc dù hơi ngây thơ: chúng ta không thể chỉ đơn giản là "khắc phục" vấn đề bằng cách làm biến dạng không gian sao cho tất cả các giá trị eigenvalues là 1. Về lý thuyết, điều này khá dễ dàng: chúng ta chỉ cần eigenvalues và eigenvectors của $\mathbf{Q}$ để giải phóng vấn đề từ $\mathbf{x}$ thành một trong $\mathbf{z} := \mathbf{\Lambda}^{\frac{1}{2}}\mathbf{U}\mathbf{x}$. Trong hệ tọa độ mới $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ có thể được đơn giản hóa thành $\|\mathbf{z}\|^2$. Than ôi, đây là một gợi ý khá không thực tế. Tính toán eigenvalues và eigenvectors nói chung là * nhiều hơn* đắt hơn so với giải quyết vấn đề thực tế.

Trong khi tính toán eigenvalues chính xác có thể đắt tiền, đoán chúng và tính toán chúng thậm chí có phần xấp xỉ có thể đã tốt hơn rất nhiều so với không làm bất cứ điều gì cả. Đặc biệt, chúng ta có thể sử dụng các mục chéo của $\mathbf{Q}$ và giải thích nó cho phù hợp. Đây là * nhiều* rẻ hơn so với tính toán eigenvalues.

$$\tilde{\mathbf{Q}} = \mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q})\mathbf{Q}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{Q}). \tag{12.7.3}$$

Trong trường hợp này, chúng tôi có $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij}/\sqrt{\mathbf{Q}_{ii}\mathbf{Q}_{jj}}$ và cụ thể là $\tilde{\mathbf{Q}}_{ii} = 1$ cho tất cả i. Trong hầu hết các trường hợp, điều này đơn giản hóa số điều kiện đáng kể. Ví dụ, các trường hợp chúng ta đã thảo luận trước đây, điều này sẽ loại bỏ hoàn toàn vấn đề trong tầm tay vì vấn đề được căn chỉnh trục.

Thật không may, chúng ta phải đối mặt với một vấn đề khác: trong học sâu, chúng ta thường thậm chí không có quyền truy cập vào đạo hàm thứ hai của hàm khách quan: cho $\mathbf{x} \in \mathbb{R}^d$, phái sinh thứ hai ngay cả trên một minibatch có thể yêu cầu $\mathcal{O}(d^2)$ không gian và hoạt động để tính toán, do đó làm cho nó thực tế không khả thi. Ý tưởng khéo léo của Adagrad là sử dụng một proxy cho đường chéo khó nắm bắt của Hessian vừa tương đối rẻ để tính toán và hiệu quả—độ lớn của bản thân gradient.

Để xem lý do tại sao điều này hoạt động, chúng ta hãy nhìn vào $\bar{f}(\bar{\mathbf{x}})$. Chúng tôi có điều đó

$$\partial_{\bar{\mathbf{x}}}\bar{f}(\bar{\mathbf{x}}) = \mathbf{\Lambda}\bar{\mathbf{x}} + \bar{\mathbf{c}} = \mathbf{\Lambda}\left(\bar{\mathbf{x}} - \bar{\mathbf{x}}_0\right), \tag{12.7.4}$$

trong đó $\bar{\mathbf{x}}_0$ là minimizer của $\bar{f}$. Do đó độ lớn của gradient phụ thuộc cả vào $\boldsymbol{\Lambda}$ và khoảng cách từ độ tối ưu. Nếu $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$ không thay đổi, đây sẽ là tất cả những gì cần thiết. Rốt cuộc, trong trường hợp này độ lớn của gradient $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$ đủ. Vì AdaGrad là một thuật toán gốc gradient ngẫu nhiên, chúng ta sẽ thấy gradient với phương sai nonzero ngay cả khi tối ưu. Kết quả là chúng ta có thể sử dụng phương sai của gradient một cách an toàn như một proxy giá rẻ cho thang đo của Hessian. Một phân tích kỹ lưỡng nằm ngoài phạm vi của phần này (nó sẽ là một số trang). Chúng tôi giới thiệu người đọc đến (Duchi et al., 2011) để biết chi tiết.

12.7.3 Các thuật toán

Hãy để chúng tôi chính thức hóa các cuộc thảo luận từ trên cao. Chúng ta sử dụng biến $\mathbf{s}_t$ để tích lũy phương sai gradient quá khứ như sau.

$$\begin{aligned} \mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t. \end{aligned} \tag{12.7.5}$$

Ở đây hoạt động được áp dụng tọa độ khôn ngoan. Đó là, $\mathbf{v}^2$ có mục v_i^2. Tương tự như vậy $\frac{1}{\sqrt{v}}$ có mục $\frac{1}{\sqrt{v_i}}$ và $\mathbf{u} \cdot \mathbf{v}$ có mục $u_i v_i$. Như trước η là tốc độ học tập và ϵ là một hằng số phụ gia đảm bảo rằng chúng ta không chia cho 0. Cuối cùng, chúng tôi khởi tạo $\mathbf{s}_0 = \mathbf{0}$.

Cũng giống như trong trường hợp động lượng chúng ta cần phải theo dõi một biến phụ trợ, trong trường hợp này để cho phép một tỷ lệ học tập cá nhân trên mỗi tọa độ. Điều này không làm tăng chi phí của Adagrad đáng kể so với SGD, đơn giản vì chi phí chính thường là tính toán $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$ và phái sinh của nó.

Lưu ý rằng tích lũy gradient bình phương trong $\mathbf{s}_t$ có nghĩa là $\mathbf{s}_t$ phát triển cơ bản ở tốc độ tuyến tính (hơi chậm hơn tuyến tính trong thực tế, vì độ dốc ban đầu giảm dần). Điều này dẫn đến tỷ lệ học tập $\mathcal{O}(t^{-\frac{1}{2}})$, mặc dù được điều chỉnh trên cơ sở tọa độ. Đối với các vấn đề lồi, điều này là hoàn toàn đầy đủ. Tuy nhiên, trong học sâu, chúng ta có thể muốn giảm tốc độ học tập khá chậm hơn. Điều này dẫn đến một số biến thể Adagrad mà chúng ta sẽ thảo luận trong các chương tiếp theo. Bây giờ chúng ta hãy xem nó hoạt động như thế nào trong một bài toán lồi bậc hai. Chúng tôi sử dụng cùng một vấn đề như trước đây:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{12.7.6}$$

Chúng tôi sẽ thực hiện Adagrad bằng cách sử dụng cùng một tốc độ học tập trước đây, tức là, $\eta = 0.4$. Như chúng ta có thể thấy, quỹ đạo lặp đi lặp lại của biến độc lập là mượt mà hơn. Tuy nhiên, do hiệu ứng tích lũy của s_t, tốc độ học tập liên tục phân rã, do đó biến độc lập không di chuyển nhiều trong các giai đoạn lặp sau này.

```
%matplotlib inline
import math
import torch
from d2l import torch as d2l
```

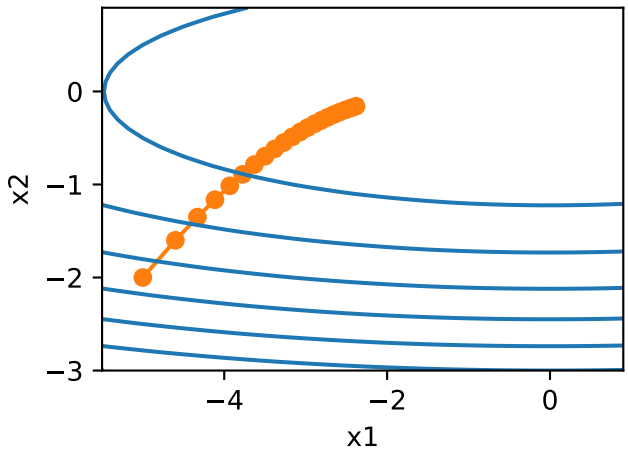
```
def adagrad_2d(x1, x2, s1, s2):
    eps = 1e-6
    g1, g2 = 0.2 * x1, 4 * x2
    s1 += g1 ** 2
    s2 += g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
```

(continues on next page)

(continued from previous page)

```
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta = 0.4
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

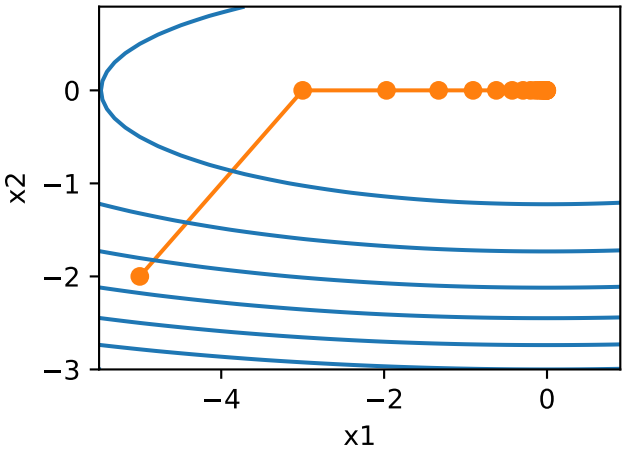
```
epoch 20, x1: -2.382563, x2: -0.158591
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Khi chúng ta tăng tỷ lệ học tập lên 2, chúng ta thấy hành vi tốt hơn nhiều. Điều này đã chỉ ra rằng tỷ lệ học tập giảm có thể khá tích cực, ngay cả trong trường hợp không có tiếng ồn và chúng ta cần đảm bảo rằng các tham số hội tụ một cách thích hợp.

```
eta = 2
d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1: -0.002295, x2: -0.000000
```



12.7.4 Thực hiện từ đầu

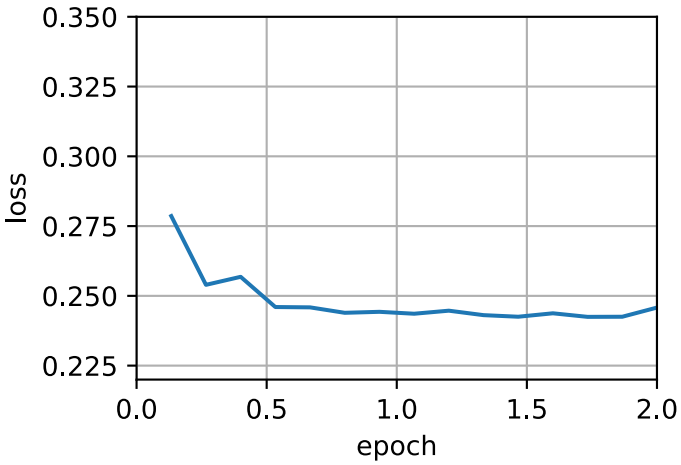
Cũng giống như phương pháp động lượng, Adagrad cần duy trì một biến trạng thái có hình dạng giống như các tham số.

```
def init_adagrad_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] += torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

So với thí nghiệm trong Section 12.5, chúng tôi sử dụng tốc độ học tập lớn hơn để đào tạo mô hình.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adagrad, init_adagrad_states(feature_dim),
               {'lr': 0.1}, data_iter, feature_dim);
```

```
loss: 0.246, 0.014 sec/epoch
```

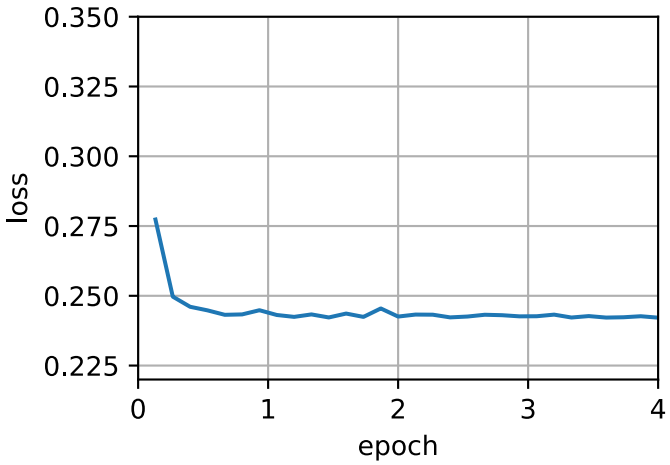


12.7.5 Thực hiện ngắn gọn

Sử dụng phiên bản `Trainer` của thuật toán `adagrad`, chúng ta có thể gọi thuật toán Adagrad trong Gluon.

```
trainer = torch.optim.Adagrad
d2l.train_concise_ch11(trainer, {'lr': 0.1}, data_iter)
```

```
loss: 0.242, 0.013 sec/epoch
```



12.7.6 Tóm tắt

- Adagrad giảm tốc độ học tập động trên cơ sở mỗi tọa độ.
- Nó sử dụng độ lớn của gradient như một phương tiện để điều chỉnh mức độ nhanh chóng đạt được tiến bộ - tọa độ với độ dốc lớn được bù với tốc độ học tập nhỏ hơn.
- Tính toán đạo hàm thứ hai chính xác thường không khả thi trong các bài toán học sâu do các ràng buộc về trí nhớ và tính toán. Gradient có thể là một proxy hữu ích.
- Nếu vấn đề tối ưu hóa có cấu trúc khá không đồng đều Adagrad có thể giúp giảm thiểu sự biến dạng.

- Adagrad đặc biệt hiệu quả đối với các tính năng thưa thớt, nơi tỷ lệ học tập cần giảm chậm hơn cho các thuật ngữ không thường xuyên xảy ra.
- Về các vấn đề học sâu Adagrad đôi khi có thể quá tích cực trong việc giảm tỷ lệ học tập. Chúng tôi sẽ thảo luận về các chiến lược để giảm thiểu điều này trong bối cảnh Section 12.10.

12.7.7 Bài tập

1. Chứng minh rằng đối với một ma trận trực giao $\mathbf{U}$ và một vector $\mathbf{c}$ các tổ chức sau đây: $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$. Tại sao điều này có nghĩa là độ lớn của nhiễu loạn không thay đổi sau khi thay đổi trực giao của các biến?
2. Hãy thử Adagrad cho $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ và cũng cho chức năng mục tiêu đã được xoay 45 độ, tức là, $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. Nó có cư xử khác nhau không?
3. Chứng minh định lý vòng tròn Gerschgorin[137] trong đó nói rằng eigenvalues λ_i của một ma trận $\mathbf{M}$ thỏa mãn $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ cho ít nhất một lựa chọn j.
4. Định lý Gerschgorin cho chúng ta biết gì về giá trị eigenvalues của ma trận điều hòa trước theo đường chéo $\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\mathrm{diag}^{-\frac{1}{2}}(\mathbf{M})$?
5. Hãy thử Adagrad cho một mạng sâu thích hợp, chẳng hạn như Section 7.6 khi áp dụng cho Fashion MNIST.
6. Làm thế nào bạn cần sửa đổi Adagrad để đạt được một sự phân rã ít hung hăng trong tốc độ học tập?

Discussions[138]

12.8 RMSProp

Một trong những vấn đề chính trong Section 12.7 là tốc độ học tập giảm theo lịch trình xác định trước là $\mathcal{O}(t^{-\frac{1}{2}})$ hiệu quả. Mặc dù điều này thường thích hợp cho các vấn đề lồi, nhưng nó có thể không lý tưởng cho những vấn đề không lồi, chẳng hạn như những vấn đề gặp phải trong học sâu. Tuy nhiên, sự thích nghi phối hợp khôn ngoan của Adagrad là rất mong muốn như một preconditioner.

(Tieleman & Hinton, 2012) đề xuất thuật toán RMSProp như một sửa chữa đơn giản để tách lập kế hoạch tỷ lệ từ tốc độ học tập thích ứng phối hợp. Vấn đề là Adagrad tích lũy các ô vuông của gradient $\mathbf{g}_t$ thành một vector trạng thái $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$. Kết quả là $\mathbf{s}_t$ tiếp tục phát triển mà không bị ràng buộc do thiếu bình thường hóa, về cơ bản tuyến tính khi thuật toán hội tụ.

Một cách để khắc phục vấn đề này là sử dụng $\mathbf{s}_t/t$. Đối với các phân phối hợp lý của $\mathbf{g}_t$, điều này sẽ hội tụ. Thật không may, nó có thể mất một thời gian rất dài cho đến khi hành vi giới hạn bắt đầu quan trọng vì thủ tục ghi nhớ quỹ đạo đầy đủ của các giá trị. Một cách khác là sử dụng trung bình rò rỉ theo cùng một cách chúng ta đã sử dụng trong phương pháp động lượng, tức là $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1-\gamma)\mathbf{g}_t^2$ cho một số tham số $\gamma > 0$. Giữ tất cả các bộ phận khác không thay đổi năng suất RMSProp.

[137] https://en.wikipedia.org/wiki/Gershgorin_circle_theorem
[138] https://discuss.d2l.ai/t/1072

12.8.1 Các thuật toán

Hãy để chúng tôi viết ra các phương trình một cách chi tiết.

$$\begin{aligned} \mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t. \end{aligned} \tag{12.8.1}$$

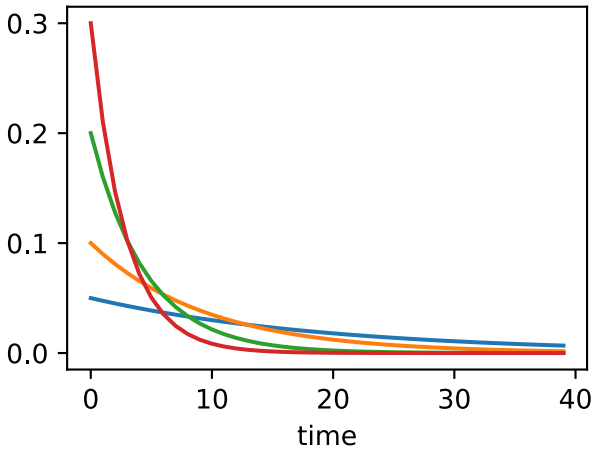
Hằng số $\epsilon > 0$ thường được đặt thành 10^{-6} để đảm bảo rằng chúng tôi không bị phân chia bằng 0 hoặc quá lớn kích thước bước. Với sự mở rộng này, chúng tôi hiện có thể tự do kiểm soát tốc độ học tập η độc lập với tỷ lệ được áp dụng trên cơ sở mỗi tọa độ. Về mặt trung bình rò rỉ, chúng ta có thể áp dụng lý luận tương tự như được áp dụng trước đây trong trường hợp phương pháp động lượng. Mở rộng định nghĩa về sản lượng $\mathbf{s}_t$

$$\begin{aligned} \mathbf{s}_t &= (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ &= (1 - \gamma) \left(\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2} + \ldots, \right). \end{aligned} \tag{12.8.2}$$

Như trước đây trong Section 12.6 chúng tôi sử dụng $1 + \gamma + \gamma^2 + \ldots, = \frac{1}{1-\gamma}$. Do đó tổng trọng lượng được chuẩn hóa thành 1 với thời gian bán hủy của một quan sát γ^{-1}. Chúng ta hãy hình dung trọng lượng trong 40 bước thời gian qua cho các lựa chọn khác nhau của γ.

```
import math
import torch
from d2l import torch as d2l
```

```
d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
for gamma in gammas:
    x = torch.arange(40).detach().numpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');
```

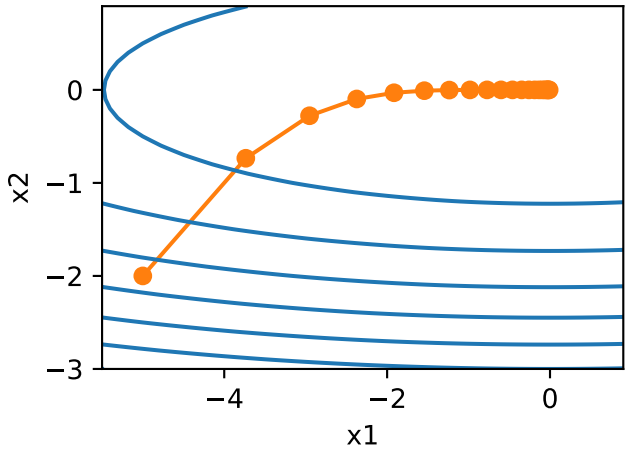


12.8.2 Thực hiện từ đầu

Như trước đây chúng ta sử dụng hàm bậc hai $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ để quan sát quỹ đạo của RMSProp. Nhớ lại rằng trong Section 12.7, khi chúng ta sử dụng Adagrad với tốc độ học 0.4, các biến chỉ di chuyển rất chậm trong các giai đoạn sau của thuật toán vì tốc độ học tập giảm quá nhanh. Vì η được kiểm soát riêng, điều này không xảy ra với RMSProp.

```
def rmsprop_2d(x1, x2, s1, s2):
    g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
    s1 = gamma * s1 + (1 - gamma) * g1 ** 2
    s2 = gamma * s2 + (1 - gamma) * g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

eta, gamma = 0.4, 0.9
d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))
```

```
epoch 20, x1: -0.010599, x2: 0.000000
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Tiếp theo, chúng tôi triển khai RMSProp để được sử dụng trong một mạng sâu. Điều này cũng đơn giản như nhau.

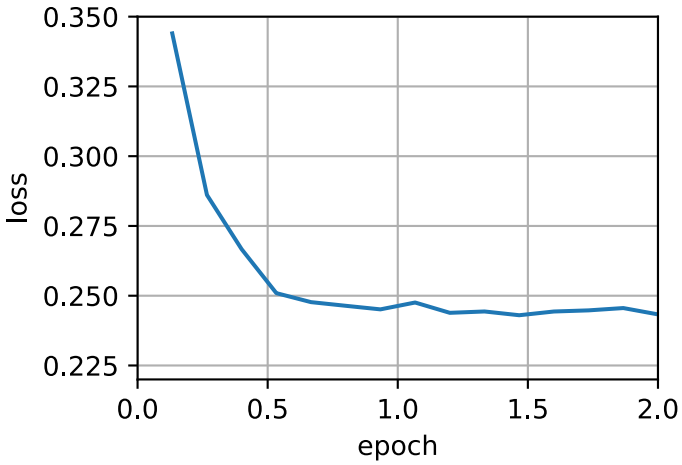
```
def init_rmsprop_states(feature_dim):
    s_w = torch.zeros((feature_dim, 1))
    s_b = torch.zeros(1)
    return (s_w, s_b)
```

```
def rmsprop(params, states, hyperparams):
    gamma, eps = hyperparams['gamma'], 1e-6
    for p, s in zip(params, states):
        with torch.no_grad():
            s[:] = gamma * s + (1 - gamma) * torch.square(p.grad)
            p[:] -= hyperparams['lr'] * p.grad / torch.sqrt(s + eps)
        p.grad.data.zero_()
```

Chúng tôi đặt tỷ lệ học tập ban đầu là 0,01 và thuật ngữ trọng số γ thành 0.9. Đó là, $\mathbf{s}$ tổng hợp trung bình trong quá khứ $1/(1-\gamma) = 10$ quan sát của gradient vuông.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(rmsprop, init_rmsprop_states(feature_dim),
               {'lr': 0.01, 'gamma': 0.9}, data_iter, feature_dim);
```

```
loss: 0.243, 0.013 sec/epoch
```

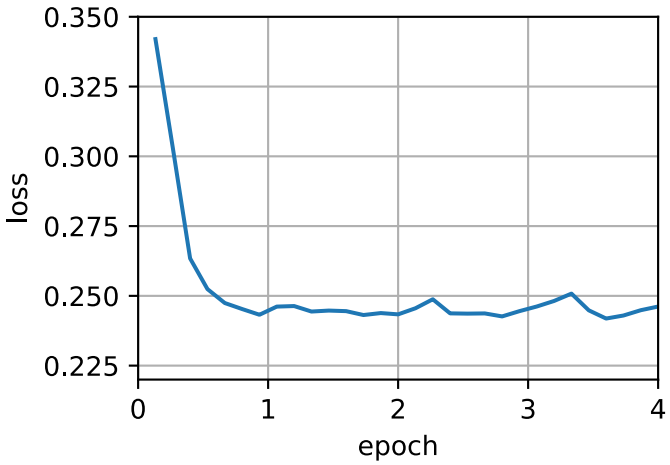


12.8.3 Thực hiện ngắn gọn

Vì RMSProp là một thuật toán khá phổ biến, nó cũng có sẵn trong phiên bản `Trainer`. Tất cả những gì chúng ta cần làm là khởi tạo nó bằng cách sử dụng một thuật toán có tên `rmsprop`, gán γ cho tham số `gamma1`.

```
trainer = torch.optim.RMSprop
d2l.train_concise_ch11(trainer, {'lr': 0.01, 'alpha': 0.9},
                       data_iter)
```

```
loss: 0.246, 0.012 sec/epoch
```



12.8.4 Tóm tắt

- RMSProp rất giống với Adagrad trong chừng mực vì cả hai đều sử dụng hình vuông của gradient để quy mô các hệ số.
- RMSProp chia sẻ với động lượng trung bình bị rò rỉ. Tuy nhiên, RMSProp sử dụng kỹ thuật này để điều chỉnh tiền điều hòa hệ số khôn ngoan.
- Tỷ lệ học tập cần được lên lịch bởi các thí nghiệm trong thực tế.
- Hệ số γ xác định lịch sử trong bao lâu khi điều chỉnh thang đo mỗi tọa độ.

12.8.5 Bài tập

1. Điều gì xảy ra bằng thực nghiệm nếu chúng ta đặt $\gamma = 1$? Tại sao?
2. Xoay vấn đề tối ưu hóa để giảm thiểu $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$. Điều gì xảy ra với sự hội tụ?
3. Hãy thử những gì xảy ra với RMSProp về một vấn đề máy học thực sự, chẳng hạn như đào tạo về Fashion-MNIST. Thử nghiệm với các lựa chọn khác nhau để điều chỉnh tốc độ học tập.
4. Bạn có muốn điều chỉnh γ khi tối ưu hóa tiến triển không? RMSProp nhạy cảm như thế nào đối với điều này?

Discussions[139]

[139] https://discuss.d2l.ai/t/1074

12.9 Adadelta

Adadelta là một biến thể khác của AdaGrad (Section 12.7). Sự khác biệt chính nằm ở thực tế là nó làm giảm số tiền mà tốc độ học tập thích ứng với tọa độ. Hơn nữa, theo truyền thống, nó được gọi là không có tỷ lệ học tập vì nó sử dụng số lượng thay đổi chính nó làm hiệu chuẩn cho sự thay đổi trong tương lai. Thuật toán được đề xuất vào năm (Zeiler, 2012). Nó khá đơn giản, được thảo luận về các thuật toán trước đó cho đến nay.

12.9.1 Các thuật toán

Tóm lại, Adadelta sử dụng hai biến trạng thái, $\mathbf{s}_t$ để lưu trữ trung bình bị rò rỉ của khoảnh khắc thứ hai của gradient và $\Delta\mathbf{x}_t$ để lưu trữ trung bình bị rò rỉ của khoảnh khắc thứ hai của sự thay đổi các tham số trong chính mô hình. Lưu ý rằng chúng tôi sử dụng ký hiệu ban đầu và đặt tên của các tác giả để tương thích với các ấn phẩm và triển khai khác (không có lý do thực sự nào khác tại sao người ta nên sử dụng các biến Hy Lạp khác nhau để chỉ ra một tham số phục vụ cùng một mục đích trong đà, Adagrad, RMSProp và Adadelta).

Dưới đây là các chi tiết kỹ thuật của Adadelta. Với tham số du jour là ρ, chúng tôi có được các bản cập nhật rò rỉ sau tương tự như Section 12.8:

$$\mathbf{s}_t = \rho \mathbf{s}_{t-1} + (1 - \rho)\mathbf{g}_t^2. \tag{12.9.1}$$

Sự khác biệt đối với Section 12.8 là chúng tôi thực hiện các bản cập nhật với gradient $\mathbf{g}_t'$ được thay đổi lại, tức là,

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{12.9.2}$$

Vậy gradient được rescaled $\mathbf{g}_t'$ là gì? Chúng ta có thể tính toán nó như sau:

$$\mathbf{g}_t' = \frac{\sqrt{\Delta\mathbf{x}_{t-1} + \epsilon}}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{12.9.3}$$

trong đó $\Delta\mathbf{x}_{t-1}$ là mức trung bình bị rò rỉ của gradient được định lại bình phương $\mathbf{g}_t'$. Chúng tôi khởi tạo $\Delta\mathbf{x}_0$ là 0 và cập nhật nó ở mỗi bước với $\mathbf{g}_t'$, tức là,

$$\Delta\mathbf{x}_t = \rho\Delta\mathbf{x}_{t-1} + (1 - \rho){\mathbf{g}_t'}^2, \tag{12.9.4}$$

và ϵ (một giá trị nhỏ như 10^{-5}) được thêm vào để duy trì sự ổn định số.

12.9.2 Thực hiện

Adadelta cần duy trì hai biến trạng thái cho mỗi biến, $\mathbf{s}_t$ và $\Delta\mathbf{x}_t$. Điều này mang lại việc thực hiện sau đây.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adadelta_states(feature_dim):
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    delta_w, delta_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((s_w, delta_w), (s_b, delta_b))
```

(continues on next page)

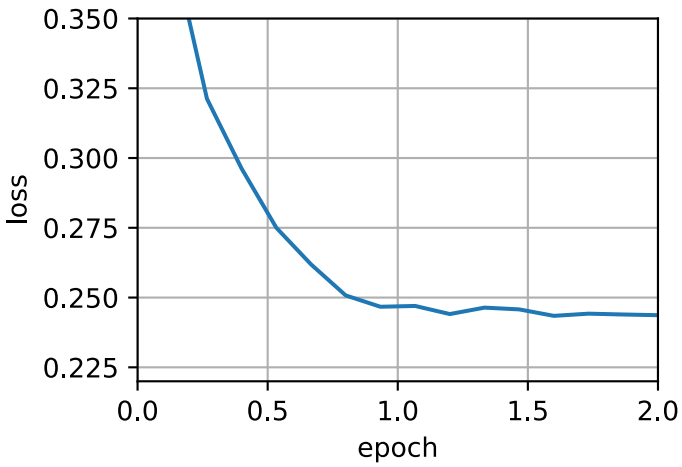
(continued from previous page)

```
def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        with torch.no_grad():
            # In-place updates via [:]
            s[:] = rho * s + (1 - rho) * torch.square(p.grad)
            g = (torch.sqrt(delta + eps) / torch.sqrt(s + eps)) * p.grad
            p[:] -= g
            delta[:] = rho * delta + (1 - rho) * g * g
        p.grad.data.zero_()
```

Chọn $\rho = 0.9$ lên tới thời gian bán hủy là 10 cho mỗi bản cập nhật tham số. Điều này có xu hướng hoạt động khá tốt. Chúng tôi nhận được hành vi sau đây.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adadelta, init_adadelta_states(feature_dim),
               {'rho': 0.9}, data_iter, feature_dim);
```

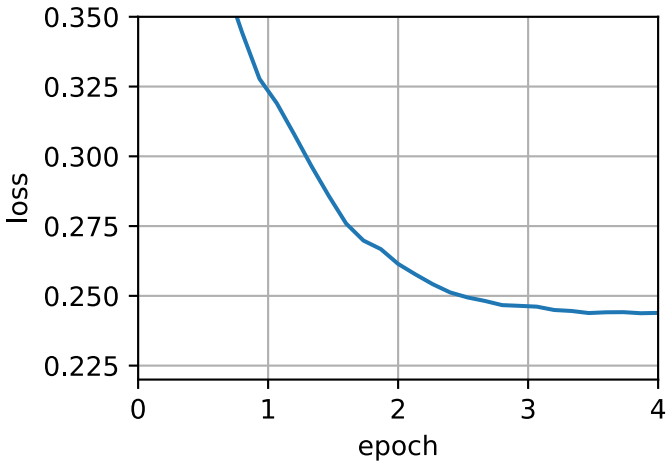
```
loss: 0.244, 0.016 sec/epoch
```



Để thực hiện ngắn gọn, chúng tôi chỉ cần sử dụng thuật toán `adadelta` từ lớp `Trainer`. Điều này mang lại một lớp lót sau đây cho một lời gọi nhỏ gọn hơn nhiều.

```
trainer = torch.optim.Adadelta
d2l.train_concise_ch11(trainer, {'rho': 0.9}, data_iter)
```

```
loss: 0.244, 0.015 sec/epoch
```



12.9.3 Tóm tắt

- Adadelta không có tham số tỷ lệ học tập. Thay vào đó, nó sử dụng tốc độ thay đổi trong chính các thông số để điều chỉnh tốc độ học tập.
- Adadelta yêu cầu hai biến trạng thái để lưu trữ những khoảnh khắc thứ hai của gradient và sự thay đổi trong các tham số.
- Adadelta sử dụng trung bình rò rỉ để giữ ước tính chạy của các số liệu thống kê thích hợp.

12.9.4 Bài tập

1. Điều chỉnh giá trị của ρ. Điều gì xảy ra?
2. Hiển thị cách thực hiện thuật toán mà không cần sử dụng $\mathbf{g}'_t$. Tại sao đây có thể là một ý tưởng tốt?
3. Adadelta có thực sự học tỷ lệ miễn phí không? Bạn có thể tìm thấy các vấn đề tối ưu hóa phá vỡ Adadelta?
4. So sánh Adadelta với Adagrad và RMS prop để thảo luận về hành vi hội tụ của họ.

Discussions[140]

12.10 Ađam

Trong các cuộc thảo luận dẫn đến phần này, chúng tôi đã gặp phải một số kỹ thuật để tối ưu hóa hiệu quả. Hãy để chúng tôi tóm tắt chi tiết chúng ở đây:

- Chúng tôi thấy rằng Section 12.4 có hiệu quả hơn Gradient Descent khi giải quyết các vấn đề tối ưu hóa, ví dụ, do khả năng phục hồi vốn có của nó đối với dữ liệu dư thừa.
- Chúng tôi thấy rằng Section 12.5 mang lại hiệu quả bổ sung đáng kể phát sinh từ vectơ hóa, sử dụng các bộ quan sát lớn hơn trong một minibatch. Đây là chìa khóa để đa máy hiệu quả, đa GPU và xử lý song song tổng thể.

[140] https://discuss.d2l.ai/t/1076

- Section 12.6 đã thêm một cơ chế để tổng hợp lịch sử của các gradient trong quá khứ để đẩy nhanh sự hội tụ.
- Section 12.7 sử dụng tỷ lệ cho mỗi tọa độ để cho phép một preconditioner hiệu quả tính toán.
- Section 12.8 tách tỷ lệ cho mỗi tọa độ từ một điều chỉnh tốc độ học tập.

Adam (Kingma & Ba, 2014) kết hợp tất cả các kỹ thuật này thành một thuật toán học tập hiệu quả. Đúng như dự đoán, đây là một thuật toán đã trở nên khá phổ biến như một trong những thuật toán tối ưu hóa mạnh mẽ và hiệu quả hơn để sử dụng trong học sâu. Nó không phải là không có vấn đề, mặc dù. Đặc biệt, (Reddi et al., 2019) cho thấy có những tình huống mà Adam có thể phân kỳ do kiểm soát phương sai kém. Trong một công việc tiếp theo (Zaheer et al., 2018) đã đề xuất một hotfix cho Adam, được gọi là Yogi giải quyết những vấn đề này. Thêm về điều này sau. Bây giờ chúng ta hãy xem xét thuật toán Adam.

12.10.1 Các thuật toán

Một trong những thành phần chính của Adam là nó sử dụng các đường trung bình động có trọng số mũ (còn được gọi là tính trung bình rò rỉ) để có được ước tính cả động lượng và cũng là khoảnh khắc thứ hai của gradient. Đó là, nó sử dụng các biến trạng thái

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned} \tag{12.10.1}$$

Ở đây β_1 và β_2 là các thông số trọng số không âm. Các lựa chọn phổ biến cho họ là $\beta_1 = 0.9$ và $\beta_2 = 0.999$. Đó là, ước tính phương sai di chuyển * chậm hơn* so với thuật ngữ động lượng. Lưu ý rằng nếu chúng ta khởi tạo $\mathbf{v}_0 = \mathbf{s}_0 = 0$, chúng ta có một lượng thiên vị đáng kể ban đầu hướng tới các giá trị nhỏ hơn. Điều này có thể được giải quyết bằng cách sử dụng thực tế là $\sum_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ để bình thường hóa các điều khoản. Tương ứng với các biến trạng thái bình thường được đưa ra bởi

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}. \tag{12.10.2}$$

Được trang bị các ước tính thích hợp bây giờ chúng ta có thể viết ra các phương trình cập nhật. Đầu tiên, chúng tôi phát lại gradient theo cách rất giống với RMSProp để có được

$$\mathbf{g}_t' = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}. \tag{12.10.3}$$

Không giống như RMSProp, bản cập nhật của chúng tôi sử dụng động lượng $\hat{\mathbf{v}}_t$ thay vì chính gradient. Hơn nữa, có một sự khác biệt nhỏ về mỹ phẩm khi việc tái cặn xảy ra bằng cách sử dụng $\frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$ thay vì $\frac{1}{\sqrt{\hat{\mathbf{s}}_t + \epsilon}}$. Các tác phẩm trước đây được cho là tốt hơn một chút trong thực tế, do đó độ lệch so với RMSProp. Thông thường, chúng tôi chọn $\epsilon = 10^{-6}$ để đánh đổi tốt giữa độ ổn định số và độ trung thực.

Bây giờ chúng tôi có tất cả các phần tại chỗ để tính toán các bản cập nhật. Đây là một chút anticlimactic và chúng tôi có một bản cập nhật đơn giản của biểu mẫu

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}_t'. \tag{12.10.4}$$

Xem xét thiết kế của Adam cảm hứng của nó là rõ ràng. Động lượng và quy mô có thể nhìn thấy rõ trong các biến trạng thái. Định nghĩa khá đặc biệt của họ buộc chúng ta phải debias thuật ngữ (điều này có thể được khắc phục bởi một điều kiện khởi tạo và cập nhật hơi khác). Thứ hai, sự kết hợp của cả hai thuật ngữ là khá đơn giản, cho RMSProp. Cuối cùng, tỷ lệ học tập rõ ràng η cho phép chúng ta kiểm soát độ dài bước để giải quyết các vấn đề hội tụ.

12.10.2 Thực hiện

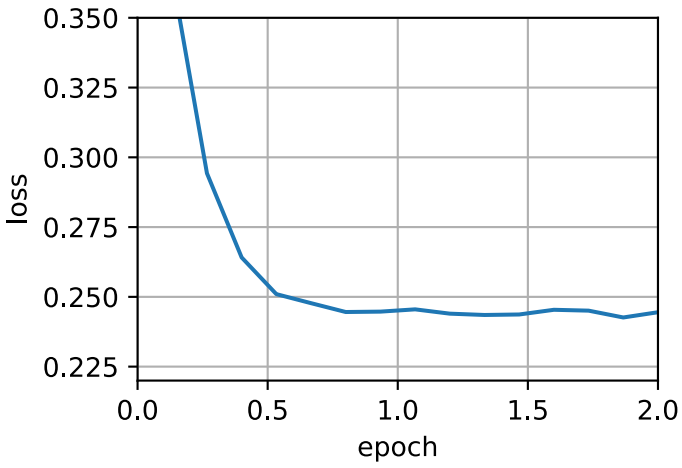
Thực hiện Adam từ đầu không phải là rất khó khăn. Để thuận tiện, chúng tôi lưu trữ thời gian bước truy cập t trong từ điển `hyperparams`. Ngoài ra tất cả là đơn giản.

```
%matplotlib inline
import torch
from d2l import torch as d2l


def init_adam_states(feature_dim):
    v_w, v_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    s_w, s_b = torch.zeros((feature_dim, 1)), torch.zeros(1)
    return ((v_w, s_w), (v_b, s_b))

def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = beta2 * s + (1 - beta2) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
        p.grad.data.zero_()
    hyperparams['t'] += 1
```

Chúng tôi đã sẵn sàng sử dụng Adam để đào tạo mô hình. Chúng tôi sử dụng tỷ lệ học tập là $\eta = 0.01$.

```
data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(adam, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.244, 0.019 sec/epoch
```

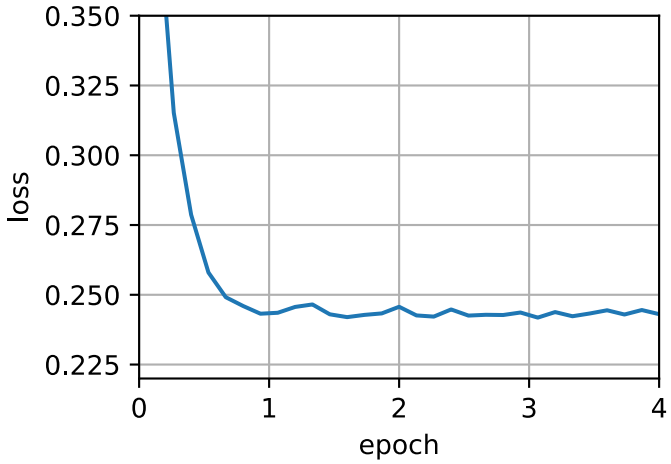


Một triển khai ngắn gọn hơn là đơn giản vì `adam` là một trong những thuật toán được cung cấp như một phần của thư viện tối ưu hóa Gluon `trainer`. Do đó chúng ta chỉ cần vượt qua các tham số cấu hình để thực hiện

trong Gluon.

```
trainer = torch.optim.Adam
d2l.train_concise_ch11(trainer, {'lr': 0.01}, data_iter)
```

```
loss: 0.243, 0.017 sec/epoch
```



12.10.3 Yogi

Một trong những vấn đề của Adam là nó có thể không hội tụ ngay cả trong các cài đặt lồi khi ước tính khoảnh khắc thứ hai trong $\mathbf{s}_t$ nổ tung. Như một sửa chữa (Zaheer et al., 2018) đề xuất một bản cập nhật tinh chế (và khởi tạo) cho $\mathbf{s}_t$. Để hiểu những gì đang xảy ra, chúng ta hãy viết lại bản cập nhật Adam như sau:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \left(\mathbf{g}_t^2 - \mathbf{s}_{t-1}\right). \tag{12.10.5}$$

Bất cứ khi nào $\mathbf{g}_t^2$ có phương sai cao hoặc cập nhật thưa thớt, $\mathbf{s}_t$ có thể quên các giá trị quá khứ quá nhanh. Một sửa chữa có thể cho điều này là thay thế $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$ bởi $\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$. Bây giờ độ lớn của bản cập nhật không còn phụ thuộc vào số lượng độ lệch. Điều này mang lại các bản cập nhật Yogi

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \odot \mathrm{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \tag{12.10.6}$$

Các tác giả hơn nữa khuyên nên khởi tạo động lượng trên một lô ban đầu lớn hơn là chỉ ước tính theo chiều ngang ban đầu. Chúng tôi bỏ qua các chi tiết vì chúng không phải là vật chất đối với cuộc thảo luận và vì ngay cả khi không có sự hội tụ này vẫn còn khá tốt.

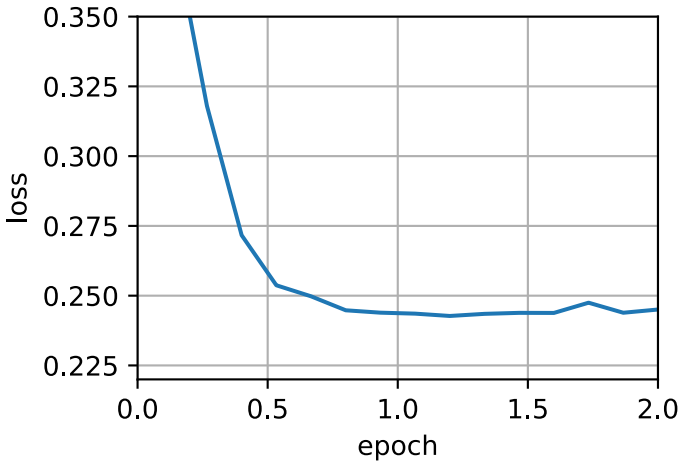
```
def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        with torch.no_grad():
            v[:] = beta1 * v + (1 - beta1) * p.grad
            s[:] = s + (1 - beta2) * torch.sign(
                torch.square(p.grad) - s) * torch.square(p.grad)
            v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
            s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
            p[:] -= hyperparams['lr'] * v_bias_corr / (torch.sqrt(s_bias_corr)
                                                       + eps)
```

(continues on next page)

(continued from previous page)

```
        p.grad.data.zero_()
    hyperparams['t'] += 1

data_iter, feature_dim = d2l.get_data_ch11(batch_size=10)
d2l.train_ch11(yogi, init_adam_states(feature_dim),
               {'lr': 0.01, 't': 1}, data_iter, feature_dim);
```

```
loss: 0.245, 0.019 sec/epoch
```



12.10.4 Tóm tắt

- Adam kết hợp các tính năng của nhiều thuật toán tối ưu hóa thành một quy tắc cập nhật khá mạnh mẽ.
- Được tạo ra trên cơ sở RMSProp, Adam cũng sử dụng EWMA trên gradient stochastic minibatch.
- Adam sử dụng hiệu chỉnh thiên vị để điều chỉnh cho một khởi động chậm khi ước tính động lượng và khoảnh khắc thứ hai.
- Đối với gradient có phương sai đáng kể, chúng ta có thể gặp phải các vấn đề với sự hội tụ. Chúng có thể được sửa đổi bằng cách sử dụng minibatches lớn hơn hoặc bằng cách chuyển sang ước tính được cải thiện cho $\mathbf{s}_t$. Yogi cung cấp một sự thay thế như vậy.

12.10.5 Bài tập

1. Điều chỉnh tốc độ học tập và quan sát và phân tích kết quả thử nghiệm.
2. Bạn có thể viết lại đà và cập nhật khoảnh khắc thứ hai sao cho nó không yêu cầu điều chỉnh thiên vị?
3. Tại sao bạn cần giảm tỷ lệ học tập η khi chúng ta hội tụ?
4. Cố gắng xây dựng một trường hợp mà Adam phân kỳ và Yogi hội tụ?

Discussions[141]

[141] https://discuss.d2l.ai/t/1078

12.11 Lập kế hoạch tỷ lệ học tập

Cho đến nay chúng tôi chủ yếu tập trung vào tối ưu hóa các thuật toán * để làm thế nào để cập nhật các vectơ trọng lượng thay vì trên * tỷ lệ* tại đó chúng đang được cập nhật. Tuy nhiên, việc điều chỉnh tốc độ học tập thường cũng quan trọng như thuật toán thực tế. Có một số khía cạnh cần xem xét:

- Rõ ràng nhất là *độ lượng* của tỷ lệ học tập quan trọng. Nếu nó quá lớn, tối ưu hóa phân kỳ, nếu nó quá nhỏ, phải mất quá nhiều thời gian để đào tạo hoặc chúng ta kết thúc với một kết quả tối ưu. Trước đây chúng tôi đã thấy rằng số điều kiện của vấn đề quan trọng (xem ví dụ, Section 12.6 để biết chi tiết). Trực giác đó là tỷ lệ giữa số lượng thay đổi theo hướng ít nhạy cảm nhất so với một nhạy cảm nhất.
- Thứ hai, tỷ lệ phân rã cũng quan trọng như vậy. Nếu tỷ lệ học tập vẫn lớn, chúng tôi có thể chỉ đơn giản là kết thúc nảy xung quanh mức tối thiểu và do đó không đạt được sự tối ưu. Section 12.5 đã thảo luận chi tiết về điều này và chúng tôi đã phân tích đảm bảo hiệu suất trong Section 12.4. Nói tóm lại, chúng tôi muốn tỷ lệ phân rã, nhưng có lẽ chậm hơn $\mathcal{O}(t^{-\frac{1}{2}})$, đây sẽ là một lựa chọn tốt cho các vấn đề lồi.
- Một khía cạnh khác quan trọng không kém là * khởi hóa*. Điều này liên quan đến cả cách các tham số được đặt ban đầu (xem lại Section 5.8 để biết chi tiết) và cũng như cách chúng phát triển ban đầu. Điều này đi theo moniker của * warmup*, tức là, nhanh chóng như thế nào chúng ta bắt đầu di chuyển về phía giải pháp ban đầu. Các bước lớn ngay từ đầu có thể không có lợi, đặc biệt là vì tập hợp các tham số ban đầu là ngẫu nhiên. Các hướng cập nhật ban đầu cũng có thể khá vô nghĩa.
- Cuối cùng, có một số biến thể tối ưu hóa thực hiện điều chỉnh tỷ lệ học tập theo chu kỳ. Điều này nằm ngoài phạm vi của chương hiện tại. Chúng tôi khuyên người đọc xem lại chi tiết trong (Izmailov et al., 2018), ví dụ, làm thế nào để có được các giải pháp tốt hơn bằng cách trung bình trên toàn bộ một * path* của các tham số.

Với thực tế là có rất nhiều chi tiết cần thiết để quản lý tỷ lệ học tập, hầu hết các khuôn khổ học sâu đều có các công cụ để xử lý điều này một cách tự động. Trong chương hiện tại, chúng tôi sẽ xem xét các hiệu ứng mà các lịch trình khác nhau có về độ chính xác và cũng cho thấy cách điều này có thể được quản lý hiệu quả thông qua một *học tỷ lệ lịch học*.

12.11.1 Vấn đề Toy

Chúng tôi bắt đầu với một vấn đề đồ chơi đủ rẻ để tính toán dễ dàng, nhưng đủ không tầm thường để minh họa một số khía cạnh chính. Đối với điều đó, chúng tôi chọn một phiên bản hơi hiện đại hóa của LeNet (`relu` thay vì kích hoạt `sigmoid`, MaxPooling thay vì AveragePooling), như áp dụng cho Fashion-MNIST. Hơn nữa, chúng tôi lai mạng để thực hiện. Vì hầu hết các mã là tiêu chuẩn, chúng tôi chỉ giới thiệu những điều cơ bản mà không cần thảo luận chi tiết thêm. Xin xem Chapter 7 để được làm mới khi cần thiết.

```
%matplotlib inline
import math
import torch
from torch import nn
from torch.optim import lr_scheduler
from d2l import torch as d2l


def net_fn():
    model = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.ReLU(),
```

(continues on next page)

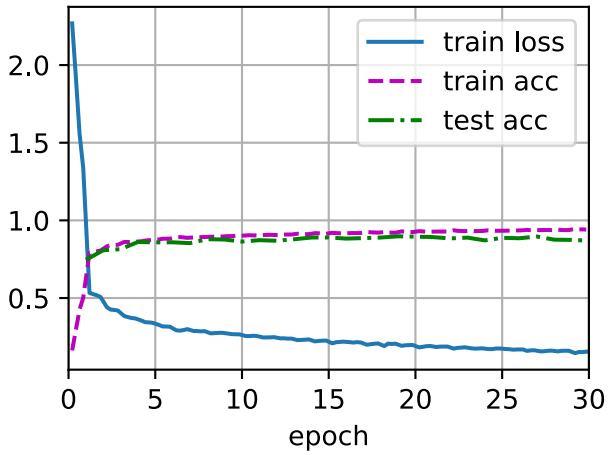
```
        nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.ReLU(),
        nn.Linear(120, 84), nn.ReLU(),
        nn.Linear(84, 10))

    return model

loss = nn.CrossEntropyLoss()
device = d2l.try_gpu()

batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)

# The code is almost identical to `d2l.train_ch6` defined in the
# lenet section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
          scheduler=None):
    net.to(device)
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])

    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)  # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            net.train()
            trainer.zero_grad()
            X, y = X.to(device), y.to(device)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            trainer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0])
            train_loss = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % 50 == 0:
                animator.add(epoch + i / len(train_iter),
                             (train_loss, train_acc, None))

        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch+1, (None, None, test_acc))

        if scheduler:
            if scheduler.__module__ == lr_scheduler.__name__:
                # Using PyTorch In-Built scheduler
                scheduler.step()
            else:
                # Using custom defined scheduler
                for param_group in trainer.param_groups:
                    param_group['lr'] = scheduler(epoch)

    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

Chúng ta hãy xem những gì sẽ xảy ra nếu chúng ta gọi thuật toán này với các cài đặt mặc định, chẳng hạn như

tốc độ học tập 0.3 và đào tạo cho 30 lặp lại. Lưu ý độ chính xác của đào tạo tiếp tục tăng lên trong khi tiến bộ về các quầy hàng chính xác thử nghiệm vượt ra ngoài một điểm. Khoảng cách giữa cả hai đường cong cho thấy quá mức.

```
lr, num_epochs = 0.3, 30
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device)
```

```
train loss 0.156, train acc 0.940, test acc 0.869
```



12.11.2 Người lập lịch

Một cách để điều chỉnh tốc độ học tập là thiết lập nó một cách rõ ràng ở mỗi bước. Điều này đạt được thuận tiện bằng phương pháp `set_learning_rate`. Chúng ta có thể điều chỉnh nó xuống sau mỗi kỷ nguyên (hoặc thậm chí sau mỗi minibatch), ví dụ, một cách năng động để đáp ứng cách tối ưu hóa đang tiến triển.

```
lr = 0.1
trainer.param_groups[0]["lr"] = lr
print(f'learning rate is now {trainer.param_groups[0]["lr"]:.2f}')
```

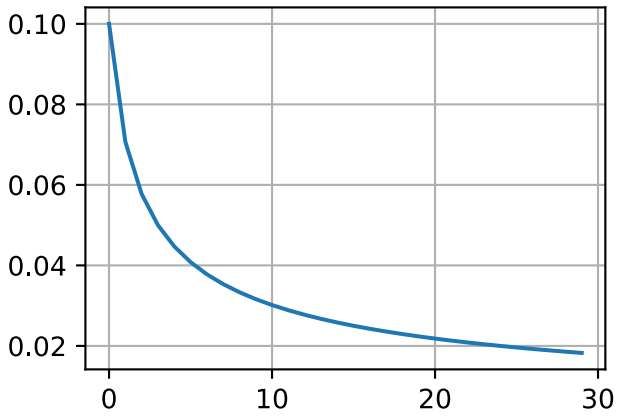
```
learning rate is now 0.10
```

Nói chung hơn chúng tôi muốn xác định một trình lập lịch. Khi được gọi với số lượng cập nhật nó trả về giá trị thích hợp của tốc độ học tập. Hãy để chúng tôi xác định một cái đơn giản đặt tỷ lệ học tập thành $\eta = \eta_0(t+1)^{-\frac{1}{2}}$.

```
class SquareRootScheduler:
    def __init__(self, lr=0.1):
        self.lr = lr

    def __call__(self, num_update):
        return self.lr * pow(num_update + 1.0, -0.5)
```

Hãy để chúng tôi vẽ hành vi của nó trên một loạt các giá trị.

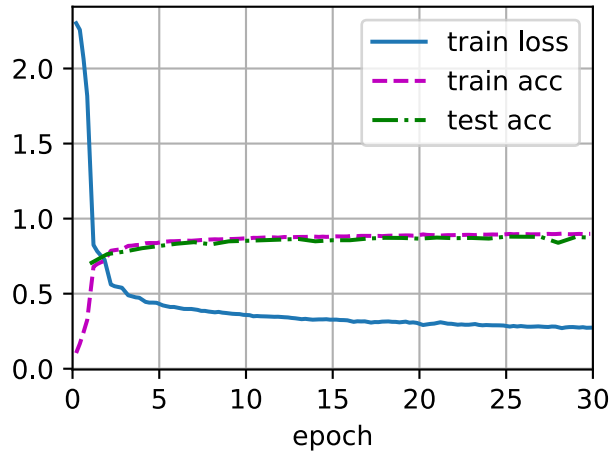
```
scheduler = SquareRootScheduler(lr=0.1)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Bây giờ chúng ta hãy xem cách điều này diễn ra để đào tạo về thời trang-MNIST. Chúng tôi chỉ cần cung cấp trình lập lịch như một đối số bổ sung cho thuật toán đào tạo.

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.275, train acc 0.898, test acc 0.873
```



Điều này làm việc khá tốt hơn một chút so với trước đây. Hai điều nổi bật: đường cong khá trơn tru hơn trước đây. Thứ hai, có ít quá nhiều hơn. Thật không may, nó không phải là một câu hỏi được giải quyết tốt về lý do tại sao một số chiến lược nhất định dẫn đến ít quá mức trong * lý đề*. Có một số lập luận rằng kích thước bước nhỏ hơn sẽ dẫn đến các tham số gần bằng 0 và do đó đơn giản hơn. Tuy nhiên, điều này không giải thích hoàn toàn hiện tượng vì chúng ta không thực sự dừng lại sớm mà chỉ đơn giản là giảm tốc độ học tập nhẹ nhàng.

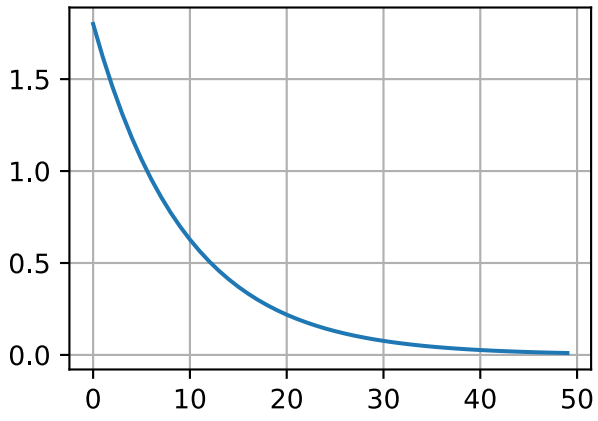
12.11.3 Chính sách

Mặc dù chúng tôi không thể bao gồm toàn bộ các lập lịch trình tỷ lệ học tập, chúng tôi cố gắng đưa ra một cái nhìn tổng quan ngắn gọn về các chính sách phổ biến dưới đây. Các lựa chọn phổ biến là phân rã đa thức và từng mảnh lịch trình liên tục. Ngoài ra, lịch trình học cosine đã được tìm thấy để làm việc tốt theo kinh nghiệm về một số vấn đề. Cuối cùng, về một số vấn đề, có lợi khi làm nóng trình tối ưu hóa trước khi sử dụng tỷ lệ học tập lớn.

Bộ lập lịch yếu tố

Một thay thế cho phân rã đa thức sẽ là một phân rã nhân, đó là $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ cho $\alpha \in (0, 1)$. Để ngăn chặn tỷ lệ học tập phân rã vượt quá giới hạn thấp hơn hợp lý, phương trình cập nhật thường được sửa đổi thành $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$.

```
class FactorScheduler:
    def __init__(self, factor=1, stop_factor_lr=1e-7, base_lr=0.1):
        self.factor = factor
        self.stop_factor_lr = stop_factor_lr
        self.base_lr = base_lr

    def __call__(self, num_update):
        self.base_lr = max(self.stop_factor_lr, self.base_lr * self.factor)
        return self.base_lr

scheduler = FactorScheduler(factor=0.9, stop_factor_lr=1e-2, base_lr=2.0)
d2l.plot(torch.arange(50), [scheduler(t) for t in range(50)])
```

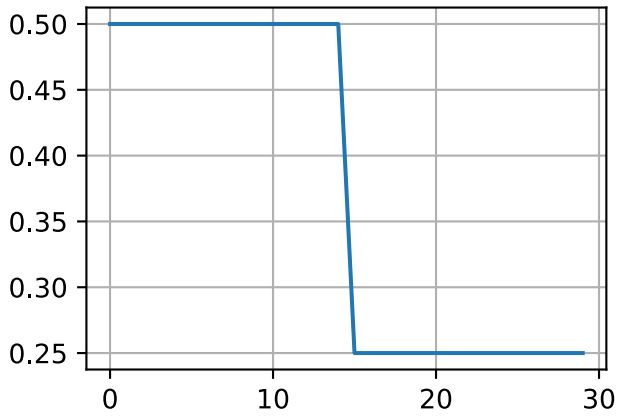


Điều này cũng có thể được thực hiện bằng trình lập lịch tích hợp trong MXNet thông qua đối tượng `lr_scheduler.FactorScheduler`. Phải mất thêm một vài tham số, chẳng hạn như thời gian khởi động, chế độ khởi động (tuyến tính hoặc không đổi), số lượng cập nhật mong muốn tối đa, v.v.; Về phía trước, chúng tôi sẽ sử dụng các bộ lập lịch tích hợp phù hợp và chỉ giải thích chức năng của chúng ở đây. Như minh họa, nó là khá đơn giản để xây dựng lịch trình của riêng bạn nếu cần thiết.

Bộ lập lịch đa yếu tố

Một chiến lược phổ biến để đào tạo các mạng sâu là giữ cho tỷ lệ học tập không đổi và giảm nó bằng một số tiền nhất định thường xuyên. Đó là, đưa ra một tập hợp thời gian khi giảm tỷ lệ, chẳng hạn như $s = \{5, 10, 20\}$ giảm $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ bất cứ khi nào $t \in s$. Giả sử rằng các giá trị giảm một nửa ở mỗi bước chúng ta có thể thực hiện điều này như sau.

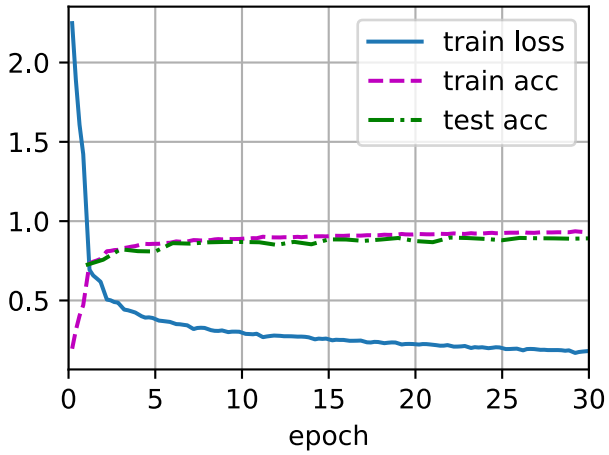
```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.5)
scheduler = lr_scheduler.MultiStepLR(trainer, milestones=[15, 30], gamma=0.5)

def get_lr(trainer, scheduler):
    lr = scheduler.get_last_lr()[0]
    trainer.step()
    scheduler.step()
    return lr

d2l.plot(torch.arange(num_epochs), [get_lr(trainer, scheduler)
                                    for t in range(num_epochs)])
```



Trực giác đằng sau lịch trình tỷ lệ học tập liên tục piecewise này là một cho phép tối ưu hóa tiến hành cho đến khi đạt được một điểm cố định về sự phân bố của vectơ trọng lượng. Sau đó (và chỉ sau đó) chúng ta có giảm tỷ lệ như để có được một proxy chất lượng cao hơn đến mức tối thiểu địa phương tốt. Ví dụ dưới đây cho thấy làm thế nào điều này có thể tạo ra các giải pháp tốt hơn bao giờ hết.

```
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.180, train acc 0.932, test acc 0.891
```



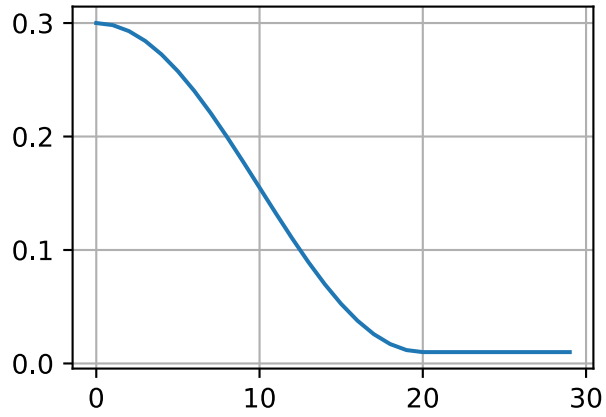
Bộ lập lịch cosine

Một heuristic khá bối rối đã được đề xuất bởi (Loshchilov & Hutter, 2016). Nó dựa vào quan sát rằng chúng ta có thể không muốn giảm tốc độ học tập quá đáng kể ngay từ đầu và hơn nữa, rằng chúng ta có thể muốn "tinh chỉnh" giải pháp cuối cùng bằng cách sử dụng một tốc độ học tập rất nhỏ. Điều này dẫn đến một lịch trình giống như cosin với dạng chức năng sau đây cho tỷ lệ học tập trong khoảng $t \in [0, T]$.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} (1 + \cos(\pi t/T)) \tag{12.11.1}$$

Ở đây η_0 là tỷ lệ học tập ban đầu, η_T là tỷ lệ mục tiêu tại thời điểm T. Hơn nữa, đối với $t > T$, chúng tôi chỉ cần ghim giá trị lên η_T mà không tăng lại. Trong ví dụ sau, chúng tôi đặt bước cập nhật tối đa $T = 20$.

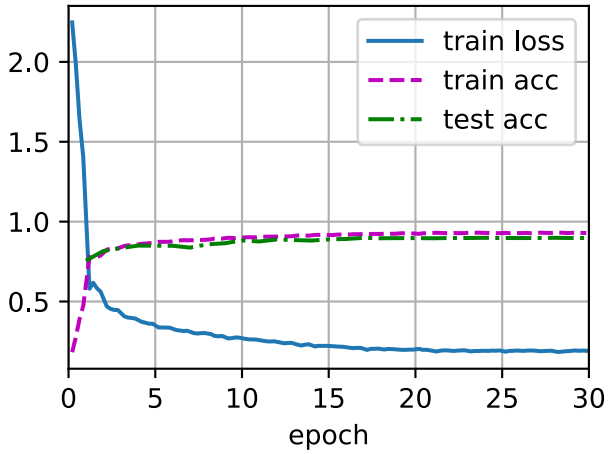
```
class CosineScheduler:
    def __init__(self, max_update, base_lr=0.01, final_lr=0,
                 warmup_steps=0, warmup_begin_lr=0):
        self.base_lr_orig = base_lr
        self.max_update = max_update
        self.final_lr = final_lr
        self.warmup_steps = warmup_steps
        self.warmup_begin_lr = warmup_begin_lr
        self.max_steps = self.max_update - self.warmup_steps

    def get_warmup_lr(self, epoch):
        increase = (self.base_lr_orig - self.warmup_begin_lr) \
                       * float(epoch) / float(self.warmup_steps)
        return self.warmup_begin_lr + increase

    def __call__(self, epoch):
        if epoch < self.warmup_steps:
            return self.get_warmup_lr(epoch)
        if epoch <= self.max_update:
            self.base_lr = self.final_lr + (
                self.base_lr_orig - self.final_lr) * (1 + math.cos(
                math.pi * (epoch - self.warmup_steps) / self.max_steps)) / 2
        return self.base_lr

scheduler = CosineScheduler(max_update=20, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Trong bối cảnh tầm nhìn máy tính, lịch trình này * có thể* dẫn đến kết quả được cải thiện. Tuy nhiên, lưu ý rằng những cải tiến như vậy không được đảm bảo (như có thể thấy dưới đây).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.190, train acc 0.929, test acc 0.898
```

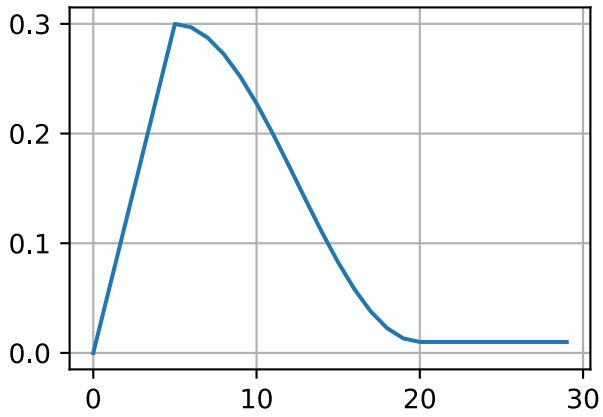


Khởi động

Trong một số trường hợp khởi tạo các tham số là không đủ để đảm bảo một giải pháp tốt. Điều này đặc biệt là một vấn đề đối với một số thiết kế mạng tiên tiến có thể dẫn đến các vấn đề tối ưu hóa không ổn định. Chúng tôi có thể giải quyết điều này bằng cách chọn một tốc độ học tập đủ nhỏ để ngăn chặn sự phân kỳ ngay từ đầu. Thật không may, điều này có nghĩa là tiến bộ chậm. Ngược lại, một tỷ lệ học tập lớn ban đầu dẫn đến sự phân kỳ.

Một khắc phục khá đơn giản cho tình huống khó xử này là sử dụng thời gian khởi động trong đó tốc độ học tập * tăng* lên mức tối đa ban đầu và để hạ nhiệt tỷ lệ cho đến khi kết thúc quá trình tối ưu hóa. Để đơn giản, người ta thường sử dụng sự gia tăng tuyến tính cho mục đích này. Điều này dẫn đến một lịch trình của biểu mẫu được chỉ định dưới đây.

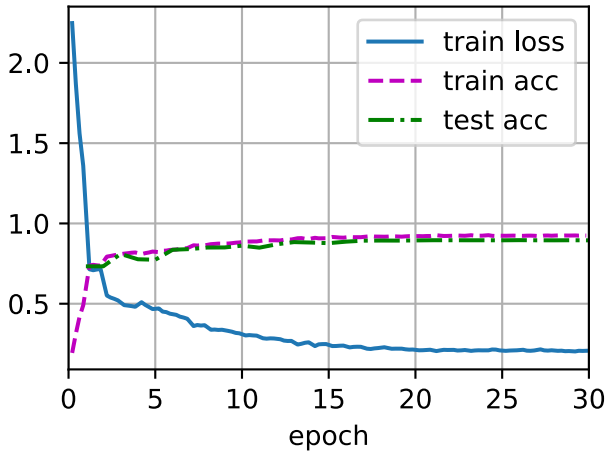
```
scheduler = CosineScheduler(20, warmup_steps=5, base_lr=0.3, final_lr=0.01)
d2l.plot(torch.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```



Lưu ý rằng mạng hội tụ tốt hơn ban đầu (đặc biệt quan sát hiệu suất trong 5 thời đại đầu tiên).

```
net = net_fn()
trainer = torch.optim.SGD(net.parameters(), lr=0.3)
train(net, train_iter, test_iter, num_epochs, loss, trainer, device,
      scheduler)
```

```
train loss 0.208, train acc 0.924, test acc 0.895
```



Khởi động có thể được áp dụng cho bất kỳ bộ lập lịch nào (không chỉ cosine). Để có một cuộc thảo luận chi tiết hơn về lịch trình học tập và nhiều thí nghiệm khác xem thêm (Gotmare et al., 2018). Đặc biệt, họ thấy rằng một giai đoạn khởi động giới hạn số lượng phân kỳ của các tham số trong các mạng rất sâu. Điều này có ý nghĩa trực giác vì chúng ta sẽ mong đợi sự phân kỳ đáng kể do khởi tạo ngẫu nhiên trong những phần của mạng mất nhiều thời gian nhất để đạt được tiến bộ ngay từ đầu.

12.11.4 Tóm tắt

- Giảm tỷ lệ học tập trong quá trình đào tạo có thể dẫn đến cải thiện độ chính xác và (khó chịu nhất) giảm quá mức của mô hình.
- Giảm từng phần tỷ lệ học tập bất cứ khi nào tiến bộ đã đạt được hiệu quả trong thực tế. Về cơ bản, điều này đảm bảo rằng chúng tôi hội tụ hiệu quả với một giải pháp phù hợp và chỉ sau đó giảm phương sai vốn có của các tham số bằng cách giảm tỷ lệ học tập.
- Cosine scheulers là phổ biến cho một số vấn đề tầm nhìn máy tính. Xem ví dụ, GluonCV[142] để biết chi tiết về trình lập lịch như vậy.
- Một thời gian khởi động trước khi tối ưu hóa có thể ngăn chặn sự phân kỳ.
- Tối ưu hóa phục vụ nhiều mục đích trong học sâu. Bên cạnh việc giảm thiểu mục tiêu đào tạo, các lựa chọn khác nhau về thuật toán tối ưu hóa và lập kế hoạch tỷ lệ học tập có thể dẫn đến số lượng tổng quát hóa và quá mức khác nhau trên bộ thử nghiệm (đối với cùng một lượng lỗi đào tạo).

12.11.5 Bài tập

1. Thử nghiệm với hành vi tối ưu hóa cho một tỷ lệ học tập cố định nhất định. Mô hình tốt nhất bạn có thể có được theo cách này là gì?
2. Làm thế nào để hội tụ thay đổi nếu bạn thay đổi số mũ của sự giảm tỷ lệ học tập? Sử dụng `PolyScheduler` để thuận tiện cho bạn trong các thí nghiệm.
3. Áp dụng bộ lập lịch cosine cho các vấn đề tầm nhìn máy tính lớn, ví dụ: đào tạo ImageNet. Làm thế nào để nó ảnh hưởng đến hiệu suất so với các lập lịch khác?
4. Thời gian khởi động nên kéo dài bao lâu?
5. Bạn có thể kết nối tối ưu hóa và lấy mẫu? Bắt đầu bằng cách sử dụng kết quả từ (Welling & Teh, 2011) trên Stochastic Gradient Langevin Dynamics.

Discussions[143]

[142] http://gluon-cv.mxnet.io
[143] https://discuss.d2l.ai/t/1080

13 | Hiệu suất tính toán

Trong học sâu, các tập dữ liệu và mô hình thường lớn, liên quan đến tính toán nặng. Do đó, hiệu suất tính toán rất quan trọng. Chương này sẽ tập trung vào các yếu tố chính ảnh hưởng đến hiệu suất tính toán: lập trình bắt buộc, lập trình tượng trưng, tính toán không đồng bộ, song song tự động và tính toán đa GPU. Bằng cách nghiên cứu chương này, bạn có thể cải thiện hơn nữa hiệu suất tính toán của các mô hình được thực hiện trong các chương trước, ví dụ, bằng cách giảm thời gian đào tạo mà không ảnh hưởng đến độ chính xác.

13.1 Trình biên dịch và phiên dịch

Cho đến nay, cuốn sách này đã tập trung vào lập trình bắt buộc, làm cho việc sử dụng các tuyên bố như `print`, `+`, và `if` để thay đổi trạng thái của một chương trình. Hãy xem xét ví dụ sau đây của một chương trình bắt buộc đơn giản.

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))
```

```
10
```

Python là một ngôn ngữ*được giải thị*. Khi đánh giá hàm `fancy_func` trên, nó thực hiện các thao tác tạo nên cơ thể của hàm * theo trình tự *. Đó là, nó sẽ đánh giá `e = add(a, b)` và lưu trữ các kết quả dưới dạng biến `e`, do đó thay đổi trạng thái của chương trình. Hai câu lệnh tiếp theo `f = add(c, d)` và `g = add(e, f)` sẽ được thực thi tương tự, thực hiện bổ sung và lưu trữ các kết quả dưới dạng biến. Fig. 13.1.1 minh họa luồng dữ liệu.

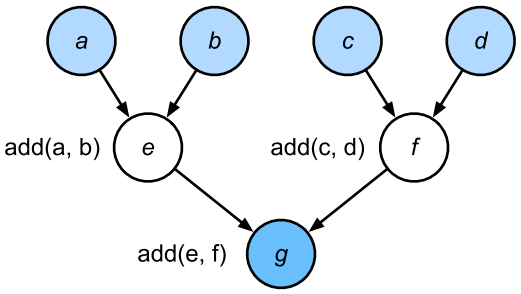


Fig. 13.1.1: Data flow in an imperative program.

Mặc dù lập trình bắt buộc là thuận tiện, nó có thể không hiệu quả. Một mặt, ngay cả khi hàm `add` được gọi nhiều lần trong suốt `fancy_func`, Python sẽ thực hiện ba cuộc gọi hàm riêng lẻ. Nếu chúng được thực thi, giả sử, trên GPU (hoặc thậm chí trên nhiều GPU), chi phí phát sinh từ trình thông dịch Python có thể trở nên áp đảo. Hơn nữa, nó sẽ cần phải lưu các giá trị biến của `e` và `f` cho đến khi tất cả các câu lệnh trong `fancy_func` đã được thực thi. Điều này là do chúng ta không biết liệu các biến `e` và `f` sẽ được sử dụng bởi các phần khác của chương trình sau khi các câu lệnh `e = add(a, b)` và `f = add(c, d)` được thực thi.

13.1.1 Lập trình tượng trưng

Hãy xem xét phương pháp thay thế, *lập trình tượng trưng, trong đó tính toán thường chỉ được thực hiện một khi quá trình đã được xác định đầy đủ. Chiến lược này được sử dụng bởi nhiều khuôn khổ học sâu, bao gồm Theano và TensorFlow (sau này đã có được các phần mở rộng bắt buộc). Nó thường bao gồm các bước sau:

1. Xác định các hoạt động sẽ được thực thi.
2. Biên dịch các hoạt động thành một chương trình thực thi.
3. Cung cấp các đầu vào cần thiết và gọi chương trình được biên dịch để thực hiện.

Điều này cho phép tối ưu hóa một lượng đáng kể. Đầu tiên, chúng ta có thể bỏ qua trình thông dịch Python trong nhiều trường hợp, do đó loại bỏ một nút cổ chai hiệu suất có thể trở nên đáng kể trên nhiều GPU nhanh được ghép nối với một luồng Python duy nhất trên CPU. Thứ hai, một trình biên dịch có thể tối ưu hóa và viết lại mã trên vào `print((1 + 2) + (3 + 4))` hoặc thậm chí `print(10)`. Điều này là có thể kể từ khi một trình biên dịch được để xem mã đầy đủ trước khi biến nó thành hướng dẫn máy. Ví dụ, nó có thể giải phóng bộ nhớ (hoặc không bao giờ phân bổ nó) bất cứ khi nào một biến không còn cần thiết nữa. Hoặc nó có thể biến đổi mã hoàn toàn thành một phần tương đương. Để có được một ý tưởng tốt hơn, hãy xem xét mô phỏng sau đây của lập trình bắt buộc (nó là Python sau tất cả) dưới đây.

```
def add_():
    return '''
def add(a, b):
    return a + b
'''

def fancy_func_():
    return '''
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
'''
```

(continues on next page)

(continued from previous page)

```
def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

```
def add(a, b):
    return a + b

def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g
print(fancy_func(1, 2, 3, 4))
10
```

Sự khác biệt giữa lập trình bắt buộc (giải thích) và lập trình tượng trưng như sau:

- Lập trình bắt buộc dễ dàng hơn. Khi lập trình bắt buộc được sử dụng trong Python, phần lớn mã là đơn giản và dễ viết. Nó cũng dễ dàng hơn để gỡ lỗi mã lập trình bắt buộc. Điều này là do dễ dàng hơn để có được và in tất cả các giá trị biến trung gian có liên quan hoặc sử dụng các công cụ gỡ lỗi tích hợp sẵn của Python.
- Lập trình tượng trưng hiệu quả hơn và dễ dàng hơn để cổng. Lập trình tượng trưng giúp tối ưu hóa mã trong quá trình biên dịch dễ dàng hơn, đồng thời có khả năng chuyển chương trình thành một định dạng độc lập với Python. Điều này cho phép chương trình được chạy trong môi trường không phải Python, do đó tránh mọi vấn đề hiệu suất tiềm ẩn liên quan đến trình thông dịch Python.

13.1.2 Lập trình lai

Trong lịch sử hầu hết các khuôn khổ học sâu lựa chọn giữa một cách tiếp cận bắt buộc hoặc mang tính biểu tượng. Ví dụ, Theano, TensorFlow (lấy cảm hứng từ trước đây), Keras và CNTK xây dựng các mô hình tượng trưng. Ngược lại, Chainer và PyTorch có một cách tiếp cận bắt buộc. Một chế độ bắt buộc đã được thêm vào TensorFlow 2.0 và Keras trong các phiên bản sau này.

Như đã đề cập ở trên, PyTorch dựa trên lập trình bắt buộc và sử dụng đồ thị tính toán động. Trong nỗ lực tận dụng tính di động và hiệu quả của lập trình tượng trưng, các nhà phát triển đã xem xét liệu có thể kết hợp lợi ích của cả hai mô hình lập trình hay không. Điều này dẫn đến một torchscript cho phép người dùng phát triển và gỡ lỗi bằng cách sử dụng lập trình bắt buộc thuần túy, trong khi có khả năng chuyển đổi hầu hết các chương trình thành các chương trình tượng trưng để chạy khi hiệu suất và triển khai tính toán cấp sản phẩm được yêu cầu.

13.1.3 Lai tạo lớp `Sequential`

Cách dễ nhất để có được cảm nhận về cách thức hoạt động của lai là xem xét các mạng sâu với nhiều lớp. Thông thường, trình thông dịch Python sẽ cần thực thi mã cho tất cả các lớp để tạo ra một lệnh sau đó có thể được chuyển tiếp đến CPU hoặc GPU. Đối với một thiết bị điện toán (nhanh) duy nhất, điều này không gây ra bất kỳ vấn đề lớn nào. Mặt khác, nếu chúng ta sử dụng một máy chủ 8 GPU tiên tiến như một phiên bản AWS P3dn.24xlarge Python sẽ phải vật lộn để giữ cho tất cả các GPU bận rộn. Trình thông dịch Python đơn luồng trở thành nút cổ chai ở đây. Hãy để chúng tôi xem cách chúng tôi có thể giải quyết vấn đề này cho các phần quan trọng của mã bằng cách thay thế `Sequential` bằng `HybridSequential`. Chúng tôi bắt đầu bằng cách xác định một MLP đơn giản.

```
import torch
from torch import nn
from d2l import torch as d2l


# Factory for networks
def get_net():
    net = nn.Sequential(nn.Linear(512, 256),
                        nn.ReLU(),
                        nn.Linear(256, 128),
                        nn.ReLU(),
                        nn.Linear(128, 2))
    return net

x = torch.randn(size=(1, 512))
net = get_net()
net(x)
```

```
tensor([[ 0.0854, -0.0074]], grad_fn=<AddmmBackward0>)
```

Bằng cách chuyển đổi mô hình bằng hàm `torch.jit.script`, chúng ta có thể biên dịch và tối ưu hóa tính toán trong MLP. Kết quả tính toán của mô hình vẫn không thay đổi.

```
net = torch.jit.script(net)
net(x)
```

```
tensor([[ 0.0854, -0.0074]], grad_fn=<AddmmBackward0>)
```

Điều này có vẻ gần như quá tốt là đúng: viết cùng một mã như trước và chỉ cần chuyển đổi mô hình bằng `torch.jit.script`. Khi điều này xảy ra, mạng được tối ưu hóa (chúng tôi sẽ chuẩn hóa hiệu suất bên dưới).

Tăng tốc bằng cách lai

Để chứng minh sự cải thiện hiệu suất đạt được bằng cách biên dịch, chúng tôi so sánh thời gian cần thiết để đánh giá `net(x)` trước và sau khi lai. Hãy để chúng tôi xác định một lớp để đo lần này đầu tiên. Nó sẽ có ích trong suốt chương khi chúng tôi đặt ra để đo lường (và cải thiện) hiệu suất.

```
#@save
class Benchmark:
    """For measuring running time."""
    def __init__(self, description='Done'):
        self.description = description

    def __enter__(self):
        self.timer = d2l.Timer()
        return self

    def __exit__(self, *args):
        print(f'{self.description}: {self.timer.stop():.4f} sec')
```

Bây giờ chúng ta có thể gọi mạng hai lần, một lần với và một lần không có torchscript.

```
net = get_net()
with Benchmark('Without torchscript'):
    for i in range(1000): net(x)

net = torch.jit.script(net)
with Benchmark('With torchscript'):
    for i in range(1000): net(x)
```

```
Without torchscript: 2.1717 sec
With torchscript: 2.1708 sec
```

Như đã quan sát thấy trong các kết quả trên, sau khi một phiên bản `nn.Sequential` được viết kịch bản bằng hàm `torch.jit.script`, hiệu suất tính toán được cải thiện thông qua việc sử dụng lập trình tượng trưng.

Lập số sê-ri

Một trong những lợi ích của việc biên dịch các mô hình là chúng ta có thể serialize (lưu) mô hình và các tham số của nó vào đĩa. Điều này cho phép chúng tôi lưu trữ một mô hình theo cách độc lập với ngôn ngữ front-end của sự lựa chọn. Điều này cho phép chúng tôi triển khai các mô hình được đào tạo cho các thiết bị khác và dễ dàng sử dụng các ngôn ngữ lập trình front-end khác. Đồng thời mã thường nhanh hơn những gì có thể đạt được trong lập trình bắt buộc. Chúng ta hãy xem chức năng `save` đang hoạt động.

```
net.save('my_mlp')
!ls -lh my_mlp*
```

```
-rw-rw-r-- 1 d2l-worker d2l-worker 651K Apr 26 12:30 my_mlp
```

13.1.4 Tóm tắt

- Lập trình bắt buộc giúp bạn dễ dàng thiết kế các mô hình mới vì có thể viết mã với luồng điều khiển và khả năng sử dụng một lượng lớn hệ sinh thái phần mềm Python.
- Lập trình tượng trưng yêu cầu chúng ta chỉ định chương trình và biên dịch nó trước khi thực hiện nó. Lợi ích là cải thiện hiệu suất.

13.1.5 Bài tập

1. Xem lại các mô hình mà bạn quan tâm trong các chương trước. Bạn có thể cải thiện hiệu suất tính toán của họ bằng cách triển khai lại chúng không?

Discussions[144]

13.2 Tính toán không đồng bộ

Máy tính ngày nay là các hệ thống song song cao, bao gồm nhiều lõi CPU (thường là nhiều luồng trên mỗi lõi), nhiều yếu tố xử lý trên mỗi GPU, và thường là nhiều GPU trên mỗi thiết bị. Nói tóm lại, chúng ta có thể xử lý nhiều thứ khác nhau cùng một lúc, thường là trên các thiết bị khác nhau. Thật không may Python không phải là một cách tuyệt vời để viết mã song song và không đồng bộ, ít nhất là không có một số trợ giúp bổ sung. Rốt cuộc, Python là một luồng và điều này không có khả năng thay đổi trong tương lai. Các khuôn khổ học sâu như MXNet và TensorFlow áp dụng mô hình lập trình không đồng bộ* * để cải thiện hiệu suất, trong khi PyTorch sử dụng trình lập lịch riêng của Python dẫn đến sự đánh đổi hiệu suất khác. Đối với PyTorch, theo mặc định, các hoạt động GPU là không đồng bộ. Khi bạn gọi một chức năng sử dụng GPU, các hoạt động được xếp hàng vào thiết bị cụ thể, nhưng không nhất thiết phải được thực hiện cho đến sau này. Điều này cho phép chúng tôi thực hiện song song nhiều tính toán hơn, bao gồm các hoạt động trên CPU hoặc các GPU khác.

Do đó, hiểu cách thức hoạt động của lập trình không đồng bộ giúp chúng ta phát triển các chương trình hiệu quả hơn, bằng cách chủ động giảm các yêu cầu tính toán và phụ thuộc lẫn nhau. Điều này cho phép chúng tôi giảm chi phí bộ nhớ và tăng mức sử dụng bộ xử lý.

```
import os
import subprocess
import numpy
import torch
from torch import nn
from d2l import torch as d2l
```

13.2.1 Asynchrony qua Backend

Đối với một khởi động hãy xem xét vấn đề đồ chơi sau: chúng tôi muốn tạo ra một ma trận ngẫu nhiên và nhân nó. Chúng ta hãy làm điều đó cả trong NumPy và trong tensor PyTorch để thấy sự khác biệt. Lưu ý rằng PyTorch `tensor` được xác định trên GPU.

[144] https://discuss.d2l.ai/t/2490

```
# Warmup for GPU computation
device = d2l.try_gpu()
a = torch.randn(size=(1000, 1000), device=device)
b = torch.mm(a, a)

with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('torch'):
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
```

```
numpy: 1.0858 sec
torch: 0.0013 sec
```

Đầu ra điểm chuẩn thông qua PyTorch là đơn đặt hàng cường độ nhanh hơn. Sản phẩm chấm NumPy được thực thi trên bộ xử lý CPU trong khi phép nhân ma trận PyTorch được thực thi trên GPU và do đó sau này dự kiến sẽ nhanh hơn nhiều. Nhưng sự khác biệt thời gian rất lớn cho thấy một cái gì đó khác phải xảy ra. Theo mặc định, các hoạt động GPU không đồng bộ trong PyTorch. Buộc PyTorch hoàn thành tất cả các tính toán trước khi trả về cho thấy những gì đã xảy ra trước đây: tính toán đang được thực hiện bởi backend trong khi frontend trả về kiểm soát Python.

```
with d2l.Benchmark():
    for _ in range(10):
        a = torch.randn(size=(1000, 1000), device=device)
        b = torch.mm(a, a)
    torch.cuda.synchronize(device)
```

```
Done: 0.0029 sec
```

Nói rộng ra, PyTorch có lối vào để tương tác trực tiếp với người dùng, ví dụ, thông qua Python, cũng như một phụ trợ được hệ thống sử dụng để thực hiện tính toán. Như thể hiện trong Fig. 13.2.1, người dùng có thể viết các chương trình PyTorch bằng nhiều ngôn ngữ lối vào khác nhau, chẳng hạn như Python và C ++. Bất kể ngôn ngữ lập trình lối vào được sử dụng, việc thực hiện các chương trình PyTorch xảy ra chủ yếu trong phần phụ trợ của các triển khai C ++. Các hoạt động do ngôn ngữ frontend phát hành được chuyển sang phần phụ trợ để thực hiện. Phụ trợ quản lý các luồng riêng của nó liên tục thu thập và thực hiện các nhiệm vụ xếp hàng đợi. Lưu ý rằng để làm việc này, phụ trợ phải có khả năng theo dõi các phụ thuộc giữa các bước khác nhau trong biểu đồ tính toán. Do đó, không thể song song hóa các hoạt động phụ thuộc vào nhau.

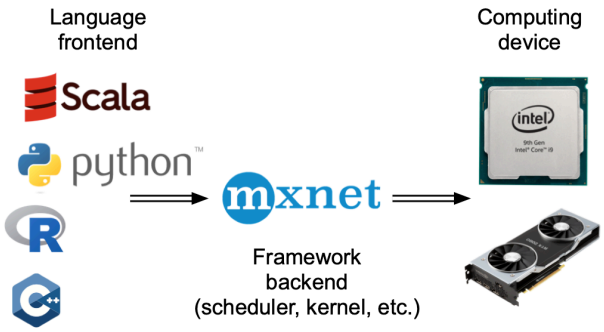


Fig. 13.2.1: Programming language frontends and deep learning framework backends.

Chúng ta hãy nhìn vào một ví dụ đồ chơi khác để hiểu biểu đồ phụ thuộc tốt hơn một chút.

```
x = torch.ones((1, 2), device=device)
y = torch.ones((1, 2), device=device)
z = x * y + 2
z
```

```
tensor([[3., 3.]], device='cuda:0')
```

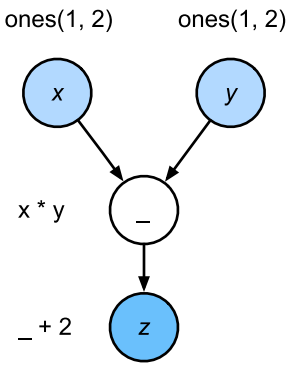


Fig. 13.2.2: The backend tracks dependencies between various steps in the computational graph.

Đoạn mã ở trên cũng được minh họa trong Fig. 13.2.2. Bất cứ khi nào Python frontend thread thực hiện một trong ba câu lệnh đầu tiên, nó chỉ đơn giản trả về nhiệm vụ cho hàng đợi phụ trợ. Khi kết quả của câu lệnh cuối cùng cần phải được * printed*, luồng giao diện người dùng Python sẽ chờ thread phụ trợ C++ kết thúc tính toán kết quả của biến `z`. Một lợi ích của thiết kế này là luồng giao diện người dùng Python không cần phải thực hiện các tính toán thực tế. Do đó, có rất ít tác động đến hiệu suất tổng thể của chương trình, bất kể hiệu suất của Python. Fig. 13.2.3 minh họa cách giao diện người dùng và phụ trợ tương tác.

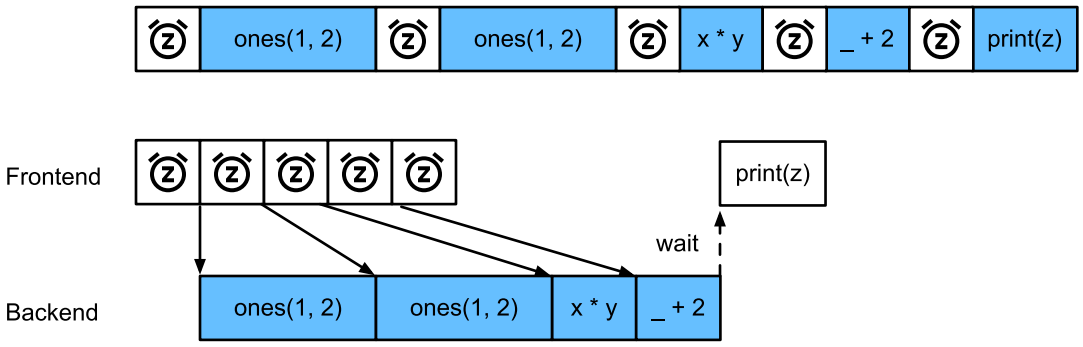


Fig. 13.2.3: Interactions of the frontend and backend.

13.2.2 Rào cản và chặn

13.2.3 Cải thiện tính toán

13.2.4 Tóm tắt

- Các khuôn khổ học sâu có thể tách các lối vào Python từ một phụ trợ thực thi. Điều này cho phép chèn nhanh không đồng bộ các lệnh vào phụ trợ và song song liên quan.
- Asynchrony dẫn đến một lối vào khá đáp ứng. Tuy nhiên, hãy thận trọng không để lấp đầy hàng đợi tác vụ vì nó có thể dẫn đến tiêu thụ bộ nhớ quá mức. Bạn nên đồng bộ hóa cho mỗi minibatch để giữ lối vào và phụ trợ xấp xỉ đồng bộ hóa.
- Các nhà cung cấp chip cung cấp các công cụ phân tích hiệu suất tinh vi để có được cái nhìn sâu sắc hơn nhiều về hiệu quả của việc học sâu.

13.2.5 Bài tập

1. Trên CPU, điểm chuẩn các hoạt động nhân ma trận tương tự trong phần này. Bạn vẫn có thể quan sát sự không đồng bộ thông qua backend?

Discussions[145]

13.3 Song song tự động

Các khung học sâu (ví dụ: MXNet và PyTorch) tự động xây dựng các đồ thị tính toán ở phụ trợ. Sử dụng đồ thị tính toán, hệ thống nhận thức được tất cả các phụ thuộc, và có thể thực hiện một cách chọn lọc nhiều tác vụ không phụ thuộc lẫn nhau song song để cải thiện tốc độ. Ví dụ, Fig. 13.2.2 trong Section 13.2 khởi tạo hai biến một cách độc lập. Do đó, hệ thống có thể chọn để thực hiện chúng song song.

Thông thường, một toán tử duy nhất sẽ sử dụng tất cả các tài nguyên tính toán trên tất cả các CPU hoặc trên một GPU duy nhất. Ví dụ, nhà điều hành `dot` sẽ sử dụng tất cả các lõi (và luồng) trên tất cả các CPU, ngay cả khi có nhiều bộ xử lý CPU trên một máy duy nhất. Điều tương tự cũng áp dụng cho một GPU duy nhất. Do đó song song không phải là khá hữu ích cho các máy tính đơn thiết bị. Với nhiều thiết bị mọi thứ quan trọng hơn. Mặc dù song song thường có liên quan nhất giữa nhiều GPU, việc thêm CPU cục bộ sẽ tăng hiệu suất một chút. Ví dụ: xem (Hadjis et al., 2016) tập trung vào đào tạo các mô hình thị giác máy tính kết hợp GPU

[145] https://discuss.d2l.ai/t/2564

và CPU. Với sự tiện lợi của một khung song song tự động, chúng ta có thể thực hiện cùng một mục tiêu trong một vài dòng mã Python. Rộng hơn, cuộc thảo luận của chúng tôi về tính toán song song tự động tập trung vào tính toán song song bằng cách sử dụng cả CPU và GPU, cũng như sự song song của tính toán và giao tiếp.

Lưu ý rằng chúng ta cần ít nhất hai GPU để chạy các thí nghiệm trong phần này.

```
import torch
from d2l import torch as d2l
```

13.3.1 Tính toán song song trên GPU

Chúng ta hãy bắt đầu bằng cách xác định khối lượng công việc tham chiếu để kiểm tra: hàm `run` dưới đây thực hiện 10 phép nhân ma trận trên thiết bị mà chúng tôi lựa chọn bằng cách sử dụng dữ liệu được phân bổ thành hai biến: `x_gpu1` và `x_gpu2`.

```
devices = d2l.try_all_gpus()
def run(x):
    return [x.mm(x) for _ in range(50)]

x_gpu1 = torch.rand(size=(4000, 4000), device=devices[0])
x_gpu2 = torch.rand(size=(4000, 4000), device=devices[1])
```

Bây giờ chúng ta áp dụng chức năng cho dữ liệu. Để đảm bảo rằng bộ nhớ đệm không đóng vai trò trong kết quả, chúng tôi làm nóng các thiết bị bằng cách thực hiện một lần vượt qua một trong hai trong số chúng trước khi đo. `torch.cuda.synchronize()` chờ tất cả các hạt nhân trong tất cả các luồng trên thiết bị CIDA hoàn thành. Phải mất một đối số `device`, thiết bị mà chúng ta cần đồng bộ hóa. Nó sử dụng thiết bị hiện tại, được đưa ra bởi `current_device()`, nếu đối số thiết bị là `None` (mặc định).

```
run(x_gpu1)
run(x_gpu2)  # Warm-up all devices
torch.cuda.synchronize(devices[0])
torch.cuda.synchronize(devices[1])

with d2l.Benchmark('GPU1 time'):
    run(x_gpu1)
    torch.cuda.synchronize(devices[0])

with d2l.Benchmark('GPU2 time'):
    run(x_gpu2)
    torch.cuda.synchronize(devices[1])
```

```
GPU1 time: 0.5028 sec
GPU2 time: 0.5072 sec
```

Nếu chúng ta loại bỏ câu lệnh `synchronize` giữa cả hai tác vụ, hệ thống sẽ tự do song song tính toán trên cả hai thiết bị một cách tự động.

```
with d2l.Benchmark('GPU1 & GPU2'):
    run(x_gpu1)
    run(x_gpu2)
    torch.cuda.synchronize()
```

```
GPU1 & GPU2: 0.5028 sec
```

Trong trường hợp trên, tổng thời gian thực thi nhỏ hơn tổng các bộ phận của nó, vì khung học sâu sẽ tự động lên lịch tính toán trên cả hai thiết bị GPU mà không cần mã phức tạp thay mặt cho người dùng.

13.3.2 Tính toán song song và truyền thông

Trong nhiều trường hợp, chúng ta cần di chuyển dữ liệu giữa các thiết bị khác nhau, nói giữa CPU và GPU hoặc giữa các GPU khác nhau. Ví dụ, điều này xảy ra khi chúng ta muốn thực hiện tối ưu hóa phân tán, nơi chúng ta cần tổng hợp các gradient trên nhiều thẻ gia tốc. Chúng ta hãy mô phỏng điều này bằng cách tính toán trên GPU và sau đó sao chép kết quả trở lại CPU.

```
def copy_to_cpu(x, non_blocking=False):
    return [y.to('cpu', non_blocking=non_blocking) for y in x]

with d2l.Benchmark('Run on GPU1'):
    y = run(x_gpu1)
    torch.cuda.synchronize()

with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    torch.cuda.synchronize()
```

```
Run on GPU1: 0.5027 sec
Copy to CPU: 2.4896 sec
```

Điều này có phần không hiệu quả. Lưu ý rằng chúng ta đã có thể bắt đầu sao chép các phần của `y` vào CPU trong khi phần còn lại của danh sách vẫn đang được tính toán. Tình huống này xảy ra, ví dụ, khi chúng ta tính toán gradient (backprop) trên một minibatch. Độ dốc của một số tham số sẽ có sẵn sớm hơn so với các tham số khác. Do đó, nó hoạt động để lợi thế của chúng tôi để bắt đầu sử dụng băng thông bus PCI-Express trong khi GPU vẫn đang chạy. Trong PyTorch, một số chức năng như `to()` và `copy_()` thừa nhận một đối số `non_blocking` rõ ràng, cho phép đồng bộ hóa người gọi bỏ qua khi không cần thiết. Đặt `non_blocking=True` cho phép chúng tôi mô phỏng kịch bản này.

```
with d2l.Benchmark('Run on GPU1 and copy to CPU'):
    y = run(x_gpu1)
    y_cpu = copy_to_cpu(y, True)
    torch.cuda.synchronize()
```

```
Run on GPU1 and copy to CPU: 1.9150 sec
```

Tổng thời gian cần thiết cho cả hai hoạt động là (như mong đợi) ít hơn tổng các bộ phận của chúng. Lưu ý rằng tác vụ này khác với tính toán song song vì nó sử dụng một tài nguyên khác: bus giữa CPU và GPU. Trên thực tế, chúng ta có thể tính toán trên cả hai thiết bị và giao tiếp, tất cả cùng một lúc. Như đã nói ở trên, có sự phụ thuộc giữa tính toán và giao tiếp: `y[i]` phải được tính toán trước khi nó có thể được sao chép vào CPU. May mắn thay, hệ thống có thể sao chép `y[i-1]` trong khi tính toán `y[i]` để giảm tổng thời gian chạy.

Chúng tôi kết luận với một minh họa của biểu đồ tính toán và phụ thuộc của nó cho một MLP hai lớp đơn giản khi đào tạo trên CPU và hai GPU, như được mô tả trong Fig. 13.3.1. Sẽ khá đau đớn khi lên lịch chương trình song song kết quả từ việc này bằng tay. Đây là nơi thuận lợi để có một phụ trợ điện toán dựa trên đồ thị để tối ưu hóa.

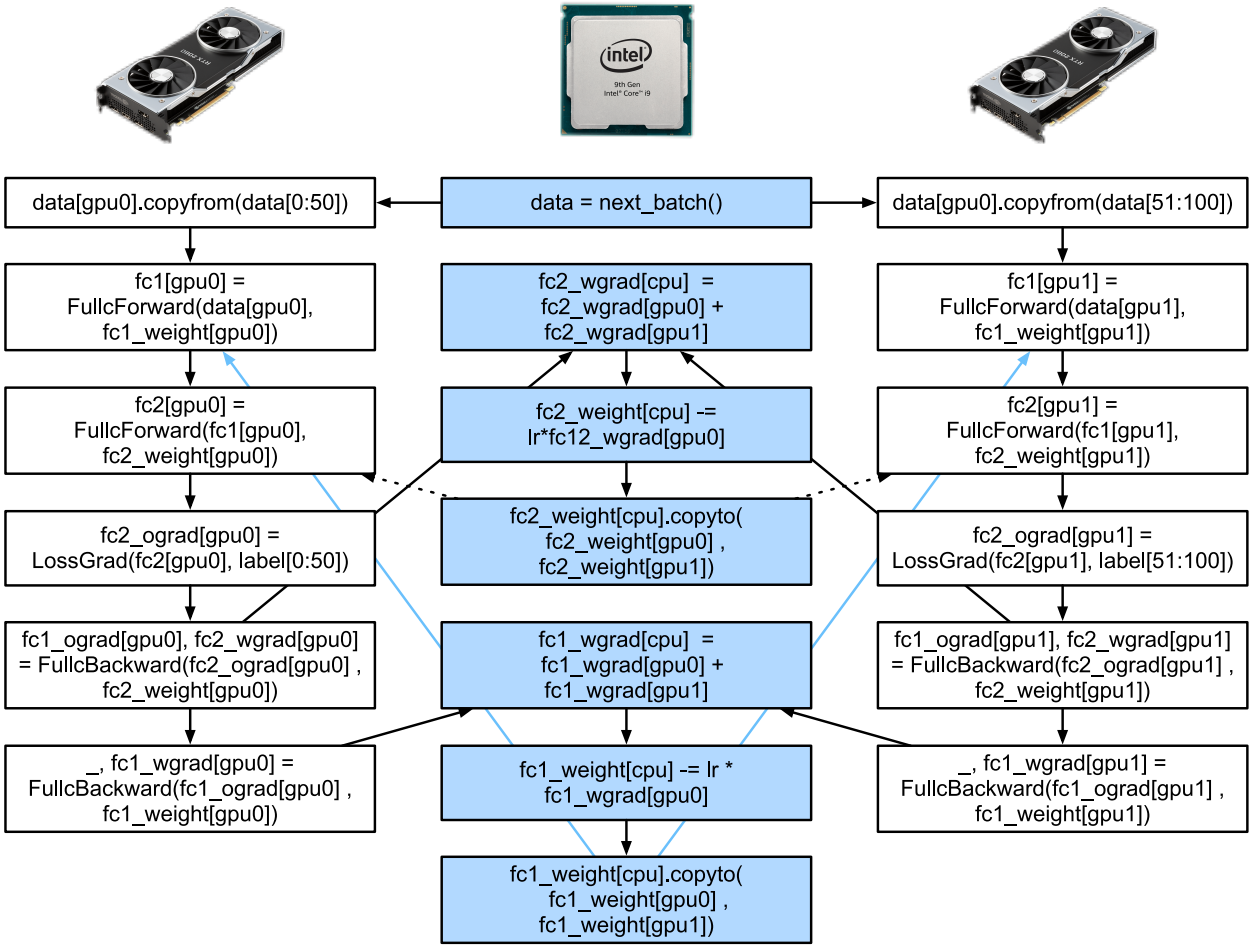


Fig. 13.3.1: The computational graph and its dependencies of a two-layer MLP on a CPU and two GPUs.

13.3.3 Tóm tắt

- Các hệ thống hiện đại có nhiều thiết bị khác nhau, chẳng hạn như nhiều GPU và CPU. Chúng có thể được sử dụng song song, không đồng bộ.
- Các hệ thống hiện đại cũng có nhiều tài nguyên khác nhau để liên lạc, chẳng hạn như PCI Express, lưu trữ (điển hình là ổ đĩa trạng thái rắn hoặc qua mạng), và băng thông mạng. Chúng có thể được sử dụng song song cho hiệu quả cao điểm.
- Các phụ trợ có thể cải thiện hiệu suất thông qua tính toán song song tự động và giao tiếp.

13.3.4 Bài tập

1. Tám thao tác đã được thực hiện trong hàm `run` được xác định trong phần này. Không có sự phụ thuộc giữa chúng. Thiết kế một thí nghiệm để xem khung học sâu sẽ tự động thực hiện chúng song song hay không.
2. Khi khối lượng công việc của một nhà điều hành cá nhân đủ nhỏ, song song có thể giúp đỡ ngay cả trên một CPU hoặc GPU duy nhất. Thiết kế một thí nghiệm để xác minh điều này.
3. Thiết kế một thí nghiệm sử dụng tính toán song song trên CPU, GPU và giao tiếp giữa cả hai thiết bị.

4. Sử dụng trình gỡ lỗi như Nsight[146] của NVIDIA để xác minh rằng mã của bạn có hiệu quả.
5. Thiết kế các tác vụ tính toán bao gồm các phụ thuộc dữ liệu phức tạp hơn và chạy thử nghiệm để xem liệu bạn có thể có được kết quả chính xác trong khi cải thiện hiệu suất hay không.

Discussions[147]

13.4 Phần cứng

Xây dựng hệ thống với hiệu suất tuyệt vời đòi hỏi một sự hiểu biết tốt về các thuật toán và mô hình để nắm bắt các khía cạnh thống kê của vấn đề. Đồng thời nó cũng không thể thiếu để có ít nhất một modicum kiến thức về phần cứng cơ bản. Phần hiện tại không thay thế cho một khóa học thích hợp về thiết kế phần cứng và hệ thống. Thay vào đó, nó có thể đóng vai trò là điểm khởi đầu để hiểu lý do tại sao một số thuật toán hiệu quả hơn các thuật toán khác và làm thế nào để đạt được thông lượng tốt. Một thiết kế tốt có thể dễ dàng tạo ra sự khác biệt về thứ tự cường độ và đến lượt nó, điều này có thể tạo ra sự khác biệt giữa việc có thể đào tạo mạng (ví dụ: trong một tuần) và hoàn toàn không (trong 3 tháng, do đó thiếu thời hạn). Chúng tôi sẽ bắt đầu bằng cách nhìn vào máy tính. Sau đó, chúng tôi sẽ phóng to để xem xét kỹ hơn về CPU và GPU. Cuối cùng, chúng tôi thu nhỏ để xem xét cách nhiều máy tính được kết nối trong một trung tâm máy chủ hoặc trên đám mây.

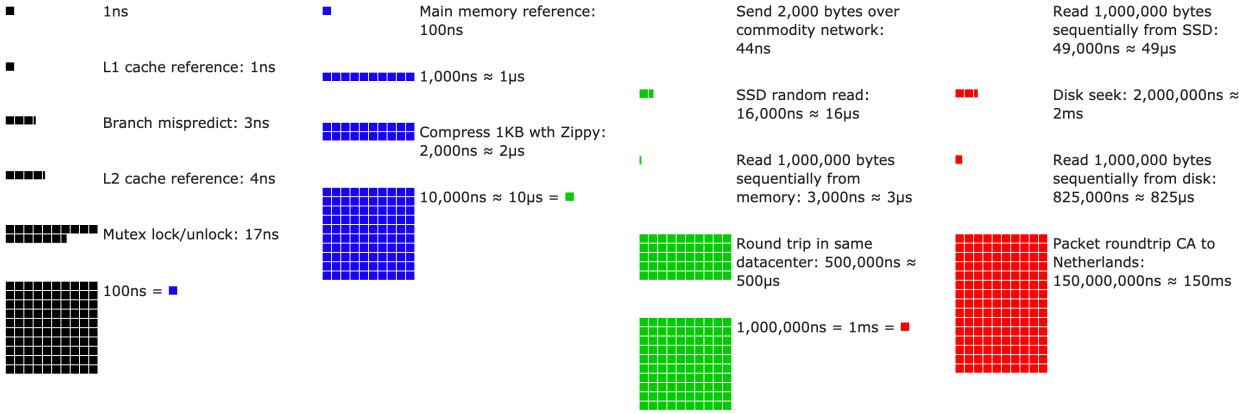


Fig. 13.4.1: Latency Numbers that every programmer should know.

Độc giả thiếu kiên nhẫn có thể có được bằng cách với Fig. 13.4.1. Nó được lấy từ [bài viết tương tác] của Colin Scott (https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html) cung cấp một cái nhìn tổng quan tốt về sự tiến bộ trong thập kỷ qua. Những con số ban đầu là do Stanford talk from 2010[148] của Jeff Dean. Cuộc thảo luận dưới đây giải thích một số lý do cho những con số này và cách họ có thể hướng dẫn chúng ta trong việc thiết kế các thuật toán. Các cuộc thảo luận dưới đây là mức độ rất cao và cursory. Nó rõ ràng là * không thay thế* cho một khóa học thích hợp mà chỉ có nghĩa là cung cấp đủ thông tin cho một người mô hình thống kê để đưa ra quyết định thiết kế phù hợp. Để biết tổng quan chuyên sâu về kiến trúc máy tính, chúng tôi giới thiệu người đọc đến (Hennessy & Patterson, 2011) hoặc một khóa học gần đây về chủ đề này, chẳng hạn như khóa học của Arste Asanovic[149].

[146] https://developer.nvidia.com/nsight-compute-2019_5
[147] https://discuss.d2l.ai/t/1681
[148] https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf
[149] http://inst.eecs.berkeley.edu/~cs152/sp19/

13.4.1 Máy vi tính

Hầu hết các nhà nghiên cứu và học viên học sâu có quyền truy cập vào một máy tính với một lượng bộ nhớ hợp lý, tính toán, một số dạng của một máy gia tốc như GPU, hoặc bội số của chúng. Một máy tính bao gồm các thành phần chính sau:

- Bộ xử lý (còn được gọi là CPU) có thể thực thi các chương trình chúng tôi cung cấp cho nó (ngoài việc chạy hệ điều hành và nhiều thứ khác), thường bao gồm 8 lõi trở lên.
- Bộ nhớ (RAM) để lưu trữ và truy xuất các kết quả từ tính toán, chẳng hạn như vectơ trọng lượng và kích hoạt, và dữ liệu đào tạo.
- Một kết nối mạng Ethernet (đôi khi nhiều) với tốc độ từ 1 Gb/s đến 100 Gb/s.
- Một bus mở rộng tốc độ cao (PCIe) để kết nối hệ thống với một hoặc nhiều GPU. Máy chủ có tới 8 máy gia tốc, thường được kết nối trong một cấu trúc liên kết tiên tiến, trong khi các hệ thống máy tính để bàn có 1 hoặc 2, tùy thuộc vào ngân sách của người dùng và kích thước của nguồn điện.
- Lưu trữ bền, chẳng hạn như ổ đĩa cứng từ tính, ổ đĩa trạng thái rắn, trong nhiều trường hợp được kết nối bằng bus PCIe. Nó cung cấp chuyển dữ liệu đào tạo hiệu quả đến hệ thống và lưu trữ các trạm kiểm soát trung gian khi cần thiết.

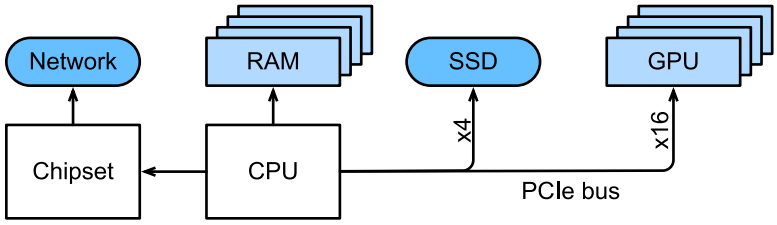


Fig. 13.4.2: Connectivity of components of a computer.

Như Fig. 13.4.2 chỉ ra, hầu hết các thành phần (mạng, GPU và lưu trữ) được kết nối với CPU trên bus PCIe. Nó bao gồm nhiều làn xe được gắn trực tiếp vào CPU. Ví dụ Threadripper 3 của AMD có 64 làn PCIe 4.0, mỗi làn có khả năng truyền dữ liệu 16 Gbit/s theo cả hai hướng. Bộ nhớ được gắn trực tiếp vào CPU với tổng băng thông lên tới 100 Gb/s.

Khi chúng ta chạy mã trên máy tính, chúng ta cần xáo trộn dữ liệu vào bộ xử lý (CPU hoặc GPU), thực hiện tính toán, sau đó di chuyển kết quả ra khỏi bộ xử lý trở lại RAM và bộ nhớ bền. Do đó, để có được hiệu suất tốt, chúng ta cần đảm bảo rằng điều này hoạt động liền mạch mà không có bất kỳ hệ thống nào trở thành nút cổ chai lớn. Ví dụ, nếu chúng ta không thể tải hình ảnh đủ nhanh, bộ xử lý sẽ không có bất kỳ công việc nào để làm. Tương tự như vậy, nếu chúng ta không thể di chuyển ma trận đủ nhanh đến CPU (hoặc GPU), các yếu tố xử lý của nó sẽ chết đói. Cuối cùng, nếu chúng ta muốn đồng bộ hóa nhiều máy tính trên mạng, sau này không nên làm chậm tính toán. Một lựa chọn là giao tiếp xen kẽ và tính toán. Hãy để chúng tôi có một cái nhìn tại các thành phần khác nhau chi tiết hơn.

13.4.2 Bộ nhớ

Tại bộ nhớ cơ bản nhất của nó được sử dụng để lưu trữ dữ liệu cần phải dễ dàng truy cập. Hiện tại CPU RAM thường thuộc loại DDR4[150], cung cấp băng thông 20—25 Gb/s cho mỗi mô-đun. Mỗi mô-đun có một bus rộng 64 bit. Thông thường các cặp mô-đun bộ nhớ được sử dụng để cho phép nhiều kênh. CPU có từ 2 đến 4 kênh bộ nhớ, tức là chúng có từ 4 0GB/s đến băng thông bộ nhớ đỉnh 100 Gb/s. Thường có hai ngân hàng trên mỗi kênh. Ví dụ Zen 3 Threadripper của AMD có 8 khe cắm.

Mặc dù những con số này rất ấn tượng, nhưng thực sự, họ chỉ kể một phần của câu chuyện. Khi chúng ta muốn đọc một phần từ bộ nhớ trước tiên chúng ta cần nói với mô-đun bộ nhớ nơi thông tin có thể được tìm thấy. Đó là, trước tiên chúng ta cần gửi *địa chỉ* đến RAM. Khi điều này được thực hiện, chúng ta có thể chọn chỉ đọc một bản ghi 64 bit duy nhất hoặc một chuỗi dài các bản ghi. Cái sau được gọi là *burst đọc*. Tóm lại, việc gửi một địa chỉ vào bộ nhớ và thiết lập việc chuyển mất khoảng 100 ns (chi tiết phụ thuộc vào hệ số thời gian cụ thể của chip bộ nhớ được sử dụng), mỗi lần chuyển tiếp theo chỉ mất 0,2 ns. Nói tóm lại, lần đọc đầu tiên đắt gấp 500 lần so với những lần tiếp theo! Lưu ý rằng chúng ta có thể thực hiện lên đến 10.000.000 lần đọc ngẫu nhiên mỗi giây. Điều này cho thấy rằng chúng ta tránh truy cập bộ nhớ ngẫu nhiên càng nhiều càng tốt và sử dụng burst reads (và ghi) thay thế.

Các vấn đề phức tạp hơn một chút khi chúng tôi tính đến việc chúng tôi có nhiều *ngân hàng*. Mỗi ngân hàng có thể đọc bộ nhớ phần lớn độc lập. Điều này có nghĩa là hai điều. Một mặt, số lần đọc ngẫu nhiên hiệu quả cao hơn tới 4 lần, miễn là chúng được trải đều trên bộ nhớ. Điều đó cũng có nghĩa là vẫn là một ý tưởng tồi để thực hiện các lần đọc ngẫu nhiên vì lần đọc burst cũng nhanh hơn 4 lần. Mặt khác, do sự liên kết bộ nhớ với ranh giới 64 bit, nên căn chỉnh bất kỳ cấu trúc dữ liệu nào có cùng ranh giới. Trình biên dịch làm điều này khá nhiều automatically[151] khi các cờ thích hợp được đặt. Độc giả tò mò được khuyến khích xem lại một bài giảng về DRams chẳng hạn như bài của Zeshan Chishti[152].

Bộ nhớ GPU phải tuân theo yêu cầu băng thông thậm chí cao hơn vì chúng có nhiều yếu tố xử lý hơn CPU. Bởi và lớn có hai lựa chọn để giải quyết chúng. Đầu tiên là làm cho bus bộ nhớ rộng hơn đáng kể. Ví dụ, RTX 2080 Ti của NVIDIA có một bus rộng 352-bit. Điều này cho phép chuyển nhiều thông tin hơn cùng một lúc. Thứ hai, GPU sử dụng bộ nhớ hiệu suất cao cụ thể. Các thiết bị cấp tiêu dùng, chẳng hạn như dòng RTX và Titan của NVIDIA thường sử dụng chip GDDR6[153] với băng thông tổng hợp hơn 500 Gb/s. Một cách khác là sử dụng các mô-đun HBM (bộ nhớ băng thông cao). Họ sử dụng một giao diện rất khác nhau và kết nối trực tiếp với GPU trên một wafer silicon chuyên dụng. Điều này làm cho chúng rất tốn kém và việc sử dụng chúng thường bị giới hạn ở các chip máy chủ cao cấp, chẳng hạn như dòng máy gia tốc NVIDIA Volta V100. Khá không có gì ngạc nhiên, bộ nhớ GPU nói chung là * nhiều* nhỏ hơn bộ nhớ CPU do chi phí cao hơn của trước đây. Đối với mục đích của chúng tôi, bởi và lớn đặc điểm hiệu suất của họ là tương tự nhau, chỉ nhanh hơn rất nhiều. Chúng ta có thể bỏ qua một cách an toàn các chi tiết cho mục đích của cuốn sách này. Chúng chỉ quan trọng khi điều chỉnh hạt GPU cho thông lượng cao.

[150] https://en.wikipedia.org/wiki/DDR4_SDRAM
[151] https://en.wikipedia.org/wiki/Data_structure_alignment
[152] http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf
[153] https://en.wikipedia.org/wiki/GDDR6_SDRAM

13.4.3 Lưu trữ

Chúng tôi thấy rằng một số đặc điểm chính của RAM là * băng thông * và * độ trễ *. Điều tương tự cũng đúng đối với các thiết bị lưu trữ, chỉ là sự khác biệt có thể còn cực đoan hơn.

Ổ đĩa cứng

Ổ đĩa cứng (HDD) đã được sử dụng trong hơn nửa thế kỷ. Tóm lại, chúng chứa một số đĩa quay có đầu có thể được định vị để đọc hoặc viết tại bất kỳ bản nhạc nào. Đĩa cao cấp chứa tới 16 TB trên 9 đĩa. Một trong những lợi ích chính của ổ cứng là chúng tương đối rẻ tiền. Một trong nhiều nhược điểm của họ là các chế độ thất bại thảm khốc điển hình của chúng và độ trễ đọc tương đối cao.

Để hiểu sau này, hãy xem xét thực tế là ổ cứng quay ở khoảng 7.200 RPM (vòng quay mỗi phút). Nếu chúng nhanh hơn nhiều, chúng sẽ vỡ do lực ly tâm tác dụng lên đĩa. Điều này có một nhược điểm lớn khi truy cập vào một khu vực cụ thể trên đĩa: chúng ta cần đợi cho đến khi đĩa đã xoay ở vị trí (chúng ta có thể di chuyển các đầu nhưng không tăng tốc các đĩa thực tế). Do đó, nó có thể mất hơn 8 ms cho đến khi dữ liệu được yêu cầu có sẵn. Một cách phổ biến mà điều này được thể hiện là nói rằng ổ cứng có thể hoạt động ở khoảng 100 IOP (hoạt động đầu vào/đầu ra mỗi giây). Con số này về cơ bản vẫn không thay đổi trong hai thập kỷ qua. Tệ hơn nữa, việc tăng băng thông cũng khó khăn như nhau (theo thứ tự 100—200 MB/s). Rốt cuộc, mỗi đầu đọc một bản nhạc của bit, do đó tốc độ bit chỉ quy mô với căn bậc hai của mật độ thông tin. Do đó, ổ cứng nhanh chóng trở nên xuống hạng lưu trữ lưu trữ và lưu trữ cấp thấp cho các bộ dữ liệu rất lớn.

Ổ đĩa trạng thái rắn

Ổ đĩa trạng thái rắn (SSD) sử dụng bộ nhớ flash để lưu trữ thông tin liên tục. Điều này cho phép truy cập * nhanh hơn nhiều* vào các bản ghi được lưu trữ. SSD hiện đại có thể hoạt động ở mức 100.000 đến 500.000 IOP, tức là nhanh hơn 3 đơn hàng độ lớn hơn ổ cứng. Hơn nữa, băng thông của chúng có thể đạt 1-3GB/s, tức là, một bậc độ lớn nhanh hơn ổ cứng. Những cải tiến này nghe gần như quá tốt để trở thành sự thật. Thật vậy, chúng đi kèm với các cảnh báo sau đây, do cách SSD được thiết kế.

- SSD lưu trữ thông tin trong các khối (256 KB hoặc lớn hơn). Chúng chỉ có thể được viết như một toàn thể, mất thời gian đáng kể. Do đó, viết ngẫu nhiên bit-wise trên SSD có hiệu suất rất kém. Tương tự như vậy, việc viết dữ liệu nói chung mất thời gian đáng kể vì khối phải được đọc, xóa và sau đó viết lại bằng thông tin mới. Đến nay bộ điều khiển SSD và firmware đã phát triển các thuật toán để giảm thiểu điều này. Tuy nhiên, ghi có thể chậm hơn nhiều, đặc biệt đối với SSD QLC (tế bào cấp bốn). Chìa khóa để cải thiện hiệu suất là duy trì một * queue* hoạt động, thích đọc và viết trong các khối lớn nếu có thể.
- Các tế bào bộ nhớ trong SSD hao mòn tương đối nhanh (thường là sau vài nghìn lần viết). Thuật toán bảo vệ cấp mài mòn có thể lây lan sự xuống cấp trên nhiều tế bào. Điều đó nói rằng, không nên sử dụng SSD để hoán đổi tệp hoặc cho các tập hợp lớn các tệp nhật ký.
- Cuối cùng, sự gia tăng lớn về băng thông đã buộc các nhà thiết kế máy tính phải gắn SSD trực tiếp vào bus PCIe. Các ổ đĩa có khả năng xử lý điều này, được gọi là NVMe (Bộ nhớ không bay hơi tăng cường), có thể sử dụng tối đa 4 làn PCIe. Số này lên tới 8GB/s trên PCIe 4.0.

Lưu trữ đám mây

Lưu trữ đám mây cung cấp một phạm vi hiệu suất có thể cấu hình. Đó là, việc phân công lưu trữ cho các máy ảo là năng động, cả về số lượng và về tốc độ, theo lựa chọn của người dùng. Chúng tôi khuyên người dùng nên tăng số lượng IOP được cung cấp bất cứ khi nào độ trễ quá cao, ví dụ, trong quá trình đào tạo với nhiều hồ sơ nhỏ.

13.4.4 CPU

Các đơn vị xử lý trung tâm (CPU) là trung tâm của bất kỳ máy tính nào. Chúng bao gồm một số thành phần chính: * lõi bộ xử lý* có thể thực thi mã máy, * bus* kết nối chúng (cấu trúc liên kết cụ thể khác biệt đáng kể giữa các mô hình bộ xử lý, thế hệ và nhà cung cấp) và * caches* để cho phép băng thông cao hơn và truy cập bộ nhớ trễ thấp hơn so với có thể bằng cách đọc từ bộ nhớ chính. Cuối cùng, hầu hết tất cả các CPU hiện đại đều chứa các đơn vị xử lý vectơ * để hỗ trợ đại số tuyến tính hiệu suất cao và các phức tạp, vì chúng phổ biến trong xử lý phương tiện truyền thông và học máy.

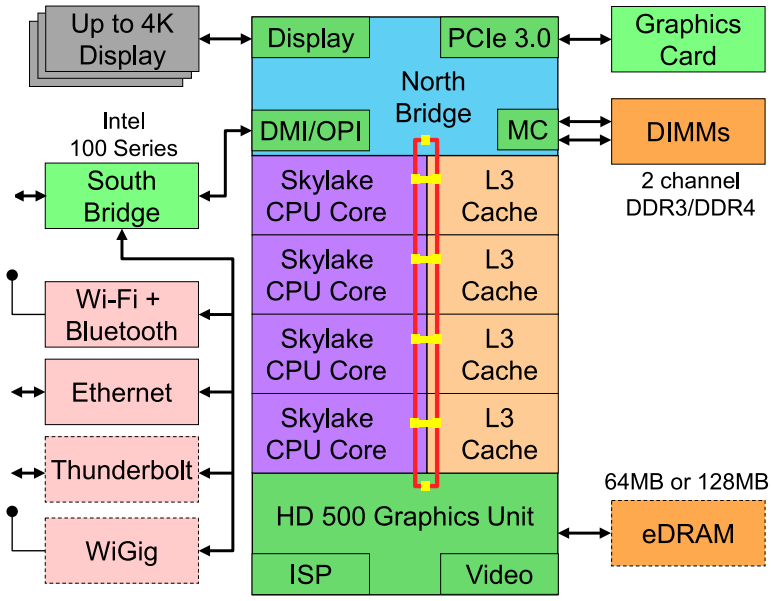


Fig. 13.4.3: Intel Skylake consumer quad-core CPU.

Fig. 13.4.3 mô tả một CPU lõi tứ cấp tiêu dùng Intel Skylake. Nó có một GPU tích hợp, bộ nhớ cache và một ringbus kết nối bốn lõi. Các thiết bị ngoại vi, chẳng hạn như Ethernet, WiFi, Bluetooth, bộ điều khiển SSD và USB, là một phần của chipset hoặc được gắn trực tiếp (PCIe) vào CPU.

Kiến trúc vi mô

Mỗi lõi bộ xử lý bao gồm một bộ thành phần khá phức tạp. Trong khi các chi tiết khác nhau giữa các thế hệ và nhà cung cấp, chức năng cơ bản là khá nhiều tiêu chuẩn. Front-end tải hướng dẫn và cố gắng dự đoán đường dẫn nào sẽ được thực hiện (ví dụ, cho luồng điều khiển). Hướng dẫn sau đó được giải mã từ mã lắp ráp sang microinstructions. Mã hội thường không phải là mã cấp thấp nhất mà một bộ xử lý thực thi. Thay vào đó, các hướng dẫn phức tạp có thể được giải mã thành một tập hợp các hoạt động cấp thấp hơn. Chúng sau đó được xử lý bởi lõi thực hiện thực tế. Thường thì cái sau có khả năng thực hiện nhiều hoạt động cùng một lúc. Ví dụ, lõi ARM Cortex A77 của Fig. 13.4.4 có thể thực hiện tối đa 8 thao tác cùng một lúc.

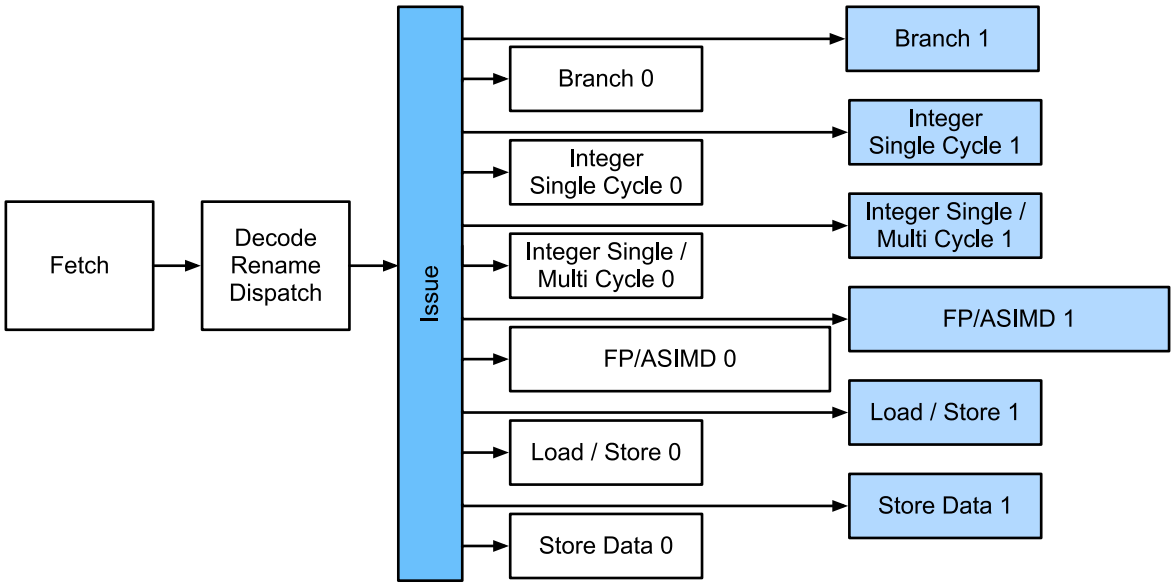


Fig. 13.4.4: ARM Cortex A77 Microarchitecture.

Điều này có nghĩa là các chương trình hiệu quả có thể thực hiện nhiều hơn một lệnh cho mỗi chu kỳ đồng hồ, với điều kiện là chúng có thể được thực hiện độc lập. Không phải tất cả các đơn vị được tạo ra bằng nhau. Một số chuyên về hướng dẫn số nguyên trong khi những người khác được tối ưu hóa cho hiệu suất điểm nổi. Để tăng thông lượng, bộ xử lý cũng có thể theo nhiều đường dẫn mã đồng thời trong một hướng dẫn phân nhánh và sau đó loại bỏ kết quả của các nhánh không được thực hiện. Đây là lý do tại sao các đơn vị dự đoán chi nhánh quan trọng (trên front-end) sao cho chỉ những con đường hứa hẹn nhất được theo đuổi.

Vector hóa

Học sâu là cực kỳ đói tính toán. Do đó, để làm cho CPU phù hợp với học máy, người ta cần thực hiện nhiều thao tác trong một chu kỳ đồng hồ. Điều này đạt được thông qua các đơn vị vector. Họ có tên khác nhau: on ARM they are called NEON, on x86 they (a recent generation) are referred to as AVX2[154] units. A common aspect is that they are able to perform SIMD (single instruction multiple data) operations. Fig. 13.4.5 cho thấy cách 8 số nguyên ngắn có thể được thêm vào trong một chu kỳ đồng hồ trên ARM.

[154] https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

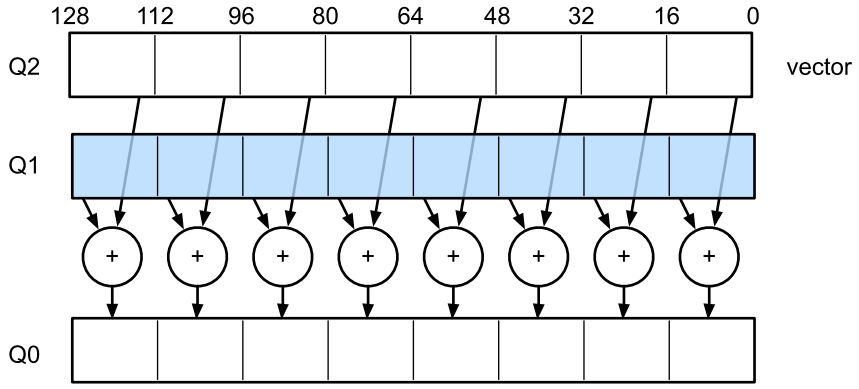


Fig. 13.4.5: 128 bit NEON vectorization.

Tùy thuộc vào lựa chọn kiến trúc, các thanh ghi như vậy dài tới 512 bit, cho phép kết hợp tối đa 64 cặp số. Ví dụ, chúng ta có thể nhân hai số và thêm chúng vào một phần ba, mà còn được gọi là một kết hợp multiply-add. OpenVino[155] của Intel sử dụng những thứ này để đạt được thông lượng đáng kính cho việc học sâu trên CPU cấp máy chủ. Tuy nhiên, lưu ý rằng con số này hoàn toàn bị lùn bởi những gì GPU có khả năng đạt được. Ví dụ, RTX 2080 Ti của NVIDIA có 4.352 lõi CIDA, mỗi lõi có khả năng xử lý một hoạt động như vậy bất cứ lúc nào.

Bộ nhớ cache

Hãy xem xét các tình huống sauđây: we have a modest CPU core with 4 cores as depicted in Fig. 13.4.3 ở trên, chạy ở tần số 2 GHz. Hơn nữa, chúng ta hãy giả định rằng chúng tôi có số lượng IPC (hướng dẫn trên mỗi đồng hồ) là 1 và các đơn vị có AVX2 với kích hoạt chiều rộng 256-bit. Chúng ta hãy hơn nữa giả định rằng ít nhất một trong những thanh ghi được sử dụng cho các hoạt động AVX2 cần phải được lấy từ bộ nhớ. Điều này có nghĩa là CPU tiêu thụ 4×256 bit $= 128$ bytes dữ liệu trên mỗi chu kỳ đồng hồ. Trừ khi chúng tôi có thể chuyển $2 \times 10^9 \times 128 = 256 \times 10^9$ byte sang bộ xử lý mỗi giây, các yếu tố xử lý sẽ chết đói. Thật không may, giao diện bộ nhớ của một con chip như vậy chỉ hỗ trợ truyền dữ liệu 20—40 Gb/s, tức là, một bậc độ lớn ít hơn. Việc khắc phục là để tránh tải dữ liệu * mới* từ bộ nhớ càng nhiều càng tốt và thay vào đó là bộ nhớ cache cục bộ trên CPU. Đây là nơi bộ nhớ cache có ích. Thông thường các tên hoặc khái niệm sau đây được sử dụng:

- **Đăng ký** được nói đúng không phải là một phần của bộ nhớ cache. Họ giúp hướng dẫn sân khấu. Điều đó nói rằng, thanh ghi CPU là vị trí bộ nhớ mà CPU có thể truy cập ở tốc độ đồng hồ mà không bị phạt chậm trễ nào. CPU có hàng chục thanh ghi. Tùy thuộc vào trình biên dịch (hoặc lập trình viên) để sử dụng thanh ghi hiệu quả. Ví dụ ngôn ngữ lập trình C có từ khóa `register`.
- ** L1 caches** là tuyến phòng thủ đầu tiên chống lại các yêu cầu băng thông bộ nhớ cao. Bộ nhớ đệm L1 rất nhỏ (kích thước điển hình có thể là 32—64 KB) và thường được chia thành các bộ nhớ cache dữ liệu và hướng dẫn. Khi dữ liệu được tìm thấy trong bộ nhớ cache L1, truy cập rất nhanh. Nếu chúng không thể được tìm thấy ở đó, tìm kiếm sẽ tiến triển xuống hệ thống phân cấp bộ nhớ cache.
- ** L2 caches** là điểm dừng tiếp theo. Tùy thuộc vào thiết kế kiến trúc và kích thước bộ xử lý chúng có thể là độc quyền. Chúng chỉ có thể truy cập được bởi một lõi nhất định hoặc được chia sẻ giữa nhiều lõi. Bộ nhớ đệm L2 lớn hơn (thường là 256—512 KB cho mỗi lõi) và chậm hơn L1. Hơn nữa, để truy cập một cái gì đó trong L2, trước tiên chúng ta cần kiểm tra để nhận ra rằng dữ liệu không có trong L1, điều này thêm một lượng nhỏ độ trễ thêm.

[155] https://01.org/openvinotoolkit

- ** L3 caches** được chia sẻ giữa nhiều lõi và có thể khá lớn. CPU máy chủ Epyc 3 của AMD có một con số khổng lồ 256 MB bộ nhớ cache trải rộng trên nhiều chiplet. Các số điển hình hơn nằm trong phạm vi 4—8 MB.

Dự đoán các yếu tố bộ nhớ nào sẽ cần thiết tiếp theo là một trong những thông số tối ưu hóa chính trong thiết kế chip. Ví dụ, nên đi qua bộ nhớ theo hướng *forward* vì hầu hết các thuật toán bộ nhớ đệm sẽ cố gắng * đọc trước* thay vì ngược lại. Tương tự như vậy, giữ các mẫu truy cập bộ nhớ cục bộ là một cách tốt để cải thiện hiệu suất.

Thêm bộ nhớ đệm là một thanh kiếm hai cạnh. Một mặt họ đảm bảo rằng các lõi bộ xử lý không chết đói dữ liệu. Đồng thời chúng tăng kích thước chip, sử dụng lên diện tích mà nếu không có thể đã được chi cho việc tăng sức mạnh xử lý. Hơn nữa, * nhớ cache * có thể tốn kém. Hãy xem xét trường hợp xấu nhất, * chia sẻ sai *, như được mô tả trong Fig. 13.4.6. Một vị trí bộ nhớ được lưu trữ trên bộ xử lý 0 khi một luồng trên bộ xử lý 1 yêu cầu dữ liệu. Để có được nó, bộ xử lý 0 cần phải dừng những gì nó đang làm, ghi lại thông tin vào bộ nhớ chính và sau đó cho phép bộ xử lý 1 đọc nó từ bộ nhớ. Trong hoạt động này cả hai bộ vi xử lý chờ đợi. Rất có khả năng mã như vậy chạy * khá chậm chừ* trên nhiều bộ xử lý khi so sánh với việc triển khai một bộ xử lý đơn hiệu quả. Đây là một lý do nữa cho lý do tại sao có một giới hạn thực tế đối với kích thước bộ nhớ cache (bên cạnh kích thước vật lý của chúng).

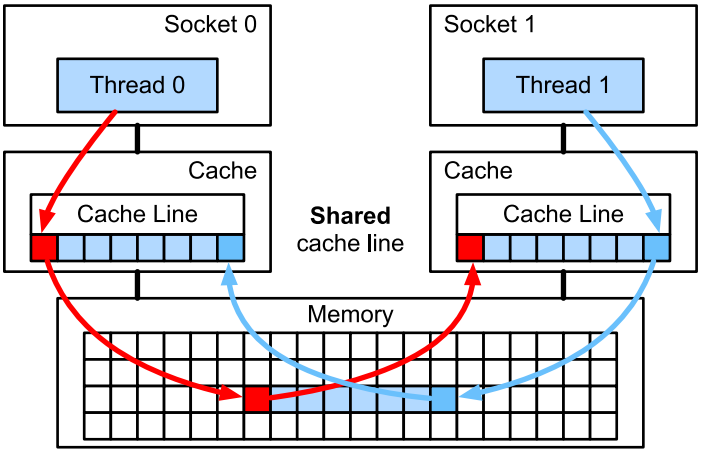


Fig. 13.4.6: False sharing (image courtesy of Intel).

13.4.5 GPU và các bộ tăng tốc khác

Không phải là một cường điệu khi tuyên bố rằng học sâu sẽ không thành công nếu không có GPU. Bằng cùng một mã thông báo, khá hợp lý khi lập luận rằng vận may của các nhà sản xuất GPU đã tăng đáng kể do học sâu. Sự đồng phát triển của phần cứng và thuật toán này đã dẫn đến một tình huống mà học sâu tốt hơn hoặc tồi tệ hơn là mô hình mô hình mô hình hóa thống kê thích hợp hơn. Do đó, nó trả tiền để hiểu những lợi ích cụ thể mà GPU và các máy gia tốc liên quan như TPU (Jouppi et al., 2017).

Lưu ý là một sự khác biệt thường được thực hiện trong thực tế: máy gia tốc được tối ưu hóa cho đào tạo hoặc suy luận. Đối với cái sau, chúng ta chỉ cần tính toán sự lan truyền chuyển tiếp trong một mạng. Không cần lưu trữ dữ liệu trung gian để truyền ngược. Hơn nữa, chúng ta có thể không cần tính toán rất chính xác (FP16 hoặc INT8 thường là đủ). Mặt khác, trong quá trình đào tạo tất cả các kết quả trung gian cần lưu trữ để tính toán độ dốc. Hơn nữa, tích lũy gradient đòi hỏi độ chính xác cao hơn để tránh dòng chảy số (hoặc tràn). Điều này có nghĩa là FP16 (hoặc độ chính xác hỗn hợp với FP32) là yêu cầu tối thiểu. Tất cả những điều này đòi hỏi bộ nhớ nhanh hơn và lớn hơn (HBM2 so với GDDR6) và sức mạnh xử lý nhiều hơn. Ví dụ, GPU Turing[156] T4 của NVIDIA được tối ưu hóa cho suy luận trong khi GPU V100 thích hợp hơn để đào tạo.

[156] https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/

Nhớ lại vectorization như minh họa trong Fig. 13.4.5. Thêm các đơn vị vector vào lõi bộ xử lý cho phép chúng tôi tăng thông lượng đáng kể. Ví dụ: trong ví dụ trong Fig. 13.4.5, chúng tôi đã có thể thực hiện 16 thao tác cùng một lúc. Đầu tiên, điều gì sẽ xảy ra nếu chúng ta thêm các hoạt động tối ưu hóa không chỉ các hoạt động giữa các vectơ mà còn giữa các ma trận? Chiến lược này dẫn đến lõi tensor (sẽ được bảo hiểm trong thời gian ngắn). Thứ hai, nếu chúng ta thêm nhiều lõi nữa thì sao? Tóm lại, hai chiến lược này tóm tắt các quyết định thiết kế trong GPU. Fig. 13.4.7 đưa ra một cái nhìn tổng quan về một khối xử lý cơ bản. Nó chứa 16 số nguyên và 16 đơn vị điểm nổi. Thêm vào đó, hai lõi tensor đẩy nhanh một tập hợp con hẹp của các hoạt động bổ sung có liên quan đến học sâu. Mỗi bộ xử lý phát trực tuyến bao gồm bốn khối như vậy.

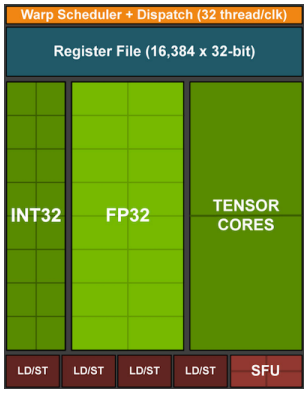


Fig. 13.4.7: NVIDIA Turing processing block (image courtesy of NVIDIA).

Tiếp theo, 12 bộ xử lý đa phát trực tuyến được nhóm thành các cụm xử lý đồ họa tạo nên bộ xử lý TU102 cao cấp. Các kênh bộ nhớ phong phú và bộ nhớ cache L2 bổ sung cho thiết lập. Fig. 13.4.8 có các chi tiết liên quan. Một trong những lý do để thiết kế một thiết bị như vậy là các khối riêng lẻ có thể được thêm hoặc loại bỏ khi cần thiết để cho phép các chip nhỏ gọn hơn và giải quyết các vấn đề năng suất (các mô-đun bị lỗi có thể không được kích hoạt). May mắn thay lập trình các thiết bị như vậy cũng được ẩn từ các nhà nghiên cứu học sâu thông thường bên dưới các lớp của CIDA và mã khung. Đặc biệt, nhiều hơn một trong các chương trình cũng có thể được thực thi đồng thời trên GPU, với điều kiện là có các tài nguyên có sẵn. Tuy nhiên, nó trả tiền để nhận thức được những hạn chế của các thiết bị để tránh chọn các mô hình không phù hợp với bộ nhớ thiết bị.

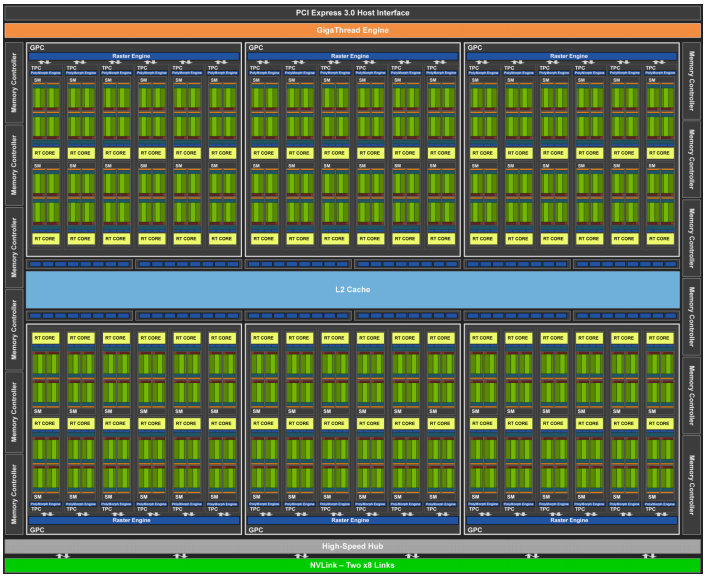


Fig. 13.4.8: NVIDIA Turing architecture (image courtesy of NVIDIA)

Một khía cạnh cuối cùng đáng nói đến chi tiết hơn là * lõi tensor*. Chúng là một ví dụ về xu hướng gần đây của việc thêm các mạch tối ưu hóa hơn có hiệu quả đặc biệt cho việc học sâu. Ví dụ, TPU đã thêm một mảng tâm thu (Kung, 1988) để nhân ma trận nhanh. Ở đó, thiết kế là hỗ trợ một số rất nhỏ (một cho thế hệ đầu tiên của TPU) của các hoạt động lớn. Lõi Tensor ở đầu kia. Chúng được tối ưu hóa cho các hoạt động nhỏ liên quan đến giữa các ma trận 4×4 và 16×16, tùy thuộc vào độ chính xác số của chúng. Fig. 13.4.9 đưa ra tổng quan về tối ưu hóa.

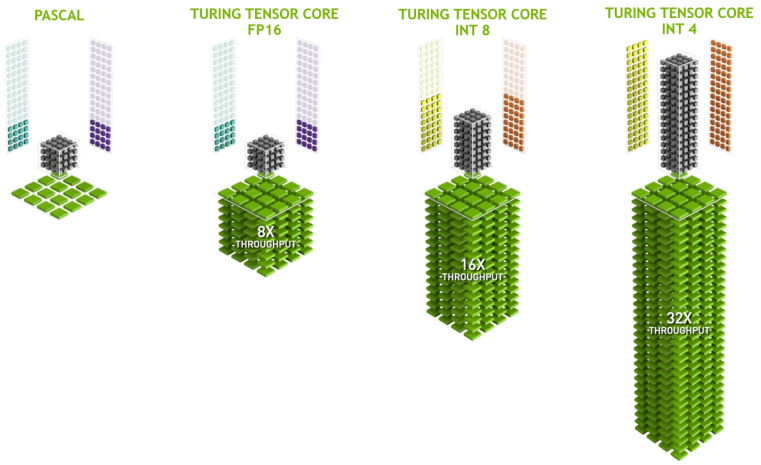


Fig. 13.4.9: NVIDIA tensor cores in Turing (image courtesy of NVIDIA).

Rõ ràng là khi tối ưu hóa tính toán, chúng ta sẽ tạo ra những thỏa hiệp nhất định. Một trong số đó là GPU không tốt trong việc xử lý các gián đoạn và dữ liệu thưa thớt. Mặc dù có những ngoại lệ đáng chú ý, chẳng hạn như Gunrock[157] (Wang et al., 2016), mô hình truy cập của ma trận thưa thớt và vectơ không phù hợp với các hoạt động đọc liên tục băng thông cao trong đó GPU vượt trội. Phù hợp với cả hai mục tiêu là một lĩnh vực nghiên cứu tích cực. Xem ví dụ, DGL[158], một thư viện được điều chỉnh để học sâu về đồ thị.

13.4.6 Mạng và xe buýt

Bất cứ khi nào một thiết bị duy nhất không đủ để tối ưu hóa, chúng ta cần chuyển dữ liệu đến và đi từ nó để đồng bộ hóa xử lý. Đây là nơi mạng và xe buýt có ích. Chúng tôi có một số thông số thiết kế: băng thông, chi phí, khoảng cách và tính linh hoạt. Ở một đầu, chúng tôi có WiFi có phạm vi khá tốt, rất dễ sử dụng (không có dây, sau tất cả), giá rẻ nhưng nó cung cấp băng thông và độ trễ tương đối tầm thường. Không có nhà nghiên cứu máy học nào trong tâm trí đúng đắn của họ sẽ sử dụng nó để xây dựng một cụm máy chủ. Trong những gì sau đây, chúng tôi tập trung vào các kết nối phù hợp cho việc học sâu.

- ** PCIe** là một xe buýt chuyên dụng cho các kết nối điểm-điểm băng thông rất cao (lên đến 32 Gb/s trên PCIe 4.0 trong một khe 16 làn) cho mỗi làn. Độ trễ theo thứ tự micro giây một chữ số (5 µs). Liên kết PCIe rất quý giá. Bộ xử lý chỉ có số lượng hạn chế: EPYC 3 của AMD có 128 làn xe, Xeon của Intel có tới 48 làn trên mỗi chip; trên CPU cấp máy tính để bàn, các con số tương ứng là 20 (Ryzen 9) và 16 (Core i9). Vì GPU thường có 16 làn, điều này giới hạn số lượng GPU có thể kết nối với CPU ở băng thông đầy đủ. Rốt cuộc, họ cần chia sẻ các liên kết với các thiết bị ngoại vi băng thông cao khác như lưu trữ và Ethernet. Cũng giống như với truy cập RAM, chuyển số lượng lớn là thích hợp hơn do giảm chi phí gói.
- **Ethernet** là cách kết nối máy tính được sử dụng phổ biến nhất. Mặc dù nó chậm hơn đáng kể so với PCIe, nhưng nó rất rẻ và đàn hồi để cài đặt và bao phủ khoảng cách xa hơn nhiều. Băng thông điển hình

[157] https://github.com/gunrock/gunrock
[158] http://dgl.ai

cho các máy chủ cấp thấp là 1 GBit/s. thiết bị cao cấp (ví dụ, C5 instances[159] trong đám mây) cung cấp giữa 10 và 100 GBit/s băng thông. Như trong tất cả các trường hợp trước truyền dữ liệu có chi phí đáng kể. Lưu ý rằng chúng ta hầu như không bao giờ sử dụng Ethernet thô trực tiếp mà là một giao thức được thực thi trên đầu trang của kết nối vật lý (chẳng hạn như UDP hoặc TCP/IP). Điều này bổ sung thêm chi phí. Giống như PCIe, Ethernet được thiết kế để kết nối hai thiết bị, ví dụ, một máy tính và một công tắc.

- ** Switches** cho phép chúng tôi kết nối nhiều thiết bị theo cách mà bất kỳ cặp nào trong số họ có thể thực hiện kết nối điểm-điểm (thường là băng thông đầy đủ) cùng một lúc. Ví dụ: các thiết bị chuyển mạch Ethernet có thể kết nối 40 máy chủ ở băng thông mặt cắt ngang cao. Lưu ý rằng các thiết bị chuyển mạch không phải là duy nhất cho các mạng máy tính truyền thống. Ngay cả làn đường PCIe cũng có thể là switched[160]. Điều này xảy ra, ví dụ, để kết nối một số lượng lớn GPU với bộ xử lý máy chủ, như trường hợp của P2 instances[161].
- **NVLink** là một thay thế cho PCIe khi nói đến kết nối băng thông rất cao. Nó cung cấp tốc độ truyền dữ liệu lên đến 300 Gbit/s cho mỗi liên kết. GPU máy chủ (Volta V100) có sáu liên kết trong khi GPU cấp người tiêu dùng (RTX 2080 Ti) chỉ có một liên kết, hoạt động với tốc độ giảm 100 Gbit/s. Chúng tôi khuyên bạn nên sử dụng NCCL[162] để đạt được truyền dữ liệu cao giữa các GPU.

13.4.7 Nhiều số độ trễ hơn

Tóm tắt trong Section 13.4.7 và Section 13.4.7 là từ Eliot Eshelman[163], người duy trì phiên bản cập nhật của các số là GitHub gist[164].

Action	Time	Notes
L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
Branch mispredict	6 ns	15 ~ 20 cycles
L3 cache hit (unshared cache)	16 ns	42 cycles
L3 cache hit (shared in another core)	25 ns	65 cycles
Mutex lock/unlock	25 ns	
L3 cache hit (modified in another core)	29 ns	75 cycles
L3 cache hit (on a remote CPU socket)	40 ns	100 ~ 300 cycles (40 ~ 116 ns)
QPI hop to a another CPU (per hop)	40 ns	
64MB memory ref. (local CPU)	46 ns	TinyMemBench on Broadwell E5-2690v4
64MB memory ref. (remote CPU)	70 ns	TinyMemBench on Broadwell E5-2690v4
256MB memory ref. (local CPU)	75 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random write	94 ns	UCSD Non-Volatile Systems Lab
256MB memory ref. (remote CPU)	120 ns	TinyMemBench on Broadwell E5-2690v4
Intel Optane random read	305 ns	UCSD Non-Volatile Systems Lab
Send 4KB over 100 Gbps HPC fabric	1 μs	MVAPICH2 over Intel Omni-Path
Compress 1KB with Google Snappy	3 μs	
Send 4KB over 10 Gbps ethernet	10 μs	

continues on next page

[159] https://aws.amazon.com/ec2/instance-types/c5/
[160] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches
[161] https://aws.amazon.com/ec2/instance-types/p2/
[162] https://github.com/NVIDIA/nccl
[163] https://gist.github.com/eshelman
[164] https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646

Table 13.4.1 – continued from previous page

Action	Time	Notes
Write 4KB randomly to NVMe SSD	30 µs	DC P3608 NVMe SSD (QOS 99% is 500µs)
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCIe 3.0 x16 link
Read 4KB randomly from NVMe SSD	120 µs	DC P3608 NVMe SSD (QOS 99%)
Read 1MB sequentially from NVMe SSD	208 µs	~4.8GB/s DC P3608 NVMe SSD
Write 4KB randomly to SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Read 4KB randomly from SATA SSD	500 µs	DC S3510 SATA SSD (QOS 99.9%)
Round trip within same datacenter	500 µs	One-way ping is ~250µs
Read 1MB sequentially from SATA SSD	2 ms	~550MB/s DC S3510 SATA SSD
Read 1MB sequentially from disk	5 ms	~200MB/s server HDD
Random Disk Access (seek+rotation)	10 ms	
Send packet CA->Netherlands->CA	150 ms	

Table: Số trễ chung.

Action	Time	Notes
GPU Shared Memory access	30 ns	30~90 cycles (bank conflicts add latency)
GPU Global Memory access	200 ns	200~800 cycles
Launch CUDA kernel on GPU	10 µs	Host CPU instructs GPU to start kernel
Transfer 1MB to/from NVLink GPU	30 µs	~33GB/s on NVIDIA 40GB NVLink
Transfer 1MB to/from PCI-E GPU	80 µs	~12GB/s on PCI-Express x16 link

Table: Số độ trễ cho GPU NVIDIA Tesla.

13.4.8 Tóm tắt

- Các thiết bị có chi phí cho các hoạt động. Do đó, điều quan trọng là nhằm mục đích cho một số lượng nhỏ các chuyển lớn hơn là nhiều chuyển nhỏ. Điều này áp dụng cho RAM, SSD, mạng và GPU.
- Vector hóa là chìa khóa cho hiệu suất. Hãy chắc chắn rằng bạn nhận thức được các khả năng cụ thể của máy gia tốc của bạn. Ví dụ, một số CPU Intel Xeon đặc biệt tốt cho các hoạt động INT8, GPU NVIDIA Volta xuất sắc ở các hoạt động ma trận ma trận FP16 và NVIDIA Turing tỏa sáng ở các hoạt động FP16, INT8 và INT4.
- Tràn số do các kiểu dữ liệu nhỏ có thể là một vấn đề trong quá trình đào tạo (và ở mức độ thấp hơn trong quá trình suy luận).
- Aliasing có thể làm giảm đáng kể hiệu suất. Ví dụ, căn chỉnh bộ nhớ trên CPU 64 bit nên được thực hiện đối với ranh giới 64 bit. Trên GPU, bạn nên giữ cho các kích thước phức tạp được căn chỉnh, ví dụ, với lõi tensor.
- Phù hợp với thuật toán của bạn với phần cứng (ví dụ: dấu chân bộ nhớ và băng thông). Tốc độ lớn (đơn đặt hàng cường độ) có thể đạt được khi lắp các tham số vào bộ nhớ đệm.
- Chúng tôi khuyên bạn nên phác thảo hiệu suất của một thuật toán mới lạ trên giấy trước khi xác minh kết quả thử nghiệm. Sự khác biệt của một trật tự cường độ trở lên là những lý do cho mối quan tâm.
- Sử dụng profilers để gỡ lỗi tắc nghẽn hiệu suất.
- Phần cứng đào tạo và suy luận có điểm ngọt khác nhau về giá cả và hiệu suất.

13.4.9 Bài tập

1. Viết mã C để kiểm tra xem có bất kỳ sự khác biệt nào về tốc độ giữa việc truy cập bộ nhớ được căn chỉnh hay không liên kết so với giao diện bộ nhớ ngoài. Gợi ý: hãy cẩn thận với các hiệu ứng bộ nhớ đệm.
2. Kiểm tra sự khác biệt về tốc độ giữa việc truy cập bộ nhớ theo trình tự hoặc với một sải chân nhất định.
3. Làm thế nào bạn có thể đo kích thước bộ nhớ cache trên CPU?
4. Làm thế nào bạn sẽ bố trí dữ liệu trên nhiều kênh bộ nhớ để có băng thông tối đa? Làm thế nào bạn sẽ đặt nó ra nếu bạn có nhiều chủ đề nhỏ?
5. Ổ cứng hạng doanh nghiệp đang quay với tốc độ 10.000 vòng/phút. Thời gian hoàn toàn tối thiểu một ổ cứng cần phải dành trường hợp xấu nhất trước khi nó có thể đọc dữ liệu (bạn có thể giả định rằng đầu di chuyển gần như ngay lập tức)? Tại sao ổ cứng 2.5 "trở nên phổ biến cho các máy chủ thương mại (tương đối với ổ đĩa 3.5" và 5,25")?
6. Giả sử rằng một nhà sản xuất HDD tăng mật độ lưu trữ từ 1 Tbit mỗi inch vuông lên 5 Tbit mỗi inch vuông. Bạn có thể lưu trữ bao nhiêu thông tin trên một chiếc nhẫn trên ổ cứng 2.5 "? Có sự khác biệt giữa các bản nhạc bên trong và bên ngoài không?
7. Đi từ 8 bit đến 16 bit loại dữ liệu làm tăng lượng silicon xấp xỉ bốn lần. Tại sao? Tại sao NVIDIA có thể thêm hoạt động INT4 vào GPU Turing của họ?
8. Làm thế nào nhanh hơn là nó để đọc về phía trước thông qua bộ nhớ so với đọc ngược? Con số này có khác nhau giữa các máy tính và nhà cung cấp CPU khác nhau không? Tại sao? Viết mã C và thử nghiệm với nó.
9. Bạn có thể đo kích thước bộ nhớ cache của đĩa của bạn? Nó là gì cho một ổ cứng điển hình? SSD có cần bộ nhớ cache không?
10. Đo chi phí gói khi gửi tin nhắn qua Ethernet. Tra cứu sự khác biệt giữa các kết nối UDP và TCP/IP.
11. Truy cập bộ nhớ trực tiếp cho phép các thiết bị khác ngoài CPU ghi (và đọc) trực tiếp vào bộ nhớ (từ). Tại sao đây là một ý tưởng tốt?
12. Nhìn vào các số hiệu suất cho GPU Turing T4. Tại sao hiệu suất "chỉ" tăng gấp đôi khi bạn đi từ FP16 sang INT8 và INT4?
13. Thời gian ngắn nhất nên mất cho một gói trong một chuyến đi khứ hồi giữa San Francisco và Amsterdam là bao nhiêu? Gợi ý: bạn có thể giả định rằng khoảng cách là 10.000 km.

Discussions[165]

13.5 Đào tạo về nhiều GPU

Cho đến nay chúng tôi đã thảo luận về cách đào tạo các mô hình hiệu quả trên CPU và GPU. Chúng tôi thậm chí còn cho thấy cách các khung học sâu cho phép người ta song song tính toán và giao tiếp tự động giữa chúng trong Section 13.3. Chúng tôi cũng đã chỉ ra trong Section 6.5 cách liệt kê tất cả các GPU có sẵn trên máy tính bằng lệnh `nvidia-smi`. Những gì chúng tôi đã làm *không* thảo luận là làm thế nào để thực sự song song hóa đào tạo học sâu. Thay vào đó, chúng tôi ngụ ý trong việc truyền rằng bằng cách nào đó người ta sẽ chia dữ liệu trên nhiều thiết bị và làm cho nó hoạt động. Phần hiện tại điền vào các chi tiết và cho thấy cách đào tạo mạng song song khi bắt đầu từ đầu. Chi tiết về cách tận dụng chức năng trong các API cấp cao được xuống Section

[165] https://discuss.d2l.ai/t/363

13.6. Chúng tôi giả định rằng bạn đã quen thuộc với các thuật toán giảm dần ngẫu nhiên minibatch chẳng hạn như các thuật toán được mô tả trong Section 12.5.

13.5.1 Chia tách vấn đề

Chúng ta hãy bắt đầu với một vấn đề tầm nhìn máy tính đơn giản và một mạng hơi cổ xưa, ví dụ, với nhiều lớp phức tạp, tổng hợp, và có thể là một vài lớp kết nối hoàn toàn cuối cùng. Đó là, chúng ta hãy bắt đầu với một mạng trông khá giống với LeNet (LeCun et al., 1998) hoặc AlexNet (Krizhevsky et al., 2012). Với nhiều GPU (2 nếu là máy chủ máy tính để bàn, 4 trên một phiên bản AWS g4dn.12xlarge, 8 trên p3.16xlarge hoặc 16 trên p2.16xlarge), chúng tôi muốn phân vùng đào tạo theo cách để đạt được tốc độ tốt trong khi đồng thời được hưởng lợi từ các lựa chọn thiết kế đơn giản và tái tạo. Sau khi tất cả, nhiều GPU tăng cả * bộ nhớ* và * tính tị* khả năng. Tóm lại, chúng tôi có các lựa chọn sau, đưa ra một loạt dữ liệu đào tạo mà chúng tôi muốn phân loại.

Đầu tiên, chúng ta có thể phân vùng mạng trên nhiều GPU. Đó là, mỗi GPU lấy như là đầu vào dữ liệu chảy vào một lớp cụ thể, xử lý dữ liệu trên một số lớp tiếp theo và sau đó gửi dữ liệu đến GPU tiếp theo. Điều này cho phép chúng tôi xử lý dữ liệu với các mạng lớn hơn khi so sánh với những gì một GPU duy nhất có thể xử lý. Bên cạnh đó, dấu chân bộ nhớ trên mỗi GPU có thể được kiểm soát tốt (đó là một phần nhỏ của tổng dấu chân mạng).

Tuy nhiên, giao diện giữa các lớp (và do đó GPU) yêu cầu đồng bộ hóa chặt chẽ. Điều này có thể khó khăn, đặc biệt nếu khối lượng công việc tính toán không khớp đúng giữa các lớp. Vấn đề trở nên trầm trọng hơn đối với số lượng lớn GPU. Giao diện giữa các lớp cũng yêu cầu một lượng lớn truyền dữ liệu, chẳng hạn như kích hoạt và gradient. Điều này có thể áp đảo băng thông của các bus GPU. Hơn nữa, các hoạt động chuyên sâu về tính toán, nhưng tuần tự là không tầm thường đối với phân vùng. Xem ví dụ, (Mirhoseini et al., 2017) để có một nỗ lực tốt nhất trong vấn đề này. Nó vẫn là một vấn đề khó khăn và không rõ liệu có thể đạt được tỷ lệ tốt (tuyến tính) trên các vấn đề không tầm thường hay không. Chúng tôi không khuyên bạn nên nó trừ khi có khung tuyệt vời hoặc hỗ trợ hệ điều hành để chuỗi nhiều GPU với nhau.

Thứ hai, chúng ta có thể chia công việc theo từng lớp. Ví dụ, thay vì tính toán 64 kênh trên một GPU duy nhất, chúng tôi có thể chia nhỏ vấn đề trên 4 GPU, mỗi kênh tạo dữ liệu cho 16 kênh. Tương tự như vậy, đối với một lớp được kết nối hoàn toàn, chúng ta có thể chia số lượng đơn vị đầu ra. Fig. 13.5.1 (lấy từ (Krizhevsky et al., 2012)) minh họa thiết kế này, nơi chiến lược này được sử dụng để đối phó với GPU có dung lượng bộ nhớ rất nhỏ (2 GB vào thời điểm đó). Điều này cho phép mở rộng quy mô tốt về tính toán, với điều kiện là số lượng kênh (hoặc đơn vị) không quá nhỏ. Bên cạnh đó, nhiều GPU có thể xử lý các mạng ngày càng lớn hơn vì bộ nhớ có sẵn quy mô tuyến tính.

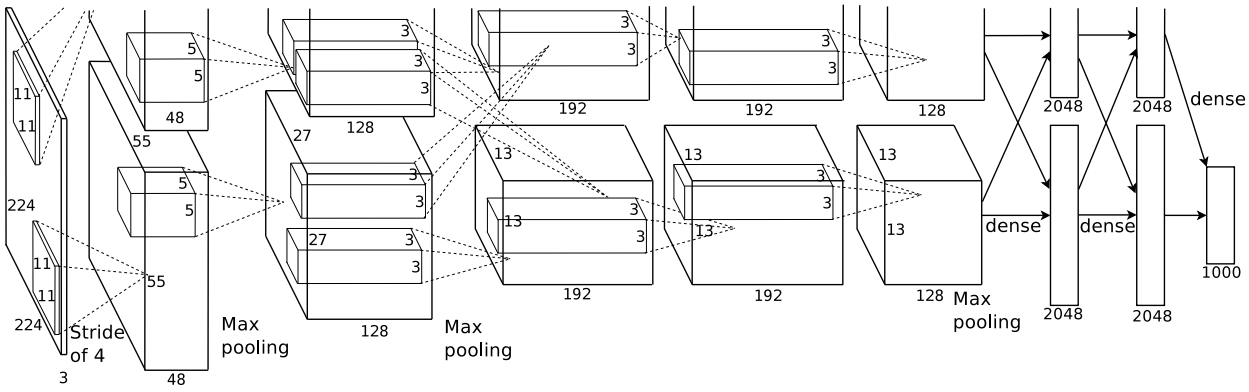


Fig. 13.5.1: Model parallelism in the original AlexNet design due to limited GPU memory.

Tuy nhiên, chúng ta cần một * rất lớn* số lượng hoạt động đồng bộ hóa hoặc rào cản vì mỗi lớp phụ thuộc

vào kết quả từ tất cả các lớp khác. Hơn nữa, lượng dữ liệu cần được truyền có khả năng thậm chí còn lớn hơn so với khi phân phối các lớp trên GPU. Do đó, chúng tôi không khuyên bạn nên cách tiếp cận này do chi phí băng thông và độ phức tạp của nó.

Cuối cùng, chúng ta có thể phân vùng dữ liệu trên nhiều GPU. Bằng cách này, tất cả các GPU thực hiện cùng một loại công việc, mặc dù trên các quan sát khác nhau. Gradient được tổng hợp trên các GPU sau mỗi minibatch dữ liệu đào tạo. Đây là cách tiếp cận đơn giản nhất và nó có thể được áp dụng trong mọi tình huống. Chúng tôi chỉ cần đồng bộ hóa sau mỗi minibatch. Điều đó nói rằng, rất mong muốn bắt đầu trao đổi các tham số gradient trong khi những người khác vẫn đang được tính toán. Hơn nữa, số lượng GPU lớn hơn dẫn đến kích thước minibatch lớn hơn, do đó tăng hiệu quả đào tạo. Tuy nhiên, việc thêm nhiều GPU không cho phép chúng tôi đào tạo các mô hình lớn hơn.

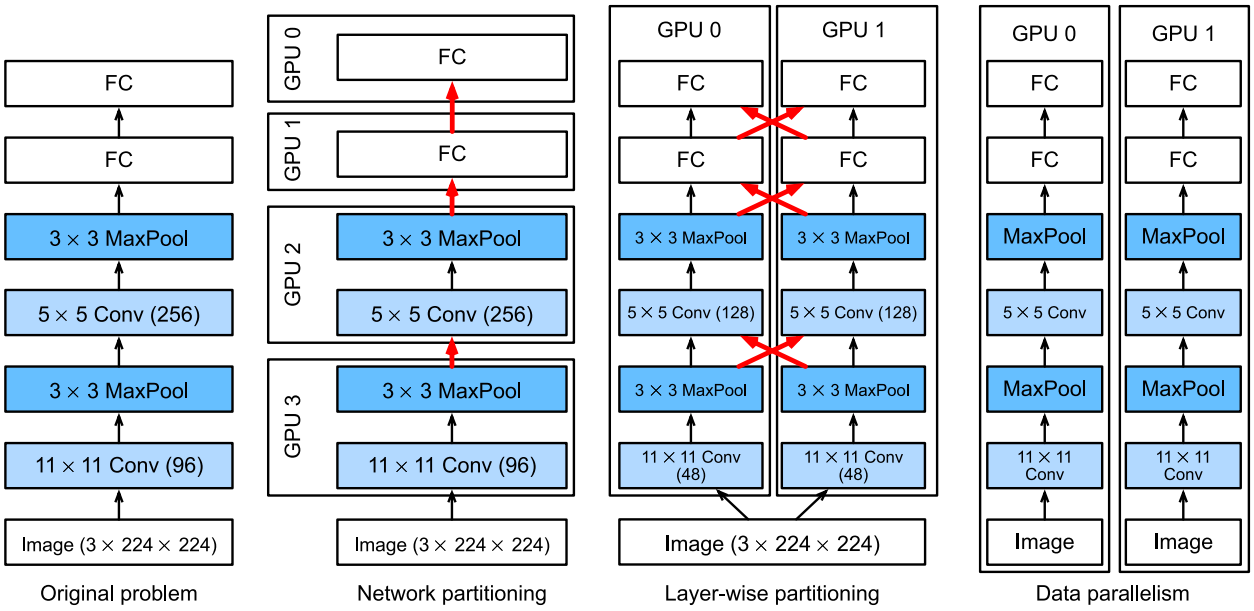


Fig. 13.5.2: Parallelization on multiple GPUs. From left to right: original problem, network partitioning, layerwise partitioning, data parallelism.

Một so sánh các cách song song khác nhau trên nhiều GPU được mô tả trong Fig. 13.5.2. Nhìn chung, song song dữ liệu là cách thuận tiện nhất để tiến hành, miễn là chúng tôi có quyền truy cập vào GPU với bộ nhớ đủ lớn. Xem thêm (Li et al., 2014) để biết mô tả chi tiết về phân vùng cho đào tạo phân tán. Bộ nhớ GPU từng là một vấn đề trong những ngày đầu học sâu. Đến bây giờ vấn đề này đã được giải quyết cho tất cả nhưng các trường hợp bất thường nhất. Chúng tôi tập trung vào sự song song dữ liệu trong những gì sau.

13.5.2 Dữ liệu song song

Giả sử rằng có k GPU trên máy. Với mô hình được đào tạo, mỗi GPU sẽ duy trì một tập hợp đầy đủ các tham số mô hình một cách độc lập mặc dù các giá trị tham số trên các GPU giống hệt nhau và đồng bộ hóa. Ví dụ, Fig. 13.5.3 minh họa đào tạo với sự song song dữ liệu khi $k = 2$.

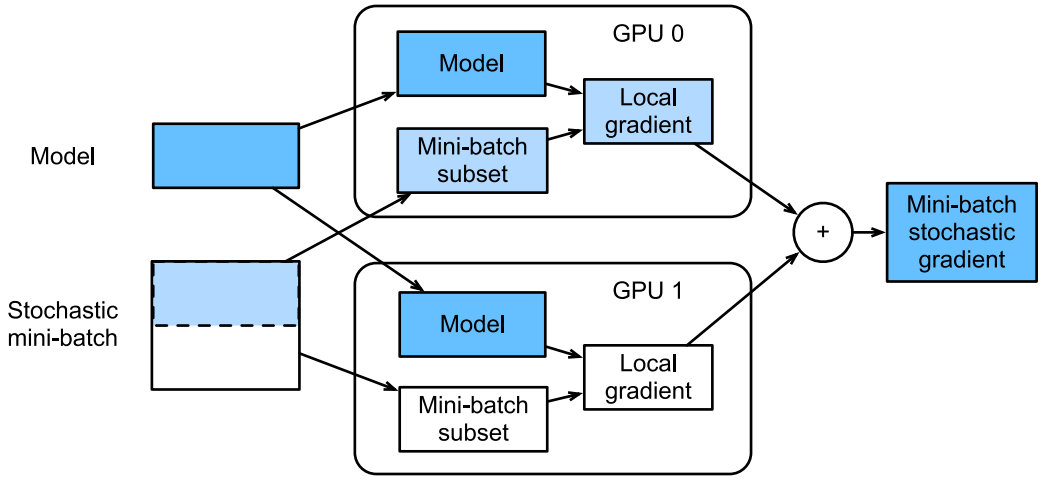


Fig. 13.5.3: Calculation of minibatch stochastic gradient descent using data parallelism on two GPUs.

Nói chung, việc đào tạo tiến hành như sau:

- Trong bất kỳ lặp lại đào tạo nào, được đưa ra một minibatch ngẫu nhiên, chúng tôi chia các ví dụ trong lô thành k phần và phân phối chúng đều trên GPU.
- Mỗi GPU tính toán tổn thất và độ dốc của các tham số mô hình dựa trên tập hợp con minibatch mà nó đã được gán.
- Các gradient cục bộ của mỗi GPU k được tổng hợp để có được gradient ngẫu nhiên minibatch hiện tại.
- Gradient tổng hợp được phân phối lại cho mỗi GPU.
- Mỗi GPU sử dụng gradient stochastic minibatch này để cập nhật bộ thông số mô hình hoàn chỉnh mà nó duy trì.

Lưu ý rằng trong thực tế, chúng tôi * tăng* kích thước minibatch k-lần khi đào tạo trên k GPU sao cho mỗi GPU có cùng một lượng công việc để làm như thể chúng tôi chỉ được đào tạo trên một GPU duy nhất. Trên máy chủ 16-GPU, điều này có thể làm tăng kích thước minibatch đáng kể và chúng tôi có thể phải tăng tốc độ học tập cho phù hợp. Cũng lưu ý rằng việc chuẩn hóa hàng loạt trong Section 8.5 cần được điều chỉnh, ví dụ, bằng cách giữ một hệ số bình thường hóa hàng loạt riêng cho mỗi GPU. Trong những gì sau đây chúng tôi sẽ sử dụng một mạng đồ chơi để minh họa đào tạo đa GPU.

```
%matplotlib inline
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

13.5.3 Một mạng đồ chơi

Chúng tôi sử dụng LeNet như được giới thiệu trong Section 7.6 (với những sửa đổi nhỏ). Chúng tôi xác định nó từ đầu để minh họa chi tiết trao đổi và đồng bộ hóa tham số.

```
# Initialize model parameters
scale = 0.01
W1 = torch.randn(size=(20, 1, 3, 3)) * scale
b1 = torch.zeros(20)
W2 = torch.randn(size=(50, 20, 5, 5)) * scale
b2 = torch.zeros(50)
W3 = torch.randn(size=(800, 128)) * scale
b3 = torch.zeros(128)
W4 = torch.randn(size=(128, 10)) * scale
b4 = torch.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = F.conv2d(input=X, weight=params[0], bias=params[1])
    h1_activation = F.relu(h1_conv)
    h1 = F.avg_pool2d(input=h1_activation, kernel_size=(2, 2), stride=(2, 2))
    h2_conv = F.conv2d(input=h1, weight=params[2], bias=params[3])
    h2_activation = F.relu(h2_conv)
    h2 = F.avg_pool2d(input=h2_activation, kernel_size=(2, 2), stride=(2, 2))
    h2 = h2.reshape(h2.shape[0], -1)
    h3_linear = torch.mm(h2, params[4]) + params[5]
    h3 = F.relu(h3_linear)
    y_hat = torch.mm(h3, params[6]) + params[7]
    return y_hat

# Cross-entropy loss function
loss = nn.CrossEntropyLoss(reduction='none')
```

13.5.4 Đồng bộ hóa dữ liệu

Để đào tạo đa GPU hiệu quả, chúng tôi cần hai hoạt động cơ bản. Đầu tiên chúng ta cần phải có khả năng phân phối một danh sách các tham số cho nhiều thiết bị và đính kèm gradient (`get_params`). Không có thông số, không thể đánh giá mạng trên GPU. Thứ hai, chúng ta cần khả năng tổng hợp các tham số trên nhiều thiết bị, tức là chúng ta cần một hàm `allreduce`.

```
def get_params(params, device):
    new_params = [p.to(device) for p in params]
    for p in new_params:
        p.requires_grad_()
    return new_params
```

Hãy để chúng tôi thử nó bằng cách sao chép các tham số mô hình vào một GPU.

```
new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)
```

```
b1 weight: tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
↪ 0., 0., 0., 0., 0.],
       device='cuda:0', requires_grad=True)
b1 grad: None
```

Vì chúng tôi chưa thực hiện bất kỳ tính toán nào, gradient liên quan đến tham số thiên vị vẫn bằng không. Bây giờ chúng ta hãy giả định rằng chúng ta có một vector phân phối trên nhiều GPU. Chức năng `allreduce` sau đây thêm tất cả các vectơ và phát sóng kết quả trở lại tất cả các GPU . Lưu ý rằng để làm việc này, chúng ta cần sao chép dữ liệu vào thiết bị tích lũy kết quả.

```
def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].to(data[0].device)
    for i in range(1, len(data)):
        data[i][:] = data[0].to(data[i].device)
```

Hãy để chúng tôi kiểm tra điều này bằng cách tạo các vectơ với các giá trị khác nhau trên các thiết bị khác nhau và tổng hợp chúng.

```
data = [torch.ones((1, 2), device=d2l.try_gpu(i)) * (i + 1) for i in range(2)]
print('before allreduce:\n', data[0], '\n', data[1])
allreduce(data)
print('after allreduce:\n', data[0], '\n', data[1])
```

```
before allreduce:
 tensor([[1., 1.]], device='cuda:0')
 tensor([[2., 2.]], device='cuda:1')
after allreduce:
 tensor([[3., 3.]], device='cuda:0')
 tensor([[3., 3.]], device='cuda:1')
```

13.5.5 Phân phối dữ liệu

Chúng ta cần một chức năng tiện ích đơn giản để phân phối một minibatch đều trên nhiều GPU . Ví dụ: trên hai GPU, chúng tôi muốn có một nửa dữ liệu được sao chép vào một trong hai GPU. Vì nó thuận tiện hơn và súc tích hơn, chúng tôi sử dụng chức năng tích hợp từ khung học sâu để thử nó trên ma trận 4×5.

```
data = torch.arange(20).reshape(4, 5)
devices = [torch.device('cuda:0'), torch.device('cuda:1')]
split = nn.parallel.scatter(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

```
input : tensor([[ 0,  1,  2,  3,  4],
        [ 5,  6,  7,  8,  9],
        [10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]])
load into [device(type='cuda', index=0), device(type='cuda', index=1)]
output: (tensor([[0, 1, 2, 3, 4],
```

(continues on next page)

(continued from previous page)

```
        [5, 6, 7, 8, 9]], device='cuda:0'), tensor([[10, 11, 12, 13, 14],
        [15, 16, 17, 18, 19]], device='cuda:1'))
```

Để tái sử dụng sau này, chúng tôi xác định một hàm `split_batch` chia cả dữ liệu và nhãn.

```
#@save
def split_batch(X, y, devices):
    """Split `X` and `y` into multiple devices."""
    assert X.shape[0] == y.shape[0]
    return (nn.parallel.scatter(X, devices),
            nn.parallel.scatter(y, devices))
```

13.5.6 Đào tạo

Bây giờ chúng ta có thể thực hiện đào tạo đa GPU trên một minibatch. Việc thực hiện nó chủ yếu dựa trên cách tiếp cận song song dữ liệu được mô tả trong phần này. Chúng tôi sẽ sử dụng các chức năng phụ mà chúng tôi vừa thảo luận, `allreduce` và `split_and_load`, để đồng bộ hóa dữ liệu giữa nhiều GPU. Lưu ý rằng chúng ta không cần phải viết bất kỳ mã cụ thể nào để đạt được sự song song. Vì đồ thị tính toán không có bất kỳ phụ thuộc nào trên các thiết bị trong một minibatch, nó được thực thi song song *automatically*.

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    # Loss is calculated separately on each GPU
    ls = [loss(lenet(X_shard, device_W), y_shard).sum()
          for X_shard, y_shard, device_W in zip(
              X_shards, y_shards, device_params)]
    for l in ls:  # Backpropagation is performed separately on each GPU
        l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    with torch.no_grad():
        for i in range(len(device_params[0])):
            allreduce([device_params[c][i].grad for c in range(len(devices))])
    # The model parameters are updated separately on each GPU
    for param in device_params:
        d2l.sgd(param, lr, X.shape[0]) # Here, we use a full-size batch
```

Bây giờ, chúng ta có thể định nghĩa chức năng đào tạo. Nó hơi khác so với các chương được sử dụng trong các chương trước: chúng ta cần phân bổ GPU và sao chép tất cả các tham số mô hình cho tất cả các thiết bị. Rõ ràng mỗi lô được xử lý bằng chức năng `train_batch` để xử lý nhiều GPU. Để thuận tiện (và ngắn gọn của mã), chúng tôi tính toán độ chính xác trên một GPU duy nhất, mặc dù điều này là * không hiệu quả* vì các GPU khác không hoạt động.

```
def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    # Copy model parameters to `num_gpus` GPUs
    device_params = [get_params(params, d) for d in devices]
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
```

(continues on next page)

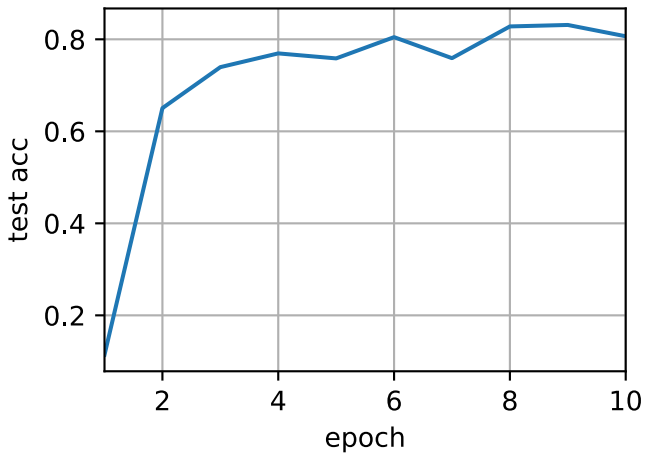
(continued from previous page)

```
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, device_params, devices, lr)
            torch.cuda.synchronize()
        timer.stop()
        # Evaluate the model on GPU 0
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(
            lambda x: lenet(x, device_params[0]), test_iter, devices[0]),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

Hãy để chúng tôi xem điều này hoạt động tốt như thế nào trên một GPU duy nhất. Đầu tiên chúng tôi sử dụng kích thước lô là 256 và tỷ lệ học tập là 0,2.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

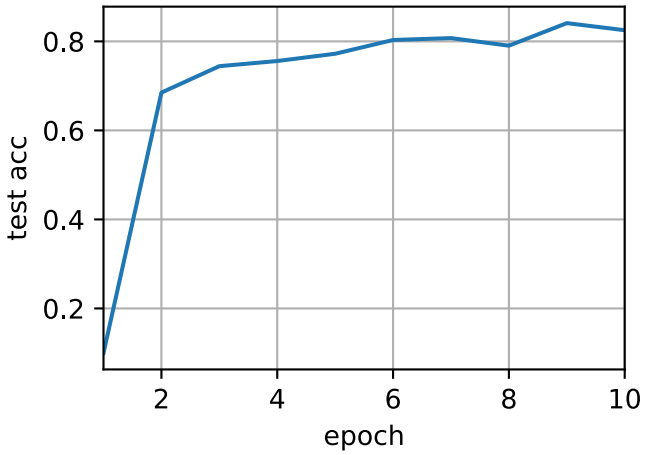
```
test acc: 0.81, 2.8 sec/epoch on [device(type='cuda', index=0)]
```



Bằng cách giữ cho kích thước lô và tỷ lệ học tập không thay đổi và tăng số lượng GPU lên 2, chúng ta có thể thấy rằng độ chính xác kiểm tra vẫn giữ nguyên so với thí nghiệm trước đó. Về các thuật toán tối ưu hóa, chúng giống hệt nhau. Thật không may, không có sự tăng tốc có ý nghĩa nào có thể đạt được ở đây: mô hình đơn giản là quá nhỏ; hơn nữa chúng tôi chỉ có một bộ dữ liệu nhỏ, nơi cách tiếp cận hơi không tinh vi của chúng tôi để thực hiện đào tạo đa GPU phải chịu đựng trên cao Python đáng kể. Chúng tôi sẽ gặp phải các mô hình phức tạp hơn và những cách song song tinh vi hơn trong tương lai. Hãy để chúng tôi xem những gì sẽ xảy ra dù sao cho thời trang-MNIST.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

```
test acc: 0.83, 2.8 sec/epoch on [device(type='cuda', index=0), device(type=
↪'cuda', index=1)]
```



13.5.7 Tóm tắt

- Có nhiều cách để chia đào tạo mạng sâu qua nhiều GPU. Chúng ta có thể chia chúng giữa các lớp, qua các lớp hoặc trên dữ liệu. Hai người trước đây yêu cầu truyền dữ liệu được biên đạo chặt chẽ. Song song dữ liệu là chiến lược đơn giản nhất.
- Đào tạo song song dữ liệu là đơn giản. Tuy nhiên, nó làm tăng kích thước minibatch hiệu quả để có hiệu quả.
- Trong tính song song dữ liệu, dữ liệu được chia thành nhiều GPU, trong đó mỗi GPU thực hiện hoạt động chuyển tiếp và lùi của riêng mình và sau đó gradient được tổng hợp và kết quả được phát trở lại GPU.
- Chúng tôi có thể sử dụng tỷ lệ học tập tăng nhẹ cho các minibatches lớn hơn.

13.5.8 Bài tập

1. Khi đào tạo trên k GPU, hãy thay đổi kích thước minibatch từ b thành $k \cdot b$, tức là, quy mô nó lên theo số lượng GPU.
2. So sánh độ chính xác cho các tỷ lệ học tập khác nhau. Làm thế nào để nó mở rộng quy mô với số lượng GPU?
3. Triển khai chức năng `allreduce` hiệu quả hơn tổng hợp các tham số khác nhau trên các GPU khác nhau? Tại sao nó hiệu quả hơn?
4. Thực hiện tính toán độ chính xác thử nghiệm đa GPU.

Discussions[166]

[166] https://discuss.d2l.ai/t/1669

13.6 Triển khai ngắn gọn cho nhiều GPU

Thực hiện song song từ đầu cho mỗi mô hình mới là không thú vị. Hơn nữa, có lợi ích đáng kể trong việc tối ưu hóa các công cụ đồng bộ hóa cho hiệu suất cao. Trong phần sau đây, chúng tôi sẽ chỉ ra cách thực hiện việc này bằng cách sử dụng API cấp cao của các framework deep learning. Toán học và các thuật toán giống như trong Section 13.5. Khá không ngạc nhiên khi bạn sẽ cần ít nhất hai GPU để chạy mã của phần này.

```
import torch
from torch import nn
from d2l import torch as d2l
```

13.6.1 Một mạng đồ chơi

Hãy để chúng tôi sử dụng một mạng có ý nghĩa hơn một chút so với LeNet từ Section 13.5 mà vẫn đủ dễ dàng và nhanh chóng để đào tạo. Chúng tôi chọn một biến thể ResNet-18 (He et al., 2016a). Vì hình ảnh đầu vào rất nhỏ, chúng tôi sửa đổi nó một chút. Đặc biệt, sự khác biệt so với Section 8.6 là chúng ta sử dụng một hạt nhân phức tạp nhỏ hơn, sải chân và đệm ở đầu. Hơn nữa, chúng tôi loại bỏ lớp tổng hợp tối đa.

```
#@save
def resnet18(num_classes, in_channels=1):
    """A slightly modified ResNet-18 model."""
    def resnet_block(in_channels, out_channels, num_residuals,
                     first_block=False):
        blk = []
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.append(d2l.Residual(in_channels, out_channels,
                                        use_1x1conv=True, strides=2))
            else:
                blk.append(d2l.Residual(out_channels, out_channels))
        return nn.Sequential(*blk)

    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the maximum pooling layer
    net = nn.Sequential(
        nn.Conv2d(in_channels, 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU())
    net.add_module("resnet_block1", resnet_block(64, 64, 2, first_block=True))
    net.add_module("resnet_block2", resnet_block(64, 128, 2))
    net.add_module("resnet_block3", resnet_block(128, 256, 2))
    net.add_module("resnet_block4", resnet_block(256, 512, 2))
    net.add_module("global_avg_pool", nn.AdaptiveAvgPool2d((1,1)))
    net.add_module("fc", nn.Sequential(nn.Flatten(),
                                       nn.Linear(512, num_classes)))
    return net
```

13.6.2 Khởi tạo mạng

Chúng tôi sẽ khởi tạo mạng bên trong vòng đào tạo. Đối với một bồi dưỡng về các phương pháp khởi tạo xem Section 5.8.

```
net = resnet18(10)
# Get a list of GPUs
devices = d2l.try_all_gpus()
# We will initialize the network inside the training loop
```

13.6.3 Đào tạo

Như trước đây, mã đào tạo cần thực hiện một số chức năng cơ bản để song song hiệu quả:

- Các tham số mạng cần được khởi tạo trên tất cả các thiết bị.
- Trong khi lặp lại các minibatches tập dữ liệu sẽ được chia trên tất cả các thiết bị.
- Chúng tôi tính toán sự mất mát và độ dốc của nó song song trên các thiết bị.
- Gradient được tổng hợp và các tham số được cập nhật cho phù hợp.

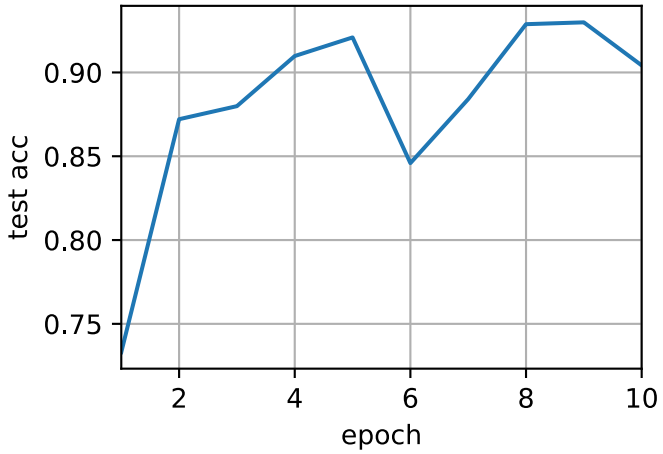
Cuối cùng, chúng tôi tính toán độ chính xác (một lần nữa song song) để báo cáo hiệu suất cuối cùng của mạng. Thói quen đào tạo khá giống với việc triển khai trong các chương trước, ngoại trừ việc chúng ta cần phân chia và tổng hợp dữ liệu.

```
def train(net, num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    devices = [d2l.try_gpu(i) for i in range(num_gpus)]
    def init_weights(m):
        if type(m) in [nn.Linear, nn.Conv2d]:
            nn.init.normal_(m.weight, std=0.01)
    net.apply(init_weights)
    # Set the model on multiple GPUs
    net = nn.DataParallel(net, device_ids=devices)
    trainer = torch.optim.SGD(net.parameters(), lr)
    loss = nn.CrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        net.train()
        timer.start()
        for X, y in train_iter:
            trainer.zero_grad()
            X, y = X.to(devices[0]), y.to(devices[0])
            l = loss(net(X), y)
            l.backward()
            trainer.step()
        timer.stop()
        animator.add(epoch + 1, (d2l.evaluate_accuracy_gpu(net, test_iter),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(devices)}')
```

Hãy để chúng tôi xem làm thế nào điều này hoạt động trong thực tế. Như một khởi động, chúng tôi đào tạo mạng trên một GPU.

```
train(net, num_gpus=1, batch_size=256, lr=0.1)
```

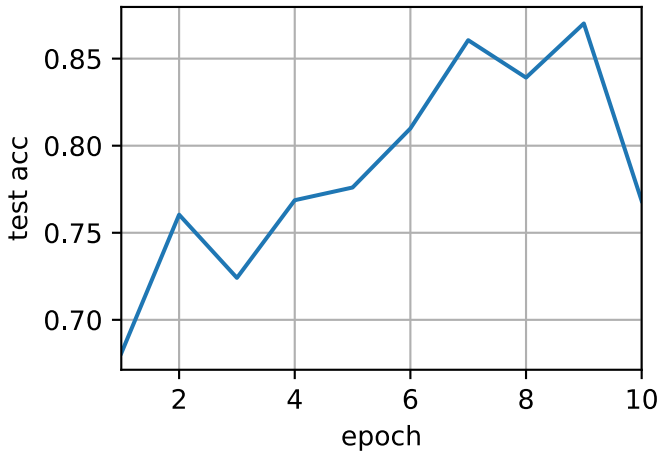
```
test acc: 0.90, 13.8 sec/epoch on [device(type='cuda', index=0)]
```



Tiếp theo chúng ta sử dụng 2 GPU để đào tạo. So với LeNet được đánh giá vào năm Section 13.5, mô hình cho ResNet-18 phức tạp hơn đáng kể. Đây là nơi song song cho thấy lợi thế của nó. Thời gian tính toán lớn hơn một cách có ý nghĩa so với thời gian đồng bộ hóa các tham số. Điều này cải thiện khả năng mở rộng vì chi phí cho song song ít liên quan hơn.

```
train(net, num_gpus=2, batch_size=512, lr=0.2)
```

```
test acc: 0.77, 8.2 sec/epoch on [device(type='cuda', index=0), device(type=
↪'cuda', index=1)]
```



13.6.4 Tóm tắt

- Dữ liệu được tự động đánh giá trên các thiết bị nơi dữ liệu có thể được tìm thấy.
- Hãy cẩn thận để khởi tạo các mạng trên mỗi thiết bị trước khi cố gắng truy cập các tham số trên thiết bị đó. Nếu không bạn sẽ gặp phải một lỗi.
- Các thuật toán tối ưu hóa tự động tổng hợp trên nhiều GPU.

13.6.5 Bài tập

1. Phần này sử dụng ResNet-18. Hãy thử các thời đại khác nhau, quy mô hàng loạt và tỷ lệ học tập. Sử dụng nhiều GPU hơn để tính toán. Điều gì xảy ra nếu bạn dùng thử điều này với 16 GPU (ví dụ: trên phiên bản AWS p2.16xlarge)?
2. Đôi khi, các thiết bị khác nhau cung cấp sức mạnh tính toán khác nhau. Chúng ta có thể sử dụng GPU và CPU cùng một lúc. Làm thế nào chúng ta nên chia công việc? Nó có đáng để nỗ lực không? Tại sao? Tại sao không?

Discussions[167]

13.7 Máy chủ tham số

Khi chúng tôi chuyển từ một GPU duy nhất sang nhiều GPU và sau đó đến nhiều máy chủ chứa nhiều GPU, có thể tất cả đều trải rộng trên nhiều giá đỡ và thiết bị chuyển mạch mạng, các thuật toán của chúng tôi để đào tạo phân tán và song song cần phải trở nên tinh vi hơn nhiều. Chi tiết quan trọng vì các kết nối khác nhau có băng thông rất khác nhau (ví dụ: NVLink có thể cung cấp tới 100 Gb/s trên 6 liên kết trong một cài đặt thích hợp, PCIe 4.0 (16 làn) cung cấp 32 Gb/s, trong khi thậm chí Ethernet 100GbE tốc độ cao chỉ lên tới 10 Gb/s). Đồng thời nó là không hợp lý để mong đợi rằng một mô hình thống kê là một chuyên gia về mạng và hệ thống.

Ý tưởng cốt lõi của máy chủ tham số đã được giới thiệu trong (Smola & Narayanamurthy, 2010) trong bối cảnh các mô hình biến tiềm ẩn phân tán. Một mô tả về ngữ nghĩa đẩy và kéo sau đó theo sau trong (Ahmed et al., 2012) và một mô tả về hệ thống và một thư viện mã nguồn mở theo sau trong (Li et al., 2014). Trong phần sau đây, chúng tôi sẽ thúc đẩy các thành phần cần thiết cho hiệu quả.

13.7.1 Đào tạo dữ liệu song song

Hãy để chúng tôi xem xét phương pháp đào tạo song song dữ liệu để đào tạo phân tán. Chúng tôi sẽ sử dụng điều này để loại trừ tất cả những người khác trong phần này vì nó đơn giản hơn đáng kể để thực hiện trong thực tế. Hầu như không có trường hợp sử dụng (bên cạnh việc học sâu trên đồ thị) trong đó bất kỳ chiến lược nào khác cho song song được ưa thích vì GPU có nhiều bộ nhớ hiện nay. Fig. 13.7.1 mô tả biến thể của sự song song dữ liệu mà chúng tôi đã thực hiện trong Section 13.5. Khía cạnh chính trong đó là sự tập hợp của gradient xảy ra trên GPU 0 trước khi các tham số cập nhật được phát lại cho tất cả các GPU.

[167] https://discuss.d2l.ai/t/1403

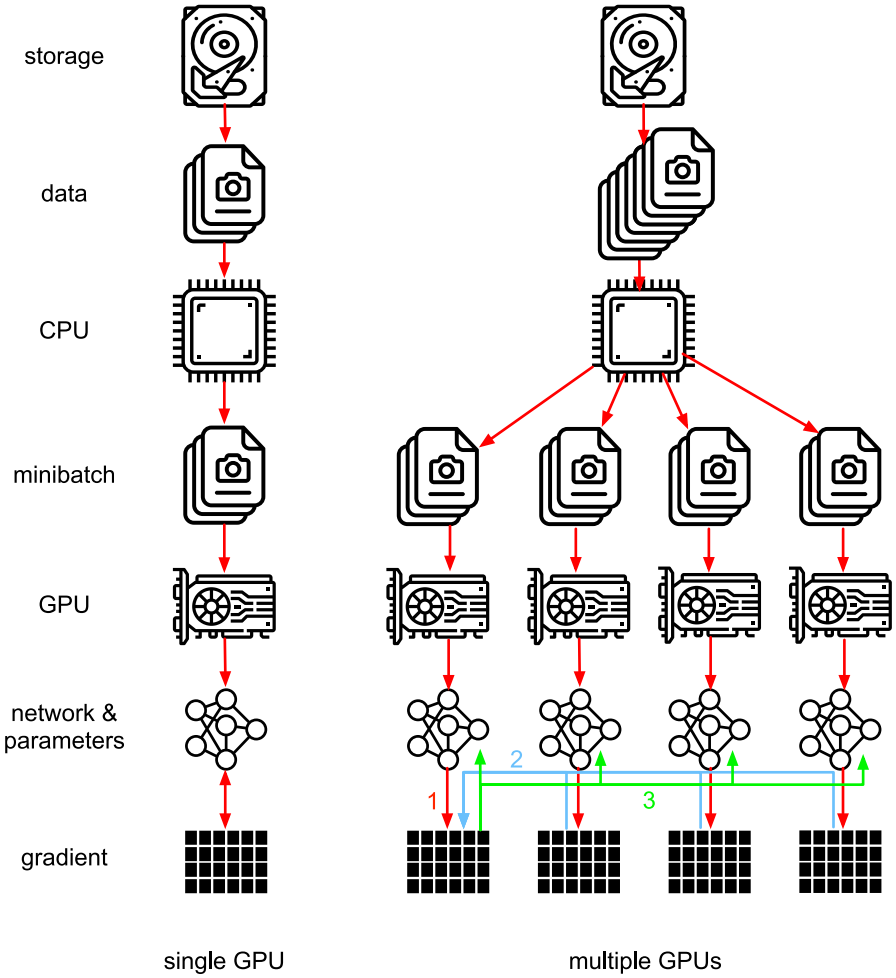


Fig. 13.7.1: Left: single GPU training. Right: a variant of multi-GPU training: (1) we compute loss and gradient, (2) all gradients are aggregated on one GPU, (3) parameter update happens and the parameters are re-distributed to all GPUs.

Nhìn lại, quyết định tổng hợp trên GPU 0 dường như khá ad-hoc. Sau khi tất cả, chúng ta có thể chỉ là cũng tổng hợp trên CPU. Trên thực tế, chúng tôi thậm chí có thể quyết định tổng hợp một số thông số trên một GPU và một số người khác trên một GPU khác. Với điều kiện là thuật toán tối ưu hóa hỗ trợ điều này, không có lý do thực sự cho lý do tại sao chúng ta không thể. Ví dụ, nếu chúng ta có bốn vectơ tham số với gradient liên quan $\mathbf{g}_1, \ldots, \mathbf{g}_4$, chúng ta có thể tổng hợp các gradient trên một GPU cho mỗi $\mathbf{g}_i$ ($i = 1, \ldots, 4$).

Lý luận này có vẻ tùy tiện và phù phiếm. Rốt cuộc, toán học là như nhau trong suốt. Tuy nhiên, chúng tôi đang đối phó với phần cứng vật lý thực sự, nơi các xe buýt khác nhau có băng thông khác nhau như đã thảo luận trong Section 13.4. Hãy xem xét một máy chủ GPU 4 chiều thực sự như được mô tả trong Fig. 13.7.2. Nếu nó được kết nối đặc biệt tốt, nó có thể có card mạng 100 Gbe. Các con số điển hình hơn nằm trong phạm vi 1—10 Gbe với băng thông hiệu quả từ 100 MB/giây đến 1 GB/giây vì các CPU có quá ít làn PCIe để kết nối trực tiếp với tất cả các GPU (ví dụ: CPU Intel cấp tiêu dùng có 24 làn), chúng ta cần multiplexer[168]. Băng thông từ CPU trên liên kết 16x Gen3 là 16 Gb/s, đây cũng là tốc độ mà * mỗi* của GPU được kết nối với công tắc. Điều này có nghĩa là nó hiệu quả hơn để giao tiếp giữa các thiết bị.

[168] https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches

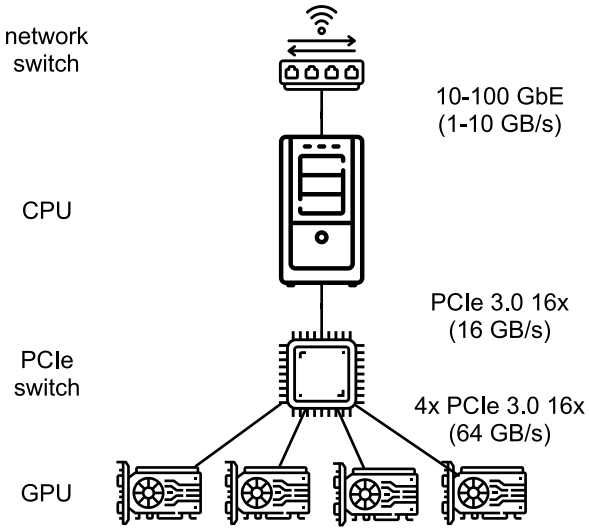


Fig. 13.7.2: A 4-way GPU server.

Vì lợi ích của đối số chúng ta hãy giả định rằng độ dốc là 160 MB. Trong trường hợp này phải mất 30 ms để gửi gradient từ tất cả 3 GPU còn lại đến lần thứ tư (mỗi lần chuyển mất 10 ms = 160 MB/ 16 Gb/s). Thêm 30 ms khác để truyền các vectơ trọng lượng trở lại, chúng tôi đến tổng cộng 60 ms. Nếu chúng tôi gửi tất cả dữ liệu đến CPU, chúng tôi phải chịu một hình phạt 40 ms kể từ * mỗi* của bốn GPU cần gửi dữ liệu đến CPU, mang lại tổng cộng 80 ms. Cuối cùng giả định rằng chúng ta có thể chia gradient thành 4 phần 40 MB mỗi phần. Bây giờ chúng ta có thể tổng hợp từng phần trên một GPU khác nhau * đồng thời* vì công tắc PCIe cung cấp hoạt động toàn băng thông giữa tất cả các liên kết. Thay vì 30 ms, điều này mất 7,5 ms, mang lại tổng cộng 15 ms cho một hoạt động đồng bộ hóa. Nói tóm lại, tùy thuộc vào cách chúng ta đồng bộ hóa các thông số, hoạt động tương tự có thể mất từ 15 ms đến 80 ms. Fig. 13.7.3 mô tả các chiến lược khác nhau để trao đổi thông số.

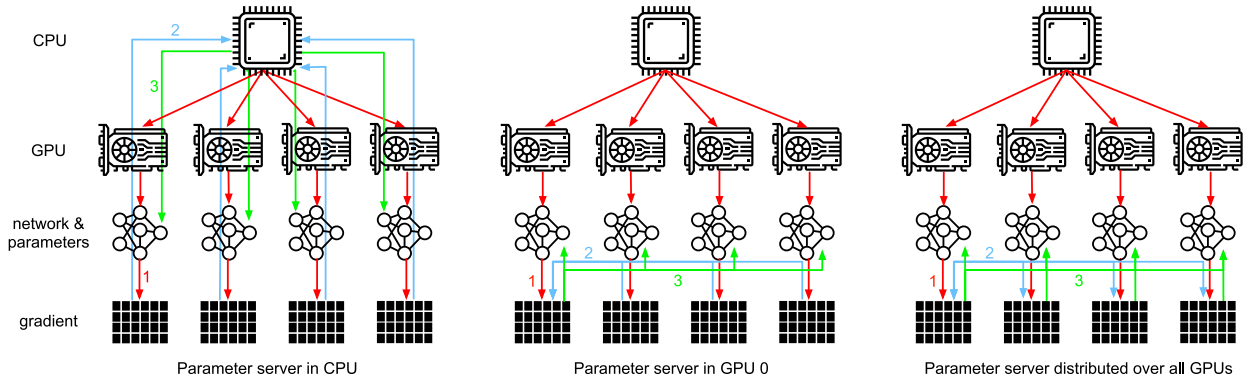


Fig. 13.7.3: Parameter synchronization strategies.

Lưu ý rằng chúng tôi có một công cụ khác theo ý của chúng tôi khi cải thiện hiệu suất: in a deep network it takes some time to compute all gradients from the top to the bottom. We can begin synchronizing gradients for some parameter groups even while we are still busy computing them for others. See e.g., (Sergeev & DelBalso, 2018) để biết chi tiết về cách thực hiện việc này trong Horovod[169].

[169] https://github.com/horovod/horovod

13.7.2 Đồng bộ hóa vòng

Khi nói đến đồng bộ hóa trên phần cứng học sâu hiện đại, chúng ta thường gặp phải kết nối mạng bespoke đáng kể. Ví dụ, các phiên bản AWS p3.16xlarge và NVIDIA DGX-2 chia sẻ cấu trúc kết nối của Fig. 13.7.4. Mỗi GPU kết nối với một CPU chủ thông qua một liên kết PCIe hoạt động tốt nhất với tốc độ 16 Gb/s Ngoài ra mỗi GPU cũng có 6 kết nối NVLink, mỗi trong số đó có khả năng truyền 300 Gbit/s hai chiều. Số này lên tới khoảng 18 Gb/s cho mỗi liên kết trên mỗi hướng. Tóm lại, băng thông NVLink tổng hợp cao hơn đáng kể so với băng thông PCIe. Câu hỏi là làm thế nào để sử dụng nó hiệu quả nhất.

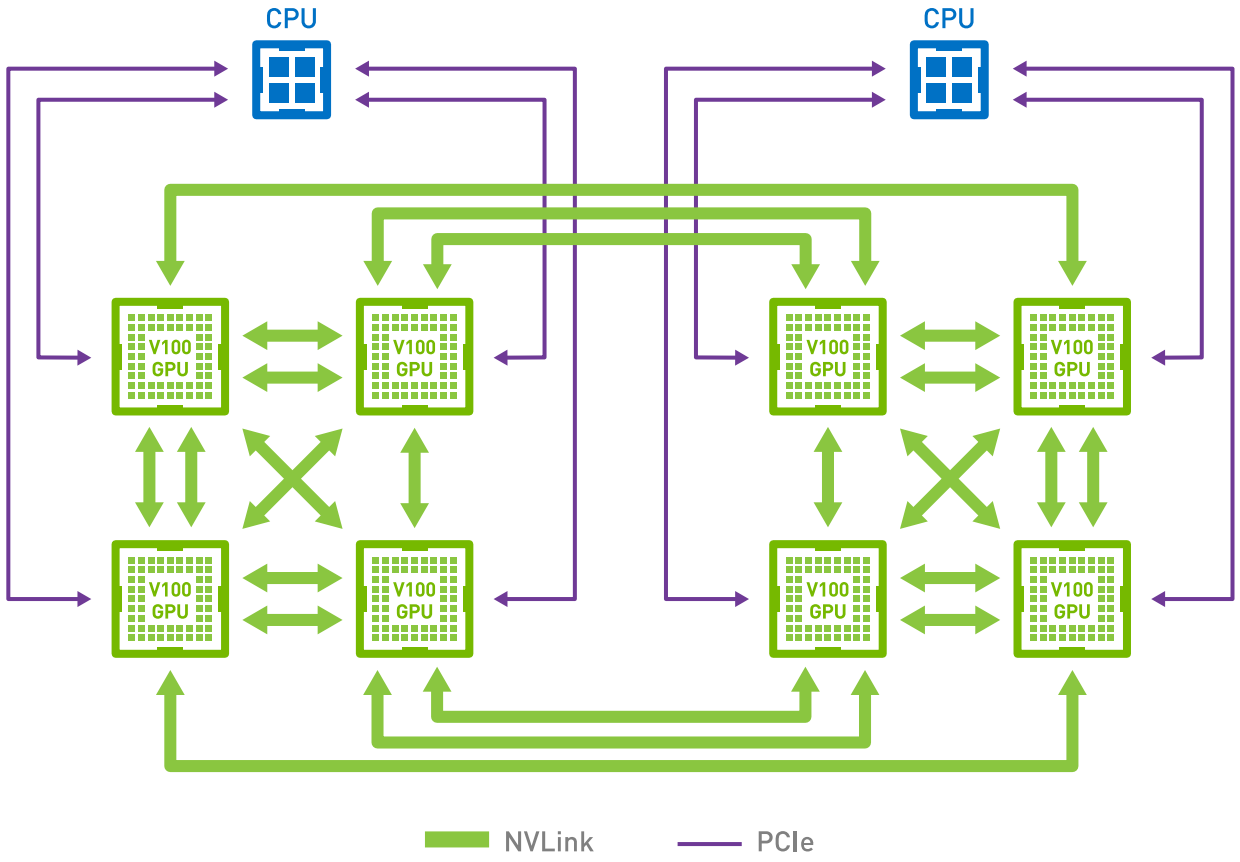


Fig. 13.7.4: NVLink connectivity on 8 V100 GPU servers (image courtesy of NVIDIA).

Nó chỉ ra rằng chiến lược đồng bộ hóa tối ưu là phân hủy mạng thành hai vòng và sử dụng chúng để đồng bộ hóa dữ liệu trực tiếp (Wang et al., 2018). Fig. 13.7.5 minh họa rằng mạng có thể bị phân hủy thành một vòng (1-2-3-4-5-6-7-8-1) với băng thông NVLink đôi và thành một (1-4-6-3-5-8-2-7-1) với băng thông thường. Thiết kế một giao thức đồng bộ hóa hiệu quả trong trường hợp này là không tầm thường.

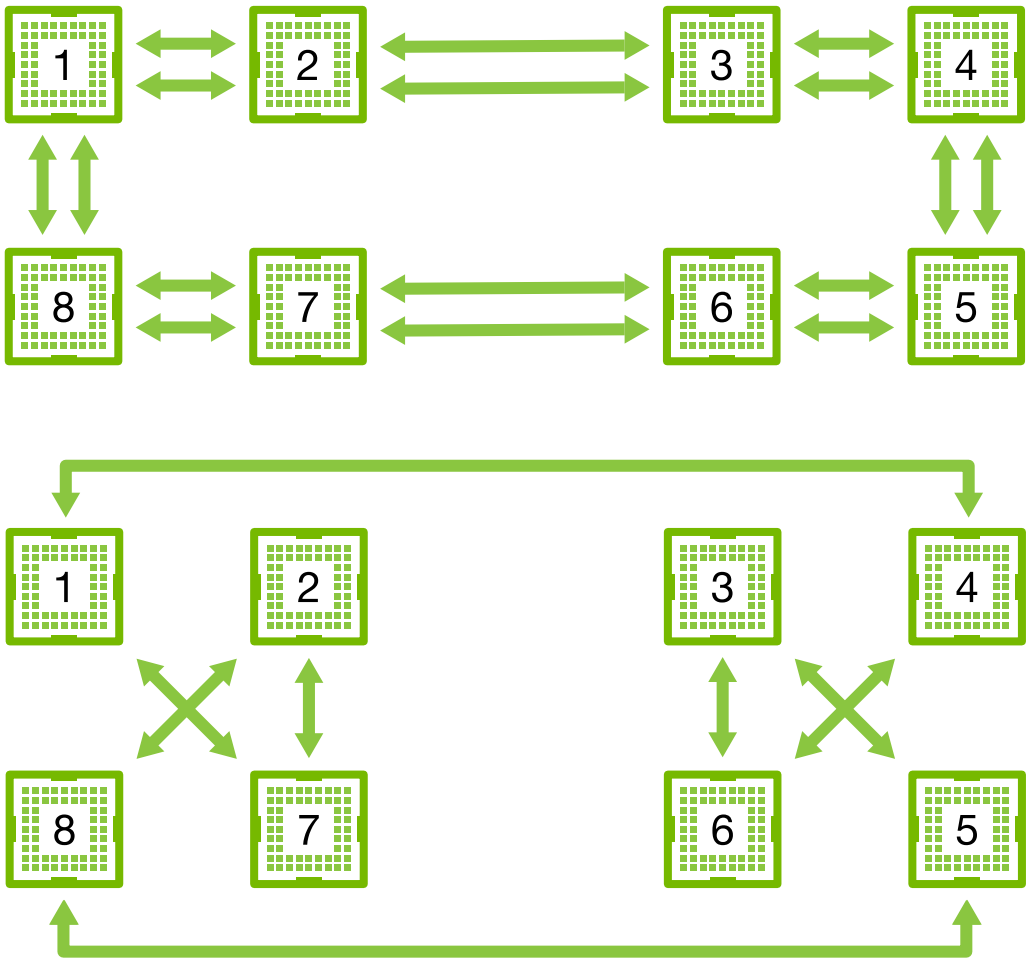


Fig. 13.7.5: Decomposition of the NVLink network into two rings.

Hãy xem xét thí nghiệm suy nghĩ sau: cho một vòng n nút tính toán (hoặc GPU) chúng ta có thể gửi gradient từ nút đầu tiên đến nút thứ hai. Ở đó nó được thêm vào gradient cục bộ và được gửi đến nút thứ ba, v.v. Sau $n-1$ bước, gradient tổng hợp có thể được tìm thấy trong nút truy cập cuối cùng. Đó là, thời gian để tổng hợp gradient phát triển tuyến tính với số lượng nút. Nhưng nếu chúng ta làm điều này thuật toán khá kém hiệu quả. Rốt cuộc, bất cứ lúc nào chỉ có một trong các nút giao tiếp. Điều gì sẽ xảy ra nếu chúng ta phá vỡ gradient thành n khối và bắt đầu đồng bộ hóa đoạn i bắt đầu từ nút i? Vì mỗi đoạn có kích thước $1/n$ tổng thời gian bây giờ là $(n-1)/n \approx 1$. Nói cách khác, thời gian dành để tổng hợp gradient * không tăng trưởng* khi chúng ta tăng kích thước của vòng. Đây là một kết quả khá đáng kinh ngạc. Fig. 13.7.6 minh họa trình tự các bước trên $n = 4$ nút.

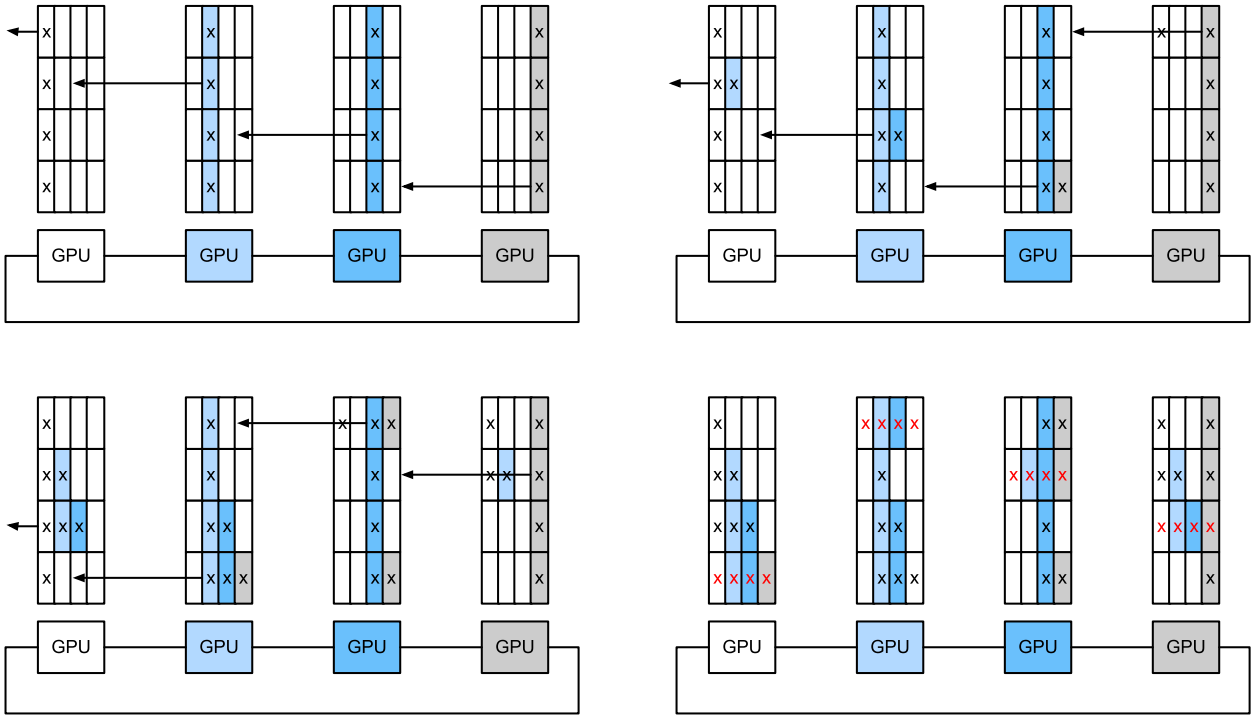


Fig. 13.7.6: Ring synchronization across 4 nodes. Each node starts transmitting parts of gradients to its left neighbor until the assembled gradient can be found in its right neighbor.

Nếu chúng ta sử dụng cùng một ví dụ về đồng bộ hóa 160 MB trên 8 GPU V100, chúng tôi sẽ đến khoảng $2 \cdot 160\text{MB}/(3 \cdot 18\text{GB}/\text{s}) \approx 6\text{ms}$. Điều này tốt hơn so với việc sử dụng bus PCIe, mặc dù chúng tôi hiện đang sử dụng 8 GPU. Lưu ý rằng trong thực tế, những con số này tồi tệ hơn một chút, vì các khuôn khổ học sâu thường không lắp ráp giao tiếp thành các chuyển vụ nổ lớn.

Lưu ý rằng có một quan niệm sai lầm phổ biến rằng đồng bộ hóa vòng về cơ bản khác với các thuật toán đồng bộ hóa khác. Sự khác biệt duy nhất là đường dẫn đồng bộ hóa có phần phức tạp hơn khi so sánh với một cây đơn giản.

13.7.3 Đào tạo đa máy

Đào tạo phân tán trên nhiều máy làm tăng thêm một thách thức: chúng ta cần giao tiếp với các máy chủ chỉ được kết nối trên một kết cấu băng thông tương đối thấp hơn có thể vượt quá mức độ chậm hơn trong một số trường hợp. Đồng bộ hóa trên các thiết bị là khó khăn. Rốt cuộc, các máy khác nhau chạy mã đào tạo sẽ có tốc độ khác nhau tinh tế. Do đó chúng ta cần phải * đồng bộ* chúng nếu chúng ta muốn sử dụng tối ưu hóa phân phối đồng bộ. Fig. 13.7.7 minh họa cách đào tạo song song phân phối xảy ra.

1. Một lô dữ liệu (khác nhau) được đọc trên mỗi máy, chia thành nhiều GPU và chuyển sang bộ nhớ GPU. Có dự đoán và độ dốc được tính toán trên mỗi lô GPU riêng biệt.
2. Các gradient từ tất cả các GPU cục bộ được tổng hợp trên một GPU (hoặc các phần của nó được tổng hợp trên các GPU khác nhau).
3. Các gradient được gửi đến các CPU.
4. Các CPU gửi gradient đến một máy chủ tham số trung tâm mà tập hợp tất cả các gradient.
5. Các gradient tổng hợp sau đó được sử dụng để cập nhật các tham số và các tham số cập nhật được phát lại cho các CPU riêng lẻ.

6. Thông tin được gửi đến một (hoặc nhiều) GPU.
7. Các thông số cập nhật được trải rộng trên tất cả các GPU.

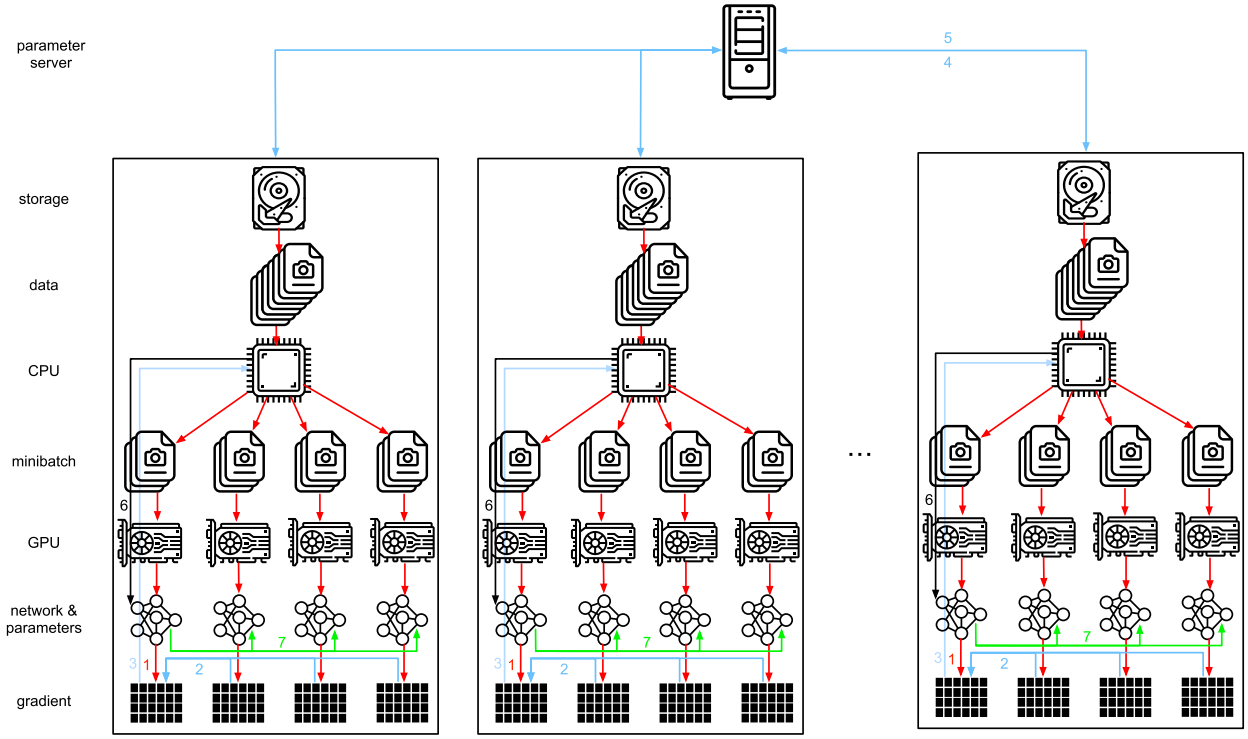


Fig. 13.7.7: Multi-machine multi-GPU distributed parallel training.

Mỗi hoạt động này có vẻ khá đơn giản. Và, thực sự, chúng có thể được thực hiện hiệu quả * trong vòng* một máy duy nhất. Tuy nhiên, khi chúng ta nhìn vào nhiều máy, chúng ta có thể thấy rằng máy chủ tham số trung tâm trở thành nút cổ chai. Rốt cuộc, băng thông trên mỗi máy chủ bị hạn chế, do đó đối với m công nhân thời gian cần thiết để gửi tất cả gradient đến máy chủ là $\mathcal{O}(m)$. Chúng ta có thể vượt qua rào cản này bằng cách tăng số lượng máy chủ lên n. Tại thời điểm này mỗi máy chủ chỉ cần lưu trữ $\mathcal{O}(1/n)$ các tham số, do đó tổng thời gian cập nhật và tối ưu hóa trở thành $\mathcal{O}(m/n)$. Phù hợp với cả hai con số đều mang lại tỷ lệ liên tục bất kể chúng tôi đang đối phó với bao nhiêu công nhân. Trong thực tế, chúng tôi sử dụng các máy * same* cả làm công nhân và làm máy chủ. Fig. 13.7.8 minh họa thiết kế (xem thêm (Li et al., 2014) để biết chi tiết). Đặc biệt, đảm bảo rằng nhiều máy hoạt động mà không có sự chậm trễ không hợp lý là không cần thiết. Chúng tôi bỏ qua chi tiết về các rào cản và sẽ chỉ chạm ngắn gọn vào các bản cập nhật đồng bộ và không đồng bộ bên dưới.

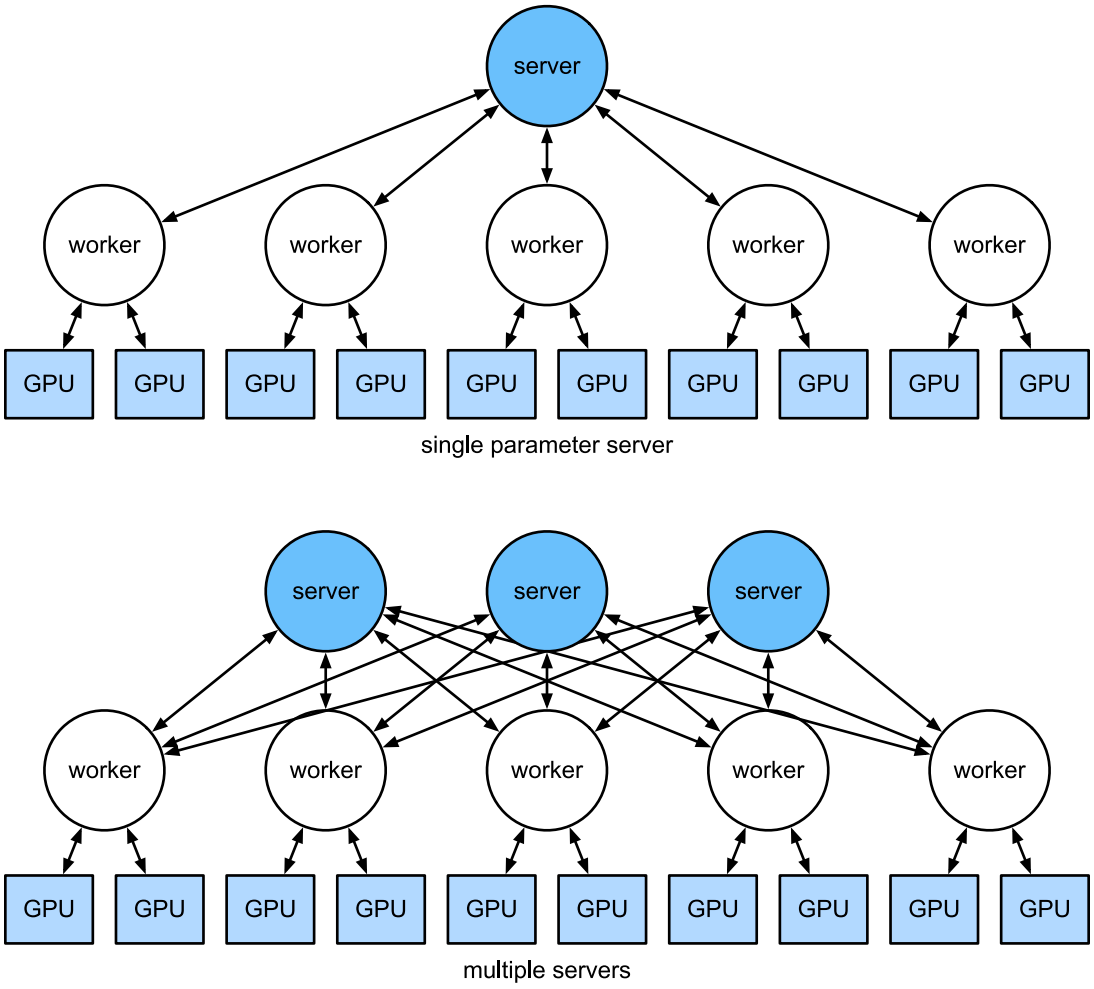


Fig. 13.7.8: Top: a single parameter server is a bottleneck since its bandwidth is finite. Bottom: multiple parameter servers store parts of the parameters with aggregate bandwidth.

13.7.4 Cửa hàng Key — Value

Thực hiện các bước cần thiết cho đào tạo đa GPU phân tán trong thực tế là không tầm thường. Đây là lý do tại sao nó trả tiền để sử dụng một trừu tượng phổ biến, cụ thể là của kho * key — value store* với ngữ nghĩa cập nhật được xác định lại.

Trên nhiều công nhân và nhiều GPU, tính toán cho gradient i có thể được định nghĩa là

$$\mathbf{g}_i = \sum_{k \in \text{workers}} \sum_{j \in \text{GPUs}} \mathbf{g}_{ijk}, \tag{13.7.1}$$

trong đó $\mathbf{g}_{ijk}$ là một phần của gradient i chia trên GPU j của công nhân k. Khía cạnh chính trong hoạt động này là nó là giảm giao hoán*, nghĩa là, nó biến nhiều vectơ thành một và thứ tự mà hoạt động được áp dụng không quan trọng. Điều này rất tốt cho mục đích của chúng tôi vì chúng tôi không (cần) có quyền kiểm soát hạt mịn khi nhận được gradient nào. Bên cạnh đó, lưu ý rằng hoạt động này độc lập giữa i khác nhau.

Điều này cho phép chúng ta xác định hai phép toán sau: * push*, tích lũy gradient và * pull*, lấy gradient tổng hợp. Vì chúng ta có nhiều bộ gradient khác nhau (sau tất cả, chúng ta có nhiều lớp), chúng ta cần lập chỉ mục gradient với một khóa i. Sự tương đồng này với các kho lưu trữ giá trị khóa, chẳng hạn như cửa hàng được giới thiệu trong Dynamo (DeCandia et al., 2007) không phải do trùng hợp ngẫu nhiên. Chúng cũng đáp ứng nhiều đặc điểm tương tự, đặc biệt là khi phân phối các tham số trên nhiều máy chủ.

Các thao tác đẩy và kéo cho các cửa hàng khóa-giá trị được mô tả như sau:

- **push (key, value) ** gửi một gradient cụ thể (giá trị) từ một công nhân đến một bộ nhớ chung. Có giá trị được tổng hợp, ví dụ, bằng cách tổng hợp nó lên.
- **pull (key, value) ** lấy một giá trị tổng hợp từ lưu trữ chung, ví dụ, sau khi kết hợp gradient từ tất cả các công nhân.

Bằng cách ẩn tất cả sự phức tạp về đồng bộ hóa đằng sau một thao tác đẩy và kéo đơn giản, chúng ta có thể tách các mối quan tâm của các nhà mô hình thống kê, những người muốn có thể thể hiện tối ưu hóa một cách đơn giản và các kỹ sư hệ thống cần đối phó với sự phức tạp vốn có trong đồng bộ hóa phân tán.

13.7.5 Tóm tắt

- Đồng bộ hóa cần phải thích ứng cao với cơ sở hạ tầng mạng cụ thể và kết nối trong một máy chủ. Điều này có thể tạo ra sự khác biệt đáng kể so với thời gian cần thiết để đồng bộ hóa.
- Đồng bộ hóa vòng có thể tối ưu cho các máy chủ p3 và DGX-2. Đối với những người khác có thể không quá nhiều.
- Chiến lược đồng bộ hóa phân cấp hoạt động tốt khi thêm nhiều máy chủ tham số để tăng băng thông.

13.7.6 Bài tập

1. Bạn có thể tăng đồng bộ hóa vòng hơn nữa không? Gợi ý: bạn có thể gửi tin nhắn theo cả hai hướng.
2. Có thể cho phép giao tiếp không đồng bộ (trong khi tính toán vẫn đang diễn ra)? Làm thế nào để nó ảnh hưởng đến hiệu suất?
3. Điều gì sẽ xảy ra nếu chúng ta mất một máy chủ trong một tính toán chạy dài? Làm thế nào chúng ta có thể thiết kế một cơ chế dung sai lỗi* để tránh khởi động lại tính toán đầy đủ?

Discussions[170]

[170] https://discuss.d2l.ai/t/366

14 | Tầm nhìn máy tính

Cho dù đó là chẩn đoán y tế, xe tự lái, giám sát camera hoặc bộ lọc thông minh, nhiều ứng dụng trong lĩnh vực thị giác máy tính có liên quan chặt chẽ đến cuộc sống hiện tại và tương lai của chúng ta. Trong những năm gần đây, deep learning là sức mạnh biến đổi để thúc đẩy hiệu suất của các hệ thống thị giác máy tính. Có thể nói rằng các ứng dụng tầm nhìn máy tính tiên tiến nhất gần như không thể tách rời khỏi học sâu. Theo quan điểm này, chương này sẽ tập trung vào lĩnh vực thị giác máy tính, và điều tra các phương pháp và ứng dụng gần đây đã có ảnh hưởng trong học viện và ngành công nghiệp.

Trong Chapter 7 và Chapter 8, chúng tôi đã nghiên cứu các mạng thần kinh phức tạp khác nhau thường được sử dụng trong tầm nhìn máy tính và áp dụng chúng cho các tác vụ phân loại hình ảnh đơn giản. Vào đầu chương này, chúng tôi sẽ mô tả hai phương pháp có thể cải thiện khái quát hóa mô hình, đó là *image augmentation* và *fine-tuning*, và áp dụng chúng vào phân loại hình ảnh. Vì các mạng thần kinh sâu có thể đại diện hiệu quả hình ảnh ở nhiều cấp độ, nên các biểu diễn theo lớp như vậy đã được sử dụng thành công trong các tác vụ thị giác máy tính khác nhau như phát hiện đối tượng**, * phân đoạn ngữ nghị* và chuyển kiểu **. Theo ý tưởng chính của việc tận dụng các biểu diễn theo lớp trong tầm nhìn máy tính, chúng ta sẽ bắt đầu với các thành phần và kỹ thuật chính để phát hiện đối tượng. Tiếp theo, chúng tôi sẽ chỉ ra cách sử dụng mạng * hoàn toàn phức tạp * để phân đoạn hình ảnh ngữ nghĩa. Sau đó, chúng tôi sẽ giải thích cách sử dụng các kỹ thuật chuyển phong cách để tạo ra hình ảnh như bìa của cuốn sách này. Cuối cùng, chúng tôi kết thúc chương này bằng cách áp dụng các tài liệu của chương này và một số chương trước trên hai bộ dữ liệu điểm chuẩn tầm nhìn máy tính phổ biến.

14.1 Hình ảnh Augmentation

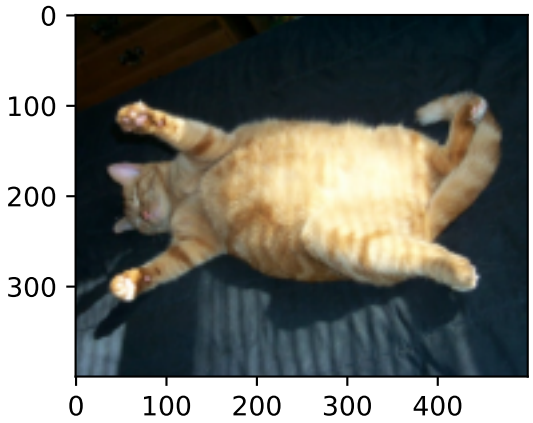
Trong Section 8.1, chúng tôi đã đề cập rằng các tập dữ liệu lớn là điều kiện tiên quyết cho sự thành công của các mạng thần kinh sâu trong các ứng dụng khác nhau. *Tăng cường hình ảnh* tạo ra các ví dụ đào tạo tương tự nhưng khác biệt sau một loạt các thay đổi ngẫu nhiên đối với hình ảnh đào tạo, từ đó mở rộng quy mô của bộ đào tạo. Ngoài ra, tăng cường hình ảnh có thể được thúc đẩy bởi thực tế là các tinh chỉnh ngẫu nhiên của các ví dụ đào tạo cho phép các mô hình ít dựa vào các thuộc tính nhất định, do đó cải thiện khả năng tổng quát của chúng. Ví dụ, chúng ta có thể cắt một hình ảnh theo những cách khác nhau để làm cho đối tượng quan tâm xuất hiện ở các vị trí khác nhau, do đó làm giảm sự phụ thuộc của một mô hình vào vị trí của đối tượng. Chúng tôi cũng có thể điều chỉnh các yếu tố như độ sáng và màu sắc để giảm độ nhạy của mô hình với màu sắc. Có lẽ đúng là nâng hình ảnh là không thể thiếu cho sự thành công của AlexNet vào thời điểm đó. Trong phần này, chúng tôi sẽ thảo luận về kỹ thuật được sử dụng rộng rãi này trong tầm nhìn máy tính.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

14.1.1 Phương pháp Augmentation hình ảnh phổ biến

Trong cuộc điều tra của chúng tôi về các phương pháp nâng hình ảnh phổ biến, chúng tôi sẽ sử dụng hình ảnh 400×500 sau đây một ví dụ.

```
d2l.set_figsize()
img = d2l.Image.open('../img/cat1.jpg')
d2l.plt.imshow(img);
```



Hầu hết các phương pháp nâng hình ảnh đều có một mức độ ngẫu nhiên nhất định. Để giúp chúng ta dễ dàng quan sát hiệu ứng của việc nâng hình ảnh hơn, tiếp theo chúng ta xác định một hàm phụ `apply`. Chức năng này chạy phương pháp augmentation hình ảnh `aug` nhiều lần trên hình ảnh đầu vào `img` và hiển thị tất cả các kết quả.

```
def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
    Y = [aug(img) for _ in range(num_rows * num_cols)]
    d2l.show_images(Y, num_rows, num_cols, scale=scale)
```

Lật và cắt xén

Lật lại hình ảnh bên trái và phải thường không thay đổi danh mục của đối tượng. Đây là một trong những phương pháp nâng hình ảnh sớm nhất và được sử dụng rộng rãi nhất. Tiếp theo, chúng ta sử dụng mô-đun `transforms` để tạo phiên bản `RandomHorizontalFlip`, nó lật một hình ảnh sang trái và phải với cơ hội 50%.

```
apply(img, torchvision.transforms.RandomHorizontalFlip())
```

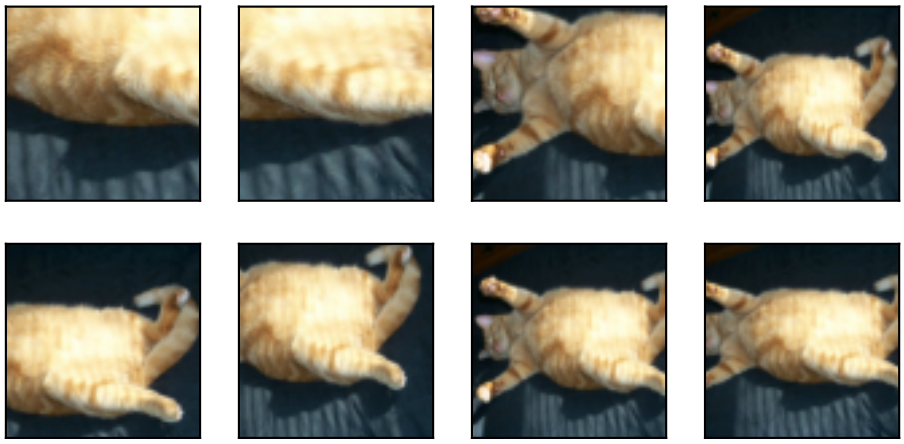
Flipping lên và xuống không phổ biến như lật trái và phải. Nhưng ít nhất đối với hình ảnh ví dụ này, lật lên xuống không cản trở sự nhận dạng. Tiếp theo, chúng ta tạo một phiên bản `RandomVerticalFlip` để lật ảnh lên xuống với 50% cơ hội.

```
apply(img, torchvision.transforms.RandomVerticalFlip())
```

Trong hình ảnh ví dụ chúng tôi đã sử dụng, con mèo nằm ở giữa hình ảnh, nhưng điều này có thể không phải là trường hợp nói chung. Trong Section 7.5, chúng tôi giải thích rằng lớp tổng hợp có thể làm giảm độ nhạy của một lớp phức tạp đến vị trí mục tiêu. Ngoài ra, chúng ta cũng có thể cắt ngẫu nhiên hình ảnh để làm cho các đối tượng xuất hiện ở các vị trí khác nhau trong ảnh ở các thang đo khác nhau, điều này cũng có thể làm giảm độ nhạy của một mô hình đến vị trí mục tiêu.

Trong mã dưới đây, chúng ta ngẫu nhiên crop một khu vực có diện tích $10\% \sim 100\%$ of the original area each time, and the ratio of width to height of this area is randomly selected from $0.5 \sim 2$. Sau đó, chiều rộng và chiều cao của vùng đều được thu nhỏ thành 200 pixel. Trừ khi có quy định khác, số ngẫu nhiên giữa a và b trong phần này đề cập đến một giá trị liên tục thu được bằng cách lấy mẫu ngẫu nhiên và thống nhất từ khoảng $[a, b]$.

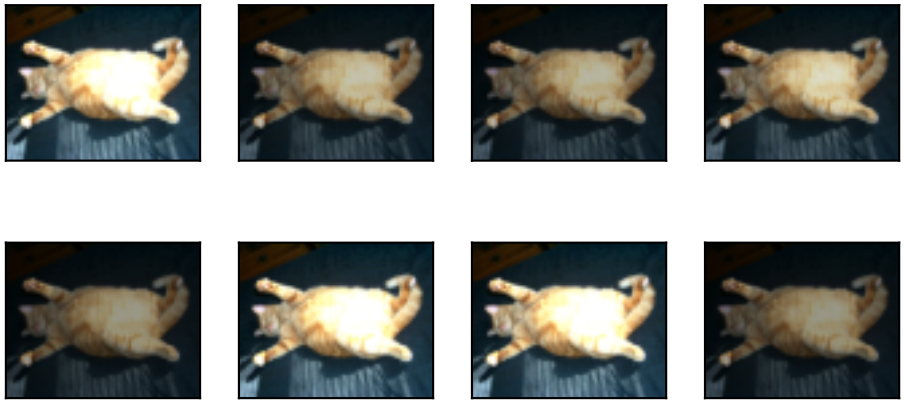
```
shape_aug = torchvision.transforms.RandomResizedCrop(
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
apply(img, shape_aug)
```



Thay đổi màu sắc

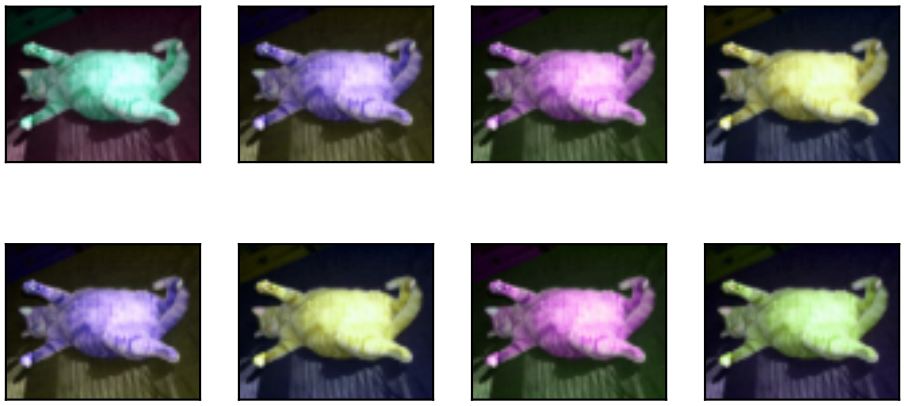
Một phương pháp nâng cao khác là thay đổi màu sắc. Chúng ta có thể thay đổi bốn khía cạnh của màu hình ảnh: độ sáng, độ tương phản, độ bão hòa và màu sắc. Trong ví dụ dưới đây, chúng ta thay đổi ngẫu nhiên độ sáng của hình ảnh thành giá trị giữa 50% $(1 - 0.5)$ và 150% $(1 + 0.5)$ của ảnh gốc.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0, saturation=0, hue=0))
```



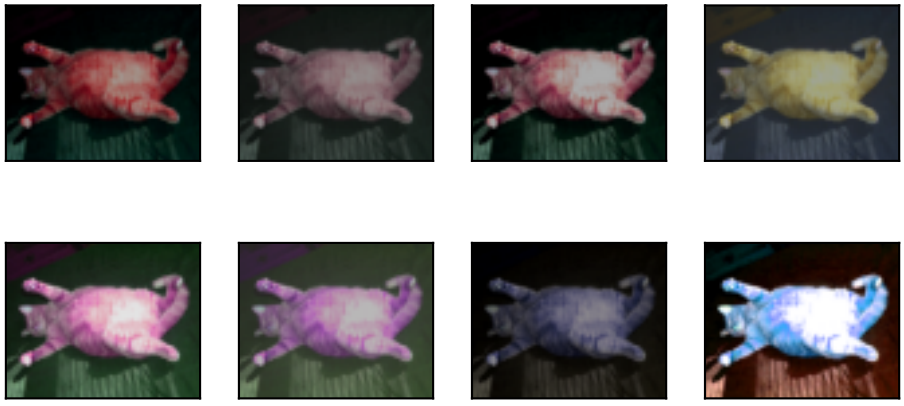
Tương tự, chúng ta có thể ngẫu nhiên thay đổi hue của hình ảnh.

```
apply(img, torchvision.transforms.ColorJitter(
    brightness=0, contrast=0, saturation=0, hue=0.5))
```



Chúng ta cũng có thể tạo một phiên bản `RandomColorJitter` và đặt cách thay đổi ngẫu nhiên `brightness`, `contrast`, `saturation` và `hue` của ảnh cùng một lúc.

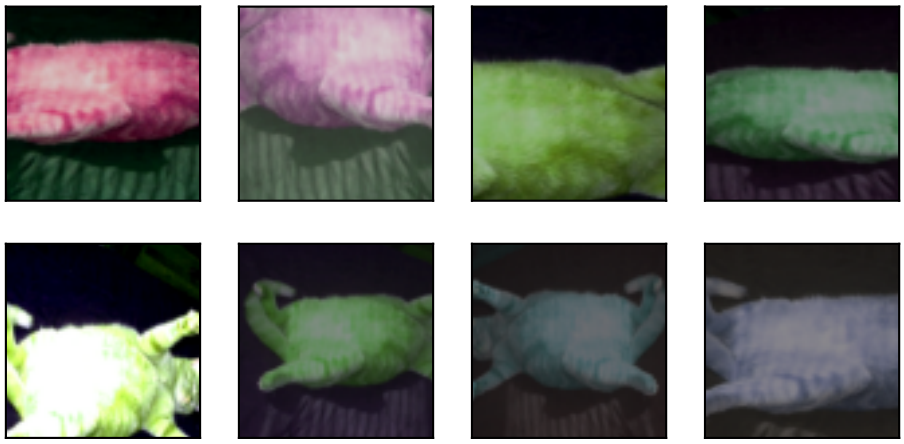
```
color_aug = torchvision.transforms.ColorJitter(
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
apply(img, color_aug)
```



Kết hợp nhiều phương pháp Augmentation ảnh

Trong thực tế, chúng ta sẽ kết hợp nhiều phương pháp tăng cường hình ảnh. Ví dụ: chúng ta có thể kết hợp các phương thức augmentation hình ảnh khác nhau được xác định ở trên và áp dụng chúng cho mỗi hình ảnh thông qua một phiên bản `Compose`.

```
augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomHorizontalFlip(), color_aug, shape_aug])
apply(img, augs)
```



14.1.2 Đào tạo với hình ảnh Augmentation

Hãy để chúng tôi đào tạo một mô hình với nâng hình ảnh. Ở đây chúng tôi sử dụng bộ dữ liệu CIFAR-10 thay vì tập dữ liệu Fashion-MNIST mà chúng tôi đã sử dụng trước đây. Điều này là do vị trí và kích thước của các đối tượng trong bộ dữ liệu Fashion-MNIST đã được chuẩn hóa, trong khi màu sắc và kích thước của các đối tượng trong tập dữ liệu CIFAR-10 có sự khác biệt đáng kể hơn. 32 hình ảnh đào tạo đầu tiên trong tập dữ liệu CIFAR-10 được hiển thị bên dưới.

```
all_images = torchvision.datasets.CIFAR10(train=True, root="../data",
                                          download=True)
d2l.show_images([all_images[i][0] for i in range(32)], 4, 8, scale=0.8);
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ../
↪data/cifar-10-python.tar.gz
97.4%
```

Để có được kết quả dứt khoát trong quá trình dự đoán, chúng ta thường chỉ áp dụng nâng hình ảnh cho các ví dụ đào tạo và không sử dụng nâng hình ảnh với các phép toán ngẫu nhiên trong quá trình dự đoán. Ở đây chúng tôi chỉ sử dụng phương pháp lật trái phải ngẫu nhiên đơn giản nhất. Ngoài ra, chúng ta sử dụng một phiên bản `ToTensor` để chuyển đổi một minibatch ảnh thành định dạng theo yêu cầu của khung học sâu, tức là các số điểm nổi 32 bit giữa 0 đến 1 với hình dạng (kích thước lô, số kênh, chiều cao, chiều rộng).

```
train_augs = torchvision.transforms.Compose([
     torchvision.transforms.RandomHorizontalFlip(),
     torchvision.transforms.ToTensor()])

test_augs = torchvision.transforms.Compose([
     torchvision.transforms.ToTensor()])
```

Tiếp theo, chúng ta xác định một hàm phụ trợ để tạo điều kiện đọc hình ảnh và áp dụng tăng cường hình ảnh. Đối số `transform` được cung cấp bởi tập dữ liệu của PyTorch áp dụng tăng cường để biến đổi hình ảnh. Để biết giới thiệu chi tiết về `DataLoader`, vui lòng tham khảo Section 4.5.

```
def load_cifar10(is_train, augs, batch_size):
    dataset = torchvision.datasets.CIFAR10(root="../data", train=is_train,
                                           transform=augs, download=True)
```

(continues on next page)

(continued from previous page)

```
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                    shuffle=is_train, num_workers=d2l.get_dataloader_
↪workers())
    return dataloader
```

Đào tạo Multi-GPU

Chúng tôi đào tạo mô hình ResNet-18 từ Section 8.6 trên bộ dữ liệu CIFAR-10. Nhớ lại phần giới thiệu về đào tạo đa GPU trong Section 13.6. Sau đây, chúng tôi định nghĩa một hàm để đào tạo và đánh giá mô hình bằng cách sử dụng nhiều GPU .

```
#@save
def train_batch_ch13(net, X, y, loss, trainer, devices):
    """Train for a minibatch with mutiple GPUs (defined in Chapter 13)."""
    if isinstance(X, list):
        # Required for BERT fine-tuning (to be covered later)
        X = [x.to(devices[0]) for x in X]
    else:
        X = X.to(devices[0])
    y = y.to(devices[0])
    net.train()
    trainer.zero_grad()
    pred = net(X)
    l = loss(pred, y)
    l.sum().backward()
    trainer.step()
    train_loss_sum = l.sum()
    train_acc_sum = d2l.accuracy(pred, y)
    return train_loss_sum, train_acc_sum
```

```
#@save
def train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
               devices=d2l.try_all_gpus()):
    """Train a model with mutiple GPUs (defined in Chapter 13)."""
    timer, num_batches = d2l.Timer(), len(train_iter)
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['train loss', 'train acc', 'test acc'])
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples,
        # no. of predictions
        metric = d2l.Accumulator(4)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = train_batch_ch13(
                net, features, labels, loss, trainer, devices)
            metric.add(l, acc, labels.shape[0], labels.numel())
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[3],
                              None))
```

(continues on next page)

(continued from previous page)

```
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {metric[0] / metric[2]:.3f}, train acc '
          f'{metric[1] / metric[3]:.3f}, test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec on '
          f'{str(devices)}')
```

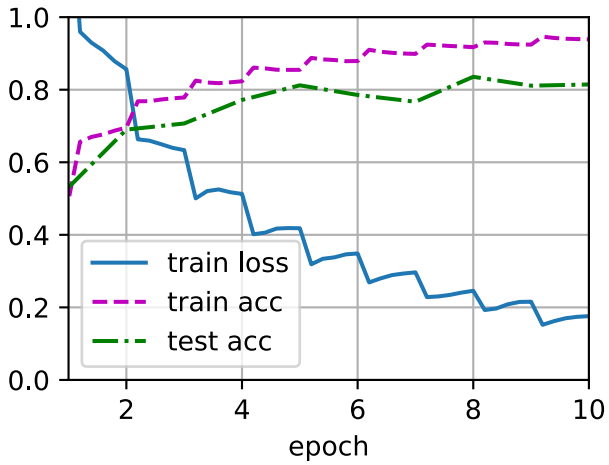
Bây giờ chúng ta có thể xác định hàm `train_with_data_aug` để đào tạo mô hình với khả năng tăng cường hình ảnh. Chức năng này nhận được tất cả các GPU có sẵn, sử dụng Adam làm thuật toán tối ưu hóa, áp dụng nâng hình ảnh cho tập dữ liệu đào tạo và cuối cùng gọi hàm `train_ch13` vừa được định nghĩa để đào tạo và đánh giá mô hình.

```
batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10, 3)

def init_weights(m):
    if type(m) in [nn.Linear, nn.Conv2d]:
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights)

def train_with_data_aug(train_augs, test_augs, net, lr=0.001):
    train_iter = load_cifar10(True, train_augs, batch_size)
    test_iter = load_cifar10(False, test_augs, batch_size)
    loss = nn.CrossEntropyLoss(reduction="none")
    trainer = torch.optim.Adam(net.parameters(), lr=lr)
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)
```

Hãy để chúng tôi đào tạo mô hình bằng cách sử dụng nâng hình ảnh dựa trên lật trái phải ngẫu nhiên.

```
train_with_data_aug(train_augs, test_augs, net)
```

```
loss 0.176, train acc 0.939, test acc 0.814
5458.4 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



14.1.3 Tóm tắt

- Nâng hình ảnh tạo ra hình ảnh ngẫu nhiên dựa trên dữ liệu đào tạo hiện có để cải thiện khả năng tổng quát hóa của các mô hình.
- Để có được kết quả dứt khoát trong quá trình dự đoán, chúng ta thường chỉ áp dụng nâng hình ảnh cho các ví dụ đào tạo và không sử dụng nâng hình ảnh với các phép toán ngẫu nhiên trong quá trình dự đoán.
- Khung học sâu cung cấp nhiều phương pháp nâng hình ảnh khác nhau, có thể được áp dụng đồng thời.

14.1.4 Bài tập

1. Đào tạo mô hình mà không cần sử dụng nâng hình ảnh: `train_with_data_aug(test_augs, test_augs)`. So sánh độ chính xác của đào tạo và kiểm tra khi sử dụng và không sử dụng nâng hình ảnh. Thí nghiệm so sánh này có thể hỗ trợ lập luận rằng việc tăng cường hình ảnh có thể giảm thiểu quá mức không? Tại sao?
2. Kết hợp nhiều phương pháp nâng hình ảnh khác nhau trong đào tạo mô hình trên tập dữ liệu CIFAR-10. Nó có cải thiện độ chính xác của thử nghiệm không?
3. Tham khảo tài liệu trực tuyến của khung học sâu. Những phương pháp nâng hình ảnh nào khác mà nó cũng cung cấp?

Discussions[171]

14.2 Tinh chỉnh

Trong các chương trước đó, chúng tôi đã thảo luận về cách đào tạo các mô hình trên tập dữ liệu đào tạo Fashion-MNIST chỉ với 60000 hình ảnh. Chúng tôi cũng mô tả ImageNet, tập dữ liệu hình ảnh quy mô lớn được sử dụng rộng rãi nhất trong học viện, có hơn 10 triệu hình ảnh và 1000 đối tượng. Tuy nhiên, kích thước của tập dữ liệu mà chúng ta thường gặp phải là giữa các tập dữ liệu của hai tập dữ liệu.

Giả sử rằng chúng tôi muốn nhận ra các loại ghế khác nhau từ hình ảnh, và sau đó khuyên bạn nên mua liên kết cho người dùng. Một phương pháp có thể là đầu tiên xác định 100 ghế thông thường, chụp 1000 hình ảnh của các góc khác nhau cho mỗi ghế, và sau đó đào tạo một mô hình phân loại trên tập dữ liệu hình ảnh thu thập được. Mặc dù bộ dữ liệu ghế này có thể lớn hơn tập dữ liệu Fashion-MNIST, số lượng ví dụ vẫn ít hơn một phần mười trong ImageNet. Điều này có thể dẫn đến quá mức các mô hình phức tạp phù hợp với ImageNet trên bộ dữ liệu ghế này. Bên cạnh đó, do số lượng ví dụ đào tạo hạn chế, độ chính xác của mô hình được đào tạo có thể không đáp ứng các yêu cầu thực tế.

Để giải quyết các vấn đề trên, một giải pháp rõ ràng là thu thập thêm dữ liệu. Tuy nhiên, việc thu thập và ghi nhãn dữ liệu có thể mất rất nhiều thời gian và tiền bạc. Ví dụ, để thu thập tập dữ liệu ImageNet, các nhà nghiên cứu đã chi hàng triệu đô la từ tài trợ nghiên cứu. Mặc dù chi phí thu thập dữ liệu hiện tại đã giảm đáng kể, chi phí này vẫn không thể bỏ qua.

Một giải pháp khác là áp dụng *transfer learning* để chuyển các kiến thức học được từ *sourcedataset* sang tập dữ liệu đích *. Ví dụ, mặc dù hầu hết các hình ảnh trong tập dữ liệu ImageNet không liên quan gì đến ghế, mô hình được đào tạo trên bộ dữ liệu này có thể trích xuất các tính năng hình ảnh chung hơn, có thể giúp xác định các cạnh, kết cấu, hình dạng và bố cục đối tượng. Những tính năng tương tự này cũng có thể có hiệu quả để nhận ra ghế.

[171] https://discuss.d2l.ai/t/1404

14.2.1 Các bước

Trong phần này, chúng tôi sẽ giới thiệu một kỹ thuật phổ biến trong chuyển learning: *fine-tuning*. As shown in Fig. 14.2.1, tinh chỉnh bao gồm bốn bước sau:

1. Pretrain một mô hình mạng thần kinh, tức là mô hình *source*, trên một tập dữ liệu nguồn (ví dụ, tập dữ liệu ImageNet).
2. Tạo một mô hình mạng thần kinh mới, tức là mô hình mục tiêu *. Điều này sao chép tất cả các thiết kế mô hình và các tham số của chúng trên mô hình nguồn ngoại trừ lớp đầu ra. Chúng tôi giả định rằng các tham số mô hình này chứa kiến thức học được từ tập dữ liệu nguồn và kiến thức này cũng sẽ được áp dụng cho tập dữ liệu đích. Chúng tôi cũng giả định rằng lớp đầu ra của mô hình nguồn có liên quan chặt chẽ đến các nhãn của tập dữ liệu nguồn; do đó nó không được sử dụng trong mô hình đích.
3. Thêm một lớp đầu ra vào mô hình đích, có số lượng đầu ra là số lượng danh mục trong tập dữ liệu đích. Sau đó khởi tạo ngẫu nhiên các tham số mô hình của lớp này.
4. Đào tạo mô hình mục tiêu trên tập dữ liệu mục tiêu, chẳng hạn như bộ dữ liệu ghế. Lớp đầu ra sẽ được đào tạo từ đầu, trong khi các tham số của tất cả các lớp khác được tinh chỉnh dựa trên các thông số của mô hình nguồn.

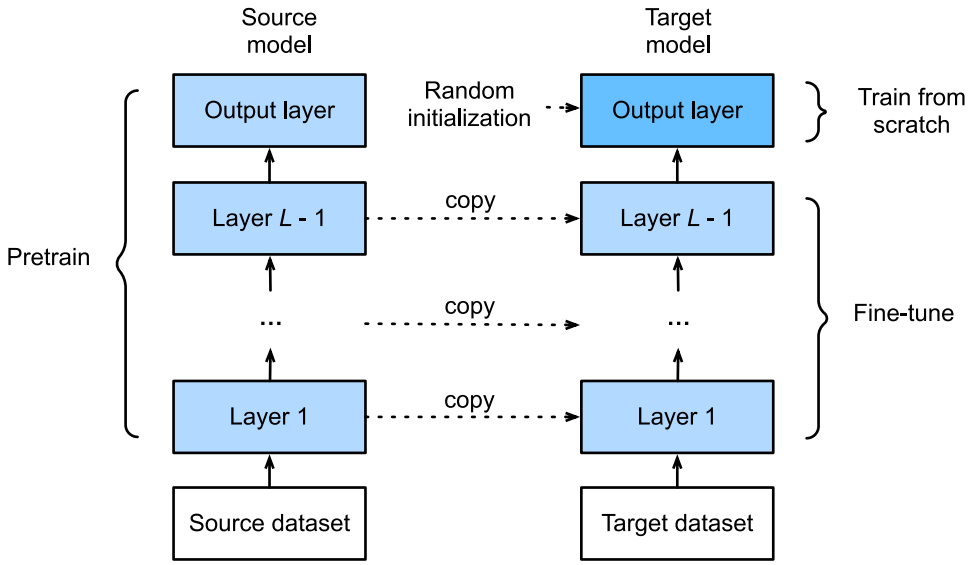


Fig. 14.2.1: Fine tuning.

Khi các bộ dữ liệu đích nhỏ hơn nhiều so với các bộ dữ liệu nguồn, tinh chỉnh giúp cải thiện khả năng tổng quát hóa của các mô hình.

14.2.2 Nhận dạng chó nóng

Hãy để chúng tôi chứng minh tinh chỉnh thông qua một trường hợp cụ thể: nhận dạng chó nóng. Chúng tôi sẽ tinh chỉnh một mô hình ResNet trên một tập dữ liệu nhỏ, được đào tạo trước trên tập dữ liệu ImageNet. Tập dữ liệu nhỏ này bao gồm hàng ngàn hình ảnh có và không có xúc xích. Chúng tôi sẽ sử dụng mô hình tinh chỉnh để nhận ra xúc xích từ hình ảnh.

```
%matplotlib inline
import os
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

Đọc tập dữ liệu

Bộ dữ liệu hot dog chúng tôi sử dụng được lấy từ hình ảnh trực tuyến. Tập dữ liệu này bao gồm 1400 hình ảnh đẳng cấp tích cực chứa xúc xích, và càng nhiều hình ảnh đẳng cấp tiêu cực chứa các loại thực phẩm khác. 1000 hình ảnh của cả hai lớp được sử dụng để đào tạo và phần còn lại là để thử nghiệm.

Sau khi giải nén tập dữ liệu đã tải xuống, chúng tôi nhận được hai thư mục `hotdog/train` và `hotdog/test`. Cả hai thư mục đều có các thư mục con `hotdog` và `not-hotdog`, trong đó có chứa hình ảnh của lớp tương ứng.

```
#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL + 'hotdog.zip',
                          'fba480ffa8aa7e0febbb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')
```

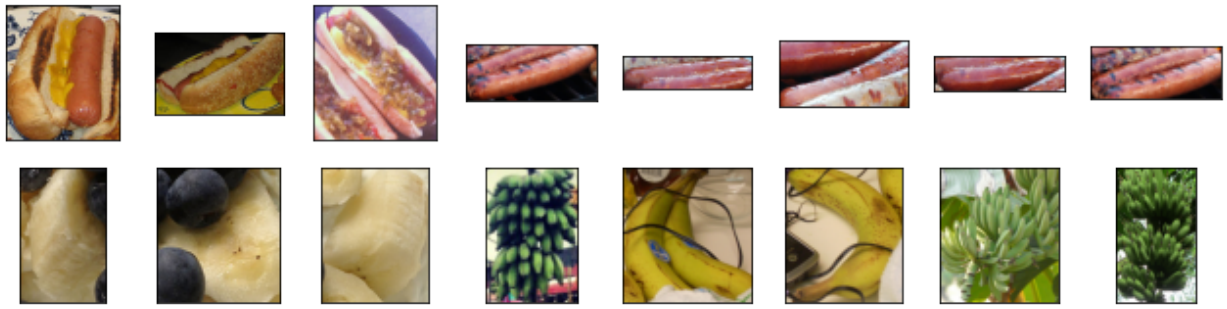
```
Downloading ../data/hotdog.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/hotdog.zip...
```

Chúng tôi tạo hai trường hợp để đọc tất cả các tệp hình ảnh trong các tập dữ liệu đào tạo và thử nghiệm, tương ứng.

```
train_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'train'))
test_imgs = torchvision.datasets.ImageFolder(os.path.join(data_dir, 'test'))
```

8 ví dụ tích cực đầu tiên và 8 hình ảnh tiêu cực cuối cùng được hiển thị bên dưới. Như bạn có thể thấy, hình ảnh khác nhau về kích thước và tỷ lệ khung hình.

```
hotdogs = [train_imgs[i][0] for i in range(8)]
not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```



Trong quá trình đào tạo, trước tiên chúng ta cắt một vùng ngẫu nhiên có kích thước ngẫu nhiên và tỷ lệ khung hình ngẫu nhiên từ hình ảnh, sau đó mở rộng khu vực này thành hình ảnh đầu vào 224×224. Trong quá trình thử nghiệm, chúng tôi chia tỷ lệ cả chiều cao và chiều rộng của hình ảnh lên 256 pixel, và sau đó cắt một khu vực trung tâm 224×224 làm đầu vào. Ngoài ra, đối với ba kênh màu RGB (đỏ, xanh lá cây và xanh dương), chúng tôi * tiêu chuẩn* giá trị của chúng theo kênh. Cụ thể, giá trị trung bình của một kênh được trừ đi từ mỗi giá trị của kênh đó và sau đó kết quả được chia cho độ lệch chuẩn của kênh đó.

```
# Specify the means and standard deviations of the three RGB channels to
# standardize each channel
normalize = torchvision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = torchvision.transforms.Compose([
    torchvision.transforms.RandomResizedCrop(224),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    normalize])

test_augs = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    normalize])
```

Xác định và khởi tạo mô hình

Chúng tôi sử dụng ResNet-18, được đào tạo trước trên tập dữ liệu ImageNet, làm mô hình nguồn. Ở đây, chúng tôi chỉ định `pretrained=True` để tự động tải xuống các tham số mô hình được đào tạo trước. Nếu mô hình này được sử dụng lần đầu tiên, cần có kết nối Internet để tải xuống.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
```

Ví dụ mô hình nguồn được đào tạo sẵn chứa một số lớp đối tượng và một lớp đầu ra `fc`. Mục đích chính của bộ phận này là tạo điều kiện thuận lợi cho việc tinh chỉnh các tham số mô hình của tất cả các lớp nhưng lớp đầu ra. Biến thành viên `fc` của mô hình nguồn được đưa ra dưới đây.

```
pretrained_net.fc
```

```
Linear(in_features=512, out_features=1000, bias=True)
```

Là một lớp được kết nối hoàn toàn, nó biến đổi các đầu ra tổng hợp trung bình toàn cầu cuối cùng của ResNet thành 1000 đầu ra lớp của tập dữ liệu ImageNet. Sau đó, chúng tôi xây dựng một mạng nơ-ron mới làm mô hình mục tiêu. Nó được định nghĩa theo cách tương tự như mô hình nguồn được đào tạo trước ngoại trừ số lượng đầu ra của nó trong lớp cuối cùng được đặt thành số lớp trong tập dữ liệu đích (chứ không phải 1000).

Trong đoạn code sau, các tham số mô hình trong các đối tượng biến thành viên của đối tượng mô hình đích finetune_net được khởi tạo thành các tham số mô hình của lớp tương ứng của mô hình nguồn. Vì các thông số mô hình trong các tính năng được đào tạo trước trên tập dữ liệu ImageNet và đủ tốt, nói chung chỉ cần một tốc độ học tập nhỏ là cần thiết để tinh chỉnh các tham số này.

Các tham số mô hình trong đầu ra biến thành viên được khởi tạo ngẫu nhiên và thường yêu cầu tốc độ học tập lớn hơn để đào tạo từ đầu. Giả sử rằng tốc độ học tập trong trường hợp Trainer là η, chúng ta đặt tốc độ học tập của các tham số mô hình trong đầu ra biến thành viên là 10η trong lần lặp.

Trong đoạn mã dưới đây, các tham số mô hình trước lớp đầu ra của đối tượng mô hình đích `finetune_net` được khởi tạo thành các tham số mô hình của các lớp tương ứng từ mô hình nguồn. Vì các thông số mô hình này thu được thông qua đào tạo trước trên ImageNet, chúng có hiệu quả. Do đó, chúng ta chỉ có thể sử dụng một tốc độ học tập nhỏ để * tốt-tune* các thông số được đào tạo trước đó. Ngược lại, các tham số mô hình trong lớp đầu ra được khởi tạo ngẫu nhiên và thường yêu cầu một tỷ lệ học tập lớn hơn để được học từ đầu. Hãy để tốc độ học cơ bản là η, tốc độ học tập là 10η sẽ được sử dụng để lặp lại các tham số mô hình trong lớp đầu ra.

```
finetune_net = torchvision.models.resnet18(pretrained=True)
finetune_net.fc = nn.Linear(finetune_net.fc.in_features, 2)
nn.init.xavier_uniform_(finetune_net.fc.weight);
```

Tinh chỉnh mô hình

Đầu tiên, chúng tôi xác định một hàm đào tạo `train_fine_tuning` sử dụng tinh chỉnh để nó có thể được gọi nhiều lần.

```
# If `param_group=True`, the model parameters in the output layer will be
# updated using a learning rate ten times greater
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5,
                      param_group=True):
    train_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'train'), transform=train_augs),
        batch_size=batch_size, shuffle=True)
    test_iter = torch.utils.data.DataLoader(torchvision.datasets.ImageFolder(
        os.path.join(data_dir, 'test'), transform=test_augs),
        batch_size=batch_size)
    devices = d2l.try_all_gpus()
    loss = nn.CrossEntropyLoss(reduction="none")
    if param_group:
        params_1x = [param for name, param in net.named_parameters()
             if name not in ["fc.weight", "fc.bias"]]
        trainer = torch.optim.SGD([{'params': params_1x},
                                   {'params': net.fc.parameters(),
                                    'lr': learning_rate * 10}],
                                  lr=learning_rate, weight_decay=0.001)
    else:
        trainer = torch.optim.SGD(net.parameters(), lr=learning_rate,
                                  weight_decay=0.001)
```

(continues on next page)

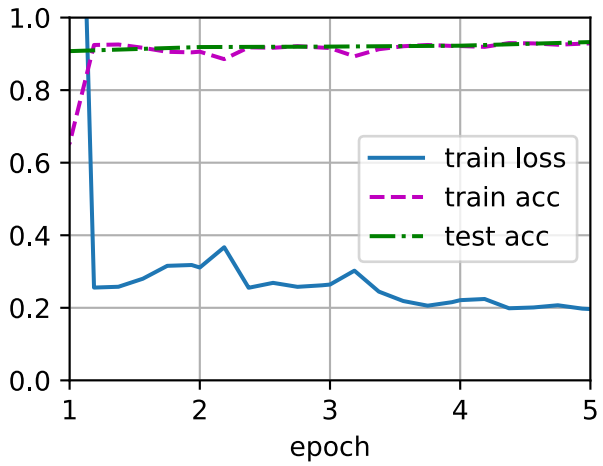
(continued from previous page)

```
    d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                   devices)
```

Chúng tôi đặt tỷ lệ học tập cơ bản thành một giá trị nhỏ để * tốt-tune* các thông số mô hình thu được thông qua pretraining. Dựa trên các cài đặt trước đó, chúng tôi sẽ đào tạo các tham số lớp đầu ra của mô hình mục tiêu từ đầu bằng cách sử dụng tốc độ học tập lớn hơn mười lần.

```
train_fine_tuning(finetune_net, 5e-5)
```

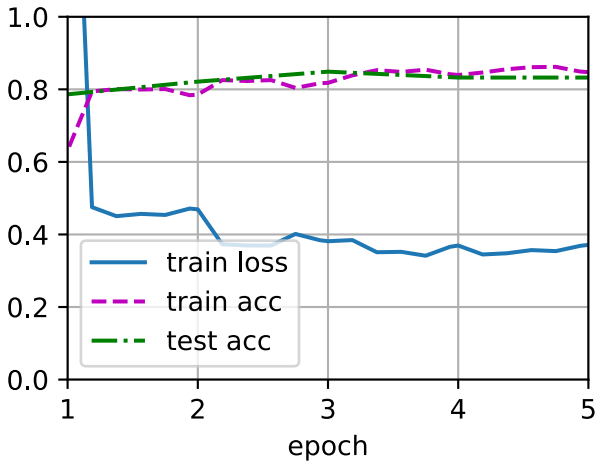
```
loss 0.197, train acc 0.929, test acc 0.932
1041.9 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Để so sánh, chúng ta định nghĩa một mô hình giống hệt nhau, nhưng khởi tạo tất cả các tham số mô hình của nó thành các giá trị ngẫu nhiên. Vì toàn bộ mô hình cần được đào tạo từ đầu, chúng ta có thể sử dụng tốc độ học tập lớn hơn.

```
scratch_net = torchvision.models.resnet18()
scratch_net.fc = nn.Linear(scratch_net.fc.in_features, 2)
train_fine_tuning(scratch_net, 5e-4, param_group=False)
```

```
loss 0.371, train acc 0.848, test acc 0.833
1614.7 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Như chúng ta có thể thấy, mô hình tinh chỉnh có xu hướng hoạt động tốt hơn cho cùng một kỷ nguyên vì các giá trị tham số ban đầu của nó hiệu quả hơn.

14.2.3 Tóm tắt

- Chuyển học tập chuyển kiến thức học được từ tập dữ liệu nguồn sang tập dữ liệu đích. Tinh chỉnh là một kỹ thuật phổ biến để học chuyển giao.
- Mô hình đích sao chép tất cả các thiết kế mô hình với các tham số của chúng từ mô hình nguồn ngoại trừ lớp đầu ra và tinh chỉnh các tham số này dựa trên tập dữ liệu đích. Ngược lại, lớp đầu ra của mô hình mục tiêu cần được đào tạo từ đầu.
- Nói chung, các thông số tinh chỉnh sử dụng tốc độ học tập nhỏ hơn, trong khi đào tạo lớp đầu ra từ đầu có thể sử dụng tốc độ học tập lớn hơn.

14.2.4 Bài tập

1. Tiếp tục tăng tỷ lệ học tập của `finetune_net`. Làm thế nào để độ chính xác của mô hình thay đổi?
2. Điều chỉnh thêm các siêu tham số của `finetune_net` và `scratch_net` trong thí nghiệm so sánh. Họ vẫn khác nhau về độ chính xác?
3. Đặt các tham số trước lớp đầu ra của `finetune_net` cho các tham số của mô hình nguồn và làm * không* cập nhật chúng trong quá trình đào tạo. Làm thế nào để độ chính xác của mô hình thay đổi? Bạn có thể sử dụng mã sau.

```
for param in finetune_net.parameters():
    param.requires_grad = False
```

4. Trên thực tế, có một lớp "hotdog" trong tập dữ liệu `ImageNet`. Tham số trọng lượng tương ứng của nó trong lớp đầu ra có thể thu được thông qua mã sau. Làm thế nào chúng ta có thể tận dụng thông số trọng lượng này?

```
weight = pretrained_net.fc.weight
hotdog_w = torch.split(weight.data, 1, dim=0)[934]
hotdog_w.shape
```

```
torch.Size([1, 512])
```

Discussions[172]

14.3 Hộp phát hiện và giới hạn đối tượng

Trong các phần trước (ví dụ: Section 8.1—Section 8.4), chúng tôi đã giới thiệu các mô hình khác nhau để phân loại hình ảnh. Trong các tác vụ phân loại hình ảnh, chúng tôi giả định rằng chỉ có * một* đối tượng chính trong hình ảnh và chúng tôi chỉ tập trung vào cách nhận dạng danh mục của nó. Tuy nhiên, thường có các đối tượng * nhiều* trong hình ảnh quan tâm. Chúng tôi không chỉ muốn biết danh mục của họ, mà còn là vị trí cụ thể của họ trong hình ảnh. Trong tầm nhìn máy tính, chúng tôi đề cập đến các tác vụ như *phát hiện đối tượng* (hoặc *nhận dạng đối tượng*).

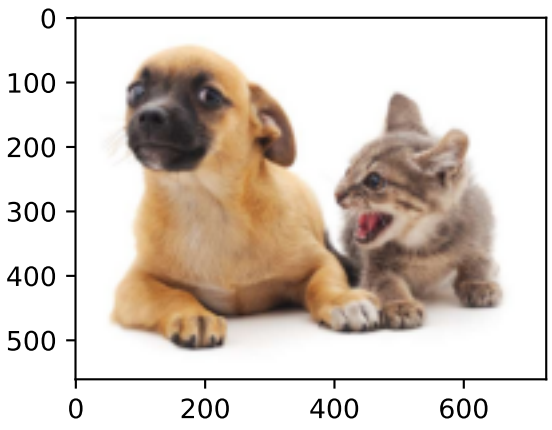
Phát hiện đối tượng đã được ứng dụng rộng rãi trong nhiều lĩnh vực. Ví dụ, tự lái cần lên kế hoạch cho các tuyến đường di chuyển bằng cách phát hiện vị trí của phương tiện, người đi bộ, đường xá và chướng ngại vật trong các hình ảnh video đã chụp. Bên cạnh đó, robot có thể sử dụng kỹ thuật này để phát hiện và bản địa hóa các đối tượng quan tâm trong suốt quá trình điều hướng môi trường. Hơn nữa, các hệ thống an ninh có thể cần phải phát hiện các vật bất thường, chẳng hạn như kẻ xâm nhập hoặc bom.

Trong vài phần tiếp theo, chúng tôi sẽ giới thiệu một số phương pháp học sâu để phát hiện đối tượng. Chúng tôi sẽ bắt đầu với phần giới thiệu về *vị trí* (hoặc *locations*) của các đối tượng.

```
%matplotlib inline
import torch
from d2l import torch as d2l
```

Chúng tôi sẽ tải hình ảnh mẫu được sử dụng trong phần này. Chúng ta có thể thấy rằng có một chú chó ở phía bên trái của hình ảnh và một con mèo bên phải. Chúng là hai đối tượng chính trong hình ảnh này.

```
d2l.set_figsize()
img = d2l.plt.imread('../img/catdog.jpg')
d2l.plt.imshow(img);
```



[172] https://discuss.d2l.ai/t/1439

14.3.1 Hộp giới hạn

Trong phát hiện đối tượng, chúng ta thường sử dụng một hộp giới hạn * để mô tả vị trí không gian của một đối tượng. Hộp giới hạn là hình chữ nhật, được xác định bởi tọa độ x và y của góc trên bên trái của hình chữ nhật và tọa độ như vậy của góc dưới bên phải. Một biểu diễn hộp giới hạn thường được sử dụng khác là tọa độ (x, y)-trục của trung tâm hộp giới hạn, và chiều rộng và chiều cao của hộp.

Ở đây chúng ta định nghĩa các hàm để chuyển đổi giữa hai các hàm này hai đại diện : `box_corner_to_center` chuyển đổi từ biểu diễn hai góc sang trình bày chiều rộng trung tâm và `box_center_to_corner` ngược lại. Đối số đầu vào `boxes` phải là một tensor hai chiều của hình dạng $(n, 4)$, trong đó n là số hộp giới hạn.

```
#@save
def box_corner_to_center(boxes):
    """Convert from (upper-left, lower-right) to (center, width, height)."""
    x1, y1, x2, y2 = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    cx = (x1 + x2) / 2
    cy = (y1 + y2) / 2
    w = x2 - x1
    h = y2 - y1
    boxes = torch.stack((cx, cy, w, h), axis=-1)
    return boxes

#@save
def box_center_to_corner(boxes):
    """Convert from (center, width, height) to (upper-left, lower-right)."""
    cx, cy, w, h = boxes[:, 0], boxes[:, 1], boxes[:, 2], boxes[:, 3]
    x1 = cx - 0.5 * w
    y1 = cy - 0.5 * h
    x2 = cx + 0.5 * w
    y2 = cy + 0.5 * h
    boxes = torch.stack((x1, y1, x2, y2), axis=-1)
    return boxes
```

Chúng tôi sẽ xác định các hộp giới hạn của chó và con mèo trong hình ảnh dựa trên thông tin tọa độ. Nguồn gốc của tọa độ trong hình ảnh là góc trên bên trái của hình ảnh, và bên phải và xuống là các hướng dương của các trục x và y, tương ứng.

```
# Here `bbox` is the abbreviation for bounding box
dog_bbox, cat_bbox = [60.0, 45.0, 378.0, 516.0], [400.0, 112.0, 655.0, 493.0]
```

Chúng tôi có thể xác minh tính chính xác của hai chức năng chuyển đổi hộp giới hạn bằng cách chuyển đổi hai lần.

```
boxes = torch.tensor((dog_bbox, cat_bbox))
box_center_to_corner(box_corner_to_center(boxes)) == boxes
```

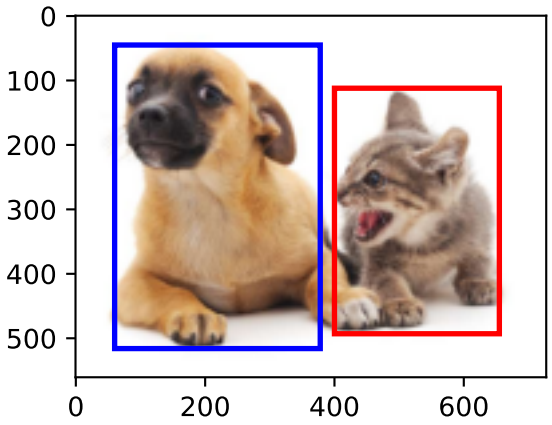
```
tensor([[True, True, True, True],
        [True, True, True, True]])
```

Hãy để chúng tôi vẽ các hộp giới hạn trong hình ảnh để kiểm tra xem chúng có chính xác không. Trước khi vẽ, chúng ta sẽ xác định một hàm helper `bbox_to_rect`. Nó đại diện cho hộp giới hạn ở định dạng hộp giới hạn của gói `matplotlib`.

```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (upper-left x, upper-left y, lower-right x,
    # lower-right y) format to the matplotlib format: ((upper-left x,
    # upper-left y), width, height)
    return d2l.plt.Rectangle(
        xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
        fill=False, edgecolor=color, linewidth=2)
```

Sau khi thêm các hộp giới hạn trên hình ảnh, chúng ta có thể thấy rằng đường viền chính của hai đối tượng về cơ bản nằm trong hai hộp.

```
fig = d2l.plt.imshow(img)
fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```



14.3.2 Tóm tắt

- Phát hiện đối tượng không chỉ nhận ra tất cả các đối tượng quan tâm trong hình ảnh, mà còn cả vị trí của chúng. Vị trí thường được biểu diễn bằng một hộp giới hạn hình chữ nhật.
- Chúng ta có thể chuyển đổi giữa hai biểu diễn hộp giới hạn thường được sử dụng.

14.3.3 Bài tập

1. Tìm một hình ảnh khác và cố gắng gắn nhãn một hộp giới hạn chứa đối tượng. So sánh các hộp và danh mục giới hạn ghi nhãn: thường mất nhiều thời gian hơn?
2. Tại sao chiều trong cùng của đối số đầu vào `boxes` của `box_corner_to_center` và `box_center_to_corner` luôn là 4?

Discussions[173]

[173] https://discuss.d2l.ai/t/1527

14.4 Hộp neo

Thuật toán phát hiện đối tượng thường lấy mẫu một số lượng lớn các vùng trong hình ảnh đầu vào, xác định xem các vùng này có chứa các đối tượng quan tâm hay không, và điều chỉnh ranh giới của các vùng để dự đoán *hộp giới hạn đất-truth* of the objects các đối tượng more accurately chính xác. Các mô hình khác nhau có thể áp dụng các sơ đồ lấy mẫu khu vực khác nhau. Ở đây chúng tôi giới thiệu một trong những phương pháp như vậy: nó tạo ra nhiều hộp giới hạn với tỷ lệ khác nhau và tỷ lệ khung hình tập trung vào mỗi pixel. Các hộp giới hạn này được gọi là hộp neo *. Chúng tôi sẽ thiết kế một mô hình phát hiện đối tượng dựa trên các hộp neo trong Section 14.7.

Đầu tiên, chúng ta hãy sửa đổi độ chính xác in ấn chỉ để kết quả đầu ra ngắn gọn hơn.

```
%matplotlib inline
import torch
from d2l import torch as d2l

torch.set_printoptions(2)  # Simplify printing accuracy
```

14.4.1 Tạo nhiều hộp neo

Giả sử hình ảnh đầu vào có chiều cao h và chiều rộng w. Chúng tôi tạo ra các hộp neo với các hình dạng khác nhau tập trung vào từng pixel của hình ảnh. Hãy để *quy mô* là $s \in (0, 1]$ và tỷ lệ khung hình* (tỷ lệ chiều rộng trên chiều cao) là $r > 0$. Sau đó chiều rộng và chiều cao của hộp neo là $ws\sqrt{r}$ và $hs/\sqrt{r}$, tôn trọng. Lưu ý rằng khi vị trí trung tâm được đưa ra, một hộp neo có chiều rộng và chiều cao đã biết được xác định.

Để tạo ra nhiều hộp neo với các hình dạng khác nhau, chúng ta hãy đặt một loạt các thang đo $s_1, \ldots, s_n$ và một loạt các tỷ lệ khung hình $r_1, \ldots, r_m$. Khi sử dụng tất cả các kết hợp của các thang đo và tỷ lệ khung hình này với mỗi pixel làm trung tâm, hình ảnh đầu vào sẽ có tổng cộng $whnm$ hộp neo. Mặc dù các hộp neo này có thể bao gồm tất cả các hộp giới hạn sự thật mặt đất, độ phức tạp tính toán dễ dàng quá cao. Trong thực tế, chúng ta chỉ có thể xem xét những kết hợp chứng s_1 hoặc r_1:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1). \tag{14.4.1}$$

Điều đó có nghĩa là, số lượng hộp neo tập trung vào cùng một điểm ảnh là $n + m - 1$. Đối với toàn bộ hình ảnh đầu vào, chúng tôi sẽ tạo tổng cộng $wh(n + m - 1)$ hộp neo.

Phương pháp tạo hộp neo trên được thực hiện trong chức năng `multibox_prior` sau đây. Chúng tôi chỉ định hình ảnh đầu vào, danh sách các thang đo và danh sách các tỷ lệ khung hình, sau đó chức năng này sẽ trả về tất cả các hộp neo.

```
#@save
def multibox_prior(data, sizes, ratios):
    """Generate anchor boxes with different shapes centered on each pixel."""
    in_height, in_width = data.shape[-2:]
    device, num_sizes, num_ratios = data.device, len(sizes), len(ratios)
    boxes_per_pixel = (num_sizes + num_ratios - 1)
    size_tensor = torch.tensor(sizes, device=device)
    ratio_tensor = torch.tensor(ratios, device=device)
    # Offsets are required to move the anchor to the center of a pixel. Since
    # a pixel has height=1 and width=1, we choose to offset our centers by 0.5
    offset_h, offset_w = 0.5, 0.5
    steps_h = 1.0 / in_height  # Scaled steps in y axis
```

(continues on next page)

(continued from previous page)

```
    steps_w = 1.0 / in_width  # Scaled steps in x axis

    # Generate all center points for the anchor boxes
    center_h = (torch.arange(in_height, device=device) + offset_h) * steps_h
    center_w = (torch.arange(in_width, device=device) + offset_w) * steps_w
    shift_y, shift_x = torch.meshgrid(center_h, center_w)
    shift_y, shift_x = shift_y.reshape(-1), shift_x.reshape(-1)

    # Generate `boxes_per_pixel` number of heights and widths that are later
    # used to create anchor box corner coordinates (xmin, xmax, ymin, ymax)
    w = torch.cat((size_tensor * torch.sqrt(ratio_tensor[0]),
                   sizes[0] * torch.sqrt(ratio_tensor[1:])))\
                   * in_height / in_width  # Handle rectangular inputs
    h = torch.cat((size_tensor / torch.sqrt(ratio_tensor[0]),
                   sizes[0] / torch.sqrt(ratio_tensor[1:])))
    # Divide by 2 to get half height and half width
    anchor_manipulations = torch.stack((-w, -h, w, h)).T.repeat(
                                        in_height * in_width, 1) / 2

    # Each center point will have `boxes_per_pixel` number of anchor boxes, so
    # generate a grid of all anchor box centers with `boxes_per_pixel` repeats
    out_grid = torch.stack([shift_x, shift_y, shift_x, shift_y],
                dim=1).repeat_interleave(boxes_per_pixel, dim=0)
    output = out_grid + anchor_manipulations
    return output.unsqueeze(0)
```

Chúng ta có thể thấy rằng hình dạng của biến hộp neo trả về `Y` là (kích thước lô, số lượng hộp neo, 4).

```
img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]

print(h, w)
X = torch.rand(size=(1, 3, h, w))  # Construct input data
Y = multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

```
561 728
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```

```
torch.Size([1, 2042040, 4])
```

Sau khi thay đổi hình dạng của biến hộp neo `Y` thành (chiều cao hình ảnh, chiều rộng hình ảnh, số hộp neo tập trung vào cùng một điểm ảnh, 4), chúng ta có thể lấy tất cả các hộp neo tập trung vào một vị trí pixel được chỉ định. Sau đây, chúng ta truy cập hộp neo đầu tiên tập trung vào (250, 250) . Nó có bốn phần tử: tọa độ (x, y)-trục ở góc trên bên trái và trục (x, y)-tọa độ ở góc dưới bên phải của hộp neo. Các giá trị tọa độ của cả hai trục được chia cho chiều rộng và chiều cao của hình ảnh, tương ứng; do đó, phạm vi nằm trong khoảng từ 0 đến 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

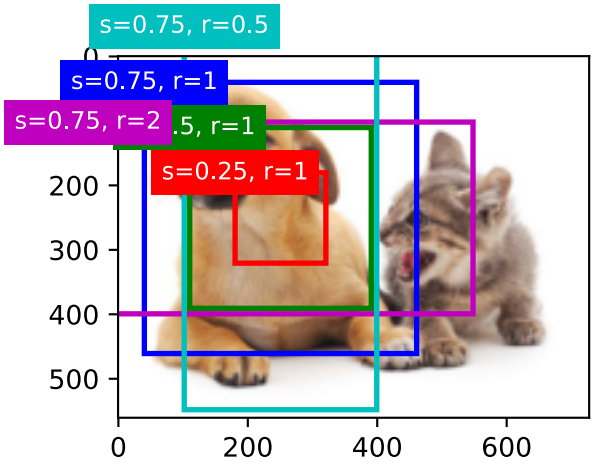
```
tensor([0.06, 0.07, 0.63, 0.82])
```

Để hiển thị tất cả các hộp neo tập trung vào một pixel trong image, chúng tôi xác định hàm `show_bboxes` sau để vẽ nhiều hộp giới hạn trên ảnh.

```
#@save
def show_bboxes(axes, bboxes, labels=None, colors=None):
    """Show bounding boxes."""

    def make_list(obj, default_values=None):
        if obj is None:
            obj = default_values
        elif not isinstance(obj, (list, tuple)):
            obj = [obj]
        return obj

    labels = make_list(labels)
    colors = make_list(colors, ['b', 'g', 'r', 'm', 'c'])
    for i, bbox in enumerate(bboxes):
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.detach().numpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

Như chúng ta vừa thấy, các giá trị tọa độ của các trục x và y trong biến `boxes` đã được chia cho chiều rộng và chiều cao của hình ảnh, tương ứng. Khi vẽ các hộp neo, chúng ta cần khôi phục các giá trị tọa độ ban đầu của chúng; do đó, chúng ta định nghĩa biến `bbox_scale` bên dưới. Bây giờ, chúng ta có thể vẽ tất cả các hộp neo tập trung vào (250, 250) trong hình ảnh. Như bạn có thể thấy, hộp neo màu xanh lam với thang điểm 0,75 và tỷ lệ khung hình là 1 bao quanh chó trong hình ảnh.

```
d2l.set_figsize()
bbox_scale = torch.tensor((w, h, w, h))
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
            ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
             's=0.75, r=0.5'])
```



14.4.2 Giao lộ qua Union (IoU)

Chúng tôi chỉ đề cập rằng một hộp neo "tốt" bao quanh chú chó trong hình ảnh. Nếu hộp giới hạn đất-chân lý của đối tượng được biết, làm thế nào có thể "tốt" ở đây được định lượng? Bằng trực giác, chúng ta có thể đo sự tương đồng giữa hộp neo và hộp giới hạn chân lý mặt đất. Chúng tôi biết rằng chỉ số *Jaccard* có thể đo lường sự tương đồng giữa hai bộ. Đưa ra các bộ $\mathcal{A}$ và $\mathcal{B}$, chỉ số Jaccard của chúng là kích thước giao điểm của chúng chia cho kích thước của công đoàn của chúng:

$$J(\mathcal{A},\mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \tag{14.4.2}$$

Trên thực tế, chúng ta có thể xem xét khu vực pixel của bất kỳ hộp giới hạn nào là một tập hợp các pixel. Bằng cách này, chúng ta có thể đo sự giống nhau của hai hộp giới hạn bằng chỉ số Jaccard của các bộ pixel của chúng. Đối với hai hộp giới hạn, chúng ta thường tham khảo chỉ số Jaccard của chúng là *giao lộ qua công đoàn * (* IoU*), đó là tỷ lệ giữa khu vực giao nhau của chúng với khu vực công đoàn của chúng, như thể hiện trong Fig. 14.4.1. Phạm vi của một IoU nằm trong khoảng từ 0 đến 1:0 có nghĩa là hai hộp giới hạn hoàn toàn không chồng lên nhau, trong khi 1 chỉ ra rằng hai hộp giới hạn bằng nhau.

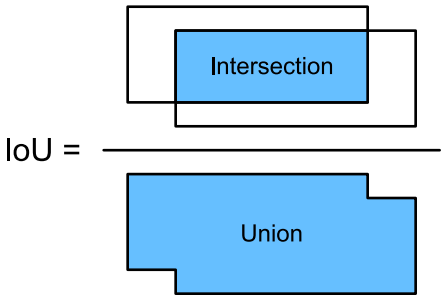


Fig. 14.4.1: IoU is the ratio of the intersection area to the union area of two bounding boxes.

Đối với phần còn lại của phần này, chúng tôi sẽ sử dụng IoU để đo sự tương đồng giữa các hộp neo và các hộp giới hạn chân lý mặt đất và giữa các hộp neo khác nhau. Cho hai danh sách các hộp neo hoặc đường viền, `box_iou` sau sẽ tính toán IoU cặp của chúng trong hai danh sách này.

```
#@save
def box_iou(boxes1, boxes2):
```

(continues on next page)

(continued from previous page)

```
    """Compute pairwise IoU across two lists of anchor or bounding boxes."""
    box_area = lambda boxes: ((boxes[:, 2] - boxes[:, 0]) *
                              (boxes[:, 3] - boxes[:, 1]))
    # Shape of `boxes1`, `boxes2`, `areas1`, `areas2`: (no. of boxes1, 4),
    # (no. of boxes2, 4), (no. of boxes1,), (no. of boxes2,)
    areas1 = box_area(boxes1)
    areas2 = box_area(boxes2)
    # Shape of `inter_upperlefts`, `inter_lowerrights`, `inters`: (no. of
    # boxes1, no. of boxes2, 2)
    inter_upperlefts = torch.max(boxes1[:, None, :2], boxes2[:, :2])
    inter_lowerrights = torch.min(boxes1[:, None, 2:], boxes2[:, 2:])
    inters = (inter_lowerrights - inter_upperlefts).clamp(min=0)
    # Shape of `inter_areas` and `union_areas`: (no. of boxes1, no. of boxes2)
    inter_areas = inters[:, :, 0] * inters[:, :, 1]
    union_areas = areas1[:, None] + areas2 - inter_areas
    return inter_areas / union_areas
```

14.4.3 Dán nhãn hộp neo trong dữ liệu đào tạo

Trong một tập dữ liệu đào tạo, chúng tôi coi mỗi hộp neo là một ví dụ đào tạo. Để đào tạo một mô hình phát hiện đối tượng, chúng ta cần nhãn * class* và *offset* cho mỗi hộp neo, trong đó trước đây là lớp của đối tượng có liên quan đến hộp neo và sau này là phần bù của hộp giới hạn sự thật mặt đất so với hộp neo. Trong quá trình dự đoán, đối với mỗi hình ảnh, chúng tôi tạo ra nhiều hộp neo, dự đoán các lớp và bù đắp cho tất cả các hộp neo, điều chỉnh vị trí của chúng theo các bù đắp dự đoán để có được các hộp giới hạn dự đoán và cuối cùng chỉ xuất ra những hộp giới hạn dự đoán đáp ứng các tiêu chí nhất định.

Như chúng ta đã biết, một bộ đào tạo phát hiện đối tượng đi kèm với nhãn cho các vị trí của các hộp giới hạn * đất-chân thất* và các lớp của các đối tượng được bao quanh của chúng. Để gắn nhãn bất kỳ hộp neo * được tạo ra*, chúng tôi đề cập đến vị trí được dán nhãn và lớp của hộp giới hạn đất*được chỉ định * của nó gần nhất với hộp neo. Sau đây, chúng tôi mô tả một thuật toán để gán các hộp giới hạn sự thật gần nhất cho các hộp neo.

Gán các hộp giới hạn mặt đất-Truth cho Hộp neo

Với một hình ảnh, giả sử rằng các hộp neo là $A_1, A_2, \ldots, A_{n_a}$ và các hộp giới hạn đất-chân lý là $B_1, B_2, \ldots, B_{n_b}$, trong đó $n_a \geq n_b$. Chúng ta hãy xác định một ma trận $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$, có phần tử x_{ij} trong hàng i^{th} và cột j^{th} là IoU của hộp neo A_i và hộp giới hạn sự thật mặt đất B_j. Thuật toán bao gồm các bước sau:

1. Tìm phần tử lớn nhất trong ma trận $\mathbf{X}$ và biểu thị các chỉ số hàng và cột của nó là i_1 và j_1, tương ứng. Sau đó, hộp giới hạn sự thật mặt đất B_{j_1} được gán cho hộp neo A_{i_1}. Điều này khá trực quan vì A_{i_1} và B_{j_1} là gần nhất trong số tất cả các cặp hộp neo và hộp giới hạn sự thật mặt đất. Sau khi gán đầu tiên, loại bỏ tất cả các phần tử trong hàng $i_1{}^{\text{th}}$ và cột $j_1{}^{\text{th}}$ trong ma trận $\mathbf{X}$.
2. Tìm lớn nhất trong số các phần tử còn lại trong ma trận $\mathbf{X}$ và biểu thị các chỉ số hàng và cột của nó là i_2 và j_2, tương ứng. Chúng tôi gán hộp giới hạn sự thật mặt đất B_{j_2} để neo hộp A_{i_2} và loại bỏ tất cả các yếu tố trong hàng $i_2{}^{\text{th}}$ và cột $j_2{}^{\text{th}}$ trong ma trận $\mathbf{X}$.
3. Tại thời điểm này, các phần tử trong hai hàng và hai cột trong ma trận $\mathbf{X}$ đã bị loại bỏ. Chúng tôi tiến hành cho đến khi tất cả các phần tử trong n_b cột trong ma trận $\mathbf{X}$ bị loại bỏ. Tại thời điểm này, chúng tôi đã chỉ định một hộp giới hạn chân lý mặt đất cho mỗi hộp neo n_b.

4. Chỉ đi qua các hộp neo $n_a - n_b$ còn lại. Ví dụ, đưa ra bất kỳ hộp neo nào A_i, hãy tìm hộp giới hạn chân lý mặt đất B_j với IoU lớn nhất với A_i trong suốt hàng i^{th} của ma trận $\mathbf{X}$ và gán B_j cho A_i chỉ khi IoU này lớn hơn ngưỡng được xác định trước.

Hãy để chúng tôi minh họa thuật toán trên bằng cách sử dụng một ví dụ cụ thể. Như thể hiện trong Fig. 14.4.2 (trái), giả sử rằng giá trị lớn nhất trong ma trận $\mathbf{X}$ là x_{23}, chúng tôi gán hộp giới hạn sự thật mặt đất B_3 vào hộp neo A_2. Sau đó, chúng ta loại bỏ tất cả các phần tử trong hàng 2 và cột 3 của ma trận, tìm x_{71} lớn nhất trong các phần tử còn lại (khu vực bóng mờ) và gán hộp giới hạn sự thật mặt đất B_1 vào hộp neo A_7. Tiếp theo, như thể hiện trong Fig. 14.4.2 (giữa), loại bỏ tất cả các phần tử trong hàng 7 và cột 1 của ma trận, tìm x_{54} lớn nhất trong các phần tử còn lại (khu vực bóng mờ) và gán hộp giới hạn sự thật mặt đất B_4 vào hộp neo A_5. Cuối cùng, như thể hiện trong Fig. 14.4.2 (phải), loại bỏ tất cả các phần tử trong hàng 5 và cột 4 của ma trận, tìm x_{92} lớn nhất trong các phần tử còn lại (khu vực bóng mờ) và gán hộp giới hạn đất-chân lý B_2 vào hộp neo A_9. Sau đó, chúng ta chỉ cần đi qua các hộp neo còn lại A_1, A_3, A_4, A_6, A_8 và xác định xem có nên gán cho chúng các hộp giới hạn chân lý mặt đất theo ngưỡng hay không.

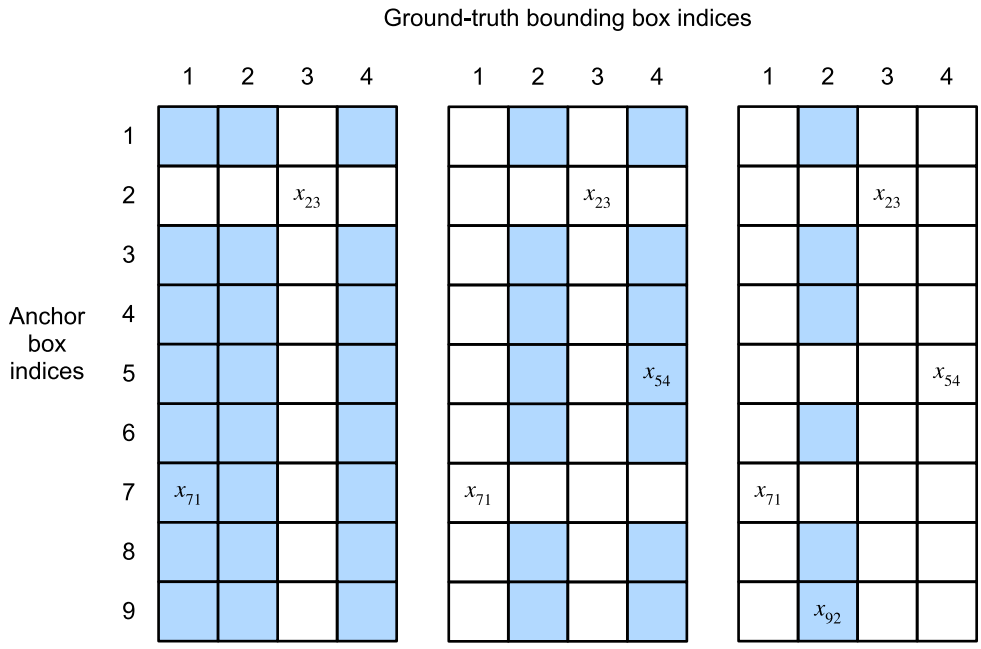


Fig. 14.4.2: Assigning ground-truth bounding boxes to anchor boxes.

Thuật toán này được thực hiện trong hàm `assign_anchor_to_bbox` sau.

```
#@save
def assign_anchor_to_bbox(ground_truth, anchors, device, iou_threshold=0.5):
    """Assign closest ground-truth bounding boxes to anchor boxes."""
    num_anchors, num_gt_boxes = anchors.shape[0], ground_truth.shape[0]
    # Element x_ij in the i-th row and j-th column is the IoU of the anchor
    # box i and the ground-truth bounding box j
    jaccard = box_iou(anchors, ground_truth)
    # Initialize the tensor to hold the assigned ground-truth bounding box for
    # each anchor
    anchors_bbox_map = torch.full((num_anchors,), -1, dtype=torch.long,
                                  device=device)
    # Assign ground-truth bounding boxes according to the threshold
    max_ious, indices = torch.max(jaccard, dim=1)
    anc_i = torch.nonzero(max_ious >= 0.5).reshape(-1)
```

(continues on next page)

(continued from previous page)

```
    box_j = indices[max_ious >= 0.5]
    anchors_bbox_map[anc_i] = box_j
    col_discard = torch.full((num_anchors,), -1)
    row_discard = torch.full((num_gt_boxes,), -1)
    for _ in range(num_gt_boxes):
        max_idx = torch.argmax(jaccard)  # Find the largest IoU
        box_idx = (max_idx % num_gt_boxes).long()
        anc_idx = (max_idx / num_gt_boxes).long()
        anchors_bbox_map[anc_idx] = box_idx
        jaccard[:, box_idx] = col_discard
        jaccard[anc_idx, :] = row_discard
    return anchors_bbox_map
```

Các lớp ghi nhãn và độ lệch

Bây giờ chúng ta có thể dán nhãn lớp và bù đắp cho mỗi hộp neo. Giả sử rằng một hộp neo A được gán một hộp giới hạn đất-chân lý B. Một mặt, lớp của hộp neo A sẽ được dán nhãn là của B. Mặt khác, độ lệch của hộp neo A sẽ được dán nhãn theo vị trí tương đối giữa tọa độ trung tâm là B và A cùng với kích thước tương đối giữa hai hộp này. Với các vị trí và kích thước khác nhau của các hộp khác nhau trong tập dữ liệu, chúng ta có thể áp dụng các biến đổi cho các vị trí và kích thước tương đối có thể dẫn đến các bù phân bố đồng đều hơn dễ phù hợp hơn. Ở đây chúng tôi mô tả một sự chuyển đổi chung. Cho tọa độ trung tâm của A và B là (x_a, y_a) và (x_b, y_b), chiều rộng của chúng là w_a và w_b, và chiều cao của chúng tương ứng là h_a và h_b. Chúng tôi có thể dán nhãn phần bù của A như

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \tag{14.4.3}$$

where default values of the constants are $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1$, and $\sigma_w = \sigma_h = 0.2$. This transformation is implemented below in the `offset_boxes` function.

```
#@save
def offset_boxes(anchors, assigned_bb, eps=1e-6):
    """Transform for anchor box offsets."""
    c_anc = d2l.box_corner_to_center(anchors)
    c_assigned_bb = d2l.box_corner_to_center(assigned_bb)
    offset_xy = 10 * (c_assigned_bb[:, :2] - c_anc[:, :2]) / c_anc[:, 2:]
    offset_wh = 5 * torch.log(eps + c_assigned_bb[:, 2:] / c_anc[:, 2:])
    offset = torch.cat([offset_xy, offset_wh], axis=1)
    return offset
```

Nếu một hộp neo không được gán một hộp giới hạn đất-truth, chúng ta chỉ cần dán nhãn lớp của hộp neo là "background". Các hộp neo có lớp nền thường được gọi là hộp neo * tiêu cực* và phần còn lại được gọi là hộp neo * tích cực*. Chúng tôi thực hiện hàm `multibox_target` sau để ** label class and offsets for anchor boxes** bằng cách sử dụng các hộp bounding ground-truth (đối số `labels`). Hàm này đặt lớp nền thành 0 và tăng chỉ số số nguyên của một lớp mới bằng một.

```
#@save
def multibox_target(anchors, labels):
    """Label anchor boxes using ground-truth bounding boxes."""
    batch_size, anchors = labels.shape[0], anchors.squeeze(0)
```

(continues on next page)

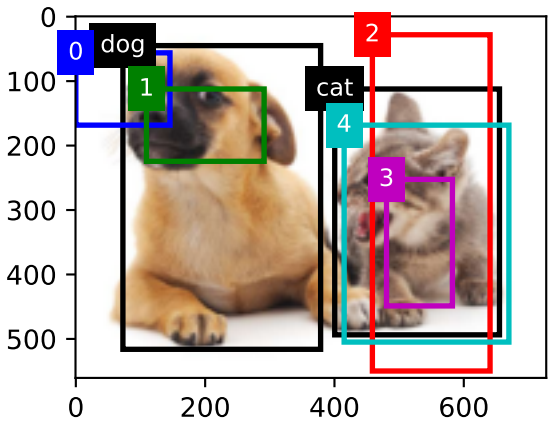
(continued from previous page)

```
    batch_offset, batch_mask, batch_class_labels = [], [], []
    device, num_anchors = anchors.device, anchors.shape[0]
    for i in range(batch_size):
        label = labels[i, :, :]
        anchors_bbox_map = assign_anchor_to_bbox(
            label[:, 1:], anchors, device)
        bbox_mask = ((anchors_bbox_map >= 0).float().unsqueeze(-1)).repeat(
            1, 4)
        # Initialize class labels and assigned bounding box coordinates with
        # zeros
        class_labels = torch.zeros(num_anchors, dtype=torch.long,
                                   device=device)
        assigned_bb = torch.zeros((num_anchors, 4), dtype=torch.float32,
                                  device=device)
        # Label classes of anchor boxes using their assigned ground-truth
        # bounding boxes. If an anchor box is not assigned any, we label its
        # class as background (the value remains zero)
        indices_true = torch.nonzero(anchors_bbox_map >= 0)
        bb_idx = anchors_bbox_map[indices_true]
        class_labels[indices_true] = label[bb_idx, 0].long() + 1
        assigned_bb[indices_true] = label[bb_idx, 1:]
        # Offset transformation
        offset = offset_boxes(anchors, assigned_bb) * bbox_mask
        batch_offset.append(offset.reshape(-1))
        batch_mask.append(bbox_mask.reshape(-1))
        batch_class_labels.append(class_labels)
    bbox_offset = torch.stack(batch_offset)
    bbox_mask = torch.stack(batch_mask)
    class_labels = torch.stack(batch_class_labels)
    return (bbox_offset, bbox_mask, class_labels)
```

Một ví dụ

Hãy để chúng tôi minh họa ghi nhãn hộp neo thông qua một ví dụ cụ thể. Chúng tôi xác định các hộp giới hạn đất cho chó và mèo trong hình ảnh được tải, trong đó phần tử đầu tiên là lớp (0 cho chó và 1 cho mèo) và bốn yếu tố còn lại là tọa độ (x, y)-trục ở góc trên bên trái và góc dưới bên phải (phạm vi nằm trong khoảng từ 0 đến 1). Chúng tôi cũng xây dựng năm hộp neo được dán nhãn bằng cách sử dụng tọa độ của góc trên bên trái và góc dưới bên phải: $A_0, \ldots, A_4$ (chỉ số bắt đầu từ 0). Sau đó, chúng ta vẽ các hộp giới hạn mặt đất và hộp neo trong hình ảnh

```
ground_truth = torch.tensor([[0, 0.1, 0.08, 0.52, 0.92],
                         [1, 0.55, 0.2, 0.9, 0.88]])
anchors = torch.tensor([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                    [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                    [0.57, 0.3, 0.92, 0.9]])

fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```



Sử dụng hàm `multibox_target` được xác định ở trên, chúng ta có thể label class và offset của các hộp neo này dựa trên các hộp giới hạn đất-chân thất cho chó và mèo. Trong ví dụ này, các chỉ số của các lớp nền, chó và mèo lần lượt là 0, 1 và 2. Dưới đây chúng tôi thêm một kích thước cho các ví dụ về hộp neo và các hộp giới hạn đất-chân lý.

```
labels = multibox_target(anchors.unsqueeze(dim=0),
                         ground_truth.unsqueeze(dim=0))
```

Có ba mục trong kết quả trả về, tất cả đều ở định dạng tensor. Mục thứ ba chứa các lớp được dán nhãn của các hộp neo đầu vào.

Chúng ta hãy phân tích các nhãn lớp trả về dưới đây dựa trên hộp neo và vị trí hộp giới hạn đất-chân lý trong hình ảnh. Đầu tiên, trong số tất cả các cặp hộp neo và hộp giới hạn sự thật mặt đất, IoU của hộp neo A_4 và hộp giới hạn sự thật mặt đất của con mèo là lớn nhất. Do đó, lớp A_4 được dán nhãn là con mèo. Lấy ra các cặp chứa A_4 hoặc hộp giới hạn sự thật mặt đất của con mèo, trong số các cặp còn lại của hộp neo A_1 và hộp giới hạn sự thật mặt đất của chó có IoU lớn nhất. Vì vậy, lớp A_1 được dán nhãn là chó. Tiếp theo, chúng ta cần phải đi qua ba hộp neo chưa được dán nhãn còn lại: A_0, A_2 và A_3. Đối với A_0, lớp của hộp giới hạn chân thất-chân lý với IoU lớn nhất là chó, nhưng IoU nằm dưới ngưỡng được xác định trước (0,5), vì vậy lớp được dán nhãn là nền; đối với A_2, lớp của hộp giới hạn sự thật mặt đất với IoU lớn nhất là con mèo và IoU vượt quá ngưỡng, vì vậy lớp được dán nhãn là con mèo; đối với A_3, lớp của hộp giới hạn đất-chân lý với IoU lớn nhất là con mèo, nhưng giá trị nằm dưới ngưỡng, do đó lớp được dán nhãn là nền.

```
labels[2]
```

```
tensor([[0, 1, 2, 0, 2]])
```

Mục trả về thứ hai là một biến mặt nạ của hình dạng (kích thước lô, gấp bốn lần số hộp neo). Mỗi bốn phần tử trong biến mask tương ứng với bốn giá trị bù của mỗi hộp neo. Vì chúng ta không quan tâm đến việc phát hiện nền, sự bù đắp của lớp phủ định này không nên ảnh hưởng đến hàm khách quan. Thông qua phép nhân elementwise, các số không trong biến mask sẽ lọc ra các bù lớp âm trước khi tính hàm mục tiêu.

```
labels[1]
```

```
tensor([[0., 0., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 1.,
↪1.,
         1., 1.]])
```

Mục trả về đầu tiên chứa bốn giá trị bù được dán nhãn cho mỗi hộp neo. Lưu ý rằng các bù đắp của các hộp neo lớp âm được dán nhãn là số không.

```
labels[0]
```

```
tensor([[-0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00,  1.40e+00,  1.00e+01,
          2.59e+00,  7.18e+00, -1.20e+00,  2.69e-01,  1.68e+00, -1.57e+00,
         -0.00e+00, -0.00e+00, -0.00e+00, -0.00e+00, -5.71e-01, -1.00e+00,
          4.17e-06,  6.26e-01]])
```

14.4.4 Dự đoán các hộp giới hạn với ức chế không tối đa

Trong quá trình dự đoán, chúng tôi tạo ra nhiều hộp neo cho hình ảnh và dự đoán các lớp và bù đắp cho mỗi hộp. Do đó, một hộp giới hạn dự đoán * thu được theo một hộp neo với độ lệch dự đoán của nó. Dưới đây chúng tôi thực hiện hàm `offset_inverse` có các neo và dự đoán bù đắp làm đầu vào và áp dụng các biến đổi bù ngược để trả về phối hợp hộp giới hạn dự đoán .

```
#@save
def offset_inverse(anchors, offset_preds):
    """Predict bounding boxes based on anchor boxes with predicted offsets."""
    anc = d2l.box_corner_to_center(anchors)
    pred_bbox_xy = (offset_preds[:, :2] * anc[:, 2:] / 10) + anc[:, :2]
    pred_bbox_wh = torch.exp(offset_preds[:, 2:] / 5) * anc[:, 2:]
    pred_bbox = torch.cat((pred_bbox_xy, pred_bbox_wh), axis=1)
    predicted_bbox = d2l.box_center_to_corner(pred_bbox)
    return predicted_bbox
```

Khi có nhiều hộp neo, nhiều hộp giới hạn dự đoán tương tự (với sự chồng chéo đáng kể) có thể có khả năng xuất ra cho xung quanh cùng một đối tượng. Để đơn giản hóa đầu ra, chúng ta có thể hợp nhất các hộp giới hạn dự đoán tương tự thuộc về cùng một đối tượng bằng cách sử dụng *non-maximression* (NMS).

Đây là cách ức chế không tối đa hoạt động. Đối với một hộp giới hạn dự đoán B, mô hình phát hiện đối tượng tính toán khả năng dự đoán cho mỗi lớp. Biểu thị bởi p khả năng dự đoán lớn nhất, lớp tương ứng với xác suất này là lớp dự đoán cho B. Cụ thể, chúng tôi đề cập đến p là *sự tự tin* (điểm) của hộp giới hạn dự đoán B. Trên cùng một hình ảnh, tất cả các hộp giới hạn không nền được dự đoán được sắp xếp theo sự tự tin theo thứ tự giảm dần để tạo ra một danh sách L. Sau đó, chúng tôi thao tác danh sách được sắp xếp L trong các bước sau:

1. Chọn hộp giới hạn dự đoán B_1 với độ tin cậy cao nhất từ L làm cơ sở và loại bỏ tất cả các hộp giới hạn dự đoán không cơ sở có IoU với B_1 vượt quá ngưỡng được xác định trước ϵ từ L. Tại thời điểm này, L giữ hộp giới hạn dự đoán với sự tự tin cao nhất nhưng giảm những người khác quá giống với nó. Tóm lại, những người có *không tối đa* điểm tin cậy là * bị ức chế *.
2. Chọn hộp giới hạn dự đoán B_2 với độ tin cậy cao thứ hai từ L làm cơ sở khác và loại bỏ tất cả các hộp giới hạn dự đoán không cơ sở có IoU với B_2 vượt quá ϵ từ L.
3. Lặp lại quy trình trên cho đến khi tất cả các hộp giới hạn dự đoán trong L đã được sử dụng làm cơ sở. Tại thời điểm này, IoU của bất kỳ cặp hộp giới hạn dự đoán nào trong L nằm dưới ngưỡng ϵ; do đó, không có cặp nào quá giống nhau.
4. Xuất tất cả các hộp giới hạn dự đoán trong danh sách L.

Hàm `nms` sau đây sắp xếp điểm tin cậy theo thứ tự giảm dần và trả về các chỉ số của chúng.

```
#@save
def nms(boxes, scores, iou_threshold):
    """Sort confidence scores of predicted bounding boxes."""
    B = torch.argsort(scores, dim=-1, descending=True)
    keep = []  # Indices of predicted bounding boxes that will be kept
    while B.numel() > 0:
        i = B[0]
        keep.append(i)
        if B.numel() == 1: break
        iou = box_iou(boxes[i, :].reshape(-1, 4),
                      boxes[B[1:], :].reshape(-1, 4)).reshape(-1)
        inds = torch.nonzero(iou <= iou_threshold).reshape(-1)
        B = B[inds + 1]
    return torch.tensor(keep, device=boxes.device)
```

Chúng tôi xác định `multibox_detection` sau đây để áp dụng ức chế không tối đa để dự đoán các hộp giới hộp. Đừng lo lắng nếu bạn thấy việc thực hiện một chút phức tạp: chúng tôi sẽ chỉ ra cách nó hoạt động với một ví dụ cụ thể ngay sau khi thực hiện.

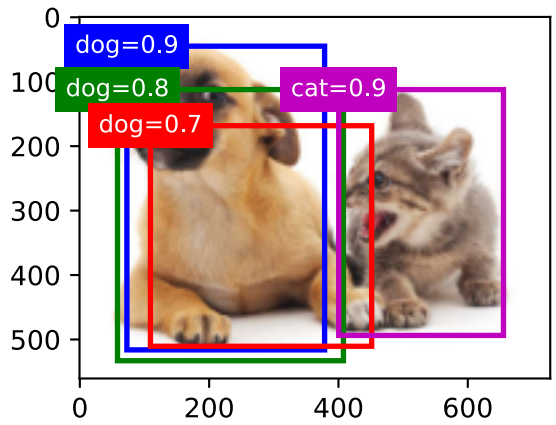
```
#@save
def multibox_detection(cls_probs, offset_preds, anchors, nms_threshold=0.5,
                       pos_threshold=0.009999999):
    """Predict bounding boxes using non-maximum suppression."""
    device, batch_size = cls_probs.device, cls_probs.shape[0]
    anchors = anchors.squeeze(0)
    num_classes, num_anchors = cls_probs.shape[1], cls_probs.shape[2]
    out = []
    for i in range(batch_size):
        cls_prob, offset_pred = cls_probs[i], offset_preds[i].reshape(-1, 4)
        conf, class_id = torch.max(cls_prob[1:], 0)
        predicted_bb = offset_inverse(anchors, offset_pred)
        keep = nms(predicted_bb, conf, nms_threshold)
        # Find all non-`keep` indices and set the class to background
        all_idx = torch.arange(num_anchors, dtype=torch.long, device=device)
        combined = torch.cat((keep, all_idx))
        uniques, counts = combined.unique(return_counts=True)
        non_keep = uniques[counts == 1]
        all_id_sorted = torch.cat((keep, non_keep))
        class_id[non_keep] = -1
        class_id = class_id[all_id_sorted]
        conf, predicted_bb = conf[all_id_sorted], predicted_bb[all_id_sorted]
        # Here `pos_threshold` is a threshold for positive (non-background)
        # predictions
        below_min_idx = (conf < pos_threshold)
        class_id[below_min_idx] = -1
        conf[below_min_idx] = 1 - conf[below_min_idx]
        pred_info = torch.cat((class_id.unsqueeze(1),
                               conf.unsqueeze(1),
                               predicted_bb), dim=1)
        out.append(pred_info)
    return torch.stack(out)
```

Bây giờ chúng ta hãy áp dụng các triển khai trên cho một ví dụ cụ thể với bốn hộp neo. Để đơn giản, chúng tôi giả định rằng các bù dự đoán là tất cả các số không. Điều này có nghĩa là các hộp giới hạn dự đoán là hộp neo. Đối với mỗi lớp trong số nền, chó và mèo, chúng tôi cũng xác định khả năng dự đoán của nó.

```
anchors = torch.tensor([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                        [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
offset_preds = torch.tensor([0] * anchors.numel())
cls_probs = torch.tensor([[0] * 4,  # Predicted background likelihood
                        [0.9, 0.8, 0.7, 0.1],  # Predicted dog likelihood
                        [0.1, 0.2, 0.3, 0.9]])  # Predicted cat likelihood
```

Chúng ta có thể vẽ những hộp giới hạn dự đoán này với sự tự tin của họ về hình ảnh

```
fig = d2l.plt.imshow(img)
show_bboxes(fig.axes, anchors * bbox_scale,
            ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```



Bây giờ chúng ta có thể gọi hàm `multibox_detection` để thực hiện triệt tiêu không tối đa, trong đó ngưỡng được đặt thành 0,5. Lưu ý rằng chúng ta thêm một chiều cho các ví dụ trong đầu vào tensor.

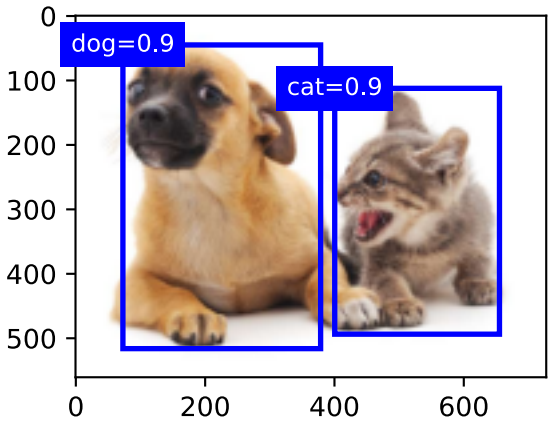
Chúng ta có thể thấy rằng hình dạng của kết quả trả về là (kích thước lô, số hộp neo, 6). Sáu yếu tố trong chiều trong cùng cung cấp thông tin đầu ra cho cùng một hộp giới hạn dự đoán. Phần tử đầu tiên là chỉ số lớp dự đoán, bắt đầu từ 0 (0 là chó và 1 là mèo). Giá trị -1 cho biết nền hoặc loại bỏ trong ức chế không tối đa. Yếu tố thứ hai là sự tự tin của hộp giới hạn dự đoán. Bốn phần tử còn lại là tọa độ (x, y)-trục của góc trên bên trái và góc dưới bên phải của hộp giới hạn dự đoán, tương ứng (phạm vi nằm trong khoảng từ 0 đến 1).

```
output = multibox_detection(cls_probs.unsqueeze(dim=0),
                            offset_preds.unsqueeze(dim=0),
                            anchors.unsqueeze(dim=0),
                            nms_threshold=0.5)
output
```

```
tensor([[[ 0.00,  0.90,  0.10,  0.08,  0.52,  0.92],
         [ 1.00,  0.90,  0.55,  0.20,  0.90,  0.88],
         [-1.00,  0.80,  0.08,  0.20,  0.56,  0.95],
         [-1.00,  0.70,  0.15,  0.30,  0.62,  0.91]]])
```

Sau khi loại bỏ những hộp giới hạn dự đoán của lớp -1, chúng ta có thể output the final predicted bounding box giữ bởi non-maximression.

```
fig = d2l.plt.imshow(img)
for i in output[0].detach().numpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [torch.tensor(i[2:]) * bbox_scale], label)
```



Trong thực tế, chúng ta có thể loại bỏ các hộp giới hạn dự đoán với độ tin cậy thấp hơn ngay cả trước khi thực hiện triệt tiêu không tối đa, do đó làm giảm tính toán trong thuật toán này. Chúng tôi cũng có thể xử lý hậu kỳ đầu ra của ức chế không tối đa, ví dụ, bằng cách chỉ giữ kết quả với sự tự tin cao hơn trong đầu ra cuối cùng.

14.4.5 Tóm tắt

- Chúng tôi tạo ra các hộp neo với các hình dạng khác nhau tập trung vào từng pixel của hình ảnh.
- Giao lộ qua liên kết (IoU), còn được gọi là chỉ số Jaccard, đo sự tương đồng của hai hộp giới hạn. Đó là tỷ lệ của khu vực giao nhau của họ với khu vực công đoàn của họ.
- Trong một bộ đào tạo, chúng ta cần hai loại nhãn cho mỗi hộp neo. Một là lớp của đối tượng có liên quan đến hộp neo và cái còn lại là độ lệch của hộp giới hạn đất-chân lý so với hộp neo.
- Trong quá trình dự đoán, chúng ta có thể sử dụng triệt tiêu không tối đa (NMS) để loại bỏ các hộp giới hạn dự đoán tương tự, từ đó đơn giản hóa đầu ra.

14.4.6 Bài tập

1. Thay đổi giá trị của `sizes` và `ratios` trong hàm `multibox_prior`. Những thay đổi đối với các hộp neo được tạo ra là gì?
2. Xây dựng và hình dung hai hộp giới hạn với IoU 0,5. Làm thế nào để họ chồng chéo với nhau?
3. Sửa đổi các biến `anchors` trong Section 14.4.3 và Section 14.4.4. Làm thế nào để kết quả thay đổi?
4. Ức chế không tối đa là một thuật toán tham lam ngăn chặn các hộp giới hạn dự đoán bằng cách * xóa* chúng. Có thể một số trong những cái bị xóa này thực sự hữu ích? Làm thế nào thuật toán này có thể được sửa đổi để ngăn chặn *softly*? Bạn có thể tham khảo Soft-NMS [Bodla.Singh.Chellappa.ea.2017].
5. Thay vì được làm thủ công, có thể học được sự đàn áp không tối đa không?

Discussions[174]

14.5 Phát hiện đối tượng đa quy mô

Trong Section 14.4, chúng tôi tạo ra nhiều hộp neo tập trung vào mỗi pixel của một hình ảnh đầu vào. Về cơ bản các hộp neo này đại diện cho các mẫu của các vùng khác nhau của hình ảnh. Tuy nhiên, chúng ta có thể kết thúc với quá nhiều hộp neo để tính toán nếu chúng được tạo cho *every* pixel. Hãy nghĩ về một hình ảnh đầu vào 561×728. Nếu năm hộp neo có hình dạng khác nhau được tạo ra cho mỗi pixel làm trung tâm của chúng, hơn hai triệu hộp neo ($561 \times 728 \times 5$) cần được dán nhãn và dự đoán trên hình ảnh.

14.5.1 Multiscale Neo Boxes

Bạn có thể nhận ra rằng không khó để giảm các hộp neo trên một hình ảnh. Ví dụ, chúng ta chỉ có thể lấy mẫu một phần nhỏ pixel từ hình ảnh đầu vào để tạo ra các hộp neo tập trung vào chúng. Ngoài ra, ở các quy mô khác nhau, chúng ta có thể tạo ra các số lượng hộp neo khác nhau có kích thước khác nhau. Về mặt trực giác, các đối tượng nhỏ hơn có nhiều khả năng xuất hiện trên một hình ảnh hơn so với các đối tượng lớn hơn. Ví dụ, các đối tượng 1×1, 1×2 và 2×2 có thể xuất hiện trên một hình ảnh 2×2 theo 4, 2 và 1 cách có thể tương ứng. Do đó, khi sử dụng các hộp neo nhỏ hơn để phát hiện các vật thể nhỏ hơn, chúng ta có thể lấy mẫu nhiều vùng hơn, trong khi đối với các vật thể lớn hơn chúng ta có thể lấy mẫu ít vùng hơn.

Để chứng minh cách tạo hộp neo ở nhiều thang đo, chúng ta hãy đọc một hình ảnh. Chiều cao và chiều rộng của nó lần lượt là 561 và 728 pixel.

```
%matplotlib inline
import torch
from d2l import torch as d2l

img = d2l.plt.imread('../img/catdog.jpg')
h, w = img.shape[:2]
h, w
```

```
(561, 728)
```

Nhớ lại rằng trong Section 7.2 chúng ta gọi một đầu ra mảng hai chiều của một lớp phức tạp là một bản đồ tính năng. Bằng cách xác định hình dạng bản đồ tính năng, chúng ta có thể xác định các trung tâm của các hộp neo được lấy mẫu đồng đều trên bất kỳ hình ảnh nào.

Hàm `display_anchors` được định nghĩa dưới đây. Chúng tôi tạo các hộp neo (`anchors`) trên bản đồ tính năng (`fmap`) với mỗi đơn vị (pixel) làm trung tâm hộp neo. Kể từ khi các giá trị tọa độ trục (x, y) trong các hộp neo (`anchors`) đã được chia cho chiều rộng và chiều cao của bản đồ tính năng (`fmap`), các giá trị này nằm giữa 0 đến 1, cho biết vị trí tương đối của các hộp neo trong bản đồ tính năng.

Vì các trung tâm của các hộp neo (`anchors`) được trải rộng trên tất cả các đơn vị trên bản đồ tính năng (`fmap`), các trung tâm này phải được * thống nhất* phân phối trên bất kỳ hình ảnh đầu vào nào về vị trí không gian tương đối của chúng. Cụ thể hơn, với chiều rộng và chiều cao của bản đồ tính năng `fmap_w` và `fmap_h`, tương ứng, chức năng sau đây sẽ * thống nhất* pixel mẫu trong `fmap_h` hàng và `fmap_w` cột trên bất kỳ hình ảnh đầu vào nào. Tập trung vào các pixel được lấy mẫu đồng đều này, các hộp neo có tỷ lệ `s` (giả sử độ dài của danh sách `s` là 1) và các tỷ lệ khung hình khác nhau (`ratios`) sẽ được tạo ra.

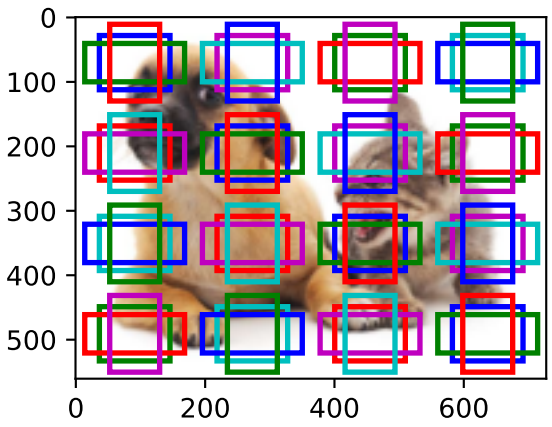
[174] https://discuss.d2l.ai/t/1603

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # Values on the first two dimensions do not affect the output
    fmap = torch.zeros((1, 10, fmap_h, fmap_w))
    anchors = d2l.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = torch.tensor((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img).axes,
                    anchors[0] * bbox_scale)
```

Đầu tiên, chúng ta hãy xem xét phát hiện các đối tượng nhỏ. Để phân biệt dễ dàng hơn khi được hiển thị, các hộp neo có các trung tâm khác nhau ở đây không chồng chéo: tỷ lệ hộp neo được đặt thành 0,15 và chiều cao và chiều rộng của bản đồ tính năng được đặt thành 4. Chúng ta có thể thấy rằng các trung tâm của các hộp neo trong 4 hàng và 4 cột trên hình ảnh được phân phối đồng đều.

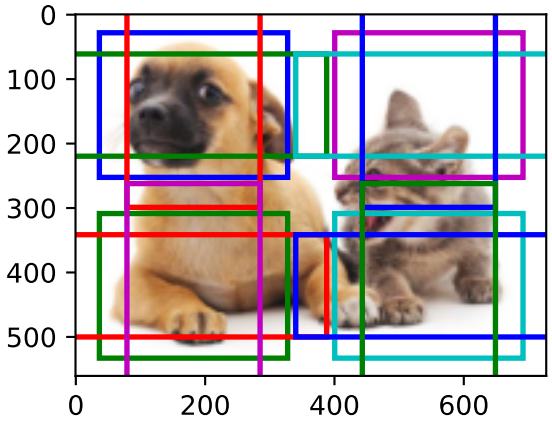
```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```

```
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



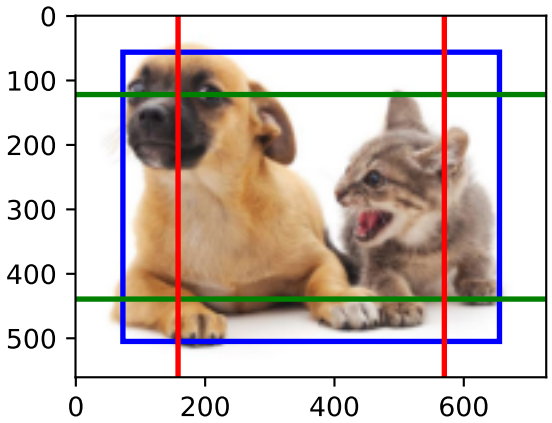
Chúng tôi chuyển sang giảm chiều cao và chiều rộng của bản đồ tính năng bằng một nửa và sử dụng các hộp neo lớn hơn để phát hiện các đối tượng lớn hơn. Khi tỷ lệ được đặt thành 0,4, một số hộp neo sẽ chồng lên nhau.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```



Cuối cùng, chúng tôi tiếp tục giảm chiều cao và chiều rộng của bản đồ tính năng bằng một nửa và tăng tỷ lệ hộp neo lên 0,8. Bây giờ trung tâm của hộp neo là trung tâm của hình ảnh.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```



14.5.2 Phát hiện đa quy mô

Vì chúng tôi đã tạo ra các hộp neo đa quy mô, chúng tôi sẽ sử dụng chúng để phát hiện các vật thể có kích thước khác nhau ở các quy mô khác nhau. Sau đây, chúng tôi giới thiệu phương pháp phát hiện đối tượng đa quy mô dựa trên CNN mà chúng tôi sẽ triển khai trong Section 14.7.

Ở một số quy mô, nói rằng chúng tôi có c tính năng bản đồ hình dạng $h \times w$. Sử dụng phương pháp trong Section 14.5.1, chúng tôi tạo ra hw bộ hộp neo, trong đó mỗi bộ có a hộp neo có cùng tâm. Ví dụ, ở thang điểm đầu tiên trong các thí nghiệm trong Section 14.5.1, cho mười (số kênh) bản đồ tính năng 4×4, chúng tôi tạo ra 16 bộ hộp neo, trong đó mỗi bộ chứa 3 hộp neo với cùng một trung tâm. Tiếp theo, mỗi hộp neo được dán nhãn với lớp và bù đắp dựa trên các hộp giới hạn chân lý mặt đất. Ở quy mô hiện tại, mô hình phát hiện đối tượng cần dự đoán các lớp và bù đắp của hw bộ hộp neo trên ảnh đầu vào, trong đó các bộ khác nhau có các trung tâm khác nhau.

Giả sử rằng các bản đồ tính năng c ở đây là các đầu ra trung gian thu được bởi CNN chuyển tiếp tuyên truyền dựa trên hình ảnh đầu vào. Vì có hw vị trí không gian khác nhau trên mỗi bản đồ tính năng, cùng một vị trí không gian có thể được coi là có c đơn vị. Theo định nghĩa của trường tiếp nhận trong Section 7.2, các đơn vị c này ở cùng vị trí không gian của các bản đồ tính năng có cùng trường tiếp nhận trên hình ảnh đầu vào: chúng

đại diện cho thông tin hình ảnh đầu vào trong cùng một trường tiếp nhận. Do đó, chúng ta có thể biến đổi các đơn vị c của các bản đồ tính năng ở cùng một vị trí không gian thành các lớp và bù đắp của các hộp neo a được tạo ra bằng cách sử dụng vị trí không gian này. Về bản chất, chúng ta sử dụng thông tin của hình ảnh đầu vào trong một trường tiếp nhận nhất định để dự đoán các lớp và bù đắp của các hộp neo gần với trường tiếp nhận đó trên hình ảnh đầu vào.

Khi các bản đồ tính năng ở các lớp khác nhau có các trường tiếp nhận kích thước khác nhau trên hình ảnh đầu vào, chúng có thể được sử dụng để phát hiện các đối tượng có kích thước khác nhau. Ví dụ, chúng ta có thể thiết kế một mạng nơ-ron trong đó các đơn vị của các bản đồ tính năng gần với lớp đầu ra có các trường tiếp nhận rộng hơn, do đó chúng có thể phát hiện các đối tượng lớn hơn từ hình ảnh đầu vào.

Tóm lại, chúng ta có thể tận dụng các biểu diễn theo lớp của hình ảnh ở nhiều cấp độ bằng các mạng thần kinh sâu để phát hiện đối tượng đa quy mô. Chúng tôi sẽ chỉ ra cách điều này hoạt động thông qua một ví dụ cụ thể trong Section 14.7.

14.5.3 Tóm tắt

- Ở nhiều thang đo, chúng ta có thể tạo ra các hộp neo với các kích cỡ khác nhau để phát hiện các vật thể có kích thước khác nhau.
- Bằng cách xác định hình dạng của bản đồ tính năng, chúng ta có thể xác định các trung tâm của các hộp neo được lấy mẫu đồng đều trên bất kỳ hình ảnh nào.
- Chúng tôi sử dụng thông tin của hình ảnh đầu vào trong một trường tiếp nhận nhất định để dự đoán các lớp và bù đắp của các hộp neo gần với trường tiếp nhận đó trên hình ảnh đầu vào.
- Thông qua deep learning, chúng ta có thể tận dụng các biểu diễn theo lớp của hình ảnh ở nhiều cấp độ để phát hiện đối tượng đa quy mô.

14.5.4 Bài tập

1. Theo các cuộc thảo luận của chúng tôi trong Section 8.1, các mạng thần kinh sâu tìm hiểu các tính năng phân cấp với mức độ trừu tượng ngày càng tăng cho hình ảnh. Trong phát hiện đối tượng đa quy mô, các bản đồ tính năng ở các thang đo khác nhau có tương ứng với các mức độ trừu tượng khác nhau không? Tại sao hoặc tại sao không?
2. Ở thang điểm đầu tiên (`fmap_w=4, fmap_h=4`) trong các thí nghiệm trong Section 14.5.1, tạo ra các hộp neo phân bố đồng đều có thể chồng lên nhau.
3. Đưa ra một biến bản đồ tính năng với hình dạng $1 \times c \times h \times w$, trong đó c, h và w là số kênh, chiều cao và chiều rộng của bản đồ tính năng, tương ứng. Làm thế nào bạn có thể chuyển đổi biến này thành các lớp và bù đắp của hộp neo? Hình dạng của đầu ra là gì?

Discussions[175]

[175] https://discuss.d2l.ai/t/1607

14.6 Bộ dữ liệu phát hiện đối tượng

Không có bộ dữ liệu nhỏ như MNIST và Fashion-MNIST trong lĩnh vực phát hiện đối tượng. Để nhanh chóng chứng minh các mô hình phát hiện đối tượng, chúng tôi đã thu thập và dán nhãn một tập dữ liệu nhỏ. Đầu tiên, chúng tôi chụp ảnh chuối miễn phí từ văn phòng của chúng tôi và tạo ra 1000 hình ảnh chuối với các vòng quay và kích cỡ khác nhau. Sau đó, chúng tôi đặt mỗi hình ảnh chuối ở một vị trí ngẫu nhiên trên một số hình nền. Cuối cùng, chúng tôi dán nhãn các hộp giới hạn cho những quả chuối trên hình ảnh.

14.6.1 Tải xuống dữ liệu

Bộ dữ liệu phát hiện chuối với tất cả các tệp nhãn hình ảnh và csv có thể được tải xuống trực tiếp từ Internet.

```
%matplotlib inline
import os
import pandas as pd
import torch
import torchvision
from d2l import torch as d2l
```

```
#@save
d2l.DATA_HUB['banana-detection'] = (
    d2l.DATA_URL + 'banana-detection.zip',
    '5de26c8fce5ccdea9f91267273464dc968d20d72')
```

14.6.2 Đọc tập dữ liệu

Chúng ta sẽ đọc dữ liệu phát hiện chuối trong hàm `read_data_bananas` bên dưới. Tập dữ liệu bao gồm một tệp csv cho nhãn lớp đối tượng và tọa độ hộp giới hạn đất-chân lý ở góc trên bên trái và dưới bên phải.

```
#@save
def read_data_bananas(is_train=True):
    """Read the banana detection dataset images and labels."""
    data_dir = d2l.download_extract('banana-detection')
    csv_fname = os.path.join(data_dir, 'bananas_train' if is_train
                             else 'bananas_val', 'label.csv')
    csv_data = pd.read_csv(csv_fname)
    csv_data = csv_data.set_index('img_name')
    images, targets = [], []
    for img_name, target in csv_data.iterrows():
        images.append(torchvision.io.read_image(
            os.path.join(data_dir, 'bananas_train' if is_train else
                         'bananas_val', 'images', f'{img_name}')))
        # Here `target` contains (class, upper-left x, upper-left y,
        # lower-right x, lower-right y), where all the images have the same
        # banana class (index 0)
        targets.append(list(target))
    return images, torch.tensor(targets).unsqueeze(1) / 256
```

Bằng cách sử dụng chức năng `read_data_bananas` để đọc hình ảnh và nhãn, lớp `BananasDataset` sau sẽ cho phép chúng ta tạo một phiên bản `Dataset` tùy chỉnh để tải bộ dữ liệu phát hiện chuối.

```
#@save
class BananasDataset(torch.utils.data.Dataset):
    """A customized dataset to load the banana detection dataset."""
    def __init__(self, is_train):
        self.features, self.labels = read_data_bananas(is_train)
        print('read ' + str(len(self.features)) + (f' training examples' if
              is_train else f' validation examples'))

    def __getitem__(self, idx):
        return (self.features[idx].float(), self.labels[idx])

    def __len__(self):
        return len(self.features)
```

Cuối cùng, chúng ta định nghĩa hàm `load_data_bananas` thành trả về hai phiên bản lặp dữ liệu cho cả bộ đào tạo và thử nghiệm. Đối với tập dữ liệu thử nghiệm, không cần phải đọc nó theo thứ tự ngẫu nhiên.

```
#@save
def load_data_bananas(batch_size):
    """Load the banana detection dataset."""
    train_iter = torch.utils.data.DataLoader(BananasDataset(is_train=True),
                                             batch_size, shuffle=True)
    val_iter = torch.utils.data.DataLoader(BananasDataset(is_train=False),
                                           batch_size)
    return train_iter, val_iter
```

Hãy để chúng tôi đọc một minibatch và in hình dạng của cả hình ảnh và nhãn trong minibatch này. Hình dạng của hình ảnh minibatch, (kích thước lô, số kênh, chiều cao, chiều rộng), trông quen thuộc: nó giống như trong các tác vụ phân loại hình ảnh trước đó của chúng tôi. Hình dạng của minibatch nhãn là (kích thước lô, m, 5), trong đó m là số hộp giới hạn lớn nhất có thể có mà bất kỳ hình ảnh nào có trong tập dữ liệu.

Mặc dù tính toán trong minibatches hiệu quả hơn, nhưng nó đòi hỏi tất cả các ví dụ hình ảnh chứa cùng một số hộp giới hạn để tạo thành một minibatch thông qua nối. Nói chung, hình ảnh có thể có một số hộp giới hạn khác nhau; do đó, hình ảnh có ít hơn m hộp giới hạn sẽ được đệm bằng các hộp giới hạn bất hợp pháp cho đến khi đạt được m. Sau đó, nhãn của mỗi hộp giới hạn được biểu diễn bằng một mảng có độ dài 5. Phần tử đầu tiên trong mảng là lớp của đối tượng trong hộp giới hạn, trong đó -1 chỉ ra một hộp giới hạn bất hợp pháp cho padding. Bốn phần tử còn lại của mảng là các giá trị tọa độ (x, y) -tọa độ của góc trên bên trái và góc dưới bên phải của hộp giới hạn (phạm vi nằm trong khoảng từ 0 đến 1). Đối với tập dữ liệu chuối, vì chỉ có một hộp giới hạn trên mỗi hình ảnh, chúng tôi có $m = 1$.

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_bananas(batch_size)
batch = next(iter(train_iter))
batch[0].shape, batch[1].shape
```

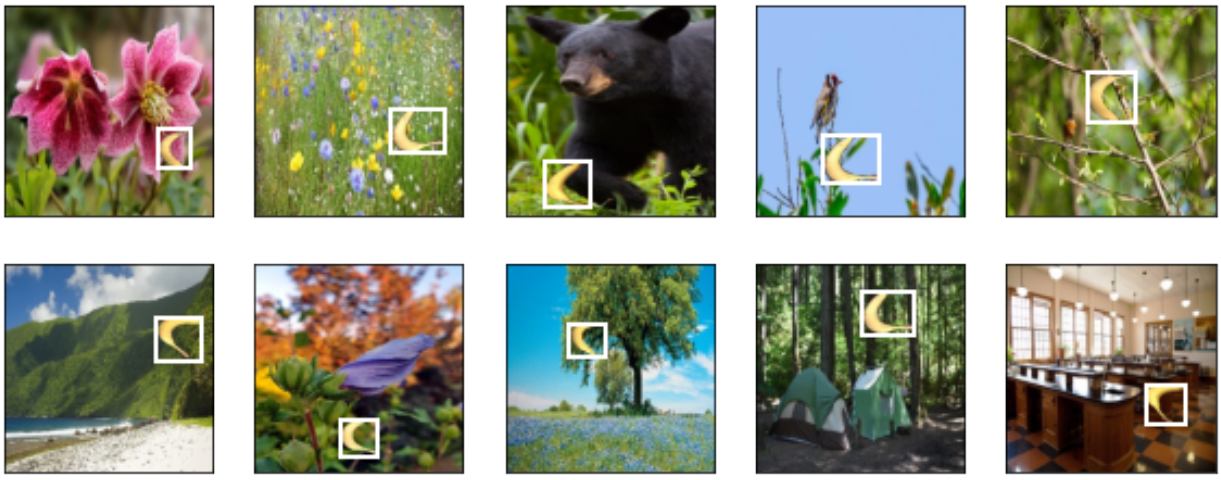
```
Downloading ../data/banana-detection.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/banana-detection.zip...
read 1000 training examples
read 100 validation examples
```

```
(torch.Size([32, 3, 256, 256]), torch.Size([32, 1, 5]))
```

14.6.3 Demonstration

Hãy để chúng tôi chứng minh mười hình ảnh với các hộp giới hạn mặt đất được dán nhãn của họ. Chúng ta có thể thấy rằng các vòng quay, kích thước và vị trí của chuối khác nhau trên tất cả các hình ảnh này. Tất nhiên, đây chỉ là một tập dữ liệu nhân tạo đơn giản. Trong thực tế, các bộ dữ liệu trong thế giới thực thường phức tạp hơn nhiều.

```
imgs = (batch[0][0:10].permute(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch[1][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```



14.6.4 Tóm tắt

- Bộ dữ liệu phát hiện chuối mà chúng tôi thu thập có thể được sử dụng để chứng minh các mô hình phát hiện đối tượng.
- Việc tải dữ liệu để phát hiện đối tượng tương tự như dữ liệu để phân loại hình ảnh. Tuy nhiên, trong việc phát hiện đối tượng, nhãn cũng chứa thông tin của các hộp giới hạn đất-chân lý, bị thiếu trong phân loại hình ảnh.

14.6.5 Bài tập

1. Thể hiện các hình ảnh khác với các hộp giới hạn sự thật mặt đất trong bộ dữ liệu phát hiện chuối. Làm thế nào để chúng khác nhau đối với các hộp giới hạn và các đối tượng?
2. Giả sử rằng chúng ta muốn áp dụng tăng cường dữ liệu, chẳng hạn như cắt xén ngẫu nhiên, để phát hiện đối tượng. Làm thế nào nó có thể khác với điều đó trong phân loại hình ảnh? Gợi ý: điều gì sẽ xảy ra nếu một hình ảnh cắt chỉ chứa một phần nhỏ của một đối tượng?

Discussions[176]

[176] https://discuss.d2l.ai/t/1608

14.7 Phát hiện Multibox Shot đơn

Trong Section 14.3—Section 14.6, chúng tôi giới thiệu các hộp giới hạn, hộp neo, phát hiện đối tượng đa quy mô và bộ dữ liệu để phát hiện đối tượng. Bây giờ chúng tôi đã sẵn sàng để sử dụng kiến thức nền như vậy để thiết kế một mô hình phát hiện đối tượng: phát hiện multibox shot đơn (SSD) (Liu et al., 2016). Mô hình này đơn giản, nhanh chóng và được sử dụng rộng rãi. Mặc dù đây chỉ là một trong số lượng lớn các mô hình phát hiện đối tượng, một số nguyên tắc thiết kế và chi tiết triển khai trong phần này cũng được áp dụng cho các mô hình khác.

14.7.1 Mô hình

Fig. 14.7.1 cung cấp một cái nhìn tổng quan về thiết kế phát hiện multibox một shot. Mô hình này chủ yếu bao gồm một mạng cơ sở theo sau là một số khối bản đồ tính năng đa thang. Mạng cơ sở là để trích xuất các tính năng từ hình ảnh đầu vào, vì vậy nó có thể sử dụng CNN sâu. Ví dụ, giấy phát hiện multibox một shot ban đầu sử dụng một mạng VGG cắt ngắn trước lớp phân loại (Liu et al., 2016), trong khi ResNet cũng đã được sử dụng phổ biến. Thông qua thiết kế của chúng tôi, chúng tôi có thể làm cho các bản đồ tính năng đầu ra mạng cơ sở lớn hơn để tạo ra nhiều hộp neo hơn để phát hiện các đối tượng nhỏ hơn. Sau đó, mỗi khối bản đồ tính năng đa thang giảm (ví dụ, bằng một nửa) chiều cao và chiều rộng của bản đồ tính năng từ khối trước đó và cho phép mỗi đơn vị của bản đồ tính năng tăng trường tiếp nhận của nó trên hình ảnh đầu vào.

Nhớ lại thiết kế phát hiện đối tượng đa quy mô thông qua các biểu diễn theo lớp của hình ảnh bằng các mạng thần kinh sâu trong Section 14.5. Vì bản đồ tính năng đa quy mô gần với đỉnh Fig. 14.7.1 nhỏ hơn nhưng có các trường tiếp nhận lớn hơn, chúng phù hợp để phát hiện các vật thể ít hơn nhưng lớn hơn.

Tóm lại, thông qua mạng cơ sở của nó và một số khối bản đồ tính năng đa quy mô, phát hiện multibox single-shot tạo ra một số lượng khác nhau của các hộp neo với kích thước khác nhau, và phát hiện các đối tượng kích thước khác nhau bằng cách dự đoán các lớp và bù đắp của các hộp neo (do đó các hộp giới hạn); do đó, đây là một mô hình phát hiện đối tượng đa quy mô.

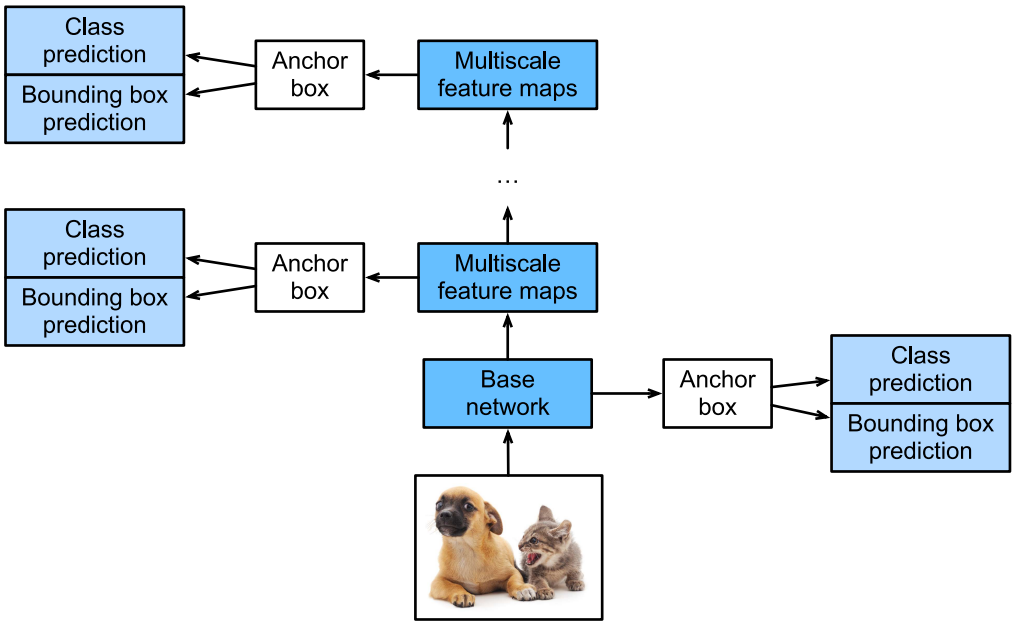


Fig. 14.7.1: As a multiscale object detection model, single-shot multibox detection mainly consists of a base network followed by several multiscale feature map blocks.

Sau đây, chúng tôi sẽ mô tả các chi tiết triển khai của các khối khác nhau trong Fig. 14.7.1. Để bắt đầu, chúng tôi thảo luận về cách thực hiện dự đoán lớp và hộp giới hạn.

Lớp dự đoán lớp

Hãy để số lượng các lớp đối tượng là q. Sau đó, hộp neo có $q+1$ lớp, trong đó lớp 0 là nền. Ở một số quy mô, giả sử rằng chiều cao và chiều rộng của bản đồ tính năng lần lượt là h và w. Khi a hộp neo được tạo ra với mỗi vị trí không gian của các bản đồ tính năng này làm trung tâm của chúng, tổng cộng hwa hộp neo cần được phân loại. Điều này thường làm cho việc phân loại với các lớp kết nối hoàn toàn không khả thi do chi phí tham số hóa nặng. Nhớ lại cách chúng ta sử dụng các kênh của các lớp phức tạp để dự đoán các lớp trong Section 8.3. Phát hiện multibox Single-shot sử dụng cùng một kỹ thuật để giảm độ phức tạp của mô hình.

Cụ thể, lớp dự đoán lớp sử dụng một lớp phức tạp mà không làm thay đổi chiều rộng hoặc chiều cao của bản đồ đối tượng. Bằng cách này, có thể có sự tương ứng một-một giữa các đầu ra và đầu vào ở cùng một kích thước không gian (chiều rộng và chiều cao) của bản đồ tính năng. Cụ thể hơn, các kênh của bản đồ tính năng đầu ra ở bất kỳ vị trí không gian nào (x, y) đại diện cho các dự đoán lớp cho tất cả các hộp neo tập trung vào (x, y) của bản đồ tính năng đầu vào. Để tạo ra các dự đoán hợp lệ, phải có $a(q+1)$ kênh đầu ra, trong đó cho cùng một vị trí không gian kênh đầu ra với chỉ số $i(q+1)+j$ đại diện cho dự đoán của lớp j $(0 \leq j \leq q)$ cho hộp neo i $(0 \leq i < a)$.

Dưới đây chúng ta xác định một lớp dự đoán lớp như vậy, chỉ định a và q thông qua các đối số `num_anchors` và `num_classes`, tương ứng. Lớp này sử dụng một lớp ghép 3×3 với lớp đệm là 1. Chiều rộng và chiều cao của đầu vào và đầu ra của lớp phức tạp này vẫn không thay đổi.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l


def cls_predictor(num_inputs, num_anchors, num_classes):
    return nn.Conv2d(num_inputs, num_anchors * (num_classes + 1),
                     kernel_size=3, padding=1)
```

Lớp dự đoán hộp giới hộp

Thiết kế của lớp dự đoán hộp giới hạn tương tự như của lớp dự đoán lớp. Sự khác biệt duy nhất nằm ở số lượng đầu ra cho mỗi hộp neo: ở đây chúng ta cần dự đoán bốn bù đắp hơn là $q+1$ lớp.

```
def bbox_predictor(num_inputs, num_anchors):
    return nn.Conv2d(num_inputs, num_anchors * 4, kernel_size=3, padding=1)
```

Concatenating dự đoán cho nhiều quy mô

Như chúng tôi đã đề cập, phát hiện multibox single shot sử dụng bản đồ tính năng đa thang để tạo ra các hộp neo và dự đoán các lớp và bù đắp của chúng. Ở các thang đo khác nhau, hình dạng của bản đồ tính năng hoặc số lượng hộp neo tập trung vào cùng một đơn vị có thể khác nhau. Do đó, hình dạng của đầu ra dự đoán ở các thang đo khác nhau có thể khác nhau.

Trong ví dụ sau, chúng tôi xây dựng các bản đồ tính năng ở hai thang khác nhau, `Y1` và `Y2`, cho cùng một minibatch, trong đó chiều cao và chiều rộng của `Y2` là một nửa so với `Y1`. Hãy để chúng tôi lấy dự đoán lớp học như một ví dụ. Giả sử rằng 5 và 3 hộp neo được tạo ra cho mỗi đơn vị trong `Y1` và `Y2`, tương ứng. Giả sử thêm rằng số lượng lớp đối tượng là 10. Đối với bản đồ tính năng `Y1` và `Y2` số kênh trong đầu ra dự đoán lớp là $5 \times (10 + 1) = 55$ và $3 \times (10 + 1) = 33$, tương ứng, trong đó một trong hai hình dạng đầu ra là (kích thước lô, số kênh, chiều cao, chiều rộng).

```
def forward(x, block):
    return block(x)

Y1 = forward(torch.zeros((2, 8, 20, 20)), cls_predictor(8, 5, 10))
Y2 = forward(torch.zeros((2, 16, 10, 10)), cls_predictor(16, 3, 10))
Y1.shape, Y2.shape
```

```
(torch.Size([2, 55, 20, 20]), torch.Size([2, 33, 10, 10]))
```

Như chúng ta có thể thấy, ngoại trừ kích thước kích thước lô, ba chiều còn lại đều có kích thước khác nhau. Để nối hai đầu ra dự đoán này để tính toán hiệu quả hơn, chúng tôi sẽ chuyển đổi các tenors này thành một định dạng nhất quán hơn.

Lưu ý rằng kích thước kênh giữ các dự đoán cho các hộp neo có cùng tâm. Đầu tiên chúng ta di chuyển kích thước này đến trong cùng. Vì kích thước lô vẫn giữ nguyên đối với các thang đo khác nhau, chúng ta có thể chuyển đổi đầu ra dự đoán thành tensor hai chiều với hình dạng (kích thước lô, chiều cao $\times$ chiều rộng $\times$ số kênh). Sau đó, chúng ta có thể nối các đầu ra như vậy ở các quy mô khác nhau dọc theo kích thước 1.

```
def flatten_pred(pred):
    return torch.flatten(pred.permute(0, 2, 3, 1), start_dim=1)

def concat_preds(preds):
    return torch.cat([flatten_pred(p) for p in preds], dim=1)
```

Bằng cách này, mặc dù `Y1` và `Y2` có kích thước khác nhau về các kênh, chiều cao và chiều rộng, chúng ta vẫn có thể nối hai đầu ra dự đoán này ở hai thang đo khác nhau cho cùng một minibatch.

```
concat_preds([Y1, Y2]).shape
```

```
torch.Size([2, 25300])
```

Khối lấy mẫu xuống

Để phát hiện các đối tượng ở nhiều thang đo, chúng tôi xác định khối lấy mẫu xuống sau `down_sample_blk` làm giảm một nửa chiều cao và chiều rộng của bản đồ tính năng đầu vào. Trên thực tế, khối này áp dụng thiết kế các khối VGG trong Section 8.2.1. Cụ thể hơn, mỗi khối lấy mẫu downsampling bao gồm hai 3×3 lớp ghép với lớp đệm 1 tiếp theo là một lớp tổng hợp tối đa 2×2 với sải chân là 2. Như chúng ta đã biết, 3×3 lớp ghép với lớp đệm 1 không thay đổi hình dạng của bản đồ tính năng. Tuy nhiên, tổng hợp tối đa 2×2 tiếp theo làm giảm chiều cao và chiều rộng của bản đồ tính năng đầu vào xuống một nửa. Đối với cả bản đồ tính năng đầu vào và đầu ra của khối lấy mẫu xuống này, vì $1 \times 2 + (3 - 1) + (3 - 1) = 6$, mỗi đơn vị trong đầu ra có một trường tiếp nhận 6×6 trên đầu vào. Do đó, khối lấy mẫu xuống mở rộng trường tiếp nhận của mỗi đơn vị trong bản đồ tính năng đầu ra của nó.

```
def down_sample_blk(in_channels, out_channels):
    blk = []
    for _ in range(2):
        blk.append(nn.Conv2d(in_channels, out_channels,
                             kernel_size=3, padding=1))
        blk.append(nn.BatchNorm2d(out_channels))
        blk.append(nn.ReLU())
        in_channels = out_channels
    blk.append(nn.MaxPool2d(2))
    return nn.Sequential(*blk)
```

Trong ví dụ sau, khối lấy mẫu downsampling xây dựng của chúng tôi thay đổi số kênh đầu vào và giảm một nửa chiều cao và chiều rộng của bản đồ tính năng đầu vào.

```
forward(torch.zeros((2, 3, 20, 20)), down_sample_blk(3, 10)).shape
```

```
torch.Size([2, 10, 10, 10])
```

Khối mạng cơ sở

Khối mạng cơ sở được sử dụng để trích xuất các tính năng từ hình ảnh đầu vào. Để đơn giản, chúng tôi xây dựng một mạng cơ sở nhỏ bao gồm ba khối lấy mẫu giảm gấp đôi số kênh tại mỗi khối. Với hình ảnh đầu vào 256×256, khối mạng cơ sở này xuất bản đồ tính năng 32×32 ($256/2^3 = 32$).

```
def base_net():
    blk = []
    num_filters = [3, 16, 32, 64]
    for i in range(len(num_filters) - 1):
        blk.append(down_sample_blk(num_filters[i], num_filters[i+1]))
    return nn.Sequential(*blk)

forward(torch.zeros((2, 3, 256, 256)), base_net()).shape
```

```
torch.Size([2, 64, 32, 32])
```

Mô hình hoàn chỉnh

Mô hình phát hiện multibox bắn hoàn chỉnh bao gồm năm khối. Các bản đồ tính năng được tạo ra bởi mỗi khối được sử dụng cho cả (i) tạo hộp neo và (ii) dự đoán các lớp và bù đắp của các hộp neo này. Trong số năm khối này, khối đầu tiên là khối mạng cơ sở, khối thứ hai đến thứ tư là các khối lấy mẫu xuống và khối cuối cùng sử dụng tổng hợp tối đa toàn cầu để giảm cả chiều cao và chiều rộng xuống 1. Về mặt kỹ thuật, khối thứ hai đến thứ năm là tất cả các khối bản đồ tính năng đa quy mô trong Fig. 14.7.1.

```
def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 1:
        blk = down_sample_blk(64, 128)
    elif i == 4:
        blk = nn.AdaptiveMaxPool2d((1,1))
    else:
        blk = down_sample_blk(128, 128)
    return blk
```

Bây giờ chúng ta xác định tuyên truyền chuyển tiếp cho mỗi khối. Khác với các tác vụ phân loại hình ảnh, đầu ra ở đây bao gồm (i) bản đồ tính năng CNN `Y`, (ii) hộp neo được tạo ra bằng cách sử dụng `Y` ở thang đo hiện tại, và (iii) lớp và bù dự đoán (dựa trên `Y`) cho các hộp neo này.

```
def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = d2l.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)
```

Nhớ lại rằng trong Fig. 14.7.1 một khối bản đồ tính năng đa quy mô gần với đỉnh là để phát hiện các đối tượng lớn hơn; do đó, nó cần phải tạo ra các hộp neo lớn hơn. Trong tuyên truyền chuyển tiếp ở trên, tại mỗi khối bản đồ tính năng đa thang chúng ta truyền trong một danh sách hai giá trị tỷ lệ thông qua đối số `sizes` của hàm `multibox_prior` được gọi (được mô tả trong Section 14.4). Sau đây, khoảng cách giữa 0,2 và 1,05 được chia đều thành năm phần để xác định các giá trị quy mô nhỏ hơn tại năm khối: 0.2, 0,37, 0,54, 0,71 và 0,88. Sau đó, các giá trị quy mô lớn hơn của chúng được đưa ra bởi $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, v.v.

```
sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
         [0.88, 0.961]]
ratios = [[1, 2, 0.5]] * 5
num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Bây giờ chúng ta có thể xác định mô hình hoàn chỉnh `TinySSD` như sau.

```
class TinySSD(nn.Module):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        idx_to_in_channels = [64, 128, 128, 128, 128]
        for i in range(5):
            # Equivalent to the assignment statement `self.blk_i = get_blk(i)`
            setattr(self, f'blk_{i}', get_blk(i))
```

(continues on next page)

(continued from previous page)

```
            setattr(self, f'cls_{i}', cls_predictor(idx_to_in_channels[i],
                                                    num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(idx_to_in_channels[i],
                                                      num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # Here `getattr(self, 'blk_%d' % i)` accesses `self.blk_i`
            X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                X, getattr(self, f'blk_{i}'), sizes[i], ratios[i],
                getattr(self, f'cls_{i}'), getattr(self, f'bbox_{i}'))
        anchors = torch.cat(anchors, dim=1)
        cls_preds = concat_preds(cls_preds)
        cls_preds = cls_preds.reshape(
            cls_preds.shape[0], -1, self.num_classes + 1)
        bbox_preds = concat_preds(bbox_preds)
        return anchors, cls_preds, bbox_preds
```

Chúng tôi tạo ra một ví dụ mô hình và sử dụng nó để thực hiện tuyên truyền chuyển tiếp trên một minibatch 256×256 hình ảnh `X`.

Như được hiển thị trước đó trong phần này, khối đầu tiên xuất bản đồ tính năng 32×32. Nhớ lại rằng khối lấy mẫu thứ hai đến thứ tư giảm một nửa chiều cao và chiều rộng và khối thứ năm sử dụng tổng hợp toàn cầu. Vì 4 hộp neo được tạo ra cho mỗi đơn vị dọc theo kích thước không gian của bản đồ tính năng, tại tất cả năm thang đo tổng cộng $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ hộp neo được tạo ra cho mỗi hình ảnh.

```
net = TinySSD(num_classes=1)
X = torch.zeros((32, 3, 256, 256))
anchors, cls_preds, bbox_preds = net(X)

print('output anchors:', anchors.shape)
print('output class preds:', cls_preds.shape)
print('output bbox preds:', bbox_preds.shape)
```

```
output anchors: torch.Size([1, 5444, 4])
output class preds: torch.Size([32, 5444, 2])
output bbox preds: torch.Size([32, 21776])
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```

14.7.2 Đào tạo

Bây giờ chúng tôi sẽ giải thích cách huấn luyện mô hình phát hiện multibox shot đơn để phát hiện đối tượng.

Đọc tập dữ liệu và khởi tạo mô hình

Để bắt đầu, chúng ta hãy đọc dữ liệu phát hiện chuối được mô tả trong Section 14.6.

```
batch_size = 32
train_iter, _ = d2l.load_data_bananas(batch_size)
```

```
read 1000 training examples
read 100 validation examples
```

Chỉ có một lớp trong bộ dữ liệu phát hiện chuối. Sau khi xác định mô hình, chúng ta cần phải khởi tạo các tham số của nó và xác định thuật toán tối ưu hoạt.

```
device, net = d2l.try_gpu(), TinySSD(num_classes=1)
trainer = torch.optim.SGD(net.parameters(), lr=0.2, weight_decay=5e-4)
```

Xác định mất và đánh giá chức năng

Phát hiện đối tượng có hai loại tổn thất. Sự mất mát đầu tiên liên quan đến các lớp của hộp neo: tính toán của nó có thể chỉ đơn giản là tái sử dụng hàm mất chéo entropy mà chúng ta sử dụng để phân loại hình ảnh. Tổn thất thứ hai liên quan đến bù đắp các hộp neo tích cực (không nền): đây là một vấn đề hồi quy. Tuy nhiên, đối với vấn đề hồi quy này, ở đây chúng tôi không sử dụng tổn thất bình phương được mô tả trong Section 4.1.3. Thay vào đó, chúng ta sử dụng mức mất định mức L_1, giá trị tuyệt đối của sự khác biệt giữa dự đoán và sự thật mặt đất. Biến mặt nạ `bbox_masks` lọc ra các hộp neo âm và các hộp neo bất hợp pháp (đệm) trong tính toán tổn thất. Cuối cùng, chúng tôi tổng hợp mất lớp hộp neo và tổn thất bù đắp hộp neo để có được chức năng mất mát cho mô hình.

```
cls_loss = nn.CrossEntropyLoss(reduction='none')
bbox_loss = nn.L1Loss(reduction='none')

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    batch_size, num_classes = cls_preds.shape[0], cls_preds.shape[2]
    cls = cls_loss(cls_preds.reshape(-1, num_classes),
                   cls_labels.reshape(-1)).reshape(batch_size, -1).mean(dim=1)
    bbox = bbox_loss(bbox_preds * bbox_masks,
                     bbox_labels * bbox_masks).mean(dim=1)
    return cls + bbox
```

Chúng ta có thể sử dụng độ chính xác để đánh giá kết quả phân loại. Do tổn thất định mức L_1 đã sử dụng cho các bù trừ, chúng tôi sử dụng lỗi tuyệt đối *trung bình để đánh giá các hộp giới hạn dự đoán. Những kết quả dự đoán này thu được từ các hộp neo được tạo ra và các bù đắp dự đoán cho chúng.

```
def cls_eval(cls_preds, cls_labels):
    # Because the class prediction results are on the final dimension,
    # `argmax` needs to specify this dimension
```

(continues on next page)

(continued from previous page)

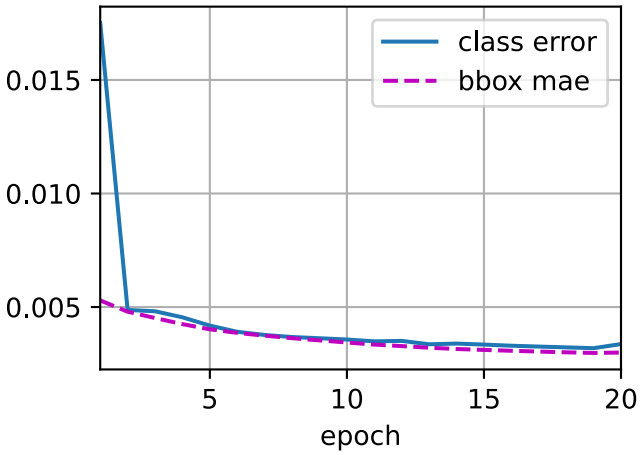
```
    return float((cls_preds.argmax(dim=-1).type(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((torch.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

Đào tạo mô hình

Khi đào tạo mô hình, chúng ta cần tạo ra các hộp neo đa quy mô (`anchors`) và dự đoán các lớp của chúng (`cls_preds`) và bù đắp (`bbox_preds`) trong tuyên truyền về phía trước. Sau đó, chúng tôi dán nhãn các lớp (`cls_labels`) và bù đắp (`bbox_labels`) của các hộp neo được tạo ra như vậy dựa trên thông tin nhãn `Y`. Cuối cùng, chúng ta tính toán hàm mất bằng cách sử dụng các giá trị dự đoán và dán nhãn của các lớp và bù đắp. Để triển khai ngắn gọn, đánh giá tập dữ liệu thử nghiệm được bỏ qua ở đây.

```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                        legend=['class error', 'bbox mae'])
net = net.to(device)
for epoch in range(num_epochs):
    # Sum of training accuracy, no. of examples in sum of training accuracy,
    # Sum of absolute error, no. of examples in sum of absolute error
    metric = d2l.Accumulator(4)
    net.train()
    for features, target in train_iter:
        timer.start()
        trainer.zero_grad()
        X, Y = features.to(device), target.to(device)
        # Generate multiscale anchor boxes and predict their classes and
        # offsets
        anchors, cls_preds, bbox_preds = net(X)
        # Label the classes and offsets of these anchor boxes
        bbox_labels, bbox_masks, cls_labels = d2l.multibox_target(anchors, Y)
        # Calculate the loss function using the predicted and labeled values
        # of the classes and offsets
        l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                      bbox_masks)
        l.mean().backward()
        trainer.step()
        metric.add(cls_eval(cls_preds, cls_labels), cls_labels.numel(),
                   bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                   bbox_labels.numel())
    cls_err, bbox_mae = 1 - metric[0] / metric[1], metric[2] / metric[3]
    animator.add(epoch + 1, (cls_err, bbox_mae))
print(f'class err {cls_err:.2e}, bbox mae {bbox_mae:.2e}')
print(f'{len(train_iter.dataset) / timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

```
class err 3.37e-03, bbox mae 3.00e-03
6317.1 examples/sec on cuda:0
```



14.7.3 Prediction

Trong quá trình dự đoán, mục tiêu là phát hiện tất cả các đối tượng quan tâm trên hình ảnh. Dưới đây chúng tôi đọc và thay đổi kích thước một hình ảnh thử nghiệm, chuyển đổi nó thành tensor bốn chiều được yêu cầu bởi các lớp phức tạp.

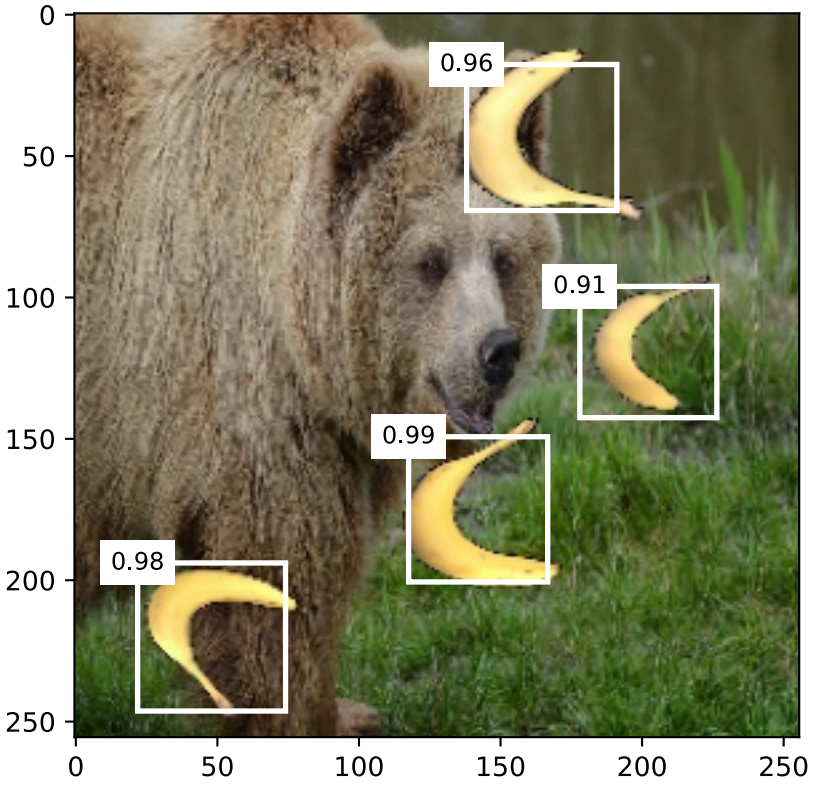
```
X = torchvision.io.read_image('../img/banana.jpg').unsqueeze(0).float()
img = X.squeeze(0).permute(1, 2, 0).long()
```

Sử dụng hàm `multibox_detection` bên dưới, các hộp giới hạn dự đoán được lấy từ các hộp neo và bù đắp dự đoán của chúng. Sau đó, ức chế không tối đa được sử dụng để loại bỏ các hộp giới hạn dự đoán tương tự.

```
def predict(X):
    net.eval()
    anchors, cls_preds, bbox_preds = net(X.to(device))
    cls_probs = F.softmax(cls_preds, dim=2).permute(0, 2, 1)
    output = d2l.multibox_detection(cls_probs, bbox_preds, anchors)
    idx = [i for i, row in enumerate(output[0]) if row[0] != -1]
    return output[0, idx]

output = predict(X)
```

Cuối cùng, chúng ta hiển thị tất cả các hộp giới hạn dự đoán với sự tự tin 0.9 trở lên làm đầu ra.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img)
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * torch.tensor((w, h, w, h), device=row.device)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output.cpu(), threshold=0.9)
```



14.7.4 Tóm tắt

- Phát hiện multibox shot đơn là một mô hình phát hiện đối tượng đa quy mô. Thông qua mạng cơ sở của nó và một số khối bản đồ tính năng đa quy mô, phát hiện multibox single shot tạo ra một số lượng khác nhau của các hộp neo với kích thước khác nhau, và phát hiện các đối tượng kích thước khác nhau bằng cách dự đoán các lớp và bù đắp của các hộp neo (do đó các hộp giới hạn).
- Khi đào tạo mô hình phát hiện multibox single shot, chức năng mất mát được tính toán dựa trên các giá trị dự đoán và gắn nhãn của các lớp và bù đắp hộp neo.

14.7.5 Bài tập

1. Bạn có thể cải thiện phát hiện multibox một lần bằng cách cải thiện chức năng mất mát? Ví dụ, thay thế mất tiêu chuẩn L_1 với tổn thất định mức L_1 trơn tru cho các bù đắp dự đoán. Hàm mất này sử dụng một hàm vuông khoảng 0 cho độ mịn, được điều khiển bởi siêu tham số σ:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases} \tag{14.7.1}$$

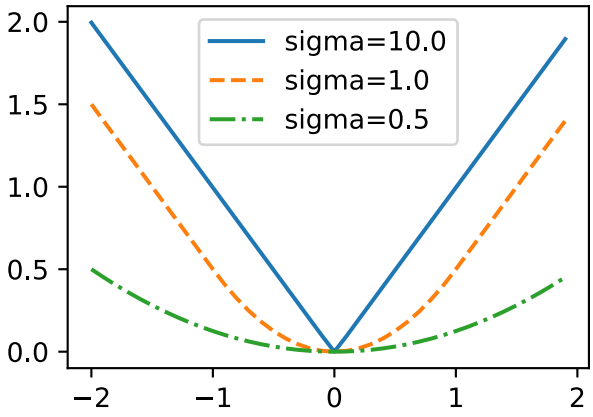
Khi σ rất lớn, tổn thất này tương tự như mức mất định mức L_1. Khi giá trị của nó nhỏ hơn, chức năng mất mát mượt mà hơn.

```
def smooth_l1(data, scalar):
    out = []
    for i in data:
        if abs(i) < 1 / (scalar ** 2):
```

(continues on next page)

(continued from previous page)

```
            out.append(((scalar * i) ** 2) / 2)
        else:
            out.append(abs(i) - 0.5 / (scalar ** 2))
    return torch.tensor(out)

sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = torch.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = smooth_l1(x, scalar=s)
    d2l.plt.plot(x, y, l, label='sigma=%.1f' % s)
d2l.plt.legend();
```

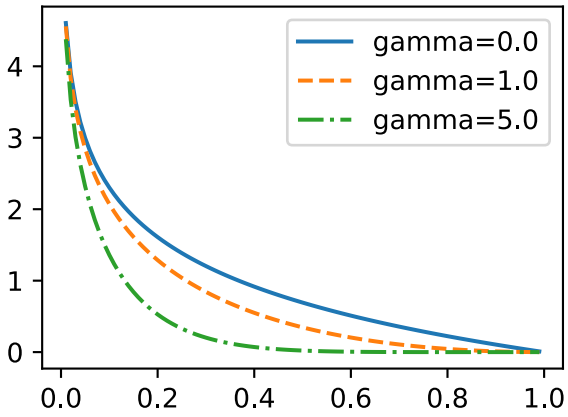


Bên cạnh đó, trong thí nghiệm, chúng tôi đã sử dụng mất chéo entropy cho dự đoán lớp: biểu thị bằng p_j xác suất dự đoán cho lớp chân lý mặt đất j, tổn thất chéo entropy là $-\log p_j$. Chúng ta cũng có thể sử dụng tổn thất tiêu cự (Lin et al., 2017): cho các siêu tham số $\gamma > 0$ và $\alpha > 0$, sự mất mát này được định nghĩa là:

$$-\alpha(1-p_j)^\gamma \log p_j. \tag{14.7.2}$$

Như chúng ta có thể thấy, tăng γ có thể làm giảm hiệu quả tổn thất tương đối cho các ví dụ được phân loại tốt (ví dụ, $p_j > 0.5$) để đào tạo có thể tập trung nhiều hơn vào những ví dụ khó khăn bị phân loại sai.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * torch.log(x)

x = torch.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x, focal_loss(gamma, x), l, label='gamma=%.1f' % gamma)
d2l.plt.legend();
```



2. Do hạn chế về không gian, chúng tôi đã bỏ qua một số chi tiết triển khai của mô hình phát hiện multibox một shot trong phần này. Bạn có thể cải thiện thêm mô hình trong các khía cạnh sau:
    1. Khi một đối tượng nhỏ hơn nhiều so với hình ảnh, mô hình có thể thay đổi kích thước hình ảnh đầu vào lớn hơn.
    2. Thường có một số lượng lớn các hộp neo âm. Để làm cho phân phối lớp cân bằng hơn, chúng ta có thể hạ mẫu các hộp neo âm.
    3. Trong chức năng mất, gán các siêu tham số trọng lượng khác nhau cho việc mất lớp và mất bù.
    4. Sử dụng các phương pháp khác để đánh giá mô hình phát hiện đối tượng, chẳng hạn như các phương pháp trong một shot multibox phát hiện giấy (Liu et al., 2016).

Discussions[177]

14.8 CNN dựa trên khu vực (R-CNN)

Bên cạnh việc phát hiện multibox shot đơn được mô tả trong Section 14.7, CNN dựa trên khu vực hoặc các khu vực có tính năng CNN (R-CNN) cũng là một trong nhiều phương pháp tiên phong trong việc áp dụng học sâu vào phát hiện đối tượng (Girshick et al., 2014). Trong phần này, chúng tôi sẽ giới thiệu R-CNN và một loạt các cải tiến của nó: R-CNN (Girshick, 2015) nhanh, R-CNN (Ren et al., 2015) nhanh hơn và mặt nạ R-CNN (He et al., 2017). Do không gian hạn chế, chúng tôi sẽ chỉ tập trung vào việc thiết kế các mô hình này.

14.8.1 R-CNN

R-CNN đầu tiên trích xuất nhiều (ví dụ: 2000) * khu vực đề xuất* từ hình ảnh đầu vào (ví dụ, hộp neo cũng có thể được coi là đề xuất khu vực), dán nhãn các lớp và hộp giới hạn của chúng (ví dụ: bù đắp). (Girshick et al., 2014) Sau đó, CNN được sử dụng để thực hiện tuyên truyền chuyển tiếp trên mỗi đề xuất khu vực để trích xuất của nó tính năng. Tiếp theo, các tính năng của mỗi đề xuất khu vực được sử dụng để dự đoán lớp và hộp giới hạn của đề xuất khu vực này.

[177] https://discuss.d2l.ai/t/1604

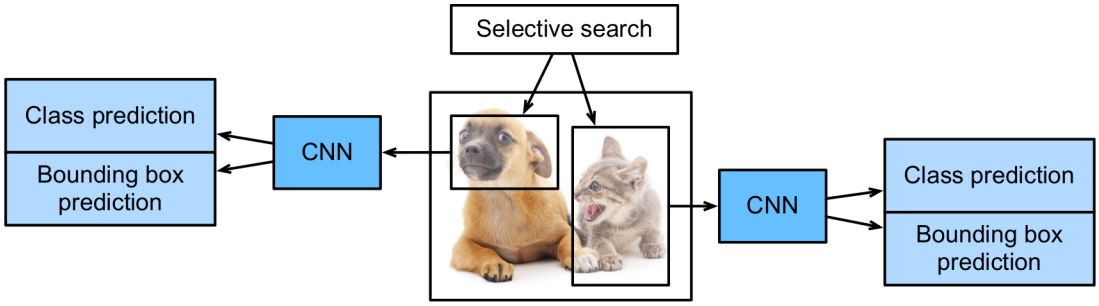


Fig. 14.8.1: The R-CNN model.

Fig. 14.8.1 cho thấy mô hình R-CNN. Cụ thể hơn, R-CNN bao gồm bốn bước sau:

1. Thực hiện * tìm kiếm chọn lọc* để trích xuất nhiều đề xuất khu vực chất lượng cao trên hình ảnh đầu vào (Uijlings et al., 2013). Những vùng đề xuất này thường được chọn ở nhiều thang đo với hình dạng và kích cỡ khác nhau. Mỗi đề xuất khu vực sẽ được dán nhãn với một lớp học và một hộp giới hạn sự thật mặt đất.
2. Chọn một CNN được đào tạo trước và cắt ngắn nó trước lớp đầu ra. Thay đổi kích thước mỗi đề xuất khu vực theo kích thước đầu vào theo yêu cầu của mạng và xuất các tính năng trích xuất cho đề xuất khu vực thông qua tuyên truyền chuyển tiếp.
3. Lấy các tính năng trích xuất và lớp được dán nhãn của mỗi đề xuất khu vực làm ví dụ. Đào tạo nhiều máy vector hỗ trợ để phân loại các đối tượng, trong đó mỗi máy vector hỗ trợ riêng xác định xem ví dụ có chứa một lớp cụ thể hay không.
4. Lấy các tính năng được trích xuất và hộp giới hạn được dán nhãn của mỗi đề xuất khu vực làm ví dụ. Đào tạo một mô hình hồi quy tuyến tính để dự đoán hộp giới hạn chân thất-chân lý.

Mặc dù mô hình R-CNN sử dụng CNN được đào tạo trước để trích xuất hiệu quả các tính năng hình ảnh, nhưng nó chậm. Hãy tưởng tượng rằng chúng tôi chọn hàng ngàn đề xuất khu vực từ một hình ảnh đầu vào duy nhất: điều này đòi hỏi hàng ngàn tuyên truyền chuyển tiếp CNN để thực hiện phát hiện đối tượng. Tải trọng điện toán lớn này làm cho việc sử dụng rộng rãi R-CNN trong các ứng dụng trong thế giới thực không khả thi.

14.8.2 R-CNN nhanh

Nút cổ chai hiệu suất chính của R-CNN nằm ở việc tuyên truyền chuyển tiếp CNN độc lập cho từng đề xuất khu vực, mà không chia sẻ tính toán. Vì những vùng này thường có sự chồng chéo, các trích xuất tính năng độc lập dẫn đến tính toán lặp đi lặp lại nhiều. Một trong những cải tiến lớn của * nhanh R-CNN* từ R-CNN là CNN chuyển tiếp CNN chỉ được thực hiện trên toàn bộ hình ảnh (Girshick, 2015).

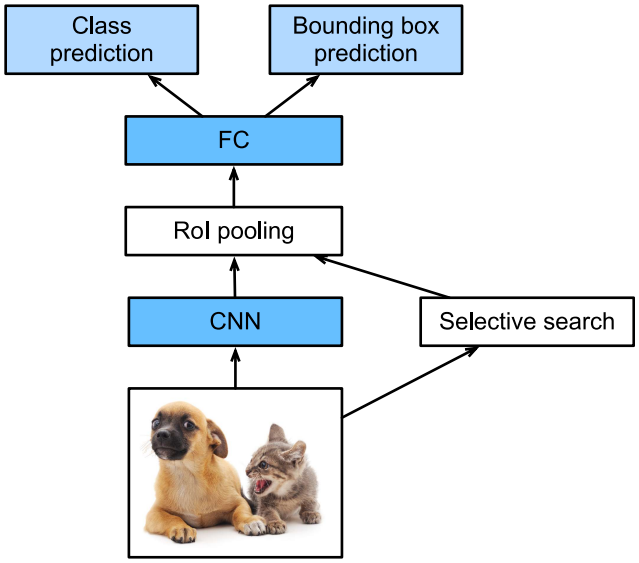


Fig. 14.8.2: The fast R-CNN model.

Fig. 14.8.2 mô tả mô hình R-CNN nhanh. Các tính toán chính của nó như sau:

1. So với R-CNN, trong R-CNN nhanh, đầu vào của CNN để trích xuất tính năng là toàn bộ hình ảnh, chứ không phải là các đề xuất khu vực riêng lẻ. Hơn nữa, CNN này là có thể đào tạo. Với một hình ảnh đầu vào, hãy để hình dạng của đầu ra CNN là $1 \times c \times h_1 \times w_1$.
2. Giả sử rằng tìm kiếm chọn lọc tạo ra n đề xuất khu vực. Các đề xuất khu vực này (có hình dạng khác nhau) đánh dấu các vùng được quan tâm (có hình dạng khác nhau) trên đầu ra CNN. Sau đó, các khu vực quan tâm hơn nữa trích xuất các tính năng của cùng một hình dạng (nói chiều cao h_2 và chiều rộng w_2 được chỉ định) để dễ dàng nối. Để đạt được điều này, R-CNN nhanh giới thiệu lớp cộp* khu vực quan tâm (ROI): các đề xuất đầu ra và khu vực CNN được nhập vào lớp này, xuất các tính năng nối của hình dạng $n \times c \times h_2 \times w_2$ được trích xuất thêm cho tất cả các đề xuất khu vực.
3. Sử dụng một lớp được kết nối hoàn toàn, biến đổi các tính năng nối thành một đầu ra của hình dạng $n \times d$, trong đó d phụ thuộc vào thiết kế mô hình.
4. Dự đoán lớp học và hộp giới hạn cho mỗi đề xuất khu vực n. Cụ thể hơn, trong lớp và dự đoán hộp giới hạn, biến đổi đầu ra lớp được kết nối hoàn toàn thành một đầu ra của hình dạng $n \times q$ (q là số lớp) và một đầu ra của hình dạng $n \times 4$, tương ứng. Dự đoán lớp sử dụng hồi quy softmax.

Khu vực của lớp tổng hợp quan tâm được đề xuất trong R-CNN nhanh khác với lớp tổng hợp được giới thiệu vào năm Section 7.5. Trong layer pooling, chúng ta gián tiếp kiểm soát hình dạng đầu ra bằng cách chỉ định kích thước của cửa sổ pooling, padding, và sải chân. Ngược lại, chúng ta có thể trực tiếp chỉ định hình dạng đầu ra trong khu vực của lớp tổng hợp quan tâm.

Ví dụ: chúng ta hãy chỉ định chiều cao và chiều rộng đầu ra cho mỗi khu vực là h_2 và w_2, tương ứng. Đối với bất kỳ khu vực nào của cửa sổ quan tâm của hình dạng $h \times w$, cửa sổ này được chia thành một lưới $h_2 \times w_2$ của subwindows, trong đó hình dạng của mỗi cửa sổ con là khoảng $(h/h_2) \times (w/w_2)$. Trong thực tế, chiều cao và chiều rộng của bất kỳ cửa sổ con nào sẽ được làm tròn và phần tử lớn nhất sẽ được sử dụng làm đầu ra của cửa sổ con. Do đó, khu vực của lớp tổng hợp quan tâm có thể trích xuất các tính năng có cùng hình dạng ngay cả khi các vùng quan tâm có hình dạng khác nhau.

Như một ví dụ minh họa, trong Fig. 14.8.3, khu vực quan tâm 3×3 phía trên bên trái được chọn trên một đầu vào 4×4. Đối với khu vực quan tâm này, chúng tôi sử dụng một 2×2 khu vực của lớp tổng hợp quan tâm để có được một sản lượng 2×2. Lưu ý rằng mỗi trong bốn phân chia chứa các phần tử 0, 1, 4 và 5 (5 là tối đa); 2 và 6 (6 là tối đa); 8 và 9 (9 là tối đa); và 10.

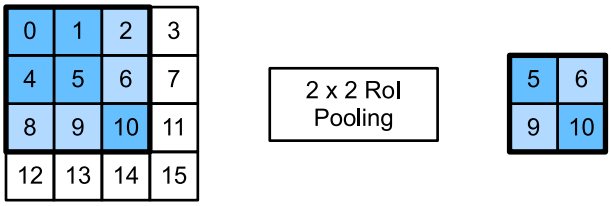


Fig. 14.8.3: A 2×2 region of interest pooling layer.

Dưới đây chúng tôi chứng minh tính toán của khu vực của lớp tổng hợp quan tâm. Giả sử rằng chiều cao và chiều rộng của các tính năng trích xuất CNN `X` đều là 4 và chỉ có một kênh duy nhất.

```
import torch
import torchvision

X = torch.arange(16.).reshape(1, 1, 4, 4)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Chúng ta hãy giả sử thêm rằng chiều cao và chiều rộng của hình ảnh đầu vào là cả 40 pixel và tìm kiếm chọn lọc tạo ra hai đề xuất khu vực trên hình ảnh này. Mỗi đề xuất vùng được thể hiện dưới dạng năm yếu tố: lớp đối tượng của nó tiếp theo là tọa độ (x, y) của các góc trên bên trái và dưới bên phải của nó.

```
rois = torch.Tensor([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Bởi vì chiều cao và chiều rộng của `X` là $1/10$ chiều cao và chiều rộng của hình ảnh đầu vào, các tọa độ của hai đề xuất khu vực được nhân với 0,1 theo đối số `spatial_scale` được chỉ định. Sau đó, hai khu vực quan tâm được đánh dấu trên `X` là `X[:, :, 0:3, 0:3]` và `X[:, :, 1:4, 0:4]`, tương ứng. Cuối cùng trong khu vực 2×2 thu hút sự quan tâm, mỗi khu vực quan tâm được chia thành một mạng lưới các cửa sổ phụ để trích xuất thêm các tính năng có cùng hình dạng 2×2.

```
torchvision.ops.roi_pool(X, rois, output_size=(2, 2), spatial_scale=0.1)
```

```
tensor([[[[ 5.,  6.],
          [ 9., 10.]]],


        [[[ 9., 11.],
          [13., 15.]]]])
```

14.8.3 R-CNN nhanh hơn

Để chính xác hơn trong việc phát hiện đối tượng, mô hình R-CNN nhanh thường phải tạo ra rất nhiều đề xuất khu vực trong tìm kiếm chọn lọc. Để giảm đề xuất khu vực mà không mất độ chính xác, R-CNN* nhanh hơn đề xuất thay thế tìm kiếm chọn lọc bằng mạng đề xuất khu vực*:cite:`Ren.He.Girshick.ea.2015`.

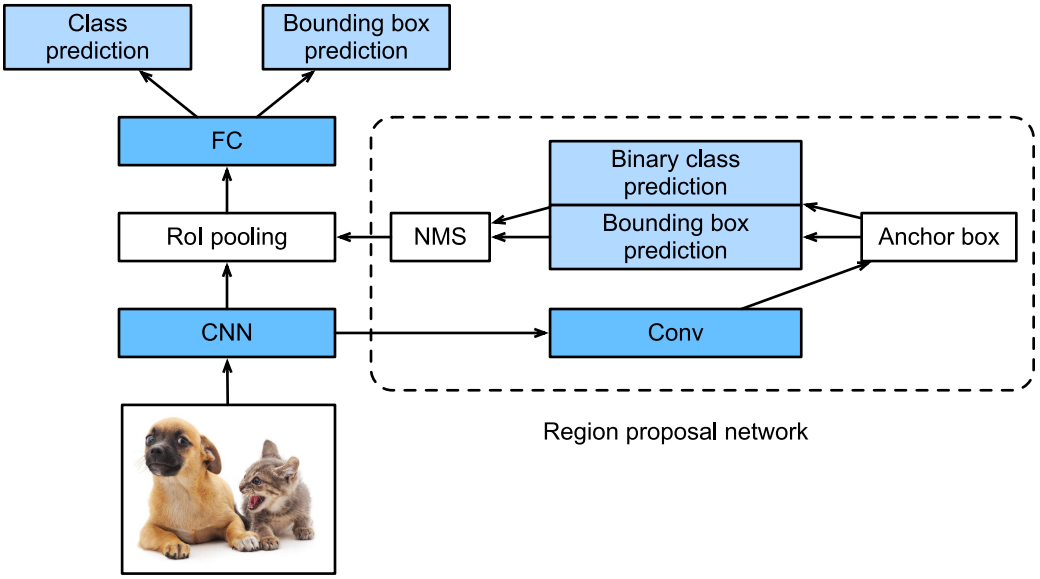


Fig. 14.8.4: The faster R-CNN model.

Fig. 14.8.4 cho thấy mô hình R-CNN nhanh hơn. So với R-CNN nhanh, R-CNN nhanh hơn chỉ thay đổi phương pháp đề xuất khu vực từ tìm kiếm chọn lọc sang mạng đề xuất khu vực. Phần còn lại của mô hình vẫn không thay đổi. Mạng đề xuất khu vực hoạt động theo các bước sau:

1. Sử dụng lớp ghép 3×3 với lớp đệm 1 để chuyển đổi đầu ra CNN sang đầu ra mới với c kênh. Bằng cách này, mỗi đơn vị dọc theo kích thước không gian của các bản đồ tính năng trích xuất CNN được một vector tính năng mới có chiều dài c.
2. Tập trung vào từng điểm ảnh của bản đồ tính năng, tạo ra nhiều hộp neo có tỷ lệ và tỷ lệ khung hình khác nhau và dán nhãn chúng.
3. Sử dụng vector tính năng length c ở giữa mỗi hộp neo, dự đoán lớp nhị phân (nền hoặc đối tượng) và hộp giới hạn cho hộp neo này.
4. Hãy xem xét những hộp giới hạn dự đoán có lớp dự đoán là các đối tượng. Loại bỏ kết quả chồng chéo bằng cách sử dụng ức chế không tối đa. Các hộp giới hạn dự đoán còn lại cho các đối tượng là các đề xuất khu vực theo yêu cầu của khu vực của lớp tổng hợp quan tâm.

Điều đáng chú ý là, là một phần của mô hình R-CNN nhanh hơn, mạng lưới đề xuất khu vực được cùng đào tạo với phần còn lại của mô hình. Nói cách khác, hàm khách quan của R-CNN nhanh hơn bao gồm không chỉ dự đoán lớp và hộp giới hạn trong phát hiện đối tượng, mà còn là lớp nhị phân và dự đoán hộp giới hạn của các hộp neo trong mạng đề xuất khu vực. Kết quả của việc đào tạo từ đầu đến cuối, mạng đề xuất khu vực học cách tạo các đề xuất khu vực chất lượng cao, để duy trì chính xác trong việc phát hiện đối tượng với số lượng đề xuất khu vực được học từ dữ liệu giảm.

14.8.4 Mặt nạ R-CNN

Trong tập dữ liệu đào tạo, nếu vị trí cấp pixel của đối tượng cũng được dán nhãn trên hình ảnh, *mask R-CNN* có thể tận dụng hiệu quả các nhãn chi tiết như vậy để cải thiện hơn nữa độ chính xác của phát hiện đối tượng (He et al., 2017).

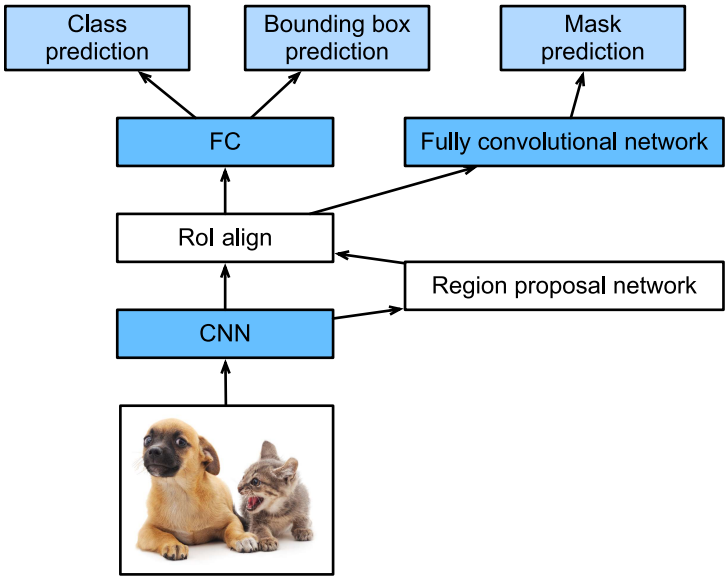


Fig. 14.8.5: The mask R-CNN model.

Như thể hiện trong Fig. 14.8.5, mặt nạ R-CNN được sửa đổi dựa trên R-CNN nhanh hơn. Cụ thể, mặt nạ R-CNN thay thế khu vực của lớp tổng hợp quan tâm bằng *khu vực quan tâm (ROI) sắp xếp* lớp. Vùng liên kết mối quan tâm này sử dụng nội suy song tuyến để bảo tồn thông tin không gian trên các bản đồ tính năng, phù hợp hơn cho dự đoán mức pixel. Đầu ra của lớp này chứa các bản đồ tính năng có cùng hình dạng cho tất cả các vùng quan tâm. Chúng được sử dụng để dự đoán không chỉ lớp và hộp giới hạn cho mỗi khu vực quan tâm, mà còn là vị trí cấp pixel của đối tượng thông qua một mạng phức tạp hoàn toàn bổ sung. Thông tin chi tiết về việc sử dụng mạng phức tạp hoàn toàn để dự đoán ngữ nghĩa cấp pixel của một hình ảnh sẽ được cung cấp trong các phần tiếp theo của chương này.

14.8.5 Tóm tắt

- R-CNN trích xuất nhiều đề xuất khu vực từ hình ảnh đầu vào, sử dụng CNN để thực hiện tuyên truyền chuyển tiếp trên mỗi đề xuất khu vực để trích xuất các tính năng của nó, sau đó sử dụng các tính năng này để dự đoán lớp và hộp giới hạn của đề xuất khu vực này.
- Một trong những cải tiến lớn của R-CNN nhanh từ R-CNN là việc truyền chuyển tiếp CNN chỉ được thực hiện trên toàn bộ hình ảnh. Nó cũng giới thiệu khu vực của lớp tổng hợp quan tâm, do đó các tính năng của cùng một hình dạng có thể được chiết xuất thêm cho các khu vực quan tâm có hình dạng khác nhau.
- R-CNN nhanh hơn thay thế tìm kiếm chọn lọc được sử dụng trong R-CNN nhanh bằng mạng lưới đề xuất khu vực được đào tạo chung, để cái trước có thể giữ chính xác trong việc phát hiện đối tượng với số lượng đề xuất khu vực giảm.
- Dựa trên R-CNN nhanh hơn, mặt nạ R-CNN cũng giới thiệu một mạng lưới phức tạp hoàn toàn, để tận dụng các nhãn cấp pixel để cải thiện hơn nữa độ chính xác của việc phát hiện đối tượng.

14.8.6 Bài tập

1. Chúng ta có thể đóng khung phát hiện đối tượng như một vấn đề hồi quy duy nhất, chẳng hạn như dự đoán các hộp giới hạn và xác suất lớp không? Bạn có thể tham khảo thiết kế của mẫu YOLO [Redmon.Divvala.Girshick.ea.2016].
2. So sánh phát hiện multibox shot đơn với các phương pháp được giới thiệu trong phần này. Sự khác biệt lớn của họ là gì? Bạn có thể tham khảo Hình 2 của [Zhao.Zheng.Xu.ea.2019].

Discussions[178]

14.9 Phân đoạn ngữ nghĩa và tập dữ liệu

Khi thảo luận về các tác vụ phát hiện đối tượng trong Section 14.3—Section 14.8, các hộp giới hạn hình chữ nhật được sử dụng để dán nhãn và dự đoán các đối tượng trong ảnh. Phần này sẽ thảo luận về vấn đề phân đoạn *ngữ nghĩa *, tập trung vào cách chia hình ảnh thành các vùng thuộc các lớp ngữ nghĩa khác nhau. Khác với phát hiện đối tượng, phân đoạn ngữ nghĩa nhận ra và hiểu những gì có trong hình ảnh ở cấp độ pixel: ghi nhãn và dự đoán các vùng ngữ nghĩa của nó ở mức pixel. Fig. 14.9.1 hiển thị nhãn của chó, mèo và nền của hình ảnh trong phân khúc ngữ nghĩa. So với trong phát hiện đối tượng, các đường viền cấp pixel được dán nhãn trong phân đoạn ngữ nghĩa rõ ràng là hạt mịn hơn.

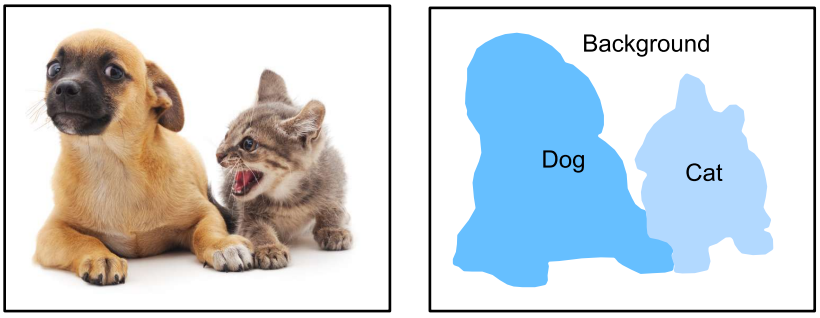


Fig. 14.9.1: Labels of the dog, cat, and background of the image in semantic segmentation.

14.9.1 Phân đoạn hình ảnh và Phân đoạn phiên bản

Ngoài ra còn có hai nhiệm vụ quan trọng trong lĩnh vực thị giác máy tính tương tự như phân đoạn ngữ nghĩa, đó là phân đoạn hình ảnh và phân đoạn phiên bản. Chúng tôi sẽ phân biệt ngắn gọn chúng với phân khúc ngữ nghĩa như sau.

- *Phân khúc hình ảnh* chia một hình ảnh thành nhiều vùng cấu thành. Các phương pháp cho loại vấn đề này thường sử dụng mối tương quan giữa các pixel trong hình ảnh. Nó không cần thông tin nhãn về pixel hình ảnh trong quá trình đào tạo và nó không thể đảm bảo rằng các vùng được phân đoạn sẽ có ngữ nghĩa mà chúng tôi hy vọng sẽ có được trong quá trình dự đoán. Chụp ảnh trong Fig. 14.9.1 làm đầu vào, phân đoạn hình ảnh có thể chia chó thành hai vùng: một vùng che miệng và mắt chủ yếu là màu đen, và phần còn lại bao phủ phần còn lại của cơ thể chủ yếu là màu vàng.
- – Phân khúc Instance* còn được gọi là * phát hiện và phân đoạn đồng thời*. Nó nghiên cứu làm thế nào để nhận ra các vùng cấp pixel của mỗi đối tượng trong một hình ảnh. Khác với phân đoạn ngữ nghĩa, phân đoạn phiên bản cần phân biệt không chỉ ngữ nghĩa, mà còn các trường hợp đối

[178] https://discuss.d2l.ai/t/1409

tượng khác nhau. Ví dụ, nếu có hai chú chó trong hình ảnh, phân đoạn ví dụ cần phân biệt một pixel thuộc về cái nào trong hai chú chó.

14.9.2 Tập dữ liệu phân đoạn ngữ nghĩa Pascal VOC2012

Trên tập dữ liệu phân đoạn ngữ nghĩa quan trọng nhất là Pascal VOC2012[179]. Sau đây, chúng ta sẽ xem tập dữ liệu này.

```
%matplotlib inline
import os
import torch
import torchvision
from d2l import torch as d2l
```

Tệp tar của tập dữ liệu khoảng 2 GB, vì vậy có thể mất một thời gian để tải xuống tệp. Tập dữ liệu được trích xuất nằm ở `../data/VOCdevkit/VOC2012`.

```
#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                           '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
```

```
Downloading ../data/VOCtrainval_11-May-2012.tar from http://d2l-data.s3-
↪accelerate.amazonaws.com/VOCtrainval_11-May-2012.tar...
```

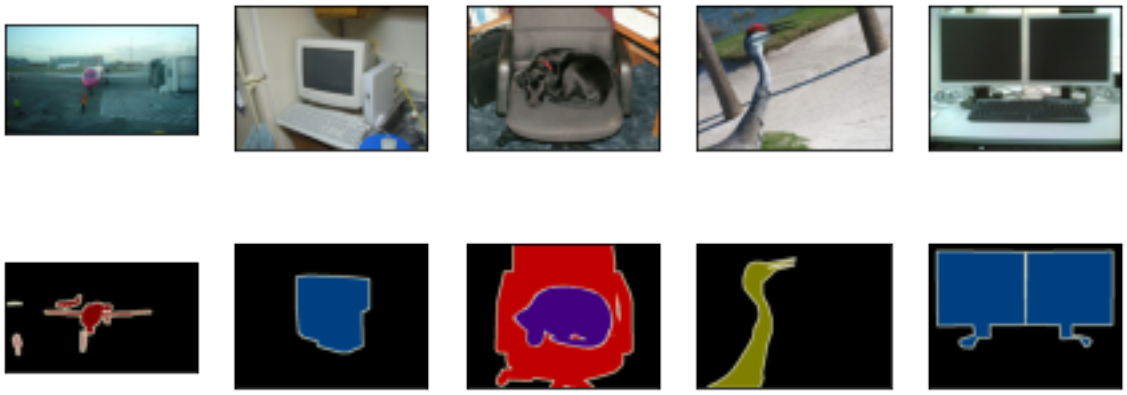
Sau khi nhập đường dẫn `../data/VOCdevkit/VOC2012`, chúng ta có thể thấy các thành phần khác nhau của tập dữ liệu. Đường dẫn `ImageSets/Segmentation` chứa các tệp văn bản chỉ định các mẫu đào tạo và thử nghiệm, trong khi các đường dẫn `JPEGImages` và `SegmentationClass` lưu trữ hình ảnh đầu vào và nhãn cho mỗi ví dụ tương ứng. Nhãn ở đây cũng ở định dạng hình ảnh, có cùng kích thước với hình ảnh đầu vào được dán nhãn của nó. Bên cạnh đó, các pixel có cùng màu trong bất kỳ hình ảnh nhãn nào thuộc cùng một lớp ngữ nghĩa. Sau đây xác định hàm `read_voc_images` thành đọc tất cả các hình ảnh đầu vào và nhãn vào bộ nhớ.

```
#@save
def read_voc_images(voc_dir, is_train=True):
    """Read all VOC feature and label images."""
    txt_fname = os.path.join(voc_dir, 'ImageSets', 'Segmentation',
                             'train.txt' if is_train else 'val.txt')
    mode = torchvision.io.image.ImageReadMode.RGB
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(torchvision.io.read_image(os.path.join(
            voc_dir, 'SegmentationClass' ,f'{fname}.png'), mode))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)
```

[179] http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

Chúng tôi vẽ năm hình ảnh đầu vào đầu tiên và nhãn của chúng. Trong hình ảnh nhãn, màu trắng và đen đại diện cho đường viền và nền tương ứng, trong khi các màu khác tương ứng với các lớp khác nhau.

```
n = 5
imgs = train_features[0:n] + train_labels[0:n]
imgs = [img.permute(1,2,0) for img in imgs]
d2l.show_images(imgs, 2, n);
```



Tiếp theo, chúng ta liệt kê các giá trị màu RGB và tên lớp cho tất cả các nhãn trong tập dữ liệu này.

```
#@save
VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                [0, 64, 128]]

#@save
VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
               'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
               'diningtable', 'dog', 'horse', 'motorbike', 'person',
               'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

Với hai hằng số được định nghĩa ở trên, chúng ta có thể thuận tiện tìm chỉ mục lớp cho mỗi pixel trong một nhãy. Chúng tôi xác định hàm `voc_colormap2label` để xây dựng ánh xạ từ các giá trị màu RGB ở trên đến các chỉ số lớp và hàm `voc_label_indices` để ánh xạ bất kỳ giá trị RGB nào với chỉ số lớp của chúng trong bộ dữ liệu Pascal VOC2012 này.

```
#@save
def voc_colormap2label():
    """Build the mapping from RGB to class indices for VOC labels."""
    colormap2label = torch.zeros(256 ** 3, dtype=torch.long)
    for i, colormap in enumerate(VOC_COLORMAP):
        colormap2label[
            (colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i
    return colormap2label

#@save
def voc_label_indices(colormap, colormap2label):
```

(continues on next page)

(continued from previous page)

```
    """Map any RGB values in VOC labels to their class indices."""
    colormap = colormap.permute(1, 2, 0).numpy().astype('int32')
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

Ví dụ, trong ảnh ví dụ đầu tiên, chỉ mục lớp cho phần trước của máy bay là 1, trong khi chỉ mục nền là 0.

```
y = voc_label_indices(train_labels[0], voc_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

```
(tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
         [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
         [0, 0, 0, 0, 0, 0, 0, 0, 1, 1]]),
 'aeroplane')
```

Xử lý sơ bộ dữ liệu

Trong các thí nghiệm trước đó như trong Section 8.1—Section 8.4, hình ảnh được thay đổi lại để phù hợp với hình dạng đầu vào yêu cầu của mô hình. Tuy nhiên, trong phân đoạn ngữ nghĩa, làm như vậy đòi hỏi phải thay đổi lại các lớp pixel dự đoán trở lại hình dạng ban đầu của hình ảnh đầu vào. Việc tái cặn như vậy có thể không chính xác, đặc biệt là đối với các vùng được phân đoạn với các lớp khác nhau. Để tránh sự cố này, chúng tôi cắt hình ảnh thành một hình dạng *fixed* thay vì rescaling. Cụ thể, sử dụng cắt xén ngẫu nhiên từ nâng hình ảnh, chúng ta cắt cùng một vùng của hình ảnh đầu vào và nhãn.

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop both feature and label images."""
    rect = torchvision.transforms.RandomCrop.get_params(
        feature, (height, width))
    feature = torchvision.transforms.functional.crop(feature, *rect)
    label = torchvision.transforms.functional.crop(label, *rect)
    return feature, label
```

```
imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)

imgs = [img.permute(1, 2, 0) for img in imgs]
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```

Custom Semantic Segmentation Dataset Class

Chúng tôi xác định một tập dữ liệu phân đoạn ngữ nghĩa tùy chỉnh lớp `VOCSegDataset` bằng cách kế thừa lớp `Dataset` được cung cấp bởi các API cấp cao. Bằng cách thực hiện hàm `__getitem__`, chúng ta có thể tùy ý truy cập vào hình ảnh đầu vào được lập chỉ mục là `idx` trong tập dữ liệu và chỉ số lớp của mỗi pixel trong hình ảnh này. Vì một số hình ảnh trong tập dữ liệu có kích thước nhỏ hơn kích thước đầu ra của cắt ngẫu nhiên, các ví dụ này được lọc ra bởi một hàm `filter` tùy chỉnh. Ngoài ra, chúng tôi cũng xác định hàm `normalize_image` để chuẩn hóa các giá trị của ba kênh RGB của hình ảnh đầu vào.

```
#@save
class VOCSegDataset(torch.utils.data.Dataset):
    """A customized dataset to load the VOC dataset."""

    def __init__(self, is_train, crop_size, voc_dir):
        self.transform = torchvision.transforms.Normalize(
            mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        self.crop_size = crop_size
        features, labels = read_voc_images(voc_dir, is_train=is_train)
        self.features = [self.normalize_image(feature)
                         for feature in self.filter(features)]
        self.labels = self.filter(labels)
        self.colormap2label = voc_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return self.transform(img.float() / 255)

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[1] >= self.crop_size[0] and
            img.shape[2] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                       *self.crop_size)
        return (feature, voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

Đọc dữ liệu

Chúng tôi sử dụng lớp `VOCSegDataset` tùy chỉnh để tạo các phiên bản của bộ đào tạo và bộ kiểm tra, tương ứng. Giả sử rằng chúng ta chỉ định rằng hình dạng đầu ra của hình ảnh được cắt ngẫu nhiên là 320×480. Dưới đây chúng ta có thể xem số ví dụ được giữ lại trong bộ đào tạo và bộ kiểm tra.

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

```
read 1114 examples
read 1078 examples
```

Đặt kích thước lô thành 64, chúng tôi xác định bộ lặp dữ liệu cho bộ đào tạo. Hãy để chúng tôi in hình dạng của minibatch đầu tiên. Khác với phân loại hình ảnh hoặc phát hiện đối tượng, nhãn ở đây là hàng chục ba chiều.

```
batch_size = 64
train_iter = torch.utils.data.DataLoader(voc_train, batch_size, shuffle=True,
                                         drop_last=True,
                                         num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break
```

```
torch.Size([64, 3, 320, 480])
torch.Size([64, 320, 480])
```

Putting All Things Together

Cuối cùng, chúng tôi xác định hàm `load_data_voc` sau để tải xuống và đọc tập dữ liệu phân đoạn ngữ nghĩa Pascal VOC2012. Nó trả về bộ lặp dữ liệu cho cả tập dữ liệu đào tạo và kiểm tra.

```
#@save
def load_data_voc(batch_size, crop_size):
    """Load the VOC semantic segmentation dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = torch.utils.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, drop_last=True, num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        drop_last=True, num_workers=num_workers)
    return train_iter, test_iter
```

14.9.3 Tóm tắt

- Phân đoạn ngữ nghĩa nhận ra và hiểu những gì trong một hình ảnh ở mức pixel bằng cách chia hình ảnh thành các vùng thuộc các lớp ngữ nghĩa khác nhau.
- Trên của tập dữ liệu phân khúc ngữ nghĩa quan trọng nhất là Pascal VOC2012.
- Trong phân đoạn ngữ nghĩa, vì hình ảnh đầu vào và nhãn tương ứng một-một trên pixel, hình ảnh đầu vào được cắt ngẫu nhiên thành một hình dạng cố định chứ không phải rescaled.

14.9.4 Bài tập

1. Làm thế nào phân khúc ngữ nghĩa có thể được áp dụng trong các phương tiện tự trị và chẩn đoán hình ảnh y tế? Bạn có thể nghĩ về các ứng dụng khác?
2. Nhớ lại các mô tả về tăng cường dữ liệu trong Section 14.1. Phương pháp nâng hình ảnh nào được sử dụng trong phân loại hình ảnh sẽ không khả thi được áp dụng trong phân khúc ngữ nghĩa?

Discussions[180]

14.10 Chuyển đổi Convolution

Các lớp CNN chúng ta đã thấy cho đến nay, chẳng hạn như các lớp phức tạp (Section 7.2) và các lớp tổng hợp (Section 7.5), thường làm giảm (downsample) kích thước không gian (chiều cao và chiều rộng) của đầu vào, hoặc giữ chúng không thay đổi. Trong phân đoạn ngữ nghĩa phân loại ở cấp pixel, sẽ thuận tiện nếu kích thước không gian của đầu vào và đầu ra giống nhau. Ví dụ: kích thước kênh tại một điểm ảnh đầu ra có thể giữ kết quả phân loại cho pixel đầu vào ở cùng một vị trí không gian.

Để đạt được điều này, đặc biệt là sau khi các kích thước không gian bị giảm bởi các lớp CNN, chúng ta có thể sử dụng một loại lớp CNN khác có thể tăng (upsample) kích thước không gian của bản đồ tính năng trung gian. Trong phần này, chúng tôi sẽ giới thiệu *chuyển đổi*, mà còn được gọi là * phân số strided convolution* (Dumoulin & Visin, 2016), để đảo ngược các hoạt động lấy mẫu xuống bằng sự covolution.

```
import torch
from torch import nn
from d2l import torch as d2l
```

14.10.1 Hoạt động cơ bản

Bỏ qua các kênh bây giờ, chúng ta hãy bắt đầu với hoạt động phức tạp chuyển tiếp cơ bản với sải chân 1 và không có đệm. Giả sử rằng chúng ta được đưa ra một tensor đầu vào $n_h \times n_w$ và một hạt nhân $k_h \times k_w$. Trượt cửa sổ hạt nhân với sải chân 1 cho n_w lần trong mỗi hàng và n_h lần trong mỗi cột mang lại tổng cộng $n_h n_w$ kết quả trung gian. Mỗi kết quả trung gian là một tensor $(n_h + k_h - 1) \times (n_w + k_w - 1)$ được khởi tạo dưới dạng số không. Để tính từng tensor trung gian, mỗi phần tử trong tensor đầu vào được nhân với hạt nhân sao cho tensor $k_h \times k_w$ kết quả thay thế một phần trong mỗi tensor trung gian. Lưu ý rằng vị trí của phần thay thế trong mỗi tensor trung gian tương ứng với vị trí của phần tử trong tensor đầu vào được sử dụng cho tính toán. Cuối cùng, tất cả các kết quả trung gian được tóm tắt để tạo ra đầu ra.

[180] https://discuss.d2l.ai/t/1480

Ví dụ, Fig. 14.10.1 minh họa cách chuyển đổi phức tạp với một hạt nhân 2×2 được tính toán cho một tensor đầu vào 2×2.

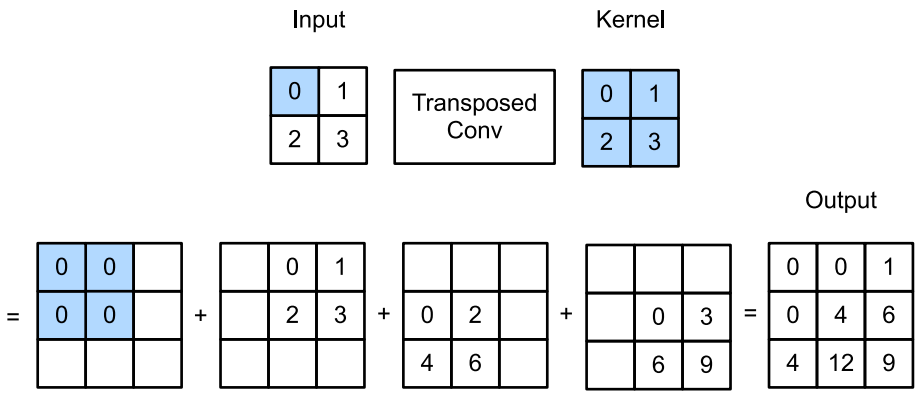


Fig. 14.10.1: Transposed convolution with a 2×2 kernel. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation.

Chúng ta có thể triển khai hoạt động chuyển tiếp cơ bản này `trans_conv` cho một ma trận đầu vào `X` và một ma trận hạt nhân `K`.

```
def trans_conv(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] + h - 1, X.shape[1] + w - 1))
    for i in range(X.shape[0]):
        for j in range(X.shape[1]):
            Y[i: i + h, j: j + w] += X[i, j] * K
    return Y
```

Trái ngược với sự phức tạp thông thường (trong Section 7.2) mà * giảm các yếu tố đầu vào thông qua hạt nhân, sự biến đổi chuyển đổi ***phát sóng** * yếu tố đầu vào thông qua hạt nhân, do đó tạo ra một đầu ra lớn hơn đầu vào. Chúng ta có thể xây dựng tensor đầu vào `X` và tensor kernel `K` từ Fig. 14.10.1 đến xác nhận đầu ra của việc triển khai ở trên của hoạt động phức tạp chuyển tiếp hai chiều cơ bản.

```
X = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
K = torch.tensor([[0.0, 1.0], [2.0, 3.0]])
trans_conv(X, K)
```

```
tensor([[ 0.,  0.,  1.],
        [ 0.,  4.,  6.],
        [ 4., 12.,  9.]])
```

Ngoài ra, khi đầu vào `X` và hạt nhân `K` đều là hàng chục bốn chiều, chúng ta có thể sử dụng API cấp cao để có được kết quả tương tự.

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[ 0.,  0.,  1.],
          [ 0.,  4.,  6.],
          [ 4., 12.,  9.]]]], grad_fn=<ConvolutionBackward0>)
```

14.10.2 Đệm, sải bước và nhiều kênh

Khác với trong sự phức tạp thông thường nơi đệm được áp dụng cho đầu vào, nó được áp dụng cho đầu ra trong quá trình chuyển đổi. Ví dụ: khi chỉ định số đệm ở hai bên của chiều cao và chiều rộng là 1, các hàng và cột đầu tiên và cuối cùng sẽ bị xóa khỏi đầu ra phức tạp được chuyển đổi.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, padding=1, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[4.]]]], grad_fn=<ConvolutionBackward0>)
```

Trong quá trình chuyển đổi, các bước tiến được chỉ định cho các kết quả trung gian (do đó đầu ra), không phải cho đầu vào. Sử dụng cùng một đầu vào và kernel tenors từ Fig. 14.10.1, thay đổi sải chân từ 1 đến 2 làm tăng cả chiều cao và trọng lượng của hàng chục trung gian, do đó tensor đầu ra trong Fig. 14.10.2.

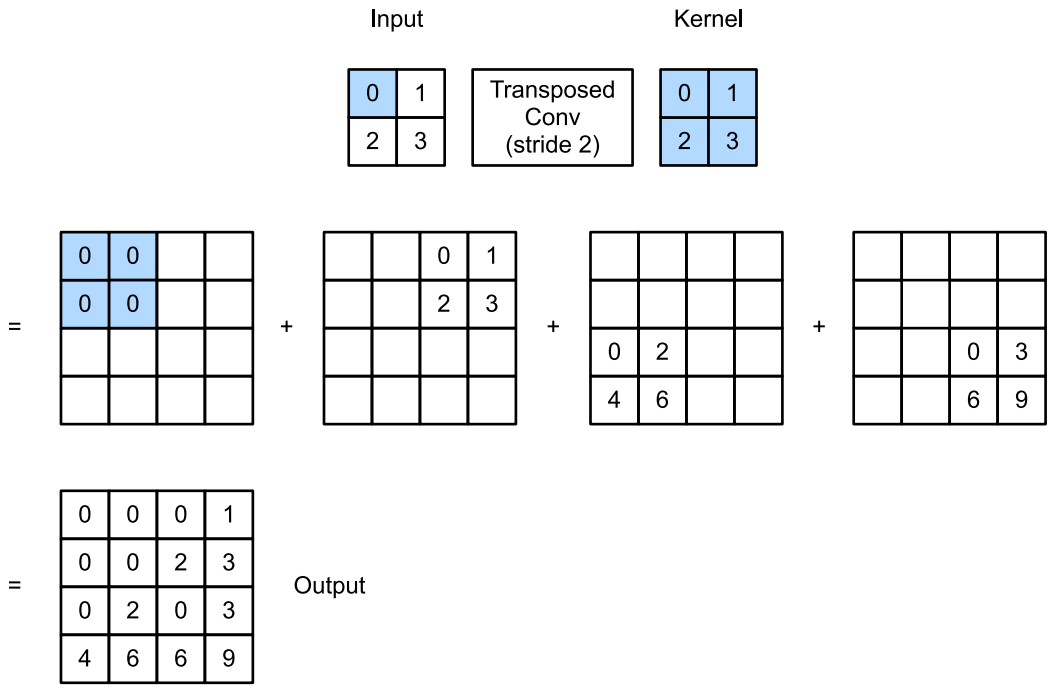


Fig. 14.10.2: Transposed convolution with a 2×2 kernel with stride of 2. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation.

Đoạn mã sau đây có thể xác nhận đầu ra phức tạp transposed cho sải chân của 2 trong Fig. 14.10.2.

```
tconv = nn.ConvTranspose2d(1, 1, kernel_size=2, stride=2, bias=False)
tconv.weight.data = K
tconv(X)
```

```
tensor([[[[0., 0., 0., 1.],
          [0., 0., 2., 3.],
          [0., 2., 0., 3.],
          [4., 6., 6., 9.]]]], grad_fn=<ConvolutionBackward0>)
```

Đối với nhiều kênh đầu vào và đầu ra, sự tích tụ được chuyển tiếp hoạt động theo cách tương tự như sự phức tạp thông thường. Giả sử rằng đầu vào có c_i các kênh và sự phức tạp transposed gán một tensor hạt nhân $k_h \times k_w$ cho mỗi kênh đầu vào. Khi nhiều kênh đầu ra được chỉ định, chúng tôi sẽ có hạt nhân $c_i \times k_h \times k_w$ cho mỗi kênh đầu ra.

Như trong tất cả, nếu chúng ta cung cấp X vào một lớp phức tạp f để xuất ra $\mathsf{Y} = f(\mathsf{X})$ và tạo ra một lớp tích hợp chuyển đổi g với các siêu tham số tương tự như f ngoại trừ số lượng kênh đầu ra là số kênh trong X, sau đó $g(Y)$ sẽ có hình dạng tương tự như X. Điều này có thể được minh họa trong ví dụ sau.

```
X = torch.rand(size=(1, 10, 16, 16))
conv = nn.Conv2d(10, 20, kernel_size=5, padding=2, stride=3)
tconv = nn.ConvTranspose2d(20, 10, kernel_size=5, padding=2, stride=3)
tconv(conv(X)).shape == X.shape
```

```
True
```

14.10.3 Kết nối với Ma trận Transposition

Sự phức tạp chuyển tiếp được đặt tên theo chuyển vị ma trận. Để giải thích, trước tiên chúng ta hãy xem làm thế nào để thực hiện các phức tạp bằng cách sử dụng phép nhân ma trận. Trong ví dụ dưới đây, chúng ta định nghĩa một 3×3 đầu vào `X` và một hạt nhân ghép 2×2 `K`, và sau đó sử dụng hàm `corr2d` để tính toán đầu ra phức tạp `Y`.

```
X = torch.arange(9.0).reshape(3, 3)
K = torch.tensor([[1.0, 2.0], [3.0, 4.0]])
Y = d2l.corr2d(X, K)
Y
```

```
tensor([[27., 37.],
        [57., 67.]])
```

Tiếp theo, chúng ta viết lại hạt nhân phức tạp `K` dưới dạng ma trận trọng lượng thưa `W` chứa rất nhiều số không. Hình dạng của ma trận trọng lượng là $(4, 9)$, trong đó các phần tử phi bằng không đến từ hạt nhân phức tạp `K`.

```
def kernel2matrix(K):
    k, W = torch.zeros(5), torch.zeros((4, 9))
    k[:2], k[3:5] = K[0, :], K[1, :]
    W[0, :5], W[1, 1:6], W[2, 3:8], W[3, 4:] = k, k, k, k
    return W

W = kernel2matrix(K)
W
```

```
tensor([[1., 2., 0., 3., 4., 0., 0., 0., 0.],
        [0., 1., 2., 0., 3., 4., 0., 0., 0.],
        [0., 0., 0., 1., 2., 0., 3., 4., 0.],
        [0., 0., 0., 0., 1., 2., 0., 3., 4.]])
```

Nối đầu vào `X` từng hàng để có được một vectơ có chiều dài 9. Sau đó, phép nhân ma trận của `W` và `X` vectơ hóa cho một vectơ có chiều dài 4. Sau khi định hình lại nó, chúng ta có thể thu được kết quả tương tự `Y` từ hoạt động phức tạp ban đầu ở trên: chúng ta chỉ thực hiện các phép phức tạp bằng cách sử dụng phép nhân ma trận.

```
Y == torch.matmul(W, X.reshape(-1)).reshape(2, 2)
```

```
tensor([[True, True],
        [True, True]])
```

Tương tự như vậy, chúng ta có thể thực hiện các phép chuyển đổi bằng cách sử dụng phép nhân ma trận. Trong ví dụ sau, chúng ta lấy đầu ra 2×2 `Y` từ sự phức tạp thông thường ở trên làm đầu vào cho sự phức tạp được chuyển tiếp. Để thực hiện thao tác này bằng cách nhân ma trận, chúng ta chỉ cần chuyển vị ma trận trọng lượng `W` với hình dạng mới $(9, 4)$.

```
Z = trans_conv(Y, K)
Z == torch.matmul(W.T, Y.reshape(-1)).reshape(3, 3)
```

```
tensor([[True, True, True],
        [True, True, True],
        [True, True, True]])
```

Cân nhắc việc thực hiện sự phức tạp bằng cách nhân ma trận. Với một vector đầu vào $\mathbf{x}$ và một ma trận trọng lượng $\mathbf{W}$, chức năng lan truyền chuyển tiếp của sự phức tạp có thể được thực hiện bằng cách nhân đầu vào của nó với ma trận trọng lượng và xuất ra một vector $\mathbf{y} = \mathbf{W}\mathbf{x}$. Kể từ khi backpropagation tuân theo quy tắc chuỗi và $\nabla_{\mathbf{x}}\mathbf{y} = \mathbf{W}^\top$, chức năng truyền ngược của sự phức tạp có thể được thực hiện bằng cách nhân đầu vào của nó với ma trận trọng lượng transposed $\mathbf{W}^\top$. Do đó, lớp tích lũy chuyển tiếp chỉ có thể trao đổi chức năng lan truyền về phía trước và chức năng lan truyền ngược của lớp ghép: các chức năng lan truyền về phía trước và lan truyền ngược của nó nhân vectơ đầu vào của chúng với $\mathbf{W}^\top$ và $\mathbf{W}$, tương ứng.

14.10.4 Tóm tắt

- Ngược lại với sự phức tạp thông thường làm giảm các phần tử đầu vào thông qua hạt nhân, sự phức tạp chuyển phát các phần tử đầu vào thông qua hạt nhân, từ đó tạo ra một đầu ra lớn hơn đầu vào.
- Nếu chúng ta cho X vào một lớp phức hợp f để xuất ra $\mathsf{Y} = f(\mathsf{X})$ và tạo ra một lớp tích hợp chuyển đổi g với các siêu tham số tương tự như f ngoại trừ số lượng kênh đầu ra là số kênh trong X, thì $g(Y)$ sẽ có hình dạng tương tự như X.
- Chúng ta có thể thực hiện các phức tạp bằng cách sử dụng phép nhân ma trận. Lớp ghép chuyển tiếp chỉ có thể trao đổi chức năng lan truyền về phía trước và chức năng lan truyền ngược của lớp ghép.

14.10.5 Bài tập

1. Trong Section 14.10.3, đầu vào covolution `X` và đầu ra phức tạp transposed `Z` có hình dạng tương tự. Họ có cùng giá trị không? Tại sao?
2. Có hiệu quả khi sử dụng phép nhân ma trận để thực hiện các phức tạp không? Tại sao?

Discussions[181]

14.11 Mạng kết nối hoàn toàn

Như đã thảo luận trong Section 14.9, phân đoạn ngữ nghĩa phân loại hình ảnh ở mức pixel. Một mạng phức tạp hoàn toàn (FCN) sử dụng mạng thần kinh phức tạp để chuyển đổi pixel hình ảnh thành các lớp pixel (Long et al., 2015). Không giống như các CNN mà chúng ta gặp phải trước đó để phân loại hình ảnh hoặc phát hiện đối tượng, một mạng phức tạp hoàn toàn biến đổi chiều cao và chiều rộng của bản đồ tính năng trung gian trở lại hình ảnh đầu vào: điều này đạt được bằng lớp biến đổi được giới thiệu trong Section 14.10. Do đó, đầu ra phân loại và hình ảnh đầu vào có sự tương ứng một-một ở mức pixel: kích thước kênh ở bất kỳ điểm ảnh đầu ra nào giữ kết quả phân loại cho pixel đầu vào ở cùng một vị trí không gian.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

14.11.1 Mô hình

Ở đây chúng tôi mô tả thiết kế cơ bản của mô hình mạng phức tạp hoàn toàn. Như thể hiện trong Fig. 14.11.1, mô hình này đầu tiên sử dụng CNN để trích xuất các tính năng hình ảnh, sau đó chuyển đổi số lượng kênh thành số lượng lớp thông qua lớp ghép 1×1 và cuối cùng biến đổi chiều cao và chiều rộng của bản đồ tính năng sang hình ảnh đầu vào thông qua sự phức tạp chuyển tiếp được giới thiệu trong Section 14.10. Do đó, đầu ra mô hình có cùng chiều cao và chiều rộng với hình ảnh đầu vào, trong đó kênh đầu ra chứa các lớp dự đoán cho pixel đầu vào ở cùng một vị trí không gian.

[181] https://discuss.d2l.ai/t/1450

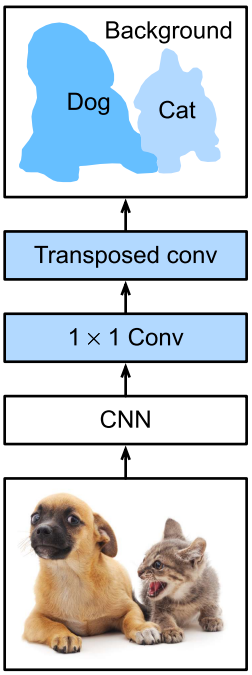


Fig. 14.11.1: Fully convolutional network.

Dưới đây, chúng ta sử dụng mô hình ResNet-18 được đào tạo trước trên bộ dữ liệu ImageNet để trích xuất các tính năng hình ảnh và biểu thị phiên bản model là `pretrained_net`. Vài lớp cuối cùng của mô hình này bao gồm một lớp tổng hợp trung bình toàn cầu và một lớp kết nối hoàn toàn: chúng không cần thiết trong mạng phức tạp hoàn toàn.

```
pretrained_net = torchvision.models.resnet18(pretrained=True)
list(pretrained_net.children())[-3:]
```

```
[Sequential(
   (0): BasicBlock(
     (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,␣
↪1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     (relu): ReLU(inplace=True)
     (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
     (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     (downsample): Sequential(
       (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
       (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
     )
   )
   (1): BasicBlock(
     (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
     (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
```

(continues on next page)

(continued from previous page)

```
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,␣
↪1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
↪running_stats=True)
  )
),
AdaptiveAvgPool2d(output_size=(1, 1)),
Linear(in_features=512, out_features=1000, bias=True)]
```

Tiếp theo, chúng ta tạo phiên bản mạng hoàn toàn phức tạp `net`. Nó sao chép tất cả các lớp được đào tạo trước trong ResNet-18 ngoại trừ lớp tổng hợp trung bình toàn cầu cuối cùng và lớp kết nối hoàn toàn gần đầu ra nhất.

```
net = nn.Sequential(*list(pretrained_net.children())[:-2])
```

Với một đầu vào với chiều cao và chiều rộng 320 và 480 tương ứng, sự lan truyền về phía trước của `net` làm giảm chiều cao và chiều rộng đầu vào xuống 1/32 của bản gốc, cụ thể là 10 và 15.

```
X = torch.rand(size=(1, 3, 320, 480))
net(X).shape
```

```
torch.Size([1, 512, 10, 15])
```

Tiếp theo, chúng ta sử dụng một lớp ghép 1×1 để chuyển đổi số kênh đầu ra thành số lớp (21) của tập dữ liệu Pascal VOC2012. Cuối cùng, chúng ta cần tăng chiều cao và chiều rộng của bản đồ tính năng lên 32 lần để thay đổi chúng trở lại chiều cao và chiều rộng của hình ảnh đầu vào. Nhớ lại cách tính hình dạng đầu ra của một lớp phức tạp trong Section 7.3. Kể từ $(320 - 64 + 16 \times 2 + 32)/32 = 10$ và $(480 - 64 + 16 \times 2 + 32)/32 = 15$, chúng tôi xây dựng một lớp ghép chuyển đổi với sải chân 32, thiết lập chiều cao và chiều rộng của hạt nhân là 64, đệm là 16. Nói chung, chúng ta có thể thấy rằng đối với sải chân s, đệm $s/2$ (giả sử $s/2$ là một số nguyên) và chiều cao và chiều rộng của hạt nhân $2s$, sự phức tạp chuyển tiếp sẽ làm tăng chiều cao và chiều rộng của đầu vào lên s lần.

```
num_classes = 21
net.add_module('final_conv', nn.Conv2d(512, num_classes, kernel_size=1))
net.add_module('transpose_conv', nn.ConvTranspose2d(num_classes, num_classes,
                                    kernel_size=64, padding=16, stride=32))
```

14.11.2 Initializing Transposed Convolutional Layers

Chúng ta đã biết rằng các lớp phức tạp chuyển tiếp có thể làm tăng chiều cao và chiều rộng của bản đồ tính năng. Trong xử lý hình ảnh, chúng ta có thể cần phải mở rộng một hình ảnh, tức là, * upsampling*. *Nội suy song sinh* là một trong những kỹ thuật lấy mẫu thường được sử dụng. Nó cũng thường được sử dụng để khởi tạo các lớp phức tạp chuyển tiếp.

Để giải thích nội suy song tuyến, hãy nói rằng cho một hình ảnh đầu vào, chúng tôi muốn tính toán từng pixel của hình ảnh đầu ra được lấy mẫu lên. Để tính điểm ảnh của hình ảnh đầu ra ở tọa độ (x, y), bản đồ đầu tiên (x, y) phối hợp (x', y') trên hình ảnh đầu vào, ví dụ, theo tỷ lệ kích thước đầu vào với kích thước đầu ra. Lưu ý rằng x' and y' ánh xạ là số thực. Sau đó, tìm bốn pixel gần nhất với tọa độ (x', y') trên hình ảnh đầu vào.

Cuối cùng, pixel của hình ảnh đầu ra ở tọa độ (x, y) được tính dựa trên bốn pixel gần nhất trên ảnh đầu vào và khoảng cách tương đối của chúng từ (x', y').

Upsampling của nội suy bilinear có thể được thực hiện bởi lớp ghép chuyển đổi với hạt nhân được xây dựng bởi hàm `bilinear_kernel` sau đây. Do hạn chế về không gian, chúng tôi chỉ cung cấp việc triển khai hàm `bilinear_kernel` bên dưới mà không cần thảo luận về thiết kế thuật toán của nó.

```
def bilinear_kernel(in_channels, out_channels, kernel_size):
    factor = (kernel_size + 1) // 2
    if kernel_size % 2 == 1:
        center = factor - 1
    else:
        center = factor - 0.5
    og = (torch.arange(kernel_size).reshape(-1, 1),
          torch.arange(kernel_size).reshape(1, -1))
    filt = (1 - torch.abs(og[0] - center) / factor) * \
           (1 - torch.abs(og[1] - center) / factor)
    weight = torch.zeros((in_channels, out_channels,
                          kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return weight
```

Hãy để chúng tôi thử nghiệm với upsampling of bilinear interpolation được thực hiện bởi một lớp ghép chuyển tiếp. Chúng tôi xây dựng một lớp tích hợp chuyển tiếp tăng gấp đôi chiều cao và trọng lượng, và khởi tạo hạt nhân của nó với hàm `bilinear_kernel`.

```
conv_trans = nn.ConvTranspose2d(3, 3, kernel_size=4, padding=1, stride=2,
                                bias=False)
conv_trans.weight.data.copy_(bilinear_kernel(3, 3, 4));
```

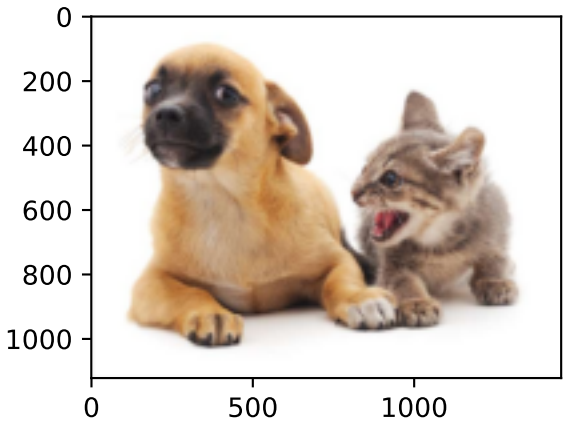
Đọc hình ảnh `X` và gán đầu ra lấy mẫu lên `Y`. Để in hình ảnh, chúng ta cần điều chỉnh vị trí của kích thước kênh.

```
img = torchvision.transforms.ToTensor()(d2l.Image.open('../img/catdog.jpg'))
X = img.unsqueeze(0)
Y = conv_trans(X)
out_img = Y[0].permute(1, 2, 0).detach()
```

Như chúng ta có thể thấy, lớp biến thể chuyển đổi làm tăng cả chiều cao và chiều rộng của hình ảnh bằng một yếu tố là hai. Ngoại trừ các thang đo khác nhau về tọa độ, hình ảnh được thu nhỏ bằng nội suy song tuyến và hình ảnh gốc được in trong Section 14.3 trông giống nhau.

```
d2l.set_figsize()
print('input image shape:', img.permute(1, 2, 0).shape)
d2l.plt.imshow(img.permute(1, 2, 0));
print('output image shape:', out_img.shape)
d2l.plt.imshow(out_img);
```

```
input image shape: torch.Size([561, 728, 3])
output image shape: torch.Size([1122, 1456, 3])
```



Trong một mạng phức tạp hoàn toàn, chúng ta khởi tạo lớp ghép chuyển tiếp với upsampling của nội suy song sinh. Đối với lớp ghép 1×1, chúng tôi sử dụng khởi tạo Xavier.

```
W = bilinear_kernel(num_classes, num_classes, 64)
net.transpose_conv.weight.data.copy_(W);
```

14.11.3 Đọc dữ liệu

Chúng tôi đọc tập dữ liệu phân đoạn ngữ nghĩa như được giới thiệu trong Section 14.9. Hình dạng hình ảnh đầu ra của cắt xén ngẫu nhiên được chỉ định là 320×480: cả chiều cao và chiều rộng đều chia hết cho 32.

```
batch_size, crop_size = 32, (320, 480)
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

```
read 1114 examples
read 1078 examples
```

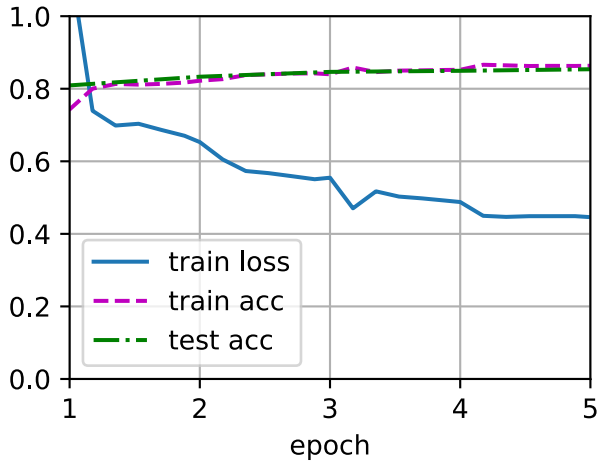
14.11.4 Đào tạo

Bây giờ chúng ta có thể đào tạo mạng lưới phức tạp được xây dựng hoàn toàn của chúng tôi. Chức năng mất mát và tính chính xác ở đây về cơ bản không khác với các chức năng trong phân loại hình ảnh của các chương trước đó. Bởi vì chúng ta sử dụng kênh đầu ra của lớp tích hợp chuyển tiếp để dự đoán lớp cho mỗi pixel, kích thước kênh được chỉ định trong phép tính tổn thất. Ngoài ra, độ chính xác được tính dựa trên tính chính xác của lớp dự đoán cho tất cả các pixel.

```
def loss(inputs, targets):
    return F.cross_entropy(inputs, targets, reduction='none').mean(1).mean(1)

num_epochs, lr, wd, devices = 5, 0.001, 1e-3, d2l.try_all_gpus()
trainer = torch.optim.SGD(net.parameters(), lr=lr, weight_decay=wd)
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.446, train acc 0.863, test acc 0.854
252.5 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```

(continues on next page)



14.11.5 Prediction

Khi dự đoán, chúng ta cần chuẩn hóa hình ảnh đầu vào trong mỗi kênh và chuyển đổi hình ảnh thành định dạng đầu vào bốn chiều theo yêu cầu của CNN.

```
def predict(img):
    X = test_iter.dataset.normalize_image(img).unsqueeze(0)
    pred = net(X.to(devices[0])).argmax(dim=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

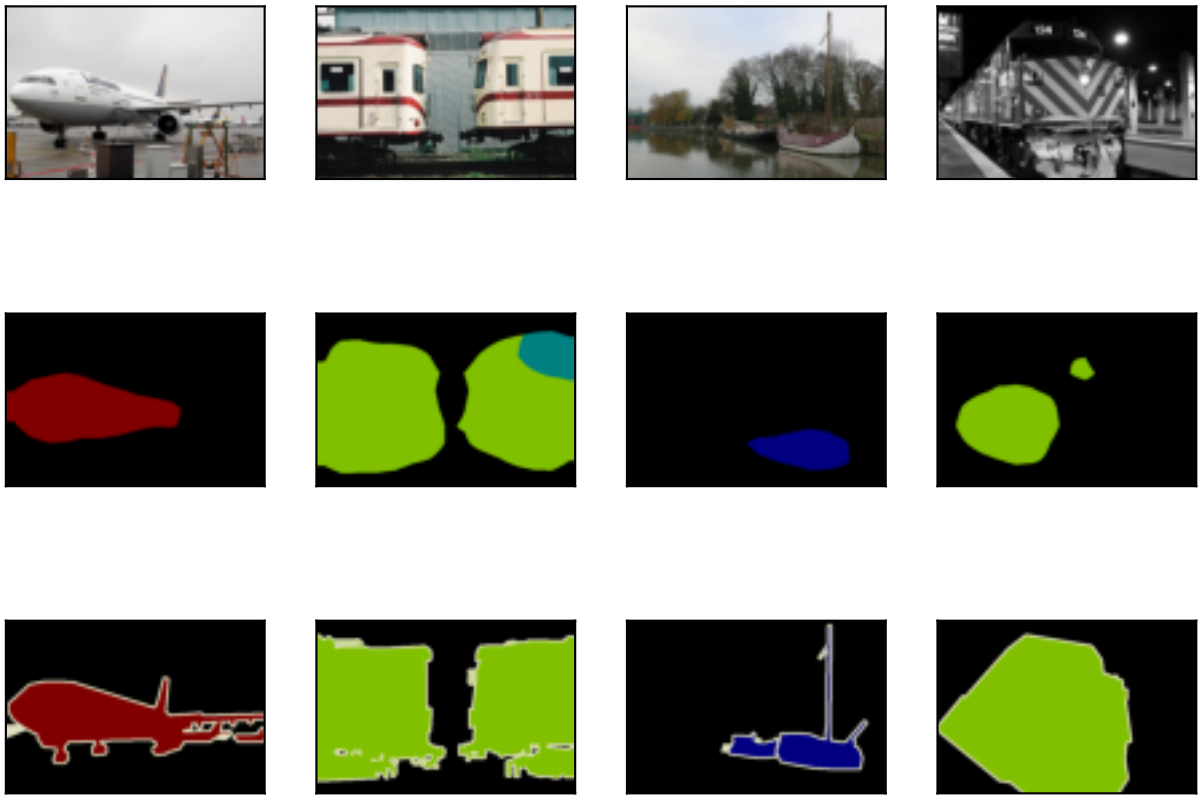
Để visualize the predicted class của mỗi pixel, chúng ta ánh xạ lớp dự đoán trở lại màu nhãn của nó trong tập dữ liệu.

```
def label2image(pred):
    colormap = torch.tensor(d2l.VOC_COLORMAP, device=devices[0])
    X = pred.long()
    return colormap[X, :]
```

Hình ảnh trong tập dữ liệu thử nghiệm khác nhau về kích thước và hình dạng. Kể từ khi mô hình sử dụng một lớp ghép chuyển tiếp với sải chân là 32, khi chiều cao hoặc chiều rộng của một hình ảnh đầu vào là không thể chia cắt bởi 32, chiều cao đầu ra hoặc chiều rộng của lớp ghép chuyển tiếp sẽ lệch khỏi hình dạng của hình ảnh đầu vào. Để giải quyết vấn đề này, chúng ta có thể cắt nhiều vùng hình chữ nhật với chiều cao và chiều rộng là bội số nguyên là 32 trong hình ảnh và thực hiện lan truyền về phía trước trên các pixel trong các khu vực này một cách riêng biệt. Lưu ý rằng sự kết hợp của các khu vực hình chữ nhật này cần phải bao phủ hoàn toàn hình ảnh đầu vào. Khi một pixel được bao phủ bởi nhiều khu vực hình chữ nhật, trung bình của các đầu ra tích hợp chuyển đổi trong các khu vực riêng biệt cho cùng một điểm ảnh này có thể được nhập vào hoạt động softmax để dự đoán lớp.

Để đơn giản, chúng tôi chỉ đọc một vài hình ảnh thử nghiệm lớn hơn và cắt một khu vực 320×480 để dự đoán bắt đầu từ góc trên bên trái của một hình ảnh. Đối với những hình ảnh thử nghiệm này, chúng tôi in các khu vực cắt, kết quả dự đoán và sự thật mặt đất từng hàng.

```
voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')
test_images, test_labels = d2l.read_voc_images(voc_dir, False)
n, imgs = 4, []
for i in range(n):
    crop_rect = (0, 0, 320, 480)
    X = torchvision.transforms.functional.crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X.permute(1,2,0), pred.cpu(),
             torchvision.transforms.functional.crop(
                 test_labels[i], *crop_rect).permute(1,2,0)]
d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```



14.11.6 Tóm tắt

- Mạng phức tạp hoàn toàn đầu tiên sử dụng CNN để trích xuất các tính năng hình ảnh, sau đó chuyển đổi số lượng kênh thành số lượng lớp thông qua lớp ghép 1×1 và cuối cùng biến đổi chiều cao và chiều rộng của bản đồ tính năng thành các hình ảnh đầu vào thông qua sự chuyển đổi.
- Trong một mạng phức tạp hoàn toàn, chúng ta có thể sử dụng upsampling của nội suy song tuyến để khởi tạo lớp chuyển đổi.

14.11.7 Bài tập

1. Nếu chúng ta sử dụng khởi tạo Xavier cho lớp tích hợp chuyển tiếp trong thí nghiệm, kết quả thay đổi như thế nào?
2. Bạn có thể cải thiện hơn nữa độ chính xác của mô hình bằng cách điều chỉnh các siêu tham số?
3. Dự đoán các lớp của tất cả các pixel trong hình ảnh thử nghiệm.
4. Giấy mạng phức tạp hoàn toàn ban đầu cũng sử dụng đầu ra của một số lớp CNN trung gian (Long et al., 2015). Cố gắng thực hiện ý tưởng này.

Discussions[182]

14.12 Chuyển kiểu thần kinh

Nếu bạn là một người đam mê nhiếp ảnh, bạn có thể đã quen thuộc với bộ lọc. Nó có thể thay đổi kiểu màu của ảnh để ảnh phong cảnh trở nên sắc nét hơn hoặc ảnh chân dung đã làm trắng da. Tuy nhiên, một bộ lọc thường chỉ thay đổi một khía cạnh của ảnh. Để áp dụng một phong cách lý tưởng cho ảnh, có lẽ bạn cần thử nhiều kết hợp bộ lọc khác nhau. Quá trình này phức tạp như điều chỉnh các siêu tham số của một mô hình.

Trong phần này, chúng ta sẽ tận dụng các biểu diễn theo lớp của CNN để tự động áp dụng kiểu của một hình ảnh cho một hình ảnh khác, tức là, * phong cách transfer* (Gatys et al., 2016). Nhiệm vụ này cần hai hình ảnh đầu vào: một là hình ảnh* nội dung hình ảnh* và cái còn lại là hình ảnh* phong cách*. Chúng tôi sẽ sử dụng mạng nơ-ron để sửa đổi hình ảnh nội dung để làm cho nó gần với hình ảnh phong cách theo phong cách. Ví dụ, hình ảnh nội dung trong Fig. 14.12.1 là một bức ảnh phong cảnh do chúng tôi chụp trong Vườn quốc gia Núi Rainier ở ngoại ô Seattle, trong khi hình ảnh phong cách là một bức tranh sơn dầu với chủ đề cây sồi mùa thu. Trong hình ảnh tổng hợp đầu ra, các nét cọ dầu của hình ảnh phong cách được áp dụng, dẫn đến màu sắc sống động hơn, đồng thời vẫn giữ được hình dạng chính của các đối tượng trong hình ảnh nội dung.

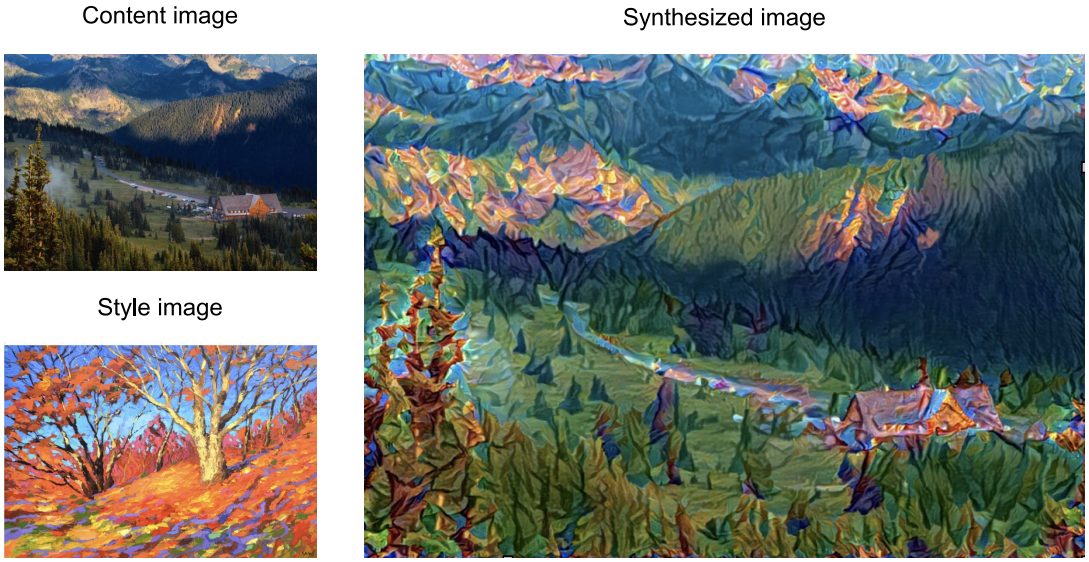


Fig. 14.12.1: Given content and style images, style transfer outputs a synthesized image.

[182] https://discuss.d2l.ai/t/1582

14.12.1 Phương pháp

Fig. 14.12.2 minh họa phương thức chuyển kiểu dựa trên CNN với một ví dụ đơn giản hóa. Đầu tiên, chúng tôi khởi tạo hình ảnh tổng hợp, ví dụ, vào hình ảnh nội dung. Hình ảnh tổng hợp này là biến duy nhất cần được cập nhật trong quá trình chuyển kiểu, tức là các tham số mô hình được cập nhật trong quá trình đào tạo. Sau đó, chúng tôi chọn CNN được đào tạo trước để trích xuất các tính năng hình ảnh và đóng băng các thông số mô hình của nó trong quá trình đào tạo. CNN sâu này sử dụng nhiều lớp để trích xuất các tính năng phân cấp cho hình ảnh. Chúng ta có thể chọn đầu ra của một số lớp này làm tính năng nội dung hoặc tính năng phong cách. Lấy Fig. 14.12.2 làm ví dụ. Mạng nơ-ron được đào tạo trước ở đây có 3 lớp phức tạp, trong đó lớp thứ hai xuất ra các tính năng nội dung, và các lớp thứ nhất và thứ ba xuất ra các tính năng kiểu.

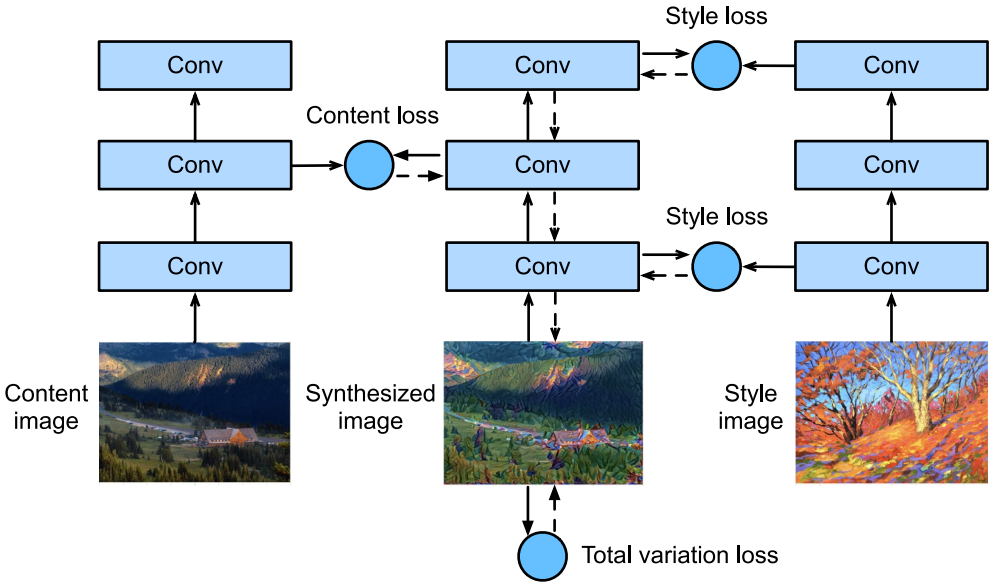


Fig. 14.12.2: CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backward propagation.

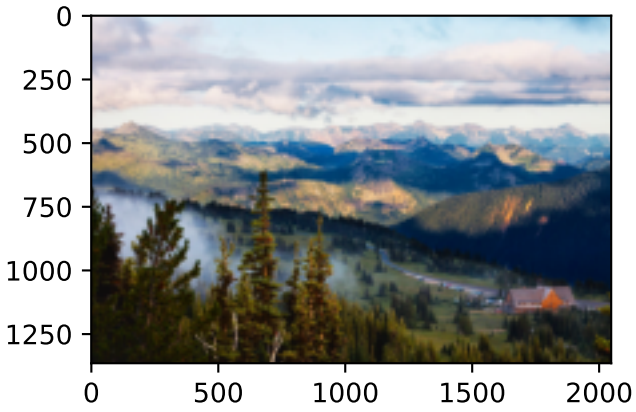
Tiếp theo, chúng ta tính toán hàm mất của chuyển kiểu thông qua lan truyền về phía trước (hướng của mũi tên rắn), và cập nhật các tham số mô hình (hình ảnh tổng hợp cho đầu ra) thông qua truyền ngược (hướng của các mũi tên đứt nét). Chức năng mất thường được sử dụng trong chuyển phong cách bao gồm ba phần: (i) * mất nội dung* làm cho hình ảnh tổng hợp và hình ảnh nội dung gần gũi trong các tính năng nội dung; (ii) * phong cách mất đi* làm cho hình ảnh tổng hợp và phong cách gần gũi trong các tính năng phong cách; và (iii) * mất biến thể* giúp giảm noise tiếng ồn in the synthesized tổng hợp image hình ảnh. Cuối cùng, khi đào tạo mô hình kết thúc, chúng tôi xuất các thông số mô hình của chuyển kiểu để tạo ra hình ảnh tổng hợp cuối cùng.

Sau đây, chúng tôi sẽ giải thích các chi tiết kỹ thuật của chuyển phong cách thông qua một thí nghiệm cụ thể.

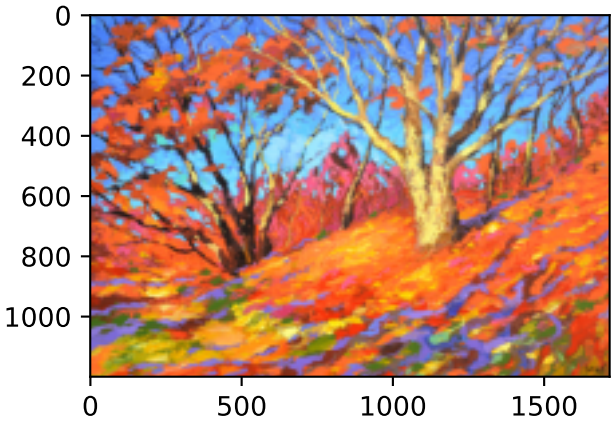
14.12.2 Đọc nội dung và phong cách hình ảnh

Đầu tiên, chúng tôi đọc nội dung và phong cách hình ảnh. Từ các trục tọa độ in của chúng, chúng ta có thể nói rằng những hình ảnh này có kích thước khác nhau.

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l

d2l.set_figsize()
content_img = d2l.Image.open('../img/rainier.jpg')
d2l.plt.imshow(content_img);
```




```
style_img = d2l.Image.open('../img/autumn-oak.jpg')
d2l.plt.imshow(style_img);
```



14.12.3 Tiền xử lý và xử lý sau

Dưới đây, chúng tôi xác định hai chức năng cho hình ảnh tiền xử lý và xử lý hậu kỳ. Chức năng `preprocess` chuẩn hóa mỗi kênh trong ba kênh RGB của hình ảnh đầu vào và biến đổi kết quả thành định dạng đầu vào CNN. Hàm `postprocess` khôi phục các giá trị điểm ảnh trong hình ảnh đầu ra về giá trị ban đầu của chúng trước khi tiêu chuẩn hóa. Vì chức năng in ảnh yêu cầu mỗi pixel có giá trị điểm nổi từ 0 đến 1, chúng ta thay thế bất kỳ giá trị nào nhỏ hơn 0 hoặc lớn hơn 1 với 0 hoặc 1, tương ứng.

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

14.12.4 tính năng chiết xuất

Chúng tôi sử dụng mô hình VGG-19 được đào tạo trước trên bộ dữ liệu ImageNet để trích xuất các tính năng hình ảnh (Gatys et al., 2016).

```
pretrained_net = torchvision.models.vgg19(pretrained=True)
```

Để trích xuất các tính năng nội dung và tính năng kiểu của hình ảnh, chúng ta có thể chọn đầu ra của các lớp nhất định trong mạng VGG. Nói chung, càng gần lớp đầu vào, dễ dàng hơn để trích xuất chi tiết của hình ảnh, và ngược lại, dễ dàng hơn để trích xuất thông tin toàn cầu của hình ảnh. Để tránh giữ lại quá mức các chi tiết của hình ảnh nội dung trong hình ảnh tổng hợp, chúng tôi chọn một lớp VGG gần đầu ra hơn làm lớp nội dung *để xuất các tính năng nội dung của hình ảnh. Chúng tôi cũng chọn đầu ra của các lớp VGG khác nhau để trích xuất các tính năng kiểu địa phương và toàn cầu. Các lớp này còn được gọi là lớp kiểu *. Như đã đề cập trong Section 8.2, mạng VGG sử dụng 5 khối phức tạp. Trong thí nghiệm, chúng ta chọn lớp phức tạp cuối cùng của khối phức tạp thứ tư làm lớp nội dung, và lớp phức tạp đầu tiên của mỗi khối phức tạp làm lớp phong cách. Các chỉ số của các lớp này có thể thu được bằng cách in phiên bản `pretrained_net`.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

Khi trích xuất các đối tượng bằng cách sử dụng các lớp VGG, chúng ta chỉ cần sử dụng tất cả các đối tượng từ lớp input đến lớp nội dung hoặc layer style gần nhất với lớp đầu ra nhất. Chúng ta hãy xây dựng một phiên bản mạng mới `net`, chỉ giữ lại tất cả các lớp VGG được sử dụng để trích xuất tính năng.

```
net = nn.Sequential(*[pretrained_net.features[i] for i in
                      range(max(content_layers + style_layers) + 1)])
```

Với đầu vào `X`, nếu chúng ta chỉ đơn giản gọi tuyên truyền chuyển tiếp `net(X)`, chúng ta chỉ có thể nhận được đầu ra của lớp cuối cùng. Vì chúng ta cũng cần các đầu ra của các lớp trung gian, chúng ta cần thực hiện tính toán từng lớp và giữ cho các đầu ra lớp nội dung và kiểu dáng.

```
def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

Hai chức năng được định nghĩa dưới đây: hàm `get_contents` trích xuất các tính năng nội dung từ hình ảnh nội dung và hàm `get_styles` trích xuất các tính năng kiểu từ hình ảnh phong cách. Vì không cần cập nhật các thông số mô hình của VGG được đào tạo trước trong quá trình đào tạo, chúng tôi có thể trích xuất nội dung và các tính năng phong cách ngay cả trước khi bắt đầu đào tạo. Vì hình ảnh tổng hợp là một tập hợp các tham số mô hình được cập nhật để chuyển kiểu, chúng ta chỉ có thể trích xuất nội dung và tính năng kiểu dáng của hình ảnh tổng hợp bằng cách gọi hàm `extract_features` trong quá trình đào tạo.

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

14.12.5 Defining the Loss Function

Bây giờ chúng ta sẽ mô tả chức năng mất để chuyển phong cách. Chức năng mất bao gồm mất nội dung, mất phong cách và mất hoàn toàn biến thể.

Mất nội dung

Tương tự như chức năng mất trong hồi quy tuyến tính, mất nội dung đo lường sự khác biệt về các tính năng nội dung giữa hình ảnh tổng hợp và hình ảnh nội dung thông qua chức năng mất bình phương. Hai đầu vào của hàm mất bình phương là cả hai đầu ra của lớp nội dung được tính toán bởi hàm `extract_features`.

```
def content_loss(Y_hat, Y):
    # We detach the target content from the tree used to dynamically compute
    # the gradient: this is a stated value, not a variable. Otherwise the loss
    # will throw an error.
    return torch.square(Y_hat - Y.detach()).mean()
```

Phong cách mất

Mất phong cách, tương tự như mất nội dung, cũng sử dụng chức năng mất bình phương để đo lường sự khác biệt về phong cách giữa hình ảnh tổng hợp và hình ảnh phong cách. Để thể hiện đầu ra kiểu của bất kỳ layer style nào, trước tiên chúng ta sử dụng hàm `extract_features` để tính toán đầu ra layer style. Giả sử rằng đầu ra có 1 ví dụ, c kênh, chiều cao h và chiều rộng w, chúng ta có thể chuyển đổi đầu ra này thành ma trận $\mathbf{X}$ với c hàng và hw cột. Ma trận này có thể được coi là sự nối của c vectơ $\mathbf{x}_1, \ldots, \mathbf{x}_c$, mỗi vectơ có chiều dài hw. Ở đây, vector $\mathbf{x}_i$ đại diện cho tính năng phong cách của kênh i.

Trong ma trận *Gram* của các vectơ $\mathbf{X}\mathbf{X}^\top \in \mathbb{R}^{c \times c}$, phần tử x_{ij} trong hàng i và cột j là tích chấm của vectơ $\mathbf{x}_i$ và $\mathbf{x}_j$. Nó đại diện cho mối tương quan của các tính năng phong cách của các kênh i và j. Chúng tôi sử dụng ma trận Gram này để đại diện cho đầu ra kiểu của bất kỳ layer style nào. Lưu ý rằng khi giá trị của hw lớn hơn, nó có thể dẫn đến các giá trị lớn hơn trong ma trận Gram. Cũng lưu ý rằng chiều cao và chiều rộng của ma trận Gram đều là số kênh c. Để cho phép mất kiểu không bị ảnh hưởng bởi các giá trị này, hàm `gram` bên dưới chia ma trận Gram cho số phần tử của nó, tức là chw.

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

Rõ ràng, hai đầu vào ma trận Gram của hàm mất bình phương để mất kiểu được dựa trên đầu ra lớp kiểu cho hình ảnh tổng hợp và hình ảnh phong cách. Người ta cho rằng ma trận Gram `gram_Y` dựa trên hình ảnh phong cách đã được tính toán trước.

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

Tổng Biến Thể Mất

Đôi khi, hình ảnh tổng hợp đã học có rất nhiều nhiễu tần số cao, tức là, đặc biệt là các pixel sáng hoặc tối. Một phương pháp giảm tiếng ồn phổ biến là *tổng biến thể biểu tượng*. Biểu thị bằng $x_{i,j}$ giá trị điểm ảnh ở tọa độ (i, j). Giảm tổng tổn thất biến thể

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}| \tag{14.12.1}$$

làm cho các giá trị của các pixel lân cận trên hình ảnh tổng hợp gần hơn.

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

Chức năng mất

Chức năng mất của chuyển phong cách là tổng trọng số của mất nội dung, mất phong cách, và mất biến thể tổng số. Bằng cách điều chỉnh các siêu tham số trọng lượng này, chúng ta có thể cân bằng giữa việc lưu giữ nội dung, chuyển kiểu và giảm nhiễu trên hình ảnh tổng hợp.

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # Calculate the content, style, and total variance losses respectively
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # Add up all the losses
    l = sum(10 * styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

14.12.6 Initializing the Synthesized Image

Trong chuyển phong cách, hình ảnh tổng hợp là biến duy nhất cần được cập nhật trong quá trình đào tạo. Do đó, chúng ta có thể xác định một mô hình đơn giản, `SynthesizedImage` và coi hình ảnh tổng hợp như các tham số mô hình. Trong mô hình này, tuyên truyền chuyển tiếp chỉ trả về các tham số mô hình.

```
class SynthesizedImage(nn.Module):
    def __init__(self, img_shape, **kwargs):
        super(SynthesizedImage, self).__init__(**kwargs)
        self.weight = nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

Tiếp theo, ta định nghĩa hàm `get_inits`. Hàm này tạo ra một ví dụ mô hình ảnh tổng hợp và khởi tạo nó thành hình ảnh `X`. Ma trận gram cho hình ảnh phong cách ở các lớp phong cách khác nhau, `styles_Y_gram`, được tính toán trước khi đào tạo.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

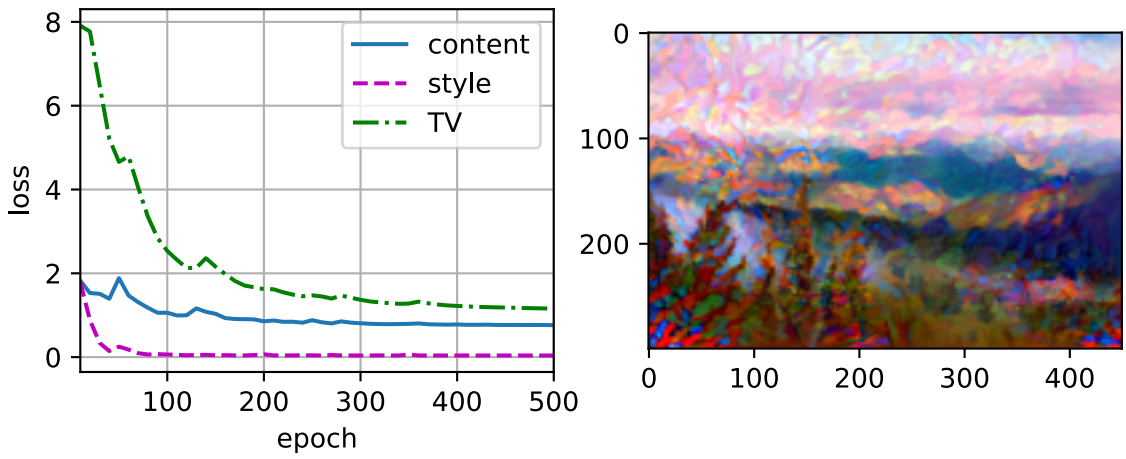
14.12.7 Đào tạo

Khi đào tạo mô hình để chuyển phong cách, chúng tôi liên tục trích xuất các tính năng nội dung và tính năng phong cách của hình ảnh tổng hợp và tính toán chức năng mất mát. Dưới đây xác định vòng lặp đào tạo.

```
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

Bây giờ chúng ta bắt đầu đào tạo mô hình. Chúng tôi giải thích chiều cao và chiều rộng của nội dung và phong cách hình ảnh lên 300 x 450 pixel. Chúng tôi sử dụng hình ảnh nội dung để khởi tạo hình ảnh tổng hợp.

```
device, image_shape = d2l.try_gpu(), (300, 450)  # PIL Image (h, w)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```



Chúng ta có thể thấy rằng hình ảnh tổng hợp giữ lại cảnh quan và đối tượng của hình ảnh nội dung và chuyển màu sắc của hình ảnh phong cách cùng một lúc. Ví dụ, hình ảnh tổng hợp có các khối màu giống như trong hình ảnh phong cách. Một số khối này thậm chí còn có kết cấu tinh tế của nét cọ.

14.12.8 Tóm tắt

- Chức năng mất mát thường được sử dụng trong chuyển phong cách bao gồm ba phần: (i) mất nội dung làm cho hình ảnh tổng hợp và hình ảnh nội dung gần với các tính năng nội dung; (ii) mất phong cách làm cho hình ảnh tổng hợp và hình ảnh phong cách gần gũi trong các tính năng phong cách; và (iii) mất biến đổi tổng thể giúp giảm nhiễu trong the synthesized tổng hợp image hình ảnh.
- Chúng ta có thể sử dụng CNN được đào tạo trước để trích xuất các tính năng hình ảnh và giảm thiểu chức năng mất mát để liên tục cập nhật hình ảnh tổng hợp dưới dạng tham số mô hình trong quá trình đào tạo.
- Chúng ta sử dụng ma trận Gram để biểu diễn các đầu ra kiểu dáng từ các layer style.

14.12.9 Bài tập

1. Làm thế nào để đầu ra thay đổi khi bạn chọn các lớp nội dung và kiểu khác nhau?
2. Điều chỉnh các siêu tham số trọng lượng trong chức năng mất mát. Đầu ra có giữ được nhiều nội dung hơn hoặc ít tiếng ồn hơn?
3. Sử dụng nội dung khác nhau và hình ảnh phong cách. Bạn có thể tạo ra những hình ảnh tổng hợp thú vị hơn không?
4. Chúng ta có thể áp dụng chuyển kiểu cho văn bản không? Hint: you may refer to the survey paper by Hu et al. [Hu.Lee.Aggarwal.ea.2020].

Discussions[183]

14.13 Phân loại hình ảnh (CIFAR-10) trên Kaggle

Cho đến nay, chúng tôi đã sử dụng các API cấp cao của các framework deep learning để trực tiếp lấy tập dữ liệu hình ảnh ở định dạng tensor. Tuy nhiên, các tập dữ liệu hình ảnh tùy chỉnh thường có dạng tệp hình ảnh. Trong phần này, chúng ta sẽ bắt đầu từ các tệp hình ảnh thô và sắp xếp, đọc, sau đó chuyển đổi chúng thành định dạng tensor từng bước.

Chúng tôi đã thử nghiệm với tập dữ liệu CIFAR-10 trong Section 14.1, đây là một tập dữ liệu quan trọng trong tầm nhìn máy tính. Trong phần này, chúng tôi sẽ áp dụng kiến thức chúng tôi đã học được trong các phần trước để thực hành cuộc thi Kaggle của phân loại hình ảnh CIFAR-10. Địa chỉ web của cuộc thi là https://www.kaggle.com/c/cifar-10

Fig. 14.13.1 hiển thị thông tin trên trang web của cuộc thi. Để gửi kết quả, bạn cần đăng ký tài khoản Kaggle.

[183] https://discuss.d2l.ai/t/1476

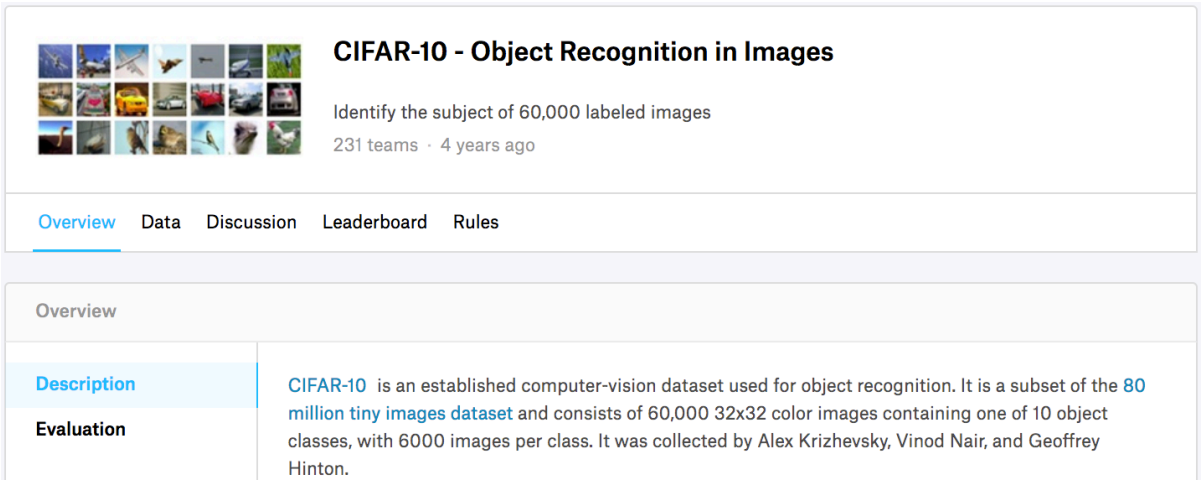


Fig. 14.13.1: CIFAR-10 image classification competition webpage information. The competition dataset can be obtained by clicking the "Data" tab.

```
import collections
import math
import os
import shutil
import pandas as pd
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

14.13.1 Lấy và tổ chức tập dữ liệu

Tập dữ liệu cuộc thi được chia thành một bộ đào tạo và một bộ thử nghiệm, có chứa 50000 và 300000 hình ảnh, tương ứng. Trong bộ thử nghiệm, 10000 hình ảnh sẽ được sử dụng để đánh giá, trong khi 290000 hình ảnh còn lại sẽ không được đánh giá: chúng được bao gồm chỉ để làm cho nó khó khăn để lừa dối *kết quả được dán nhãn bằng tay* của bộ thử nghiệm. Các hình ảnh trong tập dữ liệu này là tất cả các tệp hình ảnh màu png (kênh RGB), có chiều cao và chiều rộng đều là 32 pixel. Các hình ảnh bao gồm tổng cộng 10 loại, cụ thể là máy bay, ô tô, chim, mèo, hươu, chó, ếch, ngựa, thuyền và xe tải. Góc trên bên trái của Fig. 14.13.1 cho thấy một số hình ảnh về máy bay, ô tô và chim trong bộ dữ liệu.

Tải xuống tập dữ liệu

Sau khi đăng nhập vào Kaggle, chúng ta có thể nhấp vào tab "Dữ liệu" trên trang web cạnh tranh phân loại hình ảnh CIFAR-10 được hiển thị trong Fig. 14.13.1 và tải xuống tập dữ liệu bằng cách nhấp vào nút "Tải xuống tất cả". Sau khi giải nén tệp đã tải xuống trong `../data` và giải nén `train.7z` và `test.7z` bên trong tệp đó, bạn sẽ tìm thấy toàn bộ bộ dữ liệu trong các đường dẫn sau:

- `../data/cifar-10/train/[1-50000].png`
- `../data/cifar-10/test/[1-300000].png`
- `../data/cifar-10/trainLabels.csv`
- `../data/cifar-10/sampleSubmission.csv`

trong đó các thư mục `train` và `test` chứa các hình ảnh đào tạo và thử nghiệm, tương ứng, `trainLabels.csv` cung cấp nhãn cho các hình ảnh đào tạo, và `sample_submission.csv` là một tập tin gửi mẫu.

Để bắt đầu dễ dàng hơn, chúng tôi cung cấp một mẫu quy mô nhỏ của bộ dữ liệu chứa 1000 hình ảnh đào tạo đầu tiên và 5 hình ảnh thử nghiệm ngẫu nhiên. Để sử dụng bộ dữ liệu đầy đủ của cuộc thi Kaggle, bạn cần đặt biến `demo` sau thành `False`.

```
#@save
d2l.DATA_HUB['cifar10_tiny'] = (d2l.DATA_URL + 'kaggle_cifar10_tiny.zip',
                                '2068874e4b9a9f0fb07ebe0ad2b29754449ccacd')

# If you use the full dataset downloaded for the Kaggle competition, set
# `demo` to False
demo = True

if demo:
    data_dir = d2l.download_extract('cifar10_tiny')
else:
    data_dir = '../data/cifar-10/'
```

```
Downloading ../data/kaggle_cifar10_tiny.zip from http://d2l-data.s3-
↪accelerate.amazonaws.com/kaggle_cifar10_tiny.zip...
```

Torganizing the Dataset

Chúng ta cần tổ chức các tập dữ liệu để tạo điều kiện cho việc đào tạo và thử nghiệm mô hình. Trước tiên chúng ta hãy đọc các nhãn từ tệp csv. Hàm sau trả về một từ điển mà ánh xạ phần không mở rộng của tên tập tin vào nhãn của nó.

```
#@save
def read_csv_labels(fname):
    """Read `fname` to return a filename to label dictionary."""
    with open(fname, 'r') as f:
        # Skip the file header line (column name)
        lines = f.readlines()[1:]
    tokens = [l.rstrip().split(',') for l in lines]
    return dict(((name, label) for name, label in tokens))

labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
print('# training examples:', len(labels))
print('# classes:', len(set(labels.values())))
```

```
# training examples: 1000
# classes: 10
```

Tiếp theo, chúng ta định nghĩa hàm `reorg_train_valid` thành chia xác thực được đặt ra khỏi tập đào tạo ban đầu. Đối số `valid_ratio` trong hàm này là tỷ lệ giữa số ví dụ trong xác thực được đặt thành số ví dụ trong bộ đào tạo ban đầu. Cụ thể hơn, hãy để n là số lượng hình ảnh của lớp với ít ví dụ nhất, và r là tỷ lệ. Bộ xác nhận sẽ chia ra $\max(\lfloor nr \rfloor, 1)$ hình ảnh cho mỗi lớp. Hãy để chúng tôi sử dụng `valid_ratio=0.1` làm ví dụ. Vì bộ đào tạo ban đầu có 50000 hình ảnh, sẽ có 45000 hình ảnh được sử dụng để đào tạo trong đường `train_valid_test/train`, trong khi 5000 hình ảnh khác sẽ được chia ra như xác nhận được

đặt trong đường `train_valid_test/valid`. Sau khi tổ chức tập dữ liệu, hình ảnh của cùng một lớp sẽ được đặt dưới cùng một thư mục.

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    os.makedirs(target_dir, exist_ok=True)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    """Split the validation set out of the original training set."""
    # The number of examples of the class that has the fewest examples in the
    # training dataset
    n = collections.Counter(labels.values()).most_common()[-1][1]
    # The number of examples per class for the validation set
    n_valid_per_label = max(1, math.floor(n * valid_ratio))
    label_count = {}
    for train_file in os.listdir(os.path.join(data_dir, 'train')):
        label = labels[train_file.split('.')[0]]
        fname = os.path.join(data_dir, 'train', train_file)
        copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                     'train_valid', label))
        if label not in label_count or label_count[label] < n_valid_per_label:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

Hàm `reorg_test` bên dưới tổ chức bộ thử nghiệm để tải dữ liệu trong quá trình dự đoán.

```
#@save
def reorg_test(data_dir):
    """Organize the testing set for data loading during prediction."""
    for test_file in os.listdir(os.path.join(data_dir, 'test')):
        copyfile(os.path.join(data_dir, 'test', test_file),
                 os.path.join(data_dir, 'train_valid_test', 'test',
                              'unknown'))
```

Cuối cùng, chúng ta sử dụng một hàm để invoke `read_csv_labels`, `reorg_train_valid`, và `reorg_test` functions được định nghĩa ở trên.

```
def reorg_cifar10_data(data_dir, valid_ratio):
    labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
    reorg_train_valid(data_dir, labels, valid_ratio)
    reorg_test(data_dir)
```

Ở đây chúng tôi chỉ đặt kích thước lô thành 32 cho mẫu quy mô nhỏ của tập dữ liệu. Khi đào tạo và thử nghiệm tập dữ liệu hoàn chỉnh của cuộc thi Kaggle, `batch_size` nên được đặt thành một số nguyên lớn hơn, chẳng hạn như 128. Chúng tôi chia ra 10% các ví dụ đào tạo là xác nhận được thiết lập để điều chỉnh các siêu tham số.

```
batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

14.13.2 Image Augmentation

Chúng tôi sử dụng nâng hình ảnh để giải quyết overfitting. Ví dụ, hình ảnh có thể được lật theo chiều ngang ngẫu nhiên trong quá trình đào tạo. Chúng tôi cũng có thể thực hiện tiêu chuẩn hóa cho ba kênh RGB của hình ảnh màu. Dưới đây liệt kê một số hoạt động mà bạn có thể tinh chỉnh.

```
transform_train = torchvision.transforms.Compose([
    # Scale the image up to a square of 40 pixels in both height and width
    torchvision.transforms.Resize(40),
    # Randomly crop a square image of 40 pixels in both height and width to
    # produce a small square of 0.64 to 1 times the area of the original
    # image, and then scale it to a square of 32 pixels in both height and
    # width
    torchvision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                             ratio=(1.0, 1.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

Trong quá trình thử nghiệm, chúng tôi chỉ thực hiện tiêu chuẩn hóa trên hình ảnh để loại bỏ sự ngẫu nhiên trong kết quả đánh giá.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                     [0.2023, 0.1994, 0.2010])])
```

14.13.3 Đọc tập dữ liệu

Tiếp theo, chúng ta đọc tập dữ liệu có tổ chức bao gồm các tập tin hình ảnh thô. Mỗi ví dụ bao gồm một hình ảnh và một nhãn.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Trong quá trình đào tạo, chúng ta cần chỉ định tất cả các thao tác nâng hình ảnh được xác định ở trên. Khi bộ xác thực được sử dụng để đánh giá mô hình trong quá trình điều chỉnh siêu tham số, không nên giới thiệu tính ngẫu nhiên từ việc tăng hình ảnh. Trước khi dự đoán cuối cùng, chúng tôi đào tạo mô hình trên bộ đào tạo kết hợp và bộ xác nhận để tận dụng tối đa tất cả dữ liệu được dán nhãn.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

14.13.4 Xác định Mẫu

Chúng tôi xác định mô hình ResNet-18 được mô tả trong Section 8.6.

```
def get_net():
    num_classes = 10
    net = d2l.resnet18(num_classes, 3)
    return net

loss = nn.CrossEntropyLoss(reduction="none")
```

14.13.5 Xác định đào tạo chức năng

Chúng tôi sẽ chọn các mô hình và điều chỉnh các siêu tham số theo hiệu suất của mô hình trên bộ xác thực. Sau đây, chúng tôi xác định hàm đào tạo mô hình `train`.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = torch.optim.SGD(net.parameters(), lr=lr, momentum=0.9,
                              weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss', 'train acc']
    if valid_iter is not None:
        legend.append('valid acc')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    for epoch in range(num_epochs):
        net.train()
        metric = d2l.Accumulator(3)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(net, features, labels,
                                          loss, trainer, devices)
            metric.add(l, acc, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[2], metric[1] / metric[2],
                              None))
```

(continues on next page)

(continued from previous page)

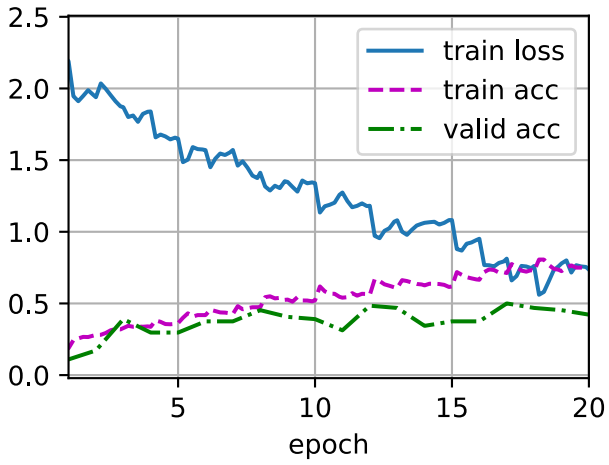
```
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy_gpu(net, valid_iter)
            animator.add(epoch + 1, (None, None, valid_acc))
        scheduler.step()
    measures = (f'train loss {metric[0] / metric[2]:.3f}, '
                f'train acc {metric[1] / metric[2]:.3f}')
    if valid_iter is not None:
        measures += f', valid acc {valid_acc:.3f}'
    print(measures + f'\n{metric[2] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

14.13.6 Đào tạo và xác thực mô hình

Bây giờ, chúng ta có thể đào tạo và xác thực mô hình. Tất cả các siêu tham số sau đây có thể được điều chỉnh. Ví dụ, chúng ta có thể tăng số lượng kỷ nguyên. Khi `lr_period` và `lr_decay` được đặt thành 4 và 0,9, tương ứng, tốc độ học tập của thuật toán tối ưu hóa sẽ được nhân với 0,9 sau mỗi 4 kỷ nguyên. Chỉ để dễ dàng trình diễn, chúng tôi chỉ đào tạo 20 kỷ nguyên ở đây.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 20, 2e-4, 5e-4
lr_period, lr_decay, net = 4, 0.9, get_net()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 0.736, train acc 0.749, valid acc 0.422
898.8 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```

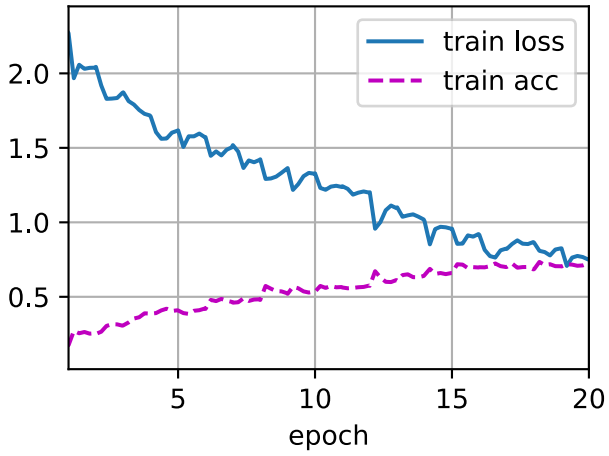


14.13.7 Phân loại bộ thử nghiệm và gửi kết quả trên Kaggle

Sau khi có được một mô hình đầy hứa hẹn với các siêu tham số, chúng tôi sử dụng tất cả dữ liệu được dán nhãn (bao gồm cả bộ xác thực) để đào tạo lại mô hình và phân loại bộ thử nghiệm.

```
net, preds = get_net(), []
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.to(devices[0]))
    preds.extend(y_hat.argmax(dim=1).type(torch.int32).cpu().numpy())
sorted_ids = list(range(1, len(test_ds) + 1))
sorted_ids.sort(key=lambda x: str(x))
df = pd.DataFrame({'id': sorted_ids, 'label': preds})
df['label'] = df['label'].apply(lambda x: train_valid_ds.classes[x])
df.to_csv('submission.csv', index=False)
```

```
train loss 0.755, train acc 0.721
1165.8 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Mã trên sẽ tạo ra một tập tin `submission.csv`, có định dạng đáp ứng yêu cầu của cuộc thi Kaggle. Phương pháp gửi kết quả cho Kaggle tương tự như trong Section 5.10.

14.13.8 Tóm tắt

- Chúng ta có thể đọc các bộ dữ liệu chứa các tệp hình ảnh thô sau khi sắp xếp chúng thành định dạng cần thiết.
- Chúng ta có thể sử dụng mạng thần kinh phức tạp và nâng hình ảnh trong một cuộc thi phân loại hình ảnh.

14.13.9 Bài tập

1. Sử dụng bộ dữ liệu CIFAR-10 hoàn chỉnh cho cuộc thi Kaggle này. Đặt các siêu tham số là `batch_size = 128`, `num_epochs = 100`, `lr = 0.1`, `lr_period = 50` và `lr_decay = 0.1`. Xem độ chính xác và thứ hạng bạn có thể đạt được trong cuộc thi này. Bạn có thể cải thiện thêm chúng?
2. Độ chính xác nào bạn có thể nhận được khi không sử dụng nâng hình ảnh?

Discussions[184]

14.14 Nhận dạng giống chó (Chó ImageNet) trên Kaggle

Trong phần này, chúng tôi sẽ thực hành vấn đề nhận dạng giống chó trên Kaggle. Địa chỉ web của cuộc thi này là https://www.kaggle.com/c/dog-breed-identification

Trong cuộc thi này, 120 giống chó khác nhau sẽ được công nhận. Trên thực tế, tập dữ liệu cho cuộc thi này là một tập hợp con của tập dữ liệu ImageNet. Không giống như các hình ảnh trong tập dữ liệu CIFAR-10 trong Section 14.13, các hình ảnh trong tập dữ liệu ImageNet đều cao hơn và rộng hơn ở các kích thước khác nhau. Fig. 14.14.1 hiển thị thông tin trên trang web của đối thủ. Bạn cần một tài khoản Kaggle để gửi kết quả của mình.

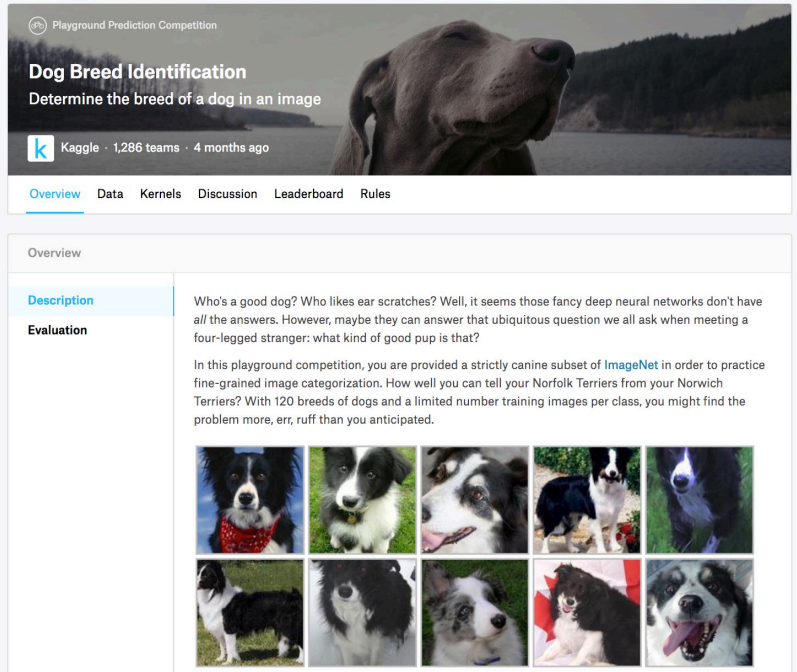


Fig. 14.14.1: The dog breed identification competition website. The competition dataset can be obtained by clicking the "Data" tab.

```
import os
import torch
import torchvision
```

(continues on next page)

[184] https://discuss.d2l.ai/t/1479

(continued from previous page)

```
from torch import nn
from d2l import torch as d2l
```

14.14.1 Lấy và tổ chức tập dữ liệu

Tập dữ liệu cạnh tranh được chia thành một bộ đào tạo và một bộ thử nghiệm, trong đó có 10222 và 10357 hình ảnh JPEG của ba kênh RGB (màu), tương ứng. Trong số các tập dữ liệu huấn luyện, có 120 giống chó như Labradors, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahua, và Yorkshire Terriers.

Tải xuống tập dữ liệu

Sau khi đăng nhập vào Kaggle, bạn có thể nhấp vào tab "Dữ liệu" trên trang web cạnh tranh được hiển thị trong Fig. 14.14.1 và tải xuống tập dữ liệu bằng cách nhấp vào nút "Tải xuống tất cả". Sau khi giải nén tệp đã tải xuống trong `../data`, bạn sẽ tìm thấy toàn bộ tập dữ liệu trong các đường dẫn sau:

-
-
-
-

Bạn có thể nhận thấy rằng cấu trúc trên tương tự như cấu trúc của cuộc thi CIFAR-10 trong Section 14.13, trong đó các thư mục `train/` và `test/` chứa các hình ảnh huấn luyện và thử nghiệm hình ảnh chó, và `labels.csv` chứa nhãn cho hình ảnh đào tạo. Tương tự, để bắt đầu dễ dàng hơn, chúng tôi cung cấp một mẫu nhỏ của bộ dữ liệu được đề cập ở trên: `train_valid_test_tiny.zip`. Nếu bạn định sử dụng bộ dữ liệu đầy đủ cho cuộc thi Kaggle, bạn cần thay đổi biến `demo` bên dưới thành `False`.

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '0cb91d09b814ecdc07b50f31f8dcad3e81d6a86d')

# If you use the full dataset downloaded for the Kaggle competition, change
# the variable below to `False`
demo = True
if demo:
    data_dir = d2l.download_extract('dog_tiny')
else:
    data_dir = os.path.join('..', 'data', 'dog-breed-identification')
```

```
Downloading ../data/kaggle_dog_tiny.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/kaggle_dog_tiny.zip...
```

Torganizing the Dataset

Chúng ta có thể sắp xếp tập dữ liệu tương tự như những gì chúng ta đã làm trong Section 14.13, cụ thể là tách ra một bộ xác thực từ bộ đào tạo ban đầu và di chuyển hình ảnh vào các thư mục con được nhóm theo nhãn.

Chức năng `reorg_dog_data` dưới đây đọc các nhãn dữ liệu đào tạo, tách bộ xác thực và tổ chức bộ đào tạo.

```
def reorg_dog_data(data_dir, valid_ratio):
    labels = d2l.read_csv_labels(os.path.join(data_dir, 'labels.csv'))
    d2l.reorg_train_valid(data_dir, labels, valid_ratio)
    d2l.reorg_test(data_dir)


batch_size = 32 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)
```

14.14.2 Image Augmentation

Nhớ lại rằng tập dữ liệu giống chó này là một tập hợp con của tập dữ liệu ImageNet, có hình ảnh lớn hơn của tập dữ liệu CIFAR-10 vào năm Section 14.13. Sau đây liệt kê một vài thao tác nâng hình ảnh có thể hữu ích cho các hình ảnh tương đối lớn hơn.

```
transform_train = torchvision.transforms.Compose([
    # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
    # the original area and height-to-width ratio between 3/4 and 4/3. Then,
    # scale the image to create a new 224 x 224 image
    torchvision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                             ratio=(3.0/4.0, 4.0/3.0)),
    torchvision.transforms.RandomHorizontalFlip(),
    # Randomly change the brightness, contrast, and saturation
    torchvision.transforms.ColorJitter(brightness=0.4,
                                       contrast=0.4,
                                       saturation=0.4),
    # Add random noise
    torchvision.transforms.ToTensor(),
    # Standardize each channel of the image
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

Trong quá trình dự đoán, chúng tôi chỉ sử dụng các thao tác tiền xử lý hình ảnh mà không có tính ngẫu nhiên.

```
transform_test = torchvision.transforms.Compose([
    torchvision.transforms.Resize(256),
    # Crop a 224 x 224 square area from the center of the image
    torchvision.transforms.CenterCrop(224),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize([0.485, 0.456, 0.406],
                                     [0.229, 0.224, 0.225])])
```

14.14.3 Đọc dữ liệu

Như trong Section 14.13, chúng ta có thể đọc tập dữ liệu có tổ chức bao gồm các tệp hình ảnh thô.

```
train_ds, train_valid_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_train) for folder in ['train', 'train_valid']]

valid_ds, test_ds = [torchvision.datasets.ImageFolder(
    os.path.join(data_dir, 'train_valid_test', folder),
    transform=transform_test) for folder in ['valid', 'test']]
```

Dưới đây chúng ta tạo các trường hợp lặp dữ liệu giống như trong Section 14.13.

```
train_iter, train_valid_iter = [torch.utils.data.DataLoader(
    dataset, batch_size, shuffle=True, drop_last=True)
    for dataset in (train_ds, train_valid_ds)]

valid_iter = torch.utils.data.DataLoader(valid_ds, batch_size, shuffle=False,
                                         drop_last=True)

test_iter = torch.utils.data.DataLoader(test_ds, batch_size, shuffle=False,
                                        drop_last=False)
```

14.14.4 Tinh chỉnh một mô hình Pretrained Model

Một lần nữa, tập dữ liệu cho cuộc thi này là một tập hợp con của tập dữ liệu ImageNet. Do đó, chúng ta có thể sử dụng phương pháp được thảo luận trong Section 14.2 để chọn một mô hình được đào tạo trước trên tập dữ liệu ImageNet đầy đủ và sử dụng nó để trích xuất các tính năng hình ảnh được đưa vào mạng đầu ra quy mô nhỏ tùy chỉnh. API cấp cao của các framework deep learning cung cấp một loạt các mô hình được đào tạo trước trên tập dữ liệu ImageNet. Ở đây, chúng tôi chọn một mô hình ResNet-34 được đào tạo trước, nơi chúng tôi chỉ cần sử dụng lại đầu vào của lớp đầu ra của mô hình này (tức là các tính năng được trích xuất). Sau đó, chúng ta có thể thay thế lớp đầu ra ban đầu bằng một mạng đầu ra tùy chỉnh nhỏ có thể được đào tạo, chẳng hạn như xếp chồng hai lớp được kết nối hoàn toàn. Khác với thí nghiệm trong Section 14.2, những điều sau đây không đào tạo lại mô hình được đào tạo trước được sử dụng để trích xuất tính năng. Điều này làm giảm thời gian đào tạo và bộ nhớ để lưu trữ gradient.

Nhớ lại rằng chúng tôi chuẩn hóa hình ảnh bằng cách sử dụng phương tiện và độ lệch chuẩn của ba kênh RGB cho tập dữ liệu ImageNet đầy đủ. Trên thực tế, điều này cũng phù hợp với hoạt động tiêu chuẩn hóa bởi mô hình được đào tạo trước trên ImageNet.

```
def get_net(devices):
    finetune_net = nn.Sequential()
    finetune_net.features = torchvision.models.resnet34(pretrained=True)
    # Define a new output network (there are 120 output categories)
    finetune_net.output_new = nn.Sequential(nn.Linear(1000, 256),
                                            nn.ReLU(),
                                            nn.Linear(256, 120))
    # Move the model to devices
    finetune_net = finetune_net.to(devices[0])
    # Freeze parameters of feature layers
    for param in finetune_net.features.parameters():
```

(continues on next page)

(continued from previous page)

```
        param.requires_grad = False
    return finetune_net
```

Trước khi tính lấu, trước tiên chúng ta có được đầu vào của lớp đầu ra của mô hình được đào tạo sẵn, tức là tính năng trích xuất. Sau đó, chúng tôi sử dụng tính năng này làm đầu vào cho mạng đầu ra tùy chỉnh nhỏ của chúng tôi để tính toán tổn thất.

```
loss = nn.CrossEntropyLoss(reduction='none')

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        features, labels = features.to(devices[0]), labels.to(devices[0])
        outputs = net(features)
        l = loss(outputs, labels)
        l_sum += l.sum()
        n += labels.numel()
    return l_sum / n
```

14.14.5 Xác định chức năng đào tạo

Chúng tôi sẽ chọn mô hình và điều chỉnh các siêu tham số theo hiệu suất của mô hình trên bộ xác thực. Chức năng đào tạo mô hình `train` chỉ lặp lại các thông số của mạng đầu ra tùy chỉnh nhỏ.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.SGD((param for param in net.parameters()
                               if param.requires_grad), lr=lr,
                              momentum=0.9, weight_decay=wd)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_period, lr_decay)
    num_batches, timer = len(train_iter), d2l.Timer()
    legend = ['train loss']
    if valid_iter is not None:
        legend.append('valid loss')
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=legend)
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            features, labels = features.to(devices[0]), labels.to(devices[0])
            trainer.zero_grad()
            output = net(features)
            l = loss(output, labels).sum()
            l.backward()
            trainer.step()
            metric.add(l, labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
```

(continues on next page)

(continued from previous page)

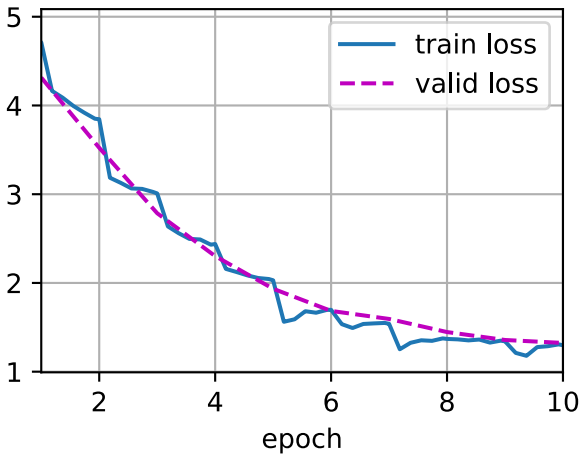
```
                            (metric[0] / metric[1], None))
        measures = f'train loss {metric[0] / metric[1]:.3f}'
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss.detach().cpu()))
        scheduler.step()
    if valid_iter is not None:
        measures += f', valid loss {valid_loss:.3f}'
    print(measures + f'\n{metric[1] * num_epochs / timer.sum():.1f}'
          f' examples/sec on {str(devices)}')
```

14.14.6 Đào tạo và xác thực mô hình

Bây giờ chúng ta có thể đào tạo và xác thực mô hình. Các siêu tham số sau đây đều có thể điều chỉnh được. Ví dụ, số lượng kỷ nguyên có thể được tăng lên. Bởi vì `lr_period` và `lr_decay` được đặt thành 2 và 0,9, tương ứng, tỷ lệ học tập của thuật toán tối ưu hóa sẽ được nhân với 0,9 sau mỗi 2 kỷ nguyên.

```
devices, num_epochs, lr, wd = d2l.try_all_gpus(), 10, 1e-4, 1e-4
lr_period, lr_decay, net = 2, 0.9, get_net(devices)
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)
```

```
train loss 1.297, valid loss 1.324
695.4 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```

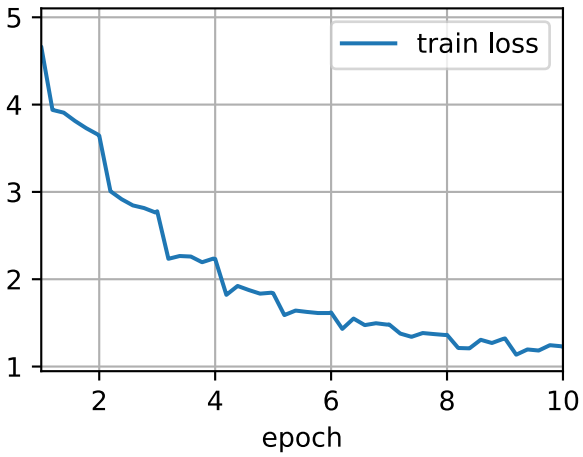


14.14.7 Phân loại bộ thử nghiệm và gửi kết quả trên Kaggle

Tương tự như bước cuối cùng trong Section 14.13, cuối cùng tất cả dữ liệu được dán nhãn (bao gồm cả bộ xác thực) được sử dụng để đào tạo mô hình và phân loại bộ thử nghiệm. Chúng tôi sẽ sử dụng mạng đầu ra tùy chỉnh được đào tạo để phân loại.

```
net = get_net(devices)
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output = torch.nn.functional.softmax(net(data.to(devices[0])), dim=0)
    preds.extend(output.cpu().detach().numpy())
ids = sorted(os.listdir(
    os.path.join(data_dir, 'train_valid_test', 'test', 'unknown')))
with open('submission.csv', 'w') as f:
    f.write('id,' + ','.join(train_valid_ds.classes) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')
```

```
train loss 1.225
969.5 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Đoạn mã trên sẽ tạo ra một tập tin `submission.csv` để gửi cho Kaggle theo cách tương tự được mô tả trong Section 5.10.

14.14.8 Tóm tắt

- Hình ảnh trong tập dữ liệu ImageNet lớn hơn (với các kích thước khác nhau) so với hình ảnh CIFAR-10. Chúng tôi có thể sửa đổi các thao tác nâng hình ảnh cho các tác vụ trên một tập dữ liệu khác.
- Để phân loại một tập hợp con của tập dữ liệu ImageNet, chúng ta có thể tận dụng các mô hình được đào tạo trước trên tập dữ liệu ImageNet đầy đủ để trích xuất các tính năng và chỉ đào tạo mạng đầu ra quy mô nhỏ tùy chỉnh. Điều này sẽ dẫn đến thời gian tính toán và chi phí bộ nhớ ít hơn.

14.14.9 Bài tập

1. Khi sử dụng bộ dữ liệu cạnh tranh Kaggle đầy đủ, bạn có thể đạt được kết quả gì khi tăng `batch_size` (kích thước lô) và `num_epochs` (số thời đại) trong khi đặt một số siêu tham số khác là `lr = 0.01`, `lr_period = 10` và `lr_decay = 0.1`?
2. Bạn có nhận được kết quả tốt hơn nếu bạn sử dụng một mô hình được đào tạo sâu hơn? Làm thế nào để bạn điều chỉnh các siêu tham số? Bạn có thể cải thiện hơn nữa kết quả?

Discussions[185]

[185] https://discuss.d2l.ai/t/1481

15 | Chế biến ngôn ngữ tự nhiên: Pretraining

Con người cần giao tiếp. Trong nhu cầu cơ bản này của tình trạng con người, một lượng lớn văn bản bằng văn bản đã được tạo ra trên cơ sở hàng ngày. Với văn bản phong phú trên phương tiện truyền thông xã hội, ứng dụng trò chuyện, email, đánh giá sản phẩm, bài báo tin tức, bài báo nghiên cứu và sách, điều quan trọng là cho phép máy tính hiểu chúng để cung cấp hỗ trợ hoặc đưa ra quyết định dựa trên ngôn ngữ của con người.

Xử lý ngôn ngữ tự nhiên nghiên cứu tương tác giữa máy tính và con người sử dụng ngôn ngữ tự nhiên. Trong thực tế, rất phổ biến khi sử dụng các kỹ thuật xử lý ngôn ngữ tự nhiên để xử lý và phân tích dữ liệu văn bản (ngôn ngữ tự nhiên của con người), chẳng hạn như các mô hình ngôn ngữ trong Section 9.3 và các mô hình dịch máy trong Section 10.5.

Để hiểu văn bản, chúng ta có thể bắt đầu bằng cách học các đại diện của nó. Tận dụng các chuỗi văn bản hiện có từ corpora lớn, *tự giám sát* đã được sử dụng rộng rãi để chuẩn bị các biểu diễn văn bản, chẳng hạn như bằng cách dự đoán một số phần ẩn của văn bản bằng cách sử dụng một số phần khác của văn bản xung quanh của chúng. Bằng cách này, các mô hình học thông qua giám sát từ dữ liệu văn bản * massive* mà không có nỗ lực ghi nhãn * chi phí*!

Như chúng ta sẽ thấy trong chương này, khi coi từng từ hoặc từ con như một mã thông báo riêng lẻ, biểu diễn của mỗi mã thông báo có thể được đào tạo trước bằng cách sử dụng các mô hình nhúng word2vec, Glove hoặc từ con trên corpora lớn. Sau khi đào tạo trước, biểu diễn của mỗi mã thông báo có thể là một vectơ, tuy nhiên, nó vẫn giữ nguyên bất kể bối cảnh là gì. Ví dụ, đại diện vector của "ngân hàng" là như nhau trong cả hai "đi đến ngân hàng để gửi một số tiền" và "đi đến ngân hàng để ngồi xuống". Do đó, nhiều mô hình pretraining gần đây hơn thích ứng đại diện của cùng một mã thông báo với các bối cảnh khác nhau. Trong số đó có BERT, một mô hình tự giám sát sâu hơn nhiều dựa trên bộ mã hóa biến áp. Trong chương này, chúng tôi sẽ tập trung vào cách chuẩn bị các đại diện như vậy cho văn bản, như được tô sáng trong Fig. 15.1.

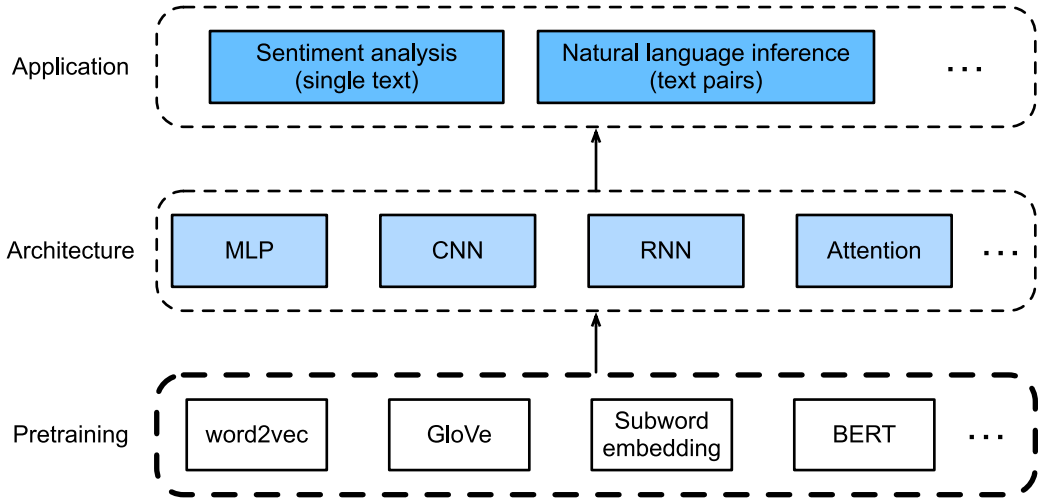


Fig. 15.1: Pretrained text representations can be fed to various deep learning architectures for different downstream natural language processing applications. This chapter focuses on the upstream text representation pretraining.

Để nhìn thấy bức tranh lớn, Fig. 15.1 cho thấy rằng các đại diện văn bản được đào tạo sẵn có thể được cung cấp cho một loạt các kiến trúc học sâu cho các ứng dụng xử lý ngôn ngữ tự nhiên hạ nguồn khác nhau. Chúng tôi sẽ bao gồm chúng trong Chapter 16.

15.1 Từ nhúng (word2vec)

Ngôn ngữ tự nhiên là một hệ thống phức tạp dùng để thể hiện ý nghĩa. Trong hệ thống này, từ ngữ là đơn vị cơ bản của nghĩa. Như tên của nó, *vectơ từ* là các vectơ được sử dụng để biểu diễn các từ, và cũng có thể được coi là vectơ tính năng hoặc biểu diễn của các từ. Kỹ thuật lập bản đồ từ với vectơ thực được gọi là *word embedding*. Trong những năm gần đây, nhúng từ đã dần trở thành kiến thức cơ bản về xử lý ngôn ngữ tự nhiên.

15.1.1 Vectơ một nóng là một lựa chọn tồi

Chúng tôi sử dụng vectơ một nóng để biểu diễn các từ (ký tự là từ) trong Section 9.5. Giả sử số từ khác nhau trong từ điển (cỡ từ điển) là N, và mỗi từ tương ứng với một số nguyên (index) khác nhau từ 0 đến $N-1$. Để có được biểu diễn vectơ một nóng cho bất kỳ từ nào có chỉ số i, chúng ta tạo một vector chiều dài-N với tất cả 0s và đặt phần tử ở vị trí i thành 1. Bằng cách này, mỗi từ được biểu diễn dưới dạng vectơ có chiều dài N, và nó có thể được sử dụng trực tiếp bởi các mạng thần kinh.

Mặc dù vectơ từ một nóng rất dễ xây dựng, chúng thường không phải là một lựa chọn tốt. Một lý do chính là các vectơ từ một nóng không thể diễn tả chính xác sự tương đồng giữa các từ khác nhau, chẳng hạn như tương tự * cosine* mà chúng ta thường sử dụng. Đối với vectơ $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, sự tương đồng cosin của chúng là cosin của góc giữa chúng:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \tag{15.1.1}$$

Vì sự tương đồng cosin giữa các vectơ một nóng của hai từ khác nhau bất kỳ là 0, các vectơ một nóng không thể mã hóa các điểm tương đồng giữa các từ.

15.1.2 Word2vec tự giám sát

Công cụ word2vec[186] đã được đề xuất để giải quyết vấn đề trên. Nó ánh xạ từng từ thành một vectơ có độ dài cố định, và các vectơ này có thể thể hiện tốt hơn mối quan hệ tương đồng và tương tự giữa các từ khác nhau. Công cụ word2vec chứa hai mô hình, cụ thể là *skip-gram* (Mikolov et al., 2013b) và * túi liên từ* (CBOW) (Mikolov et al., 2013a). Đối với các đại diện có ý nghĩa về mặt ngữ nghĩa, đào tạo của họ dựa vào xác suất có điều kiện có thể được xem như dự đoán một số từ bằng cách sử dụng một số từ xung quanh của họ trong thể. Vì sự giám sát đến từ dữ liệu không có nhãn, cả bỏ qua gram và túi từ liên tục đều là các mô hình tự giám sát.

Sau đây, chúng tôi sẽ giới thiệu hai mô hình này và phương pháp đào tạo của họ.

15.1.3 Mô hình Skip-Gram

Mô hình *skip-gram* giả định rằng một từ có thể được sử dụng để tạo ra các từ xung quanh của nó trong một chuỗi văn bản. Lấy chuỗi văn bản "các", "người đàn ông", "yêu thương", "anh ấy", "con trai" làm ví dụ. Hãy để chúng tôi chọn "loves" làm từ *center* và đặt kích thước cửa sổ ngữ cảnh thành 2. Như thể hiện trong Fig. 15.1.1, với từ trung tâm "yêu thương", mô hình skip-gram xem xét xác suất có điều kiện để tạo ra các từ ngữ cảnh *: "the", "man", "his", và "con trai", không quá 2 từ cách từ trung tâm:

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} \mid \text{"loves"}). \tag{15.1.2}$$

Giả sử rằng các từ ngữ cảnh được tạo độc lập cho từ trung tâm (tức là độc lập có điều kiện). Trong trường hợp này, xác suất có điều kiện ở trên có thể được viết lại dưới dạng

$$P(\text{"the"} \mid \text{"loves"}) \cdot P(\text{"man"} \mid \text{"loves"}) \cdot P(\text{"his"} \mid \text{"loves"}) \cdot P(\text{"son"} \mid \text{"loves"}). \tag{15.1.3}$$

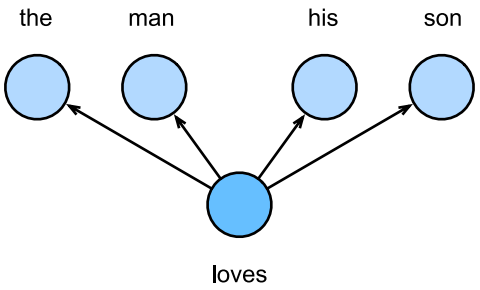


Fig. 15.1.1: The skip-gram model considers the conditional probability of generating the surrounding context words given a center word.

Trong mô hình skip-gram, mỗi từ có hai biểu diễn d-dimensional-vector để tính xác suất có điều kiện. Cụ thể hơn, đối với bất kỳ từ nào có chỉ mục i trong từ điển, biểu thị bằng $\mathbf{v}_i \in \mathbb{R}^d$ và $\mathbf{u}_i \in \mathbb{R}^d$ hai vectơ của nó khi được sử dụng như một từ *center* và một từ *context*, tương ứng. Xác suất có điều kiện tạo ra bất kỳ từ ngữ cảnh nào w_o (với chỉ số o trong từ điển) cho từ trung tâm w_c (với chỉ số c trong từ điển) có thể được mô hình hóa bằng một hoạt động softmax trên các sản phẩm chấm vector:

$$P(w_o \mid w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \tag{15.1.4}$$

nơi chỉ số từ vựng đặt $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$. Cho một chuỗi văn bản có độ dài T, trong đó từ lúc bước t được ký hiệu là $w^{(t)}$. Giả sử rằng các từ ngữ cảnh được tạo độc lập cho bất kỳ từ trung tâm nào. Đối với kích

[186] https://code.google.com/archive/p/word2vec/

thước cửa sổ ngữ cảnh m, hàm khả năng của mô hình skip-gram là xác suất tạo ra tất cả các từ ngữ cảnh cho bất kỳ từ trung tâm nào:

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{15.1.5}$$

trong đó bất kỳ bước thời gian nào nhỏ hơn 1 hoặc lớn hơn T có thể được bỏ qua.

Đào tạo

Các tham số mô hình skip-gram là vector từ trung tâm và vector từ ngữ cảnh cho mỗi từ trong từ vựng. Trong đào tạo, chúng ta tìm hiểu các tham số mô hình bằng cách tối đa hóa hàm khả năng (tức là ước tính khả năng tối đa). Điều này tương đương với việc giảm thiểu hàm mất sau:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} \log P(w^{(t+j)} \mid w^{(t)}). \tag{15.1.6}$$

Khi sử dụng stochastic gradient descent để giảm thiểu tổn thất, trong mỗi lần lặp lại, chúng ta có thể lấy mẫu ngẫu nhiên một dãy con ngắn hơn để tính toán gradient (stochastic) cho dãy tiếp theo này để cập nhật các tham số mô hình. Để tính toán gradient (stochastic) này, chúng ta cần phải có được gradient của xác suất điều kiện nhật ký đối với vector từ trung tâm và vector từ ngữ cảnh. Nói chung, theo (15.1.4) xác suất có điều kiện nhật ký liên quan đến bất kỳ cặp nào của từ trung tâm w_c và từ ngữ cảnh w_o là

$$\log P(w_o \mid w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \tag{15.1.7}$$

Thông qua sự khác biệt, chúng ta có thể có được gradient của nó đối với vector từ trung tâm $\mathbf{v}_c$ như

$$\begin{aligned} \frac{\partial \log P(w_o \mid w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j \mid w_c) \mathbf{u}_j. \end{aligned} \tag{15.1.8}$$

Lưu ý rằng việc tính toán trong (15.1.8) đòi hỏi xác suất có điều kiện của tất cả các từ trong từ điển với w_c là từ trung tâm. Các gradient cho các vectơ từ khác có thể thu được theo cùng một cách.

Sau khi đào tạo, đối với bất kỳ từ nào có chỉ mục i trong từ điển, chúng tôi có được cả hai vectơ từ $\mathbf{v}_i$ (là từ trung tâm) và $\mathbf{u}_i$ (như từ ngữ cảnh). Trong các ứng dụng xử lý ngôn ngữ tự nhiên, các vectơ từ trung tâm của mô hình skip-gram thường được sử dụng làm biểu diễn từ.

15.1.4 Mô hình túi từ liên tục (CBOW)

Mô hình túi từ* (CBOW) liên tục tương tự như mô hình bỏ qua gram. Sự khác biệt lớn so với mô hình skip-gram là mô hình túi từ liên tục giả định rằng một từ trung tâm được tạo ra dựa trên các từ ngữ cảnh xung quanh của nó trong chuỗi văn bản. Ví dụ, trong cùng một chuỗi văn bản "the", "man", "loves", "his", và "son", với "yêu" là từ trung tâm và kích thước cửa sổ ngữ cảnh là 2, túi liên tục của mô hình từ xem xét xác suất có

điều kiện tạo ra từ trung tâm "yêu" dựa trên các từ ngữ cảnh "the", "man", "his" và "son "(như thể hiện trong `fig_cbow`), đó là

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \tag{15.1.9}$$

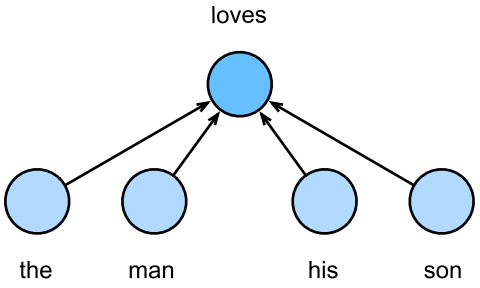


Fig. 15.1.2: The continuous bag of words model considers the conditional probability of generating the center word given its surrounding context words. :label: fig_cbow

Vì có nhiều từ ngữ cảnh trong mô hình túi từ liên tục, các vectơ từ ngữ cảnh này được tính trung bình trong việc tính xác suất có điều kiện. Cụ thể, đối với bất kỳ từ nào có chỉ số i trong từ điển, biểu thị bằng $\mathbf{v}_i \in \mathbb{R}^d$ và $\mathbf{u}_i \in \mathbb{R}^d$ hai vectơ của nó khi được sử dụng như một từ *context* và một từ *center* (nghĩa được chuyển trong mô hình skip-gram), tương ứng. Xác suất có điều kiện tạo ra bất kỳ từ trung tâm nào w_c (với chỉ số c trong từ điển) cho các từ ngữ cảnh xung quanh của nó $w_{o_1}, \ldots, w_{o_{2m}}$ (với chỉ số $o_1, \ldots, o_{2m}$ trong từ điển) có thể được mô hình hóa bởi

$$P(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top (\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top (\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}})\right)}. \tag{15.1.10}$$

Đối với ngắn gọn, hãy để $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$ và $\bar{\mathbf{v}}_o = \left(\mathbf{v}_{o_1} + \ldots, +\mathbf{v}_{o_{2m}}\right)/(2m)$. Sau đó, (15.1.10) có thể được đơn giản hóa như

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\mathbf{u}_c^\top \bar{\mathbf{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o\right)}. \tag{15.1.11}$$

Cho một chuỗi văn bản có chiều dài T, trong đó từ lúc bước t được ký hiệu là $w^{(t)}$. Đối với kích thước cửa sổ ngữ cảnh m, chức năng khả năng của túi liên tục của mô hình từ là xác suất tạo ra tất cả các từ trung tâm cho các từ ngữ cảnh của chúng:

$$\prod_{t=1}^{T} P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{15.1.12}$$

Đào tạo

Đào tạo túi liên tục của các mô hình từ gần giống như đào tạo mô hình skip-gram. Ước tính khả năng tối đa của mô hình túi từ liên tục tương đương với việc giảm thiểu chức năng mất sau:

$$-\sum_{t=1}^{T} \log P(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}). \tag{15.1.13}$$

Chú ý rằng

$$\log P(w_c \mid \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp\left(\mathbf{u}_i^\top \bar{\mathbf{v}}_o \right) \right). \tag{15.1.14}$$

Thông qua sự khác biệt, chúng ta có thể có được gradient của nó đối với bất kỳ vector từ ngữ cảnh $\mathbf{v}_{o_i}$ ($i = 1, \ldots, 2m$) như

$$\frac{\partial \log P(w_c \mid \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o)\mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left(\mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j \mid \mathcal{W}_o)\mathbf{u}_j \right). \tag{15.1.15}$$

Các gradient cho các vectơ từ khác có thể thu được theo cùng một cách. Không giống như mô hình skip-gram, mô hình túi từ liên tục thường sử dụng vectơ từ ngữ cảnh làm biểu diễn từ.

15.1.5 Tóm tắt

- Vectơ từ là các vectơ dùng để biểu diễn các từ, và cũng có thể được coi là vectơ đặc trưng hoặc biểu diễn các từ. Kỹ thuật lập bản đồ từ với vectơ thực được gọi là nhúng từ.
- Công cụ word2vec chứa cả mô hình bỏ qua gram và túi liên tục.
- Mô hình skip-gram giả định rằng một từ có thể được sử dụng để tạo ra các từ xung quanh của nó trong một chuỗi văn bản; trong khi mô hình túi từ liên tục giả định rằng một từ trung tâm được tạo ra dựa trên các từ ngữ cảnh xung quanh của nó.

15.1.6 Bài tập

1. Độ phức tạp tính toán để tính từng gradient là gì? Điều gì có thể là vấn đề nếu kích thước từ điển là rất lớn?
2. Một số cụm từ cố định trong tiếng Anh bao gồm nhiều từ, chẳng hạn như "new york". Làm thế nào để đào tạo vectơ từ của họ? Hint: see Section 4 in the word2vec paper (Mikolov et al., 2013b).
3. Chúng ta hãy suy ngẫm về thiết kế word2vec bằng cách lấy mô hình skip-gram làm ví dụ. Mối quan hệ giữa tích chấm của hai vectơ từ trong mô hình skip-gram và sự tương đồng cosin là gì? Đối với một cặp từ có ngữ nghĩa tương tự, tại sao sự tương đồng cosin của vectơ từ của chúng (được đào tạo bởi mô hình skip-gram) có thể cao?

Discussions[187]

15.2 Đào tạo gần đúng

Nhớ lại các cuộc thảo luận của chúng tôi trong Section 15.1. Ý tưởng chính của mô hình skip-gram là sử dụng các phép toán softmax để tính xác suất có điều kiện tạo ra một từ ngữ cảnh w_o dựa trên từ trung tâm đã cho w_c trong (15.1.4), có tổn thất logarit tương ứng được đưa ra bởi ngược lại (15.1.7).

Do tính chất của hoạt động softmax, vì một từ ngữ cảnh có thể là bất cứ ai trong từ điển $\mathcal{V}$, ngược lại với (15.1.7) chứa tổng các mục nhiều như toàn bộ kích thước của từ vựng. Do đó, tính toán gradient cho mô hình skip-gram trong (15.1.8) và cho mô hình túi-từ liên tục trong (15.1.15) cả hai đều chứa tổng. Thật không may,

[187] https://discuss.d2l.ai/t/381

chi phí tính toán cho gradient như vậy tổng hợp trên một từ điển lớn (thường với hàng trăm ngàn hoặc hàng triệu từ) là rất lớn!

Để giảm độ phức tạp tính toán nói trên, phần này sẽ giới thiệu hai phương pháp đào tạo gần đúng: *lấy mẫu tiêu cực* và * phân cấp softmax*. Do sự giống nhau giữa mô hình skip-gram và mô hình túi từ liên tục, chúng tôi sẽ chỉ lấy mô hình skip-gram làm ví dụ để mô tả hai phương pháp đào tạo gần đúng này.

15.2.1 Lấy mẫu âm

Lấy mẫu âm sửa đổi chức năng mục tiêu ban đầu. Với cửa sổ ngữ cảnh của một từ trung tâm w_c, thực tế là bất kỳ từ nào (ngữ cảnh) w_o xuất phát từ cửa sổ ngữ cảnh này được coi là một sự kiện với xác suất được mô hình hóa bởi

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \tag{15.2.1}$$

trong đó σ sử dụng định nghĩa của hàm kích hoạt sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \tag{15.2.2}$$

Chúng ta hãy bắt đầu bằng cách tối đa hóa xác suất chung của tất cả các sự kiện như vậy trong chuỗi văn bản để đào tạo nhúng từ. Cụ thể, đưa ra một chuỗi văn bản chiều dài T, biểu thị bằng $w^{(t)}$ từ tại bước thời điểm t và để cho kích thước cửa sổ ngữ cảnh là m, xem xét tối đa hóa xác suất chung

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \tag{15.2.3}$$

Tuy nhiên, (15.2.3) chỉ xem xét những sự kiện liên quan đến các ví dụ tích cực. Kết quả là, xác suất chung trong (15.2.3) được tối đa hóa đến 1 chỉ khi tất cả các vectơ từ bằng vô cực. Tất nhiên, kết quả như vậy là vô nghĩa. To make the objective mục tiêu function chức năng more meaningfulý nghĩa, *lấy mẫu tiêu cực* thêm các ví dụ tiêu cực lấy mẫu từ một phân phối được xác định trước.

Biểu thị bởi S sự kiện mà một từ ngữ cảnh w_o xuất phát từ cửa sổ ngữ cảnh của một từ trung tâm w_c. Đối với sự kiện này liên quan đến w_o, từ một phân phối được xác định trước $P(w)$ mẫu K * tiếng ồn từ * không phải từ cửa sổ ngữ cảnh này. Biểu thị bởi N_k sự kiện rằng một từ tiếng ồn w_k ($k = 1, \ldots, K$) không đến từ cửa sổ ngữ cảnh của w_c. Giả sử rằng những sự kiện này liên quan đến cả ví dụ tích cực và các ví dụ tiêu cực $S, N_1, \ldots, N_K$ là độc lập lẫn nhau. Lấy mẫu âm viết lại xác suất chung (chỉ liên quan đến các ví dụ tích cực) trong (15.2.3) như

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m,\ j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \tag{15.2.4}$$

trong đó xác suất có điều kiện được xấp xỉ thông qua các sự kiện $S, N_1, \ldots, N_K$:

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1,\ w_k \sim P(w)}^{K} P(D = 0 \mid w^{(t)}, w_k). \tag{15.2.5}$$

Biểu thị bởi i_t và h_k các chỉ số của một từ $w^{(t)}$ tại bước thời gian t của một chuỗi văn bản và một từ tiếng ồn

w_k, tương ứng. Sự mất mát logarit đối với xác suất có điều kiện trong (15.2.5) là

$$\begin{aligned}-\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D=1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log P(D=0 \mid w^{(t)}, w_k) \\ &= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log\left(1-\sigma\left(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right)\right) \\ &= -\log \sigma\left(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}\right) - \sum_{k=1,\ w_k \sim P(w)}^{K} \log \sigma\left(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}\right).\end{aligned} \tag{15.2.6}$$

Chúng ta có thể thấy rằng bây giờ chi phí tính toán cho gradient ở mỗi bước đào tạo không liên quan gì đến kích thước từ điển, nhưng tuyến tính phụ thuộc vào K. Khi đặt siêu tham số K thành một giá trị nhỏ hơn, chi phí tính toán cho gradient ở mỗi bước đào tạo với lấy mẫu âm nhỏ hơn.

15.2.2 Phân cấp Softmax

Như một phương pháp đào tạo gần đúng thay thế, *softmaxphân học* sử dụng cây nhị phân, một cấu trúc dữ liệu minh họa trong Fig. 15.2.1, trong đó mỗi nút lá của cây đại diện cho một từ trong từ điển $\mathcal{V}$.

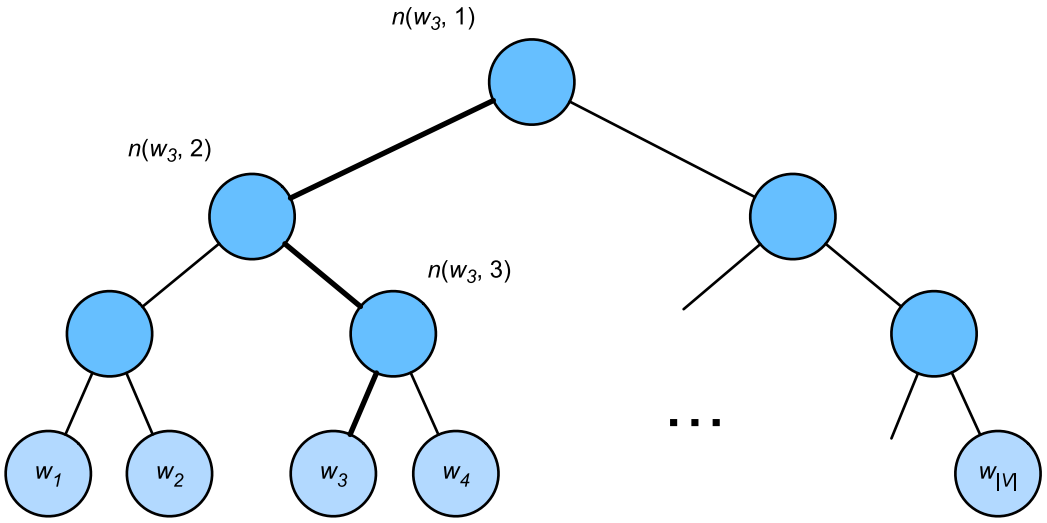


Fig. 15.2.1: Hierarchical softmax for approximate training, where each leaf node of the tree represents a word in the dictionary.

Biểu thị bằng $L(w)$ số nút (bao gồm cả hai đầu) trên đường dẫn từ nút gốc đến nút lá biểu diễn từ w trong cây nhị phân. Hãy để $n(w, j)$ là nút j^{th} trên đường dẫn này, với vector từ ngữ cảnh của nó là $\mathbf{u}_{n(w,j)}$. Ví dụ, $L(w_3) = 4$ trong Fig. 15.2.1. Softmax phân cấp xấp xỉ xác suất có điều kiện trong (15.1.4) như

$$P(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left(\llbracket n(w_o, j+1) = \text{leftChild}(n(w_o, j)) \rrbracket \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c\right), \tag{15.2.7}$$

trong đó hàm σ được định nghĩa trong (15.2.2), và leftChild(n) là nút con trái của nút n: nếu x là đúng, $ [! [x]!] = 1$; otherwise $ [! [x]!] = -1$.

Để minh họa, chúng ta hãy tính toán xác suất có điều kiện tạo từ w_3 cho từ w_c trong Fig. 15.2.1. Điều này đòi hỏi các sản phẩm chấm giữa từ vector $\mathbf{v}_c$ của w_c và vectơ nút không lá trên đường đi (đường dẫn in đậm trong Fig. 15.2.1) từ gốc đến w_3, được đi qua trái, phải, sau đó trái:

$$P(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c). \tag{15.2.8}$$

Kể từ $\sigma(x) + \sigma(-x) = 1$, nó cho rằng xác suất có điều kiện của việc tạo ra tất cả các từ trong từ điển $\mathcal{V}$ dựa trên bất kỳ từ nào w_c tổng hợp lên đến một:

$$\sum_{w \in \mathcal{V}} P(w \mid w_c) = 1. \tag{15.2.9}$$

May mắn thay, vì $L(w_o) - 1$ theo thứ tự $\mathcal{O}(\log_2|\mathcal{V}|)$ do cấu trúc cây nhị phân, khi kích thước từ điển $\mathcal{V}$ rất lớn, chi phí tính toán cho mỗi bước đào tạo sử dụng softmax phân cấp được giảm đáng kể so với điều đó mà không cần đào tạo gần đúng.

15.2.3 Tóm tắt

- Lấy mẫu âm xây dựng hàm mất bằng cách xem xét các sự kiện độc lập lẫn nhau liên quan đến cả ví dụ tích cực và tiêu cực. Chi phí tính toán để đào tạo phụ thuộc tuyến tính vào số lượng từ tiếng ồn ở mỗi bước.
- Phân cấp softmax cấu tạo hàm mất bằng cách sử dụng đường dẫn từ nút gốc đến nút lá trong cây nhị phân. Chi phí tính toán cho đào tạo phụ thuộc vào logarit của kích thước từ điển ở mỗi bước.

15.2.4 Bài tập

1. Làm thế nào chúng ta có thể lấy mẫu từ tiếng ồn trong lấy mẫu tiêu cực?
2. Xác minh rằng (15.2.9) nắm giữ.
3. Làm thế nào để đào tạo mô hình túi từ liên tục bằng cách sử dụng lấy mẫu âm và softmax phân cấp, tương ứng?

Discussions[188]

15.3 Các Dataset cho Pretraining Word Embeddings

Bây giờ chúng ta đã biết các chi tiết kỹ thuật của các mô hình word2vec và các phương pháp đào tạo gần đúng, chúng ta hãy đi qua các triển khai của họ. Cụ thể, chúng tôi sẽ lấy mô hình skip-gram trong Section 15.1 và lấy mẫu âm trong Section 15.2 làm ví dụ. Trong phần này, chúng ta bắt đầu với tập dữ liệu để đào tạo trước mô hình nhúng từ: định dạng ban đầu của dữ liệu sẽ được chuyển thành các minibatches có thể được lặp lại trong quá trình đào tạo.

```
import math
import os
import random
import torch
from d2l import torch as d2l
```

[188] https://discuss.d2l.ai/t/382

15.3.1 Đọc tập dữ liệu

Tập dữ liệu mà chúng tôi sử dụng ở đây là Penn Tree Bank (PTB)[189]. Cơ sở này được lấy mẫu từ các bài báo của Wall Street Journal, được chia thành các bộ đào tạo, xác nhận và kiểm tra. Ở định dạng ban đầu, mỗi dòng của tệp văn bản đại diện cho một câu của các từ được phân tách bằng dấu cách. Ở đây chúng ta coi từng từ như một mã thông báo.

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                       '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    """Load the PTB dataset into a list of text lines."""
    data_dir = d2l.download_extract('ptb')
    # Read the training set.
    with open(os.path.join(data_dir, 'ptb.train.txt')) as f:
        raw_text = f.read()
    return [line.split() for line in raw_text.split('\n')]

sentences = read_ptb()
f'# sentences: {len(sentences)}'
```

```
Downloading ../data/ptb.zip from http://d2l-data.s3-accelerate.amazonaws.com/
↪ptb.zip...
```

```
'# sentences: 42069'
```

Sau khi đọc bộ đào tạo, chúng tôi xây dựng một từ vựng cho corpus, trong đó bất kỳ từ nào xuất hiện dưới 10 lần được thay thế bằng mã thông báo “”. Lưu ý rằng tập dữ liệu gốc cũng chứa “” token đại diện cho các từ hiếm (không xác định).

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

```
'vocab size: 6719'
```

15.3.2 Lấy mẫu phụ

Dữ liệu văn bản thường có các từ tần số cao như “the”, “a”, và “in”: chúng thậm chí có thể xảy ra hàng tỷ lần trong thể rất lớn. Tuy nhiên, những từ này thường đồng xuất hiện với nhiều từ khác nhau trong các cửa sổ ngữ cảnh, cung cấp ít tín hiệu hữu ích. Ví dụ, hãy xem xét từ “chip” trong một cửa sổ ngữ cảnh: trực giác sự xuất hiện của nó với một từ tần số thấp “intel” hữu ích hơn trong đào tạo hơn là đồng xuất hiện với một từ tần số cao “a”. Hơn nữa, đào tạo với một lượng lớn các từ (tần số cao) là chậm. Do đó, khi đào tạo từ nhúng mô hình, các từ tần số cao có thể là * mẫu con* (Mikolov et al., 2013b). Cụ thể, mỗi từ được lập chỉ mục w_i trong tập dữ liệu sẽ bị loại bỏ với xác suất

$$P(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right), \tag{15.3.1}$$

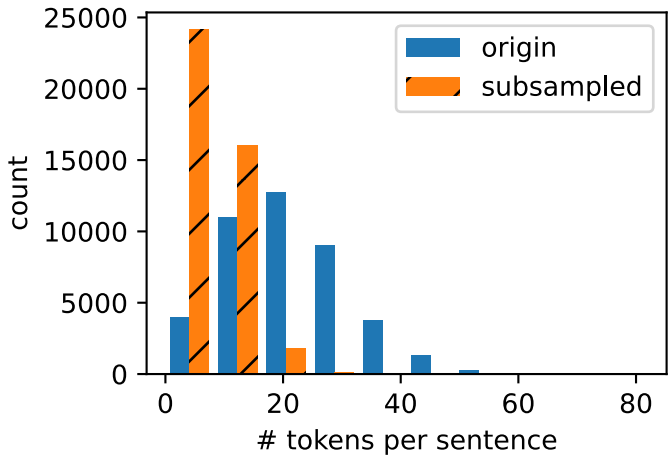
[189] https://catalog.ldc.upenn.edu/LDC99T42

trong đó $f(w_i)$ là tỷ lệ của số từ w_i với tổng số từ trong tập dữ liệu và hằng số t là một siêu tham số (10^{-4} trong thí nghiệm). Chúng ta có thể thấy rằng chỉ khi tần số tương đối $f(w_i) > t$, từ w_i mới có thể bị loại bỏ và tần số tương đối của từ càng cao thì xác suất bị loại bỏ càng lớn.

```
#@save
def subsample(sentences, vocab):
    """Subsample high-frequency words."""
    # Exclude unknown tokens '<unk>'
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Return True if `token` is kept during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

Đoạn mã sau vẽ biểu đồ của số lượng mã thông báo trên mỗi câu trước và sau khi lấy mẫu. Đúng như dự đoán, subsampling rút ngắn đáng kể các câu bằng cách thả các từ tần số cao, điều này sẽ dẫn đến tăng tốc đào tạo.

```
d2l.show_list_len_pair_hist(['origin', 'subsampled'], '# tokens per sentence',
                            'count', sentences, subsampled);
```



Đối với mã thông báo riêng lẻ, tỷ lệ lấy mẫu của từ tần số cao "the" nhỏ hơn 1/20.

```
def compare_counts(token):
    return (f'# of "{token}": '
            f'before={sum([l.count(token) for l in sentences])}, '
            f'after={sum([l.count(token) for l in subsampled])}')

compare_counts('the')
```

```
'# of "the": before=50770, after=2033'
```

Ngược lại, các từ tần số thấp "tham gia" được giữ hoàn toàn.

```
compare_counts('join')
```

```
'# of "join": before=45, after=45'
```

Sau khi lấy mẫu đăng ký, chúng tôi ánh xạ mã thông báo đến các chỉ số của họ cho corpus.

```
corpus = [vocab[line] for line in subsampled]
corpus[:3]
```

```
[[], [392, 2115, 145, 274, 406], [5277, 3054, 1580]]
```

15.3.3 Trích xuất từ trung tâm và từ ngữ cảnh

Hàm `get_centers_and_contexts` sau trích xuất tất cả các từ trung tâm và từ ngữ cảnh của chúng từ `corpus`. Nó đồng đều mẫu một số nguyên giữa 1 và `max_window_size` một cách ngẫu nhiên như kích thước cửa sổ ngữ cảnh. Đối với bất kỳ từ trung tâm nào, những từ có khoảng cách từ nó không vượt quá kích thước cửa sổ ngữ cảnh được lấy mẫu là các từ ngữ cảnh của nó.

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    """Return center words and context words in skip-gram."""
    centers, contexts = [], []
    for line in corpus:
        # To form a "center word--context word" pair, each sentence needs to
        # have at least 2 words
        if len(line) < 2:
            continue
        centers += line
        for i in range(len(line)):  # Context window centered at `i`
            window_size = random.randint(1, max_window_size)
            indices = list(range(max(0, i - window_size),
                                 min(len(line), i + 1 + window_size)))
            # Exclude the center word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

Tiếp theo, chúng ta tạo ra một tập dữ liệu nhân tạo có chứa hai câu 7 và 3 từ, tương ứng. Hãy để kích thước cửa sổ ngữ cảnh tối đa là 2 và in tất cả các từ trung tâm và các từ ngữ cảnh của chúng.

```
tiny_dataset = [list(range(7)), list(range(7, 10))]
print('dataset', tiny_dataset)
for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
    print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1, 2]
center 1 has contexts [0, 2, 3]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [1, 2, 4, 5]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [3, 4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8, 9]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```

Khi đào tạo trên tập dữ liệu PTB, chúng tôi đặt kích thước cửa sổ ngữ cảnh tối đa là 5. Sau đây trích xuất tất cả các từ trung tâm và các từ ngữ cảnh của chúng trong tập dữ liệu.

```
all_centers, all_contexts = get_centers_and_contexts(corpus, 5)
f'# center-context pairs: {sum([len(contexts) for contexts in all_contexts])}'
```

```
'# center-context pairs: 1497165'
```

15.3.4 Lấy mẫu âm

Chúng tôi sử dụng lấy mẫu tiêu cực cho đào tạo gần đúng. Để lấy mẫu các từ nhiễu theo một phân phối được xác định trước, chúng ta xác định lớp `RandomGenerator` sau, trong đó phân phối lấy mẫu (có thể không chuẩn hóa) được truyền qua đối số `sampling_weights`.

```
#@save
class RandomGenerator:
    """Randomly draw among {1, ..., n} according to n sampling weights."""
    def __init__(self, sampling_weights):
        # Exclude
        self.population = list(range(1, len(sampling_weights) + 1))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            # Cache `k` random sampling results
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i - 1]
```

Ví dụ: chúng ta có thể vẽ 10 biến ngẫu nhiên X trong số các chỉ số 1, 2 và 3 với xác suất lấy mẫu $P(X = 1) = 2/9, P(X = 2) = 3/9$ và $P(X = 3) = 4/9$ như sau.

Đối với một cặp từ trung tâm và từ ngữ cảnh, chúng tôi lấy mẫu ngẫu nhiên `K` (5 trong thí nghiệm) các từ tiếng ồn. Theo các đề xuất trong bài báo word2vec, xác suất lấy mẫu $P(w)$ của một từ tiếng ồn w được đặt thành tần số tương đối của nó trong từ điển nâng lên công suất 0,75 (Mikolov et al., 2013b).

```
#@save
def get_negatives(all_contexts, vocab, counter, K):
    """Return noise words in negative sampling."""
    # Sampling weights for words with indices 1, 2, ... (index 0 is the
    # excluded unknown token) in the vocabulary
    sampling_weights = [counter[vocab.to_tokens(i)]**0.75
                        for i in range(1, len(vocab))]
    all_negatives, generator = [], RandomGenerator(sampling_weights)
    for contexts in all_contexts:
        negatives = []
        while len(negatives) < len(contexts) * K:
            neg = generator.draw()
            # Noise words cannot be context words
            if neg not in contexts:
                negatives.append(neg)
        all_negatives.append(negatives)
    return all_negatives

all_negatives = get_negatives(all_contexts, vocab, counter, 5)
```

15.3.5 Tải ví dụ đào tạo trong Minibatches

Sau khi tất cả các từ trung tâm cùng với các từ ngữ cảnh và các từ tiếng ồn được lấy mẫu được trích xuất, chúng sẽ được chuyển thành các ví dụ nhỏ có thể được tải lặp lại trong quá trình đào tạo.

Trong một minibatch, ví dụ i^{th} bao gồm một từ trung tâm và n_i từ ngữ cảnh của nó và m_i từ nhiễu. Do kích thước cửa sổ ngữ cảnh khác nhau, $n_i + m_i$ thay đổi cho i khác nhau. Do đó, đối với mỗi ví dụ, chúng tôi nối các từ ngữ cảnh và các từ nhiễu của nó trong biến `contexts_negatives` và pad số không cho đến khi độ dài nối đạt $\max_i n_i + m_i$ (`max_len`). Để loại trừ các miếng đệm trong tính toán tổn thất, chúng tôi xác định một biến mặt nạ `masks`. Có một sự tương ứng một-một giữa các phần tử trong `masks` và các phần tử trong `contexts_negatives`, trong đó số không (nếu không) trong `masks` tương ứng với các miếng đệm trong `contexts_negatives`.

Để phân biệt giữa các ví dụ tích cực và tiêu cực, chúng ta tách các từ ngữ cảnh khỏi các từ nhiễu trong `contexts_negatives` thông qua một biến `labels`. Tương tự như `masks`, cũng có sự tương ứng một-một giữa các phần tử trong `labels` và các phần tử trong `contexts_negatives`, trong đó các phần tử (nếu không số không) trong `labels` tương ứng với các từ ngữ cảnh (ví dụ tích cực) trong `contexts_negatives`.

Ý tưởng trên được thực hiện trong hàm `batchify` sau. Đầu vào của nó `data` là một danh sách có độ dài bằng với kích thước lô, trong đó mỗi phần tử là một ví dụ bao gồm từ trung tâm `center`, các từ ngữ cảnh của nó `context`, và các từ nhiễu của nó `negative`. Hàm này trả về một minibatch có thể được nạp để tính toán trong quá trình đào tạo, chẳng hạn như bao gồm biến mask.

```
#@save
def batchify(data):
    """Return a minibatch of examples for skip-gram with negative sampling."""
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, masks, labels = [], [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
```

(continues on next page)

(continued from previous page)

```
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]
    return (torch.tensor(centers).reshape((-1, 1)), torch.tensor(
        contexts_negatives), torch.tensor(masks), torch.tensor(labels))
```

Hãy để chúng tôi kiểm tra chức năng này bằng cách sử dụng một minibatch gồm hai ví dụ.

```
x_1 = (1, [2, 2], [3, 3, 3, 3])
x_2 = (1, [2, 2, 2], [3, 3])
batch = batchify((x_1, x_2))

names = ['centers', 'contexts_negatives', 'masks', 'labels']
for name, data in zip(names, batch):
    print(name, '=', data)
```

```
centers = tensor([[1],
        [1]])
contexts_negatives = tensor([[2, 2, 3, 3, 3, 3],
        [2, 2, 2, 3, 3, 0]])
masks = tensor([[1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 0]])
labels = tensor([[1, 1, 0, 0, 0, 0],
        [1, 1, 1, 0, 0, 0]])
```

15.3.6 Đặt tất cả mọi thứ lại với nhau

Cuối cùng, chúng ta định nghĩa hàm `load_data_ptb` đọc tập dữ liệu PTB và trả về bộ lặp dữ liệu và từ vựng.

```
#@save
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    """Download the PTB dataset and then load it into memory."""
    num_workers = d2l.get_dataloader_workers()
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled, counter = subsample(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centers, all_contexts = get_centers_and_contexts(
        corpus, max_window_size)
    all_negatives = get_negatives(
        all_contexts, vocab, counter, num_noise_words)

    class PTBDataset(torch.utils.data.Dataset):
        def __init__(self, centers, contexts, negatives):
            assert len(centers) == len(contexts) == len(negatives)
            self.centers = centers
            self.contexts = contexts
            self.negatives = negatives

        def __getitem__(self, index):
            return (self.centers[index], self.contexts[index],
```

(continues on next page)

(continued from previous page)

```
                    self.negatives[index])

        def __len__(self):
            return len(self.centers)

    dataset = PTBDataset(all_centers, all_contexts, all_negatives)

    data_iter = torch.utils.data.DataLoader(dataset, batch_size, shuffle=True,
                                            collate_fn=batchify,
                                            num_workers=num_workers)
    return data_iter, vocab
```

Hãy để chúng tôi in minibatch đầu tiên của bộ lặp dữ liệu.

```
data_iter, vocab = load_data_ptb(512, 5, 5)
for batch in data_iter:
    for name, data in zip(names, batch):
        print(name, 'shape:', data.shape)
    break
```

```
centers shape: torch.Size([512, 1])
contexts_negatives shape: torch.Size([512, 60])
masks shape: torch.Size([512, 60])
labels shape: torch.Size([512, 60])
```

15.3.7 Tóm tắt

- Các từ tần số cao có thể không hữu ích trong đào tạo. Chúng tôi có thể subsample chúng để tăng tốc trong đào tạo.
- Đối với hiệu quả tính toán, chúng tôi tải các ví dụ trong minibatches. Chúng ta có thể xác định các biến khác để phân biệt các miếng đệm từ các miếng đệm không và các ví dụ tích cực với các biến tiêu cực.

15.3.8 Bài tập

1. Làm thế nào để thời gian chạy của mã trong phần này thay đổi nếu không sử dụng subsampling?
2. Các `RandomGenerator` lớp lưu trữ `k` kết quả lấy mẫu ngẫu nhiên. Đặt `k` thành các giá trị khác và xem nó ảnh hưởng đến tốc độ tải dữ liệu như thế nào.
3. Những siêu tham số khác trong mã của phần này có thể ảnh hưởng đến tốc độ tải dữ liệu?

Discussions[190]

[190] https://discuss.d2l.ai/t/1330

15.4 Pretraining word2vec

Chúng tôi tiếp tục thực hiện mô hình skip-gram được xác định trong Section 15.1. Sau đó, chúng ta sẽ pretrain word2vec bằng cách sử dụng lấy mẫu âm trên tập dữ liệu PTB. Trước hết, chúng ta hãy lấy bộ lặp dữ liệu và từ vựng cho tập dữ liệu này bằng cách gọi hàm `d2l.load_data_ptb`, được mô tả trong Section 15.3

```
import math
import torch
from torch import nn
from d2l import torch as d2l

batch_size, max_window_size, num_noise_words = 512, 5, 5
data_iter, vocab = d2l.load_data_ptb(batch_size, max_window_size,
                                     num_noise_words)
```

15.4.1 Mô hình Skip-Gram

Chúng tôi thực hiện mô hình skip-gram bằng cách sử dụng các lớp nhúng và phép nhân ma trận hàng loạt. Đầu tiên, chúng ta hãy xem lại cách nhúng lớp hoạt động.

Lớp nhúng

Như được mô tả trong Section 10.7, một lớp nhúng ánh xạ chỉ mục của mã thông báo vào vector tính năng của nó. Trọng lượng của lớp này là một ma trận có số hàng bằng với kích thước từ điển (`input_dim`) và số cột bằng với chiều vectơ cho mỗi mã thông báo (`output_dim`). Sau khi một mô hình nhúng từ được đào tạo, trọng lượng này là những gì chúng ta cần.

```
embed = nn.Embedding(num_embeddings=20, embedding_dim=4)
print(f'Parameter embedding_weight ({embed.weight.shape}, '
      f'dtype={embed.weight.dtype})')
```

```
Parameter embedding_weight (torch.Size([20, 4]), dtype=torch.float32)
```

Đầu vào của một lớp nhúng là chỉ mục của một mã thông báo (word). Đối với bất kỳ chỉ số token i, biểu diễn vectơ của nó có thể được lấy từ hàng i^{th} của ma trận trọng lượng trong lớp nhúng. Vì kích thước vectơ (`output_dim`) được đặt thành 4, lớp nhúng trả về vectơ có hình dạng (2, 3, 4) cho một minibatch các chỉ số token có hình dạng (2, 3).

```
x = torch.tensor([[1, 2, 3], [4, 5, 6]])
embed(x)
```

```
tensor([[[ 0.2342, -0.6645, -0.9946,  0.6809],
         [-0.6361,  0.8065, -0.8825,  0.9175],
         [ 0.7129,  0.6453,  0.1846,  0.5313]],

        [[-1.0855,  0.9735, -0.0118, -0.5253],
         [ 0.8133,  0.0536,  1.0759, -0.4699],
         [ 0.7664, -1.3129, -1.5751,  0.0097]]], grad_fn=<EmbeddingBackward0>)
```

Xác định tuyên truyền chuyển tiếp

Trong tuyên truyền chuyển tiếp, đầu vào của mô hình bỏ qua gram bao gồm các chỉ số từ trung tâm `center` của hình dạng (kích thước lô, 1) và ngữ cảnh nối và các chỉ số từ tiếng ồn `contexts_and_negatives` của hình dạng (kích thước lô, `max_len`), trong đó `max_len` được xác định trong Section 15.3.5. Hai biến này lần đầu tiên được chuyển đổi từ các chỉ số token thành vectơ thông qua lớp nhúng, sau đó phép nhân ma trận lô của chúng (được mô tả trong Section 11.2.4) trả về một đầu ra của hình dạng (kích thước lô, 1, `max_len`). Mỗi phần tử trong đầu ra là tích chấm của một vector từ trung tâm và một vector ngữ cảnh hoặc từ nhiễu.

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = torch.bmm(v, u.permute(0, 2, 1))
    return pred
```

Chúng ta hãy in hình dạng đầu ra của chức năng `skip_gram` này cho một số đầu vào ví dụ.

```
skip_gram(torch.ones((2, 1), dtype=torch.long),
          torch.ones((2, 4), dtype=torch.long), embed, embed).shape
```

```
torch.Size([2, 1, 4])
```

15.4.2 Đào tạo

Trước khi đào tạo mô hình skip-gram với lấy mẫu âm, trước tiên chúng ta hãy xác định chức năng mất mát của nó.

Nhị phân Cross-Entropy Mất

Theo định nghĩa của hàm mất để lấy mẫu âm trong Section 15.2.1, chúng ta sẽ sử dụng mất nhị phân cross-entropy.

```
class SigmoidBCELoss(nn.Module):
    # Binary cross-entropy loss with masking
    def __init__(self):
        super().__init__()

    def forward(self, inputs, target, mask=None):
        out = nn.functional.binary_cross_entropy_with_logits(
            inputs, target, weight=mask, reduction="none")
        return out.mean(dim=1)

loss = SigmoidBCELoss()
```

Nhớ lại mô tả của chúng tôi về biến mặt nạ và biến nhãn trong Section 15.3.5. Sau đây tính toán mất nhị phân cross-entropy cho các biến đã cho.

```
pred = torch.tensor([[1.1, -2.2, 3.3, -4.4]] * 2)
label = torch.tensor([[1.0, 0.0, 0.0, 0.0], [0.0, 1.0, 0.0, 0.0]])
```

(continues on next page)

(continued from previous page)

```
mask = torch.tensor([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
```

```
tensor([0.9352, 1.8462])
```

Dưới đây cho thấy cách tính toán các kết quả trên (theo cách kém hiệu quả hơn) bằng cách sử dụng chức năng kích hoạt sigmoid trong mất nhị phân cross-entropy. Chúng ta có thể xem xét hai đầu ra là hai tổn thất bình thường được tính trung bình so với các dự đoán không đeo mặt nạ.

```
def sigmd(x):
    return -math.log(1 / (1 + math.exp(-x)))

print(f'{(sigmd(1.1) + sigmd(2.2) + sigmd(-3.3) + sigmd(4.4)) / 4:.4f}')
print(f'{(sigmd(-1.1) + sigmd(-2.2)) / 2:.4f}')
```

```
0.9352
1.8462
```

Khởi tạo các tham số mô hình

Chúng tôi xác định hai lớp nhúng cho tất cả các từ trong từ vựng khi chúng được sử dụng làm từ trung tâm và từ ngữ cảnh, tương ứng. Kích thước vector từ `embed_size` được đặt thành 100.

```
embed_size = 100
net = nn.Sequential(nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size),
                    nn.Embedding(num_embeddings=len(vocab),
                                 embedding_dim=embed_size))
```

Xác định vòng đào tạo

Vòng đào tạo được định nghĩa dưới đây. Do sự tồn tại của đệm, việc tính toán chức năng mất mát hơi khác so với các chức năng đào tạo trước đó.

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    def init_weights(m):
        if type(m) == nn.Embedding:
            nn.init.xavier_uniform_(m.weight)
    net.apply(init_weights)
    net = net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs])
    # Sum of normalized losses, no. of normalized losses
    metric = d2l.Accumulator(2)
    for epoch in range(num_epochs):
        timer, num_batches = d2l.Timer(), len(data_iter)
        for i, batch in enumerate(data_iter):
```

(continues on next page)

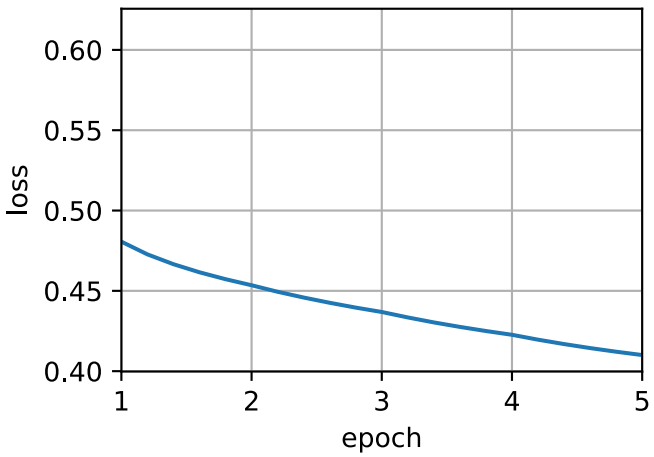
(continued from previous page)

```
            optimizer.zero_grad()
            center, context_negative, mask, label = [
                data.to(device) for data in batch]

            pred = skip_gram(center, context_negative, net[0], net[1])
            l = (loss(pred.reshape(label.shape).float(), label.float(), mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.sum().backward()
            optimizer.step()
            metric.add(l.sum(), l.numel())
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (metric[0] / metric[1],))
    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

Bây giờ chúng ta có thể đào tạo một mô hình skip-gram bằng cách sử dụng lấy mẫu âm.

```
lr, num_epochs = 0.002, 5
train(net, data_iter, lr, num_epochs)
```

```
loss 0.410, 303154.0 tokens/sec on cuda:0
```



15.4.3 Áp dụng Word Embeddings

Sau khi đào tạo mô hình word2vec, chúng ta có thể sử dụng sự tương đồng cosin của các vectơ từ từ từ mô hình được đào tạo để tìm các từ từ điển tương tự về mặt ngữ nghĩa nhất với từ đầu vào.

```
def get_similar_tokens(query_token, k, embed):
    W = embed.weight.data
    x = W[vocab[query_token]]
    # Compute the cosine similarity. Add 1e-9 for numerical stability
    cos = torch.mv(W, x) / torch.sqrt(torch.sum(W * W, dim=1) *
                                      torch.sum(x * x) + 1e-9)
    topk = torch.topk(cos, k=k+1)[1].cpu().numpy().astype('int32')
```

(continues on next page)

(continued from previous page)

```
    for i in topk[1:]:  # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.to_tokens(i)}')

get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.676: microprocessor
cosine sim=0.652: intel
cosine sim=0.617: computer
```

15.4.4 Tóm tắt

- Chúng ta có thể đào tạo một mô hình skip-gram với lấy mẫu âm bằng cách sử dụng các lớp nhúng và mất nhị phân cross-entropy.
- Các ứng dụng của nhúng từ bao gồm việc tìm các từ tương tự về mặt ngữ nghĩa cho một từ nhất định dựa trên sự tương đồng cosin của vectơ từ.

15.4.5 Bài tập

1. Sử dụng mô hình được đào tạo, tìm các từ tương tự về mặt ngữ nghĩa cho các từ đầu vào khác. Bạn có thể cải thiện kết quả bằng cách điều chỉnh các siêu tham số?
2. Khi một cơ thể đào tạo là rất lớn, chúng ta thường lấy mẫu từ ngữ cảnh và các từ tiếng ồn cho các từ trung tâm trong minibatch hiện tại * khi cập nhật tham số mô hình*. Nói cách khác, cùng một từ trung tâm có thể có các từ ngữ cảnh khác nhau hoặc từ tiếng ồn trong các kỷ nguyên đào tạo khác nhau. Lợi ích của phương pháp này là gì? Cố gắng thực hiện phương pháp đào tạo này.

Discussions[191]

15.5 Word Nhúng với Vectơ toàn cầu (Glove)

Các trường hợp đồng từ trong các cửa sổ ngữ cảnh có thể mang thông tin ngữ nghĩa phong phú. Ví dụ, trong một từ corpus lớn "rắn" có nhiều khả năng đồng xảy ra với "băng" hơn là "hơi nước", nhưng từ "khí" có lẽ đồng xảy ra với "hơi nước" thường xuyên hơn "băng". Bên cạnh đó, số liệu thống kê cơ thể toàn cầu của các lần đồng xuất hiện như vậy có thể được tính toán trước: điều này có thể dẫn đến đào tạo hiệu quả hơn. Để tận dụng thông tin thống kê trong toàn bộ corpus để nhúng từ, trước tiên chúng ta hãy xem lại mô hình skip-gram trong Section 15.1.3, nhưng giải thích nó bằng cách sử dụng số liệu thống kê cơ thể toàn cầu như số lượng đồng xuất hiện.

[191] https://discuss.d2l.ai/t/1335

15.5.1 Skip-Gram với Thống kê Corpus toàn cầu

Biểu thị bởi q_{ij} xác suất có điều kiện $P(w_j \mid w_i)$ của từ w_j cho từ w_i trong mô hình skip-gram, chúng tôi có

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \tag{15.5.1}$$

trong đó cho bất kỳ chỉ số i vectơ $\mathbf{v}_i$ và $\mathbf{u}_i$ đại diện cho từ w_i là từ trung tâm và từ ngữ cảnh, tương ứng, và $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ là tập hợp chỉ mục của từ vựng.

Xem xét từ w_i có thể xảy ra nhiều lần trong corpus. Trong toàn bộ cơ thể, tất cả các từ ngữ cảnh bất cứ nơi nào w_i được lấy làm từ trung tâm của chúng tạo thành một * multiset* $\mathcal{C}_i$ các chỉ số từ * cho phép nhiều trường hợp của cùng một phần tố*. Đối với bất kỳ phần tử nào, số phiên bản của nó được gọi là *multiplicity* của nó. Để minh họa với một ví dụ, giả sử rằng từ w_i xảy ra hai lần trong corpus và chỉ số của các từ ngữ cảnh lấy w_i làm từ trung tâm của chúng trong hai cửa sổ ngữ cảnh là k, j, m, k và k, l, k, j. Do đó, multiset $\mathcal{C}_i = \{j, j, k, k, k, k, l, m\}$, trong đó nhân của các yếu tố j, k, l, m là 2, 4, 1, 1, tương ứng.

Bây giờ chúng ta hãy biểu thị sự đa dạng của yếu tố j trong multiset $\mathcal{C}_i$ như x_{ij}. Đây là số lượng đồng xuất hiện toàn cầu của từ w_j (như từ ngữ cảnh) và từ w_i (là từ trung tâm) trong cùng một cửa sổ ngữ cảnh trong toàn bộ corpus. Sử dụng số liệu thống kê cơ thể toàn cầu như vậy, chức năng mất mát của mô hình skip-gram tương đương với

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log\, q_{ij}. \tag{15.5.2}$$

Chúng tôi tiếp tục biểu thị bằng x_i số lượng của tất cả các từ ngữ cảnh trong các cửa sổ ngữ cảnh nơi w_i xảy ra như là từ trung tâm của chúng, tương đương với $|\mathcal{C}_i|$. Cho phép p_{ij} là xác suất có điều kiện x_{ij}/x_i để tạo từ ngữ cảnh w_j cho từ trung tâm w_i, (15.5.2) có thể được viết lại như

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log\, q_{ij}. \tag{15.5.3}$$

Năm (15.5.3), $-\sum_{j \in \mathcal{V}} p_{ij} \log\, q_{ij}$ tính toán ngẫu nhiên chéo của phân phối có điều kiện p_{ij} số liệu thống kê thể toàn cầu và phân phối có điều kiện q_{ij} dự đoán mô hình. Sự mất mát này cũng được trọng số bởi x_i như đã giải thích ở trên. Giảm thiểu hàm mất trong (15.5.3) sẽ cho phép phân phối có điều kiện dự đoán đến gần với phân phối có điều kiện từ số liệu thống kê corpus toàn cầu.

Mặc dù thường được sử dụng để đo khoảng cách giữa các phân phối xác suất, chức năng mất ngẫu nhiên chéo có thể không phải là một lựa chọn tốt ở đây. Một mặt, như chúng tôi đã đề cập trong Section 15.2, chi phí bình thường hóa đúng q_{ij} dẫn đến tổng trên toàn bộ từ vựng, có thể tốn kém về mặt tính toán. Mặt khác, một số lượng lớn các sự kiện hiếm hoi từ một thể lớn thường được mô hình hóa bởi sự mất mát chéo entropy được chỉ định với quá nhiều trọng lượng.

15.5.2 Mô hình Glove

Theo quan điểm này, mô hình *GloVe* thực hiện ba thay đổi đối với mô hình skip-gram dựa trên tổn thất bình phương (Pennington et al., 2014):

1. Sử dụng các biến $p'_{ij} = x_{ij}$ và $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ đó không phải là phân phối xác suất và lấy logarit của cả hai, vì vậy thuật ngữ tổn thất bình phương là $\left(\log\, p'_{ij} - \log\, q'_{ij}\right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log\, x_{ij}\right)^2$.
2. Thêm hai tham số mô hình vô hướng cho mỗi từ w_i: thiên vị từ trung tâm b_i và thiên vị từ ngữ cảnh c_i.

3. Thay thế trọng lượng của mỗi thời hạn giảm với chức năng trọng lượng $h(x_{ij})$, trong đó $h(x)$ đang tăng trong khoảng thời gian $[0, 1]$.

Kết hợp tất cả mọi thứ lại với nhau, đào tạo Glove là để giảm thiểu các chức năng mất mát sau:

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log\, x_{ij} \right)^2 . \tag{15.5.4}$$

Đối với chức năng trọng lượng, một lựa chọn được đề xuất là: $h(x) = (x/c)^\alpha$ (ví dụ $\alpha = 0.75$) nếu $x < c$ (ví dụ, $c = 100$); nếu không thì $h(x) = 1$. Trong trường hợp này, vì $h(0) = 0$, thời hạn tổn thất bình phương cho bất kỳ $x_{ij} = 0$ nào có thể được bỏ qua cho hiệu quả tính toán. Ví dụ: khi sử dụng minibatch stochastic gradient descent để đào tạo, tại mỗi lần lặp lại, chúng tôi lấy mẫu ngẫu nhiên một minibatch của * non-zero* x_{ij} để tính toán gradient và cập nhật các tham số mô hình. Lưu ý rằng những x_{ij} không phải bằng không này là số liệu thống kê cơ thể toàn cầu được tính toán trước; do đó, mô hình được gọi là Glove cho *Global Vectors*.

Cần nhấn mạnh rằng nếu từ w_i xuất hiện trong cửa sổ ngữ cảnh của từ w_j, thì *vice versa*. Do đó, $x_{ij} = x_{ji}$. Không giống như word2vec phù hợp với xác suất điều kiện bất đối xứng p_{ij}, Glove phù hợp với đối xứng $\log\, x_{ij}$. Do đó, vector từ trung tâm và vector từ ngữ cảnh của bất kỳ từ nào tương đương về mặt toán học trong mô hình Glove. Tuy nhiên trong thực tế, do các giá trị khởi tạo khác nhau, cùng một từ vẫn có thể nhận được các giá trị khác nhau trong hai vectơ này sau khi đào tạo: Glove tổng chúng làm vectơ đầu ra.

15.5.3 Giải thích găng tay từ Tỷ lệ xác suất đồng xuất hiện

Chúng ta cũng có thể diễn giải mô hình Glove từ một góc độ khác. Sử dụng ký hiệu tương tự trong Section 15.5.1, hãy để $p_{ij} \stackrel{\text{def}}{=} P(w_j \mid w_i)$ là xác suất có điều kiện tạo ra từ ngữ cảnh w_j cho w_i làm từ trung tâm trong corpus. Section 15.5.3 liệt kê một số xác suất đồng xuất hiện cho từ "băng" và "hơi nước" và tỷ lệ của chúng dựa trên thống kê từ một thể lớn.

w_k=	solid	gas	water	fashion
$p_1 = P(w_k \mid \text{ice})$	0.00019	0.000066	0.003	0.000017
$p_2 = P(w_k \mid \text{steam})$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

Table: Word-word co-occurrence probabilities and their ratios from a large corpus (adapted from Table 1 in (Pennington et al., 2014):)

Chúng ta có thể quan sát những điều sau đây từ Section 15.5.3:

- Đối với một từ w_k có liên quan đến "băng" nhưng không liên quan đến "hơi nước", chẳng hạn như $w_k =$ solid, chúng tôi mong đợi tỷ lệ xác suất đồng xảy ra lớn hơn, chẳng hạn như 8.9.
- Đối với một từ w_k có liên quan đến "hơi nước" nhưng không liên quan đến "băng", chẳng hạn như $w_k =$ gas, chúng tôi mong đợi tỷ lệ xác suất đồng xảy ra nhỏ hơn, chẳng hạn như 0,085.
- Đối với một từ w_k có liên quan đến cả "băng" và "hơi nước", chẳng hạn như $w_k =$ water, chúng tôi mong đợi một tỷ lệ xác suất đồng chiếm gần 1, chẳng hạn như 1.36.
- Đối với một từ w_k không liên quan đến cả "băng" và "hơi nước", chẳng hạn như $w_k =$ fashion, chúng tôi mong đợi một tỷ lệ xác suất đồng chiếm gần 1, chẳng hạn như 0,96.

Có thể thấy rằng tỷ lệ xác suất đồng xuất hiện có thể thể hiện trực giác mối quan hệ giữa các từ. Do đó, chúng ta có thể thiết kế một hàm của ba vectơ từ để phù hợp với tỷ lệ này. Đối với tỷ lệ xác suất đồng xuất hiện

p_{ij}/p_{ik} với w_i là từ trung tâm và w_j và w_k là các từ ngữ cảnh, chúng tôi muốn phù hợp với tỷ lệ này bằng cách sử dụng một số hàm f:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \tag{15.5.5}$$

Trong số nhiều thiết kế có thể có cho f, chúng tôi chỉ chọn một sự lựa chọn hợp lý trong những điều sau đây. Vì tỷ lệ xác suất đồng xuất hiện là vô hướng, chúng tôi yêu cầu f là một hàm vô hướng, chẳng hạn như $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f\left((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i\right)$. Chuyển đổi các chỉ số từ j và k trong (15.5.5), nó phải giữ đó $f(x)f(-x) = 1$, vì vậy một khả năng là $f(x) = \exp(x)$, tức là,

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right)}{\exp\left(\mathbf{u}_k^\top \mathbf{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}. \tag{15.5.6}$$

Bây giờ chúng ta hãy chọn $\exp\left(\mathbf{u}_j^\top \mathbf{v}_i\right) \approx \alpha p_{ij}$, trong đó α là một hằng số. Kể từ $p_{ij} = x_{ij}/x_i$, sau khi lấy logarit ở cả hai bên, chúng tôi nhận được $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$. Chúng tôi có thể sử dụng các thuật ngữ thiên vị bổ sung để phù hợp với $-\log \alpha + \log x_i$, chẳng hạn như thiên vị từ trung tâm b_i và thiên vị từ ngữ cảnh c_j:

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log x_{ij}. \tag{15.5.7}$$

Đo sai số bình phương (15.5.7) với trọng lượng, chức năng mất Glove trong (15.5.4) thu được.

15.5.4 Tóm tắt

- Mô hình skip-gram có thể được giải thích bằng cách sử dụng thống kê corpus toàn cầu như số lượng đồng xuất hiện từ.
- Sự mất mát chéo entropy có thể không phải là một lựa chọn tốt để đo sự khác biệt của hai phân phối xác suất, đặc biệt là đối với một thể lớn. Găng tay sử dụng tổn thất bình phương để phù hợp với số liệu thống kê cơ thể toàn cầu được tính toán trước.
- Vector từ trung tâm và vector từ ngữ cảnh tương đương về mặt toán học cho bất kỳ từ nào trong Glove.
- Găng tay có thể được giải thích từ tỷ lệ xác suất đồng xuất hiện từ.

15.5.5 Bài tập

1. Nếu các từ w_i và w_j đồng xảy ra trong cùng một cửa sổ ngữ cảnh, làm thế nào chúng ta có thể sử dụng khoảng cách của chúng trong chuỗi văn bản để thiết kế lại phương pháp để tính xác suất có điều kiện p_{ij}? Hint: see Section 4.2 of the GloVe paper (Pennington et al., 2014).
2. Đối với bất kỳ từ nào, thiên vị từ trung tâm của nó và thiên vị từ ngữ cảnh tương đương về mặt toán học trong Glove không? Tại sao?

Discussions[192]

[192] https://discuss.d2l.ai/t/385

15.6 Từ tương tự và tương tự

Trong Section 15.4, chúng tôi đã đào tạo một mô hình word2vec trên một tập dữ liệu nhỏ và áp dụng nó để tìm các từ tương tự về mặt ngữ nghĩa cho một từ đầu vào. Trong thực tế, các vectơ từ được đào tạo trước trên thể lớn có thể được áp dụng cho các nhiệm vụ xử lý ngôn ngữ tự nhiên hạ nguồn, sẽ được đề cập sau này vào năm Chapter 16. Để chứng minh ngữ nghĩa của vectơ từ được đào tạo trước từ thể lớn một cách đơn giản, chúng ta hãy áp dụng chúng trong các nhiệm vụ tương tự và tương tự từ.

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

15.6.1 Đang tải Pretrained Word Vectơ

Dưới đây liệt kê các bản nhúng Glove được đào tạo trước có kích thước 50, 100 và 300, có thể tải xuống từ GloVe website[193]. Các bản nhúng fastText được đào tạo trước có sẵn bằng nhiều ngôn ngữ. Ở đây chúng tôi xem xét một phiên bản tiếng Anh (300-dimensional "wiki.en") có thể được tải xuống từ fastText website[194].

```
#@save
d2l.DATA_HUB['glove.6b.50d'] = (d2l.DATA_URL + 'glove.6B.50d.zip',
                                '0b8703943ccdb6eb788e6f091b8946e82231bc4d')

#@save
d2l.DATA_HUB['glove.6b.100d'] = (d2l.DATA_URL + 'glove.6B.100d.zip',
                                 'cd43bfb07e44e6f27cbcc7bc9ae3d80284fdaf5a')

#@save
d2l.DATA_HUB['glove.42b.300d'] = (d2l.DATA_URL + 'glove.42B.300d.zip',
                                  'b5116e234e9eb9076672cfeabf5469f3eec904fa')

#@save
d2l.DATA_HUB['wiki.en'] = (d2l.DATA_URL + 'wiki.en.zip',
                           'c1816da3821ae9f43899be655002f6c723e91b88')
```

Để tải các Glove được đào tạo trước này và các embeddings fastText, chúng tôi xác định lớp `TokenEmbedding` sau.

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
        self.unknown_idx = 0
        self.token_to_idx = {token: idx for idx, token in
                             enumerate(self.idx_to_token)}

    def _load_embedding(self, embedding_name):
```

(continues on next page)

[193] https://nlp.stanford.edu/projects/glove/
[194] https://fasttext.cc/

(continued from previous page)

```
        idx_to_token, idx_to_vec = ['<unk>'], []
        data_dir = d2l.download_extract(embedding_name)
        # GloVe website: https://nlp.stanford.edu/projects/glove/
        # fastText website: https://fasttext.cc/
        with open(os.path.join(data_dir, 'vec.txt'), 'r') as f:
            for line in f:
                elems = line.rstrip().split(' ')
                token, elems = elems[0], [float(elem) for elem in elems[1:]]
                # Skip header information, such as the top row in fastText
                if len(elems) > 1:
                    idx_to_token.append(token)
                    idx_to_vec.append(elems)
        idx_to_vec = [[0] * len(idx_to_vec[0])] + idx_to_vec
        return idx_to_token, torch.tensor(idx_to_vec)

    def __getitem__(self, tokens):
        indices = [self.token_to_idx.get(token, self.unknown_idx)
                   for token in tokens]
        vecs = self.idx_to_vec[torch.tensor(indices)]
        return vecs

    def __len__(self):
        return len(self.idx_to_token)
```

Dưới đây chúng tôi tải các nhúng Glove 50 chiều (được đào tạo trước trên một tập con Wikipedia). Khi tạo phiên bản `TokenEmbedding`, tệp nhúng được chỉ định phải được tải xuống nếu chưa.

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

```
Downloading ../data/glove.6B.50d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.50d.zip...
```

Xuất kích thước từ vựng. Từ vựng chứa 400000 từ (mã thông báo) và một mã thông báo không xác định đặc biệt.

```
len(glove_6b50d)
```

```
400001
```

Chúng ta có thể lấy chỉ mục của một từ trong từ vựng, và ngược lại.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

```
(3367, 'beautiful')
```

15.6.2 Áp dụng vectơ Word Pretrained

Sử dụng các vectơ Glove được tải, chúng tôi sẽ chứng minh ngữ nghĩa của chúng bằng cách áp dụng chúng trong các tác vụ tương tự và tương tự từ sau đây.

Từ tương tự

Tương tự như Section 15.4.3, để tìm các từ tương tự về mặt ngữ nghĩa cho một từ đầu vào dựa trên sự tương đồng cosin giữa các vectơ từ, chúng tôi thực hiện hàm `knn` (k-láng giềng gần nhất) sau đây.

```
def knn(W, x, k):
    # Add 1e-9 for numerical stability
    cos = torch.mv(W, x.reshape(-1,)) / (
        torch.sqrt(torch.sum(W * W, axis=1) + 1e-9) *
        torch.sqrt((x * x).sum()))
    _, topk = torch.topk(cos, k=k)
    return topk, [cos[int(i)] for i in topk]
```

Sau đó, chúng ta tìm kiếm các từ tương tự bằng cách sử dụng vectơ từ được đào tạo trước từ trường hợp `TokenEmbedding` `embed`.

```
def get_similar_tokens(query_token, k, embed):
    topk, cos = knn(embed.idx_to_vec, embed[[query_token]], k + 1)
    for i, c in zip(topk[1:], cos[1:]):  # Exclude the input word
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}')
```

Từ vựng của các vectơ từ được đào tạo trước trong `glove_6b50d` chứa 400000 từ và một mã thông báo không xác định đặc biệt. Không bao gồm từ đầu vào và mã thông báo không xác định, trong số từ vựng này cho phép chúng ta tìm thấy ba từ tương tự về mặt ngữ nghĩa nhất với từ "chip".

```
get_similar_tokens('chip', 3, glove_6b50d)
```

```
cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

Dưới đây xuất ra các từ tương tự như "em bé" và "đẹp".

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

```
cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Từ tương tự

Bên cạnh việc tìm các từ tương tự, chúng ta cũng có thể áp dụng vectơ từ cho các tác vụ tương tự từ. Ví dụ, "man": "woman":: "con trai": "con gái" là hình thức của một từ tương tự: "người đàn ông" là "người phụ nữ" là "con trai" là "con gái". Cụ thể, nhiệm vụ hoàn thành từ tương tự có thể được định nghĩa là: đối với một từ tương tự $a : b :: c : d$, cho ba từ đầu tiên a, b và c, tìm d. Biểu thị vector của từ w bởi $\text{vec}(w)$. Để hoàn thành sự tương tự, chúng ta sẽ tìm thấy từ có vector tương tự nhất với kết quả của $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$.

```
def get_analogy(token_a, token_b, token_c, embed):
    vecs = embed[[token_a, token_b, token_c]]
    x = vecs[1] - vecs[0] + vecs[2]
    topk, cos = knn(embed.idx_to_vec, x, 1)
    return embed.idx_to_token[int(topk[0])]  # Remove unknown words
```

Hãy để chúng tôi xác minh sự tương tự "nam-nữ" bằng cách sử dụng các vectơ từ được tải.

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

```
'daughter'
```

Dưới đây hoàn thành một cách tương tự "thủ đô-quốc gia": "beijing": "china":: "tokyo": "japan". Điều này thể hiện ngữ nghĩa trong vectơ từ được đào tạo trước.

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

```
'japan'
```

Đối với tính từ tương tự "tính từ siêu" như "bad": "worst":: "big": "big": "biggest", chúng ta có thể thấy rằng các vectơ từ được đào tạo trước có thể nắm bắt thông tin cú pháp.

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

```
'biggest'
```

Để thể hiện khái niệm bị bắt về thì quá khứ trong vectơ từ được đào tạo trước, chúng ta có thể kiểm tra cú pháp bằng cách sử dụng tương tự "thì quá khứ hiện tại": "do": "did":: "go": "went".

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

```
'went'
```

15.6.3 Tóm tắt

- Trong thực tế, các vectơ từ được đào tạo trước trên thể lớn có thể được áp dụng cho các nhiệm vụ xử lý ngôn ngữ tự nhiên hạ nguồn.
- Vectơ từ được đào tạo trước có thể được áp dụng cho các tác vụ tương tự và tương tự từ.

15.6.4 Bài tập

1. Kiểm tra kết quả fastText bằng cách sử dụng `TokenEmbedding('wiki.en')`.
2. Khi từ vựng cực kỳ lớn, làm thế nào chúng ta có thể tìm thấy các từ tương tự hoặc hoàn thành một từ tương tự nhanh hơn?

Discussions[195]

15.7 Đại diện bộ mã hóa hai chiều từ Transformers (BERT)

Chúng tôi đã giới thiệu một số mô hình nhúng từ để hiểu ngôn ngữ tự nhiên. Sau khi đào tạo trước, đầu ra có thể được coi là một ma trận trong đó mỗi hàng là một vectơ đại diện cho một từ của một từ vựng được xác định trước. Trên thực tế, các mô hình nhúng từ này đều là * bối cảnh độc lập*. Hãy để chúng tôi bắt đầu bằng cách minh họa tài sản này.

15.7.1 Từ bối cảnh độc lập đến bối cảnh nhạy cảm

Nhớ lại các thí nghiệm trong Section 15.4 và Section 15.6. Ví dụ, word2vec và Glove đều gán cùng một vector được đào tạo trước cho cùng một từ bất kể ngữ cảnh của từ (nếu có). Chính thức, một biểu diễn theo ngữ cảnh độc lập của bất kỳ mã thông báo x là một hàm $f(x)$ chỉ mất x làm đầu vào của nó. Với sự phong phú của ngữ nghĩa polysemy và phức tạp trong các ngôn ngữ tự nhiên, các đại diện độc lập ngữ cảnh có những hạn chế rõ ràng. Ví dụ, từ "crane" trong ngữ cảnh "một cần cẩu đang bay" và "một trình điều khiển cần cẩu đến" có ý nghĩa hoàn toàn khác nhau; do đó, cùng một từ có thể được gán các biểu diễn khác nhau tùy thuộc vào ngữ cảnh.

Điều này thúc đẩy sự phát triển của biểu diễn từ *context-sensitive*, trong đó biểu diễn của các từ phụ thuộc vào bối cảnh của chúng. Do đó, một biểu diễn nhạy cảm với ngữ cảnh của token x là một hàm $f(x, c(x))$ tùy thuộc vào cả x và bối cảnh của nó $c(x)$. Các biểu diễn nhạy cảm với ngữ cảnh phổ biến bao gồm TagLM (tagger trình tự tăng cường mô hình ngôn ngữ) (Peters et al., 2017b), cove (Context Vectors) (McCann et al., 2017) và ELMo (Embeddings from Language Models) (Peters et al., 2018).

Ví dụ, bằng cách lấy toàn bộ chuỗi làm đầu vào, ELMo là một hàm gán một biểu diễn cho mỗi từ từ chuỗi đầu vào. Cụ thể, ELMo kết hợp tất cả các biểu diễn lớp trung gian từ LSTM hai chiều được đào tạo trước làm đại diện đầu ra. Sau đó, đại diện ELMo sẽ được thêm vào mô hình giám sát hiện có của tác vụ hạ nguồn dưới dạng các tính năng bổ sung, chẳng hạn như bằng cách nối đại diện ELMo và biểu diễn ban đầu (ví dụ: Glove) của các mã thông báo trong mô hình hiện có. Một mặt, tất cả các trọng lượng trong mô hình LSTM hai chiều được đào tạo trước được đóng băng sau khi biểu diễn ELMo được thêm vào. Mặt khác, mô hình giám sát hiện có được tùy chỉnh đặc biệt cho một nhiệm vụ nhất định. Tận dụng các mô hình tốt nhất khác nhau cho các nhiệm vụ khác nhau tại thời điểm đó, thêm ELMo đã cải thiện trạng thái của nghệ thuật trong sáu nhiệm vụ xử lý ngôn ngữ tự nhiên: phân tích tình cảm, suy luận ngôn ngữ tự nhiên, ghi nhãn vai trò ngữ nghĩa, giải quyết coreference, nhận dạng thực thể được đặt tên và trả lời câu hỏi.

[195] https://discuss.d2l.ai/t/1336

15.7.2 Từ nhiệm vụ cụ thể đến nhiệm vụ Agnostic

Mặc dù ELMo đã cải thiện đáng kể các giải pháp cho một tập hợp các nhiệm vụ xử lý ngôn ngữ tự nhiên đa dạng, mỗi giải pháp vẫn dựa trên một kiến trúc *cụ thể tác vụ*. Tuy nhiên, thực tế là không tầm thường để tạo ra một kiến trúc cụ thể cho mọi nhiệm vụ xử lý ngôn ngữ tự nhiên. Mô hình GPT (Generative Pre-Training) đại diện cho một nỗ lực trong việc thiết kế mô hình *nhiệm vụ chung cho các biểu diễn nhạy cảm với ngữ cảnh (Radford et al., 2018). Được xây dựng trên một bộ giải mã biến áp, GPT pretrain một mô hình ngôn ngữ sẽ được sử dụng để đại diện cho chuỗi văn bản. Khi áp dụng GPT cho một tác vụ hạ lưu, đầu ra của mô hình ngôn ngữ sẽ được đưa vào một lớp đầu ra tuyến tính bổ sung để dự đoán nhãn của tác vụ. Ngược lại sắc nét với ELMo đóng băng các thông số của mô hình được đào tạo trước, GPT tinh chỉnh * tất cả* các thông số trong bộ giải mã biến áp được đào tạo trước trong quá trình học tập được giám sát về nhiệm vụ hạ lưu. GPT được đánh giá trên mười hai nhiệm vụ suy luận ngôn ngữ tự nhiên, trả lời câu hỏi, tương tự câu, và phân loại, và cải thiện trạng thái của nghệ thuật trong chín trong số đó với những thay đổi tối thiểu đối với kiến trúc mô hình.

Tuy nhiên, do tính chất tự hồi quy của các mô hình ngôn ngữ, GPT chỉ nhìn về phía trước (từ trái sang phải). Trong bối cảnh "tôi đã đến ngân hàng để gửi tiền mặt" và "tôi đã đi đến ngân hàng để ngồi xuống", vì "ngân hàng" nhạy cảm với bối cảnh bên trái của nó, GPT sẽ trả lại đại diện tương tự cho "ngân hàng", mặc dù nó có ý nghĩa khác nhau.

15.7.3 BERT: Kết hợp tốt nhất của cả hai thế giới

Như chúng ta đã thấy, ELMo mã hóa ngữ cảnh hai chiều nhưng sử dụng các kiến trúc cụ thể tác vụ; trong khi GPT là nhiệm vụ bất khả tri nhưng mã hóa ngữ cảnh từ trái sang phải. Kết hợp tốt nhất của cả hai thế giới, BERT (Bidirectional Encoder Representations from Transformers) mã hóa bối cảnh hai chiều và yêu cầu thay đổi kiến trúc tối thiểu cho một loạt các nhiệm vụ xử lý ngôn ngữ tự nhiên (Devlin et al., 2018). Sử dụng bộ mã hóa biến áp được đào tạo trước, BERT có thể đại diện cho bất kỳ mã thông báo nào dựa trên bối cảnh hai chiều của nó. Trong quá trình học tập giám sát các nhiệm vụ hạ lưu, BERT tương tự như GPT về hai khía cạnh. Đầu tiên, các đại diện BERT sẽ được đưa vào một lớp đầu ra bổ sung, với những thay đổi tối thiểu đối với kiến trúc mô hình tùy thuộc vào tính chất của các nhiệm vụ, chẳng hạn như dự đoán cho mọi token so với dự đoán cho toàn bộ chuỗi. Thứ hai, tất cả các thông số của bộ mã hóa biến áp được đào tạo trước đều được tinh chỉnh, trong khi lớp đầu ra bổ sung sẽ được đào tạo từ đầu. Fig. 15.7.1 mô tả sự khác biệt giữa ELMo, GPT và BERT.

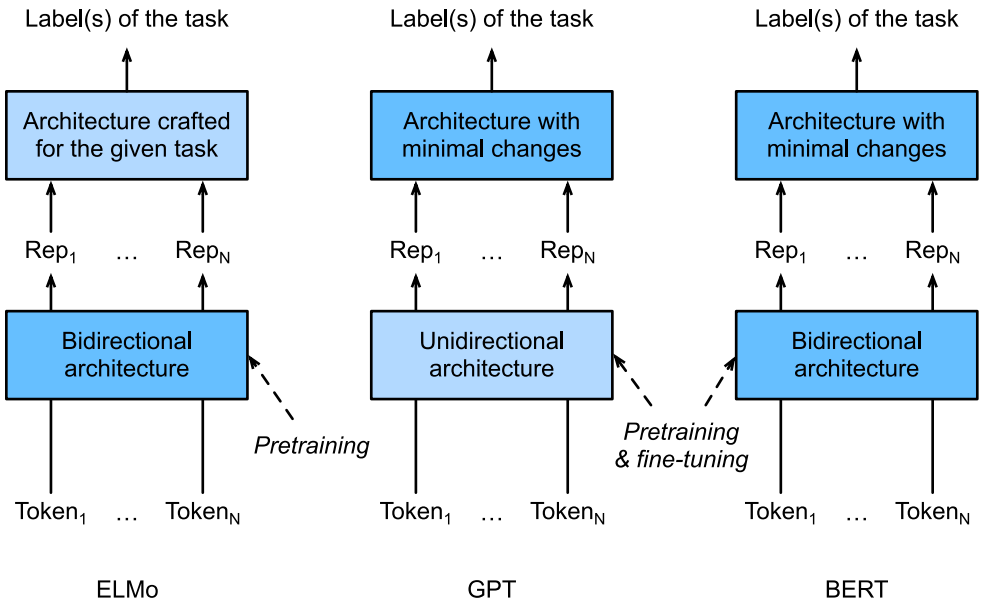


Fig. 15.7.1: A comparison of ELMo, GPT, and BERT.

BERT cải thiện hơn nữa trạng thái của nghệ thuật trên mười một nhiệm vụ xử lý ngôn ngữ tự nhiên theo các loại rộng của (i) phân loại văn bản đơn (ví dụ, phân tích tình cảm), (ii) phân loại cặp văn bản (ví dụ, suy luận ngôn ngữ tự nhiên), (iii) trả lời câu hỏi, (iv) gắn thẻ văn bản (ví dụ, nhận dạng thực thể được đặt tên) . Tất cả được đề xuất vào năm 2018, từ ELMo nhạy cảm với ngữ cảnh đến GPT và BERT bất khả tri nhiệm vụ, đơn giản về mặt khái niệm nhưng mạnh mẽ về mặt thực nghiệm về các biểu diễn sâu sắc cho các ngôn ngữ tự nhiên đã cách mạng hóa các giải pháp cho các nhiệm vụ xử lý ngôn ngữ tự nhiên khác nhau.

Trong phần còn lại của chương này, chúng tôi sẽ đi sâu vào pretraining của BERT. Khi các ứng dụng xử lý ngôn ngữ tự nhiên được giải thích trong Chapter 16, chúng tôi sẽ minh họa tinh chỉnh BERT cho các ứng dụng hạ nguồn.

```
import torch
from torch import nn
from d2l import torch as d2l
```

15.7.4 Đại diện đầu vào

Trong xử lý ngôn ngữ tự nhiên, một số nhiệm vụ (ví dụ, phân tích tình cảm) lấy văn bản đơn làm đầu vào, trong khi trong một số nhiệm vụ khác (ví dụ, suy luận ngôn ngữ tự nhiên), đầu vào là một cặp chuỗi văn bản. Chuỗi đầu vào BERT rõ ràng đại diện cho cả hai cặp văn bản và văn bản đơn lẻ. Trước đây, chuỗi đầu vào BERT là sự nối của mã thông báo phân loại đặc biệt “”, mã thông báo của một chuỗi văn bản, và mã thông báo tách đặc biệt “”. Sau này, chuỗi đầu vào BERT là sự nối của “”, mã thông báo của chuỗi văn bản đầu tiên, “”, mã thông báo của chuỗi văn bản thứ hai, và “”. Chúng tôi sẽ liên tục phân biệt thuật ngữ “chuỗi đầu vào BERT” với các loại “chuỗi” khác. Ví dụ: một trình tự đầu vào *BERT* có thể bao gồm một chuỗi văn bản* hoặc hai chuỗi văn bản*.

Để phân biệt các cặp văn bản, các nhúng phân đoạn đã học được $\mathbf{e}_A$ và $\mathbf{e}_B$ được thêm vào các bản nhúng mã thông báo của chuỗi thứ nhất và chuỗi thứ hai, tương ứng. Đối với các đầu vào văn bản đơn, chỉ có $\mathbf{e}_A$ được sử dụng.

`get_tokens_and_segments` sau lấy một câu hoặc hai câu làm đầu vào, sau đó trả về token của chuỗi đầu vào BERT và ID phân đoạn tương ứng của chúng.

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    """Get tokens of the BERT input sequence and their segment IDs."""
    tokens = ['<cls>'] + tokens_a + ['<sep>']
    # 0 and 1 are marking segment A and B, respectively
    segments = [0] * (len(tokens_a) + 2)
    if tokens_b is not None:
        tokens += tokens_b + ['<sep>']
        segments += [1] * (len(tokens_b) + 1)
    return tokens, segments
```

BERT chọn bộ mã hóa biến áp làm kiến trúc hai chiều của nó. Phổ biến trong bộ mã hóa biến áp, nhúng vị trí được thêm vào ở mọi vị trí của chuỗi đầu vào BERT. Tuy nhiên, khác với bộ mã hóa biến áp ban đầu, BERT sử dụng * learnable* embeddings vị trí. Tóm lại, `fig_bert-input` cho thấy các nhúng của chuỗi đầu vào BERT là tổng của các nhúng mã thông báo, nhúng phân đoạn và nhúng vị trí.

! Các embeddings của chuỗi đầu vào BERT là tổng của các mã thông báo embeddings, segment embeddings, and positional embeddings. .. _fig_bert-input:

Lớp `BERTEncoder` sau tương tự như lớp `TransformerEncoder` như được triển khai vào năm Section 11.7. Khác với `TransformerEncoder`, `BERTEncoder` sử dụng nhúng phân đoạn và nhúng vị trí có thể học được.

```
#@save
class BERTEncoder(nn.Module):
    """BERT encoder."""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 **kwargs):
        super(BERTEncoder, self).__init__(**kwargs)
        self.token_embedding = nn.Embedding(vocab_size, num_hiddens)
        self.segment_embedding = nn.Embedding(2, num_hiddens)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module(f"{i}", d2l.EncoderBlock(
                key_size, query_size, value_size, num_hiddens, norm_shape,
                ffn_num_input, ffn_num_hiddens, num_heads, dropout, True))
        # In BERT, positional embeddings are learnable, thus we create a
        # parameter of positional embeddings that are long enough
        self.pos_embedding = nn.Parameter(torch.randn(1, max_len,
                                                      num_hiddens))

    def forward(self, tokens, segments, valid_lens):
        # Shape of `X` remains unchanged in the following code snippet:
        # (batch size, max sequence length, `num_hiddens`)
        X = self.token_embedding(tokens) + self.segment_embedding(segments)
        X = X + self.pos_embedding.data[:, :X.shape[1], :]
        for blk in self.blks:
            X = blk(X, valid_lens)
        return X
```

Giả sử rằng kích thước từ vựng là 10000. Để chứng minh suy luận chuyển tiếp của `BERTEncoder`, chúng

ta hãy tạo một thể hiện của nó và khởi tạo các tham số của nó.

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
norm_shape, ffn_num_input, num_layers, dropout = [768], 768, 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape, ffn_num_input,
                      ffn_num_hiddens, num_heads, num_layers, dropout)
```

Chúng tôi định nghĩa `tokens` là 2 chuỗi đầu vào BERT có độ dài 8, trong đó mỗi mã thông báo là một chỉ số của từ vựng. Suy luận chuyển tiếp của `BERTEncoder` với đầu vào `tokens` trả về kết quả được mã hóa trong đó mỗi mã thông báo được biểu diễn bởi một vectơ có độ dài được xác định trước bởi siêu tham số `num_hiddens`. Siêu tham số này thường được gọi là * hidden size* (số đơn vị ẩn) của bộ mã hóa biến áp.

```
tokens = torch.randint(0, vocab_size, (2, 8))
segments = torch.tensor([[0, 0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

```
torch.Size([2, 8, 768])
```

15.7.5 Nhiệm vụ Pretraining

Suy luận chuyển tiếp của `BERTEncoder` cho phép đại diện BERT của mỗi mã thông báo của văn bản đầu vào và các mã thông báo đặc biệt được chèn "" và "". Tiếp theo, chúng tôi sẽ sử dụng các đại diện này để tính toán hàm mất cho BERT pretraining. Việc đào tạo trước bao gồm hai nhiệm vụ sau: mô hình hóa ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo.

Mô hình hóa ngôn ngữ đeo mặt nạ

Như minh họa trong Section 9.3, một mô hình ngôn ngữ dự đoán một mã thông báo sử dụng ngữ cảnh bên trái của nó. Để mã hóa ngữ cảnh hai chiều để đại diện cho mỗi mã thông báo, BERT ngẫu nhiên che dấu mã thông báo và sử dụng mã thông báo từ bối cảnh hai chiều để dự đoán các token đeo mặt nạ theo cách tự giám sát. Nhiệm vụ này được gọi là mô hình ngôn ngữ * được đeo mặt*.

Trong nhiệm vụ đào tạo trước này, 15% mã thông báo sẽ được chọn ngẫu nhiên làm mã thông báo đeo mặt nạ để dự đoán. Để dự đoán một token đeo mặt nạ mà không gian lận bằng cách sử dụng nhãn, một cách tiếp cận đơn giản là luôn thay thế nó bằng một mã thông báo "" đặc biệt trong chuỗi đầu vào BERT. Tuy nhiên, mã thông báo đặc biệt nhân tạo "" sẽ không bao giờ xuất hiện trong tinh chỉnh. Để tránh sự không phù hợp giữa đào tạo trước và tinh chỉnh, nếu một mã thông báo được che dấu để dự đoán (ví dụ: "great" được chọn để che dấu và dự đoán trong "bộ phim này là tuyệt vời"), trong đầu vào nó sẽ được thay thế bằng:

- một mã thông báo "" đặc biệt cho 80% thời gian (ví dụ: "bộ phim này thật tuyệt" trở thành "bộ phim này là ");
- một mã thông báo ngẫu nhiên cho 10% thời gian (ví dụ: "bộ phim này là tuyệt vời" trở thành "bộ phim này là đồ uống");
- mã thông báo nhãn không thay đổi trong 10% thời gian (ví dụ: "bộ phim này thật tuyệt" trở thành "bộ phim này thật tuyệt").

Lưu ý rằng trong 10% thời gian 15%, một mã thông báo ngẫu nhiên được chèn vào. Tiếng ồn thỉnh thoảng này khuyến khích BERT ít thiên vị hơn đối với mã thông báo được đeo mặt nạ (đặc biệt là khi mã thông báo nhãn vẫn không thay đổi) trong mã hóa ngữ cảnh hai chiều của nó.

Chúng tôi thực hiện lớp `MaskLM` sau đây để dự đoán các token đeo mặt nạ trong nhiệm vụ mô hình ngôn ngữ đeo mặt nạ của BERT pretraining. Dự đoán sử dụng MLP một lớp ẩn (`self.mlp`). Trong suy luận chuyển tiếp, phải mất hai đầu vào: kết quả được mã hóa của `BERTEncoder` và các vị trí mã thông báo để dự đoán. Đầu ra là kết quả dự đoán tại các vị trí này.

```
#@save
class MaskLM(nn.Module):
    """The masked language model task of BERT."""
    def __init__(self, vocab_size, num_hiddens, num_inputs=768, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential(nn.Linear(num_inputs, num_hiddens),
                                 nn.ReLU(),
                                 nn.LayerNorm(num_hiddens),
                                 nn.Linear(num_hiddens, vocab_size))

    def forward(self, X, pred_positions):
        num_pred_positions = pred_positions.shape[1]
        pred_positions = pred_positions.reshape(-1)
        batch_size = X.shape[0]
        batch_idx = torch.arange(0, batch_size)
        # Suppose that `batch_size` = 2, `num_pred_positions` = 3, then
        # `batch_idx` is `torch.tensor([0, 0, 0, 1, 1, 1])`
        batch_idx = torch.repeat_interleave(batch_idx, num_pred_positions)
        masked_X = X[batch_idx, pred_positions]
        masked_X = masked_X.reshape((batch_size, num_pred_positions, -1))
        mlm_Y_hat = self.mlp(masked_X)
        return mlm_Y_hat
```

Để chứng minh suy luận về phía trước của `MaskLM`, chúng tôi tạo ra phiên bản `mlm` của nó và khởi tạo nó. Nhớ lại rằng `encoded_X` từ suy luận về phía trước của `BERTEncoder` đại diện cho 2 chuỗi đầu vào BERT. Chúng tôi định nghĩa `mlm_positions` là 3 chỉ số để dự đoán trong một trong hai chuỗi đầu vào BERT là `encoded_X`. Suy luận về phía trước của `mlm` trả về kết quả dự đoán `mlm_Y_hat` tại tất cả các vị trí đeo mặt nạ `mlm_positions` của `encoded_X`. Đối với mỗi dự đoán, kích thước của kết quả bằng với kích thước từ vựng.

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm_positions = torch.tensor([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

```
torch.Size([2, 3, 10000])
```

Với nhãn chân lý mặt đất `mlm_Y` của các mã thông báo dự đoán `mlm_Y_hat` dưới mặt nạ, chúng ta có thể tính toán sự mất mát chéo entropy của nhiệm vụ mô hình ngôn ngữ đeo mặt nạ trong pretraining BERT.

```
mlm_Y = torch.tensor([[7, 8, 9], [10, 20, 30]])
loss = nn.CrossEntropyLoss(reduction='none')
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

```
torch.Size([6])
```

Dự đoán câu tiếp theo

Mặc dù mô hình hóa ngôn ngữ đeo mặt nạ có thể mã hóa ngữ cảnh hai chiều để biểu diễn các từ, nó không mô hình hóa rõ ràng mối quan hệ logic giữa các cặp văn bản. Để giúp hiểu mối quan hệ giữa hai chuỗi văn bản, BERT xem xét một nhiệm vụ phân loại nhị phân, * dự đoán câu tiếp theo*, trong pretraining của nó. Khi tạo ra các cặp câu cho pretraining, trong một nửa thời gian chúng thực sự là những câu liên tiếp với nhãn "True"; trong khi trong nửa còn lại của thời gian câu thứ hai được lấy mẫu ngẫu nhiên từ corpus với nhãn "False".

Lớp `NextSentencePred` sau sử dụng MLP một lớp ẩn để dự đoán câu thứ hai có phải là câu tiếp theo của câu thứ nhất trong chuỗi đầu vào BERT hay không. Do sự tự chú ý trong bộ mã hóa biến áp, đại diện BERT của mã thông báo đặc biệt "" mã hóa cả hai câu từ đầu vào. Do đó, lớp đầu ra (`self.output`) của phân loại MLP lấy `X` làm đầu vào, trong đó `X` là đầu ra của lớp ẩn MLP có đầu vào là mã thông báo "" được mã hóa.

```
#@save
class NextSentencePred(nn.Module):
    """The next sentence prediction task of BERT."""
    def __init__(self, num_inputs, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Linear(num_inputs, 2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

Chúng ta có thể thấy rằng suy luận chuyển tiếp của một phiên bản `NextSentencePred` trả về dự đoán nhị phân cho mỗi chuỗi đầu vào BERT.

```
# PyTorch by default won't flatten the tensor as seen in mxnet where, if
# flatten=True, all but the first axis of input data are collapsed together
encoded_X = torch.flatten(encoded_X, start_dim=1)
# input_shape for NSP: (batch size, `num_hiddens`)
nsp = NextSentencePred(encoded_X.shape[-1])
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

```
torch.Size([2, 2])
```

Sự mất mát cross-entropy của 2 phân loại nhị phân cũng có thể được tính toán.

```
nsp_y = torch.tensor([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

```
torch.Size([2])
```

Đáng chú ý là tất cả các nhãn trong cả hai nhiệm vụ đào tạo trước nói trên đều có thể thu được một cách tầm thường từ cơ thể đào tạo trước mà không cần nỗ lực ghi nhãn thủ công. BERT ban đầu đã được đào tạo trước về việc nối BookCorpus (Zhu et al., 2015) và Wikipedia tiếng Anh. Hai tập đoàn văn bản này rất lớn: họ có 800 triệu từ và 2,5 tỷ từ, tương ứng.

15.7.6 Đặt tất cả mọi thứ lại với nhau

Khi đào tạo trước BERT, hàm mất cuối cùng là sự kết hợp tuyến tính của cả hàm mất cho mô hình ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo. Bây giờ chúng ta có thể xác định lớp `BERTModel` bằng cách khởi tạo ba lớp `BERTEncoder`, `MaskLM` và `NextSentencePred`. Suy luận chuyển tiếp trả về các đại diện BERT được mã hóa `encoded_X`, dự đoán về mô hình hóa ngôn ngữ đeo mặt nạ `mlm_Y_hat`, và dự đoán câu tiếp theo `nsp_Y_hat`.

```
#@save
class BERTModel(nn.Module):
    """The BERT model."""
    def __init__(self, vocab_size, num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, num_layers, dropout,
                 max_len=1000, key_size=768, query_size=768, value_size=768,
                 hid_in_features=768, mlm_in_features=768,
                 nsp_in_features=768):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, norm_shape,
                     ffn_num_input, ffn_num_hiddens, num_heads, num_layers,
                     dropout, max_len=max_len, key_size=key_size,
                     query_size=query_size, value_size=value_size)
        self.hidden = nn.Sequential(nn.Linear(hid_in_features, num_hiddens),
                                    nn.Tanh())
        self.mlm = MaskLM(vocab_size, num_hiddens, mlm_in_features)
        self.nsp = NextSentencePred(nsp_in_features)

    def forward(self, tokens, segments, valid_lens=None, pred_positions=None):
        encoded_X = self.encoder(tokens, segments, valid_lens)
        if pred_positions is not None:
            mlm_Y_hat = self.mlm(encoded_X, pred_positions)
        else:
            mlm_Y_hat = None
        # The hidden layer of the MLP classifier for next sentence prediction.
        # 0 is the index of the '<cls>' token
        nsp_Y_hat = self.nsp(self.hidden(encoded_X[:, 0, :]))
        return encoded_X, mlm_Y_hat, nsp_Y_hat
```

15.7.7 Tóm tắt

- Các mô hình nhúng từ như word2vec và Glove độc lập với ngữ cảnh. Họ gán cùng một vector được đào tạo trước cho cùng một từ bất kể ngữ cảnh của từ (nếu có). Thật khó để họ xử lý tốt ngữ nghĩa polysemy hoặc phức tạp trong các ngôn ngữ tự nhiên.
- Đối với các biểu diễn từ nhạy cảm với ngữ cảnh như ELMo và GPT, biểu diễn các từ phụ thuộc vào ngữ cảnh của chúng.
- ELMo mã hóa ngữ cảnh hai chiều nhưng sử dụng kiến trúc cụ thể tác vụ (tuy nhiên, thực tế là không tầm thường để chế tạo một kiến trúc cụ thể cho mọi nhiệm vụ xử lý ngôn ngữ tự nhiên); trong khi GPT là nhiệm vụ bất khả tri nhưng mã hóa ngữ cảnh từ trái sang phải.
- BERT kết hợp tốt nhất của cả hai thế giới: nó mã hóa bối cảnh hai chiều và đòi hỏi những thay đổi kiến trúc tối thiểu cho một loạt các nhiệm vụ xử lý ngôn ngữ tự nhiên.
- Các embeddings của chuỗi đầu vào BERT là tổng của các embeddings token, phân đoạn embeddings, và embeddings vị trí.

- Pretraining BERT bao gồm hai nhiệm vụ: mô hình hóa ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo. Cái trước có thể mã hóa ngữ cảnh hai chiều để đại diện cho các từ, trong khi sau này mô hình rõ ràng mối quan hệ logic giữa các cặp văn bản.

15.7.8 Bài tập

1. Tại sao BERT lại thành công?
2. Tất cả những thứ khác đều bình đẳng, một mô hình ngôn ngữ đeo mặt nạ sẽ yêu cầu nhiều hơn hoặc ít hơn các bước đào tạo trước để hội tụ hơn một mô hình ngôn ngữ từ trái sang phải? Tại sao?
3. Trong việc triển khai ban đầu của BERT, mạng chuyển tiếp nguồn cấp dữ liệu định vị trong `BERTEncoder` (thông qua `d2l.EncoderBlock`) và lớp kết nối hoàn toàn trong `MaskLM` cả hai đều sử dụng đơn vị tuyến tính lỗi Gaussian (GELU) (Hendrycks & Gimpel, 2016) làm chức năng kích hoạt. Nghiên cứu về sự khác biệt giữa GELU và ReLU.

Discussions[196]

15.8 Tập dữ liệu cho Pretraining BERT

Để chuẩn bị mô hình BERT như được triển khai trong Section 15.7, chúng ta cần tạo bộ dữ liệu ở định dạng lý tưởng để tạo điều kiện thuận lợi cho hai nhiệm vụ đào tạo trước: mô hình hóa ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo. Một mặt, mô hình BERT ban đầu được đào tạo sơ bộ về việc nối hai corpora BookCorpus khổng lồ và Wikipedia tiếng Anh (xem Section 15.7.5), khiến hầu hết độc giả của cuốn sách này khó chạy. Mặt khác, mô hình BERT được đào tạo sẵn sẵn có thể không phù hợp với các ứng dụng từ các lĩnh vực cụ thể như y học. Do đó, nó đang trở nên phổ biến để pretrain BERT trên một tập dữ liệu tùy chỉnh. Để tạo điều kiện cho việc trình diễn pretraining BERT, chúng tôi sử dụng một cơ thể nhỏ hơn WikiText-2 (Merity et al., 2016).

So sánh với tập dữ liệu PTB được sử dụng để đào tạo trước word2vec trong Section 15.3, WikiText-2 (i) giữ lại dấu câu ban đầu, làm cho nó phù hợp với dự đoán câu tiếp theo; (ii) giữ lại trường hợp và số gốc; (iii) lớn hơn hai lần.

```
import os
import random
import torch
from d2l import torch as d2l
```

Trong tập dữ liệu WikiText-2, mỗi dòng đại diện cho một đoạn văn mà không gian được chèn giữa bất kỳ dấu chấm câu nào và mã thông báo trước đó. Các đoạn có ít nhất hai câu được giữ lại. Để chia câu, chúng ta chỉ sử dụng dấu chấm làm dấu phân cách để đơn giản. Chúng tôi để lại các cuộc thảo luận về các kỹ thuật tách câu phức tạp hơn trong các bài tập ở cuối phần này.

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
```

(continues on next page)

[196] https://discuss.d2l.ai/t/1490

(continued from previous page)

```
def _read_wiki(data_dir):
    file_name = os.path.join(data_dir, 'wiki.train.tokens')
    with open(file_name, 'r') as f:
        lines = f.readlines()
    # Uppercase letters are converted to lowercase ones
    paragraphs = [line.strip().lower().split(' . ')
                  for line in lines if len(line.split(' . ')) >= 2]
    random.shuffle(paragraphs)
    return paragraphs
```

15.8.1 Xác định hàm trợ giúp cho các nhiệm vụ Pretraining

Sau đây, chúng ta bắt đầu bằng cách thực hiện các hàm trợ giúp cho hai nhiệm vụ pretraining BERT: dự đoán câu tiếp theo và mô hình ngôn ngữ đeo mặt nạ. Các chức năng trợ giúp này sẽ được gọi sau khi chuyển đổi cơ thể văn bản thô thành tập dữ liệu của định dạng lý tưởng để pretrain BERT.

Tạo nhiệm vụ dự đoán câu tiếp theo

Theo mô tả của Section 15.7.5, hàm `_get_next_sentence` tạo ra một ví dụ đào tạo cho nhiệm vụ phân loại nhị phân.

```
#@save
def _get_next_sentence(sentence, next_sentence, paragraphs):
    if random.random() < 0.5:
        is_next = True
    else:
        # `paragraphs` is a list of lists of lists
        next_sentence = random.choice(random.choice(paragraphs))
        is_next = False
    return sentence, next_sentence, is_next
```

Hàm sau tạo ra các ví dụ đào tạo cho dự đoán câu tiếp theo từ đầu vào `paragraph` bằng cách gọi hàm `_get_next_sentence`. Ở đây `paragraph` là danh sách các câu, trong đó mỗi câu là danh sách các token. Đối số `max_len` chỉ định độ dài tối đa của một chuỗi đầu vào BERT trong quá trình đào tạo trước.

```
#@save
def _get_nsp_data_from_paragraph(paragraph, paragraphs, vocab, max_len):
    nsp_data_from_paragraph = []
    for i in range(len(paragraph) - 1):
        tokens_a, tokens_b, is_next = _get_next_sentence(
            paragraph[i], paragraph[i + 1], paragraphs)
        # Consider 1 '<cls>' token and 2 '<sep>' tokens
        if len(tokens_a) + len(tokens_b) + 3 > max_len:
            continue
        tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
        nsp_data_from_paragraph.append((tokens, segments, is_next))
    return nsp_data_from_paragraph
```

Tạo tác vụ mô hình hóa ngôn ngữ đeo mặt nạ

Để tạo ra các ví dụ đào tạo cho tác vụ mô hình hóa ngôn ngữ được đeo mặt nạ từ một chuỗi đầu vào BERT, chúng tôi xác định hàm `_replace_mlm_tokens` sau. Trong đầu vào của nó, `tokens` là danh sách các mã thông báo đại diện cho chuỗi đầu vào BERT, `candidate_pred_positions` là danh sách các chỉ số mã thông báo của chuỗi đầu vào BERT không bao gồm các mã thông báo đặc biệt (mã thông báo đặc biệt không được dự đoán trong nhiệm vụ mô hình hóa ngôn ngữ đeo mặt nạ) và `num_mlm_preds` chỉ ra số dự đoán (thu hồi 15% mã thông báo ngẫu nhiên để dự đoán). Theo định nghĩa của nhiệm vụ mô hình hóa ngôn ngữ đeo mặt nạ trong Section 15.7.5, tại mỗi vị trí dự đoán, đầu vào có thể được thay thế bằng một mã thông báo "" đặc biệt hoặc một mã thông báo ngẫu nhiên, hoặc vẫn không thay đổi. Cuối cùng, hàm trả về các token đầu vào sau khi có thể thay thế, các chỉ số mã thông báo nơi dự đoán diễn ra và dán nhãn cho các dự đoán này.

```
#@save
def _replace_mlm_tokens(tokens, candidate_pred_positions, num_mlm_preds,
                        vocab):
    # Make a new copy of tokens for the input of a masked language model,
    # where the input may contain replaced '<mask>' or random tokens
    mlm_input_tokens = [token for token in tokens]
    pred_positions_and_labels = []
    # Shuffle for getting 15% random tokens for prediction in the masked
    # language modeling task
    random.shuffle(candidate_pred_positions)
    for mlm_pred_position in candidate_pred_positions:
        if len(pred_positions_and_labels) >= num_mlm_preds:
            break
        masked_token = None
        # 80% of the time: replace the word with the '<mask>' token
        if random.random() < 0.8:
            masked_token = '<mask>'
        else:
            # 10% of the time: keep the word unchanged
            if random.random() < 0.5:
                masked_token = tokens[mlm_pred_position]
            # 10% of the time: replace the word with a random word
            else:
                masked_token = random.choice(vocab.idx_to_token)
        mlm_input_tokens[mlm_pred_position] = masked_token
        pred_positions_and_labels.append(
            (mlm_pred_position, tokens[mlm_pred_position]))
    return mlm_input_tokens, pred_positions_and_labels
```

Bằng cách gọi hàm `_replace_mlm_tokens` đã nói ở trên, hàm sau lấy chuỗi đầu vào BERT (`tokens`) làm đầu vào và trả về các chỉ số của mã thông báo đầu vào (sau khi thay thế token có thể như được mô tả trong Section 15.7.5), các chỉ số mã thông báo nơi dự đoán diễn ra và các chỉ số nhãn cho những dự đoán.

```
#@save
def _get_mlm_data_from_tokens(tokens, vocab):
    candidate_pred_positions = []
    # `tokens` is a list of strings
    for i, token in enumerate(tokens):
        # Special tokens are not predicted in the masked language modeling
        # task
        if token in ['<cls>', '<sep>']:
            continue
```

(continues on next page)

(continued from previous page)

```
        candidate_pred_positions.append(i)
    # 15% of random tokens are predicted in the masked language modeling task
    num_mlm_preds = max(1, round(len(tokens) * 0.15))
    mlm_input_tokens, pred_positions_and_labels = _replace_mlm_tokens(
        tokens, candidate_pred_positions, num_mlm_preds, vocab)
    pred_positions_and_labels = sorted(pred_positions_and_labels,
                                       key=lambda x: x[0])
    pred_positions = [v[0] for v in pred_positions_and_labels]
    mlm_pred_labels = [v[1] for v in pred_positions_and_labels]
    return vocab[mlm_input_tokens], pred_positions, vocab[mlm_pred_labels]
```

15.8.2 Chuyển văn bản thành tập dữ liệu Pretraining

Bây giờ chúng tôi gần như đã sẵn sàng để tùy chỉnh một lớp `Dataset` cho BERT pretraining. Trước đó, chúng ta vẫn cần định nghĩa một hàm helper `_pad_bert_inputs` để thêm các token "" đặc biệt vào các đầu vào. Lập luận của nó `examples` chứa các đầu ra từ các chức năng trợ giúp `_get_nsp_data_from_paragraph` và `_get_mlm_data_from_tokens` cho hai nhiệm vụ pretraining.

```
#@save
def _pad_bert_inputs(examples, max_len, vocab):
    max_num_mlm_preds = round(max_len * 0.15)
    all_token_ids, all_segments, valid_lens,  = [], [], []
    all_pred_positions, all_mlm_weights, all_mlm_labels = [], [], []
    nsp_labels = []
    for (token_ids, pred_positions, mlm_pred_label_ids, segments,
         is_next) in examples:
        all_token_ids.append(torch.tensor(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype=torch.long))
        all_segments.append(torch.tensor(segments + [0] * (
            max_len - len(segments)), dtype=torch.long))
        # `valid_lens` excludes count of '<pad>' tokens
        valid_lens.append(torch.tensor(len(token_ids), dtype=torch.float32))
        all_pred_positions.append(torch.tensor(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype=torch.long))
        # Predictions of padded tokens will be filtered out in the loss via
        # multiplication of 0 weights
        all_mlm_weights.append(
            torch.tensor([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)),
                dtype=torch.float32))
        all_mlm_labels.append(torch.tensor(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype=torch.long))
        nsp_labels.append(torch.tensor(is_next, dtype=torch.long))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

Đặt các chức năng trợ giúp để tạo ra các ví dụ đào tạo về hai nhiệm vụ đào tạo trước và chức năng trợ giúp cho đầu vào đệm lại với nhau, chúng tôi tùy chỉnh lớp `_WikiTextDataset` sau làm bộ dữ liệu WikiText-2 để đào tạo trước BERT. Bằng cách thực hiện chức năng `__getitem__`, chúng ta có thể tùy ý truy cập các ví dụ pretraining (mô hình hóa ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo) được tạo ra từ một cặp câu từ cơ thể WikiText-2.

Mô hình BERT ban đầu sử dụng nhúng WordPiece có kích thước từ vựng là 30000 (Wu et al., 2016). Phương pháp mã hóa của WordPiece là một sửa đổi nhỏ của thuật toán mã hóa cặp byte ban đầu trong `sub-sec_Byte_Pair_Encoding`. Để đơn giản, chúng tôi sử dụng hàm `d2l.tokenize` để mã hóa. Các mã thông báo không thường xuyên xuất hiện ít hơn năm lần được lọc ra.

```
#@save
class _WikiTextDataset(torch.utils.data.Dataset):
    def __init__(self, paragraphs, max_len):
        # Input `paragraphs[i]` is a list of sentence strings representing a
        # paragraph; while output `paragraphs[i]` is a list of sentences
        # representing a paragraph, where each sentence is a list of tokens
        paragraphs = [d2l.tokenize(
            paragraph, token='word') for paragraph in paragraphs]
        sentences = [sentence for paragraph in paragraphs
                     for sentence in paragraph]
        self.vocab = d2l.Vocab(sentences, min_freq=5, reserved_tokens=[
            '<pad>', '<mask>', '<cls>', '<sep>'])
        # Get data for the next sentence prediction task
        examples = []
        for paragraph in paragraphs:
            examples.extend(_get_nsp_data_from_paragraph(
                paragraph, paragraphs, self.vocab, max_len))
        # Get data for the masked language model task
        examples = [(_get_mlm_data_from_tokens(tokens, self.vocab)
                      + (segments, is_next))
                     for tokens, segments, is_next in examples]
        # Pad inputs
        (self.all_token_ids, self.all_segments, self.valid_lens,
         self.all_pred_positions, self.all_mlm_weights,
         self.all_mlm_labels, self.nsp_labels) = _pad_bert_inputs(
            examples, max_len, self.vocab)

    def __getitem__(self, idx):
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx], self.all_pred_positions[idx],
                self.all_mlm_weights[idx], self.all_mlm_labels[idx],
                self.nsp_labels[idx])

    def __len__(self):
        return len(self.all_token_ids)
```

Bằng cách sử dụng hàm `_read_wiki` và lớp `_WikiTextDataset`, chúng tôi xác định `load_data_wiki` sau đây để tải xuống và tập dữ liệu WikiText-2 và tạo ra các ví dụ đào tạo trước từ nó.

```
#@save
def load_data_wiki(batch_size, max_len):
    """Load the WikiText-2 dataset."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('wikitext-2', 'wikitext-2')
    paragraphs = _read_wiki(data_dir)
    train_set = _WikiTextDataset(paragraphs, max_len)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True, num_workers=num_workers)
    return train_iter, train_set.vocab
```

Đặt kích thước lô thành 512 và độ dài tối đa của chuỗi đầu vào BERT là 64, chúng tôi in ra các hình dạng của một minibatch các ví dụ pretraining BERT. Lưu ý rằng trong mỗi chuỗi đầu vào BERT, 10 (64×0.15) vị trí được dự đoán cho nhiệm vụ mô hình hóa ngôn ngữ đeo mặt nạ.

```
batch_size, max_len = 512, 64
train_iter, vocab = load_data_wiki(batch_size, max_len)

for (tokens_X, segments_X, valid_lens_x, pred_positions_X, mlm_weights_X,
     mlm_Y, nsp_y) in train_iter:
    print(tokens_X.shape, segments_X.shape, valid_lens_x.shape,
          pred_positions_X.shape, mlm_weights_X.shape, mlm_Y.shape,
          nsp_y.shape)
    break
```

```
Downloading ../data/wikitext-2-v1.zip from https://s3.amazonaws.com/research.
↪metamind.io/wikitext/wikitext-2-v1.zip...
torch.Size([512, 64]) torch.Size([512, 64]) torch.Size([512]) torch.Size([512,
↪ 10]) torch.Size([512, 10]) torch.Size([512, 10]) torch.Size([512])
```

Cuối cùng, chúng ta hãy nhìn vào kích thước từ vựng. Ngay cả sau khi lọc các mã thông báo không thường xuyên, nó vẫn lớn hơn hai lần so với tập dữ liệu PTB.

```
len(vocab)
```

```
20256
```

15.8.3 Tóm tắt

- So sánh với tập dữ liệu PTB, tập ngày WikiText-2 giữ lại dấu câu ban đầu, chữ hoa và số, và lớn hơn hai lần.
- Chúng ta có thể tùy ý truy cập các ví dụ pretraining (mô hình hóa ngôn ngữ đeo mặt nạ và dự đoán câu tiếp theo) được tạo ra từ một cặp câu từ WikiText-2 corpus.

15.8.4 Bài tập

1. Để đơn giản, khoảng thời gian được sử dụng làm dấu phân cách duy nhất để tách câu. Hãy thử các kỹ thuật tách câu khác, chẳng hạn như spacy và NLTK. Lấy NLTK làm ví dụ. Bạn cần cài đặt NLTK trước: `pip install nltk`. Trong mã, `import nltk` đầu tiên. Sau đó, tải xuống trình mã hóa câu Punkt: `nltk.download('punkt')`. Để chia các câu như `sentences = 'Điều này thật tuyệt! Tại sao không? '`, invoking `nltk.tokenize.sent_tokenize (câu)` will return a list of two sentence strings: `['Điều này thật tuyệt! ' , 'Tại sao không? ']`.
2. Kích thước từ vựng nếu chúng ta không lọc ra bất kỳ mã thông báo không thường xuyên nào?

Discussions[197]

[197] https://discuss.d2l.ai/t/1496

15.9 Pretraining BERT

Với mô hình BERT được triển khai trong Section 15.7 và các ví dụ đào tạo trước được tạo ra từ tập dữ liệu WikiText-2 trong Section 15.8, chúng tôi sẽ pretrain BERT trên tập dữ liệu WikiText-2 trong phần này.

```
import torch
from torch import nn
from d2l import torch as d2l
```

Để bắt đầu, chúng tôi tải tập dữ liệu WikiText-2 dưới dạng các ví dụ đào tạo trước cho mô hình ngôn ngữ được đeo mặt nạ và dự đoán câu tiếp theo. Kích thước lô là 512 và độ dài tối đa của chuỗi đầu vào BERT là 64. Lưu ý rằng trong mô hình BERT ban đầu, chiều dài tối đa là 512.

```
batch_size, max_len = 512, 64
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

15.9.1 Pretraining BERT

BERT ban đầu có hai phiên bản của các kích cỡ mô hình khác nhau (Devlin et al., 2018). Mô hình cơ sở ($\text{BERT}_{\text{BASE}}$) sử dụng 12 lớp (khối mã hóa biến áp) với 768 đơn vị ẩn (kích thước ẩn) và 12 đầu tự chú ý. Mô hình lớn ($\text{BERT}_{\text{LARGE}}$) sử dụng 24 lớp với 1024 đơn vị ẩn và 16 đầu tự chú ý. Đáng chú ý, trước đây có 110 triệu tham số trong khi sau này có 340 triệu tham số. Để trình diễn một cách dễ dàng, chúng tôi xác định một BERT nhỏ, sử dụng 2 lớp, 128 đơn vị ẩn và 2 đầu tự chú ý.

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, norm_shape=[128],
                    ffn_num_input=128, ffn_num_hiddens=256, num_heads=2,
                    num_layers=2, dropout=0.2, key_size=128, query_size=128,
                    value_size=128, hid_in_features=128, mlm_in_features=128,
                    nsp_in_features=128)
devices = d2l.try_all_gpus()
loss = nn.CrossEntropyLoss()
```

Trước khi xác định vòng đào tạo, chúng tôi xác định một hàm trợ giúp `_get_batch_loss_bert`. Với phần nhỏ của các ví dụ đào tạo, chức năng này tính toán sự mất mát cho cả mô hình ngôn ngữ đeo mặt nạ và nhiệm vụ dự đoán câu tiếp theo. Lưu ý rằng sự mất mát cuối cùng của đào tạo trước BERT chỉ là tổng của cả sự mất mát mô hình hóa ngôn ngữ đeo mặt nạ và mất dự đoán câu tiếp theo.

```
#@save
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X,
                         segments_X, valid_lens_x,
                         pred_positions_X, mlm_weights_X,
                         mlm_Y, nsp_y):
    # Forward pass
    _, mlm_Y_hat, nsp_Y_hat = net(tokens_X, segments_X,
                                  valid_lens_x.reshape(-1),
                                  pred_positions_X)
    # Compute masked language model loss
    mlm_l = loss(mlm_Y_hat.reshape(-1, vocab_size), mlm_Y.reshape(-1)) *\
    mlm_weights_X.reshape(-1, 1)
    mlm_l = mlm_l.sum() / (mlm_weights_X.sum() + 1e-8)
    # Compute next sentence prediction loss
```

(continues on next page)

```
    nsp_l = loss(nsp_Y_hat, nsp_y)
    l = mlm_l + nsp_l
    return mlm_l, nsp_l, l
```

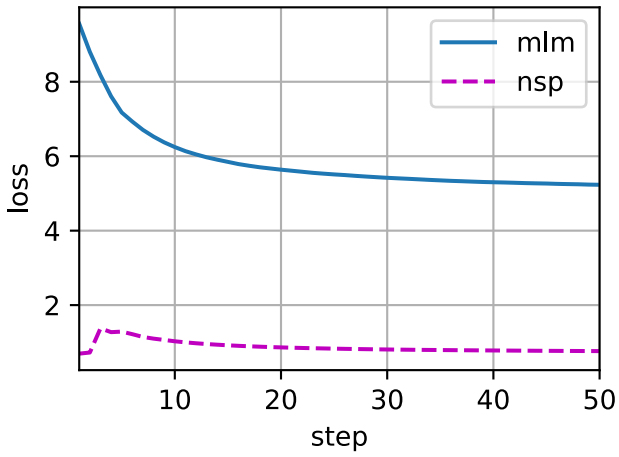
Gọi hai hàm helper nói trên, hàm `train_bert` sau đây xác định quy trình chuẩn bị BERT (`net`) trên tập dữ liệu WikiText-2 (`train_iter`). Đào tạo BERT có thể mất rất nhiều thời gian. Thay vì chỉ định số epochs để đào tạo như trong hàm `train_ch13` (xem Section 14.1), đầu vào `num_steps` của hàm sau chỉ định số bước lặp lại để đào tạo.

```
def train_bert(train_iter, net, loss, vocab_size, devices, num_steps):
    net = nn.DataParallel(net, device_ids=devices).to(devices[0])
    trainer = torch.optim.Adam(net.parameters(), lr=0.01)
    step, timer = 0, d2l.Timer()
    animator = d2l.Animator(xlabel='step', ylabel='loss',
                            xlim=[1, num_steps], legend=['mlm', 'nsp'])
    # Sum of masked language modeling losses, sum of next sentence prediction
    # losses, no. of sentence pairs, count
    metric = d2l.Accumulator(4)
    num_steps_reached = False
    while step < num_steps and not num_steps_reached:
        for tokens_X, segments_X, valid_lens_x, pred_positions_X,\
            mlm_weights_X, mlm_Y, nsp_y in train_iter:
            tokens_X = tokens_X.to(devices[0])
            segments_X = segments_X.to(devices[0])
            valid_lens_x = valid_lens_x.to(devices[0])
            pred_positions_X = pred_positions_X.to(devices[0])
            mlm_weights_X = mlm_weights_X.to(devices[0])
            mlm_Y, nsp_y = mlm_Y.to(devices[0]), nsp_y.to(devices[0])
            trainer.zero_grad()
            timer.start()
            mlm_l, nsp_l, l = _get_batch_loss_bert(
                net, loss, vocab_size, tokens_X, segments_X, valid_lens_x,
                pred_positions_X, mlm_weights_X, mlm_Y, nsp_y)
            l.backward()
            trainer.step()
            metric.add(mlm_l, nsp_l, tokens_X.shape[0], 1)
            timer.stop()
            animator.add(step + 1,
                         (metric[0] / metric[3], metric[1] / metric[3]))
            step += 1
            if step == num_steps:
                num_steps_reached = True
                break

    print(f'MLM loss {metric[0] / metric[3]:.3f}, '
          f'NSP loss {metric[1] / metric[3]:.3f}')
    print(f'{metric[2] / timer.sum():.1f} sentence pairs/sec on '
          f'{str(devices)}')
```

Chúng ta có thể vẽ cả sự mất mát mô hình hóa ngôn ngữ đeo mặt nạ và mất dự đoán câu tiếp theo trong quá trình đào tạo trước BERT.

```
train_bert(train_iter, net, loss, len(vocab), devices, 50)
```

```
MLM loss 5.232, NSP loss 0.764
4220.6 sentence pairs/sec on [device(type='cuda', index=0), device(type='cuda
↪', index=1)]
```



15.9.2 Đại diện cho văn bản với BERT

Sau khi đào tạo trước BERT, chúng ta có thể sử dụng nó để đại diện cho văn bản duy nhất, cặp văn bản hoặc bất kỳ mã thông báo nào trong đó. Hàm sau trả về các đại diện BERT (`net`) cho tất cả các token trong `tokens_a` và `tokens_b`.

```
def get_bert_encoding(net, tokens_a, tokens_b=None):
    tokens, segments = d2l.get_tokens_and_segments(tokens_a, tokens_b)
    token_ids = torch.tensor(vocab[tokens], device=devices[0]).unsqueeze(0)
    segments = torch.tensor(segments, device=devices[0]).unsqueeze(0)
    valid_len = torch.tensor(len(tokens), device=devices[0]).unsqueeze(0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X
```

Hãy xem xét câu "một cần cẩu đang bay". Nhớ lại đại diện đầu vào của BERT như đã thảo luận trong Section 15.7.4. Sau khi chèn các token đặc biệt "" (dùng để phân loại) và "" (dùng để tách), dãy đầu vào BERT có độ dài sáu. Vì số không là chỉ số của token "", `encoded_text[:, 0, :]` là đại diện BERT của toàn bộ câu đầu vào. Để đánh giá mã thông báo polysemy "crane", chúng tôi cũng in ra ba yếu tố đầu tiên của đại diện BERT của mã thông báo.

```
tokens_a = ['a', 'crane', 'is', 'flying']
encoded_text = get_bert_encoding(net, tokens_a)
# Tokens: '<cls>', 'a', 'crane', 'is', 'flying', '<sep>'
encoded_text_cls = encoded_text[:, 0, :]
encoded_text_crane = encoded_text[:, 2, :]
encoded_text.shape, encoded_text_cls.shape, encoded_text_crane[0][:3]
```

```
(torch.Size([1, 6, 128]),
 torch.Size([1, 128]),
 tensor([-0.0576, -0.0149, -1.0962], device='cuda:0', grad_fn=<SliceBackward0>
↪))
```

Bây giờ hãy xem xét một cặp câu "một người lái xe cần cẩu đến" và "anh ta vừa rời đi". Tương tự, `encoded_pair[:, 0, :]` là kết quả được mã hóa của toàn bộ cặp câu từ BERT được đào tạo trước. Lưu ý rằng ba yếu tố đầu tiên của mã thông báo polysemy "crane" khác với những yếu tố khi ngữ cảnh khác nhau. Điều này hỗ trợ rằng các đại diện BERT là nhạy cảm với ngữ cảnh.

```
tokens_a, tokens_b = ['a', 'crane', 'driver', 'came'], ['he', 'just', 'left']
encoded_pair = get_bert_encoding(net, tokens_a, tokens_b)
# Tokens: '<cls>', 'a', 'crane', 'driver', 'came', '<sep>', 'he', 'just',
# 'left', '<sep>'
encoded_pair_cls = encoded_pair[:, 0, :]
encoded_pair_crane = encoded_pair[:, 2, :]
encoded_pair.shape, encoded_pair_cls.shape, encoded_pair_crane[0][:3]
```

```
(torch.Size([1, 10, 128]),
 torch.Size([1, 128]),
 tensor([-0.1011, -0.9376, -1.2522], device='cuda:0', grad_fn=<SliceBackward0>
↪))
```

Trong Chapter 16, chúng tôi sẽ tinh chỉnh một mô hình BERT được đào tạo trước cho các ứng dụng xử lý ngôn ngữ tự nhiên ở hạ nguồn.

15.9.3 Tóm tắt

- BERT ban đầu có hai phiên bản, trong đó mô hình cơ sở có 110 triệu tham số và model lớn có 340 triệu tham số.
- Sau khi đào tạo trước BERT, chúng ta có thể sử dụng nó để đại diện cho văn bản duy nhất, cặp văn bản hoặc bất kỳ mã thông báo nào trong đó.
- Trong thí nghiệm, cùng một mã thông báo có đại diện BERT khác nhau khi ngữ cảnh của chúng khác nhau. Điều này hỗ trợ rằng các đại diện BERT là nhạy cảm với ngữ cảnh.

15.9.4 Bài tập

1. Trong thí nghiệm, chúng ta có thể thấy rằng mất mô hình hóa ngôn ngữ đeo mặt nạ cao hơn đáng kể so với mất dự đoán câu tiếp theo. Tại sao?
2. Đặt độ dài tối đa của chuỗi đầu vào BERT là 512 (giống như mô hình BERT ban đầu). Sử dụng các cấu hình của mô hình BERT ban đầu như $\text{BERT}_{\text{LARGE}}$. Bạn có gặp bất kỳ lỗi nào khi chạy phần này không? Tại sao?

Discussions[198]

[198] https://discuss.d2l.ai/t/1497

16 | Xử lý ngôn ngữ tự nhiên: Ứng dụng

Chúng tôi đã thấy cách đại diện cho thẻ trong chuỗi văn bản và đào tạo các đại diện của chúng trong Chapter 15. Các đại diện văn bản được đào tạo trước như vậy có thể được đưa vào các mô hình khác nhau cho các nhiệm vụ xử lý ngôn ngữ tự nhiên hạ nguồn khác nhau.

Trên thực tế, các chương trước đó đã thảo luận về một số ứng dụng xử lý ngôn ngữ tự nhiên *mà không cần pretraining*, just for explaining giải thích deepsâu learning học tập architectures kiến trúc. Ví dụ, trong Chapter 9, chúng tôi đã dựa vào RNN để thiết kế các mô hình ngôn ngữ để tạo ra văn bản giống như tiểu thuyết. Trong Chapter 10 và Chapter 11, chúng tôi cũng đã thiết kế các mô hình dựa trên RNNs và cơ chế chú ý cho dịch máy.

Tuy nhiên, cuốn sách này không có ý định bao gồm tất cả các ứng dụng như vậy một cách toàn diện. Thay vào đó, trọng tâm của chúng tôi là * làm thế nào để áp dụng (sâu) đại diện học ngôn ngữ để giải quyết các vấn đề xử lý ngôn ngữ tự nhiên*. Với các đại diện văn bản được đào tạo trước, chương này sẽ khám phá hai nhiệm vụ xử lý ngôn ngữ tự nhiên ở hạ nguồn phổ biến và đại diện: phân tích tình cảm và suy luận ngôn ngữ tự nhiên, phân tích văn bản đơn lẻ và mối quan hệ của các cặp văn bản, tương ứng.

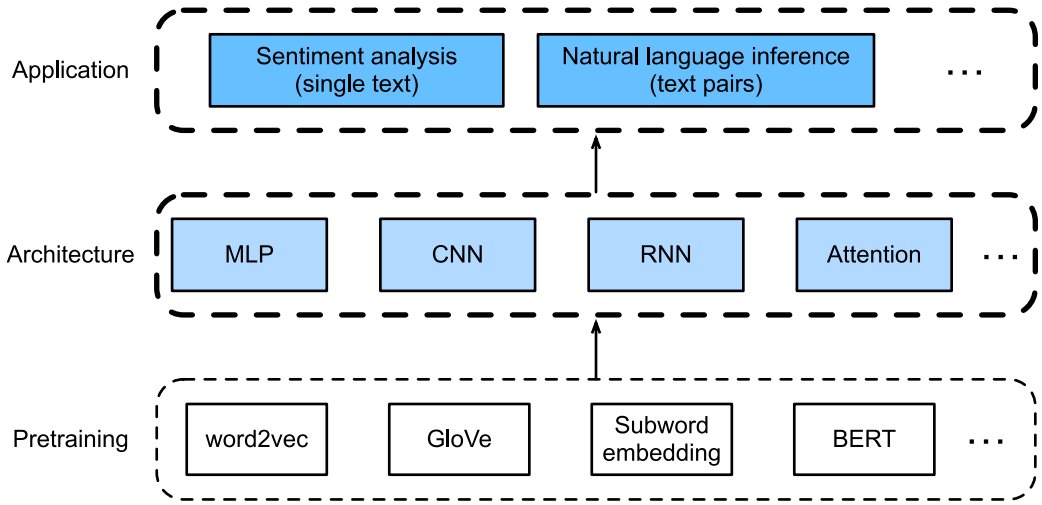


Fig. 16.1: Pretrained text representations can be fed to various deep learning architectures for different downstream natural language processing applications. This chapter focuses on how to design models for different downstream natural language processing applications.

Như được mô tả trong Fig. 16.1, chương này tập trung vào việc mô tả các ý tưởng cơ bản của việc thiết kế các mô hình xử lý ngôn ngữ tự nhiên sử dụng các loại kiến trúc học sâu khác nhau, chẳng hạn như MLP, CNN, RNN s, và sự chú ý. Mặc dù có thể kết hợp bất kỳ biểu diễn văn bản được đào tạo trước với bất kỳ kiến trúc nào cho một trong hai ứng dụng trong Fig. 16.1, chúng tôi chọn một vài kết hợp đại diện. Cụ thể, chúng tôi sẽ khám phá các kiến trúc phổ biến dựa trên RNNs và CNN để phân tích tình cảm. Đối với suy luận ngôn ngữ tự nhiên, chúng tôi chọn sự chú ý và MLP để chứng minh cách phân tích các cặp văn bản. Cuối cùng, chúng

tôi giới thiệu cách tinh chỉnh mô hình BERT được đào tạo trước cho một loạt các ứng dụng xử lý ngôn ngữ tự nhiên, chẳng hạn như ở cấp độ trình tự (phân loại văn bản đơn và phân loại cặp văn bản) và mức mã thông báo (gắn thẻ văn bản và trả lời câu hỏi). Là một trường hợp thực nghiệm cụ thể, chúng tôi sẽ tinh chỉnh BERT cho suy luận ngôn ngữ tự nhiên.

Như chúng tôi đã giới thiệu trong Section 15.7, BERT đòi hỏi những thay đổi kiến trúc tối thiểu cho một loạt các ứng dụng xử lý ngôn ngữ tự nhiên. Tuy nhiên, lợi ích này đi kèm với chi phí tinh chỉnh một số lượng lớn các thông số BERT cho các ứng dụng hạ lưu. Khi không gian hoặc thời gian bị hạn chế, những mô hình được chế tạo dựa trên MLP, CNN, RNN và sự chú ý sẽ khả thi hơn. Sau đây, chúng ta bắt đầu bằng ứng dụng phân tích tình cảm và minh họa thiết kế mô hình dựa trên RNN và CNN, tương ứng.

16.1 Phân tích tình cảm và tập dữ liệu

Với sự gia tăng của mạng xã hội trực tuyến và các nền tảng đánh giá, rất nhiều dữ liệu có ý kiến đã được ghi lại, mang tiềm năng lớn trong việc hỗ trợ các quy trình ra quyết định. *Phân tích tâm học* nghiên cứu tình cảm của mọi người trong văn bản được sản xuất của họ, chẳng hạn như đánh giá sản phẩm, bình luận blog và các cuộc thảo luận diễn đàn. Nó thích ứng dụng rộng rãi cho các lĩnh vực đa dạng như chính trị (ví dụ: phân tích tình cảm công cộng đối với các chính sách), tài chính (ví dụ: phân tích tình cảm của thị trường) và tiếp thị (ví dụ: nghiên cứu sản phẩm và quản lý thương hiệu).

Vì tình cảm có thể được phân loại là phân cực hoặc thang đo rời rạc (ví dụ: dương và âm), chúng ta có thể coi phân tích tình cảm như một nhiệm vụ phân loại văn bản, chuyển đổi một chuỗi văn bản có độ dài khác nhau thành một danh mục văn bản có độ dài cố định. Trong chương này, chúng ta sẽ sử dụng [tập dữ liệu xem xét phim lớn] của Stanford (https://ai.stanford.edu/~amaas/data/sentiment/) để phân tích tình cảm. Nó bao gồm một bộ đào tạo và một bộ thử nghiệm, có chứa 25000 đánh giá phim được tải xuống từ IMDb. Trong cả hai bộ dữ liệu, có số lượng nhãn "dương" và "tiêu cực" bằng nhau, cho thấy các cực tâm lý khác nhau.

```
import os
import torch
from torch import nn
from d2l import torch as d2l
```

16.1.1 Đọc tập dữ liệu

Đầu tiên, tải xuống và trích xuất tập dữ liệu xem xét IMDb này trong đường dẫn `../data/aclImdb`.

```
#@save
d2l.DATA_HUB['aclImdb'] = (
    'http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz',
    '01ada507287d82875905620988597833ad4e0903')

data_dir = d2l.download_extract('aclImdb', 'aclImdb')
```

Tiếp theo, đọc các tập dữ liệu đào tạo và kiểm tra. Mỗi ví dụ là một đánh giá và nhãn của nó: 1 cho "tích cực" và 0 cho "tiêu cực".

```
#@save
def read_imdb(data_dir, is_train):
    """Read the IMDb review dataset text sequences and labels."""
    data, labels = [], []
```

(continues on next page)

(continued from previous page)

```
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_train else 'test',
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') as f:
                review = f.read().decode('utf-8').replace('\n', '')
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[0:60])
```

```
# trainings: 25000
label: 1 review: Henry Hathaway was daring, as well as enthusiastic, for his
label: 1 review: An unassuming, subtle and lean film, "The Man in the White S
label: 1 review: Eddie Murphy really made me laugh my ass off on this HBO sta
```

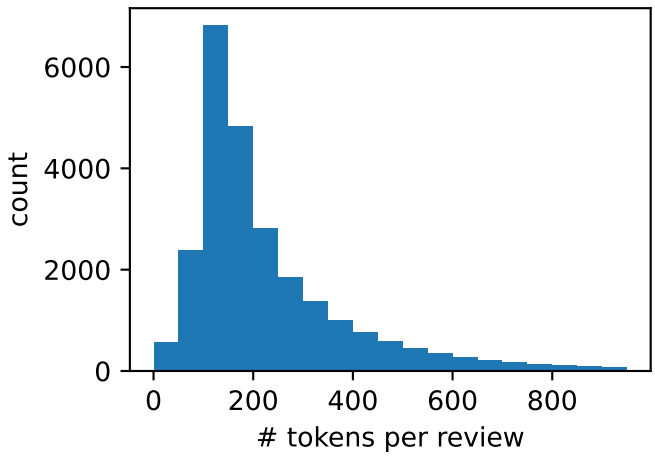
16.1.2 Xử lý trước bộ dữ liệu

Đối xử với từng từ như một mã thông báo và lọc ra các từ xuất hiện dưới 5 lần, chúng tôi tạo ra một từ vựng ra khỏi tập dữ liệu đào tạo.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])
```

Sau khi token hóa, chúng ta hãy vẽ biểu đồ của độ dài xem xét trong các mã thông báo.

```
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```



Như chúng tôi mong đợi, các đánh giá có độ dài khác nhau. Để xử lý một minibatch các đánh giá như vậy tại mỗi thời điểm, chúng tôi đặt độ dài của mỗi bài đánh giá là 500 với cắt ngắn và đệm, tương tự như bước tiền xử lý cho bộ dữ liệu dịch máy trong Section 10.5.

```
num_steps = 500  # sequence length
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
print(train_features.shape)
```

```
torch.Size([25000, 500])
```

16.1.3 Tạo bộ lặp dữ liệu

Bây giờ chúng ta có thể tạo ra các bộ lặp dữ liệu. Tại mỗi lần lặp lại, một minibatch các ví dụ được trả về.

```
train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])), 64)

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

```
X: torch.Size([64, 500]) , y: torch.Size([64])
# batches: 391
```

16.1.4 Đặt tất cả mọi thứ lại với nhau

Cuối cùng, chúng ta kết hợp các bước trên vào hàm `load_data_imdb`. Nó trả về trình lặp dữ liệu đào tạo và kiểm tra và từ vựng của tập dữ liệu xem xét IMDb.

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    """Return data iterators and the vocabulary of the IMDb review dataset."""
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in test_tokens])
    train_iter = d2l.load_array((train_features, torch.tensor(train_data[1])),
                                batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(test_data[1])),
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

16.1.5 Tóm tắt

- Phân tích tình cảm nghiên cứu tình cảm của mọi người trong văn bản được sản xuất của họ, được coi là một vấn đề phân loại văn bản biến đổi một chuỗi văn bản có độ dài khác nhau into a fixed-length cố định chiều dài text văn bản category thể loại.
- Sau khi xử lý trước, chúng ta có thể tải tập dữ liệu xem xét phim lớn của Stanford (tập dữ liệu đánh giá IMDb) vào các bộ lặp dữ liệu bằng từ vựng.

16.1.6 Bài tập

1. Những siêu tham số trong phần này chúng ta có thể sửa đổi để đẩy nhanh các mô hình phân tích tâm lý đào tạo?
2. Bạn có thể triển khai một hàm để tải tập dữ liệu của Amazon reviews[199] vào bộ lặp dữ liệu và nhãn để phân tích tình cảm không?

Discussions[200]

16.2 Phân tích tình cảm: Sử dụng mạng nơ-ron tái phát

Giống như sự tương đồng từ và các nhiệm vụ tương tự, chúng ta cũng có thể áp dụng các vectơ từ được đào tạo trước vào phân tích tình cảm. Vì tập dữ liệu đánh giá IMDb trong Section 16.1 không lớn lắm, sử dụng các đại diện văn bản đã được đào tạo trước trên corpora quy mô lớn có thể làm giảm quá mức của mô hình. Như một ví dụ cụ thể được minh họa trong Fig. 16.2.1, chúng tôi sẽ đại diện cho mỗi mã thông báo bằng cách sử dụng mô hình Glove được đào tạo trước và đưa các biểu diễn token này thành RNN hai chiều nhiều lớp để có được biểu diễn chuỗi văn bản, sẽ được chuyển thành đầu ra phân tích tình cảm (Maas et al., 2011). Đối với cùng một ứng dụng hạ lưu, chúng tôi sẽ xem xét một sự lựa chọn kiến trúc khác nhau sau này.

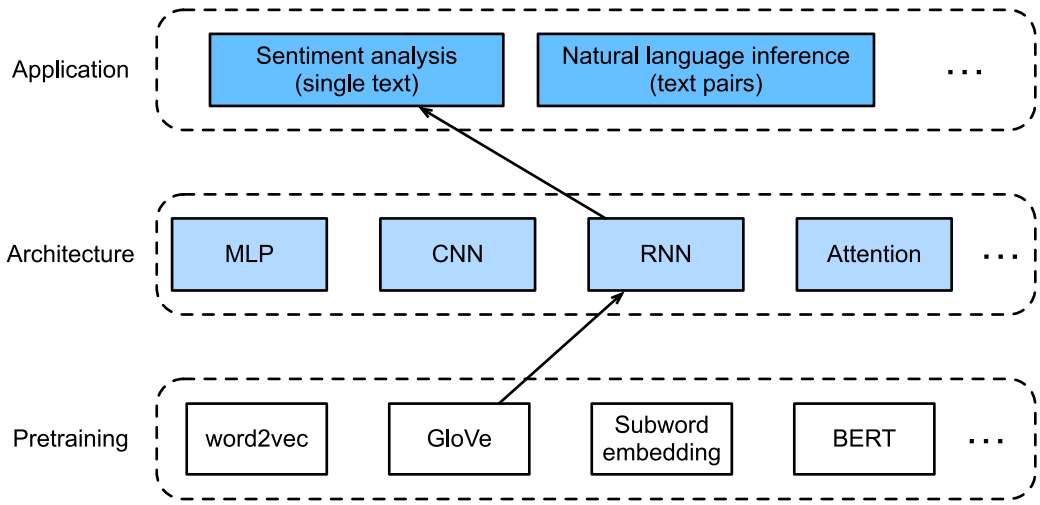


Fig. 16.2.1: This section feeds pretrained GloVe to an RNN-based architecture for sentiment analysis.

[199] https://snap.stanford.edu/data/web-Amazon.html
[200] https://discuss.d2l.ai/t/1387

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

```
Downloading ../data/aclImdb_v1.tar.gz from http://ai.stanford.edu/~amaas/data/
↪sentiment/aclImdb_v1.tar.gz...
```

16.2.1 Đại diện cho văn bản đơn với RNNs

Trong các tác vụ phân loại văn bản, chẳng hạn như phân tích tình cảm, một chuỗi văn bản có độ dài khác nhau sẽ được chuyển đổi thành các loại có độ dài cố định. Trong lớp `BiRNN` sau, trong khi mỗi token của một chuỗi văn bản được biểu diễn Glove trước riêng lẻ của nó thông qua lớp nhúng (`self.embedding`), toàn bộ chuỗi được mã hóa bởi một RNN hai chiều (`self.encoder`). Cụ thể hơn, các trạng thái ẩn (ở lớp cuối cùng) của LSTM hai chiều ở cả hai bước thời gian ban đầu và cuối cùng được nối như là biểu diễn của chuỗi văn bản. Biểu diễn văn bản đơn này sau đó được chuyển đổi thành các loại đầu ra bởi một lớp kết nối hoàn toàn (`self.decoder`) với hai đầu ra ("dương" và "âm").

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional RNN
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_layers=num_layers,
                               bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of time steps). Because
        # LSTM requires its input's first dimension to be the temporal
        # dimension, the input is transposed before obtaining token
        # representations. The output shape is (no. of time steps, batch size,
        # word vector dimension)
        embeddings = self.embedding(inputs.T)
        self.encoder.flatten_parameters()
        # Returns hidden states of the last hidden layer at different time
        # steps. The shape of `outputs` is (no. of time steps, batch size,
        # 2 * no. of hidden units)
        outputs, _ = self.encoder(embeddings)
        # Concatenate the hidden states of the initial time step and final
        # time step to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * no. of hidden units)
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1)
        # Concatenate the hidden states at the initial and final time steps as
        # the input of the fully-connected layer. Its shape is (batch size,
        # 4 * no. of hidden units)
        outs = self.decoder(encoding)
        return outs
```

Chúng ta hãy xây dựng một RNN hai chiều với hai lớp ẩn để đại diện cho văn bản duy nhất để phân tích tình

cảm.

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
```

```
def init_weights(m):
    if type(m) == nn.Linear:
        nn.init.xavier_uniform_(m.weight)
    if type(m) == nn.LSTM:
        for param in m._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(m._parameters[param])
net.apply(init_weights);
```

16.2.2 Đang tải Pretrained Word Vectơ

Dưới đây chúng tôi tải các bản nhúng Glove 100 chiều được đào tạo trước (cần phải phù hợp với `embed_size`) Glove cho các mã thông báo trong từ vựng.

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

```
Downloading ../data/glove.6B.100d.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/glove.6B.100d.zip...
```

In hình dạng của các vectơ cho tất cả các mã thông báo trong từ vựng.

```
embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

```
torch.Size([49346, 100])
```

Chúng tôi sử dụng các vectơ từ được đào tạo trước này để đại diện cho các mã thông báo trong các đánh giá và sẽ không cập nhật các vectơ này trong quá trình đào tạo.

```
net.embedding.weight.data.copy_(embeds)
net.embedding.weight.requires_grad = False
```

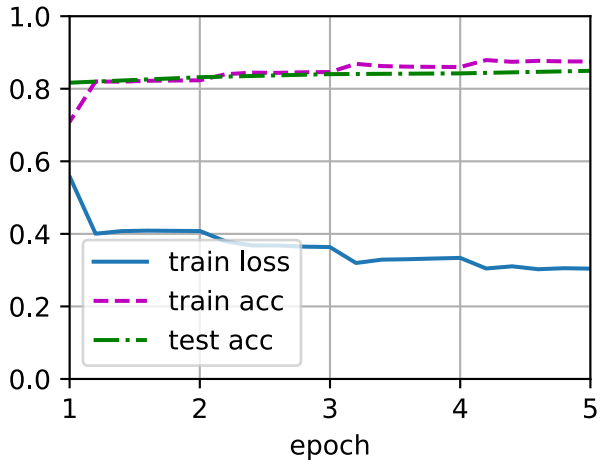
16.2.3 Đào tạo và đánh giá mô hình

Bây giờ chúng ta có thể đào tạo RNN hai chiều để phân tích tình cảm.

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.304, train acc 0.875, test acc 0.849
709.9 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```

(continues on next page)



Chúng tôi xác định hàm sau để dự đoán tình cảm của một chuỗi văn bản bằng cách sử dụng mô hình được đào tạo `net`.

```
#@save
def predict_sentiment(net, vocab, sequence):
    """Predict the sentiment of a text sequence."""
    sequence = torch.tensor(vocab[sequence.split()], device=d2l.try_gpu())
    label = torch.argmax(net(sequence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

Cuối cùng, chúng ta hãy sử dụng mô hình được đào tạo để dự đoán tình cảm cho hai câu đơn giản.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

16.2.4 Tóm tắt

- Vectơ từ được đào tạo trước có thể đại diện cho các mã thông báo riêng lẻ trong một chuỗi văn bản.
- Các RNN hai chiều có thể đại diện cho một chuỗi văn bản, chẳng hạn như thông qua việc nối các trạng thái ẩn của nó ở các bước thời gian ban đầu và cuối cùng. Biểu diễn văn bản đơn này có thể được chuyển đổi thành các danh mục bằng cách sử dụng một lớp được kết nối hoàn toàn.

16.2.5 Bài tập

1. Tăng số lượng kỷ nguyên. Bạn có thể cải thiện việc đào tạo và kiểm tra độ chính xác? Làm thế nào về điều chỉnh các siêu tham số khác?
2. Sử dụng các vectơ từ được đào tạo trước lớn hơn, chẳng hạn như nhúng găng tay 300 chiều. Nó có cải thiện độ chính xác phân loại không?
3. Chúng ta có thể cải thiện độ chính xác phân loại bằng cách sử dụng token hóa spacy không? Bạn cần cài đặt spacy (`pip install spacy`) và cài đặt gói tiếng Anh (`python -m spacy download en`). Trong mã, đầu tiên, nhập khẩu spacy (`import spacy`). Sau đó, tải gói spacy tiếng Anh (`spacy_en = spacy.load('en')`). Cuối cùng, xác định hàm `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` và thay thế hàm `tokenizer` ban đầu. Lưu ý các dạng mã thông báo cụm từ khác nhau trong Glove và spacy. Ví dụ, cụm từ token "new york" có dạng "new-york" trong Glove và dạng "new york" sau khi mã hóa spacy.

Discussions[201]

16.3 Phân tích tình cảm: Sử dụng mạng thần kinh phức tạp

Trong Chapter 7, chúng tôi đã điều tra các cơ chế xử lý dữ liệu hình ảnh hai chiều với CNN hai chiều, được áp dụng cho các tính năng cục bộ như pixel liền kề. Mặc dù ban đầu được thiết kế cho tầm nhìn máy tính, CNN cũng được sử dụng rộng rãi để xử lý ngôn ngữ tự nhiên. Nói một cách đơn giản, chỉ cần nghĩ về bất kỳ chuỗi văn bản nào như một hình ảnh một chiều. Bằng cách này, CNN một chiều có thể xử lý các tính năng địa phương như n-gram trong văn bản.

Trong phần này, chúng ta sẽ sử dụng mô hình *textCNN* để chứng minh cách thiết kế kiến trúc CNN để đại diện cho văn bản duy nhất (Kim, 2014). So với Fig. 16.2.1 sử dụng một kiến trúc RNN với Glove pretraining để phân tích tình cảm, sự khác biệt duy nhất trong Fig. 16.3.1 nằm ở sự lựa chọn của kiến trúc.

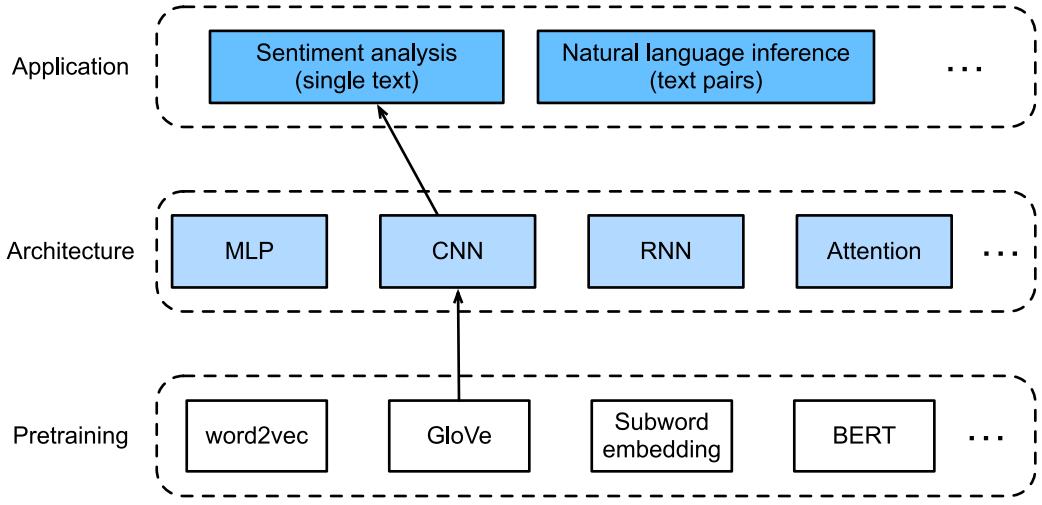


Fig. 16.3.1: This section feeds pretrained GloVe to a CNN-based architecture for sentiment analysis.

[201] https://discuss.d2l.ai/t/1424

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

16.3.1 Một chiều Convolutions

Trước khi giới thiệu mô hình, chúng ta hãy xem cách hoạt động phức tạp một chiều. Hãy nhớ rằng nó chỉ là một trường hợp đặc biệt của một sự phức tạp hai chiều dựa trên hoạt động tương quan chéo.

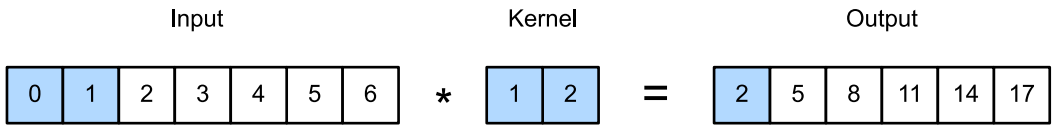


Fig. 16.3.2: One-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 1 + 1 \times 2 = 2$.

Như thể hiện trong Fig. 16.3.2, trong trường hợp một chiều, cửa sổ phức tạp trượt từ trái sang phải qua tensor đầu vào. Trong quá trình trượt, subtensor đầu vào (ví dụ, 0 và 1 trong Fig. 16.3.2) chứa trong cửa sổ covolution ở một vị trí nhất định và tensor hạt nhân (ví dụ, 1 và 2 trong Fig. 16.3.2) được nhân lên elementwise. Tổng các phép nhân này cho giá trị vô hướng đơn lẻ (ví dụ, $0 \times 1 + 1 \times 2 = 2$ trong Fig. 16.3.2) tại vị trí tương ứng của tensor đầu ra.

Chúng tôi thực hiện tương quan chéo một chiều trong hàm `corr1d` sau đây. Với tensor đầu vào `X` và tensor hạt nhân `K`, nó trả về tensor đầu ra `Y`.

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

Chúng ta có thể xây dựng tensor đầu vào `X` và tensor kernel `K` từ Fig. 16.3.2 để xác nhận đầu ra của việc thực hiện tương quan chéo một chiều ở trên.

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1, 2])
corr1d(X, K)
```

```
tensor([ 2.,  5.,  8., 11., 14., 17.])
```

Đối với bất kỳ đầu vào một chiều nào có nhiều kênh, hạt nhân tích tụ cần phải có cùng số kênh đầu vào. Sau đó, đối với mỗi kênh, thực hiện thao tác tương quan chéo trên tensor một chiều của đầu vào và tensor một chiều của hạt nhân covolution, tổng kết kết quả trên tất cả các kênh để tạo ra tensor đầu ra một chiều. Fig. 16.3.3 cho thấy một hoạt động tương quan chéo một chiều với 3 kênh đầu vào.

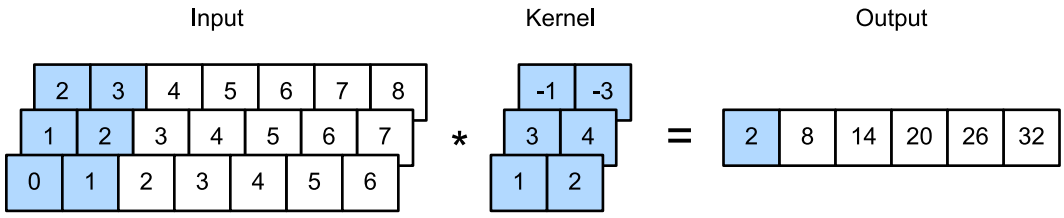


Fig. 16.3.3: One-dimensional cross-correlation operation with 3 input channels. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Chúng tôi có thể thực hiện hoạt động tương quan chéo một chiều cho nhiều kênh đầu vào và xác nhận kết quả trong Fig. 16.3.3.

```
def corr1d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimension) of `X` and
    # `K`. Then, add them together
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
              [1, 2, 3, 4, 5, 6, 7],
              [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

```
tensor([ 2.,  8., 14., 20., 26., 32.])
```

Lưu ý rằng các mối tương quan chéo một chiều đa đầu vào kênh tương đương với tương quan chéo hai chiều đơn vào-kênh. Để minh họa, một dạng tương đương của tương quan chéo một chiều đa đầu vào kênh trong Fig. 16.3.3 là tương quan chéo hai chiều kênh đầu vào đơn trong Fig. 16.3.4, trong đó chiều cao của hạt nhân phức tạp phải giống như của tensor đầu vào.

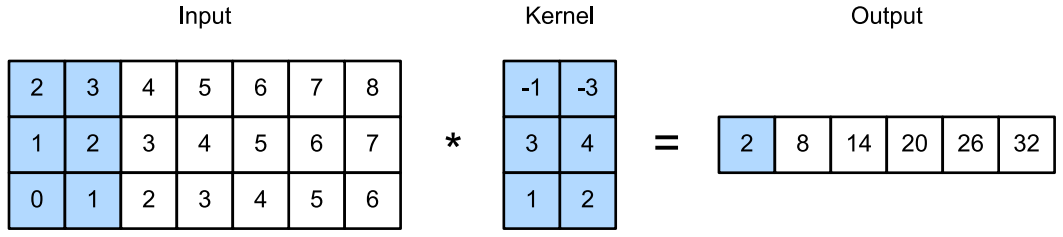


Fig. 16.3.4: Two-dimensional cross-correlation operation with a single input channel. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Cả hai đầu ra trong Fig. 16.3.2 và Fig. 16.3.3 chỉ có một kênh. Tương tự như các kết hợp hai chiều với nhiều kênh đầu ra được mô tả trong Section 7.4.2, chúng ta cũng có thể chỉ định nhiều kênh đầu ra cho các phức tạp một chiều.

16.3.2 Max-Over-Time Pooling

Tương tự, chúng ta có thể sử dụng pooling để trích xuất giá trị cao nhất từ biểu diễn trình tự như là tính năng quan trọng nhất qua các bước thời gian. * max-overtime pooling* được sử dụng trong textCNN hoạt động giống như tổng hợp tối đa toàn cầu một chiều [Collobert.Weston.Bottou.ea.2011]. Đối với đầu vào đa kênh trong đó mỗi kênh lưu trữ các giá trị ở các bước thời gian khác nhau, đầu ra tại mỗi kênh là giá trị tối đa cho kênh đó. Lưu ý rằng tổng hợp tối đa thời gian cho phép các số bước thời gian khác nhau tại các kênh khác nhau.

16.3.3 Mô hình textCNN

Sử dụng sự kết hợp một chiều và tổng hợp tối đa thời gian, mô hình textCNN lấy các biểu diễn token được đào tạo trước riêng lẻ làm đầu vào, sau đó lấy và chuyển đổi biểu diễn trình tự cho ứng dụng hạ lưu.

Đối với một chuỗi văn bản duy nhất với mã thông báo n được đại diện bởi các vectơ d chiều, chiều rộng, chiều cao và số kênh của tensor đầu vào là n, 1 và d, tương ứng. Mô hình textCNN biến đổi đầu vào thành đầu ra như sau:

1. Xác định nhiều hạt nhân phức tạp một chiều và thực hiện các thao tác phức tạp riêng biệt trên các đầu vào. Các hạt nhân có độ rộng khác nhau có thể nắm bắt các tính năng cục bộ giữa các số lượng khác nhau của các mã thông báo liền kề.
2. Thực hiện tổng hợp tối đa thời gian trên tất cả các kênh đầu ra, và sau đó nối tất cả các đầu ra tổng hợp vô hướng như một vectơ.
3. Chuyển đổi vectơ nối thành các loại đầu ra bằng cách sử dụng lớp được kết nối hoàn toàn. Dropout có thể được sử dụng để giảm overfitting.

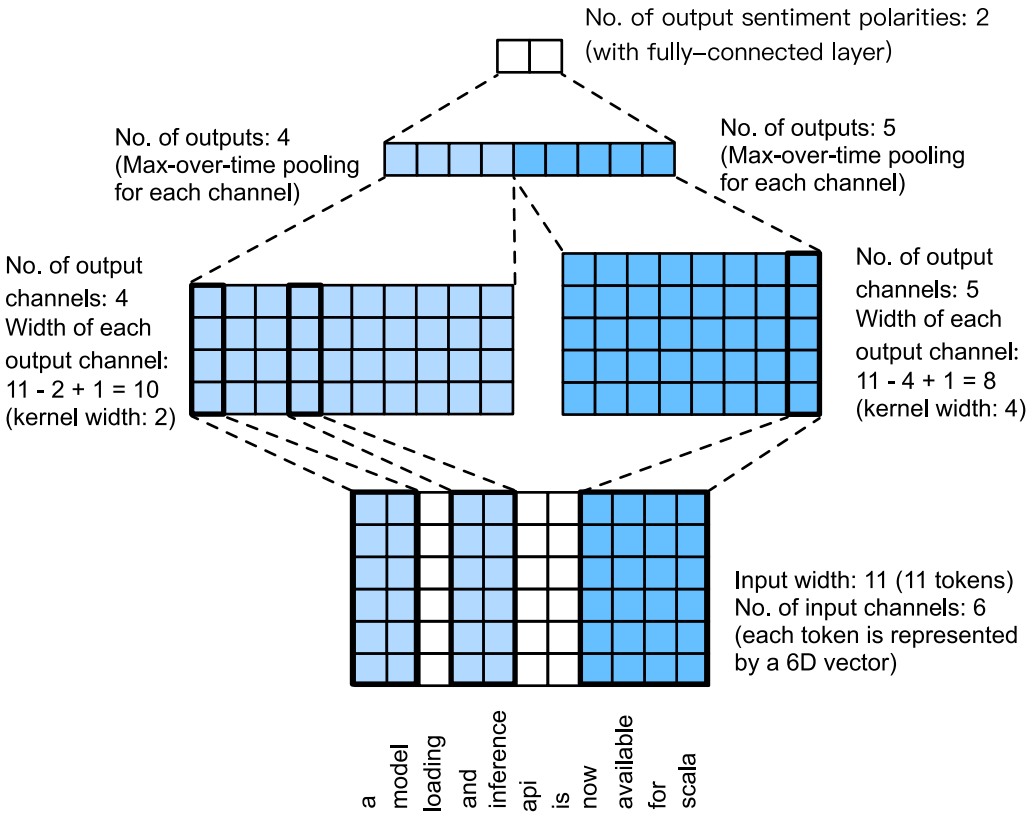


Fig. 16.3.5: The model architecture of textCNN.

Fig. 16.3.5 minh họa kiến trúc mô hình của textCNN với một ví dụ cụ thể. Đầu vào là một câu với 11 mã thông báo, trong đó mỗi mã thông báo được biểu diễn bởi một vectơ 6 chiều. Vì vậy, chúng tôi có một đầu vào 6 kênh với chiều rộng 11. Xác định hai hạt nhân phức tạp một chiều có chiều rộng 2 và 4, với 4 và 5 kênh đầu ra, tương ứng. Họ sản xuất 4 kênh đầu ra với chiều rộng $11-2+1=10$ và 5 kênh đầu ra với chiều rộng $11-4+1=8$. Mặc dù chiều rộng khác nhau của 9 kênh này, tổng hợp tối đa thời gian cho một vectơ 9 chiều nối, cuối cùng được chuyển thành vectơ đầu ra 2 chiều cho các dự đoán tình cảm nhị phân.

Xác định mô hình

Chúng tôi thực hiện mô hình textCNN trong lớp sau. So với mô hình RNN hai chiều trong Section 16.2, bên cạnh việc thay thế các lớp tái phát bằng các lớp phức tạp, chúng tôi cũng sử dụng hai lớp nhúng: một có trọng lượng có thể huấn luyện và một có trọng lượng cố định.

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes, num_channels,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer not to be trained
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # The max-over-time pooling layer has no parameters, so this instance
        # can be shared
```

(continues on next page)

(continued from previous page)

```
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k))

    def forward(self, inputs):
        # Concatenate two embedding layer outputs with shape (batch size, no.
        # of tokens, token vector dimension) along vectors
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(inputs)), dim=2)
        # Per the input format of one-dimensional convolutional layers,
        # rearrange the tensor so that the second dimension stores channels
        embeddings = embeddings.permute(0, 2, 1)
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, a tensor of shape (batch size, no. of channels, 1) is
        # obtained. Remove the last dimension and concatenate along channels
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings))), dim=-1)
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Hãy để chúng tôi tạo một ví dụ textCNN. Nó có 3 lớp phức tạp với độ rộng hạt nhân là 3, 4 và 5, tất cả đều có 100 kênh đầu ra.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_channels)

def init_weights(m):
    if type(m) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(m.weight)

net.apply(init_weights);
```

Đang tải Pretrained Word Vectơ

Tương tự như Section 16.2, chúng tôi tải các nhúng Glove 100 chiều được đào tạo trước dưới dạng các biểu diễn mã thông báo được khởi tạo. Các đại diện token này (trọng lượng nhúng) sẽ được đào tạo trong `embedding` và cố định vào `constant_embedding`.

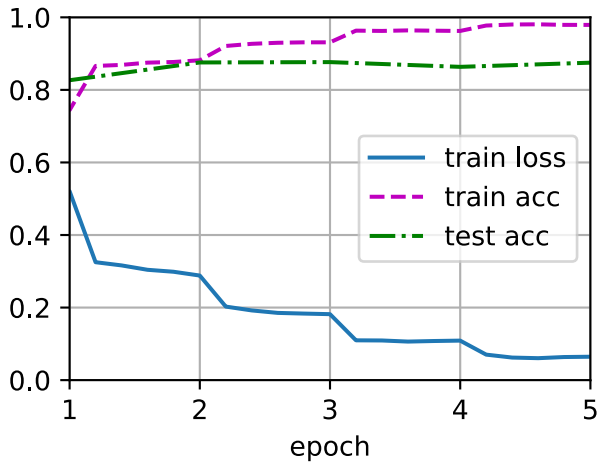
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

Đào tạo và đánh giá mô hình

Bây giờ chúng ta có thể đào tạo mô hình textCNN để phân tích tình cảm.

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.064, train acc 0.979, test acc 0.875
3284.0 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Dưới đây chúng tôi sử dụng mô hình được đào tạo để dự đoán tình cảm cho hai câu đơn giản.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
'positive'
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

```
'negative'
```

16.3.4 Tóm tắt

- CNN một chiều có thể xử lý các tính năng cục bộ như n-gram trong văn bản.
- Tương quan chéo một chiều đa đầu vào kênh tương đương với tương quan chéo hai chiều đơn vào-kênh.
- Tổng hợp tối đa thời gian cho phép các số bước thời gian khác nhau tại các kênh khác nhau.
- Mô hình textCNN biến các biểu diễn token riêng lẻ thành các đầu ra ứng dụng hạ lưu bằng cách sử dụng các lớp phức tạp một chiều và các lớp tổng hợp tối đa thời gian.

16.3.5 Bài tập

1. Điều chỉnh các siêu tham số và so sánh hai kiến trúc để phân tích tình cảm trong Section 16.2 và trong phần này, chẳng hạn như độ chính xác phân loại và hiệu quả tính toán.
2. Bạn có thể cải thiện thêm độ chính xác phân loại của mô hình bằng cách sử dụng các phương pháp được giới thiệu trong các bài tập của Section 16.2 không?
3. Thêm mã hóa vị trí trong các biểu diễn đầu vào. Nó có cải thiện độ chính xác phân loại không?

Discussions[202]

16.4 Suy luận ngôn ngữ tự nhiên và tập dữ liệu

Năm Section 16.1, chúng tôi đã thảo luận về vấn đề phân tích tình cảm. Nhiệm vụ này nhằm phân loại một chuỗi văn bản đơn lẻ thành các danh mục được xác định trước, chẳng hạn như một tập hợp các phân cực tình cảm. Tuy nhiên, khi có nhu cầu quyết định xem một câu có thể được suy ra dạng khác hay loại bỏ sự dư thừa bằng cách xác định các câu tương đương về mặt ngữ nghĩa, biết cách phân loại một chuỗi văn bản là không đủ. Thay vào đó, chúng ta cần có khả năng lý luận trên các cặp chuỗi văn bản.

16.4.1 Suy luận ngôn ngữ tự nhiên

Suy luận ngôn ngữ tự nhiên nghiên cứu liệu một giả thuyết ** có thể được suy ra từ một *premise*, trong đó cả hai đều là một chuỗi văn bản. Nói cách khác, suy luận ngôn ngữ tự nhiên xác định mối quan hệ logic giữa một cặp chuỗi văn bản. Các mối quan hệ như vậy thường rơi vào ba loại:

- *Entailment*: giả thuyết có thể được suy ra từ tiền đề.
- *Contradiction*: sự phủ định của giả thuyết có thể được suy ra từ tiền đề.
- *Trung tính*: tất cả các trường hợp khác.

Suy luận ngôn ngữ tự nhiên còn được gọi là nhiệm vụ liên kết văn bản nhận dạng. Ví dụ, cặp sau sẽ được dán nhãn là *entailment* bởi vì "thể hiện tình cảm" trong giả thuyết có thể được suy ra từ "ôm nhau" trong tiền đề.

> Tiền đề: Hai phụ nữ đang ôm nhau.
>
> Giả thuyết: Hai phụ nữ đang thể hiện tình cảm.

Sau đây là một ví dụ về *mâu thuẫn * như "chạy ví dụ mã hóa" chỉ ra "không ngủ" chứ không phải là "ngủ".

> Tiền đề: Một người đàn ông đang chạy ví dụ mã hóa từ Dive into Deep Learning.
>
> Giả thuyết: Người đàn ông đang ngủ.

Ví dụ thứ ba cho thấy mối quan hệ *trung tính* vì không "nổi tiếng" hay "không nổi tiếng" đều không thể suy ra từ thực tế là "đang biểu diễn cho chúng tôi".

> Tiền đề: Các nhạc sĩ đang biểu diễn cho chúng tôi.
>
> Giả thuyết: Các nhạc sĩ nổi tiếng.

Suy luận ngôn ngữ tự nhiên đã là một chủ đề trung tâm để hiểu ngôn ngữ tự nhiên. Nó thích các ứng dụng rộng rãi, từ truy xuất thông tin đến trả lời câu hỏi tên miền mở. Để nghiên cứu vấn đề này, chúng ta sẽ bắt đầu bằng cách điều tra một tập dữ liệu chuẩn chuẩn suy luận ngôn ngữ tự nhiên phổ biến.

[202] https://discuss.d2l.ai/t/1425

16.4.2 Bộ dữ liệu Suy luận ngôn ngữ tự nhiên Stanford (SNLI)

Stanford Natural Language Inference (SNLI) Corpus là một bộ sưu tập của hơn 500000 cặp câu tiếng Anh được dán nhãn (Bowman et al., 2015). Chúng tôi tải xuống và lưu trữ tập dữ liệu SNLI được trích xuất trong đường dẫn `../data/snli_1.0`.

```
import os
import re
import torch
from torch import nn
from d2l import torch as d2l

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')

data_dir = d2l.download_extract('SNLI')
```

```
Downloading ../data/snli_1.0.zip from https://nlp.stanford.edu/projects/snli/
↪snli_1.0.zip...
```

Đọc tập dữ liệu

Tập dữ liệu SNLI ban đầu chứa thông tin phong phú hơn nhiều so với những gì chúng ta thực sự cần trong các thí nghiệm của mình. Do đó, chúng tôi định nghĩa một hàm `read_snli` để chỉ trích xuất một phần của tập dữ liệu, sau đó trả về danh sách các cơ sở, giả thuyết và nhãn của chúng.

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and labels."""
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with space
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutral': 2}
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.txt')
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0] in label_set]
    hypotheses = [extract_text(row[2]) for row in rows if row[0] in label_set]
    labels = [label_set[row[0]] for row in rows if row[0] in label_set]
    return premises, hypotheses, labels
```

Bây giờ chúng ta hãy in 3 cặp tiền đề và giả thuyết đầu tiên, cũng như nhãn của chúng ("0", "1", và "2" tương ứng với "entailment", "mâu thuẫn", và "trung lập", tương ứng).

```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], train_data[2][:3]):
    print('premise:', x0)
    print('hypothesis:', x1)
    print('label:', y)
```

```
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is training his horse for a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A person is outdoors , on a horse .
label: 0
```

Bộ huấn luyện có khoảng 550000 cặp, và bộ thử nghiệm có khoảng 10000 cặp. Sau đây cho thấy ba nhãn "entailment", "contradiction", và "neutral" được cân bằng trong cả bộ tập huấn và bộ thử nghiệm.

```
test_data = read_snli(data_dir, is_train=False)
for data in [train_data, test_data]:
    print([[row for row in data[2]].count(i) for i in range(3)])
```

```
[183416, 183187, 182764]
[3368, 3237, 3219]
```

Xác định một lớp để tải tập dữ liệu

Dưới đây chúng ta định nghĩa một class để tải tập dữ liệu SNLI bằng cách kế thừa từ lớp `Dataset` trong Gluon. Đối số `num_steps` trong hàm tạo lớp xác định độ dài của một chuỗi văn bản sao cho mỗi minibatch của chuỗi sẽ có hình dạng giống nhau. Nói cách khác, các token sau `num_steps` đầu tiên theo trình tự dài hơn được cắt tỉa, trong khi các token đặc biệt "" sẽ được thêm vào các chuỗi ngắn hơn cho đến khi độ dài của chúng trở thành `num_steps`. Bằng cách thực hiện hàm `__getitem__`, chúng ta có thể tùy ý truy cập tiền đề, giả thuyết và nhãn với chỉ số `idx`.

```
#@save
class SNLIDataset(torch.utils.data.Dataset):
    """A customized dataset to load the SNLI dataset."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_hypothesis_tokens,
                                   min_freq=5, reserved_tokens=['<pad>'])
        else:
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = torch.tensor(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')
```

(continues on next page)

```
    def _pad(self, lines):
        return torch.tensor([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pad>'])
                             for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), self.labels[idx]

    def __len__(self):
        return len(self.premises)
```

Đặt tất cả mọi thứ lại với nhau

Bây giờ chúng ta có thể gọi hàm `read_snli` và lớp `SNLIDataset` để tải xuống tập dữ liệu SNLI và trả về `DataLoader` phiên bản cho cả bộ đào tạo và thử nghiệm, cùng với từ vựng của bộ đào tạo. Đáng chú ý là chúng ta phải sử dụng từ vựng được xây dựng từ bộ đào tạo như của bộ thử nghiệm. Do đó, bất kỳ mã thông báo mới nào từ bộ thử nghiệm sẽ không được biết đến với mô hình được đào tạo trên bộ đào tạo.

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """Download the SNLI dataset and return data iterators and vocabulary."""
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vocab)
    train_iter = torch.utils.data.DataLoader(train_set, batch_size,
                                             shuffle=True,
                                             num_workers=num_workers)
    test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                            shuffle=False,
                                            num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab
```

Ở đây chúng ta đặt kích thước lô thành 128 và độ dài chuỗi là 50 và gọi hàm `load_data_snli` để lấy các bộ lặp dữ liệu và từ vựng. Sau đó, chúng tôi in kích thước từ vựng.

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

```
read 549367 examples
read 9824 examples
```

```
18678
```

Bây giờ chúng tôi in hình dạng của minibatch đầu tiên. Trái ngược với phân tích tâm lý, chúng tôi có hai đầu vào `X[0]` và `X[1]` đại diện cho các cặp cơ sở và giả thuyết.

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

```
torch.Size([128, 50])
torch.Size([128, 50])
torch.Size([128])
```

16.4.3 Tóm tắt

- Suy luận ngôn ngữ tự nhiên nghiên cứu liệu một giả thuyết có thể được suy ra từ một tiền đề, trong đó cả hai đều là một chuỗi văn bản.
- Trong suy luận ngôn ngữ tự nhiên, các mối quan hệ giữa cơ sở và giả thuyết bao gồm sự đòi hỏi, mâu thuẫn, và trung lập.
- Stanford Natural Language Inference (SNLI) Corpus là một tập dữ liệu chuẩn phổ biến của suy luận ngôn ngữ tự nhiên.

16.4.4 Bài tập

1. Dịch máy từ lâu đã được đánh giá dựa trên sự phù hợp bề ngoài n gram giữa bản dịch đầu ra và bản dịch chân lý. Bạn có thể thiết kế một biện pháp để đánh giá kết quả dịch máy bằng cách sử dụng suy luận ngôn ngữ tự nhiên không?
2. Làm thế nào chúng ta có thể thay đổi các siêu tham số để giảm kích thước từ vựng?

Discussions[203]

16.5 Suy luận ngôn ngữ tự nhiên: Sử dụng sự chú ý

Chúng tôi đã giới thiệu nhiệm vụ suy luận ngôn ngữ tự nhiên và tập dữ liệu SNLI trong Section 16.4. Theo quan điểm của nhiều mô hình dựa trên kiến trúc phức tạp và sâu sắc, Parikh et al. đề xuất giải quyết suy luận ngôn ngữ tự nhiên với các cơ chế chú ý và gọi nó là một "mô hình chú ý có thể phân hủy" (Parikh et al., 2016). Điều này dẫn đến một mô hình không có lớp lặp lại hoặc phức tạp, đạt được kết quả tốt nhất tại thời điểm đó trên tập dữ liệu SNLI với ít tham số hơn nhiều. Trong phần này, chúng tôi sẽ mô tả và thực hiện phương pháp dựa trên sự chú ý này (với MLP s) cho suy luận ngôn ngữ tự nhiên, như được mô tả trong Fig. 16.5.1.

[203] https://discuss.d2l.ai/t/1388

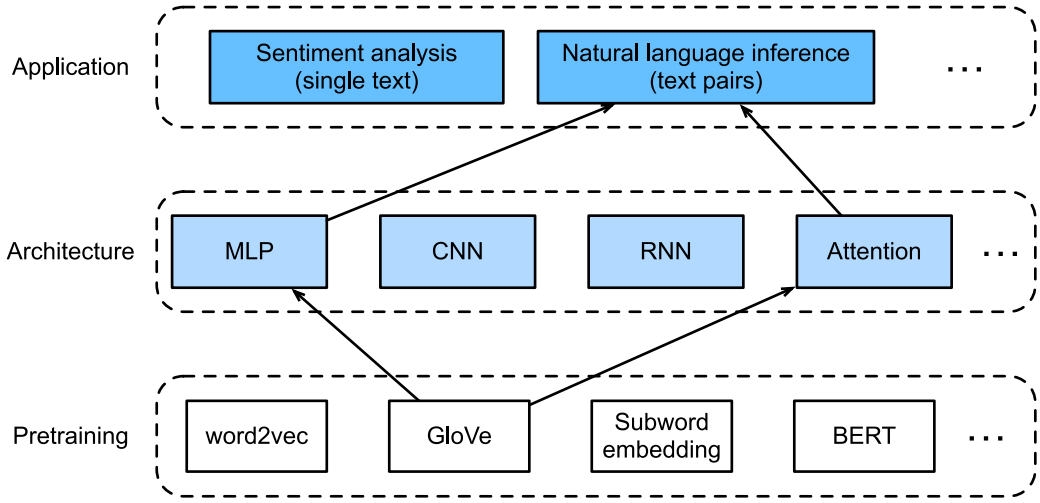


Fig. 16.5.1: This section feeds pretrained GloVe to an architecture based on attention and MLPs for natural language inference.

16.5.1 Mô hình

Đơn giản hơn là duy trì thứ tự của mã thông báo trong cơ sở và giả thuyết, chúng ta chỉ có thể sắp xếp các mã thông báo trong một chuỗi văn bản với mỗi mã thông báo khác, và ngược lại, sau đó so sánh và tổng hợp thông tin đó để dự đoán mối quan hệ logic giữa cơ sở và giả thuyết. Tương tự như liên kết các mã thông báo giữa các câu nguồn và câu đích trong dịch máy, sự liên kết của các mã thông báo giữa các cơ sở và giả thuyết có thể được thực hiện gọn gàng bởi các cơ chế chú ý.

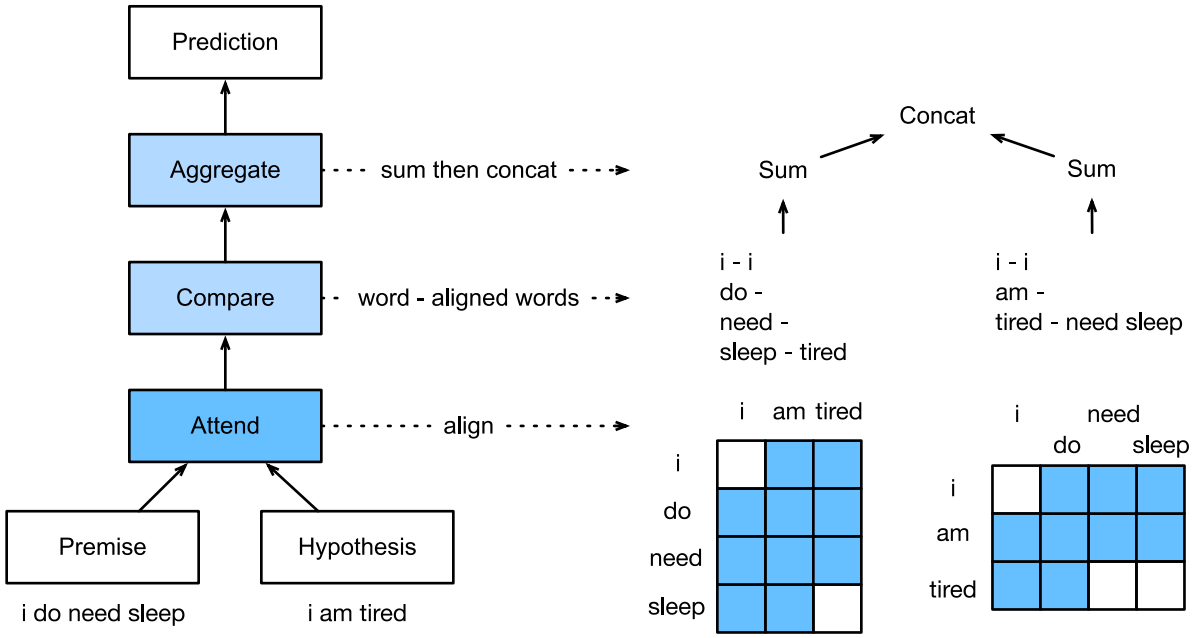


Fig. 16.5.2: Natural language inference using attention mechanisms.

Fig. 16.5.2 mô tả phương pháp suy luận ngôn ngữ tự nhiên sử dụng các cơ chế chú ý. Ở cấp độ cao, nó bao gồm ba bước được đào tạo chung: tham dự, so sánh và tổng hợp. Chúng tôi sẽ minh họa cho họ từng bước trong những điều sau đây.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

Tham dự

Bước đầu tiên là căn chỉnh mã thông báo trong một chuỗi văn bản với mỗi mã thông báo trong chuỗi khác. Giả sử rằng tiền đề là "tôi cần ngủ" và giả thuyết là "tôi mệt mỏi". Do sự giống nhau về ngữ nghĩa, chúng ta có thể muốn căn chỉnh "i" trong giả thuyết với "i" trong tiền đề, và căn chỉnh "mệt mỏi" trong giả thuyết với "ngủ" trong tiền đề. Tương tự như vậy, chúng ta có thể muốn căn chỉnh "i" trong tiền đề với "i" trong giả thuyết, và căn chỉnh "cần" và "ngủ" trong tiền đề với "mệt mỏi" trong giả thuyết. Lưu ý rằng căn chỉnh như vậy là * mềm* sử dụng trung bình có trọng số, trong đó trọng lượng lớn lý tưởng được liên kết với các mã thông báo được căn chỉnh. Để dễ trình diễn, Fig. 16.5.2 cho thấy sự liên kết như vậy theo cách * cứng*.

Bây giờ chúng tôi mô tả sự liên kết mềm bằng cách sử dụng các cơ chế chú ý chi tiết hơn. Biểu thị bằng $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m)$ và $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ tiền đề và giả thuyết, có số lượng mã thông báo là m và n, tương ứng, trong đó $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$ ($i = 1, \ldots, m, j = 1, \ldots, n$) là một vector từ d chiều. Để liên kết mềm, chúng tôi tính toán trọng lượng chú ý $e_{ij} \in \mathbb{R}$ như

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \tag{16.5.1}$$

trong đó hàm f là một MLP được định nghĩa trong hàm `mlp` sau. Kích thước đầu ra của f được chỉ định bởi đối số `num_hiddens` của `mlp`.

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

Cần nhấn mạnh rằng, trong (16.5.1) f mất đầu vào $\mathbf{a}_i$ và $\mathbf{b}_j$ riêng biệt hơn là lấy một cặp chúng lại với nhau làm đầu vào. Thủ thuật * decomposition* này dẫn đến chỉ $m + n$ ứng dụng (độ phức tạp tuyến tính) của f chứ không phải là mn ứng dụng (độ phức tạp bậc hai).

Bình thường hóa trọng lượng chú ý trong (16.5.1), chúng tôi tính toán trung bình trọng số của tất cả các vectơ mã thông báo trong giả thuyết để có được đại diện của giả thuyết được liên kết nhẹ nhàng với mã thông báo được lập chỉ mục bởi i trong tiền đề:

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n} \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j. \tag{16.5.2}$$

Tương tự như vậy, chúng tôi tính toán sự liên kết mềm của các token tiền đề cho mỗi token được lập chỉ mục

bởi j trong giả thuyết:

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i. \tag{16.5.3}$$

Dưới đây chúng tôi xác định lớp `Attend` để tính toán sự liên kết mềm của các giả thuyết (`beta`) với cơ sở đầu vào `A` và căn chỉnh mềm của cơ sở (`alpha`) với giả thuyết đầu vào `B`.

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (`batch_size`, no. of tokens in sequence A/B,
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (`batch_size`, no. of tokens in sequence A/B,
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (`batch_size`, no. of tokens in sequence A,
        # no. of tokens in sequence B)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
        # Shape of `beta`: (`batch_size`, no. of tokens in sequence A,
        # `embed_size`), where sequence B is softly aligned with each token
        # (axis 1 of `beta`) in sequence A
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # Shape of `alpha`: (`batch_size`, no. of tokens in sequence B,
        # `embed_size`), where sequence A is softly aligned with each token
        # (axis 1 of `alpha`) in sequence B
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-1), A)
        return beta, alpha
```

So sánh

Trong bước tiếp theo, chúng ta so sánh một token trong một chuỗi với chuỗi khác được liên kết nhẹ nhàng với mã thông báo đó. Lưu ý rằng trong liên kết mềm, tất cả các mã thông báo từ một chuỗi, mặc dù với trọng lượng chú ý có thể khác nhau, sẽ được so sánh với một mã thông báo trong chuỗi khác. Để dễ dàng trình diễn, Fig. 16.5.2 cặp token với mã thông báo được liên kết theo cách * hard*. Ví dụ, giả sử rằng bước tham dự xác định rằng "nhu cầu" và "ngủ" trong tiền đề đều phù hợp với "mệt mỏi" trong giả thuyết, cặp "mệt mỏi — cần ngủ" sẽ được so sánh.

Trong bước so sánh, chúng ta cho phép nối (toán tử $[\cdot, \cdot]$) của mã thông báo từ một chuỗi và liên kết các token từ chuỗi khác vào một hàm g (một MLP):

$$\begin{aligned} \mathbf{v}_{A,i} &= g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m \\ \mathbf{v}_{B,j} &= g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n. \end{aligned} \tag{16.5.4}$$

Năm (16.5.4), $\mathbf{v}_{A,i}$ là sự so sánh giữa token i trong tiền đề và tất cả các token giả thuyết được liên kết nhẹ nhàng với mã thông báo i; trong khi $\mathbf{v}_{B,j}$ là so sánh giữa token j trong giả thuyết và tất cả các token tiền đề được phù hợp nhẹ nhàng với token j. Lớp `Compare` sau đây định nghĩa như bước so sánh.

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(torch.cat([A, beta], dim=2))
        V_B = self.g(torch.cat([B, alpha], dim=2))
        return V_A, V_B
```

Tổng hợp

Với hai bộ vectơ so sánh $\mathbf{v}_{A,i}$ $(i = 1, \ldots, m)$ và $\mathbf{v}_{B,j}$ $(j = 1, \ldots, n)$ trên tay, trong bước cuối cùng chúng ta sẽ tổng hợp thông tin đó để suy ra mối quan hệ logic. Chúng tôi bắt đầu bằng cách tổng hợp cả hai bộ:

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}. \tag{16.5.5}$$

Tiếp theo chúng ta cung cấp kết nối của cả hai kết quả tóm tắt thành hàm h (một MLP) để có được kết quả phân loại của mối quan hệ logic:

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \tag{16.5.6}$$

Bước tổng hợp được xác định trong lớp `Aggregate` sau.

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # Feed the concatenation of both summarization results into an MLP
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=1)))
        return Y_hat
```

Đặt tất cả mọi thứ lại với nhau

Bằng cách đặt các bước tham dự, so sánh và tổng hợp lại với nhau, chúng tôi xác định mô hình chú ý có thể phân hủy để cùng đào tạo ba bước này.

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_inputs_attend=100,
                 num_inputs_compare=200, num_inputs_agg=400, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
```

(continues on next page)

(continued from previous page)

```
        self.compare = Compare(num_inputs_compare, num_hiddens)
        # There are 3 possible outputs: entailment, contradiction, and neutral
        self.aggregate = Aggregate(num_inputs_agg, num_hiddens, num_outputs=3)

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

16.5.2 Đào tạo và đánh giá mô hình

Bây giờ chúng ta sẽ đào tạo và đánh giá mô hình chú ý phân hủy được xác định trên bộ dữ liệu SNLI. Chúng tôi bắt đầu bằng cách đọc tập dữ liệu.

Đọc tập dữ liệu

Chúng tôi tải xuống và đọc tập dữ liệu SNLI bằng chức năng được xác định trong Section 16.4. Kích thước lô và chiều dài chuỗi được đặt thành 256 và 50, tương ứng.

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

```
read 549367 examples
read 9824 examples
```

Tạo mô hình

Chúng tôi sử dụng nhúng Glove 100 chiều được đào tạo trước để đại diện cho các mã thông báo đầu vào. Do đó, chúng ta xác định trước kích thước của vectơ $\mathbf{a}_i$ và $\mathbf{b}_j$ trong (16.5.1) là 100. Kích thước đầu ra của các chức năng f trong (16.5.1) và g trong (16.5.4) được đặt thành 200. Sau đó, chúng ta tạo ra một ví dụ mô hình, khởi tạo các tham số của nó và tải Glove nhúng để khởi tạo vectơ của mã thông báo đầu vào.

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

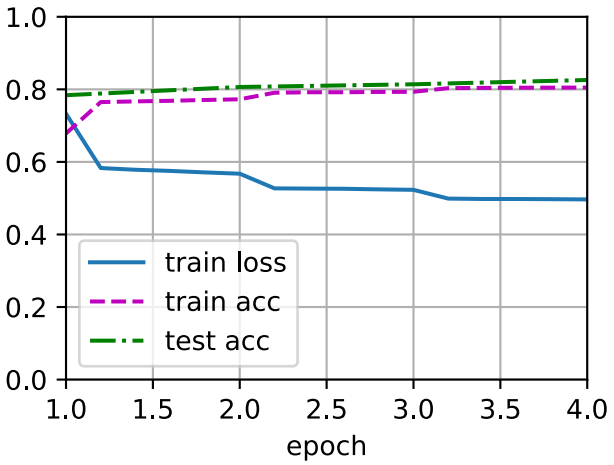
Đào tạo và đánh giá mô hình

Trái ngược với hàm `split_batch` trong Section 13.5 lấy các đầu vào đơn như chuỗi văn bản (hoặc hình ảnh), chúng tôi định nghĩa một hàm `split_batch_multi_inputs` để lấy nhiều đầu vào như cơ sở và giả thuyết trong minibatches.

Bây giờ chúng ta có thể đào tạo và đánh giá mô hình trên tập dữ liệu SNLI.

```
lr, num_epochs = 0.001, 4
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.497, train acc 0.805, test acc 0.826
19939.0 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



Sử dụng mô hình

Cuối cùng, xác định hàm dự đoán để xuất ra mối quan hệ logic giữa một cặp tiền đề và giả thuyết.

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """Predict the logical relationship between the premise and hypothesis."""
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu())
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.try_gpu())
    label = torch.argmax(net([premise.reshape((1, -1)),
                          hypothesis.reshape((1, -1))]), dim=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
            else 'neutral'
```

Chúng ta có thể sử dụng mô hình được đào tạo để có được kết quả suy luận ngôn ngữ tự nhiên cho một cặp câu mẫu.

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

```
'contradiction'
```

16.5.3 Tóm tắt

- Mô hình chú ý có thể phân hủy bao gồm ba bước để dự đoán mối quan hệ logic giữa cơ sở và giả thuyết: tham dự, so sánh và tổng hợp.
- Với các cơ chế chú ý, chúng ta có thể sắp xếp các mã thông báo theo một chuỗi văn bản với mỗi mã thông báo khác và ngược lại. Sự liên kết như vậy là mềm bằng cách sử dụng trung bình trọng số, trong đó trọng lượng lớn lý tưởng được liên kết với các mã thông báo được căn chỉnh.
- Thủ thuật phân hủy dẫn đến độ phức tạp tuyến tính mong muốn hơn so với độ phức tạp bậc hai khi tính toán trọng lượng chú ý.
- Chúng ta có thể sử dụng vectơ từ được đào tạo trước làm biểu diễn đầu vào cho nhiệm vụ xử lý ngôn ngữ tự nhiên hạ lưu như suy luận ngôn ngữ tự nhiên.

16.5.4 Bài tập

1. Đào tạo mô hình với các kết hợp khác của các siêu tham số. Bạn có thể có được độ chính xác tốt hơn trên bộ thử nghiệm?
2. Những hạn chế lớn của mô hình chú ý phân hủy đối với suy luận ngôn ngữ tự nhiên là gì?
3. Giả sử rằng chúng ta muốn có được mức độ tương đồng ngữ nghĩa (ví dụ, một giá trị liên tục giữa 0 và 1) cho bất kỳ cặp câu nào. Làm thế nào chúng ta sẽ thu thập và gắn nhãn cho tập dữ liệu? Bạn có thể thiết kế một mô hình với các cơ chế chú ý?

Discussions[204]

16.6 Tinh chỉnh BERT cho các ứng dụng cấp Sequence-Level và Token-Level

Trong các phần trước của chương này, chúng tôi đã thiết kế các mô hình khác nhau cho các ứng dụng xử lý ngôn ngữ tự nhiên, chẳng hạn như dựa trên RNN, CNN, sự chú ý và MLP. Các mô hình này rất hữu ích khi có không gian hoặc thời gian hạn chế, tuy nhiên, việc tạo ra một mô hình cụ thể cho mọi nhiệm vụ xử lý ngôn ngữ tự nhiên thực tế là không khả thi. Trong Section 15.7, chúng tôi giới thiệu một mô hình pretraining, BERT, đòi hỏi những thay đổi kiến trúc tối thiểu cho một loạt các nhiệm vụ xử lý ngôn ngữ tự nhiên. Một mặt, tại thời điểm đề xuất của mình, BERT đã cải thiện trạng thái của nghệ thuật về các nhiệm vụ xử lý ngôn ngữ tự nhiên khác nhau. Mặt khác, như đã lưu ý trong Section 15.9, hai phiên bản của mô hình BERT ban đầu đi kèm với 110 triệu và 340 triệu thông số. Do đó, khi có đủ tài nguyên tính toán, chúng ta có thể xem xét BERT tinh chỉnh cho các ứng dụng xử lý ngôn ngữ tự nhiên ở hạ nguồn.

Sau đây, chúng tôi khái quát hóa một tập hợp con của các ứng dụng xử lý ngôn ngữ tự nhiên dưới dạng cấp trình tự và cấp token. Ở cấp độ trình tự, chúng tôi giới thiệu cách chuyển đổi biểu diễn BERT của đầu vào văn bản thành nhãn đầu ra trong phân loại văn bản đơn và phân loại cặp văn bản hoặc hồi quy. Ở cấp độ token, chúng tôi sẽ giới thiệu ngắn gọn các ứng dụng mới như gắn thẻ văn bản và trả lời câu hỏi và làm sáng tỏ cách BERT có thể đại diện cho đầu vào của chúng và được chuyển đổi thành nhãn đầu ra. Trong quá trình tinh

[204] https://discuss.d2l.ai/t/1530

chỉnh, “những thay đổi kiến trúc tối thiểu” theo yêu cầu của BERT trên các ứng dụng khác nhau là các lớp được kết nối đầy đủ bổ sung. Trong quá trình học được giám sát về một ứng dụng hạ lưu, các tham số của các lớp bổ sung được học từ đầu trong khi tất cả các tham số trong mô hình BERT được đào tạo trước đều được tinh chỉnh.

16.6.1 Phân loại văn bản đơn

Phân loại văn bản đơn lấy một chuỗi văn bản duy nhất làm đầu vào và xuất kết quả phân loại của nó. Bên cạnh phân tích tình cảm mà chúng ta đã nghiên cứu trong chương này, Corpus of Linguistic Acceptability (cola) cũng là một tập dữ liệu để phân loại văn bản đơn lẻ, đánh giá liệu một câu cho trước có được chấp nhận về mặt ngữ pháp hay không (Warstadt et al., 2019). Ví dụ, “Tôi nên học.” là chấp nhận được nhưng “Tôi nên học.” thì không.

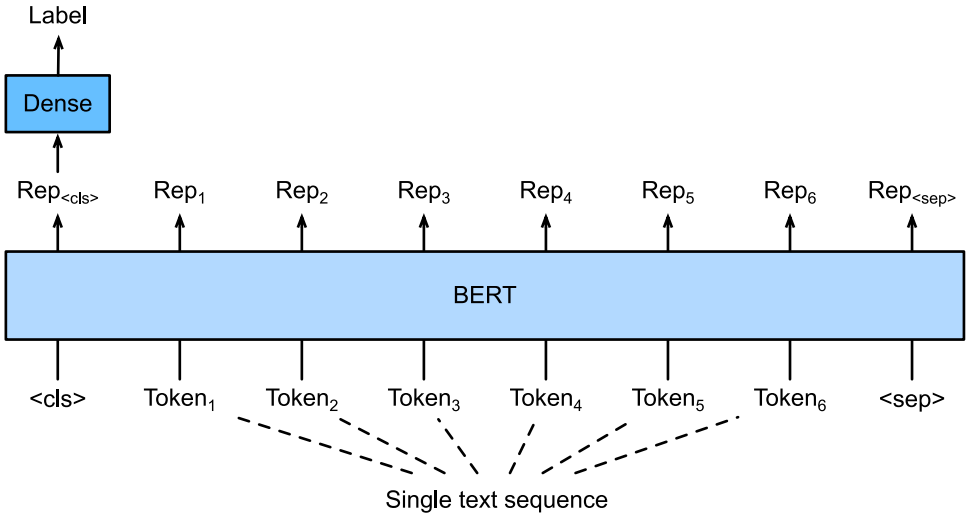


Fig. 16.6.1: Fine-tuning BERT for single text classification applications, such as sentiment analysis and testing linguistic acceptability. Suppose that the input single text has six tokens.

Section 15.7 mô tả biểu diễn đầu vào của BERT. Dãy đầu vào BERT thể hiện rõ ràng cả hai cặp văn bản và văn bản đơn lẻ, trong đó mã thông báo phân loại đặc biệt “” được sử dụng để phân loại trình tự và mã thông báo phân loại đặc biệt “” đánh dấu phần cuối của văn bản đơn hoặc tách một cặp văn bản. Như thể hiện trong Fig. 16.6.1, trong các ứng dụng phân loại văn bản đơn lẻ, đại diện BERT của mã thông báo phân loại đặc biệt “” mã hóa thông tin của toàn bộ chuỗi văn bản đầu vào. Là biểu diễn của văn bản đơn đầu vào, nó sẽ được đưa vào một MLP nhỏ bao gồm các lớp được kết nối hoàn toàn (dày đặc) để xuất ra sự phân bố của tất cả các giá trị nhãn rời rạc.

16.6.2 Phân loại cặp văn bản hoặc hồi quy

Chúng tôi cũng đã kiểm tra suy luận ngôn ngữ tự nhiên trong chương này. Nó thuộc về phân loại cặp văn bản*, một loại ứng dụng phân loại một cặp văn bản.

Lấy một cặp văn bản làm đầu vào nhưng xuất ra một giá trị liên tục, *ngữ nghĩa tương tự văn bản* là một tác vụ hồi quy cặp văn bản* phổ biến. Nhiệm vụ này đo lường sự giống nhau ngữ nghĩa của câu. Ví dụ, trong tập dữ liệu Điểm chuẩn tương tự ngữ nghĩa văn bản, điểm tương đồng của một cặp câu là một thang thứ tự khác nhau, từ 0 (không có nghĩa là chồng lên nhau) đến 5 (có nghĩa là tương đương) (Cer et al., 2017). Mục tiêu là

để dự đoán những điểm số này. Các ví dụ từ tập dữ liệu điểm chuẩn tương tự ngữ nghĩa văn bản bao gồm (câu 1, câu 2, điểm tương đồng):

- "Một chiếc máy bay đang cất cánh. ", "Một chiếc máy bay đang cất cánh. ", 5.000;
- "Một người phụ nữ đang ăn một cái gì đó. ", "Một người phụ nữ đang ăn thịt. ", 3.000;
- "Một người phụ nữ đang nhảy múa. ", "Một người đàn ông đang nói chuyện. ", 0.000.

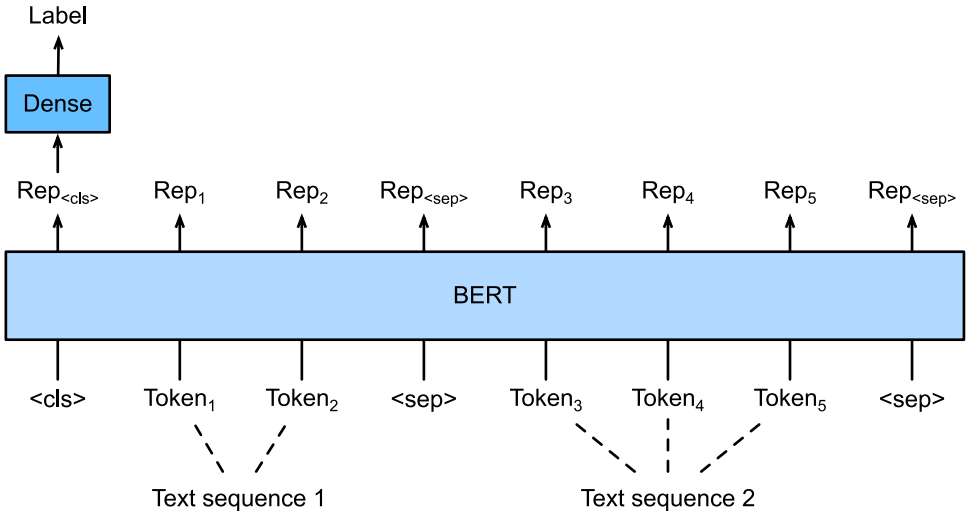


Fig. 16.6.2: Fine-tuning BERT for text pair classification or regression applications, such as natural language inference and semantic textual similarity. Suppose that the input text pair has two and three tokens.

So sánh với phân loại văn bản đơn trong Fig. 16.6.1, tinh chỉnh BERT cho phân loại cặp văn bản trong Fig. 16.6.2 là khác nhau trong biểu diễn đầu vào. Đối với các tác vụ hồi quy cặp văn bản như tương tự văn bản ngữ nghĩa, những thay đổi tầm thường có thể được áp dụng như xuất ra giá trị nhãn liên tục và sử dụng tổn thất bình phương trung bình: chúng phổ biến cho hồi quy.

16.6.3 Gắn thẻ văn bản

Bây giờ chúng ta hãy xem xét các tác vụ cấp token, chẳng hạn như *text tagging*, trong đó mỗi token được gán một nhãn. Trong số các tác vụ gắn thẻ văn bản, *một phần của lời nói tagging* gán cho mỗi từ một phần của thẻ phát biểu (ví dụ, tính từ và xác định) theo vai trò của từ trong câu. Ví dụ, theo bộ thẻ Penn Treebank II, câu "xe của John Smith là mới" nên được gắn thẻ là "NNP (danh từ, số ít thích hợp) NNP POS (sở hữu kết thúc) NN (danh từ, số ít hoặc khối lượng) VB (động từ, dạng cơ sở) JJ (tính từ)".

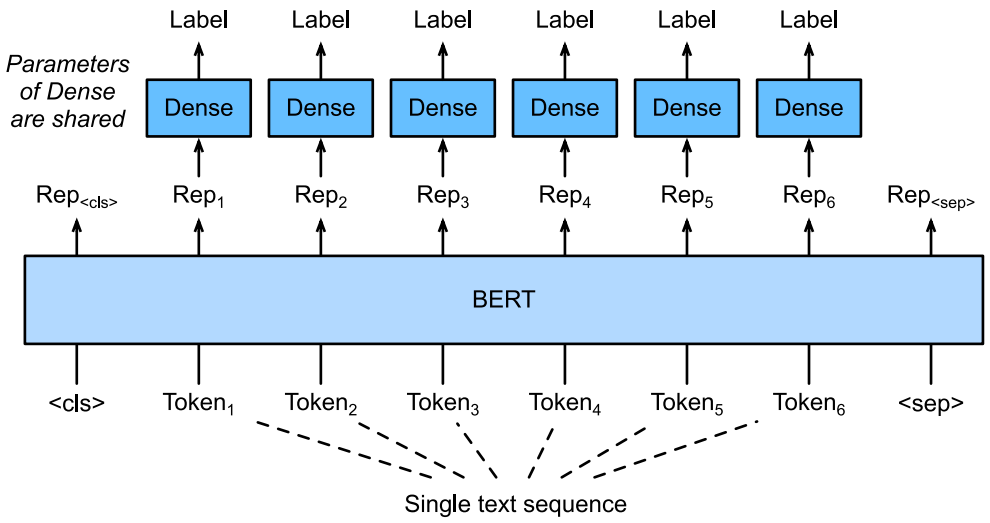


Fig. 16.6.3: Fine-tuning BERT for text tagging applications, such as part-of-speech tagging. Suppose that the input single text has six tokens.

Tinh chỉnh BERT cho các ứng dụng gắn thẻ văn bản được minh họa trong Fig. 16.6.3. So với Fig. 16.6.1, sự khác biệt duy nhất nằm ở chỗ gắn thẻ văn bản, biểu diễn BERT của * mỗi token* của văn bản đầu vào được đưa vào cùng một lớp được kết nối đầy đủ để xuất nhãn của mã thông báo, chẳng hạn như thẻ một phần lời nói.

16.6.4 Câu hỏi trả lời

As another khác token-level cấp application ứng dụng, *câu trả lời câu hỏi* phản ánh khả năng đọc hiểu. Ví dụ, tập dữ liệu trả lời câu hỏi Stanford (Squad v1.1) bao gồm đọc đoạn và câu hỏi, trong đó câu trả lời cho mọi câu hỏi chỉ là một đoạn văn bản (khoảng văn bản) từ đoạn văn mà câu hỏi là khoảng (Rajpurkar et al., 2016). Để giải thích, hãy xem xét một đoạn văn "Một số chuyên gia báo cáo rằng hiệu quả của mặt nạ là không kết luận. Tuy nhiên, các nhà sản xuất mặt nạ nhấn mạnh rằng các sản phẩm của họ, chẳng hạn như mặt nạ phòng độc N95, có thể bảo vệ chống lại virus." và một câu hỏi "Ai nói rằng mặt nạ phòng độc N95 có thể bảo vệ chống lại virus?". Câu trả lời nên là khoảng văn bản "nhà sản xuất mặt nạ" trong đoạn văn. Do đó, mục tiêu trong Squad v1.1 là dự đoán bắt đầu và kết thúc của khoảng văn bản trong đoạn văn được đưa ra một cặp câu hỏi và đoạn văn.

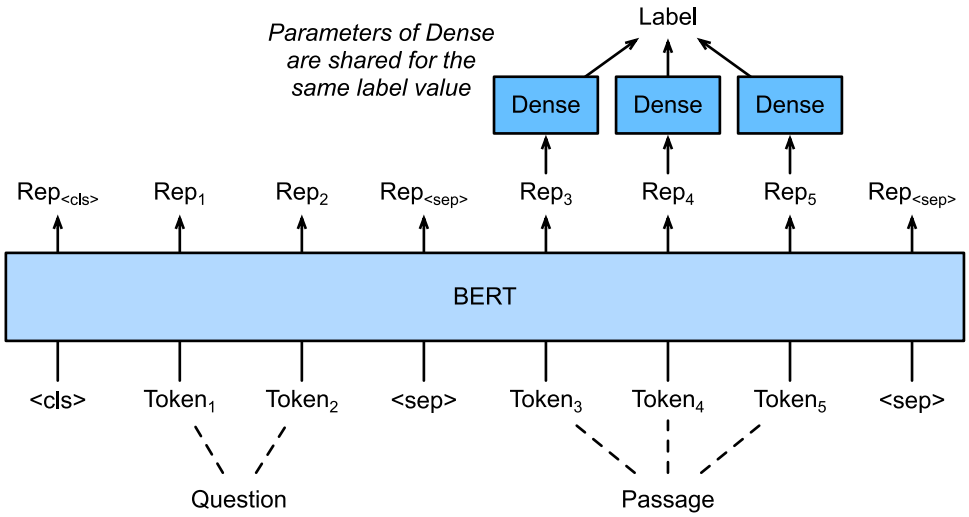


Fig. 16.6.4: Fine-tuning BERT for question answering. Suppose that the input text pair has two and three tokens.

Để tinh chỉnh BERT để trả lời câu hỏi, câu hỏi và đoạn văn được đóng gói như chuỗi văn bản đầu tiên và thứ hai, tương ứng, trong đầu vào của BERT. Để dự đoán vị trí bắt đầu của khoảng văn bản, cùng một lớp kết nối đầy đủ bổ sung sẽ biến đổi biểu diễn BERT của bất kỳ mã thông báo nào từ việc thông qua vị trí i thành một điểm vô hướng s_i. Điểm số như vậy của tất cả các mã thông qua được chuyển đổi thêm bởi hoạt động softmax thành một phân phối xác suất, do đó mỗi vị trí mã thông báo i trong đoạn văn được gán một xác suất p_i là sự bắt đầu của khoảng văn bản. Dự đoán kết thúc khoảng văn bản giống như trên, ngoại trừ các tham số trong lớp được kết nối hoàn toàn bổ sung của nó độc lập với các tham số để dự đoán bắt đầu. Khi dự đoán kết thúc, bất kỳ mã thông báo đoạn văn nào của vị trí i được chuyển đổi bởi cùng một lớp kết nối hoàn toàn thành một điểm vô hướng e_i. Fig. 16.6.4 mô tả BERT tinh chỉnh để trả lời câu hỏi.

Đối với câu trả lời câu hỏi, mục tiêu đào tạo của học tập được giám sát là đơn giản như tối đa hóa các log-likelihoods của các vị trí bắt đầu và kết thúc nền đất-chân lý. Khi dự đoán khoảng, chúng ta có thể tính điểm số $s_i + e_j$ cho một khoảng hợp lệ từ vị trí i đến vị trí j ($i \leq j$) và xuất nhịp với điểm cao nhất.

16.6.5 Tóm tắt

- BERT yêu cầu các thay đổi kiến trúc tối thiểu (các lớp kết nối đầy đủ) cho các ứng dụng xử lý ngôn ngữ tự nhiên cấp trình tự và cấp token, chẳng hạn như phân loại văn bản đơn (ví dụ: phân tích tình cảm và khả năng chấp nhận ngôn ngữ kiểm tra), phân loại cặp văn bản hoặc hồi quy (ví dụ: ngôn ngữ tự nhiên suy luận và tương tự văn bản ngữ nghĩa), gắn thẻ văn bản (ví dụ, gắn thẻ một phần lời nói) và trả lời câu hỏi.
- Trong quá trình học được giám sát về một ứng dụng hạ lưu, các tham số của các lớp bổ sung được học từ đầu trong khi tất cả các tham số trong mô hình BERT được đào tạo trước đều được tinh chỉnh.

16.6.6 Bài tập

1. Hãy để chúng tôi thiết kế một thuật toán công cụ tìm kiếm cho các bài báo tin tức. Khi hệ thống nhận được một truy vấn (ví dụ: "ngành công nghiệp dầu khí trong đợt bùng phát coronavirus"), nó sẽ trả lại một danh sách xếp hạng các bài báo có liên quan nhất đến truy vấn. Giả sử rằng chúng ta có một nhóm lớn các bài báo tin tức và một số lượng lớn các truy vấn. Để đơn giản hóa vấn đề, giả sử rằng bài viết có liên quan nhất đã được dán nhãn cho mỗi truy vấn. Làm thế nào chúng ta có thể áp dụng lấy mẫu âm (xem Section 15.2.1) và BERT trong thiết kế thuật toán?
2. Làm thế nào chúng ta có thể tận dụng BERT trong các mô hình ngôn ngữ đào tạo?
3. Chúng ta có thể tận dụng BERT trong dịch máy không?

Discussions[205]

16.7 Suy luận ngôn ngữ tự nhiên: Tinh chỉnh BERT

Trong các phần trước của chương này, chúng tôi đã thiết kế một kiến trúc dựa trên sự chú ý (trong Section 16.5) cho nhiệm vụ suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI (như được mô tả trong Section 16.4). Bây giờ chúng tôi xem lại nhiệm vụ này bằng cách tinh chỉnh BERT. Như đã thảo luận trong Section 16.6, suy luận ngôn ngữ tự nhiên là một bài toán phân loại cặp văn bản cấp trình tự, và tinh chỉnh BERT chỉ đòi hỏi một kiến trúc dựa trên MLP bổ sung, như minh họa trong Fig. 16.7.1.

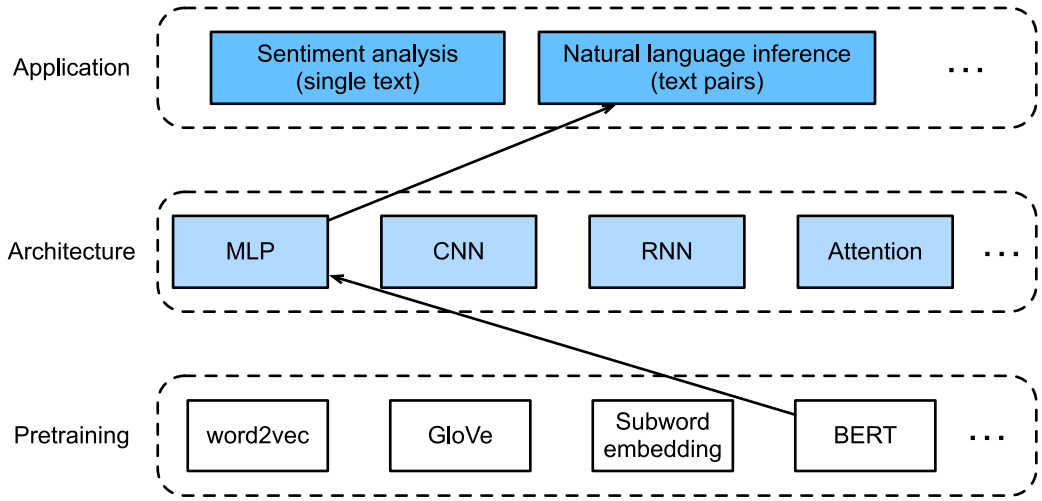


Fig. 16.7.1: This section feeds pretrained BERT to an MLP-based architecture for natural language inference.

Trong phần này, chúng tôi sẽ tải xuống một phiên bản nhỏ được đào tạo trước của BERT, sau đó tinh chỉnh nó để suy luận ngôn ngữ tự nhiên trên bộ dữ liệu SNLI.

```
import json
import multiprocessing
import os
import torch
from torch import nn
from d2l import torch as d2l
```

[205] https://discuss.d2l.ai/t/396

16.7.1 Đang tải BERT Pretrained

Chúng tôi đã giải thích cách chuẩn bị BERT trên bộ dữ liệu WikiText-2 trong Section 15.8 và Section 15.9 (lưu ý rằng mô hình BERT ban đầu được đào tạo trước trên corpora lớn hơn nhiều). Như đã thảo luận trong Section 15.9, mô hình BERT ban đầu có hàng trăm triệu thông số. Trong phần sau, chúng tôi cung cấp hai phiên bản BERT được đào tạo trước: "bert.base" lớn bằng mô hình cơ sở BERT ban đầu đòi hỏi rất nhiều tài nguyên tính toán để tinh chỉnh, trong khi "bert.small" là một phiên bản nhỏ để tạo điều kiện cho trình diễn.

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.torch.zip',
                             '225d66f04cae318b841a13d32af3acc165f253ac')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.torch.zip',
                              'c72329e68a732bef0452e4b96a1c341c8910f81f')
```

Hoặc mô hình BERT được đào tạo trước chứa một tập tin "vocab.json" xác định tập từ vựng và một tập tin "pretrained.params" của các tham số được đào tạo trước. Chúng tôi thực hiện chức năng `load_pretrained_model` sau đây để tải các thông số BERT được đào tạo trước.

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_layers, dropout, max_len, devices):
    data_dir = d2l.download_extract(pretrained_model)
    # Define an empty vocabulary to load the predefined vocabulary
    vocab = d2l.Vocab()
    vocab.idx_to_token = json.load(open(os.path.join(data_dir, 'vocab.json')))
    vocab.token_to_idx = {token: idx for idx, token in enumerate(
        vocab.idx_to_token)}
    bert = d2l.BERTModel(len(vocab), num_hiddens, norm_shape=[256],
                         ffn_num_input=256, ffn_num_hiddens=ffn_num_hiddens,
                         num_heads=4, num_layers=2, dropout=0.2,
                         max_len=max_len, key_size=256, query_size=256,
                         value_size=256, hid_in_features=256,
                         mlm_in_features=256, nsp_in_features=256)
    # Load pretrained BERT parameters
    bert.load_state_dict(torch.load(os.path.join(data_dir,
                                                 'pretrained.params')))
    return bert, vocab
```

Để tạo điều kiện cho trình diễn trên hầu hết các máy móc, chúng tôi sẽ tải và tinh chỉnh phiên bản nhỏ ("bert.small") của BERT được đào tạo trước trong phần này. Trong bài tập, chúng tôi sẽ chỉ ra cách tinh chỉnh "bert.base" lớn hơn nhiều để cải thiện đáng kể độ chính xác của thử nghiệm.

```
devices = d2l.try_all_gpus()
bert, vocab = load_pretrained_model(
    'bert.small', num_hiddens=256, ffn_num_hiddens=512, num_heads=4,
    num_layers=2, dropout=0.1, max_len=512, devices=devices)
```

```
Downloading ../data/bert.small.torch.zip from http://d2l-data.s3-accelerate.
↪amazonaws.com/bert.small.torch.zip...
```

16.7.2 Tập dữ liệu cho tinh chỉnh BERT

Đối với nhiệm vụ hạ lưu suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI, chúng tôi xác định một lớp tập dữ liệu tùy chỉnh `SNLIBERTDataset`. Trong mỗi ví dụ, tiền đề và giả thuyết tạo thành một cặp chuỗi văn bản và được đóng gói thành một chuỗi đầu vào BERT như mô tả trong Fig. 16.6.2. Nhớ lại Section 15.7.4 rằng ID phân đoạn được sử dụng để phân biệt tiền đề và giả thuyết trong một chuỗi đầu vào BERT. Với độ dài tối đa được xác định trước của chuỗi đầu vào BERT (`max_len`), token cuối cùng của cặp văn bản đầu vào dài hơn sẽ bị xóa cho đến khi `max_len` được đáp ứng. Để tăng tốc tạo bộ dữ liệu SNLI để tinh chỉnh BERT, chúng tôi sử dụng 4 quy trình công nhân để tạo ra các ví dụ đào tạo hoặc thử nghiệm song song.

```
class SNLIBERTDataset(torch.utils.data.Dataset):
    def __init__(self, dataset, max_len, vocab=None):
        all_premise_hypothesis_tokens = [[
            p_tokens, h_tokens] for p_tokens, h_tokens in zip(
            *[d2l.tokenize([s.lower() for s in sentences])
              for sentences in dataset[:2]])]

        self.labels = torch.tensor(dataset[2])
        self.vocab = vocab
        self.max_len = max_len
        (self.all_token_ids, self.all_segments,
         self.valid_lens) = self._preprocess(all_premise_hypothesis_tokens)
        print('read ' + str(len(self.all_token_ids)) + ' examples')

    def _preprocess(self, all_premise_hypothesis_tokens):
        pool = multiprocessing.Pool(4)  # Use 4 worker processes
        out = pool.map(self._mp_worker, all_premise_hypothesis_tokens)
        all_token_ids = [
            token_ids for token_ids, segments, valid_len in out]
        all_segments = [segments for token_ids, segments, valid_len in out]
        valid_lens = [valid_len for token_ids, segments, valid_len in out]
        return (torch.tensor(all_token_ids, dtype=torch.long),
                torch.tensor(all_segments, dtype=torch.long),
                torch.tensor(valid_lens))

    def _mp_worker(self, premise_hypothesis_tokens):
        p_tokens, h_tokens = premise_hypothesis_tokens
        self._truncate_pair_of_tokens(p_tokens, h_tokens)
        tokens, segments = d2l.get_tokens_and_segments(p_tokens, h_tokens)
        token_ids = self.vocab[tokens] + [self.vocab['<pad>']] \
                              * (self.max_len - len(tokens))
        segments = segments + [0] * (self.max_len - len(segments))
        valid_len = len(tokens)
        return token_ids, segments, valid_len

    def _truncate_pair_of_tokens(self, p_tokens, h_tokens):
        # Reserve slots for '<CLS>', '<SEP>', and '<SEP>' tokens for the BERT
        # input
        while len(p_tokens) + len(h_tokens) > self.max_len - 3:
            if len(p_tokens) > len(h_tokens):
                p_tokens.pop()
            else:
                h_tokens.pop()

    def __getitem__(self, idx):
```

(continues on next page)

(continued from previous page)

```
        return (self.all_token_ids[idx], self.all_segments[idx],
                self.valid_lens[idx]), self.labels[idx]

    def __len__(self):
        return len(self.all_token_ids)
```

Sau khi tải xuống tập dữ liệu SNLI, chúng tôi tạo ra các ví dụ đào tạo và thử nghiệm bằng cách khởi tạo lớp `SNLIBERTDataset`. Những ví dụ như vậy sẽ được đọc trong minibatches trong quá trình đào tạo và thử nghiệm suy luận ngôn ngữ tự nhiên.

```
# Reduce `batch_size` if there is an out of memory error. In the original BERT
# model, `max_len` = 512
batch_size, max_len, num_workers = 512, 128, d2l.get_dataloader_workers()
data_dir = d2l.download_extract('SNLI')
train_set = SNLIBERTDataset(d2l.read_snli(data_dir, True), max_len, vocab)
test_set = SNLIBERTDataset(d2l.read_snli(data_dir, False), max_len, vocab)
train_iter = torch.utils.data.DataLoader(train_set, batch_size, shuffle=True,
                                         num_workers=num_workers)
test_iter = torch.utils.data.DataLoader(test_set, batch_size,
                                        num_workers=num_workers)
```

```
read 549367 examples
read 9824 examples
```

16.7.3 Tinh chỉnh BERT

Như Fig. 16.6.2 chỉ ra, tinh chỉnh BERT cho suy luận ngôn ngữ tự nhiên chỉ yêu cầu thêm MLP bao gồm hai lớp được kết nối hoàn toàn (xem `self.hidden` và `self.output` trong lớp `BERTClassifier` sau). MLP này biến đổi đại diện BERT của mã thông báo “” đặc biệt, mã hóa thông tin của cả tiền đề và giả thuyết, thành ba đầu ra của suy luận ngôn ngữ tự nhiên: entailment, mâu thuẫn, và trung lập.

```
class BERTClassifier(nn.Module):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Linear(256, 3)

    def forward(self, inputs):
        tokens_X, segments_X, valid_lens_x = inputs
        encoded_X = self.encoder(tokens_X, segments_X, valid_lens_x)
        return self.output(self.hidden(encoded_X[:, 0, :]))
```

Sau đây, mô hình BERT được đào tạo trước `bert` được đưa vào phiên bản `BERTClassifier` `net` cho ứng dụng hạ lưu. Trong các triển khai phổ biến của tinh chỉnh BERT, chỉ các tham số của lớp đầu ra của MLP bổ sung (`net.output`) sẽ được học từ đầu. Tất cả các thông số của bộ mã hóa BERT được đào tạo trước (`net.encoder`) và lớp ẩn của MLP bổ sung (`net.hidden`) sẽ được tinh chỉnh.

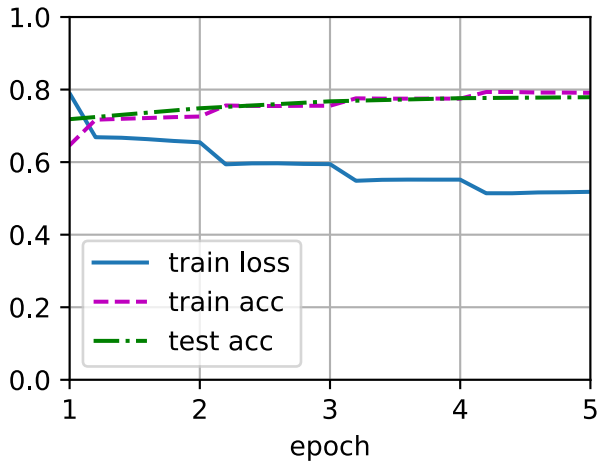
```
net = BERTClassifier(bert)
```

Nhớ lại rằng trong Section 15.7 cả lớp `MaskLM` và lớp `NextSentencePred` đều có các thông số trong MLP được sử dụng của họ. Các thông số này là một phần của những thông số trong mô hình BERT được đào tạo trước `bert`, và do đó là một phần của các thông số trong `net`. Tuy nhiên, các thông số như vậy chỉ để tính toán mất mô hình hóa ngôn ngữ đeo mặt nạ và mất dự đoán câu tiếp theo trong quá trình đào tạo trước. Hai chức năng mất mát này không liên quan đến việc tinh chỉnh các ứng dụng hạ lưu, do đó các thông số của MLP được sử dụng trong `MaskLM` và `NextSentencePred` không được cập nhật (staled) khi BERT được tinh chỉnh.

Để cho phép các tham số với gradient cũ, cờ `ignore_stale_grad=True` được đặt trong hàm `step` của `d2l.train_batch_ch13`. Chúng tôi sử dụng chức năng này để đào tạo và đánh giá mô hình `net` bằng cách sử dụng bộ đào tạo (`train_iter`) và bộ thử nghiệm (`test_iter`) của SNLI. Do các nguồn lực tính toán hạn chế, độ chính xác đào tạo và thử nghiệm có thể được cải thiện hơn nữa: chúng tôi để lại các cuộc thảo luận của nó trong các bài tập.

```
lr, num_epochs = 1e-4, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction='none')
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

```
loss 0.518, train acc 0.791, test acc 0.779
10236.7 examples/sec on [device(type='cuda', index=0), device(type='cuda',␣
↪index=1)]
```



16.7.4 Tóm tắt

- Chúng ta có thể tinh chỉnh mô hình BERT được đào tạo trước cho các ứng dụng hạ nguồn, chẳng hạn như suy luận ngôn ngữ tự nhiên trên bộ dữ liệu SNLI.
- Trong quá trình tinh chỉnh, mô hình BERT trở thành một phần của mô hình cho ứng dụng hạ lưu. Các thông số chỉ liên quan đến mất sơ bộ sẽ không được cập nhật trong quá trình tinh chỉnh.

16.7.5 Bài tập

1. Tinh chỉnh một mô hình BERT được đào tạo trước lớn hơn nhiều như mô hình cơ sở BERT ban đầu nếu tài nguyên tính toán của bạn cho phép. Đặt các đối số trong hàm `load_pretrained_model` là: thay thế 'bert.small' bằng 'bert.base', tăng giá trị lần lượt là `num_hiddens=256`, `ffn_num_hiddens=512`, `num_heads=4` và `num_layers=2` lên 768, 3072, 12 và 12. Bằng cách tăng kỷ nguyên tinh chỉnh (và có thể điều chỉnh các siêu tham số khác), bạn có thể nhận được độ chính xác thử nghiệm cao hơn 0,86 không?
2. Làm thế nào để cắt ngắn một cặp chuỗi theo tỷ lệ chiều dài của chúng? So sánh phương pháp cắt ngắn cặp này và phương pháp được sử dụng trong lớp `SNLIBERTDataset`. Ưu và nhược điểm của họ là gì?

Discussions[206]

[206] https://discuss.d2l.ai/t/1526

17 | Hệ thống Recommender

** Shuai Zhang** (*Amazon*), **Aston Zhang** (*Amazon*), và **Yi Tay** (*Google*)

Các hệ thống giới thiệu được sử dụng rộng rãi trong ngành công nghiệp và có mặt khắp nơi trong cuộc sống hàng ngày của chúng ta. Các hệ thống này được sử dụng trong một số lĩnh vực như các trang mua sắm trực tuyến (ví dụ: amazon.com), trang web dịch vụ âm nhạc/phim (ví dụ: Netflix và Spotify), cửa hàng ứng dụng di động (ví dụ: cửa hàng ứng dụng IOS và google play), quảng cáo trực tuyến, chỉ để đặt tên một vài.

Mục tiêu chính của các hệ thống giới thiệu là giúp người dùng khám phá các mặt hàng có liên quan như phim để xem, nhắn tin để đọc hoặc sản phẩm để mua, để tạo ra trải nghiệm người dùng thú vị. Hơn nữa, các hệ thống giới thiệu là một trong những hệ thống máy học mạnh nhất mà các nhà bán lẻ trực tuyến thực hiện để thúc đẩy doanh thu gia tăng. Hệ thống giới thiệu là thay thế các công cụ tìm kiếm bằng cách giảm những nỗ lực trong các tìm kiếm chủ động và người dùng đáng ngạc nhiên với các ưu đãi mà họ không bao giờ tìm kiếm. Nhiều công ty quản lý để định vị mình trước các đối thủ cạnh tranh của họ với sự trợ giúp của các hệ thống giới thiệu hiệu quả hơn. Như vậy, các hệ thống giới thiệu là trọng tâm của không chỉ cuộc sống hàng ngày của chúng ta mà còn rất không thể thiếu trong một số ngành công nghiệp.

Trong chương này, chúng tôi sẽ đề cập đến các nguyên tắc cơ bản và tiến bộ của các hệ thống giới thiệu, cùng với việc khám phá một số kỹ thuật cơ bản phổ biến để xây dựng các hệ thống giới thiệu với các nguồn dữ liệu khác nhau có sẵn và triển khai chúng. Cụ thể, bạn sẽ tìm hiểu cách dự đoán xếp hạng mà người dùng có thể cung cấp cho một mục tiềm năng, cách tạo danh sách đề xuất các mục và cách dự đoán tỷ lệ nhấp từ các tính năng phong phú. Những nhiệm vụ này là phổ biến trong các ứng dụng trong thế giới thực. Bằng cách nghiên cứu chương này, bạn sẽ nhận được kinh nghiệm thực hành liên quan đến việc giải quyết các vấn đề đề xuất thế giới thực với không chỉ các phương pháp cổ điển mà còn các mô hình dựa trên học tập sâu tiên tiến hơn.

17.1 Tổng quan về hệ thống Recommender

Trong thập kỷ qua, Internet đã phát triển thành một nền tảng cho các dịch vụ trực tuyến quy mô lớn, điều này đã thay đổi sâu sắc cách chúng ta giao tiếp, đọc tin tức, mua sản phẩm và xem phim. Trong khi đó, số lượng mặt hàng chưa từng có (chúng tôi sử dụng thuật ngữ *item* để chỉ phim, tin tức, sách và sản phẩm.) được cung cấp trực tuyến đòi hỏi một hệ thống có thể giúp chúng tôi khám phá các mặt hàng mà chúng tôi ưa thích. Do đó, hệ thống giới thiệu là các công cụ lọc thông tin mạnh mẽ có thể tạo điều kiện thuận lợi cho các dịch vụ được cá nhân hóa và cung cấp trải nghiệm phù hợp cho từng người dùng Nói tóm lại, các hệ thống giới thiệu đóng một vai trò quan trọng trong việc sử dụng sự giàu có của dữ liệu có sẵn để làm cho các lựa chọn có thể quản lý được. Ngày nay, các hệ thống giới thiệu là cốt lõi của một số nhà cung cấp dịch vụ trực tuyến như Amazon, Netflix và YouTube. Nhớ lại ví dụ về sách học sâu được Amazon đề xuất trong Fig. 2.3.3. Lợi ích của việc sử dụng các hệ thống giới thiệu là hai lần: Một mặt, nó chủ yếu có thể làm giảm nỗ lực của người dùng trong việc tìm kiếm các mục và giảm bớt vấn đề quá tải thông tin. Mặt khác, nó có thể thêm giá trị kinh doanh cho các nhà cung cấp dịch vụ trực tuyến và là một nguồn doanh thu quan trọng. Chương này sẽ giới thiệu các khái niệm cơ bản, mô hình cổ điển và những tiến bộ gần đây với học sâu trong lĩnh vực hệ thống giới thiệu, cùng với các ví dụ được thực hiện.

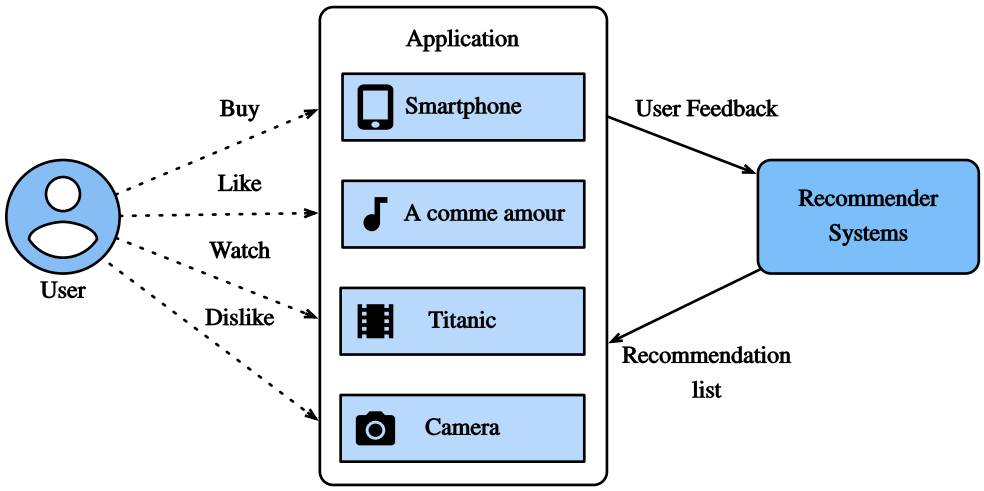


Fig. 17.1.1: Illustration of the Recommendation Process

17.1.1 Lọc hợp tác

Chúng tôi bắt đầu hành trình với khái niệm quan trọng trong hệ thống giới thiệu — lọc hợp tác (CF), lần đầu tiên được đặt ra bởi hệ thống Tapestry (Goldberg et al., 1992), đề cập đến "mọi người cộng tác để giúp đỡ lẫn nhau thực hiện quá trình lọc để xử lý một lượng lớn email và tin nhắn được đăng lên nhóm tin tức". Thuật ngữ này đã được làm giàu với nhiều giác quan hơn. Theo nghĩa rộng, đó là quá trình lọc thông tin hoặc mẫu bằng cách sử dụng các kỹ thuật liên quan đến sự cộng tác giữa nhiều người dùng, đại lý và nguồn dữ liệu. CF có nhiều hình thức và nhiều phương pháp CF được đề xuất kể từ khi ra đời.

Nhìn chung, kỹ thuật CF có thể được phân loại into: memory-based CF, model-based CF, and their hybrid (Su & Khoshgoftaar, 2009). Các kỹ thuật CF dựa trên bộ nhớ đại diện là CF dựa trên lân cận gần nhất như CF dựa trên người dùng và CF (Sarwar et al., 2001) dựa trên mặt hàng. Các mô hình yếu tố tiềm ẩn như factorization ma trận là ví dụ về CF dựa trên mô hình. CF dựa trên bộ nhớ có những hạn chế trong việc xử lý dữ liệu thưa thớt và quy mô lớn vì nó tính toán các giá trị tương đồng dựa trên các mục phổ biến. Các phương pháp dựa trên mô hình trở nên phổ biến hơn với khả năng tốt hơn trong việc đối phó với sự thưa thớt và khả năng mở rộng. Nhiều phương pháp tiếp cận CF dựa trên mô hình có thể được mở rộng với các mạng thần kinh, dẫn đến các mô hình linh hoạt và có thể mở rộng hơn với khả năng tăng tốc tính toán trong deep learning (Zhang et al., 2019). Nói chung, CF chỉ sử dụng dữ liệu tương tác mục người dùng để đưa ra dự đoán và khuyến nghị. Bên cạnh CF, các hệ thống giới thiệu dựa trên nội dung và bối cảnh cũng hữu ích trong việc kết hợp các mô tả nội dung của các mục/người dùng và các tín hiệu theo ngữ cảnh như dấu thời gian và địa điểm. Rõ ràng, chúng ta có thể cần phải điều chỉnh các loại/cấu trúc mô hình khi dữ liệu đầu vào khác nhau có sẵn.

17.1.2 Phản hồi rõ ràng và phản hồi tiềm ẩn

Để tìm hiểu sở thích của người dùng, hệ thống sẽ thu thập phản hồi từ họ. Phản hồi có thể là rõ ràng hoặc ngầm (Hu et al., 2008). Ví dụ, IMDB[207] thu thập xếp hạng sao khác nhau, từ một đến mười sao cho phim. YouTube cung cấp các nút thu nhỏ và ngón tay cái để người dùng hiển thị tùy chọn của họ. Rõ ràng là thu thập phản hồi rõ ràng đòi hỏi người dùng phải chủ động chỉ ra lợi ích của họ. Tuy nhiên, phản hồi rõ ràng không phải lúc nào cũng có sẵn vì nhiều người dùng có thể miễn cưỡng đánh giá sản phẩm. Tương đối nói, phản hồi ngầm thường có sẵn vì nó chủ yếu liên quan đến mô hình hóa hành vi ngầm như nhấp chuột của người dùng. Do đó, nhiều hệ thống giới thiệu tập trung vào phản hồi ngầm mà gián tiếp phản ánh ý kiến của người dùng thông qua việc quan sát hành vi của người dùng. Có nhiều hình thức phản hồi tiềm ẩn đa dạng bao gồm lịch

[207] https://www.imdb.com/

sử mua hàng, lịch sử duyệt web, đồng hồ và thậm chí cả chuyển động chuột. Ví dụ, một người dùng đã mua nhiều cuốn sách của cùng một tác giả có thể thích tác giả đó. Lưu ý rằng phản hồi ngầm vốn là ồn ào. Chúng tôi chỉ có thể * đoán * sở thích và động cơ thực sự của họ. Một người dùng đã xem một bộ phim không nhất thiết phải chỉ ra một cái nhìn tích cực của bộ phim đó.

17.1.3 Tác vụ đề xuất

Một số nhiệm vụ đề xuất đã được điều tra trong những thập kỷ qua. Dựa trên tên miền của các ứng dụng, có giới thiệu phim, khuyến nghị tin tức, đề xuất điểm quan tâm (Ye et al., 2011) và vân vân. Cũng có thể phân biệt các nhiệm vụ dựa trên các loại phản hồi và dữ liệu đầu vào, ví dụ, nhiệm vụ dự đoán xếp hạng nhằm dự đoán xếp hạng rõ ràng. Đề xuất Top-n (xếp hạng mục) xếp hạng tất cả các mục cho mỗi người dùng cá nhân dựa trên phản hồi ngầm. Nếu thông tin tem thời gian cũng được bao gồm, chúng tôi có thể xây dựng đề xuất nhận thức trình tự (Quadrana et al., 2018). Một nhiệm vụ phổ biến khác được gọi là dự đoán tỷ lệ nhấp, cũng dựa trên phản hồi ngầm, nhưng các tính năng phân loại khác nhau có thể được sử dụng. Đề xuất cho người dùng mới và đề xuất các mặt hàng mới cho người dùng hiện có được gọi là khuyến nghị khởi động lạnh (Schein et al., 2002).

17.1.4 Tóm tắt

- Hệ thống giới thiệu rất quan trọng đối với người dùng cá nhân và ngành công nghiệp. Lọc hợp tác là một khái niệm chính trong đề xuất.
- Có hai loại phản hồi: phản hồi ngầm và phản hồi rõ ràng. Một số nhiệm vụ đề xuất đã được khám phá trong thập kỷ qua.

17.1.5 Bài tập

1. Bạn có thể giải thích cách các hệ thống giới thiệu ảnh hưởng đến cuộc sống hàng ngày của bạn?
2. Những nhiệm vụ đề xuất thú vị nào bạn nghĩ có thể được điều tra?

Discussions[208]

[208] https://discuss.d2l.ai/t/398

18 | Mạng đối thủ thế hệ

18.1 Mạng đối thủ thế hệ

Trong suốt hầu hết cuốn sách này, chúng tôi đã nói về cách đưa ra dự đoán. Ở dạng nào đó hay hình thức khác, chúng tôi đã sử dụng các mạng thần kinh sâu đã học được ánh xạ từ các ví dụ dữ liệu đến nhãn. Loại hình học tập này được gọi là học phân biệt đối xử, như trong, chúng tôi muốn có thể phân biệt đối xử giữa những bức ảnh mèo và hình ảnh của chó. Phân loại và bộ hồi quy đều là ví dụ về việc học phân biệt đối xử. Và mạng lưới thần kinh được đào tạo bởi sự lan truyền ngược đã upended tất cả mọi thứ chúng tôi nghĩ rằng chúng tôi biết về việc học phân biệt đối xử trên các tập dữ liệu phức tạp lớn. Độ chính xác phân loại trên hình ảnh có độ phân giải cao đã đi từ vô dụng đến cấp độ con người (với một số cảnh báo) chỉ trong 5-6 năm. Chúng tôi sẽ dành cho bạn một trò chơi khác về tất cả các nhiệm vụ phân biệt đối xử khác, nơi các mạng thần kinh sâu làm tốt đáng kinh ngạc.

Nhưng có nhiều hơn để học máy hơn là chỉ giải quyết các nhiệm vụ phân biệt đối xử. Ví dụ, với một tập dữ liệu lớn, không có bất kỳ nhãn nào, chúng ta có thể muốn tìm hiểu một mô hình nắm bắt chính xác các đặc điểm của dữ liệu này. Với một mô hình như vậy, chúng tôi có thể lấy mẫu các ví dụ dữ liệu tổng hợp giống với việc phân phối dữ liệu đào tạo. Ví dụ, với một lượng lớn các bức ảnh khuôn mặt, chúng ta có thể muốn có thể tạo ra một hình ảnh thực tế mới trông giống như nó có thể hợp lý đến từ cùng một tập dữ liệu. Loại học này được gọi là mô hình thế hệ.

Cho đến gần đây, chúng tôi không có phương pháp nào có thể tổng hợp các hình ảnh photorealistic mới lạ. Nhưng sự thành công của các mạng thần kinh sâu cho việc học phân biệt đối xử đã mở ra những khả năng mới. Một xu hướng lớn trong ba năm qua là việc áp dụng các mạng lưới sâu phân biệt đối xử để vượt qua những thách thức trong các vấn đề mà chúng ta thường không nghĩ là vấn đề học tập được giám sát. Các mô hình ngôn ngữ mạng thần kinh tái phát là một ví dụ về việc sử dụng một mạng phân biệt đối xử (được đào tạo để dự đoán ký tự tiếp theo) mà một khi được đào tạo có thể hoạt động như một mô hình thế hệ.

Năm 2014, một bài báo đột phá đã giới thiệu mạng đối thủ thế hệ (GAN) (Goodfellow et al., 2014), một cách thông minh mới để tận dụng sức mạnh của các mô hình phân biệt đối xử để có được các mô hình thế hệ tốt. Tại trái tim của họ, GAN dựa vào ý tưởng rằng một trình tạo dữ liệu là tốt nếu chúng ta không thể nói dữ liệu giả mạo ngoài dữ liệu thực. Trong thống kê, đây được gọi là thử nghiệm hai mẫu - một bài kiểm tra để trả lời câu hỏi liệu các bộ dữ liệu $X = \{x_1, \ldots, x_n\}$ và $X' = \{x'_1, \ldots, x'_n\}$ có được rút ra từ cùng một phân phối hay không. Sự khác biệt chính giữa hầu hết các giấy tờ thống kê và GAN là sau này sử dụng ý tưởng này một cách mang tính xây dựng. Nói cách khác, thay vì chỉ đào tạo một mô hình để nói "này, hai bộ dữ liệu này trông không giống như chúng đến từ cùng một phân phối", họ sử dụng two-sample test[209] để cung cấp tín hiệu đào tạo cho một mô hình thế hệ. Điều này cho phép chúng tôi cải thiện trình tạo dữ liệu cho đến khi nó tạo ra một cái gì đó giống với dữ liệu thực. Ít nhất, nó cần phải đánh lừa phân loại. Ngay cả khi phân loại của chúng tôi là một nhà nước của mạng thần kinh sâu sắc nghệ thuật.

[209] https://en.wikipedia.org/wiki/Two-sample_hypothesis_testing

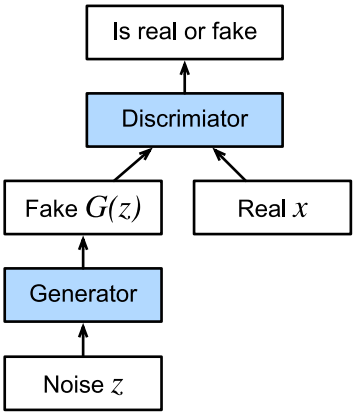


Fig. 18.1.1: Generative Adversarial Networks

Kiến trúc GAN được minh họa vào năm Fig. 18.1.1. Như bạn có thể thấy, có hai phần trong kiến trúc GAN - trước hết, chúng ta cần một thiết bị (giả sử, một mạng sâu nhưng nó thực sự có thể là bất cứ điều gì, chẳng hạn như một công cụ kết xuất trò chơi) có khả năng có thể tạo ra dữ liệu trông giống như thực tế. Nếu chúng ta đang đối phó với hình ảnh, điều này cần phải tạo ra hình ảnh. Nếu chúng ta đang đối phó với lời nói, nó cần phải tạo ra các chuỗi âm thanh, v.v. Chúng tôi gọi đây là mạng máy phát điện. Thành phần thứ hai là mạng phân biệt đối xử. Nó cố gắng phân biệt dữ liệu giả và thực với nhau. Cả hai mạng đang cạnh tranh với nhau. Mạng máy phát cố gắng đánh lừa mạng phân biệt đối xử. Tại thời điểm đó, mạng phân biệt đối xử thích ứng với dữ liệu giả mới. Thông tin này, lần lượt được sử dụng để cải thiện mạng máy phát điện, v.v.

Phân biệt đối xử là một phân loại nhị phân để phân biệt nếu đầu vào x là thực (từ dữ liệu thực) hay giả (từ máy phát điện). Thông thường, phân biệt đối xử đầu ra một dự đoán vô hướng $o \in \mathbb{R}$ cho đầu vào $\mathbf{x}$, chẳng hạn như sử dụng một lớp dày đặc với kích thước ẩn 1, và sau đó áp dụng hàm sigmoid để có được xác suất dự đoán $D(\mathbf{x}) = 1/(1 + e^{-o})$. Giả sử nhãn y cho dữ liệu thật là 1 và 0 cho dữ liệu giả mạo. Chúng tôi đào tạo người phân biệt đối xử để giảm thiểu tổn thất chéo entropy, * tức là *,

$$\min_D \{-y \log D(\mathbf{x}) - (1 - y) \log(1 - D(\mathbf{x}))\}, \tag{18.1.1}$$

Đối với máy phát điện, đầu tiên nó rút ra một số tham số $\mathbf{z} \in \mathbb{R}^d$ từ một nguồn ngẫu nhiên, *ví dụ*, một phân phối bình thường $\mathbf{z} \sim \mathcal{N}(0, 1)$. Chúng ta thường gọi $\mathbf{z}$ là biến tiềm ẩn. Sau đó, nó áp dụng một chức năng để tạo ra $\mathbf{x}' = G(\mathbf{z})$. Mục tiêu của trình tạo là đánh lừa người phân biệt đối xử để phân loại $\mathbf{x}' = G(\mathbf{z})$ là dữ liệu thật, * tức là*, chúng tôi muốn $D(G(\mathbf{z})) \approx 1$. Nói cách khác, đối với một phân biệt đối xử nhất định D, chúng tôi cập nhật các thông số của máy phát G để tối đa hóa tổn thất ngẫu nhiên chéo khi $y = 0$, * i.e.*,

$$\max_G \{-(1 - y) \log(1 - D(G(\mathbf{z})))\} = \max_G \{-\log(1 - D(G(\mathbf{z})))\}. \tag{18.1.2}$$

Nếu máy phát điện thực hiện một công việc hoàn hảo, thì $D(\mathbf{x}') \approx 1$ do đó, tổn thất trên gần 0, dẫn đến độ dốc quá nhỏ để đạt được tiến bộ tốt cho người phân biệt đối xử. Vì vậy, thông thường chúng ta giảm thiểu những tổn thất sau:

$$\min_G \{-y \log(D(G(\mathbf{z})))\} = \min_G \{-\log(D(G(\mathbf{z})))\}, \tag{18.1.3}$$

mà chỉ là thức ăn $\mathbf{x}' = G(\mathbf{z})$ vào phân biệt đối xử nhưng đưa ra nhãn $y = 1$.

Tóm lại, D và G đang chơi một trò chơi "minimax" với chức năng mục tiêu toàn diện:

$$min_D max_G \{-E_{x \sim \text{Data}} log D(\mathbf{x}) - E_{z \sim \text{Noise}} log(1 - D(G(\mathbf{z})))\}. \tag{18.1.4}$$

Nhiều ứng dụng GAN nằm trong bối cảnh hình ảnh. Là một mục đích trình diễn, chúng ta sẽ tự nội dung với việc lắp một bản phân phối đơn giản hơn nhiều trước tiên. Chúng tôi sẽ minh họa những gì xảy ra nếu chúng

ta sử dụng GAN để xây dựng ước tính không hiệu quả nhất thế giới của các tham số cho một Gaussian. Hãy để chúng tôi bắt đầu.

```
%matplotlib inline
import torch
from torch import nn
from d2l import torch as d2l
```

18.1.1 Tạo ra một số dữ liệu "thực"

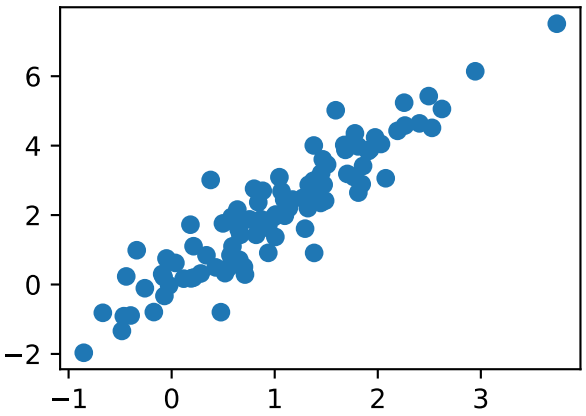
Vì đây sẽ là ví dụ đáng sợ nhất thế giới, chúng tôi chỉ đơn giản là tạo ra dữ liệu được rút ra từ Gaussian.

```
X = torch.normal(0.0, 1, (1000, 2))
A = torch.tensor([[1, 2], [-0.1, 0.5]])
b = torch.tensor([1, 2])
data = torch.matmul(X, A) + b
```

Hãy để chúng tôi xem những gì chúng tôi có. Đây phải là một Gaussian dịch chuyển theo một cách khá tùy ý với trung bình b và ma trận hiệp phương sai $A^T A$.

```
d2l.set_figsize()
d2l.plt.scatter(data[:100, (0)].detach().numpy(), data[:100, (1)].detach().
↪numpy());
print(f'The covariance matrix is\n{torch.matmul(A.T, A)}')
```

```
The covariance matrix is
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```




```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

18.1.2 Máy phát điện

Mạng máy phát của chúng tôi sẽ là mạng đơn giản nhất có thể - một mô hình tuyến tính lớp duy nhất. Điều này là vì chúng tôi sẽ lái mạng tuyến tính đó với trình tạo dữ liệu Gaussian. Do đó, nó theo nghĩa đen chỉ cần tìm hiểu các thông số để giả mạo mọi thứ một cách hoàn hảo.

```
net_G = nn.Sequential(nn.Linear(2, 2))
```

18.1.3 Phân biệt đối xử

Đối với người phân biệt đối xử, chúng ta sẽ phân biệt hơn một chút: chúng ta sẽ sử dụng MLP với 3 lớp để làm cho mọi thứ thú vị hơn một chút.

```
net_D = nn.Sequential(
    nn.Linear(2, 5), nn.Tanh(),
    nn.Linear(5, 3), nn.Tanh(),
    nn.Linear(3, 1))
```

18.1.4 Đào tạo

Đầu tiên chúng ta định nghĩa một hàm để cập nhật sự phân biệt đối xử.

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator."""
    batch_size = X.shape[0]
    ones = torch.ones((batch_size,), device=X.device)
    zeros = torch.zeros((batch_size,), device=X.device)
    trainer_D.zero_grad()
    real_Y = net_D(X)
    fake_X = net_G(Z)
    # Do not need to compute gradient for `net_G`, detach it from
    # computing gradients.
    fake_Y = net_D(fake_X.detach())
    loss_D = (loss(real_Y, ones.reshape(real_Y.shape)) +
              loss(fake_Y, zeros.reshape(fake_Y.shape))) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

Máy phát điện được cập nhật tương tự. Ở đây chúng tôi sử dụng lại tổn thất chéo entropy nhưng thay đổi nhãn của dữ liệu giả từ 0 thành 1.

```
#@save
def update_G(Z, net_D, net_G, loss, trainer_G):
    """Update generator."""
    batch_size = Z.shape[0]
    ones = torch.ones((batch_size,), device=Z.device)
    trainer_G.zero_grad()
    # We could reuse `fake_X` from `update_D` to save computation
    fake_X = net_G(Z)
```

(continues on next page)

(continued from previous page)

```
    # Recomputing `fake_Y` is needed since `net_D` is changed
    fake_Y = net_D(fake_X)
    loss_G = loss(fake_Y, ones.reshape(fake_Y.shape))
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

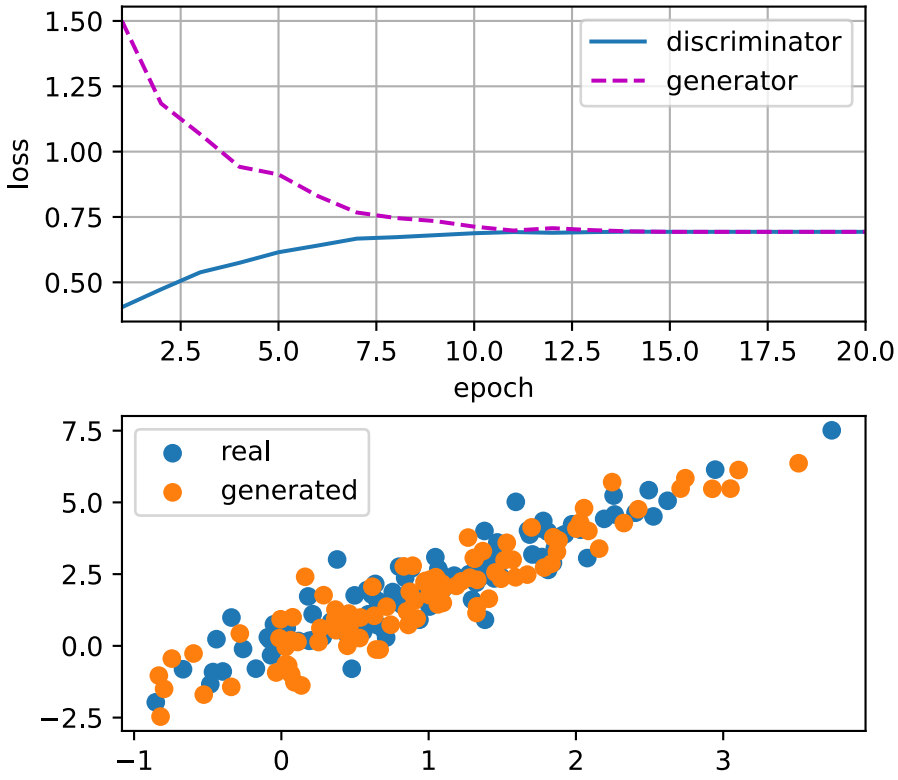
Cả phân biệt đối xử và máy phát điện đều thực hiện hồi quy hậu cần nhị phân với tổn thất chéo entropy. Chúng tôi sử dụng Adam để làm mịn quá trình đào tạo. Trong mỗi lần lặp lại, lần đầu tiên chúng ta cập nhật trình phân biệt đối xử và sau đó là trình tạo. Chúng tôi hình dung cả tổn thất và các ví dụ được tạo ra.

```
def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D)
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(num_epochs):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for (X,) in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim))
            metric.add(update_D(X, Z, net_D, net_G, loss, trainer_D),
                       update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Visualize generated examples
        Z = torch.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).detach().numpy()
        animator.axes[1].cla()
        animator.axes[1].scatter(data[:, 0], data[:, 1])
        animator.axes[1].scatter(fake_X[:, 0], fake_X[:, 1])
        animator.axes[1].legend(['real', 'generated'])
        # Show the losses
        loss_D, loss_G = metric[0]/metric[2], metric[1]/metric[2]
        animator.add(epoch + 1, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec')
```

Bây giờ chúng tôi chỉ định các siêu tham số để phù hợp với phân phối Gaussian.

```
lr_D, lr_G, latent_dim, num_epochs = 0.05, 0.005, 2, 20
train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G,
      latent_dim, data[:100].detach().numpy())
```

```
loss_D 0.693, loss_G 0.693, 1609.6 examples/sec
```



18.1.5 Tóm tắt

- Các mạng đối thủ thế hệ (GAN) bao gồm hai mạng sâu, máy phát điện và người phân biệt đối xử.
- Trình tạo tạo ra hình ảnh càng gần với hình ảnh thật càng tốt để đánh lừa sự phân biệt đối xử, thông qua việc tối đa hóa tổn thất chéo entropy, * tức là *, $\max \log(D(\mathbf{x}'))$.
- Người phân biệt đối xử cố gắng phân biệt các hình ảnh được tạo ra với hình ảnh thật, thông qua việc giảm thiểu tổn thất chéo entropy, * tức là *, $\min -y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))$.

18.1.6 Bài tập

- Liệu một trạng thái cân bằng tồn tại nơi máy phát chiến thắng, * tức là * người phân biệt đối xử cuối cùng không thể phân biệt hai phân phối trên các mẫu hữu hạn?

Discussions[210]

[210] https://discuss.d2l.ai/t/1082

18.2 Mạng đối thủ tạo phức tạp sâu

Trong Section 18.1, chúng tôi đã giới thiệu những ý tưởng cơ bản đằng sau cách GAN hoạt động. Chúng tôi đã chỉ ra rằng chúng có thể vẽ các mẫu từ một số phân phối đơn giản, dễ lấy mẫu, như phân phối thống nhất hoặc bình thường, và biến đổi chúng thành các mẫu dường như phù hợp với sự phân bố của một số tập dữ liệu. Và trong khi ví dụ của chúng tôi về việc phù hợp với phân phối 2D Gaussian đã vượt qua điểm, nó không đặc biệt thú vị.

Trong phần này, chúng tôi sẽ chứng minh cách bạn có thể sử dụng GAN để tạo ra hình ảnh thực tế. Chúng tôi sẽ dựa trên các mô hình của chúng tôi trên GAN phức tạp sâu (DCGAN) được giới thiệu trong (Radford et al., 2015). Chúng tôi sẽ mượn kiến trúc phức tạp đã chứng minh thành công cho các vấn đề thị giác máy tính phân biệt đối xử và cho thấy cách thông qua GAN, chúng có thể được tận dụng để tạo ra hình ảnh thực tế.

```
import warnings
import torch
import torchvision
from torch import nn
from d2l import torch as d2l
```

18.2.1 Các Pokemon Dataset

Tập dữ liệu chúng tôi sẽ sử dụng là một bộ sưu tập các sprites Pokemon thu được từ pokemondb[211]. Đầu tiên tải xuống, trích xuất và tải tập dữ liệu này.

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = torchvision.datasets.ImageFolder(data_dir)
```

```
Downloading ../data/pokemon.zip from http://d2l-data.s3-accelerate.amazonaws.
↪com/pokemon.zip...
```

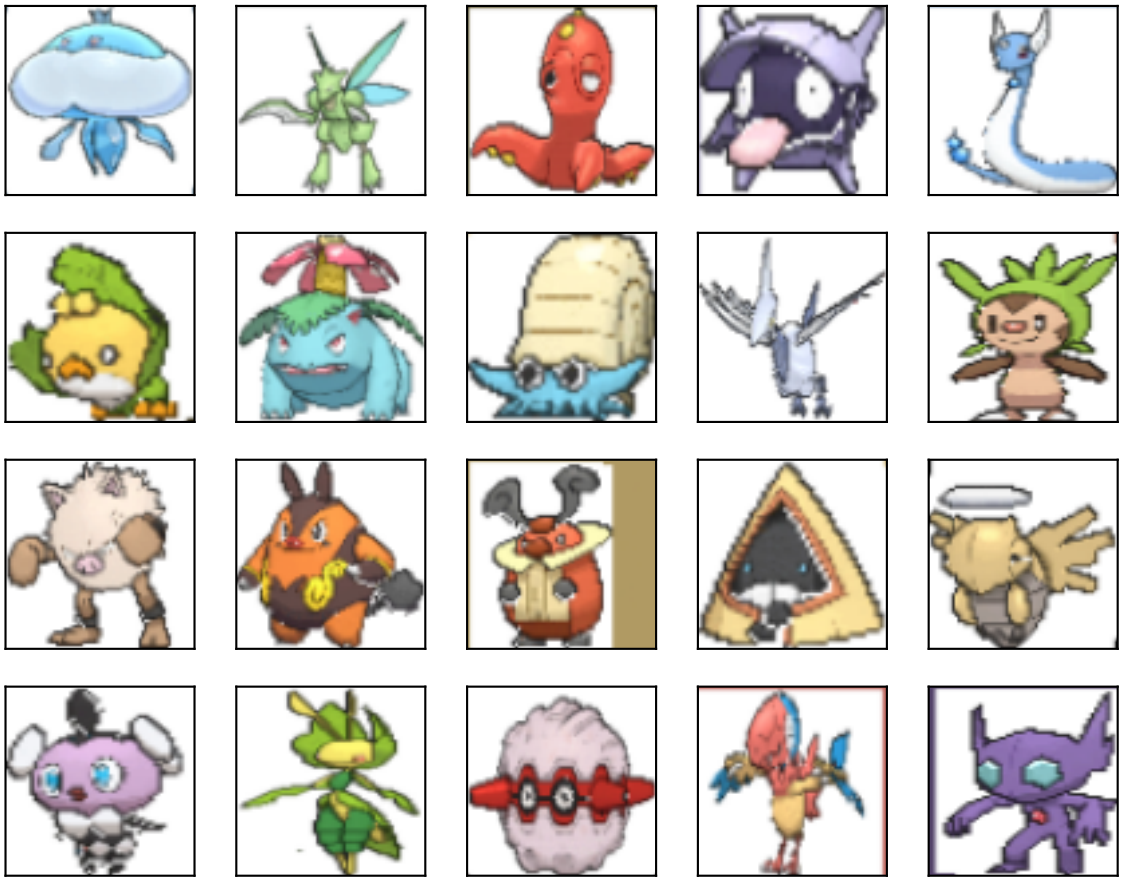
Chúng tôi thay đổi kích thước mỗi hình ảnh thành 64×64. Việc chuyển đổi `ToTensor` sẽ chiếu giá trị pixel thành $[0, 1]$, trong khi trình tạo của chúng tôi sẽ sử dụng hàm tánh để có được đầu ra trong $[-1, 1]$. Do đó, chúng tôi bình thường hóa dữ liệu với 0.5 trung bình và độ lệch chuẩn 0.5 để phù hợp với phạm vi giá trị.

```
batch_size = 256
transformer = torchvision.transforms.Compose([
    torchvision.transforms.Resize((64, 64)),
    torchvision.transforms.ToTensor(),
    torchvision.transforms.Normalize(0.5, 0.5)
])
pokemon.transform = transformer
data_iter = torch.utils.data.DataLoader(
    pokemon, batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

Hãy để chúng tôi hình dung 20 hình ảnh đầu tiên.

[211] https://pokemondb.net/sprites

```
warnings.filterwarnings('ignore')
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[0:20,:,:,:].permute(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```



18.2.2 Các máy phát điện

Máy phát điện cần ánh xạ biến nhiễu $\mathbf{z} \in \mathbb{R}^d$, một vector chiều dài-$d$, đến một hình ảnh RGB với chiều rộng và chiều cao là 64×64. Trong Section 14.11, chúng tôi đã giới thiệu mạng phức tạp hoàn toàn sử dụng lớp ghép chuyển tiếp (tham khảo Section 14.10) để phóng to kích thước đầu vào. Khối cơ bản của máy phát điện chứa một lớp phức tạp chuyển tiếp theo là chuẩn hóa hàng loạt và kích hoạt ReLU.

```
class G_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.ConvTranspose2d(in_channels, out_channels,
                                               kernel_size, strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.ReLU()
```

(continues on next page)

```
    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

Mặc định, lớp ghép chuyển tiếp sử dụng hạt nhân $k_h = k_w = 4$, một bước tiến $s_h = s_w = 2$ và đệm $p_h = p_w = 1$. Với hình dạng đầu vào là $n_h^{'} \times n_w^{'} = 16 \times 16$, khối máy phát sẽ tăng gấp đôi chiều rộng và chiều cao của đầu vào.

$$\begin{aligned} n_h^{'} \times n_w^{'} &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h] \times [(n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w] \\ &= [(k_h + s_h(n_h - 1) - 2p_h] \times [(k_w + s_w(n_w - 1) - 2p_w] \\ &= [(4 + 2 \times (16 - 1) - 2 \times 1] \times [(4 + 2 \times (16 - 1) - 2 \times 1] \\ &= 32 \times 32. \end{aligned} \tag{18.2.1}$$

```
x = torch.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk(x).shape
```

```
torch.Size([2, 20, 32, 32])
```

Nếu thay đổi lớp phức tạp chuyển tiếp thành hạt nhân 4×4, 1×1 sải bước và không đệm. Với kích thước đầu vào là 1×1, đầu ra sẽ có chiều rộng và chiều cao của nó tăng 3 lần lượt.

```
x = torch.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk(x).shape
```

```
torch.Size([2, 20, 4, 4])
```

Máy phát điện bao gồm bốn khối cơ bản giúp tăng cả chiều rộng và chiều cao đầu vào từ 1 lên 32. Đồng thời, đầu tiên nó chiếu biến tiềm ẩn thành 64×8 kênh, sau đó giảm một nửa các kênh mỗi lần. Cuối cùng, một lớp covolution được chuyển đổi được sử dụng để tạo ra đầu ra. Nó tiếp tục tăng gấp đôi chiều rộng và chiều cao để phù hợp với hình dạng 64×64 mong muốn và giảm kích thước kênh xuống 3. Hàm kích hoạt tanh được áp dụng cho các giá trị đầu ra dự án vào phạm vi $(-1, 1)$.

```
n_G = 64
net_G = nn.Sequential(
    G_block(in_channels=100, out_channels=n_G*8,
            strides=1, padding=0),                  # Output: (64 * 8, 4, 4)
    G_block(in_channels=n_G*8, out_channels=n_G*4), # Output: (64 * 4, 8, 8)
    G_block(in_channels=n_G*4, out_channels=n_G*2), # Output: (64 * 2, 16, 16)
    G_block(in_channels=n_G*2, out_channels=n_G),   # Output: (64, 32, 32)
    nn.ConvTranspose2d(in_channels=n_G, out_channels=3,
                       kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh())  # Output: (3, 64, 64)
```

Tạo ra một biến tiềm ẩn 100 chiều để xác minh hình dạng đầu ra của máy phát điện.

```
x = torch.zeros((1, 100, 1, 1))
net_G(x).shape
```

```
torch.Size([1, 3, 64, 64])
```

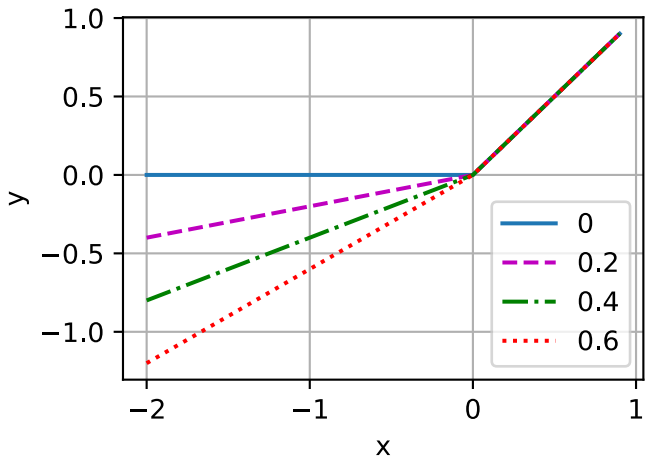
18.2.3 Phân biệt đối xử

Người phân biệt đối xử là một mạng mạng phức tạp bình thường ngoại trừ việc nó sử dụng ReLU bị rò rỉ làm chức năng kích hoạt của nó. Đưa ra $\alpha \in [0, 1]$, định nghĩa của nó là

$$\text{leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{otherwise} \end{cases}. \tag{18.2.2}$$

Như có thể thấy, nó là bình thường ReLU nếu $\alpha = 0$, và một chức năng nhận dạng nếu $\alpha = 1$. Đối với $\alpha \in (0, 1)$, ReLU bị rò rỉ là một hàm phi tuyến cung cấp đầu ra không phải bằng không cho đầu vào âm. Nó nhằm mục đích khắc phục vấn đề ReLU đang chết mà một tế bào thần kinh luôn có thể tạo ra một giá trị âm và do đó không thể thực hiện bất kỳ tiến bộ nào vì gradient của ReLU là 0.

```
alphas = [0, .2, .4, .6, .8, 1]
x = torch.arange(-2, 1, 0.1)
Y = [nn.LeakyReLU(alpha)(x).detach().numpy() for alpha in alphas]
d2l.plot(x.detach().numpy(), Y, 'x', 'y', alphas)
```



Khối cơ bản của phân biệt đối xử là một lớp phức tạp tiếp theo là một lớp chuẩn hóa hàng loạt và kích hoạt ReLU bị rò rỉ. Các siêu tham số của lớp covolution tương tự như lớp covolution transpose trong khối máy phát.

```
class D_block(nn.Module):
    def __init__(self, out_channels, in_channels=3, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2d(in_channels, out_channels, kernel_size,
                                strides, padding, bias=False)
        self.batch_norm = nn.BatchNorm2d(out_channels)
        self.activation = nn.LeakyReLU(alpha, inplace=True)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

Một khối cơ bản với cài đặt mặc định sẽ giảm một nửa chiều rộng và chiều cao của các đầu vào, như chúng ta đã chứng minh trong Section 7.3. Ví dụ, cho một hình dạng đầu vào $n_h = n_w = 16$, với một hình dạng hạt nhân $k_h = k_w = 4$, một hình sải chân $s_h = s_w = 2$, và một hình dạng đệm $p_h = p_w = 1$, hình dạng đầu ra sẽ là:

$$\begin{aligned} n_h^{'} \times n_w^{'} &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\ &= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\ &= 8 \times 8. \end{aligned} \tag{18.2.3}$$

```
x = torch.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk(x).shape
```

```
torch.Size([2, 20, 8, 8])
```

Người phân biệt đối xử là một tấm gương của máy phát điện.

```
n_D = 64
net_D = nn.Sequential(
    D_block(n_D),  # Output: (64, 32, 32)
    D_block(in_channels=n_D, out_channels=n_D*2),  # Output: (64 * 2, 16, 16)
    D_block(in_channels=n_D*2, out_channels=n_D*4),  # Output: (64 * 4, 8, 8)
    D_block(in_channels=n_D*4, out_channels=n_D*8),  # Output: (64 * 8, 4, 4)
    nn.Conv2d(in_channels=n_D*8, out_channels=1,
              kernel_size=4, bias=False))  # Output: (1, 1, 1)
```

Nó sử dụng một lớp phức tạp với kênh đầu ra 1 làm lớp cuối cùng để có được một giá trị dự đoán duy nhất.

```
x = torch.zeros((1, 3, 64, 64))
net_D(x).shape
```

```
torch.Size([1, 1, 1, 1])
```

18.2.4 Đào tạo

So với GAN cơ bản trong Section 18.1, chúng tôi sử dụng cùng một tốc độ học tập cho cả máy phát điện và phân biệt đối xử vì chúng tương tự nhau. Ngoài ra, chúng tôi thay đổi β_1 trong Adam (Section 12.10) từ 0.9 thành 0.5. Nó làm giảm độ mịn của động lượng, trung bình động có trọng số theo cấp số nhân của gradient trong quá khứ, để chăm sóc các gradient thay đổi nhanh chóng vì máy phát điện và người phân biệt đối xử chiến đấu với nhau. Bên cạnh đó, tiếng ồn tạo ngẫu nhiên `Z`, là một tensor 4-D và chúng tôi đang sử dụng GPU để tăng tốc tính toán.

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = nn.BCEWithLogitsLoss(reduction='sum')
    for w in net_D.parameters():
        nn.init.normal_(w, 0, 0.02)
    for w in net_G.parameters():
        nn.init.normal_(w, 0, 0.02)
```

(continues on next page)

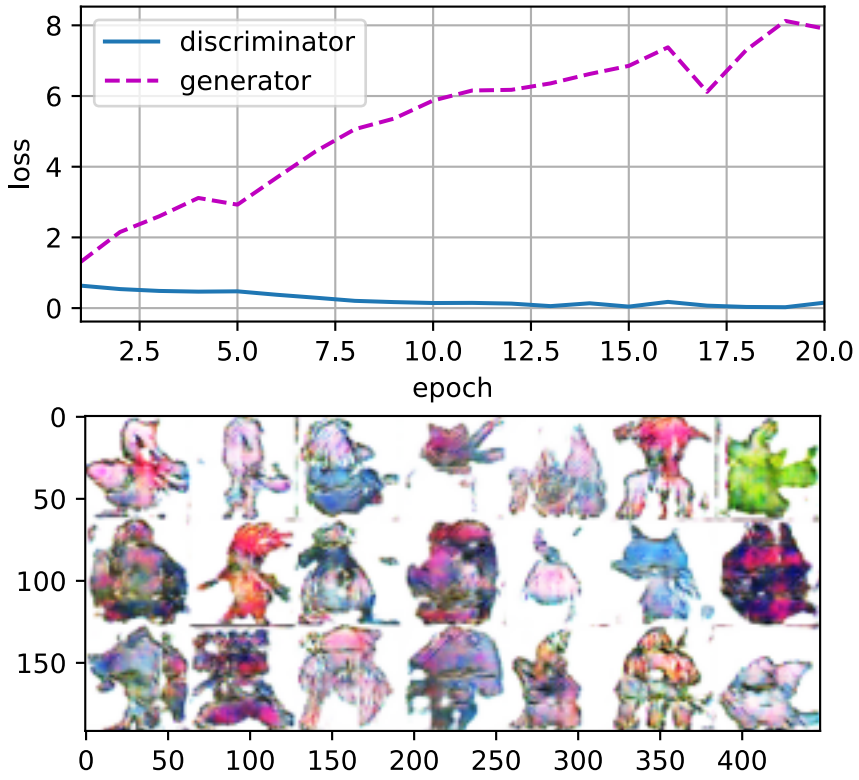
(continued from previous page)

```
    net_D, net_G = net_D.to(device), net_G.to(device)
    trainer_hp = {'lr': lr, 'betas': [0.5,0.999]}
    trainer_D = torch.optim.Adam(net_D.parameters(), **trainer_hp)
    trainer_G = torch.optim.Adam(net_G.parameters(), **trainer_hp)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], nrows=2, figsize=(5, 5),
                            legend=['discriminator', 'generator'])
    animator.fig.subplots_adjust(hspace=0.3)
    for epoch in range(1, num_epochs + 1):
        # Train one epoch
        timer = d2l.Timer()
        metric = d2l.Accumulator(3)  # loss_D, loss_G, num_examples
        for X, _ in data_iter:
            batch_size = X.shape[0]
            Z = torch.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
            X, Z = X.to(device), Z.to(device)
            metric.add(d2l.update_D(X, Z, net_D, net_G, loss, trainer_D),
                       d2l.update_G(Z, net_D, net_G, loss, trainer_G),
                       batch_size)
        # Show generated examples
        Z = torch.normal(0, 1, size=(21, latent_dim, 1, 1), device=device)
        # Normalize the synthetic data to N(0, 1)
        fake_x = net_G(Z).permute(0, 2, 3, 1) / 2 + 0.5
        imgs = torch.cat(
            [torch.cat([
                fake_x[i * 7 + j].cpu().detach() for j in range(7)], dim=1)
             for i in range(len(fake_x)//7)], dim=0)
        animator.axes[1].cla()
        animator.axes[1].imshow(imgs)
        # Show the losses
        loss_D, loss_G = metric[0] / metric[2], metric[1] / metric[2]
        animator.add(epoch, (loss_D, loss_G))
    print(f'loss_D {loss_D:.3f}, loss_G {loss_G:.3f}, '
          f'{metric[2] / timer.stop():.1f} examples/sec on {str(device)}')
```

Chúng tôi đào tạo mô hình với một số lượng nhỏ các kỷ nguyên chỉ để trình diễn. Để có hiệu suất tốt hơn, biến `num_epochs` có thể được đặt thành một số lớn hơn.

```
latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)
```

```
loss_D 0.154, loss_G 7.906, 1062.5 examples/sec on cuda:0
```



18.2.5 Tóm tắt

- Kiến trúc DCGAN có bốn lớp phức tạp cho Discriminator và bốn lớp phức tạp "phân đoạn strided" cho Generator.
- Discriminator là một sự phức tạp 4 lớp với chuẩn hóa hàng loạt (ngoại trừ lớp đầu vào của nó) và các kích hoạt ReLU bị rò rỉ.
- Leaky ReLU là một hàm phi tuyến cung cấp một đầu ra không phải bằng không cho một đầu vào âm. Nó nhằm mục đích khắc phục vấn đề "sắp chết ReLU" và giúp các gradient chảy dễ dàng hơn thông qua kiến trúc.

18.2.6 Bài tập

1. Điều gì sẽ xảy ra nếu chúng ta sử dụng kích hoạt ReLU tiêu chuẩn thay vì ReLU bị rò rỉ?
2. Áp dụng DCGAN trên Fashion-MNIST và xem danh mục nào hoạt động tốt và loại nào không.

Discussions[212]

[212] https://discuss.d2l.ai/t/1083

19 | Phụ lục: Toán học cho Deep Learning

** Brent Werness** (* Amazon*), **Rachel Hu** (* Amazon*) và tác giả của cuốn sách này

Một trong những phần tuyệt vời của học sâu hiện đại là thực tế là phần lớn nó có thể được hiểu và sử dụng mà không có sự hiểu biết đầy đủ về toán học bên dưới nó. Đây là một dấu hiệu cho thấy lĩnh vực này đang trưởng thành. Cũng giống như hầu hết các nhà phát triển phần mềm không còn cần phải lo lắng về lý thuyết về các chức năng tính toán, các học viên học sâu cũng không cần phải lo lắng về nền tảng lý thuyết của khả năng học tối đa.

Nhưng, chúng tôi chưa hoàn toàn ở đó.

Trong thực tế, đôi khi bạn sẽ cần phải hiểu làm thế nào các lựa chọn kiến trúc ảnh hưởng đến dòng chảy gradient, hoặc các giả định ngầm mà bạn thực hiện bằng cách đào tạo với một chức năng mất mát nhất định. Bạn có thể cần phải biết những gì trong các biện pháp entropy thế giới, và làm thế nào nó có thể giúp bạn hiểu chính xác ý nghĩa của bitmỗi ký tự trong mô hình của bạn. Tất cả đều đòi hỏi sự hiểu biết toán học sâu hơn.

Phụ lục này nhằm mục đích cung cấp cho bạn nền toán học bạn cần để hiểu lý thuyết cốt lõi của học sâu hiện đại, nhưng nó không đầy đủ. Chúng tôi sẽ bắt đầu với việc kiểm tra đại số tuyến tính ở độ sâu lớn hơn. Chúng tôi phát triển một sự hiểu biết hình học của tất cả các đối tượng đại số tuyến tính phổ biến và các hoạt động sẽ cho phép chúng tôi hình dung các hiệu ứng của các biến đổi khác nhau trên dữ liệu của chúng tôi. Một yếu tố chính là sự phát triển của những điều cơ bản của sự phân hủy eigen-.

Tiếp theo chúng ta phát triển lý thuyết về phép tính vi phân đến mức chúng ta có thể hiểu đầy đủ tại sao gradient là hướng của dòng dõi dốc nhất, và tại sao sự lan truyền ngược lại có dạng nó. Tích phân sau đó được thảo luận với mức độ cần thiết để hỗ trợ chủ đề tiếp theo của chúng tôi, lý thuyết xác suất.

Các vấn đề gặp phải trong thực tế thường xuyên là không chắc chắn, và do đó chúng ta cần một ngôn ngữ để nói về những điều không chắc chắn. Chúng tôi xem xét lý thuyết về các biến ngẫu nhiên và các bản phân phối thường gặp nhất để chúng tôi có thể thảo luận về các mô hình xác suất. Điều này cung cấp nền tảng cho phân loại Bayes ngây thơ, một kỹ thuật phân loại xác suất.

Liên quan chặt chẽ đến lý thuyết xác suất là nghiên cứu về thống kê. Mặc dù số liệu thống kê là một lĩnh vực quá lớn để thực hiện công lý trong một phần ngắn, chúng tôi sẽ giới thiệu các khái niệm cơ bản mà tất cả các học viên máy học cần phải nhận thức, đặc biệt là: đánh giá và so sánh các ước lượng, tiến hành các bài kiểm tra giả thuyết và xây dựng khoảng thời gian tự tin.

Cuối cùng, chúng ta chuyển sang chủ đề lý thuyết thông tin, đó là nghiên cứu toán học về lưu trữ và truyền thông tin. Điều này cung cấp ngôn ngữ cốt lõi mà theo đó chúng ta có thể thảo luận về số lượng thông tin mà một mô hình nắm giữ trên một miền diễn ngôn.

Kết hợp với nhau, những hình thành cốt lõi của các khái niệm toán học cần thiết để bắt đầu con đường hướng tới một sự hiểu biết sâu sắc về học sâu.

19.1 Hình học và các hoạt động đại số tuyến tính

Trong Section 3.3, chúng tôi gặp phải những điều cơ bản về đại số tuyến tính và thấy nó có thể được sử dụng như thế nào để thể hiện các hoạt động phổ biến để chuyển đổi dữ liệu của chúng tôi. Đại số tuyến tính là một trong những trụ cột toán học quan trọng dựa trên phần lớn công việc mà chúng ta làm trong học sâu và trong học máy rộng hơn. Trong khi Section 3.3 chứa đủ máy móc để truyền đạt cơ học của các mô hình học sâu hiện đại, có rất nhiều hơn cho chủ đề này. Trong phần này, chúng ta sẽ đi sâu hơn, làm nổi bật một số diễn giải hình học của các phép toán đại số tuyến tính, và giới thiệu một vài khái niệm cơ bản, bao gồm eigenvalues và eigenvectors.

19.1.1 Hình học của Vectơ Trước tiên, chúng ta cần thảo luận về hai cách giải thích hình học phổ biến của vectơ, như một trong hai điểm hoặc hướng trong không gian. Về cơ bản, một vectơ là danh sách các số như danh sách Python bên dưới.

```
v = [1, 7, 0, 1]
```

Các nhà toán học thường viết điều này dưới dạng vectơ * cột* hoặc * hàng*, đó là để nói là

$$\mathbf{x} = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 1 \end{bmatrix}, \tag{19.1.1}$$

hoặc là

$$\mathbf{x}^\top = \begin{bmatrix} 1 & 7 & 0 & 1 \end{bmatrix}. \tag{19.1.2}$$

Chúng thường có các diễn giải khác nhau, trong đó các ví dụ dữ liệu là vectơ cột và trọng lượng được sử dụng để tạo thành các tổng trọng số là vectơ hàng. Tuy nhiên, nó có thể có lợi để linh hoạt. Như chúng ta đã mô tả trong Section 3.3, mặc dù định hướng mặc định của một vectơ duy nhất là một vectơ cột, đối với bất kỳ ma trận nào đại diện cho một tập dữ liệu dạng bảng, coi mỗi ví dụ dữ liệu như một vectơ hàng trong ma trận là thông thường hơn.

Với một vector, cách giải thích đầu tiên mà chúng ta nên đưa ra nó là một điểm trong không gian. Trong hai hoặc ba chiều, ta có thể hình dung các điểm này bằng cách sử dụng các thành phần của các vectơ để xác định vị trí của các điểm trong không gian so với một tham chiếu cố định gọi là *origin*. Điều này có thể được nhìn thấy trong Fig. 19.1.1.

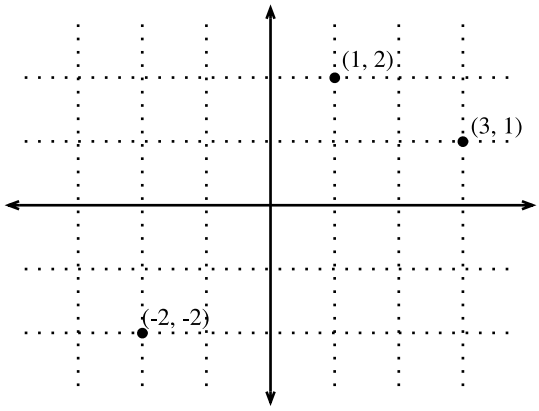


Fig. 19.1.1: An illustration of visualizing vectors as points in the plane. The first component of the vector gives the x-coordinate, the second component gives the y-coordinate. Higher dimensions are analogous, although much harder to visualize.

Quan điểm hình học này cho phép chúng ta xem xét vấn đề ở mức độ trừu tượng hơn. Không còn phải đối mặt với một số vấn đề dường như không thể vượt qua như phân loại hình ảnh là mèo hoặc chó, chúng ta có thể bắt đầu xem xét các nhiệm vụ trừu tượng như tập hợp các điểm trong không gian và hình dung nhiệm vụ khám phá cách tách hai cụm điểm riêng biệt.

Song song, có quan điểm thứ hai mà mọi người thường lấy vectơ: như hướng trong không gian. Chúng ta không chỉ có thể nghĩ về vector $\mathbf{v} = [3, 2]^\top$ là vị trí 3 đơn vị ở bên phải và 2 các đơn vị lên từ nguồn gốc, chúng ta cũng có thể nghĩ về nó như là hướng để thực hiện 3 bước sang phải và 2 bước lên. Bằng cách này, chúng tôi xem xét tất cả các vectơ trong hình Fig. 19.1.2 giống nhau.

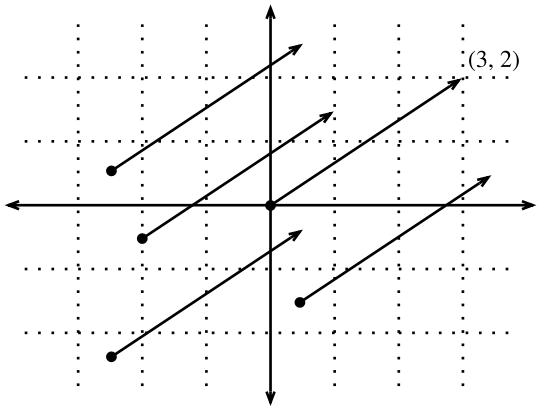


Fig. 19.1.2: Any vector can be visualized as an arrow in the plane. In this case, every vector drawn is a representation of the vector $(3, 2)^\top$.

Một trong những lợi ích của sự thay đổi này là chúng ta có thể hiểu rõ về hành động bổ sung vector. Đặc biệt, chúng ta làm theo các hướng dẫn được đưa ra bởi một vectơ, và sau đó làm theo các hướng dẫn được đưa ra bởi người kia, như được thấy trong Fig. 19.1.3.

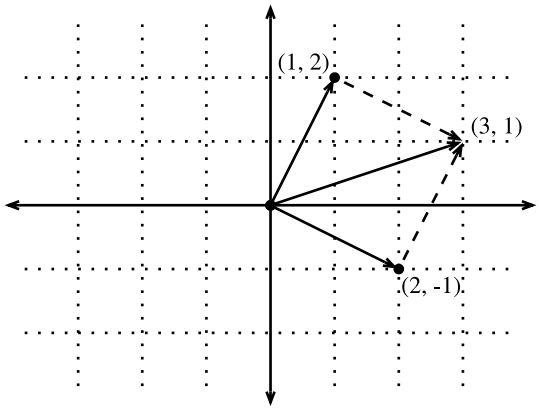


Fig. 19.1.3: We can visualize vector addition by first following one vector, and then another.

Phép trừ vector có cách giải thích tương tự. Bằng cách xem xét danh tính mà $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$, chúng ta thấy rằng vector $\mathbf{u} - \mathbf{v}$ là hướng đưa chúng ta từ điểm $\mathbf{v}$ đến điểm $\mathbf{u}$.

19.1.2 Dot Sản phẩm và Angles Như chúng ta đã thấy trong Section 3.3, nếu chúng ta lấy hai vectơ cột $\mathbf{u}$ và $\mathbf{v}$, chúng ta có thể tạo thành sản phẩm chấm của chúng bằng cách tính toán:

$$\mathbf{u}^\top \mathbf{v} = \sum_i u_i \cdot v_i. \tag{19.1.3}$$

Bởi vì (19.1.3) là đối xứng, chúng ta sẽ phản ánh ký hiệu của phép nhân cổ điển và viết

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}, \tag{19.1.4}$$

để làm nổi bật thực tế là trao đổi thứ tự của các vectơ sẽ mang lại câu trả lời tương tự.

Sản phẩm dot (19.1.3) cũng thừa nhận một giải thích hình học: it is closely related to the angle between two vectors. Consider the angle shown in Fig. 19.1.4.

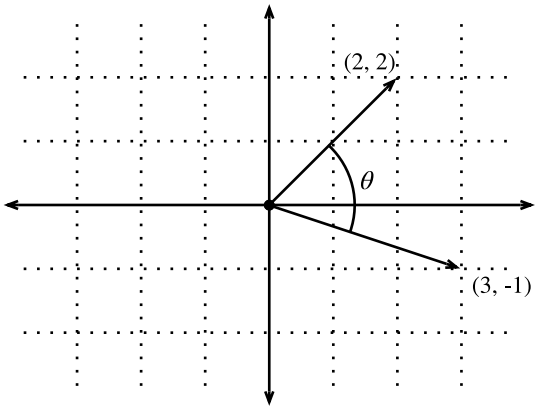


Fig. 19.1.4: Between any two vectors in the plane there is a well defined angle θ. We will see this angle is intimately tied to the dot product.

Để bắt đầu, chúng ta hãy xem xét hai vectơ cụ thể:

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s\cos(\theta), s\sin(\theta)). \tag{19.1.5}$$

Vector $\mathbf{v}$ có chiều dài r và chạy song song với trục x, và vector $\mathbf{w}$ có chiều dài s và ở góc θ với trục x. Nếu chúng ta tính toán sản phẩm chấm của hai vectơ này, chúng ta thấy rằng

$$\mathbf{v} \cdot \mathbf{w} = rs\cos(\theta) = \|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta). \tag{19.1.6}$$

Với một số thao tác đại số đơn giản, chúng ta có thể sắp xếp lại các thuật ngữ để có được

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{19.1.7}$$

Nói tóm lại, đối với hai vectơ cụ thể này, tích chấm kết hợp với các định mức cho chúng ta biết góc giữa hai vectơ. Thực tế tương tự này là đúng nói chung. Tuy nhiên, chúng tôi sẽ không lấy được biểu thức ở đây, nếu chúng ta xem xét viết $\|\mathbf{v} - \mathbf{w}\|^2$ theo hai cách: một với sản phẩm chấm và một hình học khác sử dụng định luật cosin, chúng ta có thể có được mối quan hệ đầy đủ. Thật vậy, đối với bất kỳ hai vectơ $\mathbf{v}$ và $\mathbf{w}$, góc giữa hai vectơ là

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}\right). \tag{19.1.8}$$

Đây là một kết quả tốt đẹp vì không có gì trong các tài liệu tham khảo tính toán hai chiều. Thật vậy, chúng ta có thể sử dụng điều này trong ba hoặc ba triệu chiều mà không gặp vấn đề gì.

Như một ví dụ đơn giản, chúng ta hãy xem làm thế nào để tính toán góc giữa một cặp vectơ:

```
%matplotlib inline
import torch
import torchvision
from IPython import display
from torchvision import transforms
from d2l import torch as d2l


def angle(v, w):
    return torch.acos(v.dot(w) / (torch.norm(v) * torch.norm(w)))

angle(torch.tensor([0, 1, 2], dtype=torch.float32), torch.tensor([2.0, 3, 4]))
```

```
tensor(0.4190)
```

Chúng tôi sẽ không sử dụng nó ngay bây giờ, nhưng rất hữu ích khi biết rằng chúng ta sẽ đề cập đến các vectơ mà góc là $\pi/2$ (hoặc tương đương $90°$) là * trực tho*. Bằng cách kiểm tra phương trình trên, chúng ta thấy rằng điều này xảy ra khi $\theta = \pi/2$, đó là điều tương tự như $\cos(\theta) = 0$. Cách duy nhất điều này có thể xảy ra là nếu bản thân sản phẩm chấm bằng 0, và hai vectơ là trực giao nếu và chỉ khi $\mathbf{v} \cdot \mathbf{w} = 0$. Điều này sẽ chứng minh là một công thức hữu ích khi hiểu các đối tượng về mặt hình học.

It is reasonable to ask: why is computing the angle useful? The answer comes in the kind of invariance we expect data to have. Consider an image, and a duplicate image, where every pixel value is the same but 10% the brightness. The values of the individual pixels are in general far from the original values. Thus, if one computed the distance between the original image and the darker one, the distance can be large. However, for most ML applications, the *content* is the same—it is still an image of a cat as far as a cat/dog classifier is concerned. However, if we consider the angle, it is not hard to see that for any vector $\mathbf{v}$, the angle between $\mathbf{v}$ and $0.1 \cdot \mathbf{v}$ is zero. This corresponds to the fact that scaling vectors keeps the same direction and just changes the length. The angle considers the darker image identical.

Ví dụ như thế này ở khắp mọi nơi. Trong văn bản, chúng ta có thể muốn chủ đề được thảo luận không thay đổi nếu chúng ta viết dài gấp đôi tài liệu nói cùng một điều. Đối với một số mã hóa (chẳng hạn như đếm số lần

xuất hiện của các từ trong một số từ vựng), điều này tương ứng với việc tăng gấp đôi vector mã hóa tài liệu, vì vậy một lần nữa chúng ta có thể sử dụng góc.

Cosine Similarity

In ML contexts where the angle is employed to measure the closeness of two vectors, practitioners adopt the term *cosine similarity* to refer to the portion

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\|\|\mathbf{w}\|}. \tag{19.1.9}$$

Cosin có giá trị lớn nhất là 1 khi hai vectơ trỏ theo cùng một hướng, giá trị tối thiểu là -1 khi chúng chỉ theo hướng ngược lại, và giá trị 0 khi hai vectơ là trực giao. Lưu ý rằng nếu các thành phần của vectơ chiều cao được lấy mẫu ngẫu nhiên với trung bình 0, cosin của chúng gần như sẽ luôn gần với 0.

19.1.3 Siêu máy bay

Ngoài việc làm việc với các vectơ, một đối tượng chính khác mà bạn phải hiểu để đi xa trong đại số tuyến tính là * hyperplane*, một khái quát hóa cho các kích thước cao hơn của một đường thẳng (hai chiều) hoặc của một mặt phẳng (ba chiều). Trong một không gian vectơ d chiều, một siêu phẳng có $d-1$ kích thước và chia không gian thành hai nửa không gian.

Hãy để chúng tôi bắt đầu với một ví dụ. Giả sử rằng chúng ta có một vector cột $\mathbf{w} = [2, 1]^\top$. Chúng tôi muốn biết, "những điểm $\mathbf{v}$ với $\mathbf{w} \cdot \mathbf{v} = 1$ là gì?" Bằng cách nhớ lại kết nối giữa các sản phẩm chấm và góc trên (19.1.8), chúng ta có thể thấy rằng điều này tương đương với

$$\|\mathbf{v}\|\|\mathbf{w}\|\cos(\theta) = 1 \iff \|\mathbf{v}\|\cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \tag{19.1.10}$$

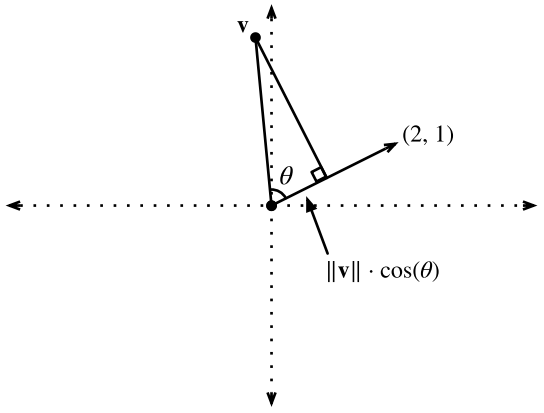


Fig. 19.1.5: Recalling trigonometry, we see the formula $\|\mathbf{v}\|\cos(\theta)$ is the length of the projection of the vector $\mathbf{v}$ onto the direction of $\mathbf{w}$

Nếu chúng ta xem xét ý nghĩa hình học của biểu thức này, chúng ta thấy rằng điều này tương đương với việc nói rằng độ dài của phép chiếu $\mathbf{v}$ theo hướng $\mathbf{w}$ chính xác là $1/\|\mathbf{w}\|$, như được thể hiện trong Fig. 19.1.5. Tập hợp tất cả các điểm mà điều này là đúng là một đường thẳng ở góc phải với vector $\mathbf{w}$. Nếu chúng ta muốn, chúng ta có thể tìm thấy phương trình cho dòng này và thấy rằng nó là $2x + y = 1$ hoặc tương đương $y = 1 - 2x$.

Nếu bây giờ chúng ta nhìn vào những gì xảy ra khi chúng ta hỏi về tập hợp các điểm với $\mathbf{w} \cdot \mathbf{v} > 1$ hoặc $\mathbf{w} \cdot \mathbf{v} < 1$, chúng ta có thể thấy rằng đây là những trường hợp các dự báo dài hơn hoặc ngắn hơn $1/\|\mathbf{w}\|$,

tương ứng. Do đó, hai bất bình đẳng đó xác định hai bên của dòng. Bằng cách này, chúng tôi đã tìm ra cách cắt không gian của chúng tôi thành hai nửa, trong đó tất cả các điểm ở một bên đều có sản phẩm chấm dưới ngưỡng và phía bên kia ở trên như chúng ta thấy trong Fig. 19.1.6.

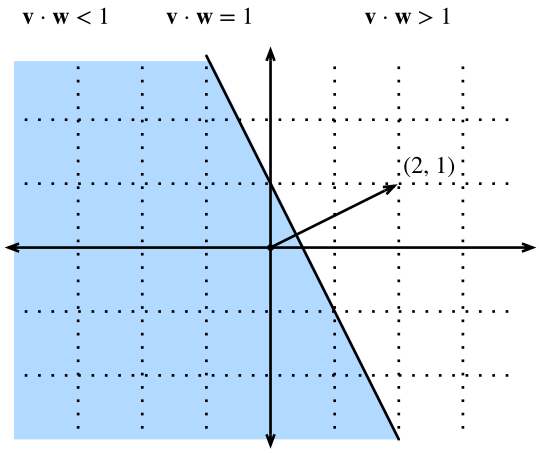


Fig. 19.1.6: If we now consider the inequality version of the expression, we see that our hyperplane (in this case: just a line) separates the space into two halves.

Câu chuyện trong chiều cao hơn là nhiều như nhau. Nếu bây giờ chúng ta lấy $\mathbf{w} = [1, 2, 3]^\top$ và hỏi về các điểm trong ba chiều với $\mathbf{w} \cdot \mathbf{v} = 1$, chúng ta có được một mặt phẳng ở góc phải với vector $\mathbf{w}$ đã cho. Hai bất đẳng thức lại xác định hai mặt của mặt phẳng như được thể hiện trong Fig. 19.1.7.

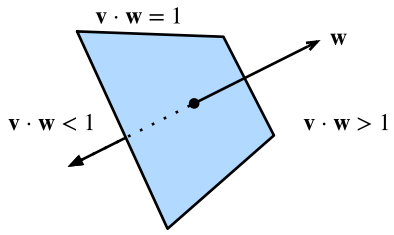


Fig. 19.1.7: Hyperplanes in any dimension separate the space into two halves.

Trong khi khả năng hình dung của chúng ta hết vào thời điểm này, không có gì ngăn cản chúng ta làm điều này trong hàng chục, hàng trăm hoặc hàng tỷ chiều. Điều này xảy ra thường xuyên khi nghĩ về các mô hình học máy. Ví dụ, chúng ta có thể hiểu các mô hình phân loại tuyến tính như các mô hình từ Section 4.4, như các phương pháp để tìm các siêu phẳng tách biệt các lớp mục tiêu khác nhau. Trong bối cảnh này, các siêu máy bay như vậy thường được gọi là *kế hoạch quyết định*. Phần lớn các mô hình phân loại học sâu kết thúc bằng một lớp tuyến tính được đưa vào một softmax, do đó người ta có thể diễn giải vai trò của mạng nơ-ron sâu là tìm một nhúng phi tuyến tính sao cho các lớp mục tiêu có thể được tách sạch bằng các siêu phẳng.

Để đưa ra một ví dụ được xây dựng bằng tay, hãy chú ý rằng chúng ta có thể tạo ra một mô hình hợp lý để phân loại hình ảnh nhỏ của áo phông và quần dài từ tập dữ liệu thời trang MNIST (thấy trong Section 4.5) bằng cách chỉ lấy vectơ giữa phương tiện của chúng để xác định mặt phẳng quyết định và nhãn cầu một ngưỡng thô. Đầu tiên chúng ta sẽ tải dữ liệu và tính toán trung bình.

```
# Load in the dataset
trans = []
trans.append(transforms.ToTensor())
```

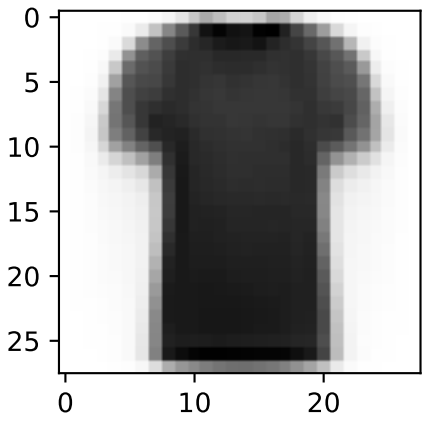
(continues on next page)

(continued from previous page)

```
trans = transforms.Compose(trans)
train = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                          train=True, download=True)
test = torchvision.datasets.FashionMNIST(root="../data", transform=trans,
                                         train=False, download=True)

X_train_0 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 0]).type(torch.float32)
X_train_1 = torch.stack(
    [x[0] * 256 for x in train if x[1] == 1]).type(torch.float32)
X_test = torch.stack(
    [x[0] * 256 for x in test if x[1] == 0 or x[1] == 1]).type(torch.float32)
y_test = torch.stack([torch.tensor(x[1]) for x in test
                      if x[1] == 0 or x[1] == 1]).type(torch.float32)

# Compute averages
ave_0 = torch.mean(X_train_0, axis=0)
ave_1 = torch.mean(X_train_1, axis=0)
```

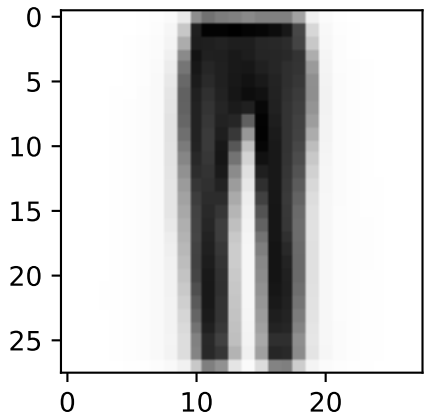
Nó có thể là thông tin để kiểm tra các trung bình này một cách chi tiết, vì vậy hãy để chúng tôi vẽ những gì họ trông như thế nào. Trong trường hợp này, chúng ta thấy rằng trung bình thực sự giống như một hình ảnh mờ của một chiếc áo phông.

```
# Plot average t-shirt
d2l.set_figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



Trong trường hợp thứ hai, chúng ta lại thấy rằng trung bình giống như một hình ảnh mờ của quần.

```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```



Trong một giải pháp hoàn toàn máy học, chúng ta sẽ tìm hiểu ngưỡng từ tập dữ liệu. Trong trường hợp này, tôi chỉ đơn giản là mắt một ngưỡng trông tốt trên dữ liệu đào tạo bằng tay.

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
# '@' is Matrix Multiplication operator in pytorch.
predictions = X_test.reshape(2000, -1) @ (w.flatten()) > -1500000

# Accuracy
torch.mean((predictions.type(y_test.dtype) == y_test).float(), dtype=torch.
↪float64)
```

```
/tmp/ipykernel_118305/3819644537.py:2: UserWarning: The use of x.
↪T on tensors of dimension other than 2 to reverse their shape is␣
↪deprecated and it will throw an error in a future release. Consider␣
↪x.mT to transpose batches of matricesor x.permute(*torch.arange(x.
↪ndim - 1, -1, -1)) to reverse the dimensions of a tensor. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2318.)
  w = (ave_1 - ave_0).T
```

```
tensor(0.7870, dtype=torch.float64)
```

19.1.4 Hình học của biến đổi tuyến tính

Thông qua Section 3.3 và các cuộc thảo luận trên, chúng ta có một sự hiểu biết vững chắc về hình học của vectơ, độ dài và góc. Tuy nhiên, có một đối tượng quan trọng mà chúng ta đã bỏ qua thảo luận, và đó là một sự hiểu biết hình học về các biến đổi tuyến tính được thể hiện bằng ma trận. Nội bộ hoàn toàn những gì ma trận có thể làm để chuyển đổi dữ liệu giữa hai không gian chiều cao có khả năng khác nhau cần thực hành đáng kể, và vượt quá phạm vi của phụ lục này. Tuy nhiên, chúng ta có thể bắt đầu xây dựng trực giác theo hai chiều.

Giả sử rằng chúng ta có một số ma trận:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \tag{19.1.11}$$

If we want to apply this to an arbitrary vector $\mathbf{v} = [x, y]^\top$, we multiply and see that

$$\begin{aligned}\mathbf{A}\mathbf{v} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \\ &= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} \\ &= x \left\{ \mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} + y \left\{ \mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}.\end{aligned} \tag{19.1.12}$$

Điều này có vẻ giống như một tính toán kỳ lạ, nơi một cái gì đó rõ ràng trở nên hơi bất khả xâm phạm. Tuy nhiên, nó cho chúng ta biết rằng chúng ta có thể viết cách mà ma trận biến đổi vectơ *any* về cách nó biến đổi * hai vectơ cụ thể*: :math:`[1,0]^top` và :math:`[0,1]^top`. Điều này đáng để xem xét trong một khoảnh khắc. Về cơ bản chúng ta đã giảm một bài toán vô hạn (những gì xảy ra với bất kỳ cặp số thực nào) thành một vấn đề hữu hạn (những gì xảy ra với các vectơ cụ thể này). Các vectơ này là một ví dụ a *basis*, nơi chúng ta có thể viết bất kỳ vectơ nào trong không gian của chúng ta dưới dạng tổng trọng số của các vectơ cơ bản* này*.

Hãy để chúng tôi vẽ những gì xảy ra khi chúng ta sử dụng ma trận cụ thể

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \tag{19.1.13}$$

Nếu chúng ta nhìn vào vector cụ thể $\mathbf{v} = [2, -1]^\top$, chúng ta thấy đây là $2 \cdot [1, 0]^\top + -1 \cdot [0, 1]^\top$ và do đó chúng ta biết rằng ma trận A sẽ gửi điều này đến $2(\mathbf{A}[1, 0]^\top) + -1(\mathbf{A}[0, 1])^\top = 2[1, -1]^\top - [2, 3]^\top = [0, -5]^\top$. Nếu chúng ta làm theo logic này thông qua một cách cẩn thận, nói bằng cách xem xét lưới của tất cả các cặp số nguyên của điểm, chúng ta thấy rằng những gì xảy ra là phép nhân ma trận có thể nghiêng, xoay, và quy mô lưới điện, nhưng cấu trúc lưới phải vẫn như bạn thấy trong Fig. 19.1.8.

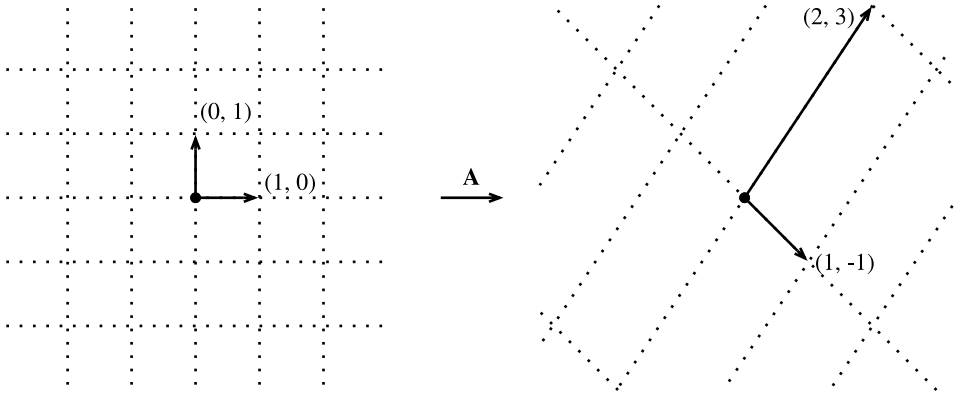


Fig. 19.1.8: The matrix **A** acting on the given basis vectors. Notice how the entire grid is transported along with it.

Đây là điểm trực quan quan trọng nhất để nội tâm hóa về các biến đổi tuyến tính được thể hiện bằng ma trận. Ma trận không có khả năng làm biến dạng một số phần của không gian khác với những phần khác. Tất cả những gì họ có thể làm là lấy tọa độ ban đầu trên không gian của chúng ta và nghiêng, xoay và quy mô chúng.

Một số biến dạng có thể nghiêm trọng. For instance ví dụ the matrix ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \tag{19.1.14}$$

nén toàn bộ mặt phẳng hai chiều xuống một dòng duy nhất. Xác định và làm việc với các biến đổi như vậy là chủ đề của một phần sau, nhưng về mặt hình học chúng ta có thể thấy rằng điều này về cơ bản khác với các loại biến đổi mà chúng ta đã thấy ở trên. Ví dụ, kết quả từ ma trận **A** có thể là "uốn cong trở lại" vào lưới gốc. Kết quả từ ma trận **B** không thể vì chúng ta sẽ không bao giờ biết vector $[1, 2]^\top$ đến từ đâu - nó là $[1, 1]^\top$ hay $[0, -1]^\top$?

Trong khi bức ảnh này dành cho ma trận 2×2, không có gì ngăn cản chúng ta tham gia các bài học đã học vào các chiều cao hơn. Nếu chúng ta lấy vectơ cơ bản tương tự như $[1, 0, \ldots, 0]$ và xem ma trận của chúng ta gửi chúng ở đâu, chúng ta có thể bắt đầu có cảm giác về cách nhân ma trận biến dạng toàn bộ không gian trong bất kỳ không gian chiều nào chúng ta đang xử lý.

19.1.5 Sự phụ thuộc tuyến tính

Xem xét lại ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \tag{19.1.15}$$

Điều này nén toàn bộ mặt phẳng xuống để sống trên dòng đơn $y = 2x$. Câu hỏi bây giờ đặt ra: có cách nào đó chúng ta có thể phát hiện ra điều này chỉ nhìn vào chính ma trận? Câu trả lời là thực sự chúng ta có thể. Chúng ta hãy lấy $\mathbf{b}_1 = [2, 4]^\top$ và $\mathbf{b}_2 = [-1, -2]^\top$ là hai cột của **B**. Hãy nhớ rằng chúng ta có thể viết mọi thứ được chuyển đổi bởi ma trận **B** như một tổng trọng số của các cột của ma trận: như $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$. Chúng tôi gọi đây là một kết hợp tuyến tính*. Thực tế là $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ có nghĩa là chúng ta có thể viết bất kỳ sự kết hợp tuyến tính nào của hai cột đó hoàn toàn về nói $\mathbf{b}_2$ kể từ

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \tag{19.1.16}$$

Điều này có nghĩa là một trong các cột, theo một nghĩa nào đó, dư thừa vì nó không xác định một hướng duy nhất trong không gian. Điều này không nên làm chúng ta ngạc nhiên quá nhiều vì chúng ta đã thấy rằng ma trận này sụp đổ toàn bộ mặt phẳng xuống thành một dòng duy nhất. Hơn nữa, chúng ta thấy rằng sự phụ thuộc tuyến tính $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ nắm bắt điều này. Để làm cho điều này đối xứng hơn giữa hai vectơ, chúng ta sẽ viết cái này là

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \tag{19.1.17}$$

Nói chung, chúng ta sẽ nói rằng một tập hợp các vectơ $\mathbf{v}_1, \ldots, \mathbf{v}_k$ là * phụ thuộc tuyến tính* nếu có hệ số tồn tại $a_1, \ldots, a_k$ * không bằng 0* để

$$\sum_{i=1}^{k} a_i\mathbf{v_i} = 0. \tag{19.1.18}$$

Trong trường hợp này, chúng ta có thể giải quyết cho một trong các vectơ về một số sự kết hợp của những người khác, và hiệu quả làm cho nó dư thừa. Do đó, sự phụ thuộc tuyến tính trong các cột của ma trận là một nhân chứng cho thực tế là ma trận của chúng ta đang nén không gian xuống một số chiều thấp hơn. Nếu không có sự phụ thuộc tuyến tính, chúng tôi nói rằng các vectơ là * độc lập tuyến tính*. Nếu các cột của ma trận độc lập tuyến tính, không xảy ra nén và thao tác có thể được hoàn tác.

19.1.6 thứ hạng

Nếu chúng ta có một ma trận $n \times m$ chung, nó là hợp lý để hỏi không gian kích thước mà ma trận ánh xạ vào. Một khái niệm được gọi là *rank* sẽ là câu trả lời của chúng tôi. Trong phần trước, chúng tôi lưu ý rằng một sự phụ thuộc tuyến tính làm chứng cho việc nén không gian vào một chiều thấp hơn và vì vậy chúng tôi sẽ có thể sử dụng điều này để xác định khái niệm về cấp bậc. Đặc biệt, cấp bậc của một ma trận $\mathbf{A}$ là số lượng lớn nhất các cột độc lập tuyến tính giữa tất cả các tập con của cột. Ví dụ, ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \tag{19.1.19}$$

có rank$(B) = 1$, vì hai cột phụ thuộc tuyến tính, nhưng bản thân một trong hai cột không phụ thuộc tuyến tính. Đối với một ví dụ khó khăn hơn, chúng ta có thể xem xét

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \tag{19.1.20}$$

và cho thấy $\mathbf{C}$ đã xếp hạng hai vì, ví dụ, hai cột đầu tiên độc lập tuyến tính, tuy nhiên bất kỳ bộ sưu tập nào trong bốn cột phụ thuộc.

Thủ tục này, như mô tả, rất không hiệu quả. Nó đòi hỏi phải xem xét mọi tập hợp con của các cột của ma trận nhất định của chúng ta, và do đó có khả năng hàm mũ trong số cột. Sau đó chúng ta sẽ thấy một cách hiệu quả hơn về tính toán để tính toán thứ hạng của ma trận, nhưng hiện tại, điều này là đủ để thấy rằng khái niệm này được xác định rõ và hiểu ý nghĩa.

19.1.7 Khả năng đảo ngược

Chúng ta đã thấy ở trên rằng phép nhân của một ma trận với các cột phụ thuộc tuyến tính không thể hoàn tác, tức là, không có hoạt động nghịch đảo nào luôn có thể khôi phục đầu vào. Tuy nhiên, nhân với ma trận cấp bậc đầy đủ (tức là, một số $\mathbf{A}$ là ma trận $n \times n$ với thứ hạng n), chúng ta nên luôn luôn có thể hoàn tác nó. Xem xét ma trận

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \tag{19.1.21}$$

mà là ma trận với những người dọc theo đường chéo, và số không ở nơi khác. Chúng tôi gọi đây là ma trận *identity*. Đó là ma trận khiến dữ liệu của chúng tôi không thay đổi khi áp dụng. Để tìm một ma trận hoàn tác những gì ma trận của chúng tôi $\mathbf{A}$ đã làm, chúng tôi muốn tìm một ma trận $\mathbf{A}^{-1}$ sao cho

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \tag{19.1.22}$$

Nếu chúng ta xem xét điều này như một hệ thống, chúng ta có $n \times n$ ẩn số (các mục của $\mathbf{A}^{-1}$) và $n \times n$ phương trình (bình đẳng cần giữ giữa mọi mục nhập của sản phẩm $\mathbf{A}^{-1}\mathbf{A}$ và mọi mục nhập của $\mathbf{I}$) vì vậy chúng ta nên khái quát mong đợi một giải pháp tồn tại. Thật vậy, trong phần tiếp theo chúng ta sẽ thấy một đại lượng được gọi là *determinant*, có thuộc tính miễn là yếu tố quyết định không phải bằng 0, chúng ta có thể tìm ra giải pháp. Chúng tôi gọi ma trận $\mathbf{A}^{-1}$ như ma trận *inverse*. Ví dụ, nếu $\mathbf{A}$ là ma trận 2×2 chung

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{19.1.23}$$

sau đó chúng ta có thể thấy rằng nghịch đảo là

$$\frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \tag{19.1.24}$$

Chúng ta có thể kiểm tra để xem điều này bằng cách thấy rằng nhân với nghịch đảo được đưa ra bởi công thức trên hoạt động trong thực tế.

```
M = torch.tensor([[1, 2], [1, 4]], dtype=torch.float32)
M_inv = torch.tensor([[2, -1], [-0.5, 0.5]])
M_inv @ M
```

```
tensor([[1., 0.],
        [0., 1.]])
```

Các vấn đề số Trong khi nghịch đảo của ma trận rất hữu ích về lý thuyết, chúng ta phải nói rằng hầu hết thời gian chúng ta không muốn *sử dụng* ma trận nghịch đảo để giải quyết một vấn đề trong thực tế. Nói chung, có nhiều thuật toán ổn định số hơn để giải các phương trình tuyến tính như

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{19.1.25}$$

hơn là tính toán nghịch đảo và nhân lên để có được

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \tag{19.1.26}$$

Cũng giống như phân chia bởi một số nhỏ có thể dẫn đến sự bất ổn số, vì vậy có thể đảo ngược của một ma trận gần với có thứ hạng thấp.

Hơn nữa, thông thường ma trận $\mathbf{A}$ là * thưa thớt *, có nghĩa là nó chỉ chứa một số lượng nhỏ các giá trị không phải bằng không. Nếu chúng ta khám phá các ví dụ, chúng ta sẽ thấy rằng điều này không có nghĩa là nghịch đảo thưa thớt. Ngay cả khi $\mathbf{A}$ là một 1 triệu bởi 1 triệu ma trận với chỉ 5 triệu mục không phải bằng không (và do đó chúng tôi chỉ cần lưu trữ những 5 triệu), nghịch đảo thường sẽ có hầu hết mọi mục không âm, yêu cầu chúng tôi lưu trữ tất cả 1M^2 mục đó là 1 nghìn tỷ mục!

Mặc dù chúng ta không có thời gian để lặn tất cả các cách vào các vấn đề số gai góc thường gặp khi làm việc với đại số tuyến tính, chúng tôi muốn cung cấp cho bạn một số trực giác về khi nào cần tiến hành thận trọng, và nói chung tránh đảo ngược trong thực tế là một nguyên tắc tốt.

19.1.8 Xác định Quan điểm hình học của đại số tuyến tính đưa ra một cách trực quan để diễn giải một đại lượng cơ bản được gọi là *quyết định*. Hãy xem xét hình ảnh lưới từ trước, nhưng bây giờ với một khu vực được tô sáng (Fig. 19.1.9).

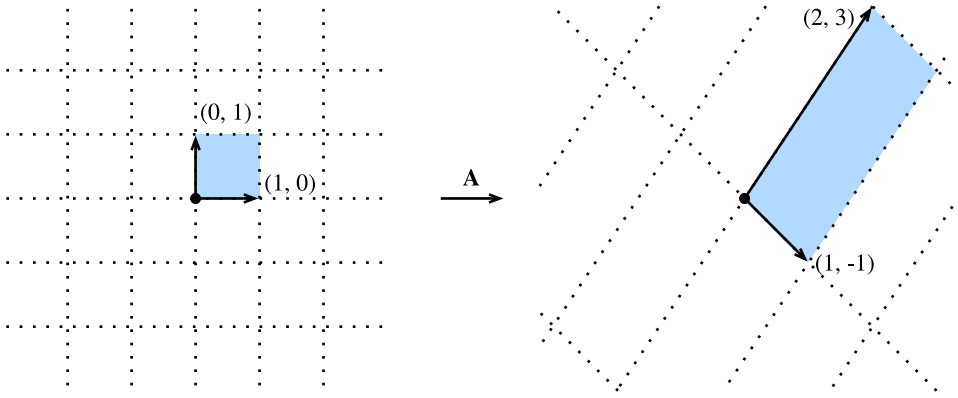


Fig. 19.1.9: The matrix **A** again distorting the grid. This time, I want to draw particular attention to what happens to the highlighted square.

Nhìn vào hình vuông được tô sáng. Đây là một hình vuông với các cạnh được đưa ra bởi $(0, 1)$ và $(1, 0)$ và do đó nó có diện tích một. Sau **A** biến hình vuông này, chúng ta thấy rằng nó trở thành một hình bình hành. Không có lý do gì hình bình hành này nên có cùng một khu vực mà chúng ta bắt đầu, và thực sự trong trường hợp cụ thể được hiển thị ở đây

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}, \tag{19.1.27}$$

nó là một bài tập trong hình học tọa độ để tính toán diện tích của hình bình hành này và có được rằng khu vực này là 5.

Nói chung, nếu chúng ta có một ma trận

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{19.1.28}$$

chúng ta có thể thấy với một số tính toán rằng diện tích của hình bình hành kết quả là $ad - bc$. Khu vực này được gọi là *quyết định*.

Hãy để chúng tôi kiểm tra điều này một cách nhanh chóng với một số mã ví dụ.

```
torch.det(torch.tensor([[1, -1], [2, 3]], dtype=torch.float32))
```

```
tensor(5.)
```

Mắt đại bàng giữa chúng ta sẽ nhận thấy rằng biểu thức này có thể bằng 0 hoặc thậm chí tiêu cực. Đối với thuật ngữ tiêu cực, đây là vấn đề quy ước được thực hiện chung trong toán học: nếu ma trận lật hình, chúng ta nói khu vực này bị phủ định. Chúng ta hãy xem bây giờ rằng khi yếu tố quyết định bằng không, chúng ta tìm hiểu thêm.

Hãy để chúng tôi xem xét

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \tag{19.1.29}$$

Nếu chúng ta tính toán yếu tố quyết định của ma trận này, chúng ta nhận được $2 \cdot (-2) - 4 \cdot (-1) = 0$. Với sự hiểu biết của chúng tôi ở trên, điều này có ý nghĩa. **B** nén hình vuông từ hình ảnh gốc xuống một đoạn đường, có diện tích bằng không. Và quả thực, bị nén thành một không gian chiều thấp hơn là cách duy nhất để có diện tích bằng 0 sau khi biến đổi. Như vậy chúng ta thấy kết quả sau là đúng: một ma trận A có thể đảo ngược nếu và chỉ khi yếu tố quyết định không bằng 0.

Như một nhận xét cuối cùng, hãy tưởng tượng rằng chúng ta có bất kỳ con số nào được vẽ trên mặt phẳng. Suy nghĩ giống như các nhà khoa học máy tính, chúng ta có thể phân hủy con số đó thành một tập hợp các ô vuông nhỏ để diện tích của hình là về bản chất chỉ là số lượng hình vuông trong phân hủy. Nếu bây giờ chúng ta biến đổi con số đó bằng một ma trận, chúng ta sẽ gửi từng ô vuông này thành hình bình hành, mỗi ô có diện tích được đưa ra bởi yếu tố quyết định. Chúng ta thấy rằng đối với bất kỳ con số nào, yếu tố quyết định cho số (ký) mà ma trận quy mô diện tích của bất kỳ hình nào.

Các yếu tố quyết định tính toán cho ma trận lớn hơn có thể tốn nhiều công sức, nhưng trực giác là như nhau. Yếu tố quyết định vẫn là yếu tố mà $n \times n$ ma trận quy mô n-khối lượng chiều.

19.1.9 Tensors và hoạt động đại số tuyến tính phổ biến

Năm Section 3.3 khái niệm về hàng chục đã được giới thiệu. Trong phần này, chúng ta sẽ đi sâu hơn vào các cơn co thắt tensor (tensor tương đương với phép nhân ma trận), và xem làm thế nào nó có thể cung cấp một cái nhìn thống nhất về một số hoạt động ma trận và vector.

Với ma trận và vectơ, chúng ta biết cách nhân chúng để chuyển đổi dữ liệu. Chúng ta cần phải có một định nghĩa tương tự cho hàng chục nếu chúng hữu ích cho chúng ta. Hãy suy nghĩ về phép nhân ma trận:

$$\mathbf{C} = \mathbf{AB}, \tag{19.1.30}$$

hoặc tương đương

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \tag{19.1.31}$$

Mô hình này là một trong những chúng ta có thể lặp lại cho hàng chục. Đối với hàng chục, không có một trường hợp nào để tổng hợp những gì có thể được lựa chọn phổ biến, vì vậy chúng ta cần xác định chính xác những chỉ số chúng ta muốn tổng hợp lại. For instance ví dụ we could consider xem xét

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \tag{19.1.32}$$

Một sự chuyển đổi như vậy được gọi là co lại *tensor*. Nó có thể đại diện cho một gia đình biến đổi linh hoạt hơn nhiều mà phép nhân ma trận một mình.

Là một đơn giản hóa ký hiệu thường được sử dụng, chúng ta có thể nhận thấy rằng tổng là trên chính xác những chỉ số xảy ra nhiều hơn một lần trong biểu thức, do đó mọi người thường làm việc với ký hiệu * Einstein*, trong đó tổng kết được thực hiện trong tất cả các chỉ số lặp đi lặp lại. Điều này cho phép biểu thức nhỏ gọn:

$$y_{il} = x_{ijkl} a_{jk}. \tag{19.1.33}$$

Ví dụ phổ biến từ đại số tuyến tính

Chúng ta hãy xem có bao nhiêu định nghĩa đại số tuyến tính mà chúng ta đã thấy trước đây có thể được thể hiện trong ký hiệu tensor nén này:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$
- $(\mathbf{A}\mathbf{v})_i = \sum_j a_{ij} v_j$
- $(\mathbf{A}\mathbf{B})_{ik} = \sum_j a_{ij} b_{jk}$
- $\text{tr}(\mathbf{A}) = \sum_i a_{ii}$

Bằng cách này, chúng ta có thể thay thế vô số ký hiệu chuyên biệt bằng các biểu thức tensor ngắn.

Expressing in Code Tensors cũng có thể được vận hành linh hoạt trong mã. Như đã thấy trong Section 3.3, chúng ta có thể tạo ra hàng chục như được hiển thị dưới đây.

```
# Define tensors
B = torch.tensor([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
A = torch.tensor([[1, 2], [3, 4]])
v = torch.tensor([1, 2])

# Print out the shapes
A.shape, B.shape, v.shape
```

```
(torch.Size([2, 2]), torch.Size([2, 2, 3]), torch.Size([2]))
```

Tổng kết Einstein đã được thực hiện trực tiếp. Các chỉ số xảy ra trong tổng kết Einstein có thể được truyền như một chuỗi, tiếp theo là các hàng chục đang được hành động. Ví dụ, để thực hiện phép nhân ma trận, chúng ta có thể xem xét tổng kết Einstein thấy ở trên ($\mathbf{A}\mathbf{v} = a_{ij} v_j$) và tự loại bỏ các chỉ số để thực hiện:

```
# Reimplement matrix multiplication
torch.einsum("ij, j -> i", A, v), A@v
```

```
(tensor([ 5, 11]), tensor([ 5, 11]))
```

Đây là một ký hiệu rất linh hoạt. Ví dụ nếu chúng ta muốn tính toán những gì sẽ được viết theo truyền thống như

$$c_{kl} = \sum_{ij} \mathbf{b}_{ijk} \mathbf{a}_{il} v_j. \tag{19.1.34}$$

nó có thể được thực hiện thông qua tổng kết Einstein như:

```
torch.einsum("ijk, il, j -> kl", B, A, v)
```

```
tensor([[ 90, 126],
        [102, 144],
        [114, 162]])
```

Ký hiệu này có thể đọc được và hiệu quả đối với con người, tuy nhiên cồng kềnh nếu vì bất kỳ lý do gì chúng ta cần tạo ra sự co lại căng thẳng theo chương trình. Vì lý do này, `einsum` cung cấp một ký hiệu thay thế bằng cách cung cấp các chỉ số nguyên cho mỗi tensor. Ví dụ, cùng một sự co lại tensor cũng có thể được viết là:

```
# PyTorch doesn't support this type of notation.
```

Một trong hai ký hiệu cho phép đại diện ngắn gọn và hiệu quả các cơn co thắt tensor trong mã.

19.1.10 Tóm tắt * Vectơ có thể được giải thích về mặt hình học như một trong hai điểm hoặc hướng trong không gian * Các sản phẩm chấm xác định khái niệm góc đến không gian chiều cao tùy ý. Chúng có thể được sử dụng để xác định các mặt phẳng quyết định thường được sử dụng như là bước cuối cùng trong một nhiệm vụ phân loại. * Phép nhân ma trận có thể được giải thích về mặt hình học là biến dạng thống nhất của tọa độ bên dưới. Chúng đại diện cho một cách rất hạn chế, nhưng về mặt toán học sạch, để biến đổi vectơ. * Phụ thuộc tuyến tính là một cách để biết khi một tập hợp các vectơ nằm trong một không gian chiều thấp hơn chúng ta mong đợi (giả sử bạn có 3 vectơ sống trong một không gian 2 chiều). Thứ hạng của ma trận là kích thước của tập hợp con lớn nhất của các cột của nó độc lập tuyến tính * Khi nghịch đảo của ma trận được xác định, đảo ngược ma trận cho phép chúng ta tìm một ma trận khác hoàn tác hành động của đầu tiên. Ma trận đảo ngược là hữu ích trong lý thuyết, nhưng đòi hỏi sự chăm sóc trong thực tế do sự bất ổn số.* yếu tố quyết định cho phép chúng ta đo lường bao nhiêu ma trận mở rộng hoặc ký hợp đồng một không gian. Một yếu tố quyết định nonzero ngụ ý một ma trận đảo ngược (không số ít) và một yếu tố xác định có giá trị không có nghĩa là ma trận không thể đảo ngược (số ít). * co thắt tensor và tổng kết Einstein cung cấp cho một ký hiệu gọn gàng và sạch sẽ để thể hiện nhiều tính toán được nhìn thấy trong học máy.

Exercises

1. What is the angle between

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \qquad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix}? \tag{19.1.35}$$

2. True or false: $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$ are inverses of one another?

3. Suppose that we draw a shape in the plane with area 100m^2. What is the area after transforming the figure by the matrix

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \tag{19.1.36}$$

4. Which of the following sets of vectors are linearly independent?

- $\left\{\begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}\right\}$

- $\left\{\begin{pmatrix}3\\1\\1\end{pmatrix}, \begin{pmatrix}1\\1\\1\end{pmatrix}, \begin{pmatrix}0\\0\\0\end{pmatrix}\right\}$
- $\left\{\begin{pmatrix}1\\1\\0\end{pmatrix}, \begin{pmatrix}0\\1\\-1\end{pmatrix}, \begin{pmatrix}1\\0\\1\end{pmatrix}\right\}$

5. Suppose that you have a matrix written as $A = \begin{bmatrix}c\\d\end{bmatrix} \cdot \begin{bmatrix}a & b\end{bmatrix}$ for some choice of values a, b, c, and d. True or false: the determinant of such a matrix is always 0?
6. The vectors $e_1 = \begin{bmatrix}1\\0\end{bmatrix}$ and $e_2 = \begin{bmatrix}0\\1\end{bmatrix}$ are orthogonal. What is the condition on a matrix A so that Ae_1 and Ae_2 are orthogonal?
7. How can you write $\text{tr}(\mathbf{A}^4)$ in Einstein notation for an arbitrary matrix A?

Discussions[213]

19.2 Eigendecompositions

Eigenvalues thường là một trong những quan niệm hữu ích nhất mà chúng ta sẽ gặp phải khi nghiên cứu đại số tuyến tính, tuy nhiên, với tư cách là người mới bắt đầu, thật dễ dàng để bỏ qua tầm quan trọng của chúng. Dưới đây, chúng tôi giới thiệu eigendecomposition và cố gắng truyền đạt một số ý thức về lý do tại sao nó lại quan trọng như vậy.

Giả sử rằng chúng ta có một ma trận A với các mục sau:

$$\mathbf{A} = \begin{bmatrix}2 & 0\\0 & -1\end{bmatrix}. \tag{19.2.1}$$

Nếu chúng ta áp dụng A cho bất kỳ vector $\mathbf{v} = [x, y]^\top$ nào, chúng ta có được một vector $\mathbf{Av} = [2x, -y]^\top$. Điều này có một cách giải thích trực quan: kéo dài vectơ rộng gấp đôi theo hướng x-, sau đó lật nó theo hướng y-.

Tuy nhiên, có * một số* vectơ mà một cái gì đó vẫn không thay đổi. Cụ thể là $[1, 0]^\top$ được gửi đến $[2, 0]^\top$ và $[0, 1]^\top$ được gửi đến $[0, -1]^\top$. Các vectơ này vẫn nằm trong cùng một dòng, và sửa đổi duy nhất là ma trận kéo dài chúng bằng một hệ số lần lượt là 2 và -1. Chúng tôi gọi các vectơ như vậy * eigenvectors* và yếu tố chúng được kéo dài bởi *eigenvalues*.

Nói chung, nếu chúng ta có thể tìm thấy một số λ và một vector $\mathbf{v}$ sao cho

$$\mathbf{Av} = \lambda\mathbf{v}. \tag{19.2.2}$$

Chúng tôi nói rằng $\mathbf{v}$ là một eigenvector cho A và λ là một eigenvalue.

[213] https://discuss.d2l.ai/t/1084

19.2.1 Tìm kiếm Eigenvalues Hãy để chúng tôi tìm ra cách tìm chúng. Bằng cách trừ $\lambda \mathbf{v}$ từ cả hai phía, và sau đó bao thanh toán ra vectơ, ta thấy ở trên tương đương với:

$$(\mathbf{A} - \lambda \mathbf{I})\mathbf{v} = 0. \tag{19.2.3}$$

Để (19.2.3) xảy ra, chúng ta thấy rằng $(\mathbf{A} - \lambda \mathbf{I})$ phải nén một số hướng xuống 0, do đó nó không thể đảo ngược, và do đó yếu tố quyết định bằng 0. Do đó, chúng ta có thể tìm thấy *eigenvalues* bằng cách tìm kiếm những gì λ là $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$. Khi chúng ta tìm thấy eigenvalues, chúng ta có thể giải $\mathbf{A}\mathbf{v} = \lambda \mathbf{v}$ để tìm (các) *eigenvector (s) * được liên kết.

Một ví dụ Hãy để chúng tôi xem điều này với một ma trận thách thức hơn

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \tag{19.2.4}$$

Nếu chúng ta xem xét $\det(\mathbf{A} - \lambda \mathbf{I}) = 0$, ta thấy điều này tương đương với phương trình đa thức $0 = (2 - \lambda)(3 - \lambda) - 2 = (4 - \lambda)(1 - \lambda)$. Do đó, hai eigenvalues là 4 và 1. Để tìm các vectơ liên quan, sau đó chúng ta cần phải giải quyết

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ and } \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \tag{19.2.5}$$

Chúng ta có thể giải quyết điều này với các vectơ $[1, -1]^\top$ và $[1, 2]^\top$ tương ứng.

Chúng ta có thể kiểm tra điều này trong mã bằng cách sử dụng thói quen `numpy.linalg.eig` tích hợp.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64),
          eigenvectors=True)
```

```
/tmp/ipykernel_121067/1653148152.py:6: UserWarning: torch.eig is deprecated␣
↪in favor of torch.linalg.eig and will be removed in a future PyTorch␣
↪release.
torch.linalg.eig returns complex tensors of dtype cfloat or cdouble rather␣
↪than real tensors mimicking complex tensors.
L, _ = torch.eig(A)
should be replaced with
L_complex = torch.linalg.eigvals(A)
and
L, V = torch.eig(A, eigenvectors=True)
should be replaced with
L_complex, V_complex = torch.linalg.eig(A) (Triggered internally at  ../aten/
↪src/ATen/native/BatchLinearAlgebra.cpp:2910.)
  torch.eig(torch.tensor([[2, 1], [2, 3]], dtype=torch.float64),
```

```
torch.return_types.eig(
eigenvalues=tensor([[1., 0.],
```

(continues on next page)

(continued from previous page)

```
        [4., 0.]], dtype=torch.float64),
eigenvectors=tensor([[-0.7071, -0.4472],
        [ 0.7071, -0.8944]], dtype=torch.float64))
```

Lưu ý rằng `numpy` bình thường hóa các eigenvectors có chiều dài một, trong khi chúng ta lấy chúng ta có chiều dài tùy ý. Ngoài ra, sự lựa chọn của dấu hiệu là tùy ý. Tuy nhiên, các vectơ được tính toán song song với các vectơ mà chúng ta tìm thấy bằng tay với cùng một giá trị eigenvalues.

19.2.2 Phân hủy ma trận Hãy để chúng tôi tiếp tục ví dụ trước một bước nữa. Hãy để

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \tag{19.2.6}$$

là ma trận nơi các cột là eigenvectors của ma trận $\mathbf{A}$. Hãy để

$$\mathbf{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \tag{19.2.7}$$

là ma trận với các giá trị eigenvalues liên quan trên đường chéo. Sau đó, định nghĩa của eigenvalues và eigenvectors nói với chúng ta rằng

$$\mathbf{AW} = \mathbf{W\Sigma}. \tag{19.2.8}$$

Ma trận W có thể đảo ngược, vì vậy chúng ta có thể nhân cả hai bên với W^{-1} ở bên phải, chúng ta thấy rằng chúng ta có thể viết

$$\mathbf{A} = \mathbf{W\Sigma W}^{-1}. \tag{19.2.9}$$

Trong phần tiếp theo, chúng ta sẽ thấy một số hậu quả tốt đẹp của điều này, nhưng bây giờ chúng ta chỉ cần biết rằng sự phân hủy như vậy sẽ tồn tại miễn là chúng ta có thể tìm thấy một bộ sưu tập đầy đủ các eigenvectors độc lập tuyến tính (để W là nghịch).

19.2.3 Hoạt động trên Eigendecomposition Một điều hay về eigendecomposition (19.2.9) là chúng ta có thể viết nhiều thao tác mà chúng ta thường gặp phải một cách sạch sẽ về sự phân hủy. Như một ví dụ đầu tiên, hãy xem xét:

$$\mathbf{A}^n = \overbrace{\mathbf{A}\cdots\mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W\Sigma W}^{-1})\cdots(\mathbf{W\Sigma W}^{-1})}^{n \text{ times}} = \mathbf{W}\overbrace{\mathbf{\Sigma}\cdots\mathbf{\Sigma}}^{n \text{ times}}\mathbf{W}^{-1} = \mathbf{W\Sigma}^n\mathbf{W}^{-1}. \tag{19.2.10}$$

Điều này cho chúng ta biết rằng đối với bất kỳ sức mạnh tích cực nào của ma trận, sự phân hủy eigencomposition thu được bằng cách chỉ nâng eigenvalues lên cùng một sức mạnh. Điều tương tự cũng có thể được hiển thị cho các quyền hạn tiêu cực, vì vậy nếu chúng ta muốn đảo ngược một ma trận, chúng ta chỉ cần xem xét

$$\mathbf{A}^{-1} = \mathbf{W\Sigma}^{-1}\mathbf{W}^{-1}, \tag{19.2.11}$$

or in other wordstừ ngữ, just invert đảo ngược each mỗi eigenvalue giá trị. Điều này sẽ hoạt động miễn là mỗi eigenvalue là không, vì vậy chúng ta thấy rằng invertible giống như không có eigenvalues bằng không.

Thật vậy, công việc bổ sung có thể cho thấy nếu $\lambda_1, \ldots, \lambda_n$ là giá trị eigenvalues của ma trận, thì yếu tố quyết định của ma trận đó là

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \tag{19.2.12}$$

hoặc sản phẩm của tất cả các giá trị eigenvalues. Điều này có ý nghĩa trực giác bởi vì bất cứ điều gì kéo dài $\mathbf{W}$ làm, W^{-1} hoàn tác nó, vì vậy cuối cùng sự kéo dài duy nhất xảy ra là nhân với ma trận chéo $\mathbf{\Sigma}$, kéo dài khối lượng theo sản phẩm của các yếu tố đường chéo.

Cuối cùng, nhớ lại rằng thứ hạng là số lượng tối đa các cột độc lập tuyến tính của ma trận của bạn. Bằng cách kiểm tra chặt chẽ eigendecomposition, chúng ta có thể thấy rằng thứ hạng giống như số eigenvalues không bằng không của $\mathbf{A}$.

Các ví dụ có thể tiếp tục, nhưng hy vọng điểm rõ ràng: eigendecomposition có thể đơn giản hóa nhiều tính toán tuyến tính-đại số và là một hoạt động cơ bản dựa trên nhiều thuật toán số và phần lớn phân tích mà chúng ta làm trong đại số tuyến tính.

19.2.4 Eigendecompositions of Symmetric Matrận Không phải lúc nào cũng có thể tìm thấy đủ các eigenvectơ độc lập tuyến tính cho quá trình trên hoạt động. For instance ví dụ the matrix ma trận

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \tag{19.2.13}$$

has only a single Độc thân eigenvector, cụ thể là $(1, 0)^\top$. Để xử lý các ma trận như vậy, chúng tôi yêu cầu các kỹ thuật tiên tiến hơn chúng ta có thể bao gồm (chẳng hạn như Dạng bình thường Jordan, hoặc Phân hủy giá trị số ít). Chúng ta thường sẽ cần phải hạn chế sự chú ý của mình đối với những ma trận nơi chúng ta có thể đảm bảo sự tồn tại của một bộ đầy đủ các eigenvectors.

Gia đình thường gặp nhất là * đối xứng matrices*, đó là những ma trận nơi $\mathbf{A} = \mathbf{A}^\top$. Trong trường hợp này, chúng ta có thể lấy W là một ma trận * trực giao ma trận có cột có tất cả chiều dài một vectơ ở góc vuông với nhau, trong đó $\mathbf{W}^\top = \mathbf{W}^{-1}$—và tất cả các giá trị eigenvalue sẽ là thật. Do đó, trong trường hợp đặc biệt này, chúng ta có thể viết (19.2.9) là

$$\mathbf{A} = \mathbf{W}\mathbf{\Sigma}\mathbf{W}^\top. \tag{19.2.14}$$

19.2.5 Định lý vòng tròn Gershgorin Eigenvalues thường khó lý luận với trực giác. Nếu trình bày một ma trận tùy ý, có rất ít điều có thể nói về những gì eigenvalues là mà không tính toán chúng. Tuy nhiên, có một định lý có thể làm cho nó dễ dàng gần đúng nếu các giá trị lớn nhất nằm trên đường chéo.

Hãy để $\mathbf{A} = (a_{ij})$ là bất kỳ ma trận vuông $(n \times n)$. Chúng tôi sẽ xác định $r_i = \sum_{j \neq i} |a_{ij}|$. Hãy để $\mathcal{D}_i$ đại diện cho đĩa trong mặt phẳng phức tạp với trung tâm a_{ii} bán kính r_i. Sau đó, mỗi eigenvalue của $\mathbf{A}$ được chứa trong một trong $\mathcal{D}_i$.

Điều này có thể là một chút để giải nén, vì vậy chúng ta hãy xem xét một ví dụ. Hãy xem xét ma trận:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \tag{19.2.15}$$

Chúng ta có $r_1 = 0.3$, $r_2 = 0.6$, $r_3 = 0.8$ và $r_4 = 0.9$. Ma trận là đối xứng, vì vậy tất cả các giá trị eigenvalues là có thật. Điều này có nghĩa là tất cả các giá trị eigenvalues của chúng tôi sẽ nằm trong một trong các phạm vi

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \tag{19.2.16}$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \tag{19.2.17}$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \tag{19.2.18}$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \tag{19.2.19}$$

Thực hiện tính toán số cho thấy các giá trị eigenvalues là khoảng 0.99, 2.97, 4.95, 9.08, tất cả đều thoải mái bên trong các phạm vi được cung cấp.

```
A = torch.tensor([[1.0, 0.1, 0.1, 0.1],
                  [0.1, 3.0, 0.2, 0.3],
                  [0.1, 0.2, 5.0, 0.5],
                  [0.1, 0.3, 0.5, 9.0]])

v, _ = torch.eig(A)
v
```

```
tensor([[0.9923, 0.0000],
        [9.0803, 0.0000],
        [4.9539, 0.0000],
        [2.9734, 0.0000]])
```

Bằng cách này, eigenvalues có thể được xấp xỉ và các xấp xỉ sẽ khá chính xác trong trường hợp đường chéo lớn hơn đáng kể so với tất cả các yếu tố khác.

Đó là một điều nhỏ, nhưng với một chủ đề phức tạp và tinh tế như eigendecomposition, thật tốt để có được bất kỳ sự nắm bắt trực quan nào chúng ta có thể.

19.2.6 Một ứng dụng hữu ích: Sự phát triển của bản đồ lặp

Bây giờ chúng ta hiểu nguyên tắc eigenvectors là gì, chúng ta hãy xem chúng có thể được sử dụng như thế nào để cung cấp một sự hiểu biết sâu sắc về một vấn đề trung tâm đến hành vi mạng thần kinh: khởi tạo trọng lượng thích hợp.

Eigenvectors như hành vi dài hạn

Cuộc điều tra toán học đầy đủ về việc khởi tạo các mạng thần kinh sâu nằm ngoài phạm vi của văn bản, nhưng chúng ta có thể thấy một phiên bản đồ chơi ở đây để hiểu cách thức eigenvalues có thể giúp chúng ta thấy các mô hình này hoạt động như thế nào. Như chúng ta đã biết, các mạng thần kinh hoạt động bằng cách xen kẽ các lớp biến đổi tuyến tính với các hoạt động phi tuyến tính. Để đơn giản ở đây, chúng ta sẽ giả định rằng không có phi tuyến tính và chuyển đổi là một hoạt động ma trận lặp lại duy nhất A, để đầu ra của mô hình của chúng tôi là

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A}\mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \tag{19.2.20}$$

Khi các mô hình này được khởi tạo, A được coi là một ma trận ngẫu nhiên với các mục Gaussian, vì vậy chúng ta hãy tạo một trong những mô hình đó. Để được cụ thể, chúng ta bắt đầu với một trung bình 0, phương sai một Gaussian phân phối 5×5 ma trận.

```
torch.manual_seed(42)

k = 5
A = torch.randn(k, k, dtype=torch.float64)
A
```

```
tensor([[ 0.2996,  0.2424,  0.2832, -0.2329,  0.6712],
        [ 0.7818, -1.7903, -1.7484,  0.1735, -0.1182],
        [-1.7446, -0.4695,  0.4573,  0.5177, -0.2771],
        [-0.6641,  0.6551,  0.2616, -1.5265, -0.3311],
        [-0.6378,  0.1072,  0.7096,  0.3009, -0.2869]], dtype=torch.float64)
```

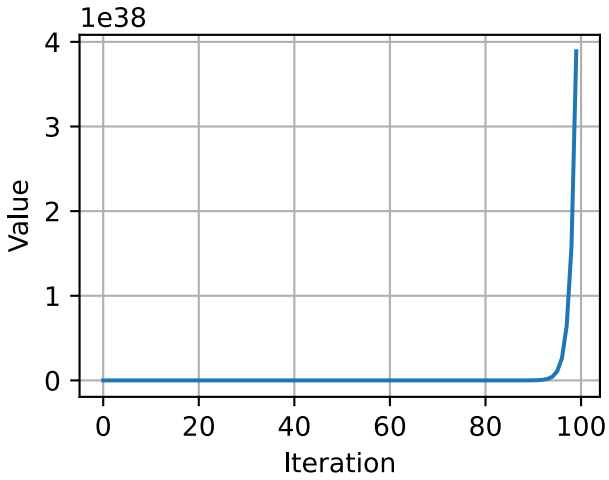
Hành vi trên dữ liệu ngẫu nhiên Để đơn giản trong mô hình đồ chơi của chúng ta, chúng ta sẽ giả định rằng vector dữ liệu chúng ta cung cấp trong $\mathbf{v}_{in}$ là một vector Gaussian năm chiều ngẫu nhiên. Chúng ta hãy suy nghĩ về những gì chúng ta muốn có xảy ra. Đối với bối cảnh, hãy nghĩ về một vấn đề ML chung, nơi chúng ta đang cố gắng biến dữ liệu đầu vào, giống như một hình ảnh, thành một dự đoán, giống như xác suất hình ảnh là một hình ảnh của một con mèo. Nếu ứng dụng lặp đi lặp lại của $\mathbf{A}$ kéo dài một vectơ ngẫu nhiên ra rất dài, thì những thay đổi nhỏ trong đầu vào sẽ được khuếch đại thành những thay đổi lớn trong đầu ra—những sửa đổi nhỏ của hình ảnh đầu vào sẽ dẫn đến những dự đoán rất khác nhau. Điều này dường như không đúng!

Mặt trái, nếu $\mathbf{A}$ thu nhỏ các vectơ ngẫu nhiên để ngắn hơn, thì sau khi chạy qua nhiều lớp, vectơ về cơ bản sẽ co lại thành không có gì, và đầu ra sẽ không phụ thuộc vào đầu vào. Điều này cũng rõ ràng cũng không đúng!

Chúng ta cần phải đi bộ đường hẹp giữa tăng trưởng và phân rã để đảm bảo rằng đầu ra của chúng tôi thay đổi tùy thuộc vào đầu vào của chúng tôi, nhưng không nhiều!

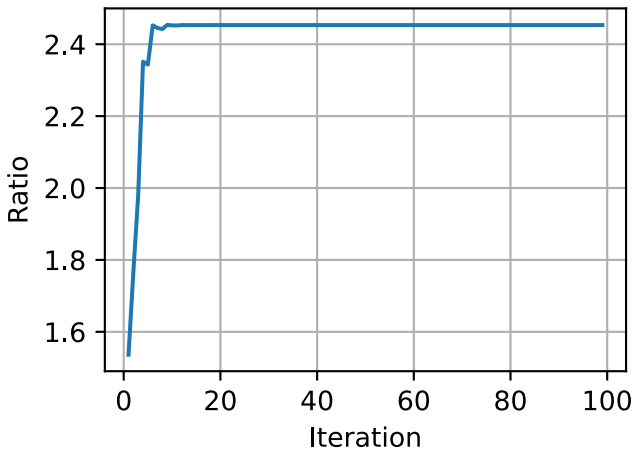
Chúng ta hãy xem những gì xảy ra khi chúng ta nhiều lần nhân ma trận của chúng ta $\mathbf{A}$ chống lại một vector đầu vào ngẫu nhiên, và theo dõi định mức.

```
# Calculate the sequence of norms after repeatedly applying `A`
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



Định mức đang phát triển không kiểm soát được! Thật vậy nếu chúng ta lấy danh sách các thương, chúng ta sẽ thấy một mô hình.

```
# Compute the scaling factor of the norms
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i - 1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



Nếu chúng ta nhìn vào phần cuối cùng của tính toán trên, chúng ta thấy rằng vectơ ngẫu nhiên được kéo dài bởi một hệ số `1.974459321485[...]`, trong đó phần ở cuối thay đổi một chút, nhưng yếu tố kéo dài ổn định.

Liên quan Trở lại Eigenvectors

Chúng ta đã thấy rằng eigenvectors và eigenvalues tương ứng với số lượng một cái gì đó được kéo dài, nhưng đó là cho vectơ cụ thể, và các trải dài cụ thể. Chúng ta hãy xem chúng là gì cho **A**. Một chút cảnh báo ở đây: hóa ra để xem tất cả, chúng ta sẽ cần phải đi đến các số phức. Bạn có thể nghĩ về những điều này như kéo dài và quay. Bằng cách lấy định mức của số phức (căn bậc hai của tổng các ô vuông của các phần thực và tưởng tượng), chúng ta có thể đo được yếu tố kéo dài đó. Hãy để chúng tôi cũng sắp xếp chúng.

```
# Compute the eigenvalues
eigs = torch.eig(A)[0][:,0].tolist()
norm_eigs = [torch.abs(torch.tensor(x)) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')
```

```
norms of eigenvalues: [tensor(0.3490), tensor(0.5691), tensor(0.5691),␣
↪tensor(1.1828), tensor(2.4532)]
```

Một quan sát

Chúng ta thấy một điều gì đó một chút bất ngờ xảy ra ở đây: số đó chúng tôi đã xác định trước đây cho kéo dài dài hạn ma trận của chúng tôi **A** áp dụng cho một vector ngẫu nhiên là * chính xác* (chính xác đến mười ba chữ số thập phân!) the largest lớn nhất eigenvalue giá trị of **A**. Đây rõ ràng không phải là một sự trùng hợp ngẫu nhiên!

Nhưng, nếu bây giờ chúng ta nghĩ về những gì đang xảy ra về mặt hình học, điều này bắt đầu có ý nghĩa. Hãy xem xét một vector ngẫu nhiên. Vector ngẫu nhiên này chỉ một chút theo mọi hướng, vì vậy đặc biệt, nó chỉ ít nhất một chút theo cùng hướng với eigenvector của **A** liên quan đến eigenvalue lớn nhất. Điều này quan trọng đến mức nó được gọi là *nguyên tắc eigenvalue* và *nguyên tắc eigenvector*. Sau khi áp dụng **A**, vector ngẫu nhiên của chúng ta được kéo dài theo mọi hướng có thể, như được liên kết với mọi eigenvector có thể, nhưng nó được kéo dài hầu hết theo hướng liên quan đến nguyên tắc này eigenvector. Điều này có nghĩa là sau khi áp dụng trong A, vector ngẫu nhiên của chúng ta dài hơn, và các điểm theo hướng gần hơn để được liên kết với nguyên tắc eigenvector. Sau khi áp dụng ma trận nhiều lần, sự liên kết với nguyên tắc eigenvector trở nên gần gũi hơn và gần hơn cho đến khi, cho tất cả các mục đích thực tế, vector ngẫu nhiên của chúng ta đã được chuyển thành nguyên tắc eigenvector! Thật vậy, thuật toán này là cơ sở cho cái được gọi là lần lặp điện* để tìm giá trị eigenvalue và eigenvector lớn nhất của ma trận. Để biết chi tiết, hãy xem, ví dụ, (VanLoan & Golub, 1983).

Sửa chữa bình thường hóa

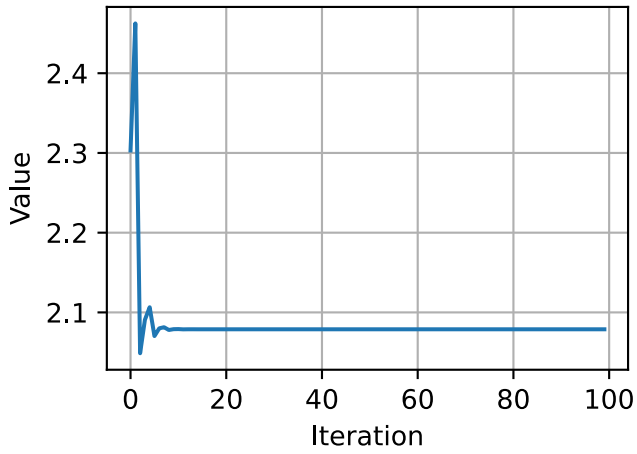
Bây giờ, từ các cuộc thảo luận trên, chúng tôi kết luận rằng chúng tôi không muốn một vector ngẫu nhiên được kéo dài hoặc squished ở tất cả, chúng tôi muốn vectơ ngẫu nhiên ở lại với cùng một kích thước trong toàn bộ quá trình. Để làm như vậy, bây giờ chúng ta giải phóng ma trận của chúng tôi bằng nguyên tắc eigenvalue này để eigenvalue lớn nhất thay vào đó bây giờ chỉ là một. Hãy để chúng tôi xem những gì xảy ra trong trường hợp này.

```
# Rescale the matrix `A`
A /= norm_eigs[-1]

# Do the same experiment again
```

(continues on next page)

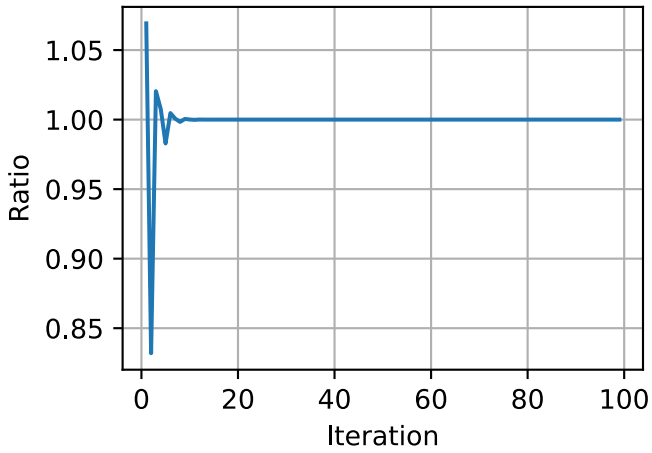
(continued from previous page)

```
v_in = torch.randn(k, 1, dtype=torch.float64)

norm_list = [torch.norm(v_in).item()]
for i in range(1, 100):
    v_in = A @ v_in
    norm_list.append(torch.norm(v_in).item())

d2l.plot(torch.arange(0, 100), norm_list, 'Iteration', 'Value')
```



Chúng ta cũng có thể vẽ tỷ lệ giữa các định mức liên tiếp như trước và thấy rằng thực sự nó ổn định.

```
# Also plot the ratio
norm_ratio_list = []
for i in range(1, 100):
    norm_ratio_list.append(norm_list[i]/norm_list[i-1])

d2l.plot(torch.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```



19.2.7 Kết luận

Bây giờ chúng ta thấy chính xác những gì chúng tôi hy vọng! Sau khi bình thường hóa các ma trận theo nguyên tắc eigenvalue, chúng ta thấy rằng dữ liệu ngẫu nhiên không phát nổ như trước, mà cuối cùng cân bằng với một giá trị cụ thể. Nó sẽ là tốt đẹp để có thể làm những điều này từ các nguyên tắc đầu tiên, và nó chỉ ra rằng nếu chúng ta nhìn sâu vào toán học của nó, chúng ta có thể thấy rằng eigenvalue lớn nhất của một ma trận ngẫu nhiên lớn với trung bình zero độc lập, phương sai một mục Gaussian là trung bình khoảng $\sqrt{n}$, hoặc trong trường hợp của chúng tôi $\sqrt{5} \approx 2.2$, do một thực tế hấp dẫn được gọi là luật thông tư*:cite:`Ginibre.1965`. Mối quan hệ giữa các eigenvalues (và một đối tượng liên quan gọi là giá trị số ít) của ma trận ngẫu nhiên đã được chứng minh là có các kết nối sâu sắc để khởi tạo thích hợp các mạng thần kinh như đã được thảo luận trong (Pennington et al., 2017) và các tác phẩm tiếp theo.

19.2.8 Tóm tắt * Eigenvectors là các vectơ được kéo dài bởi một ma trận mà không thay đổi hướng. * Eigenvalues là số tiền mà các eigenvectors được kéo dài bởi ứng dụng của ma trận. * Sự phân hủy của ma trận có thể cho phép nhiều phép toán được giảm xuống các hoạt động trên eigenvalues. Định lý vòng tròn Gershgorin có thể cung cấp các giá trị gần đúng cho eigenvalues của ma trận * hành vi của các công suất ma trận lặp lại phụ thuộc chủ yếu vào kích thước của eigenvalue lớn nhất. Sự hiểu biết này có nhiều ứng dụng trong lý thuyết khởi tạo mạng thần kinh.

Exercises

1. What are the eigenvalues and eigenvectors of

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}? \tag{19.2.21}$$

2. What are the eigenvalues and eigenvectors of the following matrix, and what is strange about this example compared to the previous one?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix}. \tag{19.2.22}$$

3. Without computing the eigenvalues, is it possible that the smallest eigenvalue of the following matrix is less that 0.5? *Note*: this problem can be done in your head.

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix}. \tag{19.2.23}$$

Discussions[214]

[214] https://discuss.d2l.ai/t/1086

19.3 Tính toán biến đơn

Năm Section 3.4, chúng ta đã thấy các yếu tố cơ bản của phép tính vi phân. Phần này sẽ tìm hiểu sâu hơn về các nguyên tắc cơ bản của giải tích và cách chúng ta có thể hiểu và áp dụng nó trong bối cảnh học máy.

19.3.1 Vi phân Giải tích vi phân về cơ bản là nghiên cứu về cách các chức năng hoạt động dưới những thay đổi nhỏ. Để xem lý do tại sao điều này là cốt lõi để học sâu, chúng ta hãy xem xét một ví dụ.

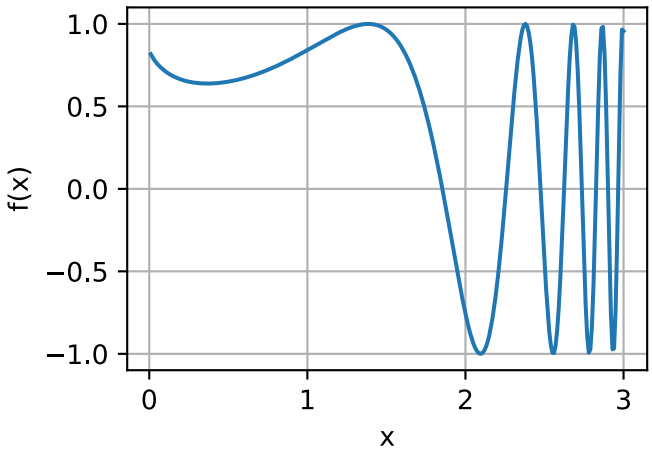
Giả sử rằng chúng ta có một mạng thần kinh sâu, nơi các trọng lượng, để thuận tiện, được nối thành một vector $\mathbf{w} = (w_1, \ldots, w_n)$ duy nhất. Với một tập dữ liệu đào tạo, chúng tôi xem xét việc mất mạng thần kinh của chúng tôi trên tập dữ liệu này, mà chúng tôi sẽ viết là $\mathcal{L}(\mathbf{w})$.

Chức năng này cực kỳ phức tạp, mã hóa hiệu suất của tất cả các mô hình có thể có của kiến trúc đã cho trên tập dữ liệu này, vì vậy gần như không thể biết bộ trọng lượng $\mathbf{w}$ sẽ giảm thiểu tổn thất. Do đó, trong thực tế, chúng ta thường bắt đầu bằng cách khởi tạo trọng lượng* ngẫu nhiên*, và sau đó lặp đi lặp lại các bước nhỏ theo hướng làm cho tổn thất giảm càng nhanh càng tốt.

Câu hỏi sau đó trở thành một cái gì đó mà trên bề mặt không dễ dàng hơn: làm thế nào để chúng ta tìm thấy hướng làm cho trọng lượng giảm càng nhanh càng tốt? Để đào sâu vào điều này, trước tiên chúng ta hãy kiểm tra trường hợp chỉ với một trọng lượng duy nhất: $L(\mathbf{w}) = L(x)$ cho một giá trị thực duy nhất x.

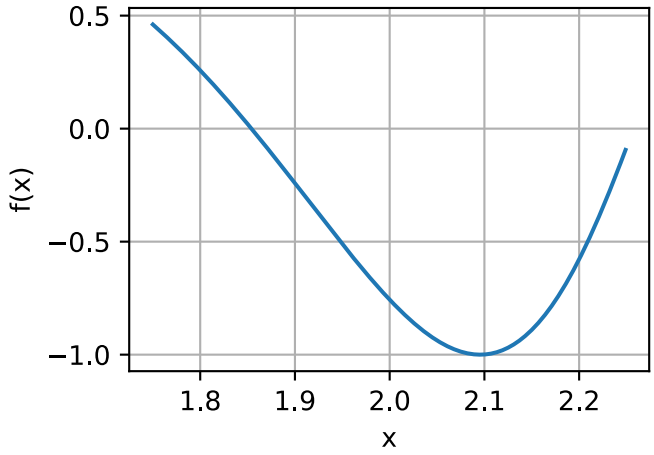
Chúng ta hãy lấy x và cố gắng hiểu những gì sẽ xảy ra khi chúng ta thay đổi nó bằng một lượng nhỏ thành $x + \epsilon$. Nếu bạn muốn cụ thể, hãy nghĩ một con số như $\epsilon = 0.0000001$. Để giúp chúng ta hình dung những gì xảy ra, chúng ta hãy biểu đồ một hàm ví dụ, $f(x) = \sin(x^x)$, trên $[0, 3]$.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

# Plot a function in a normal range
x_big = torch.arange(0.01, 3.01, 0.01)
ys = torch.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```



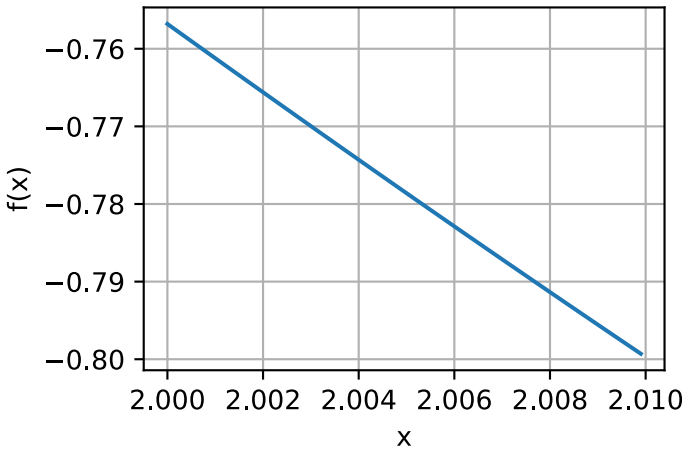
Ở quy mô lớn này, hành vi của hàm không đơn giản. Tuy nhiên, nếu chúng ta giảm phạm vi của mình xuống một cái gì đó nhỏ hơn như $[1.75, 2.25]$, chúng ta thấy rằng biểu đồ trở nên đơn giản hơn nhiều.

```
# Plot a the same function in a tiny range
x_med = torch.arange(1.75, 2.25, 0.001)
ys = torch.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```



Đưa điều này đến cực đoan, nếu chúng ta phóng to thành một đoạn nhỏ, hành vi trở nên đơn giản hơn nhiều: nó chỉ là một đường thẳng.

```
# Plot a the same function in a tiny range
x_small = torch.arange(2.0, 2.01, 0.0001)
ys = torch.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```



Đây là quan sát chính của phép tính biến đơn lẻ: hành vi của các hàm quen thuộc có thể được mô hình hóa bởi một dòng trong một phạm vi đủ nhỏ. Điều này có nghĩa là đối với hầu hết các chức năng, thật hợp lý khi mong đợi rằng khi chúng ta thay đổi giá trị x của hàm một chút, đầu ra $f(x)$ cũng sẽ được dịch chuyển một chút. Câu hỏi duy nhất chúng ta cần trả lời là, "Sự thay đổi trong đầu ra lớn bao nhiêu so với sự thay đổi trong đầu vào? Nó có lớn một nửa không? Gấp đôi lớn?"

Do đó, chúng ta có thể xem xét tỷ lệ thay đổi trong đầu ra của một hàm cho một sự thay đổi nhỏ trong đầu vào của hàm. Chúng tôi có thể viết điều này chính thức như

$$\frac{L(x+\epsilon) - L(x)}{(x+\epsilon) - x} = \frac{L(x+\epsilon) - L(x)}{\epsilon}. \tag{19.3.1}$$

Điều này đã đủ để bắt đầu chơi xung quanh với mã. Ví dụ, giả sử rằng chúng ta biết rằng $L(x) = x^2 + 1701(x-4)^3$, sau đó chúng ta có thể thấy giá trị này lớn như thế nào tại điểm $x = 4$ như sau.

```
# Define our function
def L(x):
    return x**2 + 1701*(x-4)**3

# Print the difference divided by epsilon for several epsilon
for epsilon in [0.1, 0.001, 0.0001, 0.00001]:
    print(f'epsilon = {epsilon:.5f} -> {(L(4+epsilon) - L(4)) / epsilon:.5f}')
```

```
epsilon = 0.10000 -> 25.11000
epsilon = 0.00100 -> 8.00270
epsilon = 0.00010 -> 8.00012
epsilon = 0.00001 -> 8.00001
```

Bây giờ, nếu chúng ta quan sát, chúng ta sẽ nhận thấy rằng đầu ra của con số này là đáng ngờ gần với 8. Thật vậy, nếu chúng ta giảm ϵ, chúng ta sẽ thấy giá trị trở nên dần dần gần hơn với 8. Vì vậy, chúng ta có thể kết luận, một cách chính xác, rằng giá trị chúng ta tìm kiếm (mức độ thay đổi trong đầu vào thay đổi đầu ra) nên là 8 tại điểm $x = 4$. Cách mà một nhà toán học mã hóa thực tế này là

$$\lim_{\epsilon \to 0} \frac{L(4+\epsilon) - L(4)}{\epsilon} = 8. \tag{19.3.2}$$

Như một chút của một cuộc di truyền lịch sử: trong vài thập kỷ đầu tiên của nghiên cứu mạng thần kinh, các nhà khoa học đã sử dụng thuật toán này (phương pháp * của sự khác biệt hữu hạn *) để đánh giá làm thế nào một chức năng mất thay đổi theo nhiễu loạn nhỏ: chỉ cần thay đổi trọng lượng và xem mất thay đổi như thế nào. Đây là tính toán không hiệu quả, đòi hỏi hai đánh giá của hàm mất để xem một thay đổi duy nhất của một biến ảnh hưởng đến tổn thất như thế nào. Nếu chúng tôi cố gắng làm điều này với thậm chí một vài nghìn tham số nhẹ, nó sẽ yêu cầu vài nghìn đánh giá của mạng trên toàn bộ dữ liệu! Nó không được giải quyết cho đến năm 1986 rằng thuật toán *backpropagation* được giới thiệu trong :cite:`Rumelhart.Hinton.Williams.ea.1988` cung cấp một cách để tính toán cách *any* thay đổi trọng số cùng nhau sẽ thay đổi tổn thất trong cùng một thời gian tính toán như một dự đoán duy nhất của mạng qua tập dữ liệu.

Trở lại trong ví dụ của chúng tôi, giá trị này 8 là khác nhau cho các giá trị khác nhau của x, vì vậy nó có ý nghĩa để xác định nó như một hàm của x. Chính thức hơn, tỷ lệ thay đổi phụ thuộc vào giá trị này được gọi là *phái sinh* được viết là

$$\frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{19.3.3}$$

Các văn bản khác nhau sẽ sử dụng các ký hiệu khác nhau cho đạo hàm. Ví dụ, tất cả các ký hiệu dưới đây chỉ ra điều tương tự:

$$\frac{df}{dx} = \frac{d}{dx}f = f' = \nabla_x f = D_x f = f_x. \tag{19.3.4}$$

Hầu hết các tác giả sẽ chọn một ký hiệu duy nhất và gắn bó với nó, tuy nhiên ngay cả điều đó không được đảm bảo. Tốt nhất là làm quen với tất cả những điều này. Chúng ta sẽ sử dụng ký hiệu $\frac{df}{dx}$ trong suốt văn bản này,

trừ khi chúng ta muốn lấy đạo hàm của một biểu thức phức tạp, trong trường hợp đó chúng ta sẽ sử dụng $\frac{d}{dx}f$ để viết các biểu thức như

$$fracddx\,left[x^4 +\ cos\,left(\ fracx^2 + 12x - 1\,right)\,right]. \tag{19.3.5}$$

Thông thường, nó là trực giác hữu ích để làm sáng tỏ định nghĩa của đạo hàm (19.3.3) một lần nữa để xem một hàm thay đổi như thế nào khi chúng ta thực hiện một thay đổi nhỏ x:

$$\begin{aligned} \frac{df}{dx}(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon} &\implies \frac{df}{dx}(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \\ &\implies \epsilon \frac{df}{dx}(x) \approx f(x+\epsilon) - f(x) \\ &\implies f(x+\epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x). \end{aligned} \tag{19.3.6}$$

Phương trình cuối cùng đáng để gọi ra một cách rõ ràng. Nó cho chúng ta biết rằng nếu bạn lấy bất kỳ hàm nào và thay đổi đầu vào bằng một lượng nhỏ, đầu ra sẽ thay đổi theo số lượng nhỏ đó được thu nhỏ theo đạo hàm.

Bằng cách này, chúng ta có thể hiểu đạo hàm là hệ số mở rộng cho chúng ta biết thay đổi lớn như thế nào chúng ta nhận được trong đầu ra từ một sự thay đổi trong đầu vào.

19.3.2 Quy tắc của Calculus

Bây giờ chúng ta chuyển sang nhiệm vụ hiểu làm thế nào để tính toán đạo hàm của một hàm rõ ràng. Một điều trị chính thức đầy đủ của giải tích sẽ lấy được tất cả mọi thứ từ các nguyên tắc đầu tiên. Chúng tôi sẽ không thưởng thức sự cám dỗ này ở đây, mà là cung cấp một sự hiểu biết về các quy tắc chung gặp phải.

Các dẫn xuất phổ biến Như đã thấy trong Section 3.4, khi các dẫn xuất tính toán người ta thường có thể sử dụng một loạt các quy tắc để giảm tính toán cho một vài chức năng cốt lõi. Chúng tôi lặp lại chúng ở đây để dễ tham khảo.

- ** Derivative của constants.** $\frac{d}{dx}c = 0$.
- ** Derivative của các chức năng tuyến tính** $\frac{d}{dx}(ax) = a$.
- ** Quy tắc điện.** $\frac{d}{dx}x^n = nx^{n-1}$.
- ****Derivative của số mũ. ** $\frac{d}{dx}e^x = e^x$.
- ** Derivative của logarit** $\frac{d}{dx}\log(x) = \frac{1}{x}$.

Quy tắc phái sinh Nếu mọi đạo hàm cần được tính toán riêng và lưu trữ trong một bảng, phép tính vi phân sẽ gần như không thể. Đó là một món quà của toán học mà chúng ta có thể khái quát hóa các dẫn xuất trên và tính toán các dẫn xuất phức tạp hơn như tìm ra đạo hàm của $f(x) = \log\left(1 + (x-1)^{10}\right)$. Như đã đề cập trong Section 3.4, chìa khóa để làm như vậy là mã hóa những gì xảy ra khi chúng ta thực hiện các chức năng và kết hợp chúng theo nhiều cách khác nhau, quan trọng nhất là: tổng, sản phẩm và thành phần.

- **Sum rule.** $\frac{d}{dx}\left(g(x) + h(x)\right) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$.
- ** Quy tắc sản phẩm.** $\frac{d}{dx}\left(g(x) \cdot h(x)\right) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$.

- ** Quy tắc dây chuyền.** $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x)) \cdot \frac{dh}{dx}(x)$.

Hãy để chúng tôi xem cách chúng tôi có thể sử dụng (19.3.6) để hiểu các quy tắc này. Đối với quy tắc tổng, hãy xem xét chuỗi lý luận sau:

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) + h(x+\epsilon) \\ &\approx g(x) + \epsilon\frac{dg}{dx}(x) + h(x) + \epsilon\frac{dh}{dx}(x) \\ &= g(x) + h(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right) \\ &= f(x) + \epsilon\left(\frac{dg}{dx}(x) + \frac{dh}{dx}(x)\right). \end{aligned} \tag{19.3.7}$$

Bằng cách so sánh kết quả này với thực tế là $f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x)$, chúng ta thấy rằng $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ như mong muốn. Trực giác ở đây là: khi chúng ta thay đổi đầu vào x, g và h cùng góp phần vào sự thay đổi đầu ra của $\frac{dg}{dx}(x)$ và $\frac{dh}{dx}(x)$.

Sản phẩm tinh tế hơn và sẽ yêu cầu một quan sát mới về cách làm việc với các biểu thức này. Chúng tôi sẽ bắt đầu như trước khi sử dụng (19.3.6):

$$\begin{aligned} f(x+\epsilon) &= g(x+\epsilon) \cdot h(x+\epsilon) \\ &\approx \left(g(x) + \epsilon\frac{dg}{dx}(x)\right) \cdot \left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &= g(x) \cdot h(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x) \\ &= f(x) + \epsilon\left(g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)\right) + \epsilon^2\frac{dg}{dx}(x)\frac{dh}{dx}(x). \end{aligned} \tag{19.3.8}$$

Điều này giống như tính toán được thực hiện ở trên, và thực sự chúng ta thấy câu trả lời của chúng tôi ($\frac{df}{dx}(x) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$) ngồi bên cạnh ϵ, nhưng có vấn đề về thuật ngữ đó có kích thước ϵ^2. Chúng tôi sẽ đề cập đến điều này là một điều khoản * bậc cao hơn*, vì sức mạnh của ϵ^2 cao hơn sức mạnh của ϵ^1. Chúng ta sẽ thấy trong một phần sau mà đôi khi chúng ta sẽ muốn theo dõi những điều này, tuy nhiên bây giờ quan sát thấy rằng nếu $\epsilon = 0.0000001$, thì $\epsilon^2 = 0.0000000000001$, nhỏ hơn rất nhiều. Khi chúng tôi gửi $\epsilon \to 0$, chúng tôi có thể bỏ qua một cách an toàn các điều khoản đặt hàng cao hơn. Như một quy ước chung trong phụ lục này, chúng ta sẽ sử dụng "≈" để biểu thị rằng hai thuật ngữ này bằng với các điều khoản bậc cao hơn. Tuy nhiên, nếu chúng ta muốn chính thức hơn, chúng ta có thể kiểm tra sự khác biệt thương

$$\frac{f(x+\epsilon) - f(x)}{\epsilon} = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon\frac{dg}{dx}(x)\frac{dh}{dx}(x), \tag{19.3.9}$$

và thấy rằng khi chúng tôi gửi $\epsilon \to 0$, thuật ngữ tay phải cũng đi đến không.

Cuối cùng, với quy tắc chuỗi, chúng ta có thể tiến bộ một lần nữa như trước khi sử dụng (19.3.6) và thấy rằng

$$\begin{aligned} f(x+\epsilon) &= g(h(x+\epsilon)) \\ &\approx g\left(h(x) + \epsilon\frac{dh}{dx}(x)\right) \\ &\approx g(h(x)) + \epsilon\frac{dh}{dx}(x)\frac{dg}{dh}(h(x)) \\ &= f(x) + \epsilon\frac{dg}{dh}(h(x))\frac{dh}{dx}(x), \end{aligned} \tag{19.3.10}$$

trong đó trong dòng thứ hai, chúng tôi xem hàm g là có đầu vào của nó ($h(x)$) được dịch chuyển bởi số lượng nhỏ $\epsilon\frac{dh}{dx}(x)$.

Quy tắc này cung cấp cho chúng ta một bộ công cụ linh hoạt để tính toán về cơ bản bất kỳ biểu thức nào mong muốn. For instance ví dụ,

$$\begin{aligned}\frac{d}{dx}\left[\log\left(1+(x-1)^{10}\right)\right] &= \left(1+(x-1)^{10}\right)^{-1}\frac{d}{dx}\left[1+(x-1)^{10}\right] \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(\frac{d}{dx}[1]+\frac{d}{dx}[(x-1)^{10}]\right) \\ &= \left(1+(x-1)^{10}\right)^{-1}\left(0+10(x-1)^{9}\frac{d}{dx}[x-1]\right) \\ &= 10\left(1+(x-1)^{10}\right)^{-1}(x-1)^{9} \\ &= \frac{10(x-1)^{9}}{1+(x-1)^{10}}.\end{aligned} \tag{19.3.11}$$

Trong trường hợp mỗi dòng đã sử dụng các quy tắc sau:

1. Quy tắc chuỗi và đạo hàm của logarit.
2. Quy tắc tổng.
3. Đạo hàm của hằng số, quy tắc chuỗi, và quy tắc quyền lực.
4. Quy tắc tổng, đạo hàm của các hàm tuyến tính, đạo hàm của hằng số.

Hai điều nên rõ ràng sau khi thực hiện ví dụ này:

1. Bất kỳ chức năng nào chúng ta có thể viết ra bằng cách sử dụng tổng, sản phẩm, hằng số, quyền hạn, số mũ và logarit có thể có đạo hàm của nó được tính toán một cách cơ học bằng cách tuân theo các quy tắc này.
2. Có một con người tuân theo các quy tắc này có thể tẻ nhạt và dễ bị lỗi!

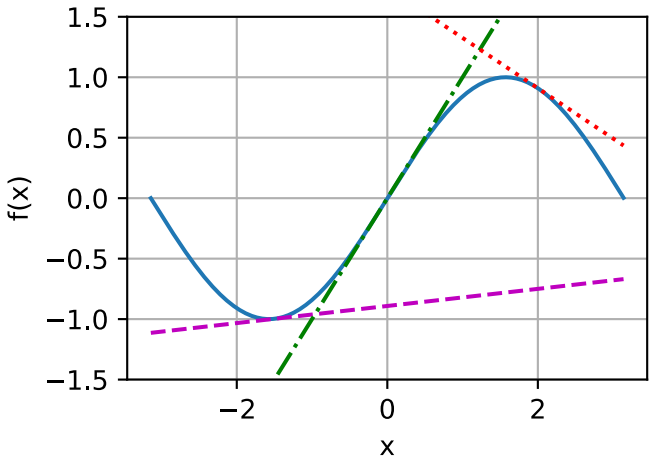
Rất may, hai sự kiện này cùng nhau gợi ý hướng tới một con đường phía trước: đây là một ứng cử viên hoàn hảo cho cơ giới hóa! Thật vậy, tuyên truyền ngược, mà chúng ta sẽ xem lại sau trong phần này, chính xác là điều đó.

Xấp xỉ tuyến tính Khi làm việc với các dẫn xuất, nó thường hữu ích để giải thích hình học xấp xỉ được sử dụng ở trên. Đặc biệt, lưu ý rằng phương trình

$$f(x+\epsilon) \approx f(x) + \epsilon\frac{df}{dx}(x), \tag{19.3.12}$$

xấp xỉ giá trị của f bởi một đường đi qua điểm $(x, f(x))$ và có độ dốc $\frac{df}{dx}(x)$. Bằng cách này, chúng tôi nói rằng đạo hàm đưa ra một xấp xỉ tuyến tính cho hàm f, như minh họa dưới đây:

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some linear approximations. Use d(sin(x))/dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)))

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Cao hơn Thiết bị trật tự

Bây giờ chúng ta hãy làm một cái gì đó có thể trên bề mặt có vẻ lạ. Lấy một hàm f và tính toán đạo hàm $\frac{df}{dx}$. Điều này cho chúng ta tỷ lệ thay đổi f tại bất kỳ điểm nào.

Tuy nhiên, đạo hàm, $\frac{df}{dx}$, có thể được xem như một hàm chính nó, vì vậy không có gì ngăn cản chúng ta tính toán đạo hàm của $\frac{df}{dx}$ để có được $\frac{d^2f}{dx^2} = \frac{df}{dx}\left(\frac{df}{dx}\right)$. Chúng ta sẽ gọi đây là đạo hàm thứ hai của f. Chức năng này là tốc độ thay đổi tỷ lệ thay đổi f, hay nói cách khác, tốc độ thay đổi đang thay đổi như thế nào. Chúng tôi có thể áp dụng đạo hàm bất kỳ số lần nào để có được cái được gọi là đạo hàm n-th. Để giữ cho ký hiệu sạch sẽ, chúng ta sẽ biểu thị đạo hàm n-th là

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{19.3.13}$$

Hãy để chúng tôi cố gắng hiểu * tại sao * đây là một khái niệm hữu ích. Dưới đây, chúng tôi hình dung $f^{(2)}(x)$, $f^{(1)}(x)$ và $f(x)$.

Đầu tiên, hãy xem xét trường hợp đạo hàm thứ hai $f^{(2)}(x)$ là một hằng số dương. Điều này có nghĩa là độ dốc của đạo hàm đầu tiên là dương. Kết quả là, đạo hàm đầu tiên $f^{(1)}(x)$ có thể bắt đầu âm, trở thành 0 tại một điểm, và sau đó trở thành dương cuối cùng. Điều này cho chúng ta biết độ dốc của chức năng ban đầu của chúng tôi f và do đó, chức năng f tự giảm, làm phẳng, sau đó tăng lên. Nói cách khác, hàm f cong lên, và có một mức tối thiểu duy nhất như được hiển thị trong Fig. 19.3.1.

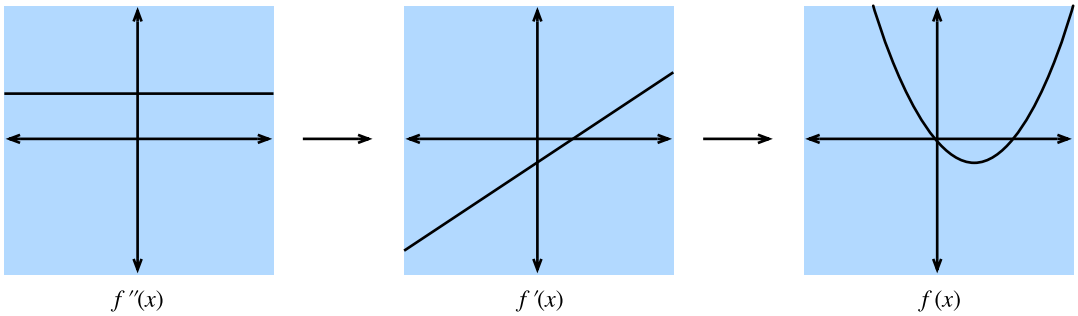


Fig. 19.3.1: If we assume the second derivative is a positive constant, then the fist derivative in increasing, which implies the function itself has a minimum.

Thứ hai, nếu đạo hàm thứ hai là hằng số âm, điều đó có nghĩa là đạo hàm thứ nhất đang giảm. Điều này ngụ ý đạo hàm đầu tiên có thể bắt đầu dương, trở thành 0 tại một điểm, và sau đó trở thành âm. Do đó, chức năng f tự tăng lên, làm phẳng ra, sau đó giảm. Nói cách khác, hàm f cong xuống, và có một tối đa duy nhất như được thể hiện trong Fig. 19.3.2.

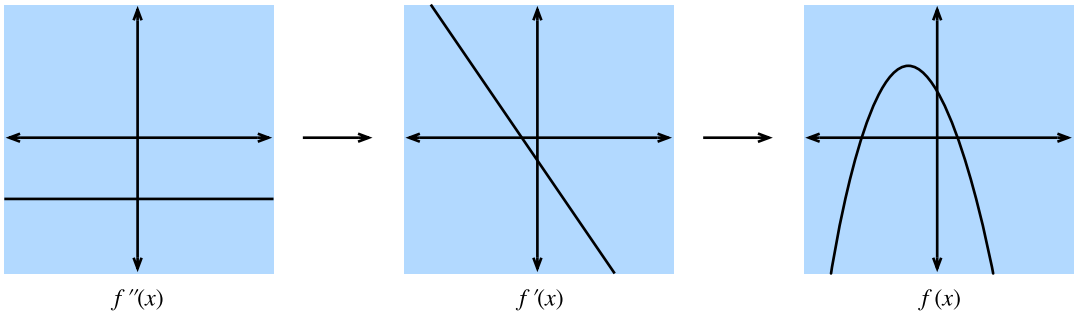


Fig. 19.3.2: If we assume the second derivative is a negative constant, then the fist derivative in decreasing, which implies the function itself has a maximum.

Thứ ba, nếu đạo hàm thứ hai là một luôn bằng 0, thì đạo hàm đầu tiên sẽ không bao giờ thay đổi—nó là hằng số! Điều này có nghĩa là f tăng (hoặc giảm) với tốc độ cố định và f bản thân nó là một đường thẳng như thể hiện trong Fig. 19.3.3.

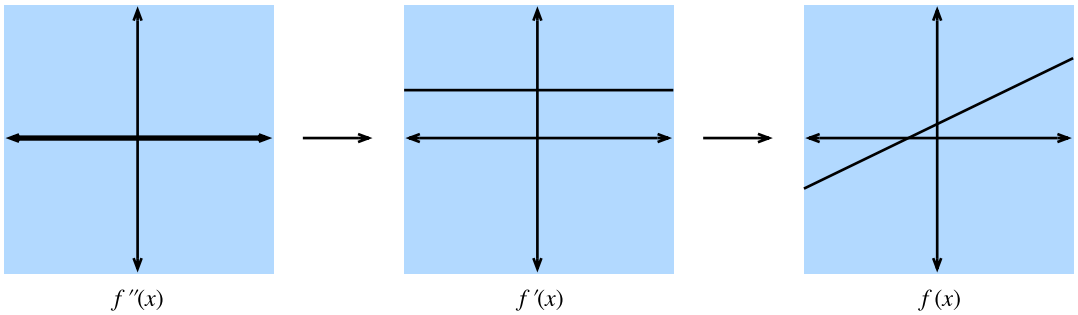


Fig. 19.3.3: If we assume the second derivative is zero, then the fist derivative is constant, which implies the function itself is a straight line.

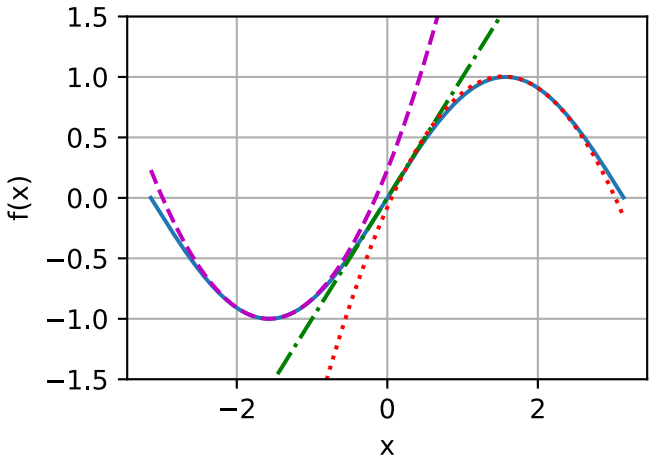
Tóm lại, đạo hàm thứ hai có thể được hiểu là mô tả cách hàm f đường cong. Một đạo hàm thứ hai dương dẫn đến một đường cong lên trên, trong khi đạo hàm thứ hai âm có nghĩa là f đường cong xuống dưới, và đạo hàm thứ hai bằng 0 có nghĩa là f không cong chút nào.

Hãy để chúng tôi thực hiện điều này thêm một bước nữa. Hãy xem xét chức năng $g(x) = ax^2 + bx + c$. We can then computetính toán that

$$\begin{aligned}\frac{dg}{dx}(x) &= 2ax + b \\ \frac{d^2g}{dx^2}(x) &= 2a.\end{aligned} \tag{19.3.14}$$

Nếu chúng ta có một số hàm gốc $f(x)$ trong tâm trí, chúng ta có thể tính toán hai dẫn xuất đầu tiên và tìm các giá trị cho a, b, và c mà làm cho chúng phù hợp với tính toán này. Tương tự như phần trước, nơi chúng ta thấy rằng đạo hàm đầu tiên đã đưa ra xấp xỉ tốt nhất với một đường thẳng, cấu trúc này cung cấp xấp xỉ tốt nhất bằng một bậc hai. Hãy để chúng tôi hình dung điều này cho $f(x) = \sin(x)$.

```
# Compute sin
xs = torch.arange(-torch.pi, torch.pi, 0.01)
plots = [torch.sin(xs)]

# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0.0, 2.0]:
    plots.append(torch.sin(torch.tensor(x0)) + (xs - x0) *
                 torch.cos(torch.tensor(x0)) - (xs - x0)**2 *
                 torch.sin(torch.tensor(x0)) / 2)

d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```



Chúng tôi sẽ mở rộng ý tưởng này cho ý tưởng về một loạt *Taylor * trong phần tiếp theo.

Dòng Taylor

Các dòng *Taylor * cung cấp một phương thức để xấp xỉ hàm $f(x)$ nếu chúng ta được đưa ra các giá trị cho các dẫn xuất n đầu tiên tại một điểm x_0, tức là, $\{f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \ldots, f^{(n)}(x_0)\}$. Ý tưởng sẽ là tìm một đa thức độ n phù hợp với tất cả các dẫn xuất đã cho tại x_0.

Chúng tôi đã thấy trường hợp của $n = 2$ trong phần trước và một đại số nhỏ cho thấy điều này là

$$f(x) \approx \frac{1}{2}\frac{d^2 f}{dx^2}(x_0)(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{19.3.15}$$

Như chúng ta có thể thấy ở trên, mẫu số của 2 có để hủy bỏ 2 chúng ta nhận được khi chúng ta lấy hai dẫn xuất là x^2, trong khi các thuật ngữ khác đều bằng 0. Logic tương tự áp dụng cho đạo hàm đầu tiên và chính giá trị.

Nếu chúng ta đẩy logic hơn nữa lên $n = 3$, chúng ta sẽ kết luận rằng

$$f(x) \approx \frac{\frac{d^3 f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2 f}{dx^2}(x_0)}{2}(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0). \tag{19.3.16}$$

nơi $6 = 3 lần 2 = 3! $ comes from the constant we get in front if we take three derivatives of x^3.

Hơn nữa, chúng ta có thể nhận được một mức độ n đa thức bởi

$$P_n(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i. \tag{19.3.17}$$

where the notation ký hiệu

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f. \tag{19.3.18}$$

Thật vậy, $P_n(x)$ có thể được xem là xấp xỉ đa thức độ n-th tốt nhất với chức năng của chúng tôi $f(x)$.

Mặc dù chúng ta sẽ không đi sâu vào lỗi của các xấp xỉ trên, nhưng điều đáng nói là giới hạn vô hạn. Trong trường hợp này, đối với các hàm được xử lý tốt (được gọi là hàm phân tích thực) như $\cos(x)$ hoặc e^x, chúng ta có thể viết ra số lượng thuật ngữ vô hạn và xấp xỉ cùng một hàm

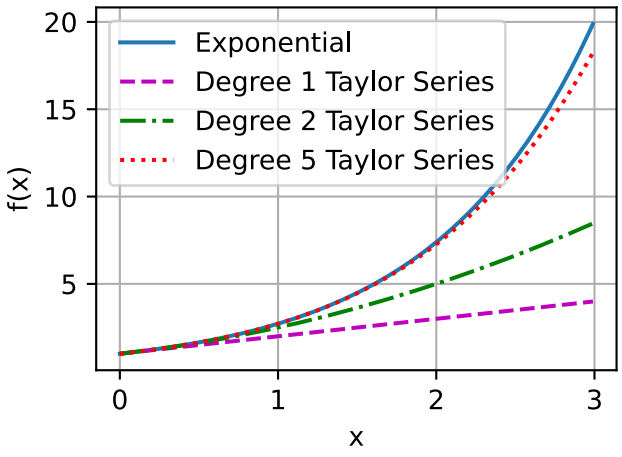
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n. \tag{19.3.19}$$

Lấy $f(x) = e^x$ làm ví dụ. Vì e^x là đạo hàm riêng của nó, chúng ta biết rằng $f^{(n)}(x) = e^x$. Do đó, e^x có thể được xây dựng lại bằng cách lấy loạt Taylor tại $x_0 = 0$, tức là,

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \cdots. \tag{19.3.20}$$

Chúng ta hãy xem cách điều này hoạt động trong mã và quan sát cách tăng mức độ xấp xỉ Taylor đưa chúng ta đến gần hơn với chức năng mong muốn e^x.

```
# Compute the exponential function
xs = torch.arange(0, 3, 0.01)
ys = torch.exp(xs)

# Compute a few Taylor series approximations
P1 = 1 + xs
P2 = 1 + xs + xs**2 / 2
P5 = 1 + xs + xs**2 / 2 + xs**3 / 6 + xs**4 / 24 + xs**5 / 120

d2l.plot(xs, [ys, P1, P2, P5], 'x', 'f(x)', legend=[
    "Exponential", "Degree 1 Taylor Series", "Degree 2 Taylor Series",
    "Degree 5 Taylor Series"])
```



Taylor series có hai ứng dụng chính:

1. • Các ứng dụng lý đầu*: Thường khi chúng ta cố gắng hiểu một hàm quá phức tạp, sử dụng Taylor series cho phép chúng ta biến nó thành một đa thức mà chúng ta có thể làm việc trực tiếp.

2. *Các ứng dụng số*: Một số chức năng như e^x hoặc $\cos(x)$ rất khó để máy tính toán. Họ có thể lưu trữ các bảng giá trị ở độ chính xác cố định (và điều này thường được thực hiện), nhưng nó vẫn để lại các câu hỏi mở như "Chữ số 1000 của $\cos(1)$ là gì?" Taylor loạt thường hữu ích để trả lời những câu hỏi như vậy.

19.3.3 Tóm tắt

- Các dẫn xuất có thể được sử dụng để diễn tả các hàm thay đổi như thế nào khi chúng ta thay đổi đầu vào bằng một lượng nhỏ.
- Các dẫn xuất tiểu học có thể được kết hợp bằng cách sử dụng các quy tắc phái sinh để tạo ra các dẫn xuất phức tạp tùy ý.
- Các dẫn xuất có thể được lặp lại để có được các dẫn xuất bậc hai hoặc cao hơn. Mỗi sự gia tăng theo thứ tự cung cấp thông tin hạt mịn hơn về hành vi của chức năng.
- Sử dụng thông tin trong các dẫn xuất của một ví dụ dữ liệu duy nhất, chúng ta có thể xấp xỉ các hàm hoạt động tốt bằng các đa thức thu được từ chuỗi Taylor.

19.3.4 Bài tập

1. Đạo hàm của $x^3 - 4x + 1$ là gì?
2. Đạo hàm của $\log(\frac{1}{x})$ là gì?
3. Đúng hay Sai: Nếu $f'(x) = 0$ thì f có tối đa hoặc tối thiểu là x?
4. Nơi tối thiểu là $f(x) = x\log(x)$ cho $x \geq 0$ (trong đó chúng tôi giả định rằng f có giá trị giới hạn là 0 tại $f(0)$)?

Discussions[215]

19.4 Tính toán đa năng

Bây giờ chúng ta có một sự hiểu biết khá mạnh mẽ về các dẫn xuất của một hàm của một biến duy nhất, chúng ta hãy quay trở lại câu hỏi ban đầu của chúng tôi, nơi chúng tôi đang xem xét một chức năng mất của hàng tỷ trọng lượng có khả năng.

19.4.1 Sự khác biệt chiều cao hơn Điều Section 19.3 nói với chúng ta là nếu chúng ta thay đổi một trong số hàng tỷ trọng này để lại mọi trọng lượng khác cố định, chúng ta biết điều gì sẽ xảy ra! Điều này không có gì khác hơn là một hàm của một biến duy nhất, vì vậy chúng ta có thể viết

$$L(w_1 + \epsilon_1, w_2, \ldots, w_N) \approx L(w_1, w_2, \ldots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \ldots, w_N). \tag{19.4.1}$$

Chúng ta sẽ gọi đạo hàm trong một biến trong khi sửa các biến khác là đạo hàm *partial *, và chúng ta sẽ sử dụng ký hiệu $\frac{\partial}{\partial w_1}$ cho đạo hàm trong (19.4.1).

[215] https://discuss.d2l.ai/t/1088

Bây giờ, chúng ta hãy lấy điều này và thay đổi w_2 một chút thành $w_2 + \epsilon_2$:

$$\begin{aligned}
L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N) &\approx L(w_1, w_2 + \epsilon_2, \ldots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \epsilon_2 \frac{\partial}{\partial w_2} \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N) \\
&\approx L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \ldots, w_N) \\
&\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \ldots, w_N).
\end{aligned} \tag{19.4.2}$$

Chúng tôi đã một lần nữa sử dụng ý tưởng rằng $\epsilon_1 \epsilon_2$ là một thuật ngữ thứ tự cao hơn mà chúng tôi có thể loại bỏ theo cách tương tự chúng tôi có thể loại bỏ ϵ^2 trong phần trước, cùng với những gì chúng tôi đã thấy trong (19.4.1). Bằng cách tiếp tục theo cách này, chúng tôi có thể viết rằng

$$L(w_1 + \epsilon_1, w_2 + \epsilon_2, \ldots, w_N + \epsilon_N) \approx L(w_1, w_2, \ldots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \ldots, w_N). \tag{19.4.3}$$

Điều này có thể trông giống như một mớ hỗn độn, nhưng chúng ta có thể làm cho điều này quen thuộc hơn bằng cách lưu ý rằng tổng bên phải trông giống hệt như một sản phẩm chấm, vì vậy nếu chúng ta để

$$\boldsymbol{\epsilon} = [\epsilon_1, \ldots, \epsilon_N]^\top \text{ and } \nabla_{\mathbf{x}} L = \left[\frac{\partial L}{\partial x_1}, \ldots, \frac{\partial L}{\partial x_N}\right]^\top, \tag{19.4.4}$$

sau đó

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{19.4.5}$$

Chúng ta sẽ gọi vector $\nabla_{\mathbf{w}} L$ là * gradient* của L.

Phương trình (19.4.5) đáng để suy ngẫm trong giây lát. Nó có chính xác định dạng mà chúng ta gặp phải trong một chiều, chỉ cần chúng ta đã chuyển đổi mọi thứ thành vectơ và các sản phẩm chấm. Nó cho phép chúng ta biết khoảng cách chức năng L sẽ thay đổi cho bất kỳ nhiễu loạn nào đối với đầu vào. Như chúng ta sẽ thấy trong phần tiếp theo, điều này sẽ cung cấp cho chúng ta một công cụ quan trọng trong việc hiểu về mặt hình học cách chúng ta có thể học bằng cách sử dụng thông tin có trong gradient.

Nhưng trước tiên, chúng ta hãy xem xấp xỉ này tại nơi làm việc với một ví dụ. Giả sử rằng chúng ta đang làm việc với hàm

$$f(x, y) = \log(e^x + e^y) \text{ with gradient } \nabla f(x, y) = \left[\frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y}\right]. \tag{19.4.6}$$

Nếu chúng ta nhìn vào một điểm như $(0, \log(2))$, chúng ta thấy rằng

$$f(x, y) = \log(3) \text{ with gradient } \nabla f(x, y) = \left[\frac{1}{3}, \frac{2}{3}\right]. \tag{19.4.7}$$

Do đó, nếu chúng ta muốn xấp xỉ f tại $(\epsilon_1, \log(2) + \epsilon_2)$, chúng ta thấy rằng chúng ta nên có ví dụ cụ thể là (19.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \tag{19.4.8}$$

Chúng ta có thể kiểm tra điều này trong mã để xem xấp xỉ tốt như thế nào.

```
%matplotlib inline
import numpy as np
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l


def f(x, y):
    return torch.log(torch.exp(x) + torch.exp(y))
def grad_f(x, y):
    return torch.tensor([torch.exp(x) / (torch.exp(x) + torch.exp(y)),
                         torch.exp(y) / (torch.exp(x) + torch.exp(y))])

epsilon = torch.tensor([0.01, -0.03])
grad_approx = f(torch.tensor([0.]), torch.log(
    torch.tensor([2.]))) + epsilon.dot(
    grad_f(torch.tensor([0.]), torch.log(torch.tensor(2.))))
true_value = f(torch.tensor([0.]) + epsilon[0], torch.log(
    torch.tensor([2.])) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

```
'approximation: tensor([1.0819]), true Value: tensor([1.0821])'
```

19.4.2 Hình học của Gradient và Gradient Descent Xem xét biểu thức từ (19.4.5) một lần nữa:

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \tag{19.4.9}$$

Chúng ta hãy giả sử rằng tôi muốn sử dụng điều này để giúp giảm thiểu thiệt hại của chúng tôi L. Chúng ta hãy hiểu về mặt hình học thuật toán của gradient gốc đầu tiên được mô tả trong Section 3.5. Những gì chúng tôi sẽ làm là như sau:

1. Bắt đầu với một sự lựa chọn ngẫu nhiên cho các tham số ban đầu $\mathbf{w}$.
2. Tìm hướng $\mathbf{v}$ làm cho L giảm nhanh nhất ở $\mathbf{w}$.
3. Thực hiện một bước nhỏ theo hướng đó: $\mathbf{w} \to \mathbf{w} + \epsilon\mathbf{v}$.
4. Lặp lại.

Điều duy nhất chúng ta không biết chính xác cách làm là tính toán vector $\mathbf{v}$ trong bước thứ hai. Chúng tôi sẽ gọi một hướng như vậy là *hướng đi xuống dốc nhất*. Sử dụng sự hiểu biết hình học của các sản phẩm chấm từ Section 19.1, chúng ta thấy rằng chúng ta có thể viết lại (19.4.5) như

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = L(\mathbf{w}) + \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \tag{19.4.10}$$

Lưu ý rằng chúng tôi đã thực hiện hướng của chúng tôi để có chiều dài một để thuận tiện và sử dụng θ cho góc giữa $\mathbf{v}$ và $\nabla_{\mathbf{w}} L(\mathbf{w})$. Nếu chúng ta muốn tìm hướng giảm L càng nhanh càng tốt, chúng ta muốn làm cho biểu thức này là tiêu cực nhất có thể. Cách duy nhất mà hướng chúng ta chọn đi vào phương trình này là thông qua $\cos(\theta)$, và do đó chúng ta muốn làm cho cosin này là âm nhất có thể. Bây giờ, nhắc lại hình dạng của cosin, chúng ta có thể làm cho điều này càng âm càng tốt bằng cách tạo $\cos(\theta) = -1$ hoặc tương đương làm cho góc giữa gradient và hướng đã chọn của chúng ta là π radian, hoặc tương đương 180 độ. Cách duy nhất để đạt được điều này là đi theo hướng ngược lại chính xác: chọn $\mathbf{v}$ để chỉ theo hướng ngược lại chính xác với $\nabla_{\mathbf{w}} L(\mathbf{w})$!

Điều này đưa chúng ta đến một trong những khái niệm toán học quan trọng nhất trong học máy: hướng của các điểm phong nha dốc nhất theo hướng $-\nabla_{\mathbf{w}} L(\mathbf{w})$. Do đó thuật toán không chính thức của chúng tôi có thể được viết lại như sau.

1. Bắt đầu với một sự lựa chọn ngẫu nhiên cho các tham số ban đầu $\mathbf{w}$.
2. Tính toán $\nabla_{\mathbf{w}} L(\mathbf{w})$.
3. Thực hiện một bước nhỏ theo hướng ngược lại với hướng đó: $\mathbf{w} \rightarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} L(\mathbf{w})$.
4. Lặp lại.

Thuật toán cơ bản này đã được nhiều nhà nghiên cứu sửa đổi và điều chỉnh nhiều cách, nhưng khái niệm cốt lõi vẫn giữ nguyên trong tất cả chúng. Sử dụng gradient để tìm hướng làm giảm tổn thất càng nhanh càng tốt và cập nhật các tham số để thực hiện một bước theo hướng đó.

19.4.3 Một lưu ý về tối ưu hóa toán học Trong suốt cuốn sách này, chúng tôi tập trung thẳng vào các kỹ thuật tối ưu hóa số vì lý do thực tế mà tất cả các hàm chúng ta gặp phải trong cài đặt deep learning là quá phức tạp để giảm thiểu rõ ràng.

Tuy nhiên, đó là một bài tập hữu ích để xem xét những gì sự hiểu biết hình học mà chúng tôi thu được ở trên cho chúng ta biết về việc tối ưu hóa các chức năng trực tiếp.

Giả sử rằng chúng ta muốn tìm giá trị của $\mathbf{x}_0$ mà giảm thiểu một số chức năng $L(\mathbf{x})$. Chúng ta hãy giả sử rằng hơn nữa ai đó mang lại cho chúng ta một giá trị và cho chúng ta biết rằng đó là giá trị giảm thiểu L. Có bất cứ điều gì chúng ta có thể kiểm tra để xem câu trả lời của họ có hợp lý không?

Một lần nữa xem xét (19.4.5):

$$L(mathbf x_0 + boldsymbol epsilon) xpx L(mathbf x_0) + boldsymbol epsilon cdot nabla _{mathbf x} L(mathbf x_0). \tag{19.4.11}$$

Nếu gradient không phải bằng 0, chúng ta biết rằng chúng ta có thể thực hiện một bước theo hướng $-\epsilon \nabla_{\mathbf{x}} L(\mathbf{x}_0)$ để tìm giá trị L nhỏ hơn. Vì vậy, nếu chúng ta thực sự ở mức tối thiểu, đây không thể là trường hợp! Chúng ta có thể kết luận rằng nếu $\mathbf{x}_0$ là mức tối thiểu, thì $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$. Chúng tôi gọi điểm với $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ *điểm quan trọng*.

Điều này thật tốt, bởi vì trong một số cài đặt hiếm, chúng ta * có thể* rõ ràng tìm thấy tất cả các điểm có gradient bằng 0 và tìm điểm có giá trị nhỏ nhất.

Đối với một ví dụ cụ thể, hãy xem xét hàm

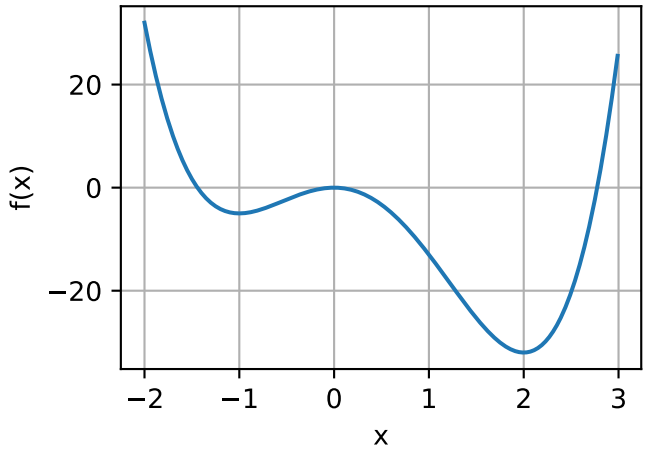
$$f(x) = 3x^4 - 4x^3 - 12x^2. \tag{19.4.12}$$

Hàm này có đạo hàm

$$fracdfdx = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \tag{19.4.13}$$

Vị trí duy nhất có thể của minima là $x = -1, 0, 2$, trong đó hàm lấy các giá trị tương ứng $-5, 0, -32$ và do đó chúng ta có thể kết luận rằng chúng ta giảm thiểu hàm của mình khi $x = 2$. Một cốt truyện nhanh chóng xác nhận điều này.

```
x = torch.arange(-2, 3, 0.01)
f = (3 * x**4) - (4 * x**3) - (12 * x**2)

d2l.plot(x, f, 'x', 'f(x)')
```



Điều này nhấn mạnh một thực tế quan trọng cần biết khi làm việc theo lý thuyết hoặc số: các điểm duy nhất có thể mà chúng ta có thể thu nhỏ (hoặc tối đa hóa) một hàm sẽ có gradient bằng 0, tuy nhiên, không phải mọi điểm có gradient 0 là đúng *toàn cầu* tối thiểu (hoặc tối đa).

19.4.4 Multivariate Chain Rule

Let us suppose that we have a function of four variables (w, x, y, and z) which we can make by composing many terms:

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \qquad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \qquad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \tag{19.4.14}$$

Các chuỗi phương trình như vậy là phổ biến khi làm việc với các mạng thần kinh, vì vậy cố gắng hiểu cách tính toán độ dốc của các chức năng như vậy là chìa khóa. Chúng ta có thể bắt đầu thấy gợi ý trực quan của kết nối này trong Fig. 19.4.1 nếu chúng ta xem những biến liên quan trực tiếp đến nhau.

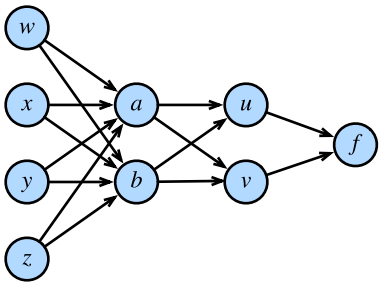


Fig. 19.4.1: The function relations above where nodes represent values and edges show functional dependence.

Không có gì ngăn cản chúng ta chỉ sáng tác tất cả mọi thứ từ (19.4.14) và viết ra rằng

$$f(w,x,y,z) = \left(\left((w+x+y+z)^2 + (w+x-y-z)^2\right)^2 + \left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2\right)^2. \tag{19.4.15}$$

Sau đó chúng ta có thể lấy đạo hàm bằng cách chỉ sử dụng các dẫn xuất biến duy nhất, nhưng nếu chúng ta đã làm điều đó, chúng ta sẽ nhanh chóng thấy mình đầm lầy với các thuật ngữ, nhiều trong số đó được lặp lại! Thật vậy, người ta có thể thấy rằng, ví dụ:

$$\begin{aligned}\frac{\partial f}{\partial w} = 2\Big(2\left(2(w+x+y+z) - 2(w+x-y-z)\right)\left((w+x+y+z)^2 - (w+x-y-z)^2\right) + \\ 2\left(2(w+x-y-z) + 2(w+x+y+z)\right)\left((w+x-y-z)^2 + (w+x+y+z)^2\right)\Big) \times \\ \left(\left((w+x+y+z)^2 - (w+x-y-z)^2\right)^2 + \left((w+x-y-z)^2 + (w+x+y+z)^2\right)^2\right).\end{aligned} \tag{19.4.16}$$

Nếu sau đó chúng ta cũng muốn tính toán $\frac{\partial f}{\partial x}$, chúng ta sẽ kết thúc với một phương trình tương tự một lần nữa với nhiều thuật ngữ lặp đi lặp lại và nhiều thuật ngữ lặp đi lặp lại * shared* giữa hai dẫn xuất. Điều này đại diện cho một lượng lớn công việc lãng phí, và nếu chúng ta cần tính toán các dẫn xuất theo cách này, toàn bộ cuộc cách mạng học sâu sẽ bị đình trệ trước khi nó bắt đầu!

Hãy để chúng tôi chia tay vấn đề. Chúng ta sẽ bắt đầu bằng cách cố gắng hiểu f thay đổi như thế nào khi chúng ta thay đổi a, về cơ bản giả định rằng w, x, y và z tất cả không tồn tại. Chúng tôi sẽ lý do như chúng tôi đã làm trở lại khi chúng tôi làm việc với gradient lần đầu tiên. Hãy để chúng tôi lấy a và thêm một lượng nhỏ ϵ vào nó.

$$\begin{aligned}&f(u(a+\epsilon,b), v(a+\epsilon,b))\\ \approx& f\left(u(a,b) + \epsilon\frac{\partial u}{\partial a}(a,b), v(a,b) + \epsilon\frac{\partial v}{\partial a}(a,b)\right)\\ \approx& f(u(a,b),v(a,b)) + \epsilon\left[\frac{\partial f}{\partial u}(u(a,b),v(a,b))\frac{\partial u}{\partial a}(a,b) + \frac{\partial f}{\partial v}(u(a,b),v(a,b))\frac{\partial v}{\partial a}(a,b)\right].\end{aligned} \tag{19.4.17}$$

Dòng đầu tiên theo sau từ định nghĩa đạo hàm từng phần, và dòng thứ hai theo sau từ định nghĩa của gradient. Việc theo dõi chính xác nơi chúng ta đánh giá mọi phái sinh, như trong biểu thức $\frac{\partial f}{\partial u}(u(a,b),v(a,b))$, vì vậy chúng ta thường viết tắt điều này thành đáng nhớ hơn nhiều

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a}. \tag{19.4.18}$$

Nó rất hữu ích để suy nghĩ về ý nghĩa của quá trình. Chúng tôi đang cố gắng hiểu làm thế nào một hàm của biểu mẫu $f(u(a,b),v(a,b))$ thay đổi giá trị của nó với một sự thay đổi trong a. Có hai con đường này có thể xảy ra: có con đường nơi $a \to u \to f$ và nơi $a \to v \to f$. Chúng ta có thể tính toán cả hai đóng góp này thông qua quy tắc chuỗi: $\frac{\partial w}{\partial u}\cdot\frac{\partial u}{\partial x}$ và $\frac{\partial w}{\partial v}\cdot\frac{\partial v}{\partial x}$ tương ứng, và được thêm vào.

Hãy tưởng tượng chúng ta có một mạng khác nhau của các chức năng trong đó các chức năng bên phải phụ thuộc vào những chức năng được kết nối với bên trái như được hiển thị trong Fig. 19.4.2.

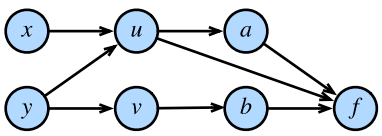


Fig. 19.4.2: Another more subtle example of the chain rule.

Để tính toán một cái gì đó như $\frac{\partial f}{\partial y}$, chúng ta cần tổng hợp tất cả (trong trường hợp này là 3) đường dẫn từ y đến f cho

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial v}\frac{\partial v}{\partial y}. \tag{19.4.19}$$

Hiểu quy tắc chuỗi theo cách này sẽ trả cổ tức lớn khi cố gắng hiểu cách gradient chảy qua mạng và tại sao các lựa chọn kiến trúc khác nhau như trong LSTMs (Section 10.2) hoặc các lớp còn lại (Section 8.6) có thể giúp định hình quá trình học tập bằng cách kiểm soát dòng chảy gradient.

19.4.5 Thuật toán Backpropagation

Hãy để chúng tôi quay lại ví dụ về (19.4.14) phần trước, nơi

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \qquad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \qquad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \tag{19.4.20}$$

Nếu chúng ta muốn tính toán $\frac{\partial f}{\partial w}$, chúng ta có thể áp dụng quy tắc chuỗi đa biến thể để xem:

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial w} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial u}{\partial b}\frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial v}{\partial b}\frac{\partial b}{\partial w}. \end{aligned} \tag{19.4.21}$$

Chúng ta hãy thử sử dụng phân hủy này để tính toán $\frac{\partial f}{\partial w}$. Lưu ý rằng tất cả những gì chúng ta cần ở đây là các phần bước duy nhất khác nhau:

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u + v), \quad \frac{\partial f}{\partial v} = 2(u + v), \\ \frac{\partial u}{\partial a} &= 2(a + b), \quad \frac{\partial u}{\partial b} = 2(a + b), \\ \frac{\partial v}{\partial a} &= 2(a - b), \quad \frac{\partial v}{\partial b} = -2(a - b), \\ \frac{\partial a}{\partial w} &= 2(w + x + y + z), \quad \frac{\partial b}{\partial w} = 2(w + x - y - z). \end{aligned} \tag{19.4.22}$$

Nếu chúng ta viết ra mã này sẽ trở thành một biểu thức khá dễ quản lý.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'    f at {w}, {x}, {y}, {z} is {f}')

# Compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
```

(continues on next page)

(continued from previous page)

```
# Compute the final result from inputs to outputs
du_dw, dv_dw = du_da*da_dw + du_db*db_dw, dv_da*da_dw + dv_db*db_dw
df_dw = df_du*du_dw + df_dv*dv_dw
print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
```

```
    f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
```

Tuy nhiên, lưu ý rằng điều này vẫn không làm cho nó dễ dàng để tính toán một cái gì đó như $\frac{\partial f}{\partial x}$. Lý do cho điều đó là * cách chúng tôi đã chọn áp dụng quy tắc chuỗi. Nếu chúng ta nhìn vào những gì chúng ta đã làm ở trên, chúng ta luôn giữ ∂w trong mẫu số khi chúng ta có thể. Bằng cách này, chúng tôi đã chọn áp dụng quy tắc chuỗi xem w đã thay đổi mọi biến khác như thế nào. Nếu đó là những gì chúng tôi muốn, đây sẽ là một ý tưởng hay. Tuy nhiên, hãy nghĩ lại động lực của chúng tôi từ deep learning: chúng tôi muốn xem cách mọi tham số thay đổi *loss*. Về bản chất, chúng tôi muốn áp dụng quy tắc chuỗi giữ ∂f trong tử số bất cứ khi nào chúng tôi có thể!

Để rõ ràng hơn, lưu ý rằng chúng ta có thể viết

$$\begin{aligned}\frac{\partial f}{\partial w} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w},\\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial a} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial a},\\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial u}\frac{\partial u}{\partial b} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial b}.\end{aligned} \tag{19.4.23}$$

Lưu ý rằng ứng dụng này của quy tắc chuỗi đã cho chúng tôi tính toán một cách rõ ràng $\frac{\partial f}{\partial u}$, $\frac{\partial f}{\partial v}$, $\frac{\partial f}{\partial a}$, $\frac{\partial f}{\partial b}$, and $\frac{\partial f}{\partial w}$. Không có gì ngăn cản chúng ta cũng bao gồm cả các phương trình:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial x} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial x},\\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial y} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial y},\\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial a}\frac{\partial a}{\partial z} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial z}.\end{aligned} \tag{19.4.24}$$

và sau đó theo dõi cách f thay đổi khi chúng ta thay đổi nút * bất kỳ* nào trong toàn bộ mạng. Hãy để chúng tôi thực hiện nó.

```
# Compute the value of the function from inputs to outputs
w, x, y, z = -1, 0, -2, 1
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2
print(f'f at {w}, {x}, {y}, {z} is {f}')

# Compute the derivative using the decomposition above
# First compute the single step partials
df_du, df_dv = 2*(u + v), 2*(u + v)
du_da, du_db, dv_da, dv_db = 2*(a + b), 2*(a + b), 2*(a - b), -2*(a - b)
da_dw, db_dw = 2*(w + x + y + z), 2*(w + x - y - z)
da_dx, db_dx = 2*(w + x + y + z), 2*(w + x - y - z)
da_dy, db_dy = 2*(w + x + y + z), -2*(w + x - y - z)
da_dz, db_dz = 2*(w + x + y + z), -2*(w + x - y - z)
```

(continues on next page)

(continued from previous page)

```
# Now compute how f changes when we change any value from output to input
df_da, df_db = df_du*du_da + df_dv*dv_da, df_du*du_db + df_dv*dv_db
df_dw, df_dx = df_da*da_dw + df_db*db_dw, df_da*da_dx + df_db*db_dx
df_dy, df_dz = df_da*da_dy + df_db*db_dy, df_da*da_dz + df_db*db_dz

print(f'df/dw at {w}, {x}, {y}, {z} is {df_dw}')
print(f'df/dx at {w}, {x}, {y}, {z} is {df_dx}')
print(f'df/dy at {w}, {x}, {y}, {z} is {df_dy}')
print(f'df/dz at {w}, {x}, {y}, {z} is {df_dz}')
```

```
f at -1, 0, -2, 1 is 1024
df/dw at -1, 0, -2, 1 is -4096
df/dx at -1, 0, -2, 1 is -4096
df/dy at -1, 0, -2, 1 is -4096
df/dz at -1, 0, -2, 1 is -4096
```

Thực tế là chúng ta tính toán các dẫn xuất từ f trở lại các đầu vào chứ không phải từ các đầu vào chuyển tiếp đến đầu ra (như chúng ta đã làm trong đoạn mã đầu tiên ở trên) là những gì mang lại cho thuật toán này tên của nó: * backpropagation*. Lưu ý rằng có hai bước: 1. Tính giá trị của hàm, và các phần bước duy nhất từ trước ra sau. Mặc dù không được thực hiện ở trên, điều này có thể được kết hợp thành một * chuyển tiếp duy nhất*. 2. Tính toán độ dốc của f từ sau ra trước. Chúng tôi gọi đây là * đèo ngược *.

Đây chính xác là những gì mọi thuật toán học sâu thực hiện để cho phép tính toán độ dốc của sự mất mát đối với mọi trọng lượng trong mạng tại một lần vượt qua. Đó là một thực tế đáng kinh ngạc rằng chúng ta có một sự phân hủy như vậy.

Để xem làm thế nào để đóng gói này, chúng ta hãy xem nhanh ví dụ này.

```
# Initialize as ndarrays, then attach gradients
w = torch.tensor([-1.], requires_grad=True)
x = torch.tensor([0.], requires_grad=True)
y = torch.tensor([-2.], requires_grad=True)
z = torch.tensor([1.], requires_grad=True)
# Do the computation like usual, tracking gradients
a, b = (w + x + y + z)**2, (w + x - y - z)**2
u, v = (a + b)**2, (a - b)**2
f = (u + v)**2

# Execute backward pass
f.backward()

print(f'df/dw at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {w.grad.data.item()}')
print(f'df/dx at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {x.grad.data.item()}')
print(f'df/dy at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {y.grad.data.item()}')
print(f'df/dz at {w.data.item()}, {x.data.item()}, {y.data.item()}, '
      f'{z.data.item()} is {z.grad.data.item()}')
```

```
df/dw at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dx at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

(continues on next page)

(continued from previous page)

```
df/dy at -1.0, 0.0, -2.0, 1.0 is -4096.0
df/dz at -1.0, 0.0, -2.0, 1.0 is -4096.0
```

Tất cả những gì chúng tôi đã làm ở trên có thể được thực hiện tự động bằng cách gọi `f.backwards()`.

19.4.6 Hessia Như với phép tính biến đơn lẻ, nó rất hữu ích để xem xét các dẫn xuất bậc cao hơn để có được một xử lý về cách chúng ta có thể có được một xấp xỉ tốt hơn với một hàm hơn là sử dụng gradient một mình.

Có một vấn đề ngay lập tức mà người ta gặp phải khi làm việc với các dẫn xuất bậc cao hơn của các hàm của một số biến, và đó là có một số lượng lớn trong số chúng. Nếu chúng ta có một hàm $f(x_1, \ldots, x_n)$ của n biến, thì chúng ta có thể lấy n^2 nhiều dẫn xuất thứ hai, cụ thể là cho bất kỳ sự lựa chọn nào của i và j:

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i}\left(\frac{d}{dx_j} f\right). \tag{19.4.25}$$

Điều này theo truyền thống được lắp ráp thành một ma trận gọi là *Hessian*:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \tag{19.4.26}$$

Không phải mọi mục nhập của ma trận này đều độc lập. Thật vậy, chúng ta có thể chỉ ra rằng miễn là cả hai * hỗn hợp* (phái sinh một phần liên quan đến nhiều biến) tồn tại và liên tục, chúng ta có thể nói rằng đối với bất kỳ i, và j,

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \tag{19.4.27}$$

Điều này theo sau bằng cách xem xét đầu tiên perturbing một chức năng theo hướng x_i, và sau đó xáo trộn nó trong x_j và sau đó so sánh kết quả của nó với những gì xảy ra nếu chúng ta perturb trước x_j và sau đó x_i, với kiến thức rằng cả hai đơn đặt hàng này dẫn đến cùng một thay đổi cuối cùng trong sản lượng của f.

Như với các biến đơn lẻ, chúng ta có thể sử dụng các dẫn xuất này để hiểu rõ hơn về cách hàm hoạt động gần một điểm. Đặc biệt, chúng ta có thể sử dụng nó để tìm bậc hai phù hợp nhất gần một điểm $\mathbf{x}_0$, như chúng ta đã thấy trong một biến duy nhất.

Hãy để chúng tôi xem một ví dụ. Giả sử rằng $f(x_1, x_2) = a + b_1 x_1 + b_2 x_2 + c_{11} x_1^2 + c_{12} x_1 x_2 + c_{22} x_2^2$. Đây là dạng chung cho một bậc hai trong hai biến. Nếu chúng ta nhìn vào giá trị của hàm, gradient của nó và Hessian (19.4.26) của nó, tất cả tại điểm zero:

$$\begin{aligned} f(0,0) &= a, \\ \nabla f(0,0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H} f(0,0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}, \end{aligned} \tag{19.4.28}$$

we can get our original nguyên polynomial đa thức back by saying nói

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2}\mathbf{x}^\top \mathbf{H} f(0) \mathbf{x}. \tag{19.4.29}$$

Nói chung, nếu chúng ta tính toán mở rộng này bất kỳ điểm $\mathbf{x}_0$, chúng ta thấy rằng

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H} f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0). \tag{19.4.30}$$

Điều này hoạt động cho bất kỳ đầu vào chiều nào và cung cấp bậc hai xấp xỉ tốt nhất cho bất kỳ hàm nào tại một điểm. Để đưa ra một ví dụ, chúng ta hãy vẽ hàm

$$f(x, y) = xe^{-x^2-y^2}. \tag{19.4.31}$$

One can compute that the gradient and Hessian are

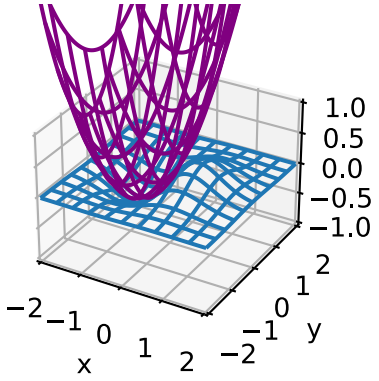
$$\nabla f(x, y) = e^{-x^2-y^2} \begin{pmatrix} 1 - 2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H} f(x, y) = e^{-x^2-y^2} \begin{pmatrix} 4x^3 - 6x & 4x^2y - 2y \\ 4x^2y - 2y & 4xy^2 - 2x \end{pmatrix}. \tag{19.4.32}$$

Và do đó, với một đại số nhỏ, thấy rằng bậc hai xấp xỉ tại $[-1, 0]^\top$ là

$$f(x, y) \approx e^{-1}\left(-1 - (x + 1) + (x + 1)^2 + y^2\right). \tag{19.4.33}$$

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101),
                      torch.linspace(-2, 2, 101))

z = x*torch.exp(- x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = torch.exp(torch.tensor([-1.]))*(-1 - (x + 1) + 2 * (x + 1)**2 + 2 * y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x.numpy(), y.numpy(), z.numpy(),
                  **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x.numpy(), y.numpy(), w.numpy(),
                  **{'rstride': 10, 'cstride': 10}, color='purple')
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(-1, 1)
ax.dist = 12
```

```
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Điều này tạo thành cơ sở cho Thuật toán Newton được thảo luận trong Section 12.3, nơi chúng tôi thực hiện tối ưu hóa số lặp đi lặp lại việc tìm kiếm bậc hai phù hợp nhất, và sau đó chính xác giảm thiểu bậc hai đó.

19.4.7 A Little Matrix Calculus Dẫn xuất của các hàm liên quan đến ma trận hóa ra là đặc biệt tốt đẹp. Phần này có thể trở nên nặng nề về mặt công thức, vì vậy có thể bị bỏ qua trong lần đọc đầu tiên, nhưng rất hữu ích khi biết các dẫn xuất của các hàm liên quan đến hoạt động ma trận phổ biến thường sạch hơn nhiều so với người ta có thể dự đoán ban đầu, đặc biệt là cho các hoạt động ma trận trung tâm như thế nào đối với các ứng dụng học sâu.

Hãy để chúng tôi bắt đầu với một ví dụ. Giả sử rằng chúng ta có một số vector cột cố định $\boldsymbol{\beta}$, và chúng ta muốn lấy hàm sản phẩm $f(\mathbf{x}) = \boldsymbol{\beta}^\top \mathbf{x}$, và hiểu làm thế nào các sản phẩm chấm thay đổi khi chúng ta thay đổi $\mathbf{x}$.

Một chút ký hiệu sẽ hữu ích khi làm việc với các dẫn xuất ma trận trong ML được gọi là đạo hàm ma trận bố trí *mẫu số* nơi chúng ta lắp ráp các dẫn xuất một phần của chúng ta vào hình dạng của bất kỳ vectơ, ma trận hoặc tensor nào nằm trong mẫu số của vi phân. Trong trường hợp này, chúng tôi sẽ viết

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}, \tag{19.4.34}$$

nơi chúng tôi phù hợp với hình dạng của vector cột $\mathbf{x}$.

Nếu chúng ta viết ra chức năng của chúng ta vào các thành phần thì đây là

$$f(\mathbf{x}) = \sum_{i=1}^{n} \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \tag{19.4.35}$$

Nếu bây giờ chúng ta lấy phái sinh một phần liên quan đến việc nói β_1, lưu ý rằng mọi thứ đều bằng không nhưng thuật ngữ đầu tiên, chỉ x_1 nhân với β_1, vì vậy chúng tôi có được điều đó

$$\frac{df}{dx_1} = \beta_1, \tag{19.4.36}$$

or more generally nói chung that

$$\frac{df}{dx_i} = \beta_i. \tag{19.4.37}$$

Bây giờ chúng ta có thể lắp ráp lại this into a matrix ma trận to see

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}. \tag{19.4.38}$$

Điều này minh họa một vài yếu tố về phép tính ma trận mà chúng ta thường sẽ phản ứng trong suốt phần này:

- Đầu tiên, Các tính toán sẽ nhận được khá tham gia.
- Thứ hai, Kết quả cuối cùng sạch hơn nhiều so với quá trình trung gian và sẽ luôn trông giống với trường hợp biến đơn lẻ. Trong trường hợp này, lưu ý rằng $\frac{d}{dx}(bx) = b$ và $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}^\top \mathbf{x}) = \boldsymbol{\beta}$ đều giống nhau.
- Thứ ba, transposes thường có thể xuất hiện dường như từ hư không. Lý do cốt lõi cho điều này là quy ước mà chúng ta khớp với hình dạng của mẫu số, do đó khi chúng ta nhân ma trận, chúng ta sẽ cần phải lấy transposes để khớp trở lại hình dạng của thuật ngữ gốc.

Để tiếp tục xây dựng trực giác, chúng ta hãy thử một tính toán khó khăn hơn một chút. Giả sử rằng chúng ta có một vector cột $\mathbf{x}$, và một ma trận vuông A và chúng tôi muốn tính toán

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}). \tag{19.4.39}$$

Để hướng tới dễ thao tác ký hiệu hơn, chúng ta hãy xem xét vấn đề này bằng ký hiệu Einstein. Trong trường hợp này chúng ta có thể viết hàm như

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \tag{19.4.40}$$

Để tính toán đạo hàm của chúng ta, chúng ta cần phải hiểu cho mỗi k, giá trị của

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \tag{19.4.41}$$

Theo quy tắc sản phẩm, đây là

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \tag{19.4.42}$$

Đối với một thuật ngữ như $\frac{dx_i}{dx_k}$, không khó để thấy rằng đây là một khi $i = k$ và không khác. Điều này có nghĩa là mọi thuật ngữ mà i và k khác nhau biến mất so với số tiền này, vì vậy các thuật ngữ duy nhất còn lại trong số tiền đầu tiên đó là những điều khoản mà $i = k$. Lý do tương tự giữ cho nhiệm kỳ thứ hai mà chúng ta cần $j = k$. Điều này cho

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \tag{19.4.43}$$

Bây giờ, tên của các chỉ số trong ký hiệu Einstein là trọng tài - thực tế là i và j khác nhau là không quan trọng đối với tính toán này vào thời điểm này, vì vậy chúng ta có thể lập chỉ mục lại để cả hai đều sử dụng i để thấy rằng

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \tag{19.4.44}$$

Bây giờ, đây là nơi chúng ta bắt đầu cần một số thực hành để đi xa hơn. Chúng ta hãy thử và xác định kết quả này về hoạt động ma trận. $a_{ki} + a_{ik}$ là thành phần thứ k, i của $\mathbf{A} + \mathbf{A}^\top$. Điều này cho

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \tag{19.4.45}$$

Tương tự, thuật ngữ này bây giờ là tích của ma trận $\mathbf{A} + \mathbf{A}^\top$ bởi vector $\mathbf{x}$, vì vậy chúng ta thấy rằng

$$\left[\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x})\right]_k = \frac{d}{dx_k}x_i a_{ij}x_j = [(\mathbf{A} + \mathbf{A}^\top)\mathbf{x}]_k. \tag{19.4.46}$$

Do đó, chúng ta thấy rằng mục k-th của đạo hàm mong muốn từ (19.4.39) chỉ là mục k-th của vectơ ở bên phải, và do đó hai là như nhau. Do đó sản lượng

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{19.4.47}$$

Điều này đòi hỏi nhiều công việc hơn đáng kể so với công việc cuối cùng của chúng tôi, nhưng kết quả cuối cùng là nhỏ. Hơn thế nữa, hãy xem xét tính toán sau cho các dẫn xuất biến đơn truyền thống:

$$\frac{d}{dx}(xax) = \frac{dx}{dx}ax + xa\frac{dx}{dx} = (a + a)x. \tag{19.4.48}$$

Tương đương $\frac{d}{dx}(ax^2) = 2ax = (a + a)x$. Một lần nữa, chúng tôi nhận được một kết quả trông khá giống như kết quả biến duy nhất nhưng với một transpose ném vào.

Tại thời điểm này, mô hình nên trông khá đáng ngờ, vì vậy chúng ta hãy cố gắng tìm ra lý do tại sao. Khi chúng ta lấy các dẫn xuất ma trận như thế này, trước tiên chúng ta giả định rằng biểu thức chúng ta nhận được sẽ là một biểu thức ma trận khác: một biểu thức chúng ta có thể viết nó về sản phẩm và tổng của ma trận và các transposes của chúng. Nếu một biểu thức như vậy tồn tại, nó sẽ cần phải đúng đối với tất cả các ma trận. Đặc biệt, nó sẽ cần phải đúng với ma trận 1×1, trong trường hợp đó sản phẩm ma trận chỉ là tích của các số, tổng ma trận chỉ là tổng, và transpose không làm gì cả! Nói cách khác, bất cứ biểu thức nào chúng ta nhận được *phải* khớp với biểu thức biến đơn. Điều này có nghĩa là, với một số thực hành, người ta thường có thể đoán các dẫn xuất ma trận chỉ bằng cách biết biểu thức biến đơn liên quan phải trông như thế nào!

Hãy để chúng tôi thử điều này ra. Giả sử rằng $\mathbf{X}$ là một ma trận $n \times m$, $\mathbf{U}$ là một $n \times r$ và $\mathbf{V}$ là một $r \times m$. Let us try to computetính toán

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = ? \tag{19.4.49}$$

Tính toán này rất quan trọng trong một khu vực gọi là tính toán ma trận. Đối với chúng tôi, tuy nhiên, nó chỉ là một dẫn xuất để tính toán. Hãy để chúng tôi cố gắng để hình ảnh những gì điều này sẽ là cho 1×1 ma trận. Trong trường hợp đó, chúng tôi nhận được biểu thức

$$\frac{d}{dv}(x - uv)^2 = -2(x - uv)u, \tag{19.4.50}$$

ở đâu, đạo hàm là khá chuẩn. Nếu chúng ta cố gắng chuyển đổi này trở lại thành một biểu thức ma trận, chúng ta nhận được

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2(\mathbf{X} - \mathbf{UV})\mathbf{U}. \tag{19.4.51}$$

Tuy nhiên, nếu chúng ta nhìn vào điều này nó không hoàn toàn hoạt động. Nhớ lại rằng $\mathbf{X}$ là $n \times m$, như là $\mathbf{UV}$, vì vậy ma trận $2(\mathbf{X} - \mathbf{UV})$ là $n \times m$. Mặt khác, $\mathbf{U}$ là $n \times r$ và chúng ta không thể nhân $n \times m$ và ma trận $n \times r$ vì kích thước không khớp!

Chúng tôi muốn nhận được $\frac{d}{d\mathbf{V}}$, có hình dạng tương tự như $\mathbf{V}$, đó là $r \times m$. Vì vậy, bằng cách nào đó chúng ta cần phải có một ma trận $n \times m$ và ma trận $n \times r$, nhân chúng lại với nhau (có lẽ với một số transposes) để có được một $r \times m$. Chúng ta có thể làm điều này bằng cách nhân $U^\top$ bởi $(\mathbf{X} - \mathbf{UV})$. Như vậy, chúng ta có thể đoán được giải pháp cho (19.4.49) là

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \tag{19.4.52}$$

Để chứng minh rằng điều này hoạt động, chúng tôi sẽ được remiss để không cung cấp một tính toán chi tiết. Nếu chúng ta đã tin rằng quy tắc ngón tay cái này hoạt động, hãy bỏ qua quá khứ nguồn gốc này. Để tính toán

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2, \tag{19.4.53}$$

chúng ta phải tìm cho mỗi a, và b

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \frac{d}{dv_{ab}}\sum_{i,j}\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)^2. \tag{19.4.54}$$

Nhắc lại rằng tất cả các mục của $\mathbf{X}$ và $\mathbf{U}$ là hằng số như xa như $\frac{d}{dv_{ab}}$ có liên quan, chúng tôi có thể đẩy dẫn xuất bên trong tổng, và áp dụng quy tắc chuỗi cho hình vuông để có được

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = \sum_{i,j} 2\left(x_{ij} - \sum_k u_{ik}v_{kj}\right)\left(-\sum_k u_{ik}\frac{dv_{kj}}{dv_{ab}}\right). \tag{19.4.55}$$

Như trong nguồn gốc trước đó, chúng ta có thể lưu ý rằng $\frac{dv_{kj}}{dv_{ab}}$ chỉ không phải bằng không nếu $k = a$ và $j = b$. Nếu một trong những điều kiện đó không giữ, thuật ngữ trong tổng là 0 và chúng tôi có thể tự do loại bỏ nó. Chúng tôi thấy rằng

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i\left(x_{ib} - \sum_k u_{ik}v_{kb}\right)u_{ia}. \tag{19.4.56}$$

Một sự tinh tế quan trọng ở đây là yêu cầu $k = a$ không xảy ra bên trong tổng trong vì k đó là một biến giả mà chúng ta đang tóm tắt bên trong thuật ngữ bên trong. Đối với một ví dụ sạch hơn về mặt công thức, hãy xem xét lý do tại sao

$$\frac{d}{dx_1}\left(\sum_i x_i\right)^2 = 2\left(\sum_i x_i\right). \tag{19.4.57}$$

Từ thời điểm này, chúng ta có thể bắt đầu xác định các thành phần của tổng. Đầu tiên,

$$\sum_k u_{ik}v_{kb} = [\mathbf{U}\mathbf{V}]_{ib}. \tag{19.4.58}$$

Vì vậy, toàn bộ biểu thức ở bên trong tổng là

$$x_{ib} - \sum_k u_{ik}v_{kb} = [\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib}. \tag{19.4.59}$$

Điều này có nghĩa là bây giờ chúng ta có thể viết đạo hàm của chúng tôi như

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i[\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib}u_{ia}. \tag{19.4.60}$$

Chúng ta muốn điều này trông giống như phần tử a, b của ma trận để chúng ta có thể sử dụng kỹ thuật như trong ví dụ trước để đến một biểu thức ma trận, có nghĩa là chúng ta cần trao đổi thứ tự của các chỉ số trên u_{ia}. Nếu chúng ta nhận thấy rằng $u_{ia} = [\mathbf{U}^\top]_{ai}$, sau đó chúng ta có thể viết

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{U}\mathbf{V}\|_2^2 = -2\sum_i[\mathbf{U}^\top]_{ai}[\mathbf{X} - \mathbf{U}\mathbf{V}]_{ib}. \tag{19.4.61}$$

Đây là một sản phẩm ma trận, và do đó chúng ta có thể kết luận rằng

$$\frac{d}{dv_{ab}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2[\mathbf{U}^\top(\mathbf{X} - \mathbf{UV})]_{ab}. \tag{19.4.62}$$

và do đó chúng tôi có thể viết giải pháp cho (19.4.49)

$$\frac{d}{d\mathbf{V}}\|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \tag{19.4.63}$$

Điều này phù hợp với giải pháp mà chúng tôi đoán ở trên!

Nó là hợp lý để hỏi tại thời điểm này, "Tại sao tôi không thể chỉ viết ra các phiên bản ma trận của tất cả các quy tắc giải tích tôi đã học được? Rõ ràng đây vẫn là cơ khí. Tại sao chúng ta không chỉ vượt qua nó với! Và thực sự có những quy tắc như vậy và (Petersen et al., 2008) cung cấp một bản tóm tắt tuyệt vời. Tuy nhiên, do có rất nhiều cách các phép toán ma trận có thể được kết hợp so với các giá trị đơn lẻ, có nhiều quy tắc đạo hàm ma trận nhiều hơn các quy tắc biến đơn lẻ. Nó thường là trường hợp tốt nhất để làm việc với các chỉ số, hoặc để nó lên đến sự khác biệt tự động khi thích hợp.

19.4.8 Tóm tắt

- Trong các chiều cao hơn, chúng ta có thể định nghĩa gradient phục vụ cùng mục đích như các dẫn xuất trong một chiều. Những điều này cho phép chúng ta xem hàm đa biến thay đổi như thế nào khi chúng ta thực hiện một thay đổi nhỏ tùy ý đối với các đầu vào.
- Thuật toán lan truyền ngược có thể thấy là một phương pháp tổ chức quy tắc chuỗi đa biến để cho phép tính toán hiệu quả của nhiều dẫn xuất từng phần.
- Phép tính ma trận cho phép chúng ta viết các dẫn xuất của các biểu thức ma trận theo những cách súc tích.

19.4.9 Bài tập 1. Cho một vector cột β, tính toán các dẫn xuất của cả $f(\mathbf{x}) = \beta^\top\mathbf{x}$ và $g(\mathbf{x}) = \mathbf{x}^\top\beta$. Tại sao bạn nhận được câu trả lời tương tự? 2. Hãy để $\mathbf{v}$ là một vector kích thước n. $\frac{\partial}{\partial\mathbf{v}}\|\mathbf{v}\|_2$ là cái gì? 3. Hãy để $L(x, y) = \log(e^x + e^y)$. Tính toán gradient. Tổng của các thành phần của gradient là gì? 4. Hãy để $f(x, y) = x^2y + xy^2$. Cho thấy điểm quan trọng duy nhất là $(0, 0)$. Bằng cách xem xét $f(x, x)$, xác định xem $(0, 0)$ có phải là tối đa, tối thiểu hay không. Giả sử rằng chúng ta đang giảm thiểu một hàm $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$. Làm thế nào chúng ta có thể giải thích hình học tình trạng của $\nabla f = 0$ về g và h?

Discussions[216]

19.5 Integral Calculus

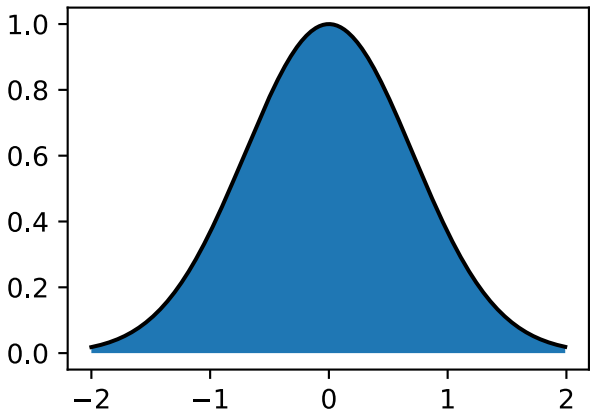
Sự khác biệt chỉ chiếm một nửa nội dung của một nền giáo dục giải tích truyền thống. Trụ cột khác, hội nhập, bắt đầu dường như một câu hỏi khá tách biệt, "Khu vực bên dưới đường cong này là gì?" Mặc dù dường như không liên quan, hội nhập được đan xen chặt chẽ với sự khác biệt thông qua cái được gọi là định lý cơ bản * của giải tích*.

[216] https://discuss.d2l.ai/t/1090

Ở cấp độ học máy mà chúng ta thảo luận trong cuốn sách này, chúng ta sẽ không cần sự hiểu biết sâu sắc về hội nhập. Tuy nhiên, chúng tôi sẽ cung cấp một giới thiệu ngắn gọn để đặt nền tảng cho bất kỳ ứng dụng tiếp theo nào chúng tôi sẽ gặp sau này.

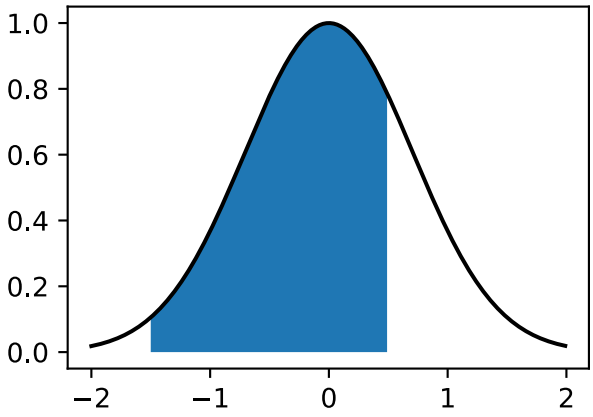
19.5.1 Giải thích hình học Giả sử chúng ta có hàm $f(x)$. Để đơn giản, chúng ta hãy giả định rằng $f(x)$ là không âm (không bao giờ có giá trị nhỏ hơn 0). Điều chúng ta muốn thử và hiểu là: khu vực chứa giữa $f(x)$ và trục x-trục là gì?

```
%matplotlib inline
import torch
from IPython import display
from mpl_toolkits import mplot3d
from d2l import torch as d2l

x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist(), f.tolist())
d2l.plt.show()
```



Trong hầu hết các trường hợp, khu vực này sẽ là vô hạn hoặc không xác định (xem xét khu vực dưới $f(x) = x^2$), vì vậy mọi người thường sẽ nói về khu vực giữa một cặp kết thúc, nói a và b.

```
x = torch.arange(-2, 2, 0.01)
f = torch.exp(-x**2)

d2l.set_figsize()
d2l.plt.plot(x, f, color='black')
d2l.plt.fill_between(x.tolist()[50:250], f.tolist()[50:250])
d2l.plt.show()
```



Chúng ta sẽ biểu thị khu vực này bằng ký hiệu tích phân bên dưới:

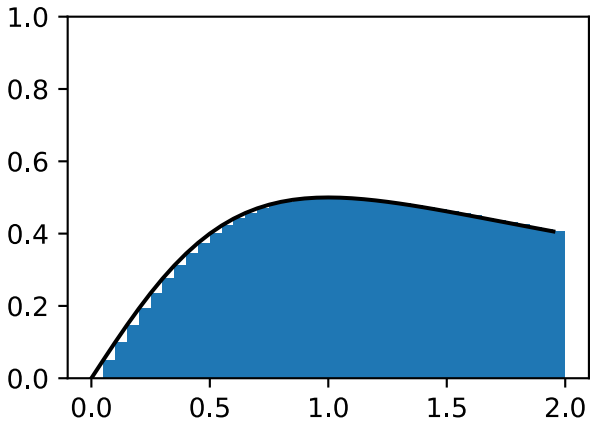
$$\text{Area}(\mathcal{A}) = \int_a^b f(x)\,dx. \tag{19.5.1}$$

Biến bên trong là một biến giả, giống như chỉ số của một tổng trong một $\sum$, và do đó điều này có thể được viết tương đương với bất kỳ giá trị bên trong chúng ta thích:

$$\int_a^b f(x)\,dx = \int_a^b f(z)\,dz. \tag{19.5.2}$$

Có một cách truyền thống để thử và hiểu cách chúng ta có thể cố gắng xấp xỉ các tích phân như vậy: chúng ta có thể tưởng tượng việc lấy khu vực nằm trong khoảng a và b và cắt nó thành N lát dọc. Nếu N lớn, chúng ta có thể xấp xỉ diện tích của mỗi lát bằng một hình chữ nhật, và sau đó thêm các khu vực để lấy tổng diện tích dưới đường cong. Chúng ta hãy xem xét một ví dụ làm điều này trong mã. Chúng ta sẽ thấy làm thế nào để có được giá trị thực sự trong một phần sau.

```
epsilon = 0.05
a = 0
b = 2

x = torch.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = torch.sum(epsilon*f)
true = torch.log(torch.tensor([5.])) / 2

d2l.set_figsize()
d2l.plt.bar(x, f, width=epsilon, align='edge')
d2l.plt.plot(x, f, color='black')
d2l.plt.ylim([0, 1])
d2l.plt.show()

f'approximation: {approx}, truth: {true}'
```




```
'approximation: 0.7944855690002441, truth: tensor([0.8047])'
```

Vấn đề là mặc dù nó có thể được thực hiện bằng số, chúng ta có thể thực hiện cách tiếp cận này một cách phân tích chỉ cho các chức năng đơn giản nhất như

$$\int_a^b x \, dx. \tag{19.5.3}$$

Bất cứ điều gì hơi phức tạp hơn như ví dụ của chúng tôi từ mã trên

$$\int_a^b \frac{x}{1+x^2} \, dx. \tag{19.5.4}$$

vượt ra ngoài những gì chúng ta có thể giải quyết bằng phương pháp trực tiếp như vậy.

Thay vào đó, chúng tôi sẽ có một cách tiếp cận khác. Chúng ta sẽ làm việc trực giác với khái niệm về khu vực, và tìm hiểu công cụ tính toán chính được sử dụng để tìm tích phân: định lý * cơ bản của giải tính*. Đây sẽ là cơ sở cho việc nghiên cứu hội nhập của chúng tôi.

19.5.2 Định lý cơ bản của giải tích

Để đi sâu hơn vào lý thuyết hội nhập, chúng ta hãy giới thiệu một hàm

$$F(x) = \int_0^x f(y)dy. \tag{19.5.5}$$

Chức năng này đo diện tích giữa 0 và x tùy thuộc vào cách chúng ta thay đổi x. Lưu ý rằng đây là tất cả mọi thứ chúng ta cần kể từ

$$\int_a^b f(x) \, dx = F(b) - F(a). \tag{19.5.6}$$

Đây là một mã hóa toán học của thực tế là chúng ta có thể đo diện tích ra đến điểm cuối xa và sau đó trừ khu vực vào điểm cuối gần như được chỉ ra trong Fig. 19.5.1.

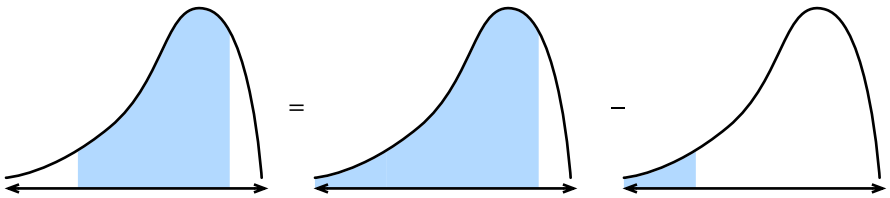


Fig. 19.5.1: Visualizing why we may reduce the problem of computing the area under a curve between two points to computing the area to the left of a point.

Do đó, chúng ta có thể tìm ra tích phân trong bất kỳ khoảng thời gian nào bằng cách tìm ra $F(x)$ là gì.

Để làm như vậy, chúng ta hãy xem xét một thí nghiệm. Như chúng ta thường làm trong giải tích, chúng ta hãy tưởng tượng những gì xảy ra khi chúng ta thay đổi giá trị bằng một chút nhỏ. Từ nhận xét ở trên, chúng tôi biết rằng

$$F(x+\epsilon) - F(x) = \int_x^{x+\epsilon} f(y)\,dy. \tag{19.5.7}$$

Điều này cho chúng ta biết rằng chức năng thay đổi bởi khu vực dưới một mảnh nhỏ của một hàm.

Đây là điểm mà chúng tôi thực hiện một xấp xỉ. Nếu chúng ta nhìn vào một mảnh nhỏ của khu vực như thế này, có vẻ như khu vực này gần với khu vực hình chữ nhật với chiều cao giá trị $f(x)$ và chiều rộng cơ sở ϵ. Thật vậy, người ta có thể cho thấy rằng như $\epsilon \to 0$ xấp xỉ này trở nên tốt hơn và tốt hơn. Vì vậy chúng ta có thể kết luận:

$$F(x+\epsilon) - F(x) \approx \epsilon f(x). \tag{19.5.8}$$

Tuy nhiên, bây giờ chúng ta có thể nhận thấy: đây chính xác là mô hình mà chúng ta mong đợi nếu chúng ta đang tính toán phái sinh của F! Vì vậy, chúng ta thấy thực tế khá đáng ngạc nhiên sau đây:

$$\frac{dF}{dx}(x) = f(x). \tag{19.5.9}$$

Đây là định lý * cơ bản của giải tí*. Chúng tôi có thể viết nó ở dạng mở rộng như

$$\frac{d}{dx}\int_{-\infty}^{x} f(y)\,dy = f(x). \tag{19.5.10}$$

Nó có khái niệm tìm kiếm các khu vực (*a priori* khá khó), và giảm nó thành một dẫn xuất tuyên bố (một cái gì đó hoàn toàn hiểu hơn nhiều). Một nhận xét cuối cùng mà chúng ta phải đưa ra là điều này không cho chúng ta biết chính xác $F(x)$ là gì. Thật vậy $F(x) + C$ cho bất kỳ C nào có cùng phái sinh. Đây là một thực tế của cuộc sống trong lý thuyết hội nhập. Rất may, lưu ý rằng khi làm việc với tích phân xác định, các hằng số sẽ bỏ ra, và do đó không liên quan đến kết quả.

$$\int_a^b f(x)\,dx = (F(b) + C) - (F(a) + C) = F(b) - F(a). \tag{19.5.11}$$

Điều này có vẻ như là vô nghĩa trừu tượng, nhưng chúng ta hãy dành một chút thời gian để đánh giá cao rằng nó đã cho chúng ta một quan điểm hoàn toàn mới về tích phân tính toán. Mục tiêu của chúng tôi là không còn để thực hiện một số loại quá trình chop-and-sum để thử và phục hồi khu vực, thay vào đó chúng ta chỉ cần tìm một hàm mà đạo hàm là hàm chúng ta có! Điều này thật đáng kinh ngạc vì bây giờ chúng ta có thể liệt kê nhiều tích phân khá khó khăn bằng cách đảo ngược bảng từ Section 19.3.2. Ví dụ, chúng ta biết rằng đạo hàm của x^n là nx^{n-1}. Như vậy, chúng ta có thể nói bằng cách sử dụng định lý cơ bản (19.5.10) rằng

$$\int_0^x ny^{n-1}\,dy = x^n - 0^n = x^n. \tag{19.5.12}$$

Tương tự, chúng ta biết rằng đạo hàm của e^x là chính nó, vì vậy điều đó có nghĩa là

$$\int_0^x e^x \, dx = e^x - e^0 = e^x - 1. \tag{19.5.13}$$

Bằng cách này, chúng ta có thể phát triển toàn bộ lý thuyết hội nhập tận dụng các ý tưởng từ phép tính vi phân một cách tự do. Mỗi quy tắc tích hợp bắt nguồn từ một thực tế này.

19.5.3 Thay đổi các biến

Cũng giống như với sự khác biệt, có một số quy tắc làm cho việc tính toán tích phân có thể truy xuất hơn. Trên thực tế, mọi quy tắc của phép tính vi phân (như quy tắc sản phẩm, quy tắc tổng, và quy tắc chuỗi) đều có quy tắc tương ứng cho phép tích phân (tích hợp theo các phần, tuyến tính tích hợp, và sự thay đổi công thức biến tương ứng). Trong phần này, chúng ta sẽ đi sâu vào những gì được cho là quan trọng nhất từ danh sách: sự thay đổi của công thức biến.

Đầu tiên, giả sử rằng chúng ta có một hàm mà chính nó là một tích phân:

$$F(x) = \int_0^x f(y) \, dy. \tag{19.5.14}$$

Chúng ta hãy giả sử rằng chúng ta muốn biết hàm này trông như thế nào khi chúng ta soạn nó với một chức năng khác để có được $F(u(x))$. Theo quy tắc chuỗi, chúng ta biết

$$\frac{d}{dx}F(u(x)) = \frac{dF}{du}(u(x)) \cdot \frac{du}{dx}. \tag{19.5.15}$$

Chúng ta có thể biến điều này thành một tuyên bố về hội nhập bằng cách sử dụng định lý cơ bản (19.5.10) như trên. Điều này cho

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{19.5.16}$$

Nhắc lại rằng F chính nó là một tích phân cho rằng phía bên tay trái có thể được viết lại để được

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} \, dy. \tag{19.5.17}$$

Tương tự, nhắc lại rằng F là một tích phân cho phép chúng ta nhận ra rằng $\frac{dF}{dx} = f$ sử dụng định lý cơ bản (19.5.10), và do đó chúng ta có thể kết luận

$$\int_{u(0)}^{u(x)} f(y) \, dy = \int_0^x f(u(y)) \cdot \frac{du}{dy} \, dy. \tag{19.5.18}$$

Đây là công thức *thay đổi biến*.

Để có một nguồn gốc trực quan hơn, hãy xem xét những gì xảy ra khi chúng ta lấy một tích phân của $f(u(x))$ giữa x và $x + \epsilon$. Đối với một ϵ nhỏ, tích phân này là khoảng $\epsilon f(u(x))$, diện tích của hình chữ nhật liên quan. Bây giờ, chúng ta hãy so sánh điều này với tích phân của $f(y)$ từ $u(x)$ đến $u(x + \epsilon)$. Chúng ta biết rằng $u(x + \epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$, vì vậy diện tích của hình chữ nhật này là khoảng $\epsilon \frac{du}{dx}(x) f(u(x))$. Do đó, để làm cho diện tích của hai hình chữ nhật này đồng ý, chúng ta cần nhân hình đầu tiên với $\frac{du}{dx}(x)$ như được minh họa trong Fig. 19.5.2.

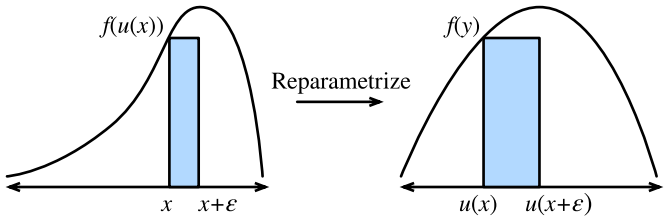


Fig. 19.5.2: Visualizing the transformation of a single thin rectangle under the change of variables.

Điều này cho chúng ta biết rằng

$$\int_{x}^{x+\epsilon} f(u(y))\frac{du}{dy}(y)\,dy = \int_{u(x)}^{u(x+\epsilon)} f(y)\,dy. \tag{19.5.19}$$

Đây là sự thay đổi của công thức biến thể hiện cho một hình chữ nhật nhỏ duy nhất.

Nếu $u(x)$ và $f(x)$ được chọn đúng cách, điều này có thể cho phép tính toán các tích phân cực kỳ phức tạp. Ví dụ, nếu chúng ta thậm chí chọn $f(y) = 1$ và $u(x) = e^{-x^2}$ (có nghĩa là $\frac{du}{dx}(x) = -2xe^{-x^2}$), điều này có thể cho thấy ví dụ

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1\,dy = -2\int_{0}^{1} ye^{-y^2}\,dy, \tag{19.5.20}$$

và do đó bằng cách sắp xếp lại điều đó

$$\int_{0}^{1} ye^{-y^2}\,dy = \frac{1-e^{-1}}{2}. \tag{19.5.21}$$

19.5.4 Một bình luận về Sign Conventions

Độc giả mắt quan sát sẽ quan sát một điều gì đó kỳ lạ về các tính toán ở trên. Cụ thể là, tính toán như

$$\int_{e^{-0}}^{e^{-1}} 1\,dy = e^{-1} - 1 < 0, \tag{19.5.22}$$

can produce sản xuất negative âm numbers số. Khi suy nghĩ về các khu vực, có thể lạ khi thấy một giá trị âm, và vì vậy nó đáng để đào sâu vào quy ước là gì.

Các nhà toán học lấy khái niệm về các khu vực có chữ ký. Điều này thể hiện theo hai cách. Đầu tiên, nếu chúng ta xem xét một hàm $f(x)$ đôi khi nhỏ hơn 0, thì khu vực cũng sẽ âm. Vì vậy, ví dụ

$$\int_{0}^{1} (-1)\,dx = -1. \tag{19.5.23}$$

Tương tự như vậy, tích phân tiến từ phải sang trái, thay vì trái sang phải cũng được coi là các khu vực âm

$$\int_{0}^{-1} 1\,dx = -1. \tag{19.5.24}$$

Khu vực tiêu chuẩn (từ trái sang phải của một hàm tích cực) luôn là tích cực. Bất cứ thứ gì thu được bằng cách lật nó (giả sử lật qua trục x-để lấy tích phân của một số âm, hoặc lật qua trục y-để lấy tích phân theo thứ

tự sai) sẽ tạo ra một vùng âm. Và thực sự, lật hai lần sẽ đưa ra một cặp dấu hiệu tiêu cực hủy bỏ để có khu vực tích cực

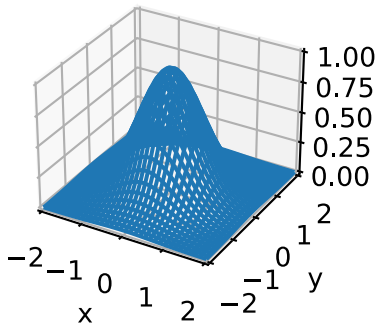
$$\int_0^{-1} (-1)\,dx = 1. \tag{19.5.25}$$

Nếu cuộc thảo luận này nghe có vẻ quen thuộc, đó là! Trong Section 19.1, chúng tôi đã thảo luận về cách yếu tố quyết định đại diện cho khu vực đã ký theo cùng một cách.

19.5.5 Nhiều tích hợp Trong một số trường hợp, chúng ta sẽ cần phải làm việc ở các kích thước cao hơn. Ví dụ, giả sử rằng chúng ta có một hàm của hai biến, như $f(x, y)$ và chúng ta muốn biết khối lượng dưới f khi x phạm vi trên $[a, b]$ và y phạm vi trên $[c, d]$.

```
# Construct grid and compute function
x, y = torch.meshgrid(torch.linspace(-2, 2, 101), torch.linspace(-2, 2, 101))
z = torch.exp(- x**2 - y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y')
d2l.plt.xticks([-2, -1, 0, 1, 2])
d2l.plt.yticks([-2, -1, 0, 1, 2])
d2l.set_figsize()
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_zlim(0, 1)
ax.dist = 12
```

```
/home/d2l-worker/miniconda3/envs/d2l-vi-release-1/lib/python3.8/site-packages/
↪torch/functional.py:568: UserWarning: torch.meshgrid: in an upcoming␣
↪release, it will be required to pass the indexing argument. (Triggered␣
↪internally at  ../aten/src/ATen/native/TensorShape.cpp:2228.)
  return _VF.meshgrid(tensors, **kwargs)  # type: ignore[attr-defined]
```



Chúng tôi viết cái này là

$$\int_{[a,b]\times[c,d]} f(x, y)\,dx\,dy. \tag{19.5.26}$$

Giả sử rằng chúng ta muốn tính toán tích phân này. Tuyên bố của tôi là chúng ta có thể làm điều này bằng cách tính toán lặp đi lặp lại đầu tiên tích phân trong x và sau đó chuyển sang tích phân trong y, có nghĩa là

$$\int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy = \int_c^d \left(\int_a^b f(x,y)\,dx\right)\,dy. \tag{19.5.27}$$

Hãy để chúng tôi xem tại sao điều này là.

Hãy xem xét hình trên nơi chúng ta đã chia hàm thành $\epsilon \times \epsilon$ ô vuông mà chúng ta sẽ lập chỉ mục với tọa độ số nguyên i, j. Trong trường hợp này, tích phân của chúng tôi là khoảng

$$\sum_{i,j} \epsilon^2 f(\epsilon i, \epsilon j). \tag{19.5.28}$$

Khi chúng tôi phân biệt vấn đề, chúng tôi có thể thêm các giá trị trên các ô vuông này theo bất kỳ thứ tự nào chúng tôi thích và không lo lắng về việc thay đổi các giá trị. Điều này được minh họa trong Fig. 19.5.3. Đặc biệt, chúng ta có thể nói rằng

$$\sum_j \epsilon \left(\sum_i \epsilon f(\epsilon i, \epsilon j)\right). \tag{19.5.29}$$

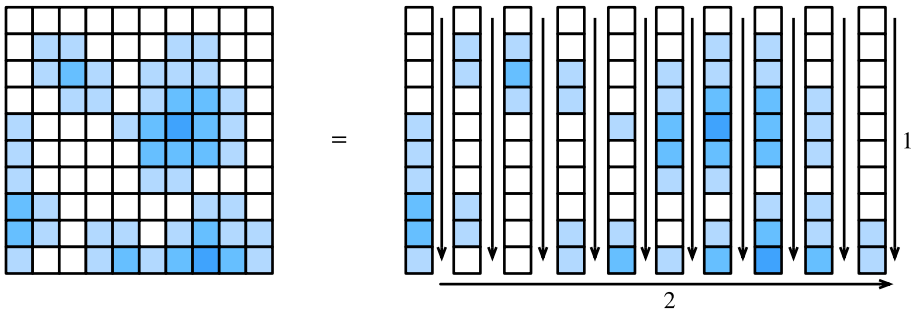


Fig. 19.5.3: Illustrating how to decompose a sum over many squares as a sum over first the columns (1), then adding the column sums together (2).

Tổng ở bên trong chính xác là sự khác biệt của tích phân

$$G(\epsilon j) = \int_a^b f(x, \epsilon j)\,dx. \tag{19.5.30}$$

Cuối cùng, hãy chú ý rằng nếu chúng ta kết hợp hai biểu thức này, chúng ta sẽ nhận được

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y)\,dy = \int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy. \tag{19.5.31}$$

Vì vậy, đặt tất cả lại với nhau, chúng tôi có điều đó

$$\int_{[a,b]\times[c,d]} f(x,y)\,dx\,dy = \int_c^d \left(\int_a^b f(x,y)\,dx\right)\,dy. \tag{19.5.32}$$

Lưu ý rằng, một khi kín đáo, tất cả những gì chúng tôi đã làm là sắp xếp lại thứ tự mà chúng tôi đã thêm một danh sách các số. Điều này có thể làm cho nó có vẻ như không có gì, tuy nhiên kết quả này (được gọi là *Fubini's Theorem*) không phải lúc nào cũng đúng! Đối với loại toán học gặp phải khi thực hiện học máy

(hàm liên tục), không có mối quan tâm, tuy nhiên có thể tạo ra các ví dụ mà nó không thành công (ví dụ hàm $f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3$ trên hình chữ nhật $[0, 2] \times [0, 1]$).

Lưu ý rằng sự lựa chọn để thực hiện tích phân trong x đầu tiên, và sau đó tích phân trong y là tùy ý. Chúng ta có thể chọn tốt như nhau để làm y đầu tiên và sau đó là x để xem

$$\int_{[a,b]\times[c,d]} f(x, y)\, dx\, dy = \int_a^b \left(\int_c^d f(x, y)\, dy \right) dx. \tag{19.5.33}$$

Thông thường, chúng ta sẽ ngưng tụ thành ký hiệu vector, và nói rằng đối với $U = [a, b] \times [c, d]$ đây là

$$\int_U f(\mathbf{x})\, d\mathbf{x}. \tag{19.5.34}$$

19.5.6 Thay đổi biến trong nhiều tích phân Như với các biến đơn trong (19.5.18), khả năng thay đổi các biến bên trong tích phân chiều cao hơn là một công cụ quan trọng. Hãy để chúng tôi tóm tắt kết quả mà không có nguồn gốc.

Chúng ta cần một chức năng reparameterizes miền tích hợp của chúng ta. Chúng ta có thể lấy điều này là $\phi : \mathbb{R}^n \to \mathbb{R}^n$, đó là bất kỳ chức năng mà mất trong n biến thực và trả về n khác. Để giữ cho các biểu thức sạch sẽ, chúng ta sẽ giả định rằng ϕ là * injective* đó là để nói rằng nó không bao giờ tự gấp lại ($\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$).

In this case, we can say that

$$\int_{\phi(U)} f(\mathbf{x})\, d\mathbf{x} = \int_U f(\phi(\mathbf{x}))\, |\det(D\phi(\mathbf{x}))|\, d\mathbf{x}. \tag{19.5.35}$$

where $D\phi$ is the *Jacobian* of ϕ, which is the matrix of partial derivatives of $\boldsymbol{\phi} = (\phi_1(x_1, \ldots, x_n), \ldots, \phi_n(x_1, \ldots, x_n))$,

$$D\boldsymbol{\phi} = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \cdots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \cdots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \tag{19.5.36}$$

Looking closely, we see that this is similar to the single variable chain rule (19.5.18), except we have replaced the term $\frac{du}{dx}(x)$ with $|\det(D\phi(\mathbf{x}))|$. Let us see how we can to interpret this term. Recall that the $\frac{du}{dx}(x)$ term existed to say how much we stretched our x-axis by applying u. The same process in higher dimensions is to determine how much we stretch the area (or volume, or hyper-volume) of a little square (or little *hyper-cube*) by applying $\boldsymbol{\phi}$. If $\boldsymbol{\phi}$ was the multiplication by a matrix, then we know how the determinant already gives the answer.

With some work, one can show that the *Jacobian* provides the best approximation to a multivariable function $\boldsymbol{\phi}$ at a point by a matrix in the same way we could approximate by lines or planes with derivatives and gradients. Thus the determinant of the Jacobian exactly mirrors the scaling factor we identified in one dimension.

It takes some work to fill in the details to this, so do not worry if they are not clear now. Let us see at least one example we will make use of later on. Consider the integral

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2 - y^2}\, dx\, dy. \tag{19.5.37}$$

Playing with this integral directly will get us no-where, but if we change variables, we can make significant progress. If we let $\boldsymbol{\phi}(r, \theta) = (r\cos(\theta), r\sin(\theta))$ (which is to say that $x = r\cos(\theta)$, $y = r\sin(\theta)$), then we

can apply the change of variable formula to see that this is the same thing as

$$\int_0^\infty \int_0^{2\pi} e^{-r^2} \left|\det(D\phi(\mathbf{x}))\right| \, d\theta \, dr, \tag{19.5.38}$$

where

$$\left|\det(D\phi(\mathbf{x}))\right| = \left|\det \begin{bmatrix} \cos(\theta) & -r\sin(\theta) \\ \sin(\theta) & r\cos(\theta) \end{bmatrix}\right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \tag{19.5.39}$$

Thus, the integral is

$$\int_0^\infty \int_0^{2\pi} re^{-r^2} \, d\theta \, dr = 2\pi \int_0^\infty re^{-r^2} \, dr = \pi, \tag{19.5.40}$$

where the final equality follows by the same computation that we used in section Section 19.5.3.

We will meet this integral again when we study continuous random variables in Section 19.6.

19.5.7 Tóm tắt

- Lý thuyết hội nhập cho phép chúng ta trả lời các câu hỏi về các khu vực hoặc khối lượng.
- Định lý cơ bản của giải tích cho phép chúng ta tận dụng kiến thức về các dẫn xuất để tính toán các khu vực thông qua quan sát rằng đạo hàm của khu vực lên đến một số điểm được đưa ra bởi giá trị của hàm được tích hợp.
- Tích phân trong các kích thước cao hơn có thể được tính toán bằng cách lặp các tích phân biến đơn lẻ.

Exercises

1. What is $\int_1^2 \frac{1}{x} \, dx$?
2. Use the change of variables formula to integrate $\int_0^{\sqrt{\pi}} x\sin(x^2) \, dx$.
3. What is $\int_{[0,1]^2} xy \, dx \, dy$?
4. Use the change of variables formula to compute $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 \, dy \, dx$ and $\int_0^1 \int_0^2 f(x,y) = xy(x^2 - y^2)/(x^2 + y^2)^3 \, dx \, dy$ to see they are different.

Discussions[217]

19.6 Biến ngẫu nhiên

Trong Section 3.6, chúng ta đã thấy những điều cơ bản về cách làm việc với các biến ngẫu nhiên rời rạc, trong trường hợp của chúng ta đề cập đến những biến ngẫu nhiên đó lấy một tập hợp hữu hạn các giá trị có thể hoặc các số nguyên. Trong phần này, chúng ta phát triển lý thuyết về biến ngẫu nhiên *liên tục*, là các biến ngẫu nhiên có thể mang theo bất kỳ giá trị thực nào.

[217] https://discuss.d2l.ai/t/1092

19.6.1 Biến ngẫu nhiên liên tục

Biến ngẫu nhiên liên tục là một chủ đề tinh tế hơn đáng kể so với các biến ngẫu nhiên rời rạc. Một sự tương tự công bằng để thực hiện là bước nhảy kỹ thuật có thể so sánh với bước nhảy giữa việc thêm danh sách các số và các chức năng tích hợp. Như vậy, chúng ta sẽ cần phải mất một thời gian để phát triển lý thuyết.

Từ rời rạc đến liên tục

Để hiểu những thách thức kỹ thuật bổ sung gặp phải khi làm việc với các biến ngẫu nhiên liên tục, chúng ta hãy thực hiện một thử nghiệm tư tưởng. Giả sử rằng chúng ta đang ném một phi tiêu vào bảng phi tiêu, và chúng tôi muốn biết xác suất rằng nó chạm chính xác 2cm từ trung tâm của bảng.

Để bắt đầu, chúng tôi tưởng tượng việc đo một chữ số chính xác duy nhất, nghĩa là với các thùng cho 0cm, 1cm, 2cm, v.v. Chúng tôi ném nói 100 phi tiêu vào bảng phi tiêu, và nếu 20 trong số họ rơi vào thùng cho 2cm, chúng tôi kết luận rằng 20% of the darts we throw hit the board 2 $vnbncm$ cách trung tâm.

Tuy nhiên, khi chúng ta nhìn kỹ hơn, điều này không phù hợp với câu hỏi của chúng tôi! Chúng tôi muốn bình đẳng chính xác, trong khi những thùng chứa tất cả những gì rơi vào giữa 1.5cm và 2.5cm.

Không ngăn cản, chúng tôi tiếp tục xa hơn. Chúng tôi đo thậm chí chính xác hơn, nói 1.9cm, 2.0cm, 2.1cm, và bây giờ thấy rằng có lẽ 3 của phi tiêu 100 đánh vào bảng trong xô 2.0cm. Vì vậy, chúng tôi kết luận xác suất là 3%.

Tuy nhiên, điều này không giải quyết được bất cứ điều gì! Chúng tôi vừa đẩy vấn đề xuống một chữ số xa hơn. Hãy để chúng tôi trừu tượng một chút. Hãy tưởng tượng chúng ta biết xác suất k chữ số đầu tiên khớp với $2.00000\ldots$ và chúng tôi muốn biết xác suất nó khớp với $k+1$ chữ số đầu tiên. Khá hợp lý khi giả định rằng chữ số $k+1^{\text{th}}$ về cơ bản là một lựa chọn ngẫu nhiên từ bộ $\{0, 1, 2, \ldots, 9\}$. Ít nhất, chúng ta không thể hình thành một quá trình có ý nghĩa về thể chất, điều này sẽ buộc số lượng micromet tạo thành trung tâm thích kết thúc trong một 7 so với 3.

Điều này có nghĩa là về bản chất mỗi chữ số bổ sung độ chính xác mà chúng ta yêu cầu sẽ giảm xác suất khớp theo hệ số 10. Hoặc đặt một cách khác, chúng tôi sẽ mong đợi rằng

$$P(\text{distance is } 2.00\ldots, \text{ to } k \text{ digits}) \approx p \cdot 10^{-k}. \tag{19.6.1}$$

Giá trị p về cơ bản mã hóa những gì xảy ra với vài chữ số đầu tiên và 10^{-k} xử lý phần còn lại.

Lưu ý rằng nếu chúng ta biết vị trí chính xác đến $k = 4$ chữ số sau thập phân. Điều đó có nghĩa là chúng ta biết giá trị nằm trong khoảng thời gian nói $[(1.99995, 2.00005]$ đó là một khoảng thời gian dài $2.00005 - 1.99995 = 10^{-4}$. Do đó, nếu chúng ta gọi độ dài của khoảng thời gian này ϵ, chúng ta có thể nói

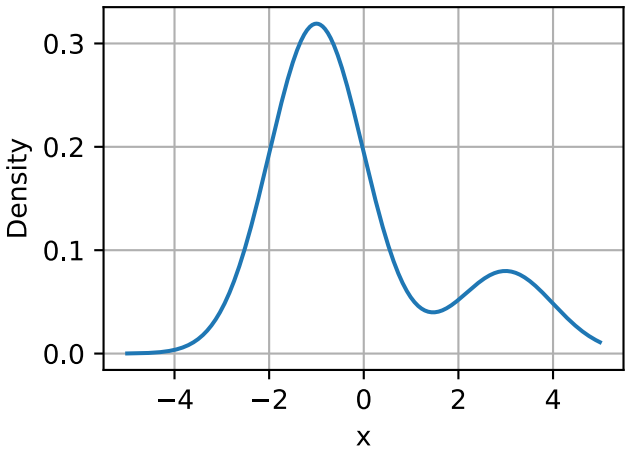
$$P(\text{distance is in an } \epsilon\text{-sized interval around } 2) \approx \epsilon \cdot p. \tag{19.6.2}$$

Hãy để chúng tôi thực hiện một bước cuối cùng này hơn nữa. Chúng tôi đã suy nghĩ về điểm 2 toàn bộ thời gian, nhưng không bao giờ nghĩ về những điểm khác. Không có gì khác nhau ở đó về cơ bản, nhưng đó là trường hợp giá trị p có thể sẽ khác nhau. Ít nhất chúng tôi hy vọng rằng một người ném phi tiêu có nhiều khả năng đạt một điểm gần trung tâm, như 2cm hơn là 20cm. Do đó, giá trị p không cố định, mà phải phụ thuộc vào điểm x. Điều này cho chúng ta biết rằng chúng ta nên mong đợi

$$P(\text{distance is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{19.6.3}$$

Thật vậy, (19.6.3) xác định chính xác hàm mật độ *xác suất*. Nó là một chức năng $p(x)$ mã hóa xác suất tương đối của đánh gần một điểm so với điểm khác. Hãy để chúng tôi hình dung những gì một chức năng như vậy có thể trông như thế nào.

```
%matplotlib inline
import torch
from IPython import display
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)).item() * 2  # Define pi in torch

# Plot the probability density function for some random variable
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi)) + \
    0.8*torch.exp(-(x + 1)**2 / 2)/torch.sqrt(2 * torch.tensor(torch.pi))

d2l.plot(x, p, 'x', 'Density')
```



Các vị trí có giá trị hàm lớn cho biết các vùng mà chúng ta có nhiều khả năng tìm thấy giá trị ngẫu nhiên hơn. Các phần thấp là những khu vực mà chúng ta không có khả năng tìm thấy giá trị ngẫu nhiên.

Hàm mật độ xác suất

Bây giờ chúng ta hãy điều tra điều này thêm. Chúng ta đã thấy hàm mật độ xác suất là gì một cách trực giác cho một biến ngẫu nhiên X, cụ thể là hàm mật độ là một hàm $p(x)$ sao cho

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(x). \tag{19.6.4}$$

Nhưng điều này ngụ ý những gì đối với các thuộc tính của $p(x)$?

Đầu tiên, xác suất không bao giờ tiêu cực, do đó chúng ta nên mong đợi rằng $p(x) \geq 0$ là tốt.

Thứ hai, chúng ta hãy tưởng tượng rằng chúng ta cắt $\mathbb{R}$ thành một số lượng vô hạn các lát rộng ϵ, nói với lát $(\epsilon \cdot i, \epsilon \cdot (i+1)]$. Đối với mỗi trong số này, chúng tôi biết từ (19.6.4) xác suất là xấp xỉ

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x) \approx \epsilon \cdot p(\epsilon \cdot i), \tag{19.6.5}$$

vì vậy tóm tắt tất cả trong số họ nó nên được

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \tag{19.6.6}$$

Đây không gì khác hơn là xấp xỉ của một tích phân được thảo luận trong Section 19.5, do đó chúng ta có thể nói rằng

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x) \, dx. \tag{19.6.7}$$

Chúng ta biết rằng $P(X \in \mathbb{R}) = 1$, vì biến ngẫu nhiên phải đảm nhận số * some*, chúng ta có thể kết luận rằng đối với bất kỳ mật độ nào

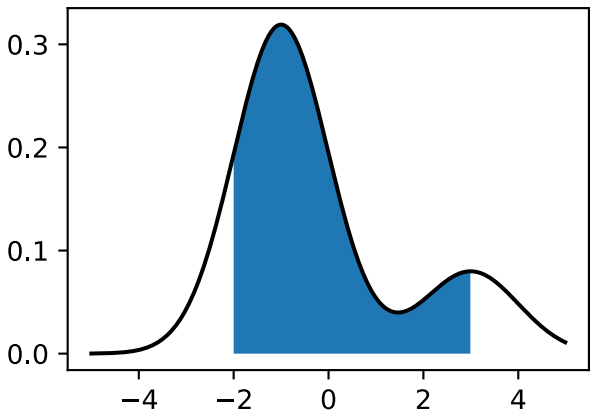
$$\int_{-\infty}^{\infty} p(x) \, dx = 1. \tag{19.6.8}$$

Thật vậy, đào sâu vào điều này cho thấy rằng đối với bất kỳ a và b, chúng ta thấy rằng

$$P(X \in (a, b]) = \int_{a}^{b} p(x) \, dx. \tag{19.6.9}$$

Chúng ta có thể xấp xỉ điều này trong mã bằng cách sử dụng các phương thức xấp xỉ rời rạc giống như trước. Trong trường hợp này, chúng ta có thể xấp xỉ xác suất rơi vào vùng màu xanh.

```
# Approximate probability using numerical integration
epsilon = 0.01
x = torch.arange(-5, 5, 0.01)
p = 0.2*torch.exp(-(x - 3)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi)) +\
    0.8*torch.exp(-(x + 1)**2 / 2) / torch.sqrt(2 * torch.tensor(torch.pi))

d2l.set_figsize()
d2l.plt.plot(x, p, color='black')
d2l.plt.fill_between(x.tolist()[300:800], p.tolist()[300:800])
d2l.plt.show()

f'approximate Probability: {torch.sum(epsilon*p[300:800])}'
```




```
'approximate Probability: 0.7736171483993530'
```

Nó chỉ ra rằng hai thuộc tính này mô tả chính xác không gian của các hàm mật độ xác suất có thể (hoặc *p.d.f.* cho chữ viết tắt thường gặp). Chúng là các chức năng không tiêu cực $p(x) \geq 0$ sao cho

$$\int_{-\infty}^{\infty} p(x) \, dx = 1. \tag{19.6.10}$$

Chúng ta diễn giải hàm này bằng cách sử dụng tích hợp để có được xác suất biến ngẫu nhiên của chúng ta nằm trong một khoảng thời gian cụ thể:

$$P(X \in (a, b]) = \int_a^b p(x) \, dx. \tag{19.6.11}$$

Trong `sec_distributions`, chúng ta sẽ thấy một số bản phân phối phổ biến, nhưng chúng ta hãy tiếp tục làm việc trong bản tóm tắt.

Hàm phân phối tích lũy

Trong phần trước, chúng ta đã thấy khái niệm p.d.f Trong thực tế, đây là một phương pháp thường gặp để thảo luận về các biến ngẫu nhiên liên tục, nhưng nó có một cạm bẫy đáng kể: rằng các giá trị của p.d.f. không phải là xác suất, mà là một hàm mà chúng ta phải tích hợp để mang lại xác suất. Không có gì sai với mật độ lớn hơn 10, miễn là nó không lớn hơn 10 trong hơn một khoảng thời gian dài $1/10$. Điều này có thể phản trực quan, vì vậy mọi người thường nghĩ về hàm phân phối tích luỹ* hoặc c.d.f., mà * là* một xác suất.

Đặc biệt, bằng cách sử dụng (19.6.11), ta định nghĩa c.d.f. cho một biến ngẫu nhiên X với mật độ $p(x)$ bởi

$$F(x) = \int_{-\infty}^x p(x) \, dx = P(X \leq x). \tag{19.6.12}$$

Hãy để chúng tôi quan sát một vài tài sản.

- $F(x) \to 0$ như $x \to -\infty$.
- $F(x) \to 1$ như $x \to \infty$.
- $F(x)$ không giảm ($y > x \implies F(y) \geq F(x)$).
- $F(x)$ là liên tục (không có nhảy) nếu X là một biến ngẫu nhiên liên tục.

Với dấu đầu dòng thứ tư, lưu ý rằng điều này sẽ không đúng nếu X rời rạc, giả sử lấy các giá trị 0 và 1 cả hai với xác suất $1/2$. Trong trường hợp đó

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \geq 1. \end{cases} \tag{19.6.13}$$

Trong ví dụ này, chúng ta thấy một trong những lợi ích của việc làm việc với c.d.f., khả năng đối phó với các biến ngẫu nhiên liên tục hoặc rời rạc trong cùng một khuôn khổ, hoặc thực sự hỗn hợp của hai (lật một đồng xu: nếu đầu trả lại cuộn chết, nếu đuôi trả lại khoảng cách ném phi tiêu từ trung tâm của một phi tiêu bảng).

Phương tiện

Giả sử rằng chúng ta đang đối phó với một biến ngẫu nhiên X. Bản thân phân phối có thể khó giải thích. Nó thường hữu ích để có thể tóm tắt hành vi của một biến ngẫu nhiên một cách chính xác. Các số giúp chúng ta nắm bắt hành vi của một biến ngẫu nhiên được gọi là *thống kê tóm lượng*. Những cái thường gặp nhất là *mean*, *variance*, và * độ lệch chuẩn*.

mean mã hóa giá trị trung bình của một biến ngẫu nhiên. Nếu chúng ta có một biến ngẫu nhiên rời rạc X, lấy các giá trị x_i với xác suất p_i, thì trung bình được đưa ra bởi trung bình có trọng số: tổng các giá trị lần xác suất biến ngẫu nhiên có giá trị đó:

$$\mu_X = E[X] = \sum_i x_i p_i. \tag{19.6.14}$$

Cách chúng ta nên giải thích trung bình (mặc dù thận trọng) là nó cho chúng ta biết về cơ bản nơi biến ngẫu nhiên có xu hướng được đặt.

Như một ví dụ tối giản mà chúng ta sẽ kiểm tra trong suốt phần này, chúng ta hãy lấy X là biến ngẫu nhiên lấy giá trị $a-2$ với xác suất p, $a+2$ với xác suất p và a với xác suất $1-2p$. Chúng ta có thể tính toán bằng cách sử dụng (19.6.14) rằng, cho bất kỳ lựa chọn nào có thể là a và p, trung bình là

$$\mu_X = E[X] = \sum_i x_i p_i = (a-2)p + a(1-2p) + (a+2)p = a. \tag{19.6.15}$$

Vì vậy, chúng ta thấy rằng trung bình là a. Điều này phù hợp với trực giác kể từ a là vị trí xung quanh mà chúng tôi tập trung biến ngẫu nhiên của chúng tôi.

Bởi vì chúng rất hữu ích, chúng ta hãy tóm tắt một vài thuộc tính.

- Đối với bất kỳ biến ngẫu nhiên X và số a và b, chúng tôi có $\mu_{aX+b} = a\mu_X + b$.
- Nếu chúng ta có hai biến ngẫu nhiên X và Y, chúng ta có $\mu_{X+Y} = \mu_X + \mu_Y$.

Phương tiện rất hữu ích để hiểu hành vi trung bình của một biến ngẫu nhiên, tuy nhiên trung bình là không đủ để thậm chí có một sự hiểu biết trực quan đầy đủ. Kiếm lợi nhuận $$10 pm 1 mỗi lần bán rất khác so với việc tạo ra $$10 pm 15 mỗi lần bán mặc dù có cùng giá trị trung bình. Cái thứ hai có mức độ dao động lớn hơn nhiều, và do đó đại diện cho rủi ro lớn hơn nhiều. Do đó, để hiểu được hành vi của một biến ngẫu nhiên, chúng ta sẽ cần tối thiểu thêm một thước đo nữa: một số thước đo về cách biến ngẫu nhiên dao động rộng rãi.

Phương sai

Điều này dẫn chúng ta xem xét *variance* của một biến ngẫu nhiên. Đây là một thước đo định lượng về cách xa một biến ngẫu nhiên lệch khỏi trung bình. Hãy xem xét biểu thức $X - \mu_X$. Đây là độ lệch của biến ngẫu nhiên so với trung bình của nó. Giá trị này có thể là dương hoặc âm, vì vậy chúng ta cần phải làm một cái gì đó để làm cho nó tích cực để chúng ta đo độ lớn của độ lệch.

Một điều hợp lý để thử là nhìn vào $|X - \mu_X|$, và thực sự điều này dẫn đến một đại lượng hữu ích gọi là độ lệch tuyệt đối * trung bình, tuy nhiên do kết nối với các lĩnh vực toán học và thống kê khác, người ta thường sử dụng một giải pháp khác.

Đặc biệt, họ nhìn vào $(X - \mu_X)^2$. Nếu chúng ta nhìn vào kích thước điển hình của số lượng này bằng cách lấy trung bình, chúng tôi đến phương sai

$$\sigma_X^2 = \mathrm{Var}(X) = E\left[(X - \mu_X)^2\right] = E[X^2] - \mu_X^2. \tag{19.6.16}$$

Sự bình đẳng cuối cùng trong (19.6.16) nắm giữ bằng cách mở rộng định nghĩa ở giữa và áp dụng các thuộc tính của kỳ vọng.

Chúng ta hãy nhìn vào ví dụ của chúng tôi nơi X là biến ngẫu nhiên mà lấy giá trị $a-2$ với xác suất p, $a+2$ với xác suất p và a với xác suất $1-2p$. Trong trường hợp này $\mu_X = a$, vì vậy tất cả chúng ta cần phải tính toán là $E\left[X^2\right]$. Điều này có thể dễ dàng được thực hiện:

$$E\left[X^2\right] = (a-2)^2 p + a^2(1-2p) + (a+2)^2 p = a^2 + 8p. \tag{19.6.17}$$

Do đó, chúng ta thấy rằng bởi (19.6.16) phương sai của chúng tôi là

$$\sigma_X^2 = \mathrm{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \tag{19.6.18}$$

Kết quả này một lần nữa có ý nghĩa. p lớn nhất có thể là $1/2$ tương ứng với việc chọn $a-2$ hoặc $a+2$ với lật đồng xu. Phương sai của việc này là 4 tương ứng với thực tế là cả $a-2$ và $a+2$ đều là 2 đơn vị cách xa

trung bình và $2^2 = 4$. Ở đầu kia của quang phổ, nếu $p = 0$, biến ngẫu nhiên này luôn lấy giá trị 0 và do đó nó không có phương sai nào cả.

Chúng tôi sẽ liệt kê một vài thuộc tính của phương sai dưới đây:

- Đối với bất kỳ biến ngẫu nhiên X, $\mathrm{Var}(X) \geq 0$, với $\mathrm{Var}(X) = 0$ nếu và chỉ khi X là một hằng số.
- Đối với bất kỳ biến ngẫu nhiên X và số a và b, chúng tôi có $\mathrm{Var}(aX + b) = a^2\mathrm{Var}(X)$.
- Nếu chúng ta có hai biến ngẫu nhiên * độc lập* X và Y, chúng ta có $\mathrm{Var}(X+Y) = \mathrm{Var}(X)+\mathrm{Var}(Y)$.

Khi giải thích các giá trị này, có thể có một chút nấc cục. Đặc biệt, chúng ta hãy thử tưởng tượng điều gì sẽ xảy ra nếu chúng ta theo dõi các đơn vị thông qua tính toán này. Giả sử rằng chúng tôi đang làm việc với xếp hạng sao được gán cho một sản phẩm trên trang web. Sau đó, a, $a - 2$, và $a + 2$ đều được đo bằng đơn vị sao. Tương tự, trung bình μ_X sau đó cũng được đo bằng sao (là trung bình có trọng số). Tuy nhiên, nếu chúng ta đi đến phương sai, chúng ta ngay lập tức gặp phải một vấn đề, đó là chúng ta muốn xem xét $(X - \mu_X)^2$, đó là đơn vị của * bình phương sao*. Điều này có nghĩa là bản thân phương sai không thể so sánh với các phép đo ban đầu. Để làm cho nó có thể hiểu được, chúng ta sẽ cần phải trở lại các đơn vị ban đầu của chúng tôi.

Độ lệch chuẩn

Thống kê tóm tắt này luôn có thể được suy ra từ phương sai bằng cách lấy căn bậc hai! Do đó chúng tôi xác định độ lệch tiêu chuẩn* là

$$\sigma_X = \sqrt{\mathrm{Var}(X)}. \tag{19.6.19}$$

Trong ví dụ của chúng tôi, điều này có nghĩa là bây giờ chúng ta có độ lệch chuẩn là $\sigma_X = 2\sqrt{2p}$. Nếu chúng ta đang đối phó với các đơn vị sao cho ví dụ đánh giá của chúng tôi, σ_X một lần nữa trong các đơn vị sao.

Các thuộc tính chúng tôi có cho phương sai có thể được đặt lại cho độ lệch chuẩn.

- Đối với bất kỳ biến ngẫu nhiên X, $\sigma_X \geq 0$.
- Đối với bất kỳ biến ngẫu nhiên X và số a và b, chúng tôi có $\sigma_{aX+b} = |a|\sigma_X$
- Nếu chúng ta có hai biến ngẫu nhiên * độc lập* X và Y, chúng ta có $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$.

Nó là tự nhiên tại thời điểm này để hỏi, "Nếu độ lệch chuẩn nằm trong các đơn vị của biến ngẫu nhiên ban đầu của chúng ta, nó có đại diện cho một cái gì đó chúng ta có thể vẽ liên quan đến biến ngẫu nhiên đó?" Câu trả lời là một có vang dội! Thật vậy giống như trung bình nói với chúng ta vị trí điển hình của biến ngẫu nhiên của chúng ta, độ lệch chuẩn cho phạm vi điển hình của biến thể của biến ngẫu nhiên đó. Chúng ta có thể làm cho điều này nghiêm ngặt với cái được gọi là bất bình đẳng của Chebyshev:

$$P\left(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]\right) \leq \frac{1}{\alpha^2}. \tag{19.6.20}$$

Or to state it verbally in the case of $\alpha = 10$, 99% of the samples from any random variable fall within 10 standard deviations of the mean. This gives an immediate interpretation to our standard summary statistics.

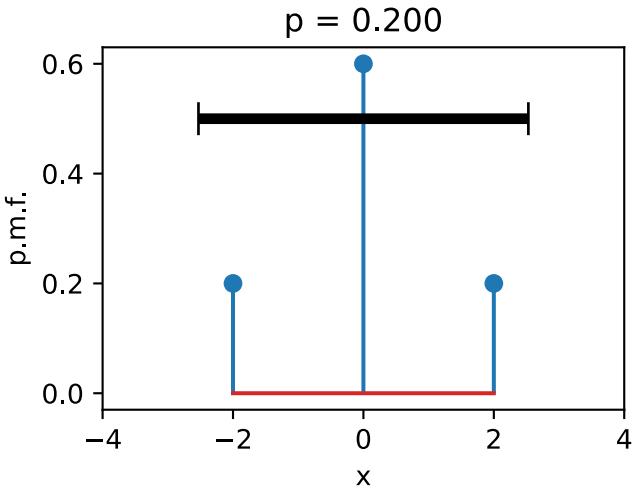
Để xem cách tuyên bố này là khá tinh tế, chúng ta hãy xem xét ví dụ chạy của chúng tôi một lần nữa nơi X là biến ngẫu nhiên mà lấy giá trị $a - 2$ với xác suất p, $a + 2$ với xác suất p và a với xác suất $1 - 2p$. Chúng tôi thấy rằng trung bình là a và độ lệch chuẩn là $2\sqrt{2p}$. Điều này có nghĩa là, nếu chúng ta lấy sự bất bình đẳng của Chebyshev (19.6.20) với $\alpha = 2$, chúng ta thấy rằng biểu thức là

$$P\left(X \notin [a - 4\sqrt{2p}, a + 4\sqrt{2p}]\right) \leq \frac{1}{4}. \tag{19.6.21}$$

This means that 75% of the time, this random variable will fall within this interval for any value of p. Now, notice that as $p \to 0$, this interval also converges to the single point a. But we know that our random variable takes the values $a-2, a$, and $a+2$ only so eventually we can be certain $a-2$ and $a+2$ will fall outside the interval! The question is, at what p does that happen. So we want to solve: for what p does $a+4\sqrt{2p} = a+2$, which is solved when $p = 1/8$, which is *exactly* the first p where it could possibly happen without violating our claim that no more than $1/4$ of samples from the distribution would fall outside the interval ($1/8$ to the left, and $1/8$ to the right).

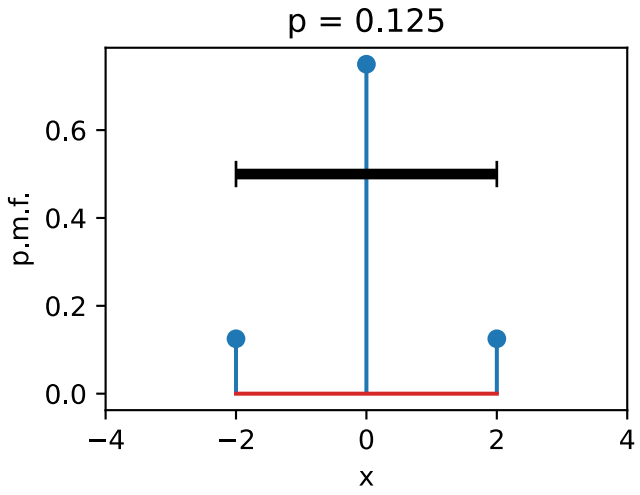
Hãy để chúng tôi hình dung điều này. Chúng tôi sẽ hiển thị xác suất nhận được ba giá trị dưới dạng ba thanh dọc với chiều cao tỷ lệ thuận với xác suất. Khoảng thời gian sẽ được vẽ dưới dạng một đường ngang ở giữa. Cốt truyện đầu tiên cho thấy những gì xảy ra cho $p > 1/8$ trong đó khoảng thời gian an toàn chứa tất cả các điểm.

```
# Define a helper to plot these figures
def plot_chebyshev(a, p):
    d2l.set_figsize()
    d2l.plt.stem([a-2, a, a+2], [p, 1-2*p, p], use_line_collection=True)
    d2l.plt.xlim([-4, 4])
    d2l.plt.xlabel('x')
    d2l.plt.ylabel('p.m.f.')

    d2l.plt.hlines(0.5, a - 4 * torch.sqrt(2 * p),
                   a + 4 * torch.sqrt(2 * p), 'black', lw=4)
    d2l.plt.vlines(a - 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.vlines(a + 4 * torch.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l.plt.title(f'p = {p:.3f}')

    d2l.plt.show()

# Plot interval when p > 1/8
plot_chebyshev(0.0, torch.tensor(0.2))
```



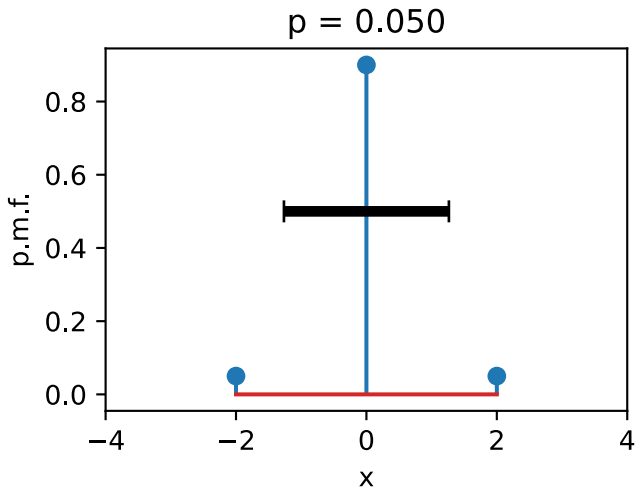
Thứ hai cho thấy ở $p = 1/8$, khoảng thời gian chính xác chạm vào hai điểm. Điều này cho thấy bất đẳng thức là * sharp*, vì không có khoảng thời gian nhỏ hơn có thể được thực hiện trong khi vẫn giữ sự bất bình đẳng đúng.

```
# Plot interval when p = 1/8
plot_chebyshev(0.0, torch.tensor(0.125))
```



Thứ ba cho thấy rằng đối với $p < 1/8$ khoảng thời gian chỉ chứa trung tâm. Điều này không làm mất hiệu lực bất bình đẳng vì chúng ta chỉ cần đảm bảo rằng không quá $1/4$ xác suất nằm ngoài khoảng thời gian, có nghĩa là một lần $p < 1/8$, hai điểm ở $a - 2$ và $a + 2$ có thể bị loại bỏ.

```
# Plot interval when p < 1/8
plot_chebyshev(0.0, torch.tensor(0.05))
```



Phương tiện và phương sai trong liên tục

Điều này đã được tất cả về các biến ngẫu nhiên rời rạc, nhưng trường hợp của các biến ngẫu nhiên liên tục là tương tự nhau. Để hiểu trực giác điều này hoạt động như thế nào, hãy tưởng tượng rằng chúng ta chia dòng số thực thành các khoảng thời gian ϵ được đưa ra bởi $(\epsilon i, \epsilon(i+1)]$. Khi chúng tôi làm điều này, biến ngẫu nhiên liên tục của chúng tôi đã được thực hiện rời rạc và chúng tôi có thể sử dụng (19.6.14) nói rằng

$$\begin{aligned}\mu_X &\approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ &\approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon,\end{aligned} \tag{19.6.22}$$

trong đó p_X là mật độ X. Đây là một xấp xỉ với tích phân của $xp_X(x)$, vì vậy chúng ta có thể kết luận rằng

$$\mu_X = \int_{-\infty}^{\infty} x p_X(x) \, dx. \tag{19.6.23}$$

Tương tự, sử dụng (19.6.16) phương sai có thể được viết là

$$\sigma_X^2 = E[X^2] - \mu_X^2 = \int_{-\infty}^{\infty} x^2 p_X(x) \, dx - \left(\int_{-\infty}^{\infty} x p_X(x) \, dx \right)^2. \tag{19.6.24}$$

Tất cả mọi thứ đã nêu ở trên về trung bình, phương sai và độ lệch chuẩn vẫn được áp dụng trong trường hợp này. Ví dụ, nếu chúng ta xem xét biến ngẫu nhiên với mật độ

$$p(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{otherwise.} \end{cases} \tag{19.6.25}$$

we can compute tính toán

$$\mu_X = \int_{-\infty}^{\infty} x p(x) \, dx = \int_0^1 x \, dx = \frac{1}{2}. \tag{19.6.26}$$

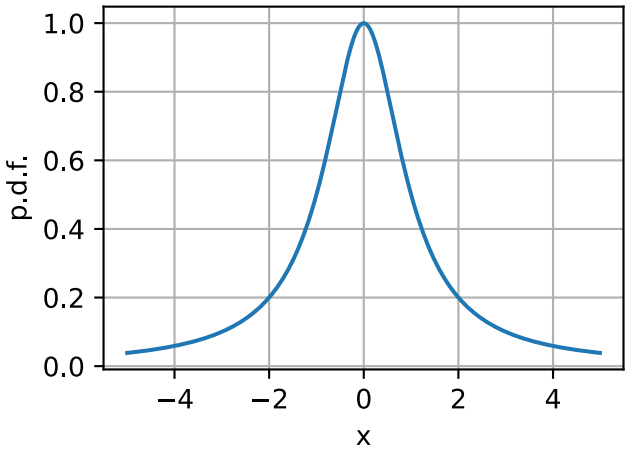
và

$$\sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x) \, dx - \left(\frac{1}{2} \right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}. \tag{19.6.27}$$

Như một cảnh báo, chúng ta hãy xem xét thêm một ví dụ, được gọi là bản phân phối *Cauchy*. Đây là bản phân phối với p.d.f. được đưa ra bởi

$$p(x) = \frac{1}{1+x^2}. \tag{19.6.28}$$

```
# Plot the Cauchy distribution p.d.f.
x = torch.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```



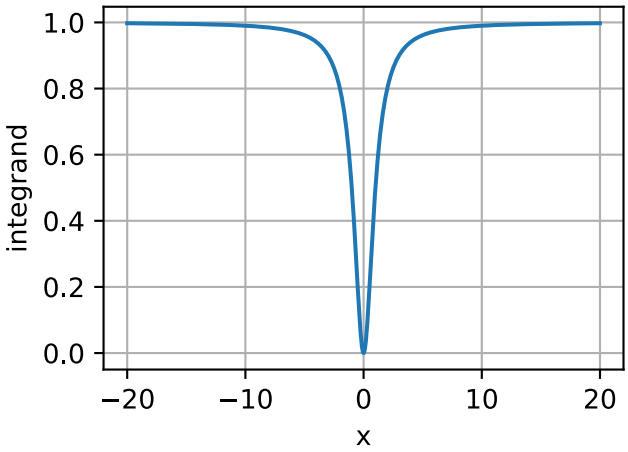
Hàm này trông vô tội, và thực sự tham khảo một bảng tích phân sẽ cho thấy nó có diện tích một bên dưới nó, và do đó nó định nghĩa một biến ngẫu nhiên liên tục.

Để xem những gì đi lạc lối, chúng ta hãy cố gắng tính toán phương sai của điều này. Điều này sẽ liên quan đến việc sử dụng máy tính (19.6.16)

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx. \tag{19.6.29}$$

Chức năng ở bên trong trông như thế này:

```
# Plot the integrand needed to compute the variance
x = torch.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```



Chức năng này rõ ràng có diện tích vô hạn dưới nó vì về cơ bản nó là hằng số với một cú nhúng nhỏ gần 0, và thực sự chúng ta có thể chỉ ra rằng

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2}\,dx = \infty. \tag{19.6.30}$$

Điều này có nghĩa là nó không có phương sai hữu hạn được xác định rõ.

Tuy nhiên, nhìn sâu hơn cho thấy một kết quả thậm chí còn đáng lo ngại hơn. Hãy để chúng tôi cố gắng tính toán trung bình bằng cách sử dụng (19.6.14). Sử dụng sự thay đổi của công thức biến, chúng ta thấy

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2} \, dx = \frac{1}{2} \int_{1}^{\infty} \frac{1}{u} \, du. \tag{19.6.31}$$

Tích phân bên trong là định nghĩa của logarit, vì vậy điều này là về bản chất $\log(\infty) = \infty$, do đó không có giá trị trung bình được xác định rõ cả!

Các nhà khoa học máy học xác định mô hình của họ để chúng ta thường không cần phải đối phó với những vấn đề này và trong phần lớn các trường hợp sẽ đối phó với các biến ngẫu nhiên với các phương tiện và phương sai được xác định rõ ràng. Tuy nhiên, mọi biến ngẫu nhiên thường có * đuôi nặng* (đó là những biến ngẫu nhiên trong đó xác suất nhận được các giá trị lớn đủ lớn để làm cho những thứ như trung bình hoặc phương sai không xác định) rất hữu ích trong việc mô hình hóa các hệ thống vật lý, do đó đáng để biết rằng chúng tồn tại.

Chức năng mật độ khớp

Công việc trên tất cả giả định chúng ta đang làm việc với một biến ngẫu nhiên có giá trị thực duy nhất. Nhưng điều gì sẽ xảy ra nếu chúng ta đang đối phó với hai hoặc nhiều biến ngẫu nhiên có khả năng tương quan cao? Hoàn cảnh này là chuẩn mực trong học máy: tưởng tượng các biến ngẫu nhiên như $R_{i,j}$ mã hóa giá trị màu đỏ của pixel tại tọa độ (i, j) trong một hình ảnh, hoặc P_t là một biến ngẫu nhiên được đưa ra bởi giá cổ phiếu tại thời điểm t. Các pixel gần đó có xu hướng có màu tương tự và thời gian gần đó có xu hướng có giá tương tự. Chúng ta không thể coi chúng như các biến ngẫu nhiên riêng biệt và mong đợi tạo ra một mô hình thành công (chúng ta sẽ thấy trong Section 19.8 một mô hình hoạt động kém do giả định như vậy). Chúng ta cần phát triển ngôn ngữ toán học để xử lý các biến ngẫu nhiên liên tục tương quan này.

Rất may, với nhiều tích phân trong Section 19.5, chúng ta có thể phát triển một ngôn ngữ như vậy. Giả sử rằng chúng ta có, để đơn giản, hai biến ngẫu nhiên X, Y có thể tương quan. Sau đó, tương tự như trường hợp của một biến duy nhất, chúng ta có thể đặt câu hỏi:

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y). \tag{19.6.32}$$

Lý luận tương tự như trường hợp biến duy nhất cho thấy điều này nên xấp xỉ

$$P(X \text{ is in an } \epsilon\text{-sized interval around } x \text{ and } Y \text{ is in an } \epsilon\text{-sized interval around } y) \approx \epsilon^2 p(x, y), \tag{19.6.33}$$

cho một số chức năng $p(x, y)$. Đây được gọi là mật độ khớp của X và Y. Các thuộc tính tương tự đúng với điều này như chúng ta đã thấy trong trường hợp biến duy nhất. Cụ thể là:

- $p(x, y) \geq 0$;
- $\int_{\mathbb{R}^2} p(x, y) \, dx \, dy = 1$;
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y) \, dx \, dy$.

Bằng cách này, chúng ta có thể đối phó với nhiều biến ngẫu nhiên có khả năng tương quan. Nếu chúng ta muốn làm việc với nhiều hơn hai biến ngẫu nhiên, chúng ta có thể mở rộng mật độ đa biến đến nhiều tọa độ như mong muốn bằng cách xem xét $p(\mathbf{x}) = p(x_1, \ldots, x_n)$. Các tính chất tương tự là không âm, và có tổng tích phân của một vẫn giữ.

Phân phối biên Khi xử lý nhiều biến, chúng ta thường muốn có thể bỏ qua các mối quan hệ và hỏi, "biến này được phân phối như thế nào?" Phân phối như vậy được gọi là phân phối cận biên *.

Để được cụ thể, chúng ta hãy giả sử rằng chúng ta có hai biến ngẫu nhiên X, Y với mật độ khớp được đưa ra bởi $p_{X,Y}(x, y)$. Chúng ta sẽ sử dụng chỉ số dưới để chỉ ra các biến ngẫu nhiên mà mật độ dùng để làm gì. Câu hỏi tìm ra sự phân bố cận biên là dùng chức năng này, và sử dụng nó để tìm $p_X(x)$.

Như với hầu hết mọi thứ, tốt nhất là quay lại hình ảnh trực quan để tìm ra những gì nên đúng. Nhớ lại rằng mật độ là chức năng p_X để

$$P(X \in [x, x + \epsilon]) \approx \epsilon \cdot p_X(x). \tag{19.6.34}$$

Không có đề cập đến Y, nhưng nếu tất cả chúng ta được đưa ra là $p_{X,Y}$, chúng ta cần bao gồm Y bằng cách nào đó. Trước tiên chúng ta có thể quan sát rằng điều này giống như

$$P(X \in [x, x + \epsilon], \text{ and } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \tag{19.6.35}$$

Mật độ của chúng tôi không trực tiếp cho chúng tôi biết về những gì xảy ra trong trường hợp này, chúng ta cần phải chia thành các khoảng nhỏ trong y là tốt, vì vậy chúng tôi có thể viết điều này như

$$\begin{aligned} \epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x + \epsilon], \text{ and } Y \in [\epsilon \cdot i, \epsilon \cdot (i + 1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i). \end{aligned} \tag{19.6.36}$$

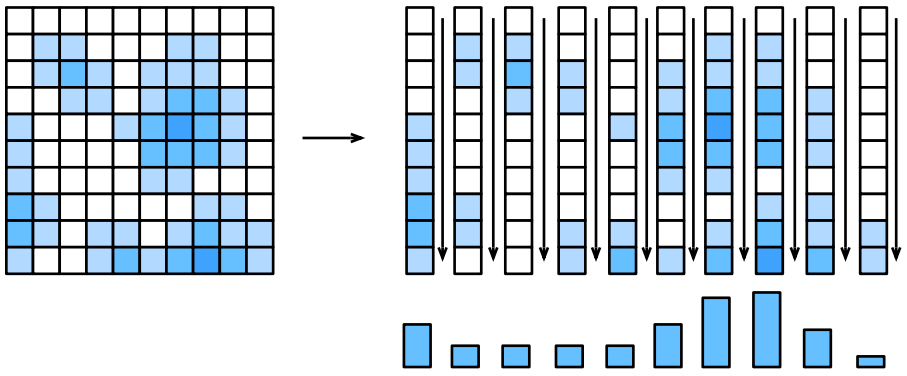


Fig. 19.6.1: By summing along the columns of our array of probabilities, we are able to obtain the marginal distribution for just the random variable represented along the x-axis.

Điều này cho chúng ta biết thêm giá trị của mật độ dọc theo một loạt các ô vuông trong một đường như được thể hiện trong Fig. 19.6.1. Thật vậy, sau khi hủy bỏ một yếu tố epsilon từ cả hai phía, và nhận ra tổng ở bên phải là tích phân trên y, chúng ta có thể kết luận rằng

$$\begin{aligned} p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y) \, dy. \end{aligned} \tag{19.6.37}$$

Vì vậy chúng ta thấy

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)\ dy. \tag{19.6.38}$$

Điều này cho chúng ta biết rằng để có được một phân phối biên, chúng tôi tích hợp trên các biến mà chúng tôi không quan tâm. Quá trình này thường được gọi là *tích hợp out* hoặc *marginalized out* các biến không cần thiết.

Hiệp phương sai

Khi xử lý nhiều biến ngẫu nhiên, có một thống kê tóm tắt bổ sung rất hữu ích để biết: * covariance*. Điều này đo mức độ mà hai biến ngẫu nhiên dao động với nhau.

Giả sử rằng chúng ta có hai biến ngẫu nhiên X và Y, để bắt đầu, chúng ta hãy giả sử chúng rời rạc, lấy giá trị (x_i, y_j) với xác suất p_{ij}. Trong trường hợp này, phương sai được định nghĩa là

$$\sigma_{XY} = \text{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij}. = E[XY] - E[X]E[Y]. \tag{19.6.39}$$

Để suy nghĩ về điều này một cách trực giác: hãy xem xét cặp biến ngẫu nhiên sau đây. Giả sử rằng X lấy các giá trị 1 và 3, và Y lấy các giá trị -1 và 3. Giả sử rằng chúng ta có xác suất sau

$$\begin{aligned} P(X = 1 \text{ and } Y = -1) &= \frac{p}{2}, \\ P(X = 1 \text{ and } Y = 3) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = -1) &= \frac{1-p}{2}, \\ P(X = 3 \text{ and } Y = 3) &= \frac{p}{2}, \end{aligned} \tag{19.6.40}$$

trong đó p là một tham số trong $[0, 1]$ chúng tôi nhận được để chọn. Lưu ý rằng nếu $p = 1$ thì cả hai đều luôn là giá trị tối thiểu hoặc tối đa của chúng cùng một lúc và nếu $p = 0$, chúng được đảm bảo lấy giá trị lật của chúng đồng thời (một giá trị lớn khi giá kia nhỏ và ngược lại). Nếu $p = 1/2$, thì bốn khả năng đều có khả năng như nhau và không nên liên quan. Let us computetính toán the covariance đồng phương sai. Đầu tiên, lưu ý $\mu_X = 2$ và $\mu_Y = 1$, vì vậy chúng tôi có thể tính toán bằng (19.6.39):

$$\begin{aligned} \text{Cov}(X, Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y) p_{ij} \\ &= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\ &= 4p - 2. \end{aligned} \tag{19.6.41}$$

Khi $p = 1$ (trường hợp cả hai đều dương tính tối đa hoặc tiêu cực cùng một lúc) có sự đồng phương sai là 2. Khi $p = 0$ (trường hợp chúng bị lật) thì đồng phương sai là -2. Cuối cùng, khi $p = 1/2$ (trường hợp chúng không liên quan), phương sai là 0. Do đó chúng ta thấy rằng sự đồng phương sai đo lường cách thức hai biến ngẫu nhiên này có liên quan.

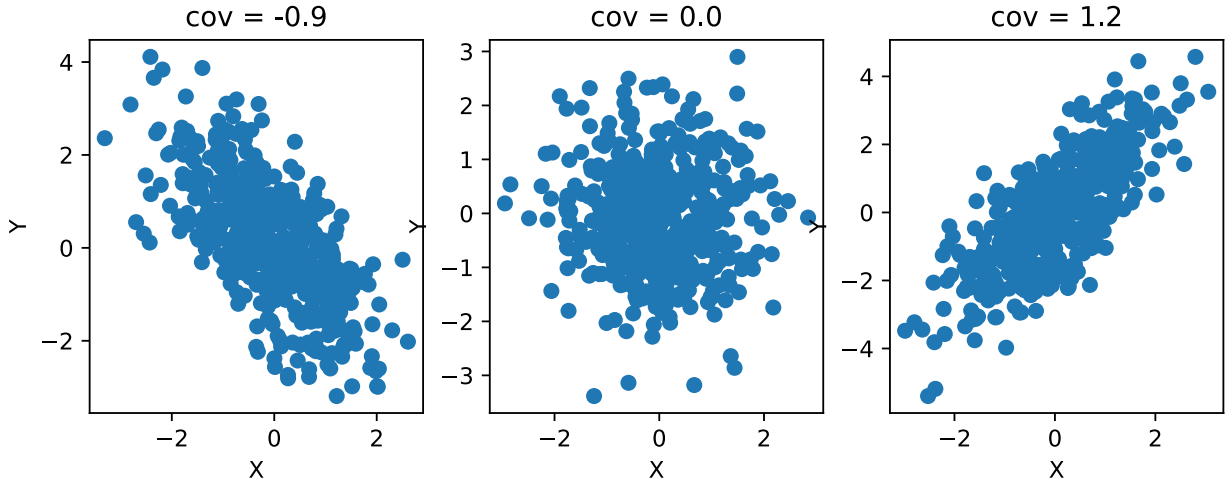
Một lưu ý nhanh về sự đồng phương sai là nó chỉ đo các mối quan hệ tuyến tính này. Các mối quan hệ phức tạp hơn như $X = Y^2$ trong đó Y được chọn ngẫu nhiên từ $\{-2, -1, 0, 1, 2\}$ với xác suất bằng nhau có thể bị bỏ qua. Thật vậy một tính toán nhanh chóng cho thấy các biến ngẫu nhiên này có sự đồng phương sai 0, mặc dù một là một hàm xác định của cái kia.

Đối với các biến ngẫu nhiên liên tục, nhiều câu chuyện tương tự giữ. Tại thời điểm này, chúng tôi khá thoải mái khi thực hiện quá trình chuyển đổi giữa rời rạc và liên tục, vì vậy chúng tôi sẽ cung cấp tương tự liên tục của (19.6.39) mà không có bất kỳ nguồn gốc nào.

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y)p(x, y) \, dx \, dy. \tag{19.6.42}$$

Để trực quan hóa, chúng ta hãy xem một tập hợp các biến ngẫu nhiên với phương sai có thể điều chỉnh.

```
# Plot a few random variables adjustable covariance
covs = [-0.9, 0.0, 1.2]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = covs[i]*X + torch.randn(500)

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```



Hãy để chúng tôi xem một số tính chất của đồng phương sai:

- Đối với bất kỳ biến ngẫu nhiên X, $\mathrm{Cov}(X, X) = \mathrm{Var}(X)$.
- Đối với bất kỳ biến ngẫu nhiên X, Y và số a và b, $\mathrm{Cov}(aX + b, Y) = \mathrm{Cov}(X, aY + b) = a\mathrm{Cov}(X, Y)$.
- Nếu X và Y độc lập thì $\mathrm{Cov}(X, Y) = 0$.

Ngoài ra, chúng ta có thể sử dụng phương sai để mở rộng mối quan hệ mà chúng ta đã thấy trước đây. Nhớ lại đó là X và Y là hai biến ngẫu nhiên độc lập sau đó

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{19.6.43}$$

Với kiến thức về đồng phương sai, chúng ta có thể mở rộng mối quan hệ này. Indeedthật vậy, some algebrađại số can showchỉ that in generalchung,

$$\mathrm{Var}(X + Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\mathrm{Cov}(X, Y). \tag{19.6.44}$$

Điều này cho phép chúng ta khái quát hóa quy tắc tổng hợp phương sai cho các biến ngẫu nhiên tương quan.

Tương quan

As we did in the case of means and variances, let us now consider units. If X is measured in one unit (say inches), and Y is measured in another (say dollars), the covariance is measured in the product of these two units inches $\times$ dollars. These units can be hard to interpret. What we will often want in this case is a unit-less measurement of relatedness. Indeed, often we do not care about exact quantitative correlation, but rather ask if the correlation is in the same direction, and how strong the relationship is.

To see what makes sense, let us perform a thought experiment. Suppose that we convert our random variables in inches and dollars to be in inches and cents. In this case the random variable Y is multiplied by 100. If we work through the definition, this means that $\text{Cov}(X, Y)$ will be multiplied by 100. Thus we see that in this case a change of units change the covariance by a factor of 100. Thus, to find our unit-invariant measure of correlation, we will need to divide by something else that also gets scaled by 100. Indeed we have a clear candidate, the standard deviation! Indeed if we define the *correlation coefficient* to be

we see that this is a unit-less value. A little mathematics can show that this number is between -1 and 1 with 1 meaning maximally positively correlated, whereas -1 means maximally negatively correlated.

Returning to our explicit discrete example above, we can see that $\sigma_X = 1$ and $\sigma_Y = 2$, so we can compute the correlation between the two random variables using `eq_cor_def` to see that

$$\rho(X, Y) = \frac{4p - 2}{1 \cdot 2} = 2p - 1. \tag{19.6.45}$$

This now ranges between -1 and 1 with the expected behavior of 1 meaning most correlated, and -1 meaning minimally correlated.

As another example, consider X as any random variable, and $Y = aX + b$ as any linear deterministic function of X. Then, one can compute that

$$\sigma_Y = \sigma_{aX+b} = |a|\sigma_X, \tag{19.6.46}$$

$$\text{Cov}(X, Y) = \text{Cov}(X, aX + b) = a\text{Cov}(X, X) = a\text{Var}(X), \tag{19.6.47}$$

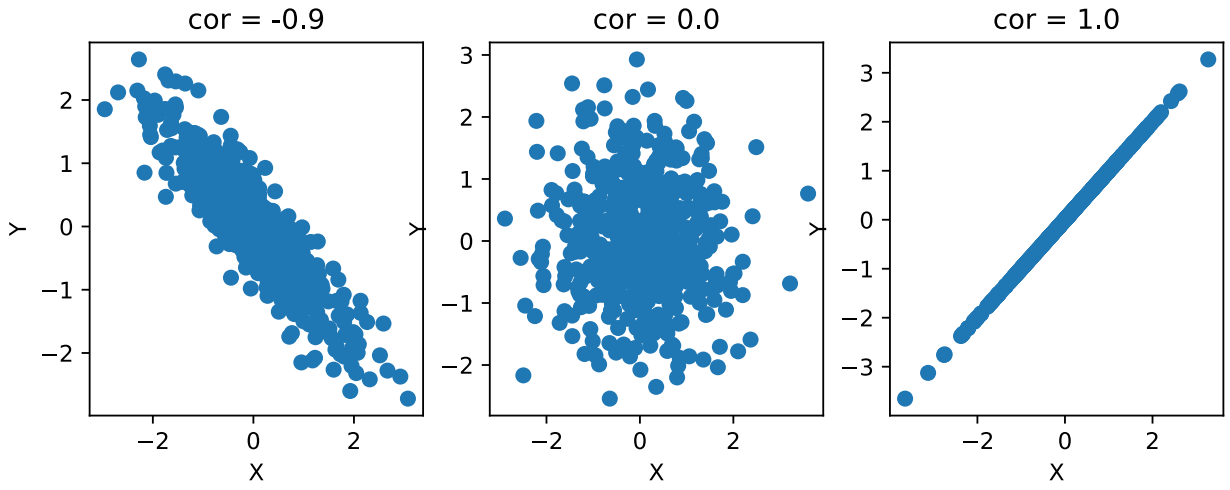
and thus by `eq_cor_def` that

$$\rho(X, Y) = \frac{a\text{Var}(X)}{|a|\sigma_X^2} = \frac{a}{|a|} = \text{sign}(a). \tag{19.6.48}$$

Thus we see that the correlation is $+1$ for any $a > 0$, and -1 for any $a < 0$ illustrating that correlation measures the degree and directionality the two random variables are related, not the scale that the variation takes.

Let us again plot a collection of random variables with tunable correlation.

```
# Plot a few random variables adjustable correlations
cors = [-0.9, 0.0, 1.0]
d2l.plt.figure(figsize=(12, 3))
for i in range(3):
    X = torch.randn(500)
    Y = cors[i] * X + torch.sqrt(torch.tensor(1) -
                                 cors[i]**2) * torch.randn(500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.numpy(), Y.numpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```



Let us list a few properties of the correlation below.

- For any random variable X, $\rho(X, X) = 1$.
- For any random variables X, Y and numbers a and b, $\rho(aX + b, Y) = \rho(X, aY + b) = \rho(X, Y)$.
- If X and Y are independent with non-zero variance then $\rho(X, Y) = 0$.

As a final note, you may feel like some of these formulae are familiar. Indeed, if we expand everything out assuming that $\mu_X = \mu_Y = 0$, we see that this is

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \tag{19.6.49}$$

This looks like a sum of a product of terms divided by the square root of sums of terms. This is exactly the formula for the cosine of the angle between two vectors $\mathbf{v}, \mathbf{w}$ with the different coordinates weighted by p_{ij}:

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2} \sqrt{\sum_i w_i^2}}. \tag{19.6.50}$$

Thật vậy, nếu chúng ta nghĩ về các định mức là liên quan đến độ lệch chuẩn, và tương quan như là cosin của các góc, phần lớn trực giác chúng ta có từ hình học có thể được áp dụng để suy nghĩ về các biến ngẫu nhiên.

19.6.2 Tóm lượng* Biến ngẫu nhiên liên tục là các biến ngẫu nhiên có thể thực hiện một liên tục của các giá trị. Chúng có một số khó khăn kỹ thuật khiến chúng trở nên khó khăn hơn khi làm việc so với các biến ngẫu nhiên rời rạc * Hàm mật độ xác suất cho phép chúng ta làm việc với các biến ngẫu nhiên liên tục bằng cách đưa ra một hàm trong đó khu vực dưới đường cong trên một khoảng thời gian cho xác suất tìm thấy một điểm mẫu trong khoảng thời gian đó* Hàm phân phối tích lũy là xác suất quan sát biến ngẫu nhiên nhỏ hơn một ngưỡng nhất định. Nó có thể cung cấp một quan điểm thay thế hữu ích thống nhất các biến rời rạc và liên tục.* trung bình là giá trị trung bình của một biến ngẫu nhiên. * phương sai là bình phương dự kiến của sự khác biệt giữa biến ngẫu nhiên và trung bình của nó.* Độ lệch chuẩn là căn bậc hai của phương sai. Nó có thể được coi là đo phạm vi các giá trị mà biến ngẫu nhiên có thể mất. * Bất đẳng thức của Chebyshev cho phép chúng ta làm cho trực giác này nghiêm ngặt bằng cách đưa ra một khoảng rõ ràng chứa biến ngẫu nhiên hầu hết thời gian* Mật độ chung cho phép chúng ta làm việc với các biến ngẫu nhiên tương quan. Chúng ta có thể lề mật độ chung bằng cách tích hợp các biến ngẫu nhiên không mong muốn để có được sự phân bố của biến ngẫu nhiên mong muốn* hệ số đồng phương sai và hệ số tương quan cung cấp một cách để đo bất kỳ mối quan hệ tuyến tính nào giữa hai biến ngẫu nhiên tương quan.

Exercises

1. Suppose that we have the random variable with density given by $p(x) = \frac{1}{x^2}$ for $x \geq 1$ and $p(x) = 0$ otherwise. What is $P(X > 2)$?
2. The Laplace distribution is a random variable whose density is given by $p(x = \frac{1}{2}e^{-|x|}$. What is the mean and the standard deviation of this function? As a hint, $\int_0^\infty xe^{-x}\ dx = 1$ and $\int_0^\infty x^2e^{-x}\ dx = 2$.
3. I walk up to you on the street and say "I have a random variable with mean 1, standard deviation 2, and I observed 25% of my samples taking a value larger than 9." Do you believe me? Why or why not?
4. Suppose that you have two random variables X, Y, with joint density given by $p_{XY}(x, y) = 4xy$ for $x, y \in [0, 1]$ and $p_{XY}(x, y) = 0$ otherwise. What is the covariance of X and Y?

Discussions[218]

19.7 Khả năng tối đa

Một trong những cách suy nghĩ phổ biến nhất trong học máy là quan điểm khả năng tối đa. Đây là khái niệm rằng khi làm việc với một mô hình xác suất với các tham số không xác định, các tham số làm cho dữ liệu có xác suất cao nhất là những thông số có khả năng nhất.

[218] https://discuss.d2l.ai/t/1094

19.7.1 Nguyên tắc khả năng tối đa

Điều này có một cách giải thích Bayesian có thể hữu ích để suy nghĩ về. Giả sử rằng chúng ta có một mô hình với các tham số $\boldsymbol{\theta}$ và một tập hợp các ví dụ dữ liệu X. Đối với sự cụ thể, chúng ta có thể tưởng tượng rằng $\boldsymbol{\theta}$ là một giá trị duy nhất đại diện cho xác suất một đồng xu xuất hiện đầu khi lật, và X là một chuỗi các lật đồng xu độc lập. Chúng tôi sẽ xem xét ví dụ này trong chiều sâu sau.

Nếu chúng ta muốn tìm giá trị có khả năng nhất cho các tham số của mô hình của chúng ta, điều đó có nghĩa là chúng ta muốn tìm

$$\operatorname{argmax} P(\boldsymbol{\theta} \mid X). \tag{19.7.1}$$

Theo quy tắc của Bayes, đây là điều tương tự như

$$\operatorname{argmax} \frac{P(X \mid \boldsymbol{\theta})P(\boldsymbol{\theta})}{P(X)}. \tag{19.7.2}$$

Biểu thức $P(X)$, một tham số xác suất bất khả tri của việc tạo dữ liệu, không phụ thuộc vào $\boldsymbol{\theta}$ ở tất cả, và do đó có thể được giảm mà không thay đổi sự lựa chọn tốt nhất của $\boldsymbol{\theta}$. Tương tự, bây giờ chúng ta có thể khẳng định rằng chúng tôi không có giả định trước về tập hợp các tham số nào tốt hơn bất kỳ tham số nào khác, vì vậy chúng tôi có thể tuyên bố rằng $P(\boldsymbol{\theta})$ cũng không phụ thuộc vào theta! Ví dụ, điều này có ý nghĩa trong ví dụ lật đồng xu của chúng tôi, nơi xác suất nó xuất hiện đầu có thể là bất kỳ giá trị nào trong $[0, 1]$ mà không có bất kỳ niềm tin nào trước đó là công bằng hay không (thường được gọi là ưu tiên * không thông tin*). Do đó, chúng tôi thấy rằng việc áp dụng quy tắc Bayes' cho thấy sự lựa chọn tốt nhất của chúng tôi về $\boldsymbol{\theta}$ là ước tính khả năng tối đa cho $\boldsymbol{\theta}$:

$$\hat{\boldsymbol{\theta}} = \operatorname*{argmax}_{\boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}). \tag{19.7.3}$$

Như một vấn đề của thuật ngữ phổ biến, xác suất của dữ liệu được đưa ra các tham số ($P(X \mid \boldsymbol{\theta})$) được gọi là *likelihood*.

Một ví dụ cụ thể

Hãy để chúng tôi xem làm thế nào điều này hoạt động trong một ví dụ cụ thể. Giả sử rằng chúng ta có một tham số duy nhất θ đại diện cho xác suất rằng một đồng xu lật là đầu. Sau đó, xác suất nhận được đuôi là $1 - \theta$, và vì vậy nếu dữ liệu quan sát của chúng tôi X là một chuỗi với n_H đầu và n_T đuôi, chúng ta có thể sử dụng thực tế là xác suất độc lập nhân lên để thấy rằng

$$P(X \mid \theta) = \theta^{n_H}(1-\theta)^{n_T}. \tag{19.7.4}$$

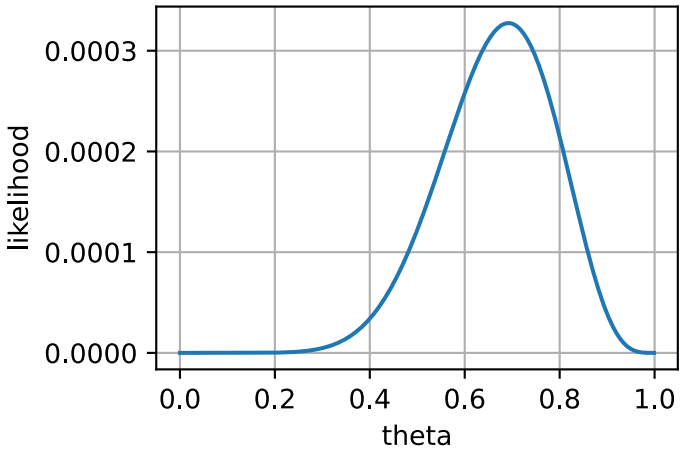
Nếu chúng ta lật 13 coin và nhận được chuỗi "HHHTHTTHHHHHT", có $n_H = 9$ và $n_T = 4$, chúng ta thấy rằng đây là

$$P(X \mid \theta) = \theta^9(1-\theta)^4. \tag{19.7.5}$$

Một điều hay về ví dụ này sẽ là chúng ta biết câu trả lời đang diễn ra. Thật vậy, nếu chúng tôi nói bằng lời nói, "Tôi lật 13 đồng xu, và 9 người đứng đầu, đoán tốt nhất của chúng tôi cho xác suất đồng xu đến chúng ta đứng đầu là gì? , "mọi người sẽ đoán chính xác $9/13$. Phương pháp khả năng tối đa này sẽ cung cấp cho chúng ta là một cách để có được con số đó từ các hiệu trưởng đầu tiên theo cách sẽ khái quát hóa đến các tình huống phức tạp hơn rất nhiều.

Đối với ví dụ của chúng tôi, cốt truyện của $P(X \mid \theta)$ như sau:

```
%matplotlib inline
import torch
from d2l import torch as d2l

theta = torch.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```



Điều này có giá trị tối đa của nó ở đâu đó gần $9/13 \approx 0.7\ldots$ dự kiến của chúng tôi. Để xem liệu nó có chính xác ở đó không, chúng ta có thể chuyển sang tính toán. Lưu ý rằng ở mức tối đa, gradient của hàm là phẳng. Do đó, chúng ta có thể tìm thấy ước tính khả năng tối đa (19.7.1) bằng cách tìm các giá trị của θ trong đó đạo hàm bằng 0 và tìm giá trị cho xác suất cao nhất. Chúng tôi tính toán:

$$\begin{aligned} 0 &= \frac{d}{d\theta} P(X \mid \theta) \\ &= \frac{d}{d\theta} \theta^9 (1-\theta)^4 \\ &= 9\theta^8 (1-\theta)^4 - 4\theta^9 (1-\theta)^3 \\ &= \theta^8 (1-\theta)^3 (9 - 13\theta). \end{aligned} \tag{19.7.6}$$

Điều này có ba giải pháp: 0, 1 và $9/13$. Hai đầu tiên rõ ràng là minima, không phải maxima như họ gán xác suất 0 cho trình tự của chúng tôi. Giá trị cuối cùng không * không* gán xác suất bằng 0 cho trình tự của chúng tôi, và do đó phải là ước tính khả năng tối đa $\hat{\theta} = 9/13$.

19.7.2 Tối ưu hóa số và tiêu cực Log-Likelihood

Ví dụ trước là tốt đẹp, nhưng nếu chúng ta có hàng tỷ tham số và ví dụ dữ liệu thì sao?

Đầu tiên, lưu ý rằng nếu chúng ta đưa ra giả định rằng tất cả các ví dụ dữ liệu đều độc lập, chúng ta không còn thực tế có thể xem xét khả năng vì nó là sản phẩm của nhiều xác suất. Thật vậy, mỗi xác suất là trong $[0, 1]$, nói thường có giá trị khoảng $1/2$, và sản phẩm của $(1/2)^{1000000000}$ thấp hơn nhiều so với độ chính xác của máy. Chúng tôi không thể làm việc trực tiếp với điều đó.

Tuy nhiên, hãy nhớ lại rằng logarit biến sản phẩm thành một khoản tiền, trong trường hợp đó

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6\ldots \tag{19.7.7}$$

Con số này phù hợp hoàn hảo ngay cả trong một độ chính xác duy nhất 32-bit float. Vì vậy, chúng ta nên xem xét *log-likelihood*, đó là

$$\log(P(X \mid \boldsymbol{\theta})). \tag{19.7.8}$$

Kể từ khi chức năng $x \mapsto \log(x)$ đang tăng lên, tối đa hóa khả năng là điều tương tự như tối đa hóa khả năng log. Thật vậy trong Section 19.8, chúng ta sẽ thấy lý do này được áp dụng khi làm việc với ví dụ cụ thể về phân loại Bayes ngây thơ.

Chúng tôi thường làm việc với các chức năng mất mát, nơi chúng tôi muốn giảm thiểu tổn thất. Chúng tôi có thể biến khả năng tối đa thành việc giảm thiểu tổn thất bằng cách lấy $-\log(P(X \mid \boldsymbol{\theta}))$, đó là *âm log-likelihood*.

Để minh họa điều này, hãy xem xét vấn đề lật đồng xu từ trước, và giả vờ rằng chúng ta không biết giải pháp hình thức đóng. We may Tháng Năm compute tính toán that

$$-\log(P(X \mid \boldsymbol{\theta})) = -\log(\theta^{n_H}(1-\theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1-\theta)). \tag{19.7.9}$$

Điều này có thể được viết thành mã, và tự do tối ưu hóa ngay cả đối với hàng tỷ flips đồng xu.

```
# Set up our data
n_H = 8675309
n_T = 25624

# Initialize our paramteres
theta = torch.tensor(0.5, requires_grad=True)

# Perform gradient descent
lr = 0.00000000001
for iter in range(10):
    loss = -(n_H * torch.log(theta) + n_T * torch.log(1 - theta))
    loss.backward()
    with torch.no_grad():
        theta -= lr * theta.grad
    theta.grad.zero_()

# Check output
theta, n_H / (n_H + n_T)
```

```
(tensor(0.5017, requires_grad=True), 0.9970550284664874)
```

Sự tiện lợi về số không phải là lý do duy nhất tại sao mọi người thích sử dụng tiêu cực log-likelihoods. Có một số lý do khác tại sao nó là thích hợp hơn.

Lý do thứ hai chúng tôi xem xét khả năng đăng nhập là việc áp dụng đơn giản hóa các quy tắc tính toán. Như đã thảo luận ở trên, do các giả định độc lập, hầu hết các xác suất chúng ta gặp phải trong học máy là sản phẩm của xác suất cá nhân.

$$P(X \mid \boldsymbol{\theta}) = p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \tag{19.7.10}$$

Điều này có nghĩa là nếu chúng ta trực tiếp áp dụng quy tắc sản phẩm để tính toán một đạo hàm, chúng ta nhận

được

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}} P(X \mid \boldsymbol{\theta}) = &\left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_2 \mid \boldsymbol{\theta})\right) \cdots P(x_n \mid \boldsymbol{\theta}) \\ &\vdots \\ &+ P(x_1 \mid \boldsymbol{\theta}) \cdot P(x_2 \mid \boldsymbol{\theta}) \cdots \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right).\end{aligned} \tag{19.7.11}$$

Điều này đòi hỏi $n(n-1)$ phép nhân, cùng với $(n-1)$ bổ sung, vì vậy nó tỷ lệ thuận với thời gian bậc hai trong các đầu vào! Đủ thông minh trong các thuật ngữ nhóm sẽ làm giảm điều này đến thời gian tuyến tính, nhưng nó đòi hỏi một số suy nghĩ. Đối với bản ghi âm khả năng chúng tôi có thay vào đó

$$-\log\left(P(X \mid \boldsymbol{\theta})\right) = -\log(P(x_1 \mid \boldsymbol{\theta})) - \log(P(x_2 \mid \boldsymbol{\theta})) \cdots - \log(P(x_n \mid \boldsymbol{\theta})), \tag{19.7.12}$$

mà sau đó cho

$$-\frac{\partial}{\partial \boldsymbol{\theta}} \log\left(P(X \mid \boldsymbol{\theta})\right) = \frac{1}{P(x_1 \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_1 \mid \boldsymbol{\theta})\right) + \cdots + \frac{1}{P(x_n \mid \boldsymbol{\theta})} \left(\frac{\partial}{\partial \boldsymbol{\theta}} P(x_n \mid \boldsymbol{\theta})\right). \tag{19.7.13}$$

Điều này chỉ đòi hỏi n phân chia và $n-1$ tổng, và do đó là thời gian tuyến tính trong các đầu vào.

Lý do thứ ba và cuối cùng để xem xét khả năng đăng nhập tiêu cực là mối quan hệ với lý thuyết thông tin, mà chúng ta sẽ thảo luận chi tiết trong Section 19.10. Đây là một lý thuyết toán học nghiêm ngặt mà đưa ra một cách để đo mức độ thông tin hoặc ngẫu nhiên trong một biến ngẫu nhiên. Đối tượng quan trọng của nghiên cứu trong lĩnh vực đó là entropy

$$H(p) = -\sum_i p_i \log_2(p_i), \tag{19.7.14}$$

which đo the randomnessngẫu nhiên of a sourcenguồn. Lưu ý rằng đây không có gì khác hơn là xác suất $-\log$ trung bình, và do đó nếu chúng ta lấy khả năng log âm của chúng tôi và chia cho số lượng các ví dụ dữ liệu, chúng tôi nhận được một người họ hàng của entropy được gọi là cross-entropy. Việc giải thích lý thuyết này một mình sẽ đủ hấp dẫn để thúc đẩy báo cáo khả năng nhật ký âm trung bình trên tập dữ liệu như một cách đo lường hiệu suất mô hình.

19.7.3 Khả năng tối đa cho các biến liên tục

Tất cả mọi thứ mà chúng ta đã làm cho đến nay giả định chúng ta đang làm việc với các biến ngẫu nhiên rời rạc, nhưng nếu chúng ta muốn làm việc với các biến liên tục thì sao?

Tóm tắt ngắn gọn là không có gì ở tất cả các thay đổi, ngoại trừ chúng tôi thay thế tất cả các trường hợp của xác suất với mật độ xác suất. Nhắc lại rằng chúng tôi viết mật độ với chữ thường p, điều này có nghĩa là ví dụ chúng tôi nói

$$-\log\left(p(X \mid \boldsymbol{\theta})\right) = -\log(p(x_1 \mid \boldsymbol{\theta})) - \log(p(x_2 \mid \boldsymbol{\theta})) \cdots - \log(p(x_n \mid \boldsymbol{\theta})) = -\sum_i \log(p(x_i \mid \theta)). \tag{19.7.15}$$

Câu hỏi trở thành, "Tại sao điều này ổn?" Rốt cuộc, lý do chúng tôi giới thiệu mật độ là do xác suất nhận được kết quả cụ thể là bằng không, và do đó không phải là xác suất tạo dữ liệu của chúng tôi cho bất kỳ tập hợp các tham số không?

Thật vậy, đây là trường hợp, và hiểu lý do tại sao chúng ta có thể chuyển sang mật độ là một bài tập trong việc truy tìm những gì xảy ra với epsilon.

Trước tiên chúng ta hãy xác định lại mục tiêu của chúng tôi. Giả sử rằng đối với các biến ngẫu nhiên liên tục, chúng ta không còn muốn tính toán xác suất nhận được chính xác giá trị phù hợp, mà thay vào đó phù hợp với trong một số phạm vi ϵ. Để đơn giản, chúng tôi giả định dữ liệu của chúng tôi được lặp đi lặp lại quan sát $x_1, \ldots, x_N$ của các biến ngẫu nhiên phân phối giống hệt nhau $X_1, \ldots, X_N$. Như chúng ta đã thấy trước đây, điều này có thể được viết là

$$\begin{aligned} &P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta}) \\ \approx &\epsilon^N p(x_1 \mid \boldsymbol{\theta}) \cdot p(x_2 \mid \boldsymbol{\theta}) \cdots p(x_n \mid \boldsymbol{\theta}). \end{aligned} \tag{19.7.16}$$

Do đó, nếu chúng ta lấy logarit âm của điều này, chúng ta có được

$$\begin{aligned} &-\log(P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \ldots, X_N \in [x_N, x_N + \epsilon] \mid \boldsymbol{\theta})) \\ \approx &-N \log(\epsilon) - \sum_i \log(p(x_i \mid \boldsymbol{\theta})). \end{aligned} \tag{19.7.17}$$

Nếu chúng ta kiểm tra biểu thức này, nơi duy nhất mà ϵ xảy ra là trong hằng số phụ gia $-N \log(\epsilon)$. Điều này hoàn toàn không phụ thuộc vào các thông số $\boldsymbol{\theta}$, vì vậy sự lựa chọn tối ưu của $\boldsymbol{\theta}$ không phụ thuộc vào sự lựa chọn của chúng tôi là ϵ! Nếu chúng tôi yêu cầu bốn chữ số hoặc bốn trăm, sự lựa chọn tốt nhất của $\boldsymbol{\theta}$ vẫn giữ nguyên, do đó chúng tôi có thể tự do thả epsilon để thấy rằng những gì chúng tôi muốn tối ưu hóa là

$$-\sum_i \log(p(x_i \mid \boldsymbol{\theta})). \tag{19.7.18}$$

Do đó, chúng ta thấy rằng quan điểm khả năng tối đa có thể hoạt động với các biến ngẫu nhiên liên tục dễ dàng như với các biến rời rạc bằng cách thay thế xác suất bằng mật độ xác suất.

19.7.4 Tóm tắt * Nguyên tắc khả năng tối đa cho chúng ta biết rằng mô hình phù hợp nhất cho một tập dữ liệu nhất định là mô hình tạo ra dữ liệu có xác suất cao nhất.* Thường mọi người làm việc với khả năng ghi âm thay vì nhiều lý do: ổn định số, chuyển đổi sản phẩm thành tổng (và kết quả đơn giản hóa tính toán gradient), và mối quan hệ lý thuyết với lý thuyết thông tin * Mặc dù đơn giản nhất để thúc đẩy trong cài đặt rời rạc, nó có thể được tự do khái quát hóa để cài đặt liên tục cũng như bằng cách tối đa hóa mật độ xác suất được gán cho các datapoints.

19.7.5 Bài tập 1. Giả sử rằng bạn biết rằng một biến ngẫu nhiên có mật độ $\frac{1}{\alpha} e^{-\alpha x}$ cho một số giá trị α. Bạn có được một quan sát duy nhất từ biến ngẫu nhiên đó là số 3. Ước tính khả năng tối đa cho α là gì? 2. Giả sử rằng bạn có một tập dữ liệu của các mẫu $\{x_i\}_{i=1}^N$ rút ra từ một Gaussian với trung bình không xác định, nhưng phương sai 1. Ước tính khả năng tối đa cho nghĩa là gì?

Discussions[219]

[219] https://discuss.d2l.ai/t/1096

19.8 Bayes ngây thơ

Trong suốt các phần trước, chúng ta đã học về lý thuyết xác suất và các biến ngẫu nhiên. Để đưa lý thuyết này vào hoạt động, chúng ta hãy giới thiệu bộ phân loại Bayes* ngây thơ. Điều này không sử dụng gì ngoài các nguyên tắc cơ bản xác suất để cho phép chúng tôi thực hiện phân loại các chữ số.

Học tập là tất cả về việc đưa ra các giả định. Nếu chúng ta muốn phân loại một ví dụ dữ liệu mới mà chúng ta chưa bao giờ thấy trước đây chúng ta phải đưa ra một số giả định về ví dụ dữ liệu nào tương tự nhau. Phân loại Bayes ngây thơ, một thuật toán phổ biến và rõ ràng đáng kể, giả định tất cả các tính năng đều độc lập với nhau để đơn giản hóa việc tính toán. Trong phần này, chúng tôi sẽ áp dụng mô hình này để nhận dạng các ký tự trong hình ảnh.

```
%matplotlib inline
import math
import torch
import torchvision
from d2l import torch as d2l

d2l.use_svg_display()
```

19.8.1 Nhận dạng ký tự quang

MNIST (LeCun et al., 1998) là một trong những bộ dữ liệu được sử dụng rộng rãi. Nó chứa 60.000 hình ảnh để đào tạo và 10.000 hình ảnh để xác nhận. Mỗi hình ảnh chứa một chữ số viết tay từ 0 đến 9. Nhiệm vụ là phân loại từng hình ảnh thành chữ số tương ứng.

Gluon cung cấp lớp `MNIST` trong mô-đun `data.vision` để tự động lấy tập dữ liệu từ Internet. Sau đó, Gluon sẽ sử dụng bản sao cục bộ đã tải xuống. Chúng tôi chỉ định xem chúng tôi đang yêu cầu bộ đào tạo hay bộ thử nghiệm bằng cách đặt giá trị của tham số `train` thành `True` hoặc `False` tương ứng. Mỗi ảnh là một hình ảnh thang màu xám với cả chiều rộng và chiều cao 28 với hình dạng (28,28,1). Chúng tôi sử dụng một chuyển đổi tùy chỉnh để loại bỏ kích thước kênh cuối cùng. Ngoài ra, tập dữ liệu đại diện cho mỗi pixel bằng một số nguyên 8-bit không dấu. Chúng tôi định lượng chúng thành các tính năng nhị phân để đơn giản hóa vấn đề.

```
data_transform = torchvision.transforms.Compose([
    torchvision.transforms.ToTensor(),
    lambda x: torch.floor(x * 255 / 128).squeeze(dim=0)
])

mnist_train = torchvision.datasets.MNIST(
    root='./temp', train=True, transform=data_transform, download=True)
mnist_test = torchvision.datasets.MNIST(
    root='./temp', train=False, transform=data_transform, download=True)
```

```
0.6%Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./
↪temp/MNIST/raw/train-images-idx3-ubyte.gz
100.0%
Extracting ./temp/MNIST/raw/train-images-idx3-ubyte.gz to ./temp/MNIST/raw
102.8%
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
```

(continues on next page)

(continued from previous page)

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./
↪temp/MNIST/raw/train-labels-idx1-ubyte.gz
Extracting ./temp/MNIST/raw/train-labels-idx1-ubyte.gz to ./temp/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

8.0%Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to␣
↪./temp/MNIST/raw/t10k-images-idx3-ubyte.gz
100.0%
Extracting ./temp/MNIST/raw/t10k-images-idx3-ubyte.gz to ./temp/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./
↪temp/MNIST/raw/t10k-labels-idx1-ubyte.gz
112.7%Extracting ./temp/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./temp/MNIST/
↪raw
```

Chúng ta có thể truy cập một ví dụ cụ thể, chứa hình ảnh và nhãn tương ứng.

```
image, label = mnist_train[2]
image.shape, label
```

```
(torch.Size([28, 28]), 4)
```

Ví dụ của chúng tôi, được lưu trữ ở đây trong biến `image`, tương ứng với một hình ảnh có chiều cao và chiều rộng 28 pixel.

```
image.shape, image.dtype
```

```
(torch.Size([28, 28]), torch.float32)
```

Mã của chúng tôi lưu trữ nhãn của mỗi hình ảnh dưới dạng vô hướng. Loại của nó là một số nguyên 32-bit.

```
label, type(label)
```

```
(4, int)
```

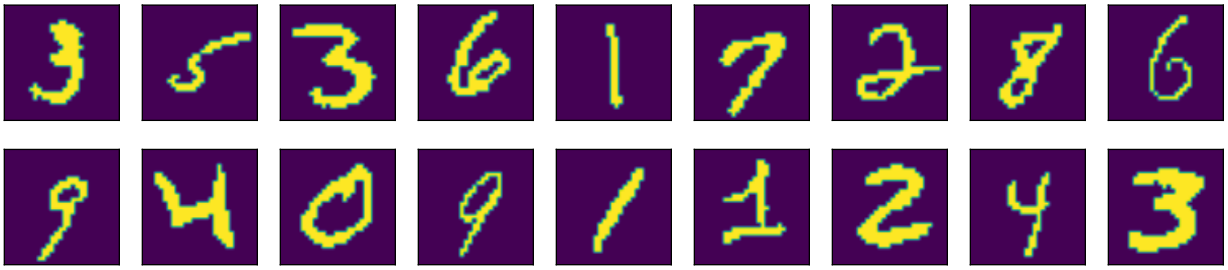
Chúng tôi cũng có thể truy cập nhiều ví dụ cùng một lúc.

```
images = torch.stack([mnist_train[i][0] for i in range(10, 38)], dim=0)
labels = torch.tensor([mnist_train[i][1] for i in range(10, 38)])
images.shape, labels.shape
```

```
(torch.Size([28, 28, 28]), torch.Size([28]))
```

Hãy để chúng tôi hình dung những ví dụ này.

```
d2l.show_images(images, 2, 9);
```



19.8.2 Mô hình xác suất để phân loại

Trong một nhiệm vụ phân loại, chúng tôi ánh xạ một ví dụ vào một danh mục. Ở đây một ví dụ là một hình ảnh 28×28 xám và một danh mục là một chữ số. (Tham khảo Section 4.4 để được giải thích chi tiết hơn.) Một cách tự nhiên để thể hiện nhiệm vụ phân loại là thông qua câu hỏi xác suất: nhãn có khả năng nhất được đưa ra các tính năng (tức là pixel hình ảnh) là gì? Biểu thị bởi $\mathbf{x} \in \mathbb{R}^d$ các tính năng của ví dụ và $y \in \mathbb{R}$ nhãn. Ở đây các tính năng là pixel hình ảnh, nơi chúng ta có thể định hình lại hình ảnh 2 chiều thành một vectơ sao cho $d = 28^2 = 784$ và nhãn là chữ số. Xác suất của nhãn cho các tính năng là $p(y \mid \mathbf{x})$. Nếu chúng ta có thể tính toán các xác suất này, đó là $p(y \mid \mathbf{x})$ cho $y = 0, \ldots, 9$ trong ví dụ của chúng tôi, thì phân loại sẽ đưa ra dự đoán $\hat{y}$ được đưa ra bởi biểu thức:

$$\hat{y} = \text{argmax}\, p(y \mid \mathbf{x}). \tag{19.8.1}$$

Thật không may, điều này đòi hỏi chúng tôi ước tính $p(y \mid \mathbf{x})$ cho mọi giá trị của $\mathbf{x} = x_1, ..., x_d$. Hãy tưởng tượng rằng mỗi tính năng có thể lấy một trong 2 giá trị. Ví dụ: tính năng $x_1 = 1$ có thể biểu thị rằng từ apple xuất hiện trong một tài liệu nhất định và $x_1 = 0$ sẽ biểu thị rằng nó không. Nếu chúng ta có 30 các tính năng nhị phân như vậy, điều đó có nghĩa là chúng ta cần phải chuẩn bị để phân loại bất kỳ 2^{30} nào (hơn 1 tỷ!) giá trị có thể của vector đầu vào $\mathbf{x}$.

Hơn nữa, việc học ở đâu? Nếu chúng ta cần xem mọi ví dụ có thể để dự đoán nhãn tương ứng thì chúng ta không thực sự học một mẫu mà chỉ ghi nhớ tập dữ liệu.

19.8.3 Phân loại Bayes ngây thơ

May mắn thay, bằng cách đưa ra một số giả định về tính độc lập có điều kiện, chúng ta có thể giới thiệu một số thiên vị quy nạp và xây dựng một mô hình có khả năng khái quát hóa từ một lựa chọn tương đối khiêm tốn các ví dụ đào tạo. Để bắt đầu, chúng ta hãy sử dụng định lý Bayes, để thể hiện phân loại như

$$\hat{y} = \text{argmax}_y\, p(y \mid \mathbf{x}) = \text{argmax}_y\, \frac{p(\mathbf{x} \mid y)p(y)}{p(\mathbf{x})}. \tag{19.8.2}$$

Lưu ý rằng mẫu số là thuật ngữ bình thường hóa $p(\mathbf{x})$ mà không phụ thuộc vào giá trị của nhãn y. Kết quả là, chúng ta chỉ cần lo lắng về việc so sánh tử số trên các giá trị khác nhau của y. Ngay cả khi tính toán mẫu số hóa ra là khó chữa, chúng ta có thể thoát khỏi việc bỏ qua nó, miễn là chúng ta có thể đánh giá tử số. May mắn thay, ngay cả khi chúng ta muốn phục hồi hằng số bình thường hóa, chúng ta có thể. Chúng tôi luôn có thể phục hồi thời hạn bình thường hóa kể từ $\sum_y p(y \mid \mathbf{x}) = 1$.

Bây giờ, chúng ta hãy tập trung vào $p(\mathbf{x} \mid y)$. Sử dụng quy tắc chuỗi xác suất, chúng ta có thể thể hiện thuật ngữ $p(\mathbf{x} \mid y)$ như

$$p(x_1 \mid y) \cdot p(x_2 \mid x_1, y) \cdot ... \cdot p(x_d \mid x_1, ..., x_{d-1}, y). \tag{19.8.3}$$

Bản thân nó, biểu thức này không giúp chúng ta xa hơn. Chúng ta vẫn phải ước tính khoảng 2^d tham số. Tuy nhiên, nếu chúng ta giả định rằng * các tính năng độc lập có điều kiện với nhau, với nhãn*, thì đột nhiên chúng ta có hình dạng tốt hơn nhiều, vì thuật ngữ này đơn giản hóa thành $\prod_i p(x_i \mid y)$, cho chúng ta bộ dự đoán

$$\hat{y} = \text{argmax}_y \prod_{i=1}^{d} p(x_i \mid y)p(y). \tag{19.8.4}$$

Nếu chúng ta có thể ước tính $p(x_i = 1 \mid y)$ cho mỗi i và y và lưu giá trị của nó trong $P_{xy}[i, y]$, ở đây P_{xy} là ma trận $d \times n$ với n là số lớp và $y \in \{1, \ldots, n\}$, thì chúng ta cũng có thể sử dụng điều này để ước tính $p(x_i = 0 \mid y)$, tức là,

$$p(x_i = t_i \mid y) = \begin{cases} P_{xy}[i, y] & \text{for } t_i = 1; \\ 1 - P_{xy}[i, y] & \text{for } t_i = 0. \end{cases} \tag{19.8.5}$$

Ngoài ra, chúng tôi ước tính $p(y)$ cho mỗi y và lưu nó trong $P_y[y]$, với P_y một vector chiều dài n. Then, for any newMới example thí dụ $\mathbf{t} = (t_1, t_2, \ldots, t_d)$, we could computetính toán

$$\begin{aligned} \hat{y} &= \text{argmax}_y \, p(y) \prod_{i=1}^{d} p(x_t = t_i \mid y) \\ &= \text{argmax}_y \, P_y[y] \prod_{i=1}^{d} P_{xy}[i, y]^{t_i} \, (1 - P_{xy}[i, y])^{1-t_i} \end{aligned} \tag{19.8.6}$$

for any y. Vì vậy, giả định của chúng tôi về sự độc lập có điều kiện đã lấy sự phức tạp của mô hình của chúng tôi từ sự phụ thuộc theo cấp số nhân vào số tính năng $\mathcal{O}(2^d n)$ đến một sự phụ thuộc tuyến tính, đó là $\mathcal{O}(dn)$.

19.8.4 Đào tạo

Vấn đề bây giờ là chúng ta không biết P_{xy} và P_y. Vì vậy, chúng ta cần phải ước tính giá trị của họ cho một số dữ liệu đào tạo đầu tiên. Đây là * đào tạo* mô hình. Ước tính P_y không quá khó. Vì chúng ta chỉ xử lý các lớp 10, chúng ta có thể đếm số lần xuất hiện n_y cho mỗi chữ số và chia nó cho tổng lượng dữ liệu n. Ví dụ: nếu chữ số 8 xảy ra $n_8 = 5,800$ lần và chúng ta có tổng cộng $n = 60,000$ hình ảnh, ước tính xác suất là $p(y = 8) = 0.0967$.

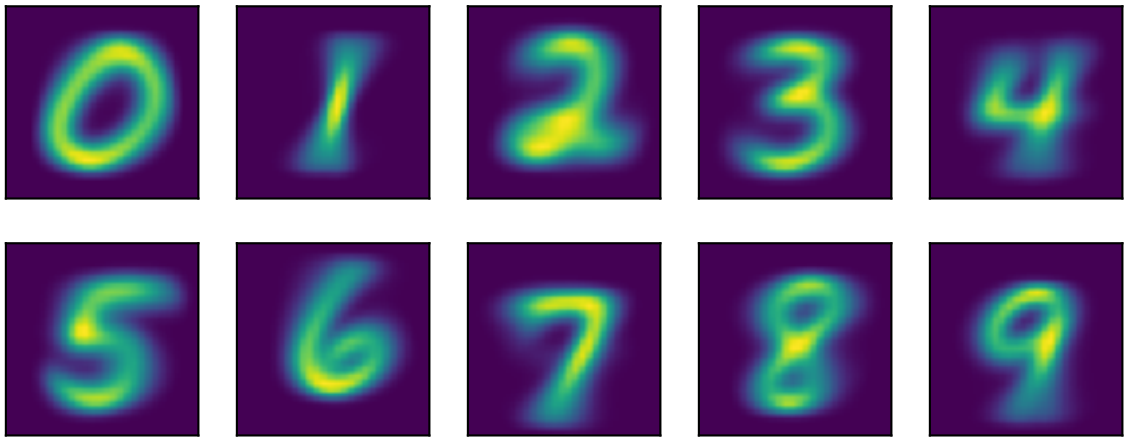
```
X = torch.stack([mnist_train[i][0] for i in range(len(mnist_train))], dim=0)
Y = torch.tensor([mnist_train[i][1] for i in range(len(mnist_train))])

n_y = torch.zeros(10)
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

```
tensor([0.0987, 0.1124, 0.0993, 0.1022, 0.0974, 0.0904, 0.0986, 0.1044, 0.
↪0975,
        0.0992])
```

Bây giờ để điều hơi khó khăn hơn P_{xy}. Vì chúng tôi chọn hình ảnh đen trắng, $p(x_i \mid y)$ biểu thị xác suất pixel i được bật cho lớp y. Giống như trước khi chúng ta có thể đi và đếm số lần n_{iy} sao cho một sự kiện xảy ra và chia nó cho tổng số lần xuất hiện y, tức là n_y. Nhưng có một cái gì đó hơi rắc rối: một số pixel nhất định có thể không bao giờ có màu đen (ví dụ: đối với hình ảnh được cắt tốt, các pixel góc có thể luôn có màu trắng).

Một cách thuận tiện để các nhà thống kê giải quyết vấn đề này là thêm số giả cho tất cả các lần xuất hiện. Do đó, thay vì n_{iy} chúng tôi sử dụng $n_{iy} + 1$ và thay vì n_y chúng tôi sử dụng $n_y + 1$. Điều này còn được gọi là *Laplace Smoothing*. Nó có vẻ đặc biệt, tuy nhiên nó có thể được thúc đẩy tốt từ một quan điểm Bayesian.

```
n_x = torch.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = torch.tensor(X.numpy()[Y.numpy() == y].sum(axis=0))
P_xy = (n_x + 1) / (n_y + 1).reshape(10, 1, 1)

d2l.show_images(P_xy, 2, 5);
```



Bằng cách hình dung các xác suất $10 \times 28 \times 28$ này (đối với mỗi pixel cho mỗi lớp), chúng ta có thể nhận được một số chữ số có vẻ trung bình.

Bây giờ chúng ta có thể sử dụng (19.8.6) để dự đoán một hình ảnh mới. Cho $\mathbf{x}$, các chức năng sau tính $p(\mathbf{x} \mid y)p(y)$ cho mỗi y.

```
def bayes_pred(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(dim=1)  # p(x|y)
    return p_xy * P_y

image, label = mnist_test[0]
bayes_pred(image)
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

Điều này đã đi sai khủng khiếp! Để tìm hiểu lý do tại sao, chúng ta hãy nhìn vào xác suất trên mỗi pixel. Chúng thường là những con số giữa 0.001 và 1. Chúng tôi đang nhân 784 trong số họ. Tại thời điểm này, điều đáng nói là chúng ta đang tính toán những con số này trên máy tính, do đó với một phạm vi cố định cho số mũ. Điều gì xảy ra là chúng ta trải nghiệm * underflow số*, tức là, nhân tất cả các số nhỏ dẫn đến một cái gì đó thậm chí còn nhỏ hơn cho đến khi nó được làm tròn xuống 0. Chúng tôi đã thảo luận về điều này như một vấn đề lý thuyết trong Section 19.7, nhưng chúng ta thấy rõ các hiện tượng ở đây trong thực tế.

Như đã thảo luận trong phần đó, chúng tôi khắc phục điều này bằng cách sử dụng thực tế là $\log ab = \log a + \log b$, tức là, chúng tôi chuyển sang tổng hợp logarit. Ngay cả khi cả a và b đều là những con số nhỏ, các giá trị logarit phải nằm trong một phạm vi thích hợp.

```
a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))
```

```
underflow: 0.0
logarithm is normal: -1805.2267129073316
```

Vì logarit là một hàm ngày càng tăng, chúng ta có thể viết lại (19.8.6) như

$$\hat{y} = \text{argmax}_y \ \log P_y[y] + \sum_{i=1}^{d} \Big[t_i \log P_{xy}[x_i, y] + (1 - t_i) \log(1 - P_{xy}[x_i, y])\Big]. \tag{19.8.7}$$

Chúng ta có thể thực hiện phiên bản ổn định sau:

```
log_P_xy = torch.log(P_xy)
log_P_xy_neg = torch.log(1 - P_xy)
log_P_y = torch.log(P_y)

def bayes_pred_stable(x):
    x = x.unsqueeze(0)  # (28, 28) -> (1, 28, 28)
    p_xy = log_P_xy * x + log_P_xy_neg * (1 - x)
    p_xy = p_xy.reshape(10, -1).sum(axis=1)  # p(x|y)
    return p_xy + log_P_y

py = bayes_pred_stable(image)
py
```

```
tensor([-269.0042, -301.7345, -245.2146, -218.8941, -193.4691, -206.1031,
        -292.5432, -114.6283, -220.3562, -163.1888])
```

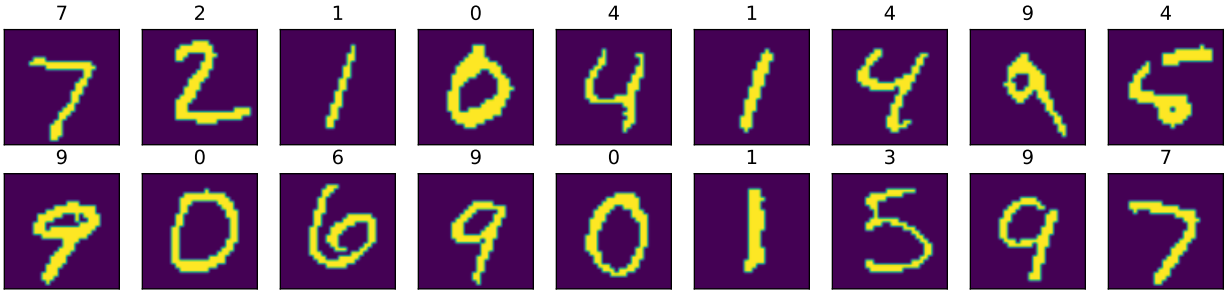
Bây giờ chúng ta có thể kiểm tra xem dự đoán có đúng không.

```
py.argmax(dim=0) == label
```

```
tensor(True)
```

Nếu bây giờ chúng ta dự đoán một vài ví dụ xác nhận, chúng ta có thể thấy bộ phân loại Bayes hoạt động khá tốt.

```
def predict(X):
    return [bayes_pred_stable(x).argmax(dim=0).type(torch.int32).item()
            for x in X]

X = torch.stack([mnist_test[i][0] for i in range(18)], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(18)])
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);
```



Cuối cùng, chúng ta hãy tính toán độ chính xác tổng thể của bộ phân loại.

```
X = torch.stack([mnist_test[i][0] for i in range(len(mnist_test))], dim=0)
y = torch.tensor([mnist_test[i][1] for i in range(len(mnist_test))])
preds = torch.tensor(predict(X), dtype=torch.int32)
float((preds == y).sum()) / len(y)  # Validation accuracy
```

```
0.8426
```

Các mạng sâu hiện đại đạt được tỷ lệ lỗi dưới 0.01. Hiệu suất tương đối kém là do các giả định thống kê không chính xác mà chúng tôi đã thực hiện trong mô hình của mình: chúng tôi giả định rằng mỗi điểm ảnh được tạo ra * độc lập*, chỉ tùy thuộc vào nhãn. Đây rõ ràng không phải là cách con người viết chữ số, và giả định sai lầm này dẫn đến sự sụp đổ của phân loại quá ngây thơ (Bayes) của chúng ta.

19.8.5 Tóm tắt * Sử dụng quy tắc của Bayes', một phân loại có thể được thực hiện bằng cách giả sử tất cả các tính năng quan sát là độc lập. * Phân loại này có thể được đào tạo trên một tập dữ liệu bằng cách đếm số lần xuất hiện của các kết hợp của nhãn và giá trị pixel * Phân loại này là tiêu chuẩn vàng trong nhiều thập kỷ cho các tác vụ như thư rác phát hiện.

19.8.6 Bài tập 1. Xem xét tập dữ liệu $[[0, 0], [0, 1], [1, 0], [1, 1]]$ với các nhãn được đưa ra bởi XOR của hai phần tử $[0, 1, 1, 0]$. Xác suất cho một phân loại ngây thơ Bayes được xây dựng trên bộ dữ liệu này là gì. Nó có phân loại thành công điểm của chúng tôi không? Nếu không, những giả định nào bị vi phạm? 1. Giả sử rằng chúng tôi đã không sử dụng Laplace làm mịn khi ước tính xác suất và một ví dụ dữ liệu đến lúc thử nghiệm trong đó có một giá trị không bao giờ quan sát thấy trong đào tạo. Sản lượng mô hình sẽ là gì? 1. Phân loại Bayes ngây thơ là một ví dụ cụ thể của một mạng Bayesian, trong đó sự phụ thuộc của các biến ngẫu nhiên được mã hóa bằng cấu trúc đồ thị. Trong khi lý thuyết đầy đủ nằm ngoài phạm vi của phần này (xem [Koller.Friedman.2009] để biết chi tiết đầy đủ), giải thích tại sao cho phép phụ thuộc rõ ràng giữa hai biến đầu vào trong mô hình XOR cho phép tạo ra một phân loại thành công.

Discussions[220]

[220] https://discuss.d2l.ai/t/1100

19.9 Thống kê

Không còn nghi ngờ gì nữa, để trở thành một học viên học sâu hàng đầu, khả năng đào tạo các mô hình hiện đại và chính xác cao là rất quan trọng. Tuy nhiên, người ta thường không rõ khi nào cải tiến là đáng kể, hoặc chỉ là kết quả của những biến động ngẫu nhiên trong quá trình đào tạo. Để có thể thảo luận về sự không chắc chắn về các giá trị ước tính, chúng ta phải tìm hiểu một số thống kê.

Tài liệu tham khảo sớm nhất của *thống kê * có thể được bắt nguồn từ một học giả Ả Rập Al-Kindi trong thế kỷ 9^{th}, người đã đưa ra một mô tả chi tiết về cách sử dụng thống kê và phân tích tần số để giải mã các tin nhắn được mã hóa. Sau 800 năm, số liệu thống kê hiện đại phát sinh từ Đức vào những năm 1700, khi các nhà nghiên cứu tập trung vào việc thu thập và phân tích dữ liệu nhân khẩu học và kinh tế. Ngày nay, thống kê là môn khoa học liên quan đến việc thu thập, xử lý, phân tích, giải thích và trực quan hóa dữ liệu. Hơn nữa, lý thuyết cốt lõi của thống kê đã được sử dụng rộng rãi trong nghiên cứu trong học viện, công nghiệp và chính phủ.

Cụ thể hơn, số liệu thống kê có thể được chia thành *thống kê mô tả* và * suy luận thống ký. *Tập trung trước đây vào việc tóm tắt và minh họa các tính năng của một bộ sưu tập dữ liệu quan sát được, được gọi là mẫu **. Mẫu được rút ra từ một * dân số*, biểu thị tổng số các cá nhân, vật phẩm hoặc sự kiện tương tự của lợi ích thí nghiệm của chúng tôi. Trái ngược với thống kê mô tả, *suy luận thống ký* suy luận thêm các đặc điểm của một quần thể từ các mẫu* đã cho, dựa trên các giả định rằng phân phối mẫu có thể nhân rộng phân bố dân số ở một mức độ nào đó.

Bạn có thể tự hỏi: "Sự khác biệt thiết yếu giữa học máy và thống kê là gì?" Nói về cơ bản, thống kê tập trung vào bài toán suy luận. Loại bài toán này bao gồm mô hình hóa mối quan hệ giữa các biến, chẳng hạn như suy luận nhân quả, và kiểm tra ý nghĩa thống kê của các tham số mô hình, chẳng hạn như thử nghiệm A/B. Ngược lại, machine learning nhấn mạnh vào việc đưa ra các dự đoán chính xác, mà không cần lập trình và hiểu rõ chức năng của từng tham số.

Trong phần này, chúng tôi sẽ giới thiệu ba loại phương pháp suy luận thống kê: đánh giá và so sánh các ước lượng, tiến hành các bài kiểm tra giả thuyết và xây dựng khoảng thời gian tin cậy. Những phương pháp này có thể giúp chúng ta suy ra các đặc điểm của một quần thể nhất định, tức là tham số thực sự θ. Đối với ngắn gọn, chúng ta giả định rằng tham số thật θ của một quần thể nhất định là một giá trị vô hướng. Thật đơn giản để mở rộng đến trường hợp θ là một vectơ hoặc tensor, do đó chúng tôi bỏ qua nó trong cuộc thảo luận của chúng tôi.

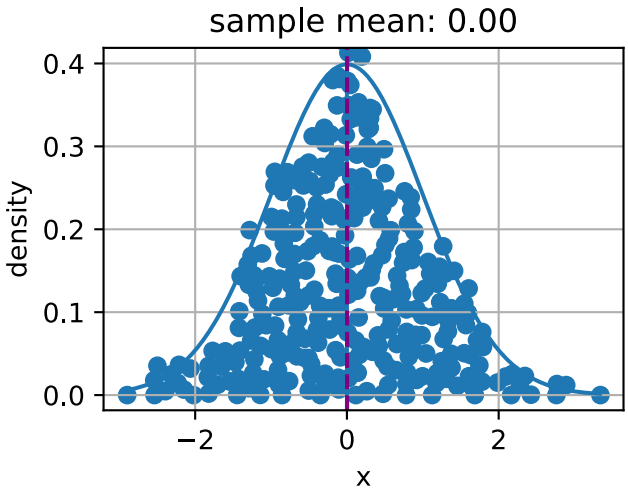
19.9.1 Đánh giá và so sánh ước tính

Trong thống kê, một *estimator* là một hàm của các mẫu đã cho được sử dụng để ước tính tham số đúng θ. Chúng tôi sẽ viết $\hat{\theta}_n = \hat{f}(x_1, \ldots, x_n)$ cho ước tính θ sau khi quan sát các mẫu $\{x_1, x_2, \ldots, x_n\}$.

Chúng ta đã thấy các ví dụ đơn giản về các chứng thực trước đây trong phần Section 19.7. Nếu bạn có một số mẫu từ một biến ngẫu nhiên Bernoulli, thì ước tính khả năng tối đa cho xác suất biến ngẫu nhiên là một có thể thu được bằng cách đếm số cái quan sát và chia cho tổng số mẫu. Tương tự, một bài tập yêu cầu bạn chỉ ra rằng ước tính khả năng tối đa của trung bình của một Gaussian được đưa ra một số mẫu được đưa ra bởi giá trị trung bình của tất cả các mẫu. Những ước lượng này hầu như sẽ không bao giờ cung cấp giá trị thực sự của tham số, nhưng lý tưởng cho một số lượng lớn các mẫu ước tính sẽ gần gũi.

Ví dụ, chúng ta hiển thị bên dưới mật độ thực của một biến ngẫu nhiên Gaussian với trung bình 0 và phương sai một, cùng với một mẫu thu thập từ Gaussian đó. Chúng tôi xây dựng tọa độ y để mọi điểm đều có thể nhìn thấy và mối quan hệ với mật độ ban đầu rõ ràng hơn.

```
import torch
from d2l import torch as d2l

torch.pi = torch.acos(torch.zeros(1)) * 2  #define pi in torch

# Sample datapoints and create y coordinate
epsilon = 0.1
torch.manual_seed(8675309)
xs = torch.randn(size=(300,))

ys = torch.tensor(
    [torch.sum(torch.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2))\
               / torch.sqrt(2*torch.pi*epsilon**2)) / len(xs)\
     for i in range(len(xs))])

# Compute true density
xd = torch.arange(torch.min(xs), torch.max(xs), 0.01)
yd = torch.exp(-xd**2/2) / torch.sqrt(2 * torch.pi)

# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=torch.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(torch.mean(xs).item()):.2f}')
d2l.plt.show()
```



Có thể có nhiều cách để tính toán một ước tính của một tham số $\hat{\theta}_n$. Trong phần này, chúng tôi giới thiệu ba phương pháp phổ biến để đánh giá và so sánh các ước lượng: lỗi bình phương trung bình, độ lệch chuẩn và thiên vị thống kê.

Lỗi bình phương trung bình

Có lẽ số liệu đơn giản nhất được sử dụng để đánh giá các ước lượng là lỗi bình phương * trung bình (MSE) * (hoặc mất l_2) của một ước tính có thể được định nghĩa là

$$\text{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \tag{19.9.1}$$

This allows us to quantify the average squared deviation from the true value. MSE is always non-negative. If you have read Section 4.1, you will recognize it as the most commonly used regression loss function. As a measure to evaluate an estimator, the closer its value to zero, the closer the estimator is close to the true parameter θ.

Thiên vị thống kê

MSE cung cấp một số liệu tự nhiên, nhưng chúng ta có thể dễ dàng tưởng tượng nhiều hiện tượng khác nhau có thể làm cho nó lớn. Hai cơ bản quan trọng là biến động trong ước tính do tính ngẫu nhiên trong tập dữ liệu, và lỗi có hệ thống trong ước tính do quy trình ước tính.

Đầu tiên, chúng ta hãy đo lỗi có hệ thống. Đối với một ước tính $\hat{\theta}_n$, minh họa toán học của *thiên vị thống ký* có thể được định nghĩa là

$$\text{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \tag{19.9.2}$$

Lưu ý rằng khi $\text{bias}(\hat{\theta}_n) = 0$, kỳ vọng của ước tính $\hat{\theta}_n$ bằng giá trị thực của tham số. Trong trường hợp này, chúng tôi nói $\hat{\theta}_n$ là một ước tính không thiên vị. Nói chung, một ước tính không thiên vị tốt hơn một ước tính thiên vị vì kỳ vọng của nó giống như tham số thực sự.

Tuy nhiên, điều đáng để nhận thức được rằng những người dự kiến thiên vị thường được sử dụng trong thực tế. Có những trường hợp mà các ước lượng không thiên vị không tồn tại mà không có giả định thêm, hoặc không thể chữa được để tính toán. Điều này có vẻ như là một lỗ hổng đáng kể trong một nhà ước tính, tuy nhiên phần lớn các ước lượng gặp phải trong thực tế ít nhất là không thiên vị theo nghĩa là sự thiên vị có xu hướng bằng không vì số lượng mẫu có sẵn có xu hướng vô cùng: $\lim_{n \to \infty} \text{bias}(\hat{\theta}_n) = 0$.

Phương sai và độ lệch chuẩn

Thứ hai, chúng ta hãy đo lường sự ngẫu nhiên trong ước tính. Nhớ lại từ Section 19.6, *độ lệch chuẩn* (hoặc *lỗi tiêu chuẩn*) được định nghĩa là gốc bình phương của phương sai. Chúng tôi có thể đo mức độ dao động của một ước tính bằng cách đo độ lệch chuẩn hoặc phương sai của ước tính đó.

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \tag{19.9.3}$$

Điều quan trọng là so sánh (19.9.3) đến (19.9.1). Trong phương trình này, chúng ta không so sánh với giá trị dân số thực sự θ, mà thay vào đó là $E(\hat{\theta}_n)$, mẫu dự kiến có nghĩa là. Do đó, chúng tôi không đo lường mức độ ước tính có xu hướng từ giá trị thực sự, mà thay vào đó chúng ta đo lường sự dao động của chính ước tính.

Sự Bias-Variance Trade-off

Rõ ràng bằng trực giác rằng hai thành phần chính này góp phần vào lỗi bình phương trung bình. Điều hơi gây sốc là chúng ta có thể cho thấy rằng đây thực sự là một * decomposition* của lỗi bình phương trung bình vào hai đóng góp này cộng với một phần ba. Điều đó có nghĩa là chúng ta có thể viết sai số bình phương trung bình là tổng của bình phương của sự thiên vị, phương sai và lỗi không thể khắc phục được.

$$\begin{aligned}\mathrm{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - \theta)^2] \\ &= E[(\hat{\theta}_n)^2] + E[\theta^2] - 2E[\hat{\theta}_n\theta] \\ &= \mathrm{Var}[\hat{\theta}_n] + E[\hat{\theta}_n]^2 + \mathrm{Var}[\theta] + E[\theta]^2 - 2E[\hat{\theta}_n]E[\theta] \\ &= (E[\hat{\theta}_n] - E[\theta])^2 + \mathrm{Var}[\hat{\theta}_n] + \mathrm{Var}[\theta] \\ &= (E[\hat{\theta}_n - \theta])^2 + \mathrm{Var}[\hat{\theta}_n] + \mathrm{Var}[\theta] \\ &= (\mathrm{bias}[\hat{\theta}_n])^2 + \mathrm{Var}(\hat{\theta}_n) + \mathrm{Var}[\theta].\end{aligned} \tag{19.9.4}$$

Chúng tôi đề cập đến công thức trên là *bias-variance trade-off*. Lỗi bình phương trung bình có thể được chia thành ba nguồn error: the error from high bias, the error from high variance and the irreducible error. The bias error is commonly seen in a simple model (such as a linear regression model), which cannot extract high dimensional relations between the features and the outputs. If a model suffers from high bias error, we often say it is *underfitting* or lack of *flexibilty* as introduced in (Section 5.4). Phương sai cao thường là kết quả từ một mô hình quá phức tạp, vượt quá dữ liệu đào tạo. Do đó, mô hình * overfitting* nhạy cảm với các biến động nhỏ trong dữ liệu. Nếu một mô hình bị phương sai cao, chúng ta thường nói rằng đó là * overfitting* và thiếu *khái quát hóa* như được giới thiệu trong (Section 5.4). Lỗi không thể khắc phục là kết quả từ tiếng ồn trong chính θ.

Đánh giá Ước tính trong Mã

Vì độ lệch chuẩn của một ước tính đã được thực hiện bằng cách gọi đơn giản `a.std()` cho tensor `a`, chúng tôi sẽ bỏ qua nó nhưng thực hiện thiên vị thống kê và sai số bình phương trung bình.

```
# Statistical bias
def stat_bias(true_theta, est_theta):
    return(torch.mean(est_theta) - true_theta)

# Mean squared error
def mse(data, true_theta):
    return(torch.mean(torch.square(data - true_theta)))
```

Để minh họa phương trình của sự cân bằng sai lệch, chúng ta hãy mô phỏng phân phối bình thường $\mathcal{N}(\theta, \sigma^2)$ với $10,000$ mẫu. Ở đây, chúng tôi sử dụng một $\theta = 1$ và $\sigma = 4$. Vì ước tính là một chức năng của các mẫu đã cho, ở đây chúng tôi sử dụng trung bình của các mẫu như một ước tính cho θ đúng trong phân phối bình thường này $\mathcal{N}(\theta, \sigma^2)$.

```
theta_true = 1
sigma = 4
sample_len = 10000
samples = torch.normal(theta_true, sigma, size=(sample_len, 1))
theta_est = torch.mean(samples)
theta_est
```

```
tensor(1.0170)
```

Chúng ta hãy xác nhận phương trình đánh đổi bằng cách tính tổng của sự thiên vị bình phương và phương sai của ước tính của chúng ta. Đầu tiên, tính toán MSE của ước tính của chúng tôi.

```
mse(samples, theta_true)
```

```
tensor(16.0298)
```

Tiếp theo, chúng tôi tính $\text{Var}(\hat{\theta}_n) + [\text{bias}(\hat{\theta}_n)]^2$ như dưới đây. Như bạn có thể thấy, hai giá trị đồng ý với độ chính xác số.

```
bias = stat_bias(theta_true, theta_est)
torch.square(samples.std(unbiased=False)) + torch.square(bias)
```

```
tensor(16.0298)
```

19.9.2 Tiến hành các bài kiểm tra giả thuyết

Chủ đề thường gặp nhất trong suy luận thống kê là thử nghiệm giả thuyết. Trong khi thử nghiệm giả thuyết được phổ biến vào đầu thế kỷ 20, việc sử dụng đầu tiên có thể được truy trở lại John Arbuthnot vào những năm 1700. John đã theo dõi kỷ lục sinh 80 năm ở London và kết luận rằng nhiều nam giới được sinh ra hơn phụ nữ mỗi năm. Tiếp theo đó, thử nghiệm ý nghĩa hiện đại là di sản tình báo của Karl Pearson, người đã phát minh ra p giá trị và bài kiểm tra chi bình phương của Pearson, William Gosset là cha đẻ của phân phối t của Sinh viên, và Ronald Fisher, người đã khởi xướng giả thuyết vô giá trị và bài kiểm tra ý nghĩa.

A *giả thuyết test* là một cách để đánh giá một số bằng chứng chống lại tuyên bố mặc định về một dân số. Chúng tôi tham khảo câu lệnh mặc định là giả thuyết *null * H_0, mà chúng tôi cố gắng từ chối bằng cách sử dụng dữ liệu quan sát. Ở đây, chúng tôi sử dụng H_0 làm điểm khởi đầu cho việc kiểm tra ý nghĩa thống kê. Giả thuyết thay thế* H_A (hoặc H_1) là một tuyên bố trái với giả thuyết null. Một giả thuyết null thường được nêu trong một hình thức khai báo đặt ra một mối quan hệ giữa các biến. Nó sẽ phản ánh ngắn gọn càng rõ ràng càng tốt, và được kiểm tra bởi lý thuyết thống kê.

Hãy tưởng tượng bạn là một nhà hóa học. Sau khi dành hàng ngàn giờ trong phòng thí nghiệm, bạn phát triển một loại thuốc mới có thể cải thiện đáng kể khả năng hiểu toán của một người. Để thể hiện sức mạnh ma thuật của nó, bạn cần phải kiểm tra nó. Đương nhiên, bạn có thể cần một số tình nguyện viên uống thuốc và xem liệu nó có thể giúp họ học toán tốt hơn hay không. Làm thế nào để bạn bắt đầu?

Đầu tiên, bạn sẽ cần hai nhóm tình nguyện viên được lựa chọn ngẫu nhiên cẩn thận, để không có sự khác biệt giữa khả năng hiểu biết toán học của họ được đo bằng một số số chỉ số. Hai nhóm thường được gọi là nhóm thử nghiệm và nhóm kiểm soát. *kiểm tra nhóm* (hoặc *nhóm điều trị*) là một nhóm các cá nhân sẽ trải nghiệm thuốc, trong khi nhóm kiểm soát * đại diện cho nhóm người dùng được đặt sang một bên như một chuẩn mực, tức là thiết lập môi trường giống hệt nhau trừ khi dùng thuốc này. Bằng cách này, ảnh hưởng của tất cả các biến được giảm thiểu, ngoại trừ tác động của biến độc lập trong điều trị.

Thứ hai, sau một thời gian dùng thuốc, bạn sẽ cần đo lường sự hiểu biết toán của hai nhóm bằng các chỉ số tương tự, chẳng hạn như để các tình nguyện viên làm các bài kiểm tra tương tự sau khi học một công thức toán mới. Sau đó, bạn có thể thu thập hiệu suất của họ và so sánh kết quả. Trong trường hợp này, giả thuyết null của chúng ta sẽ là không có sự khác biệt giữa hai nhóm, và sự thay thế của chúng ta sẽ là có.

Điều này vẫn chưa hoàn toàn chính thức. Có rất nhiều chi tiết bạn phải suy nghĩ cẩn thận. Ví dụ, các số liệu phù hợp để kiểm tra khả năng hiểu biết toán học của họ là gì? Có bao nhiêu tình nguyện viên cho bài kiểm tra của bạn để bạn có thể tự tin để khẳng định hiệu quả của thuốc của bạn? Bạn nên chạy bài kiểm tra trong bao lâu? Làm thế nào để bạn quyết định nếu có sự khác biệt giữa hai nhóm? Bạn có quan tâm đến hiệu suất trung bình chỉ, hoặc cũng là phạm vi biến thể của điểm số? Và như vậy.

Bằng cách này, thử nghiệm giả thuyết cung cấp một khuôn khổ cho thiết kế thực nghiệm và lý luận về sự chắc chắn trong các kết quả quan sát được. Nếu bây giờ chúng ta có thể chỉ ra rằng giả thuyết null là rất khó có thể là đúng, chúng ta có thể từ chối nó với sự tự tin.

Để hoàn thành câu chuyện về cách làm việc với thử nghiệm giả thuyết, bây giờ chúng ta cần giới thiệu một số thuật ngữ bổ sung và đưa ra một số khái niệm của chúng tôi ở trên chính thức.

Statistical Significance

The *statistical significance* measures the probability of erroneously rejecting the null hypothesis, H_0, when it should not be rejected, i.e.,

$$\text{statistical significance } = 1 - \alpha = 1 - P(\text{reject } H_0 \mid H_0 \text{ is true}). \tag{19.9.5}$$

It is also referred to as the *type I error* or *false positive*. The α, is called as the *significance level* and its commonly used value is 5%, i.e., $1 - \alpha = 95\%$. The significance level can be explained as the level of risk that we are willing to take, when we reject a true null hypothesis.

Fig. 19.9.1 shows the observations' values and probability of a given normal distribution in a two-sample hypothesis test. If the observation data example is located outsides the 95% threshold, it will be a very unlikely observation under the null hypothesis assumption. Hence, there might be something wrong with the null hypothesis and we will reject it.

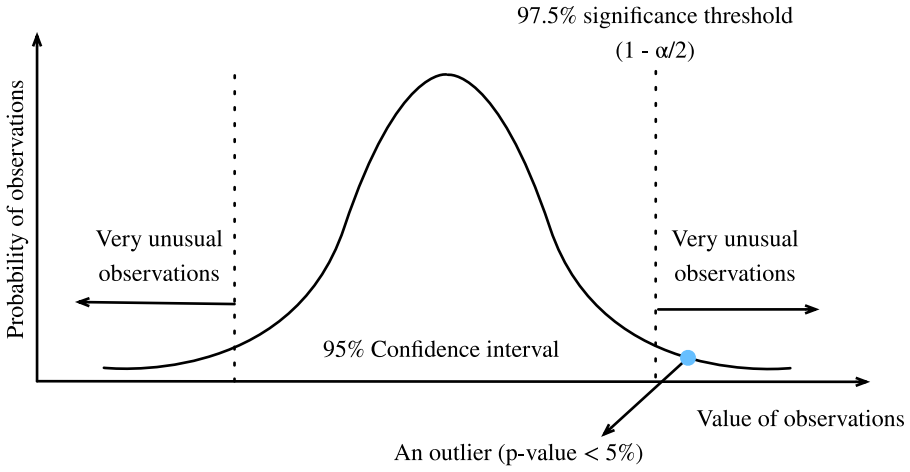


Fig. 19.9.1: Statistical significance.

Sức mạnh thống kê

The *statistical power* (or *sensitivity*) measures the probability of reject the null hypothesis, H_0, when it should be rejected, i.e.,

$$\text{statistical power } = 1 - \beta = 1 - P(\text{ fail to reject } H_0 \mid H_0 \text{ is false}). \tag{19.9.6}$$

Recall that a *type I error* is error caused by rejecting the null hypothesis when it is true, whereas a *type II error* is resulted from failing to reject the null hypothesis when it is false. A type II error is usually denoted as β, and hence the corresponding statistical power is $1 - \beta$.

Intuitively, statistical power can be interpreted as how likely our test will detect a real discrepancy of some minimum magnitude at a desired statistical significance level. 80% is a commonly used statistical power threshold. The higher the statistical power, the more likely we are to detect true differences.

Một trong những cách sử dụng phổ biến nhất của sức mạnh thống kê là xác định số lượng mẫu cần thiết. Xác suất bạn từ chối giả thuyết null khi nó là sai phụ thuộc vào mức độ sai (được gọi là *kích thước hiệu ứng*) và số lượng mẫu bạn có. Như bạn có thể mong đợi, kích thước hiệu ứng nhỏ sẽ yêu cầu một số lượng rất lớn các mẫu được phát hiện với xác suất cao. Trong khi vượt quá phạm vi của phụ lục ngắn gọn này để lấy ra chi tiết, như một ví dụ, muốn có thể từ chối một giả thuyết null rằng mẫu của chúng tôi đến từ một phương sai trung bình bằng 0 một Gaussian, và chúng tôi tin rằng trung bình mẫu của chúng tôi thực sự gần với một, chúng tôi có thể làm như vậy với tỷ lệ lỗi chấp nhận được với kích thước mẫu của chỉ 8. Tuy nhiên, nếu chúng ta nghĩ rằng trung bình thực sự dân số mẫu của chúng ta gần 0.01, thì chúng ta cần một kích thước mẫu gần 80000 để phát hiện sự khác biệt.

Chúng ta có thể tưởng tượng sức mạnh như một bộ lọc nước. Trong sự tương tự này, một bài kiểm tra giả thuyết công suất cao giống như một hệ thống lọc nước chất lượng cao sẽ làm giảm các chất độc hại trong nước càng nhiều càng tốt. Mặt khác, sự khác biệt nhỏ hơn giống như một bộ lọc nước chất lượng thấp, nơi một số chất nhỏ tương đối có thể dễ dàng thoát ra khỏi các khoảng trống. Tương tự, nếu sức mạnh thống kê không đủ công suất cao, thì thử nghiệm có thể không bắt được sự khác biệt nhỏ hơn.

Test Statistic

Một * thống kê thử nghiệm* $T(x)$ là một vô hướng tóm tắt một số đặc điểm của dữ liệu mẫu. Mục tiêu của việc xác định một thống kê như vậy là nó sẽ cho phép chúng ta phân biệt giữa các phân phối khác nhau và tiến hành kiểm tra giả thuyết của chúng tôi. Suy nghĩ lại ví dụ của nhà hóa học của chúng tôi, nếu chúng ta muốn chứng minh rằng một dân số hoạt động tốt hơn người kia, nó có thể là hợp lý để lấy trung bình như thống kê thử nghiệm. Các lựa chọn khác nhau của thống kê thử nghiệm có thể dẫn đến kiểm tra thống kê với sức mạnh thống kê khác nhau đáng kể.

Thông thường, $T(X)$ (sự phân bố của thống kê thử nghiệm theo giả thuyết null của chúng tôi) sẽ theo sau, ít nhất là khoảng, một phân phối xác suất chung như phân phối bình thường khi được xem xét theo giả thuyết null. Nếu chúng ta có thể lấy được một phân phối rõ ràng như vậy, và sau đó đo thống kê thử nghiệm của chúng tôi về tập dữ liệu của chúng tôi, chúng ta có thể từ chối một cách an toàn giả thuyết null nếu thống kê của chúng tôi nằm ngoài phạm vi mà chúng ta mong đợi. Làm cho định lượng này dẫn chúng ta đến khái niệm p-giá trị.

p-value

Giá trị p-(hoặc giá trị xác suất *) là xác suất $T(X)$ ít nhất là cực đoan như thống kê thử nghiệm quan sát $T(x)$ giả định rằng giả thuyết null là * true*, tức là,

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \tag{19.9.7}$$

Nếu giá trị p-nhỏ hơn hoặc bằng một mức ý nghĩa thống kê được xác định trước và cố định α, chúng ta có thể từ chối giả thuyết null. Nếu không, chúng ta sẽ kết luận rằng chúng ta thiếu bằng chứng để từ chối giả thuyết null. Đối với một phân bố dân số nhất định, *khu vực từ khóa* sẽ là khoảng thời gian chứa tất cả các điểm có giá trị p nhỏ hơn mức ý nghĩa thống kê α.

Kiểm tra một mặt và thử nghiệm hai mặt

Thông thường có hai loại kiểm tra ý nghĩa: bài kiểm tra một mặt và thử nghiệm hai mặt. Kiểm tra * một mặt* (hoặc *kiểm tra một đuôi *) được áp dụng khi giả thuyết null và giả thuyết thay thế chỉ có một hướng. Ví dụ, giả thuyết null có thể nói rằng tham số thực θ nhỏ hơn hoặc bằng một giá trị c. Giả thuyết thay thế sẽ là θ lớn hơn c. Đó là, khu vực từ chối chỉ ở một bên của phân phối lấy mẫu. Trái ngược với thử nghiệm một mặt, thử nghiệm hai mặt* (hoặc * kiểm tra hai đuôi *) được áp dụng khi vùng từ chối nằm ở cả hai mặt của phân phối lấy mẫu. Một ví dụ trong trường hợp này có thể có một trạng thái giả thuyết null rằng tham số đúng θ bằng một giá trị c. Giả thuyết thay thế sẽ là θ không bằng c.

Các bước chung của thử nghiệm giả thuyết

Sau khi làm quen với các khái niệm trên, chúng ta hãy trải qua các bước chung của thử nghiệm giả thuyết.

1. Nêu câu hỏi và thiết lập một giả thuyết vô giá trị H_0.
2. Đặt mức ý nghĩa thống kê α và một sức mạnh thống kê $(1 - \beta)$.
3. Lấy mẫu thông qua các thí nghiệm. Số lượng mẫu cần thiết sẽ phụ thuộc vào sức mạnh thống kê và kích thước hiệu ứng mong đợi.
4. Tính số liệu thống kê thử nghiệm và giá trị p-.
5. Đưa ra quyết định giữ hoặc từ chối giả thuyết null dựa trên giá trị p-giá trị và mức ý nghĩa thống kê α.

Để tiến hành một bài kiểm tra giả thuyết, chúng ta bắt đầu bằng cách xác định một giả thuyết null và mức độ rủi ro mà chúng tôi sẵn sàng thực hiện. Sau đó, chúng tôi tính toán thống kê thử nghiệm của mẫu, lấy một giá trị cực đoan của thống kê thử nghiệm làm bằng chứng chống lại giả thuyết null. Nếu thống kê thử nghiệm nằm trong khu vực từ chối, chúng ta có thể từ chối giả thuyết null có lợi cho sự thay thế.

Xét nghiệm giả thuyết được áp dụng trong một loạt các tình huống như đường mòn lâm sàng và thử nghiệm A/B.

19.9.3 Xây dựng khoảng tự tin

Khi ước tính giá trị của một tham số θ, các chứng thực điểm như $\hat{\theta}$ có tiện ích hạn chế vì chúng không chứa khái niệm về sự không chắc chắn. Thay vào đó, sẽ tốt hơn nhiều nếu chúng ta có thể tạo ra một khoảng thời gian có chứa tham số thực θ với xác suất cao. Nếu bạn quan tâm đến những ý tưởng như vậy một thế kỷ trước, thì bạn sẽ rất vui mừng khi đọc "Phác thảo của một lý thuyết ước tính thống kê dựa trên lý thuyết xác suất cổ điển" của Jerzy Neyman (Neyman, 1937), người lần đầu tiên giới thiệu khái niệm khoảng thời gian tin cậy vào năm 1937.

Để hữu ích, một khoảng thời gian tin cậy nên càng nhỏ càng tốt cho một mức độ chắc chắn nhất định. Hãy để chúng tôi xem làm thế nào để lấy được nó.

Definition

Mathematically, a *confidence interval* for the true parameter θ is an interval C_n that computed from the sample data such that

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \tag{19.9.8}$$

Here $\alpha \in (0, 1)$, and $1 - \alpha$ is called the *confidence level* or *coverage* of the interval. This is the same α as the significance level as we discussed about above.

Note that (19.9.8) is about variable C_n, not about the fixed θ. To emphasize this, we write $P_\theta(C_n \ni \theta)$ rather than $P_\theta(\theta \in C_n)$.

Interpretation

It is very tempting to interpret a 95% confidence interval as an interval where you can be 95% sure the true parameter lies, however this is sadly not true. The true parameter is fixed, and it is the interval that is random. Thus a better interpretation would be to say that if you generated a large number of confidence intervals by this procedure, 95% of the generated intervals would contain the true parameter.

This may seem pedantic, but it can have real implications for the interpretation of the results. In particular, we may satisfy (19.9.8) by constructing intervals that we are *almost certain* do not contain the true value, as long as we only do so rarely enough. We close this section by providing three tempting but false statements. An in-depth discussion of these points can be found in (Morey et al., 2016).

- **Fallacy 1**. Narrow confidence intervals mean we can estimate the parameter precisely.
- **Fallacy 2**. The values inside the confidence interval are more likely to be the true value than those outside the interval.
- **Fallacy 3**. The probability that a particular observed 95% confidence interval contains the true value is 95%.

Sufficed to say, confidence intervals are subtle objects. However, if you keep the interpretation clear, they can be powerful tools.

A Gaussian Example

Let us discuss the most classical example, the confidence interval for the mean of a Gaussian of unknown mean and variance. Suppose we collect n samples $\{x_i\}_{i=1}^n$ from our Gaussian $\mathcal{N}(\mu, \sigma^2)$. We can compute estimators for the mean and standard deviation by taking

$$\hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \hat{\sigma}_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2. \tag{19.9.9}$$

If we now consider the random variable

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}}, \tag{19.9.10}$$

we obtain a random variable following a well-known distribution called the *Student's t-distribution on* $n-1$ *degrees of freedom*.

This distribution is very well studied, and it is known, for instance, that as $n \to \infty$, it is approximately a standard Gaussian, and thus by looking up values of the Gaussian c.d.f. in a table, we may conclude that the value of T is in the interval $[-1.96, 1.96]$ at least 95% of the time. For finite values of n, the interval needs to be somewhat larger, but are well known and precomputed in tables.

Thus, we may conclude that for large n,

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n/\sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \tag{19.9.11}$$

Rearranging this by multiplying both sides by $\hat{\sigma}_n/\sqrt{n}$ and then adding $\hat{\mu}_n$, we obtain

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \tag{19.9.12}$$

Thus we know that we have found our 95% confidence interval:

$$\left[\hat{\mu}_n - 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96\frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \tag{19.9.13}$$

It is safe to say that (19.9.13) is one of the most used formula in statistics. Let us close our discussion of statistics by implementing it. For simplicity, we assume we are in the asymptotic regime. Small values of N should include the correct value of `t_star` obtained either programmatically or from a t-table.

```
# PyTorch uses Bessel's correction by default, which means the use of ddof=1
# instead of default ddof=0 in numpy. We can use unbiased=False to imitate
# ddof=0.

# Number of samples
N = 1000

# Sample dataset
samples = torch.normal(0, 1, size=(N,))

# Lookup Students's t-distribution c.d.f.
t_star = 1.96

# Construct interval
mu_hat = torch.mean(samples)
sigma_hat = samples.std(unbiased=True)
(mu_hat - t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)),\
 mu_hat + t_star*sigma_hat/torch.sqrt(torch.tensor(N, dtype=torch.float32)))
```

```
(tensor(-0.0568), tensor(0.0704))
```

19.9.4 Tóm tắt

- Thống kê tập trung vào các bài toán suy luận, trong khi học sâu nhấn mạnh vào việc đưa ra các dự đoán chính xác mà không cần lập trình và hiểu biết rõ ràng.
- Có ba phương pháp suy luận thống kê phổ biến: đánh giá và so sánh các ước lượng, tiến hành các bài kiểm tra giả thuyết, và xây dựng các khoảng tự tin.
- Có ba điều kiện phổ biến nhất: thiên vị thống kê, độ lệch chuẩn và sai số vuông trung bình.
- Khoảng thời gian tin cậy là một phạm vi ước tính của một tham số dân số thực sự mà chúng ta có thể xây dựng bằng cách đưa ra các mẫu.
- Xét nghiệm giả thuyết là một cách để đánh giá một số bằng chứng chống lại tuyên bố mặc định về một dân số.

19.9.5 Bài tập

1. Hãy để $X_1, X_2, \ldots, X_n \overset{\text{iid}}{\sim} \text{Unif}(0, \theta)$, trong đó "iid" là viết tắt của * độc lập và phân phối giống hệt nhau*. Hãy xem xét các chứng thực sau đây của θ:

$$\hat{\theta} = \max\{X_1, X_2, \ldots, X_n\}; \tag{19.9.14}$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n}\sum_{i=1}^{n} X_i. \tag{19.9.15}$$

   - Tìm thiên vị thống kê, độ lệch chuẩn và sai số vuông trung bình của $\hat{\theta}$.
   - Tìm thiên vị thống kê, độ lệch chuẩn và sai số vuông trung bình của $\tilde{\theta}$.
   - Ước tính nào tốt hơn?

2. Đối với ví dụ nhà hóa học của chúng tôi trong phần giới thiệu, bạn có thể lấy được 5 bước để tiến hành thử nghiệm giả thuyết hai mặt không? Với mức ý nghĩa thống kê $\alpha = 0.05$ và sức mạnh thống kê $1 - \beta = 0.8$.

3. Chạy mã khoảng thời gian tin cậy với $N = 2$ và $\alpha = 0.5$ cho 100 tập dữ liệu được tạo độc lập và vẽ các khoảng thời gian kết quả (trong trường hợp này là `t_star = 1.0`). Bạn sẽ thấy một số khoảng thời gian rất ngắn rất xa chứa trung bình thực sự 0. Điều này có mâu thuẫn với việc giải thích khoảng thời gian tin cậy không? Bạn có cảm thấy thoải mái khi sử dụng khoảng thời gian ngắn để chỉ ra ước tính chính xác cao không?

Discussions[221]

[221] https://discuss.d2l.ai/t/1102

19.10 Lý thuyết thông tin

Vũ trụ tràn đầy thông tin. Thông tin cung cấp một ngôn ngữ chung giữa các rạn nứt kỷ luật: từ Sonnet của Shakespeare đến bài báo của các nhà nghiên cứu trên Cornell ArXiv, từ Starry Night in của Van Gogh đến Bản giao hưởng âm nhạc số 5 của Beethoven, từ ngôn ngữ lập trình đầu tiên Plankalkül đến các thuật toán máy học hiện đại. Mọi thứ phải tuân theo các quy tắc của lý thuyết thông tin, bất kể định dạng. Với lý thuyết thông tin, chúng ta có thể đo lường và so sánh lượng thông tin có mặt trong các tín hiệu khác nhau. Trong phần này, chúng tôi sẽ điều tra các khái niệm cơ bản về lý thuyết thông tin và ứng dụng lý thuyết thông tin trong học máy.

Trước khi bắt đầu, chúng ta hãy phác thảo mối quan hệ giữa học máy và lý thuyết thông tin. Machine learning nhằm mục đích trích xuất các tín hiệu thú vị từ dữ liệu và đưa ra các dự đoán quan trọng. Mặt khác, lý thuyết thông tin nghiên cứu mã hóa, giải mã, truyền và thao túng thông tin. Kết quả là lý thuyết thông tin cung cấp ngôn ngữ cơ bản để thảo luận về việc xử lý thông tin trong các hệ thống học máy. Ví dụ, nhiều ứng dụng máy học sử dụng mất liên ngẫu nhiên như được mô tả trong Section 4.4. Sự mất mát này có thể được bắt nguồn trực tiếp từ những cân nhắc lý thuyết thông tin.

19.10.1 Thông tin

Hãy để chúng tôi bắt đầu với "linh hồn" của lý thuyết thông tin: thông tin. *Thông tin* có thể được mã hóa trong bất cứ điều gì với một chuỗi cụ thể của một hoặc nhiều định dạng mã hóa. Giả sử rằng chúng ta tự nhiệm vụ với việc cố gắng xác định một khái niệm về thông tin. Điều gì có thể là điểm khởi đầu của chúng ta?

Hãy xem xét thí nghiệm suy nghĩ sau đây. Chúng tôi có một người bạn với một bộ bài. Họ sẽ xáo trộn bộ bài, lật qua một số thẻ, và cho chúng tôi biết tuyên bố về các thẻ. Chúng tôi sẽ cố gắng đánh giá nội dung thông tin của từng tuyên bố.

Đầu tiên, họ lật qua một lá bài và nói với chúng tôi, "Tôi thấy một lá bài." Điều này cung cấp cho chúng tôi không có thông tin nào cả. Chúng tôi đã chắc chắn rằng đây là trường hợp vì vậy chúng tôi hy vọng thông tin sẽ bằng không.

Tiếp theo, họ lật qua một lá bài và nói, "Tôi thấy một trái tim." Điều này cung cấp cho chúng tôi một số thông tin, nhưng trong thực tế chỉ có 4 bộ đồ khác nhau có thể, mỗi bộ đồ có khả năng như nhau, vì vậy chúng tôi không ngạc nhiên bởi kết quả này. Chúng tôi hy vọng rằng bất kể thước đo thông tin, sự kiện này nên có nội dung thông tin thấp.

Tiếp theo, họ lật qua một lá bài và nói, "Đây là 3 của spades." Đây là thêm thông tin. Thật vậy, có 52 kết quả có khả năng như nhau, và bạn của chúng tôi đã nói với chúng tôi đó là cái nào. Đây phải là một lượng thông tin trung bình.

Hãy để chúng tôi đưa điều này đến cực đoan logic. Giả sử rằng cuối cùng họ lật qua mỗi thẻ từ boong và đọc toàn bộ chuỗi của boong shuffled. Có 52 đô la! $ đơn đặt hàng khác nhau để boong, một lần nữa tất cả đều có khả năng, vì vậy chúng tôi cần rất nhiều thông tin để biết đó là cái nào.

Bất kỳ khái niệm về thông tin chúng tôi phát triển phải phù hợp với trực giác này. Thật vậy, trong các phần tiếp theo, chúng ta sẽ học cách tính toán rằng các sự kiện này có 0 bits, 2 bits, $~5.7 text {bits} $, and $ ~ 225.6 text {bit} $ thông tin tương ứng.

Nếu chúng ta đọc qua những thí nghiệm tư tưởng này, chúng ta thấy một ý tưởng tự nhiên. Là một điểm khởi đầu, thay vì quan tâm đến kiến thức, chúng ta có thể xây dựng ra ý tưởng rằng thông tin đại diện cho mức độ bất ngờ hoặc khả năng trừu tượng của sự kiện. Ví dụ, nếu chúng ta muốn mô tả một sự kiện bất thường, chúng ta cần rất nhiều thông tin. Đối với một sự kiện phổ biến, chúng tôi có thể không cần nhiều thông tin.

Năm 1948, Claude E. Shannon xuất bản *A Math Theory of Communication* (Shannon, 1948) thiết lập lý thuyết thông tin. Trong bài viết của mình, Shannon lần đầu tiên giới thiệu khái niệm về entropy thông tin. Chúng tôi

sẽ bắt đầu hành trình của chúng tôi ở đây.

Tự thông tin

Vì thông tin thể hiện khả năng trừu tượng của một sự kiện, làm thế nào để chúng ta lập bản đồ khả năng với số bit? Shannon giới thiệu thuật ngữ * bit* là đơn vị thông tin, ban đầu được tạo ra bởi John Tukey. Vậy "bit" là gì và tại sao chúng ta sử dụng nó để đo lường thông tin? Trong lịch sử, một máy phát cổ chỉ có thể gửi hoặc nhận hai loại mã: 0 và 1. Thật vậy, mã hóa nhị phân vẫn được sử dụng phổ biến trên tất cả các máy tính kỹ thuật số hiện đại. Bằng cách này, bất kỳ thông tin nào được mã hóa bởi một loạt 0 và 1. Và do đó, một loạt các chữ số nhị phân có chiều dài n chứa n bit thông tin.

Bây giờ, giả sử rằng đối với bất kỳ loạt mã nào, mỗi mã 0 hoặc 1 xảy ra với xác suất $\frac{1}{2}$. Do đó, một sự kiện X với một loạt các mã có độ dài n, xảy ra với xác suất $\frac{1}{2^n}$. Đồng thời, như chúng tôi đã đề cập trước đây, loạt bài này chứa n bit thông tin. Vì vậy, chúng ta có thể khái quát hóa một hàm toán học mà có thể chuyển xác suất p đến số bit? Shannon đã đưa ra câu trả lời bằng cách xác định *tự thông tin*

$$I(X) = -\log_2(p), \tag{19.10.1}$$

như *bit* thông tin chúng tôi đã nhận được cho sự kiện này X. Lưu ý rằng chúng ta sẽ luôn sử dụng logarit base-2 trong phần này. Vì lợi ích của sự đơn giản, phần còn lại của phần này sẽ bỏ qua chỉ số dưới 2 trong ký hiệu logarit, tức là, $\log(.)$ luôn đề cập đến $\log_2(.)$. Ví dụ, mã "0010" có một thông tin tự

$$I(\text{"0010"}) = -\log(p(\text{"0010"})) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits}. \tag{19.10.2}$$

Chúng ta có thể tính toán thông tin tự như hình dưới đây. Trước đó, trước tiên chúng ta hãy nhập tất cả các gói cần thiết trong phần này.

```
import torch
from torch.nn import NLLLoss


def nansum(x):
    # Define nansum, as pytorch doesn't offer it inbuilt.
    return x[~torch.isnan(x)].sum()

def self_information(p):
    return -torch.log2(torch.tensor(p)).item()

self_information(1 / 64)
```

```
6.0
```

19.10.2 Entropy

Vì tự thông tin chỉ đo lường thông tin của một sự kiện rời rạc duy nhất, chúng ta cần một thước đo tổng quát hơn cho bất kỳ biến ngẫu nhiên nào của phân phối rời rạc hoặc liên tục.

Thúc đẩy Entropy

Hãy để chúng tôi cố gắng để có được cụ thể về những gì chúng tôi muốn. Đây sẽ là một tuyên bố không chính thức về những gì được gọi là * tiên đề của entropy Shannon*. Nó sẽ chỉ ra rằng bộ sưu tập các tuyên bố thông thường sau đây buộc chúng ta phải có một định nghĩa duy nhất về thông tin. Một phiên bản chính thức của các tiên đề này, cùng với một số tiên đề khác có thể được tìm thấy trong (Csiszar, 2008).

1. Thông tin chúng ta đạt được bằng cách quan sát một biến ngẫu nhiên không phụ thuộc vào cái mà chúng ta gọi là các phần tử, hoặc sự hiện diện của các yếu tố bổ sung có xác suất không.
2. Thông tin chúng tôi đạt được bằng cách quan sát hai biến ngẫu nhiên không nhiều hơn tổng thông tin chúng tôi đạt được bằng cách quan sát chúng một cách riêng biệt. Nếu chúng độc lập, thì đó chính xác là tổng.
3. Thông tin thu được khi quan sát (gần) một số sự kiện nhất định là (gần) bằng không.

Trong khi chứng minh thực tế này nằm ngoài phạm vi văn bản của chúng ta, điều quan trọng là phải biết rằng điều này xác định duy nhất hình thức mà entropy phải thực hiện. Sự mơ hồ duy nhất mà chúng cho phép là lựa chọn các đơn vị cơ bản, thường được bình thường hóa bằng cách đưa ra lựa chọn chúng ta đã thấy trước đó thông tin được cung cấp bởi một lần lật đồng xu công bằng là một bit.

Definition

Đối với bất kỳ biến ngẫu nhiên X nào theo phân phối xác suất P với hàm mật độ xác suất (p.d.f.) hoặc hàm khối xác suất (p.m.f.) $p(x)$, chúng tôi đo lượng thông tin dự kiến thông qua *entropy* (hoặc *entropy* Shannon*)

$$H(X) = -E_{x \sim P}[\log p(x)]. \tag{19.10.3}$$

Cụ thể, nếu X là rời rạc,

$$H(X) = -\sum_i p_i \log p_i, \text{ where } p_i = P(X_i). \tag{19.10.4}$$

Nếu không, nếu X là liên tục, chúng tôi cũng đề cập đến entropy là * entropyvisai*

$$H(X) = -\int_x p(x) \log p(x) \; dx. \tag{19.10.5}$$

Chúng ta có thể định nghĩa entropy như dưới đây.

```
def entropy(p):
    entropy = - p * torch.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(entropy)
    return out

entropy(torch.tensor([0.1, 0.5, 0.1, 0.3]))
```

```
tensor(1.6855)
```

Giải thích

Bạn có thể tò mò: in the entropy definition (19.10.3), tại sao chúng ta sử dụng kỳ vọng về một logarit tiêu cực? Dưới đây là một số trực giác.

Đầu tiên, tại sao chúng ta sử dụng hàm *logarit* log? Giả sử rằng $p(x) = f_1(x)f_2(x)\ldots, f_n(x)$, trong đó mỗi hàm thành phần $f_i(x)$ độc lập với nhau. Điều này có nghĩa là mỗi $f_i(x)$ đóng góp độc lập vào tổng thông tin thu được từ $p(x)$. Như đã thảo luận ở trên, chúng ta muốn công thức entropy là phụ gia trên các biến ngẫu nhiên độc lập. May mắn thay, log có thể tự nhiên biến một sản phẩm phân phối xác suất thành tổng hợp các thuật ngữ riêng lẻ.

Tiếp theo, tại sao chúng ta sử dụng * âm* log? Trực giác, các sự kiện thường xuyên hơn nên chứa ít thông tin hơn các sự kiện ít phổ biến hơn, vì chúng ta thường thu được nhiều thông tin hơn từ một trường hợp bất thường hơn là từ một trường hợp thông thường. Tuy nhiên, log đang tăng đơn điệu với xác suất, và thực sự tiêu cực cho tất cả các giá trị trong $[0, 1]$. Chúng ta cần xây dựng một mối quan hệ giảm đơn điệu giữa xác suất của các sự kiện và entropy của chúng, điều này lý tưởng sẽ luôn tích cực (không có gì chúng ta quan sát nên buộc chúng ta quên những gì chúng ta đã biết). Do đó, chúng tôi thêm một dấu âm ở phía trước của chức năng log.

Cuối cùng, hàm *expectation* đến từ đâu? Xem xét một biến ngẫu nhiên X. Chúng tôi có thể giải thích thông tin bản thân ($-\log(p)$) là số lượng *ngạc nhiên* chúng tôi có khi thấy một kết quả cụ thể. Thật vậy, khi xác suất tiếp cận bằng không, sự ngạc nhiên trở nên vô hạn. Tương tự, chúng ta có thể giải thích entropy là lượng bất ngờ trung bình từ quan sát X. Ví dụ, hãy tưởng tượng rằng một hệ thống máy đánh bạc phát ra các ký hiệu độc lập thống kê $s_1, \ldots, s_k$ với xác suất tương ứng $p_1, \ldots, p_k$. Sau đó, entropy của hệ thống này bằng với thông tin tự trung bình từ quan sát từng đầu ra, tức là,

$$H(S) = \sum_i p_i \cdot I(s_i) = -\sum_i p_i \cdot \log p_i. \tag{19.10.6}$$

Tính chất của Entropy

Bằng các ví dụ và giải thích trên, chúng ta có thể lấy được các thuộc tính sau của entropy (19.10.3). Ở đây, chúng tôi đề cập đến X là một sự kiện và P là phân phối xác suất của X.

- H (X) geq 0$ for all discrete X (entropy can be negative for continuous X).
- Nếu $X \sim P$ với một p.d.f. hoặc một p.m.f. $p(x)$, và chúng tôi cố gắng ước tính P bởi một phân phối xác suất mới Q với p.d.f. hoặc một p.m.f. 73229363620, sau đó

  $$H(X) = -E_{x\sim P}[\log p(x)] \leq -E_{x\sim P}[\log q(x)], \text{ with equality if and only if } P = Q. \tag{19.10.7}$$

  Alternatively, $H (X) $ gives a lower bound of the average number of bits needed to encode symbols drawn from $P $.

- Nếu $X \sim P$, thì x truyền tải lượng thông tin tối đa nếu nó lây lan đều trong tất cả các kết quả có thể. Cụ thể, nếu phân phối xác suất P là rời rạc với k-lớp $\{p_1, \ldots, p_k\}$, thì

  $$H(X) \leq \log(k), \text{ with equality if and only if } p_i = \frac{1}{k}, \forall i. \tag{19.10.8}$$

  If P is a continuous random variable, then the story becomes much more complicated. However, if we additionally impose that P is supported on a finite interval (with all values between 0 and 1), then $P $ có entropy cao nhất nếu đó là sự phân bố thống nhất trong khoảng thời gian đó.

19.10.3 Thông tin lẫn nhau

Trước đây chúng ta đã định nghĩa entropy của một biến ngẫu nhiên duy nhất X, làm thế nào về entropy của một cặp biến ngẫu nhiên (X, Y)? Chúng ta có thể nghĩ về những kỹ thuật này là cố gắng trả lời loại câu hỏi sau đây, "Thông tin nào được chứa trong X và Y cùng nhau so với từng loại riêng biệt? Có thông tin dư thừa, hay tất cả là duy nhất?"

Đối với các cuộc thảo luận sau đây, chúng tôi luôn sử dụng (X, Y) như một cặp biến ngẫu nhiên mà sau một phân phối xác suất chung P với một p.d.f. hoặc một p.m.f. $p_{X,Y}(x, y)$, trong khi X và Y theo phân phối xác suất $p_X(x)$ và $p_Y(y)$, tương ứng.

Entropy chung

Tương tự như entropy của một biến ngẫu nhiên duy nhất (19.10.3), chúng tôi xác định entropy* doanh entropy* $H(X, Y)$ của một cặp biến ngẫu nhiên (X, Y) như

$$H(X, Y) = -E_{(x,y)\sim P}[\log p_{X,Y}(x, y)]. \tag{19.10.9}$$

Chính xác, một mặt, nếu (X, Y) là một cặp biến ngẫu nhiên rời rạc, thì

$$H(X, Y) = -\sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \tag{19.10.10}$$

Mặt khác, nếu (X, Y) là một cặp biến ngẫu nhiên liên tục, thì ta định nghĩa entropydoanh vi sai * như

$$H(X, Y) = -\int_{x,y} p_{X,Y}(x, y)\ \log p_{X,Y}(x, y)\ dx\ dy. \tag{19.10.11}$$

Chúng ta có thể nghĩ về (19.10.9) như cho chúng ta biết tổng ngẫu nhiên trong cặp biến ngẫu nhiên. Là một cặp cực đoan, nếu $X = Y$ là hai biến ngẫu nhiên giống hệt nhau, thì thông tin trong cặp chính xác là thông tin trong một và chúng ta có $H(X, Y) = H(X) = H(Y)$. Ở thái cực khác, nếu X và Y độc lập thì $H(X, Y) = H(X) + H(Y)$. Thật vậy chúng ta sẽ luôn có rằng thông tin chứa trong một cặp biến ngẫu nhiên không nhỏ hơn entropy của một trong hai biến ngẫu nhiên và không nhiều hơn tổng của cả hai.

$$H(X), H(Y) \leq H(X, Y) \leq H(X) + H(Y). \tag{19.10.12}$$

Hãy để chúng tôi thực hiện entropy chung từ đầu.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * torch.log2(p_xy)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(joint_ent)
    return out

joint_entropy(torch.tensor([[0.1, 0.5], [0.1, 0.3]]))
```

```
tensor(1.6855)
```

Chú ý rằng đây là cùng *code* như trước đây, nhưng bây giờ chúng ta giải thích nó khác nhau như làm việc trên phân phối chung của hai biến ngẫu nhiên.

Entropy có điều kiện

Entropy chung được xác định phía trên lượng thông tin chứa trong một cặp biến ngẫu nhiên. Điều này rất hữu ích, nhưng đôi khi nó không phải là những gì chúng ta quan tâm. Xem xét các thiết lập của máy học. Chúng ta hãy lấy X là biến ngẫu nhiên (hoặc vector của các biến ngẫu nhiên) mô tả các giá trị điểm ảnh của một hình ảnh, và Y là biến ngẫu nhiên là nhãn lớp. X nên chứa thông tin đáng kể - một hình ảnh tự nhiên là một điều phức tạp. Tuy nhiên, thông tin chứa trong Y khi hình ảnh đã được hiển thị nên thấp. Thật vậy, hình ảnh của một chữ số nên đã chứa thông tin về chữ số nào trừ khi chữ số không thể đọc được. Do đó, để tiếp tục mở rộng vốn từ vựng của chúng ta về lý thuyết thông tin, chúng ta cần có khả năng lý luận về nội dung thông tin trong một biến ngẫu nhiên có điều kiện trên một biến khác.

Trong lý thuyết xác suất, chúng ta đã thấy định nghĩa về xác suất có điều kiện * để đo lường mối quan hệ giữa các biến. Bây giờ chúng ta muốn xác định tương tự các entropy* có điều kiện* $H(Y \mid X)$. Chúng ta có thể viết cái này như

$$H(Y \mid X) = -E_{(x,y)\sim P}[\log p(y \mid x)], \tag{19.10.13}$$

trong đó $p(y \mid x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$ là xác suất có điều kiện. Cụ thể, nếu (X, Y) là một cặp biến ngẫu nhiên rời rạc, thì

$$H(Y \mid X) = -\sum_x \sum_y p(x,y) \log p(y \mid x). \tag{19.10.14}$$

Nếu (X, Y) là một cặp biến ngẫu nhiên liên tục, thì entropy có điều kiện vi sai * được định nghĩa tương tự như

$$H(Y \mid X) = -\int_x \int_y p(x,y)\ \log p(y \mid x)\ dx\ dy. \tag{19.10.15}$$

Bây giờ nó là tự nhiên để hỏi, làm thế nào để entropy* có điều kiện* $H(Y \mid X)$ liên quan đến entropy $H(X)$ và entropy khớp $H(X, Y)$? Sử dụng các định nghĩa ở trên, chúng ta có thể thể hiện điều này một cách sạch sẽ:

$$H(Y \mid X) = H(X, Y) - H(X). \tag{19.10.16}$$

Điều này có một cách giải thích trực quan: thông tin trong Y được đưa ra X ($H(Y \mid X)$) giống như thông tin trong cả X và Y cùng nhau ($H(X, Y)$) trừ đi thông tin đã có trong X. Điều này cung cấp cho chúng ta thông tin trong Y mà cũng không được đại diện trong X.

Bây giờ, chúng ta hãy thực hiện entropy có điều kiện (19.10.13) từ đầu.

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * torch.log2(p_y_given_x)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(cond_ent)
    return out

conditional_entropy(torch.tensor([[0.1, 0.5], [0.2, 0.3]]),
                    torch.tensor([0.2, 0.8]))
```

```
tensor(0.8635)
```

Thông tin lẫn nhau

Với cài đặt trước đó của các biến ngẫu nhiên (X, Y), bạn có thể tự hỏi: "Bây giờ chúng ta biết có bao nhiêu thông tin được chứa trong Y nhưng không phải trong X, chúng ta có thể hỏi tương tự bao nhiêu thông tin được chia sẻ giữa X và Y không?" Câu trả lời sẽ là * thông tin lẫn nhau* của (X, Y), mà chúng tôi sẽ viết là $I(X, Y)$.

Thay vì đi thẳng vào định nghĩa chính thức, chúng ta hãy thực hành trực giác của mình bằng cách đầu tiên cố gắng lấy được một biểu hiện cho thông tin lẫn nhau hoàn toàn dựa trên các thuật ngữ mà chúng ta đã xây dựng trước đây. Chúng tôi muốn tìm thông tin được chia sẻ giữa hai biến ngẫu nhiên. Một cách chúng ta có thể cố gắng làm điều này là bắt đầu với tất cả thông tin có trong cả X và Y cùng nhau, và sau đó chúng tôi gỡ bỏ các phần không được chia sẻ. Thông tin chứa trong cả X và Y cùng nhau được viết là $H(X, Y)$. Chúng tôi muốn trừ đi thông tin này trong X nhưng không phải trong Y và thông tin chứa trong Y nhưng không phải trong X. Như chúng ta đã thấy trong phần trước, điều này được đưa ra bởi $H(X \mid Y)$ và $H(Y \mid X)$ tương ứng. Vì vậy, chúng tôi có rằng thông tin lẫn nhau nên được

$$I(X, Y) = H(X, Y) - H(Y \mid X) - H(X \mid Y). \tag{19.10.17}$$

Thật vậy, đây là một định nghĩa hợp lệ cho các thông tin lẫn nhau. Nếu chúng ta mở rộng các định nghĩa của các thuật ngữ này và kết hợp chúng, một chút đại số cho thấy điều này giống như

$$I(X, Y) = E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x) p_Y(y)} \right\}. \tag{19.10.18}$$

Chúng ta có thể tóm tắt tất cả các mối quan hệ này trong hình Fig. 19.10.1. Đó là một thử nghiệm tuyệt vời của trực giác để xem tại sao các tuyên bố sau đây đều tương đương với $I(X, Y)$.

- $H (X) – H (X giữa Y) $
- $H (Y) – H (Y giữa X) $
- $H (X) + H (Y) – H (X, Y) $

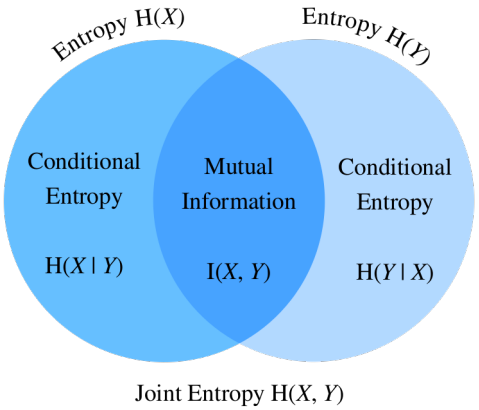


Fig. 19.10.1: Mutual information's relationship with joint entropy and conditional entropy.

Theo nhiều cách, chúng ta có thể nghĩ về thông tin lẫn nhau (19.10.18) là phần mở rộng nguyên tắc của hệ số tương quan mà chúng ta đã thấy trong Section 19.6. Điều này cho phép chúng tôi yêu cầu không chỉ mối quan hệ tuyến tính giữa các biến, mà còn cho thông tin tối đa được chia sẻ giữa hai biến ngẫu nhiên của bất kỳ loại nào.

Bây giờ, chúng ta hãy thực hiện thông tin lẫn nhau từ đầu.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * torch.log2(p)
    # Operator `nansum` will sum up the non-nan number
    out = nansum(mutual)
    return out

mutual_information(torch.tensor([[0.1, 0.5], [0.1, 0.3]]),
                   torch.tensor([0.2, 0.8]), torch.tensor([[0.75, 0.25]]))
```

```
tensor(0.7195)
```

Properties of Mutual Information

Rather than memorizing the definition of mutual information (19.10.18), you only need to keep in mind its notable properties:

- Mutual information is symmetric, i.e., $I(X, Y) = I(Y, X)$.
- Mutual information is non-negative, i.e., $I(X, Y) \geq 0$.
- $I(X, Y) = 0$ if and only if X and Y are independent. For example, if X and Y are independent, then knowing Y does not give any information about X and vice versa, so their mutual information is zero.
- Alternatively, if X is an invertible function of Y, then Y and X share all information and

$$I(X, Y) = H(Y) = H(X). \tag{19.10.19}$$

Pointwise Thông tin lẫn nhau

Khi chúng tôi làm việc với entropy vào đầu chương này, chúng tôi đã có thể cung cấp một giải thích về $-\log(p_X(x))$ như thế nào *ngạc nhiên* chúng tôi đã với kết quả cụ thể. Chúng tôi có thể đưa ra một cách giải thích tương tự cho thuật ngữ logarit trong thông tin lẫn nhau, thường được gọi là thông tin lẫn nhau * pointwise*:

$$\text{pmi}(x, y) = \log \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}. \tag{19.10.20}$$

Chúng ta có thể nghĩ về (19.10.20) là đo lường bao nhiêu nhiều khả năng sự kết hợp cụ thể của kết quả x và y được so sánh với những gì chúng ta mong đợi cho kết quả ngẫu nhiên độc lập. Nếu nó lớn và tích cực, thì hai kết quả cụ thể này xảy ra thường xuyên hơn nhiều so với cơ hội ngẫu nhiên (* lưu ý*: mẫu số là $p_X(x)p_Y(y)$, xác suất của hai kết quả là độc lập), trong khi nếu nó lớn và tiêu cực, nó đại diện cho hai kết quả xảy ra xa lessít hơn than we would expect đợi by randomngẫu nhiên chancecơ hội.

Điều này cho phép chúng tôi giải thích thông tin lẫn nhau (19.10.18) là số tiền trung bình mà chúng tôi ngạc nhiên khi thấy hai kết quả xảy ra cùng nhau so với những gì chúng tôi mong đợi nếu chúng độc lập.

Các ứng dụng của thông tin lẫn nhau

Thông tin lẫn nhau có thể là một chút trừu tượng trong nó định nghĩa thuần túy, vậy nó liên quan đến máy học như thế nào? Trong xử lý ngôn ngữ tự nhiên, một trong những vấn đề khó khăn nhất là độ phân giải *mơ hồ *, hoặc vấn đề về ý nghĩa của một từ không rõ ràng từ ngữ cảnh. Ví dụ, gần đây một tiêu đề trong tin tức báo cáo rằng "Amazon đang cháy". Bạn có thể tự hỏi liệu công ty Amazon có một tòa nhà đang cháy, hoặc rừng mưa Amazon đang cháy.

Trong trường hợp này, thông tin lẫn nhau có thể giúp chúng ta giải quyết sự mơ hồ này. Trước tiên, chúng tôi tìm thấy nhóm từ mà mỗi từ có thông tin tương đối lớn với công ty Amazon, chẳng hạn như thương mại điện tử, công nghệ và trực tuyến. Thứ hai, chúng tôi tìm thấy một nhóm từ khác mà mỗi từ có thông tin tương đối lớn với rừng mưa Amazon, chẳng hạn như mưa, rừng và nhiệt đới. Khi chúng ta cần định hướng "Amazon", chúng ta có thể so sánh nhóm nào có nhiều sự xuất hiện hơn trong bối cảnh của từ Amazon. Trong trường hợp này, bài viết sẽ tiếp tục mô tả rừng và làm cho bối cảnh rõ ràng.

19.10.4 Kullback — Phân kỳ Leibler

Như những gì chúng ta đã thảo luận trong Section 3.3, chúng ta có thể sử dụng các định mức để đo khoảng cách giữa hai điểm trong không gian của bất kỳ chiều chiều nào. Chúng tôi muốn có thể thực hiện một nhiệm vụ tương tự với các phân phối xác suất. Có nhiều cách để đi về điều này, nhưng lý thuyết thông tin cung cấp một trong những cách đẹp nhất. Bây giờ chúng ta khám phá sự phân kỳ *Kullback—Leibler (KL) *, cung cấp một cách để đo lường nếu hai phân phối có gần nhau hay không.

Definition

Cho một biến ngẫu nhiên X mà sau phân phối xác suất P với một p.d.f. hoặc một p.m.f. $p(x)$, và chúng tôi ước tính P bởi một phân phối xác suất khác Q với một p.d.f. hoặc một p.m.f. $q(x)$. Sau đó, phân kỳ *Kullback—Leibler (KL) * (hoặc * entropy tương tự*) giữa P và Q là

$$D_{\mathrm{KL}}(P\|Q) = E_{x\sim P}\left[\log \frac{p(x)}{q(x)}\right]. \tag{19.10.21}$$

Như với thông tin tương hỗ theo chiều ngang (19.10.20), chúng ta lại có thể cung cấp một cách giải thích thuật ngữ logarit: $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$ sẽ lớn và tích cực nếu chúng ta thấy x thường xuyên hơn nhiều dưới P so với chúng ta mong đợi cho Q, và lớn và tiêu cực nếu chúng ta thấy kết quả ít hơn nhiều so với dự kiến. Bằng cách này, chúng tôi có thể hiểu nó là bất ngờ * tương tự* của chúng tôi khi quan sát kết quả so với việc chúng tôi sẽ ngạc nhiên như thế nào chúng tôi sẽ quan sát nó từ phân phối tham chiếu của chúng tôi.

Hãy để chúng tôi thực hiện sự phân kỳ KL từ Scratch.

```
def kl_divergence(p, q):
    kl = p * torch.log2(p / q)
    out = nansum(kl)
    return out.abs().item()
```

Thuộc tính phân kỳ KL

Chúng ta hãy xem xét một số thuộc tính của sự phân kỳ KL (19.10.21).

- Sự phân kỳ KL là không đối xứng, tức là, có P, Q sao cho

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P). \tag{19.10.22}$$

- Sự phân kỳ KL là không âm tính, tức là

$$D_{\mathrm{KL}}(P\|Q) \geq 0. \tag{19.10.23}$$

  Note that the equality holds only when $P = Q$.

- Nếu tồn tại một x như vậy mà $p(x) > 0$ và $q(x) = 0$, sau đó $D_{\mathrm{KL}}(P\|Q) = \infty$.
- Có một mối quan hệ chặt chẽ giữa sự phân kỳ KL và thông tin lẫn nhau. Bên cạnh mối quan hệ thể hiện trong Fig. 19.10.1, $I(X, Y)$ cũng tương đương số với các thuật ngữ sau:
  1. $D_{\mathrm{KL}}(P(X, Y) \parallel P(X)P(Y))$;
  2. $E_Y\{D_{\mathrm{KL}}(P(X \mid Y) \parallel P(X))\}$;
  3. $E_X\{D_{\mathrm{KL}}(P(Y \mid X) \parallel P(Y))\}$.

Đối với nhiệm kỳ đầu tiên, chúng tôi giải thích thông tin lẫn nhau là sự phân kỳ KL giữa $P(X, Y)$ và sản phẩm của $P(X)$ và $P(Y)$, và do đó là thước đo sự khác biệt của sự phân phối chung so với phân phối nếu chúng độc lập. Đối với thuật ngữ thứ hai, thông tin lẫn nhau cho chúng ta biết sự giảm trung bình về sự không chắc chắn về Y là kết quả từ việc tìm hiểu giá trị của phân phối X. Tương tự như nhiệm kỳ thứ ba.

Ví dụ

Chúng ta hãy đi qua một ví dụ đồ chơi để xem sự không đối xứng một cách rõ ràng.

Đầu tiên, chúng ta hãy tạo ra và sắp xếp ba hàng chục chiều dài $10,000$: một tensor khách quan p mà sau một phân phối bình thường $N(0, 1)$, và hai học sinh ứng cử viên q_1 và q_2 mà theo phân phối bình thường $N(-1, 1)$ và $N(1, 1)$ tương ứng.

```
torch.manual_seed(1)

tensor_len = 10000
p = torch.normal(0, 1, (tensor_len, ))
q1 = torch.normal(-1, 1, (tensor_len, ))
q2 = torch.normal(1, 1, (tensor_len, ))

p = torch.sort(p)[0]
q1 = torch.sort(q1)[0]
q2 = torch.sort(q2)[0]
```

Vì q_1 và q_2 đối xứng với trục y (tức là $x = 0$), chúng tôi mong đợi một giá trị tương tự của sự phân kỳ KL giữa $D_{\mathrm{KL}}(p\|q_1)$ và $D_{\mathrm{KL}}(p\|q_2)$. Như bạn có thể thấy bên dưới, chỉ có một ít hơn 3% tắt giữa $D_{\mathrm{KL}}(p\|q_1)$ và $D_{\mathrm{KL}}(p\|q_2)$.

```
kl_pq1 = kl_divergence(p, q1)
kl_pq2 = kl_divergence(p, q2)
similar_percentage = abs(kl_pq1 - kl_pq2) / ((kl_pq1 + kl_pq2) / 2) * 100

kl_pq1, kl_pq2, similar_percentage
```

```
(8582.0341796875, 8828.3095703125, 2.8290698237936858)
```

Ngược lại, bạn có thể thấy rằng $D_{\mathrm{KL}}(q_2\|p)$ và $D_{\mathrm{KL}}(p\|q_2)$ bị tắt rất nhiều, với khoảng 40% giảm giá như hình dưới đây.

```
kl_q2p = kl_divergence(q2, p)
differ_percentage = abs(kl_q2p - kl_pq2) / ((kl_q2p + kl_pq2) / 2) * 100

kl_q2p, differ_percentage
```

```
(14130.125, 46.18621024399691)
```

19.10.5 Cross-Entropy

Nếu bạn tò mò về các ứng dụng của lý thuyết thông tin trong học sâu, đây là một ví dụ nhanh. Chúng tôi xác định phân phối thực sự P với phân phối xác suất $p(x)$ và phân phối ước tính Q với phân phối xác suất $q(x)$ và chúng tôi sẽ sử dụng chúng trong phần còn lại của phần này.

Giả sử chúng ta cần giải quyết một bài toán phân loại nhị phân dựa trên n ví dụ dữ liệu cho $\{x_1, \ldots, x_n\}$. Giả sử rằng chúng tôi mã hóa 1 và 0 như là nhãn lớp dương và âm y_i tương ứng, và mạng thần kinh của chúng tôi được tham số hóa bởi θ. Nếu chúng ta muốn tìm một θ tốt nhất để $\hat{y}_i = p_\theta(y_i \mid x_i)$, nó là tự nhiên để áp dụng cách tiếp cận khả năng đăng nhập tối đa như đã thấy trong Section 19.7. Cụ thể, đối với nhãn thực y_i và dự đoán $\hat{y}_i = p_\theta(y_i \mid x_i)$, xác suất được phân loại là dương tính là $\pi_i = p_\theta(y_i = 1 \mid x_i)$. Do đó, hàm log-likelihood sẽ là

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^{n} \pi_i^{y_i} (1 - \pi_i)^{1 - y_i} \\ &= \sum_{i=1}^{n} y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{19.10.24}$$

Tối đa hóa chức năng log-likelihood $l(\theta)$ giống hệt với việc giảm thiểu $-l(\theta)$, và do đó chúng ta có thể tìm thấy θ tốt nhất từ đây. Để khái quát hóa tổn thất trên đối với bất kỳ bản phân phối nào, chúng tôi cũng gọi $-l(\theta)$ là mất *cross-entropy* $\mathrm{CE}(y, \hat{y})$, trong đó y theo dõi phân phối thực sự P và $\hat{y}$ theo phân phối ước tính Q.

Tất cả điều này có nguồn gốc bằng cách làm việc từ quan điểm khả năng tối đa. Tuy nhiên, nếu chúng ta nhìn kỹ, chúng ta có thể thấy rằng các thuật ngữ như $\log(\pi_i)$ đã nhập vào tính toán của chúng ta, đó là một dấu hiệu vững chắc cho thấy chúng ta có thể hiểu được biểu thức từ quan điểm lý thuyết thông tin.

Formal Definition

Giống như sự phân kỳ KL, đối với một biến ngẫu nhiên X, chúng ta cũng có thể đo phân kỳ giữa phân phối ước tính Q và phân phối thật P qua *cross-entropy*,

$$\text{CE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{19.10.25}$$

Bằng cách sử dụng các thuộc tính của entropy được thảo luận ở trên, chúng ta cũng có thể giải thích nó như là tổng kết của entropy $H(P)$ và phân kỳ KL giữa P và Q, tức là,

$$\text{CE}(P, Q) = H(P) + D_{\text{KL}}(P \| Q). \tag{19.10.26}$$

Chúng ta có thể thực hiện sự mất mát cross-entropy như dưới đây.

```
def cross_entropy(y_hat, y):
    ce = -torch.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Bây giờ xác định hai hàng chục cho các nhãn và dự đoán, và tính toán sự mất mát cross-entropy của chúng.

```
labels = torch.tensor([0, 2])
preds = torch.tensor([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
tensor(0.9486)
```

Thuộc tính

Như đã ám chỉ ở phần đầu của phần này, cross-entropy (19.10.25) có thể được sử dụng để định nghĩa một hàm mất mát trong bài toán tối ưu hóa. Nó chỉ ra rằng những điều sau đây là tương đương:

1. Tối đa hóa xác suất dự đoán của Q cho phân phối P, (tức là, $ E_ {x sim P} [log (q (x))] $);
2. Giảm thiểu chéo entropy $\text{CE}(P, Q)$;
3. Giảm thiểu sự phân kỳ KL $D_{\text{KL}}(P \| Q)$.

Định nghĩa của cross-entropy gián tiếp chứng minh mối quan hệ tương đương giữa mục tiêu 2 và mục tiêu 3, miễn là entropy của dữ liệu thật $H(P)$ là không đổi.

Cross-Entropy như một chức năng mục tiêu của phân loại đa lớp

Nếu chúng ta đi sâu vào hàm mục tiêu phân loại với tổn thất chéo entropy CE, chúng ta sẽ thấy việc giảm thiểu CE tương đương với việc tối đa hóa hàm khả năng log L.

Để bắt đầu, giả sử rằng chúng ta được cung cấp một tập dữ liệu với n ví dụ và nó có thể được phân loại thành k-lớp. Đối với mỗi ví dụ dữ liệu i, chúng tôi đại diện cho bất kỳ nhãn k-lớp $\mathbf{y}_i = (y_{i1}, \ldots, y_{ik})$ bằng cách * mã hóa một nóng*. Cụ thể, nếu ví dụ i thuộc về lớp j, thì chúng ta đặt mục j-th thành 1 và tất cả các thành phần khác thành 0, tức là,

$$y_{ij} = \begin{cases} 1 & j \in J; \\ 0 & \text{otherwise.} \end{cases} \tag{19.10.27}$$

Ví dụ, nếu một bài toán phân loại nhiều lớp chứa ba lớp A, B và C, thì nhãn $\mathbf{y}_i$ có thể được mã hóa trong $\{A : (1, 0, 0); B : (0, 1, 0); C : (0, 0, 1)\}$.

Giả sử rằng mạng thần kinh của chúng tôi được tham số hóa bởi θ. Đối với vectơ nhãn thật $\mathbf{y}_i$ và dự đoán

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^{k} y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{19.10.28}$$

Do đó, *cross-entropy mất* sẽ là

$$\mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} \mathbf{y}_i \log \hat{\mathbf{y}}_i = -\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \tag{19.10.29}$$

Mặt khác, chúng ta cũng có thể tiếp cận vấn đề thông qua ước tính khả năng tối đa. Để bắt đầu, chúng ta hãy nhanh chóng giới thiệu một phân phối multinoulli k lớp. Nó là một phần mở rộng của phân phối Bernoulli từ lớp nhị phân sang đa lớp. Nếu một biến ngẫu nhiên $\mathbf{z} = (z_1, \ldots, z_k)$ theo sau một k-class *multinoulli phân phối * với xác suất $\mathbf{p} = (p_1, \ldots, p_k)$, tức là

$$p(\mathbf{z}) = p(z_1, \ldots, z_k) = \mathrm{Multi}(p_1, \ldots, p_k), \text{ where } \sum_{i=1}^{k} p_i = 1, \tag{19.10.30}$$

then the joint probability mass function(p.m.f.) of $ mathbf {z} $ là

$$\mathbf{p}^{\mathbf{z}} = \prod_{j=1}^{k} p_j^{z_j}. \tag{19.10.31}$$

Có thể thấy rằng nhãn của mỗi ví dụ dữ liệu, $\mathbf{y}_i$, đang theo một phân phối multinoulli k-lớp với xác suất $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$. Do đó, p.m.f. chung của mỗi ví dụ dữ liệu $\mathbf{y}_i$ là $\pi^{\mathbf{y}_i} = \prod_{j=1}^{k} \pi_j^{y_{ij}}$. Do đó, chức năng log-likelihood sẽ là

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^{n} \boldsymbol{\pi}^{\mathbf{y}_i} = \log \prod_{i=1}^{n} \prod_{j=1}^{k} \pi_j^{y_{ij}} = \sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} \log \pi_j. \tag{19.10.32}$$

Kể từ khi ước tính khả năng tối đa, chúng tôi tối đa hóa hàm khách quan $l(\theta)$ bằng cách có $\pi_j = p_\theta(y_{ij} \mid \mathbf{x}_i)$. Do đó, đối với bất kỳ phân loại đa lớp nào, tối đa hóa hàm log-likelihood trên $l(\theta)$ tương đương với việc giảm thiểu tổn thất CE $\mathrm{CE}(y, \hat{y})$.

Để kiểm tra bằng chứng trên, chúng ta hãy áp dụng biện pháp tích hợp `NegativeLogLikelihood`. Sử dụng tương tự `labels` và `preds` như trong ví dụ trước đó, chúng ta sẽ nhận được tổn thất số tương tự như ví dụ trước cho đến 5 vị trí thập phân.

```
# Implementation of cross-entropy loss in PyTorch combines `nn.LogSoftmax()`
# and `nn.NLLLoss()`
nll_loss = NLLLoss()
loss = nll_loss(torch.log(preds), labels)
loss
```

```
tensor(0.9486)
```

19.10.6 Tóm tắt

- Lý thuyết thông tin là một lĩnh vực nghiên cứu về mã hóa, giải mã, truyền tải, và thao túng thông tin.
- Entropy là đơn vị để đo lượng thông tin được trình bày trong các tín hiệu khác nhau.
- Sự phân kỳ KL cũng có thể đo lường sự phân kỳ giữa hai bản phân phối.
- Cross-entropy có thể được xem như là một chức năng khách quan của phân loại nhiều lớp. Giảm thiểu tổn thất chéo entropy tương đương với việc tối đa hóa chức năng log-likelihood.

19.10.7 Bài tập

1. Xác minh rằng các ví dụ thẻ từ phần đầu tiên thực sự có entropy tuyên bố.
2. Cho thấy sự phân kỳ KL $D(p\|q)$ là không âm tính cho tất cả các bản phân phối p và q. Gợi ý: sử dụng bất đẳng thức của Jensen, tức là sử dụng thực tế là $-\log x$ là một hàm lồi.
3. Hãy để chúng tôi tính toán entropy từ một vài nguồn dữ liệu:
    - Giả sử rằng bạn đang xem đầu ra được tạo ra bởi một con khỉ tại một máy đánh chữ. Khỉ nhấn bất kỳ phím 44 nào của máy đánh chữ một cách ngẫu nhiên (bạn có thể giả định rằng nó chưa phát hiện ra bất kỳ phím đặc biệt nào hoặc phím shift nào). Bạn quan sát bao nhiêu bit ngẫu nhiên trên mỗi ký tự?
    - Không hài lòng với con khỉ, bạn đã thay thế nó bằng một máy sắp chữ say rượu. Nó có thể tạo ra từ, mặc dù không mạch lạc. Thay vào đó, nó chọn một từ ngẫu nhiên từ một từ vựng gồm $2,000$ từ. Chúng ta hãy giả định rằng độ dài trung bình của một từ là 4.5 chữ cái trong tiếng Anh. Bạn quan sát bao nhiêu bit ngẫu nhiên trên mỗi ký tự bây giờ?
    - Vẫn không hài lòng với kết quả, bạn thay thế máy sắp chữ bằng một mô hình ngôn ngữ chất lượng cao. Mô hình ngôn ngữ hiện có thể có được sự bối rối thấp tới 15 điểm cho mỗi từ. Ký tự *perplexity* của một mô hình ngôn ngữ được định nghĩa là nghịch đảo của trung bình hình học của một tập hợp xác suất, mỗi xác suất tương ứng với một ký tự trong từ. Cụ thể, nếu độ dài của một từ nhất định là l, thì $\text{PPL}(\text{word}) = [\prod_i p(\text{character}_i)]^{-\frac{1}{l}} = \exp\left[-\frac{1}{l}\sum_i \log p(\text{character}_i)\right]$. Giả sử rằng từ kiểm tra có 4,5 chữ cái, bạn quan sát bao nhiêu bit ngẫu nhiên trên mỗi ký tự bây giờ?
4. Giải thích trực giác tại sao $I(X,Y) = H(X) - H(X|Y)$. Sau đó, cho thấy điều này là đúng bằng cách thể hiện cả hai bên như một kỳ vọng liên quan đến phân phối chung.
5. Sự phân kỳ KL giữa hai bản phân phối Gaussian $\mathcal{N}(\mu_1, \sigma_1^2)$ và $\mathcal{N}(\mu_2, \sigma_2^2)$ là gì?

Discussions[222]

[222] https://discuss.d2l.ai/t/1104

20 | Phụ lục: Các công cụ cho Deep Learning

Trong chương này, chúng tôi sẽ hướng dẫn bạn qua các công cụ chính để học sâu, từ giới thiệu máy tính xách tay Jupyter trong Section 20.1 để trao quyền cho bạn các mô hình đào tạo trên Cloud như Amazon SageMaker trong Section 20.2, Amazon EC2 trong Section 20.3 và Google Colab trong Section 20.4. Bên cạnh đó, nếu bạn muốn mua GPU của riêng mình, chúng tôi cũng lưu ý đến một số gợi ý thiết thực trong Section 20.5. Nếu bạn quan tâm đến việc trở thành người đóng góp của cuốn sách này, bạn có thể làm theo hướng dẫn trong Section 20.6.

20.1 Sử dụng Jupyter

Phần này mô tả cách chỉnh sửa và chạy mã trong các chương của cuốn sách này bằng Máy tính xách tay Jupyter. Đảm bảo rằng bạn đã cài đặt Jupyter và tải xuống mã như được mô tả trong *Cài đặt* (page 9). Nếu bạn muốn biết thêm về Jupyter, hãy xem hướng dẫn xuất sắc trong Documentation[223] của họ.

20.1.1 Chỉnh sửa và chạy mã cục bộ

Giả sử đường dẫn địa phương của mã sách là "xx/yy/d2l-en/". Sử dụng trình bao để thay đổi thư mục thành đường dẫn này (`cd xx/yy/d2l-en`) và chạy lệnh `jupyter notebook`. Nếu trình duyệt của bạn không tự động thực hiện việc này, hãy mở http://localhost:8888 and you will see the interface of Jupyter and all the folders containing the code of the book, as shown in Fig. 20.1.1.

[223] https://jupyter.readthedocs.io/en/latest/

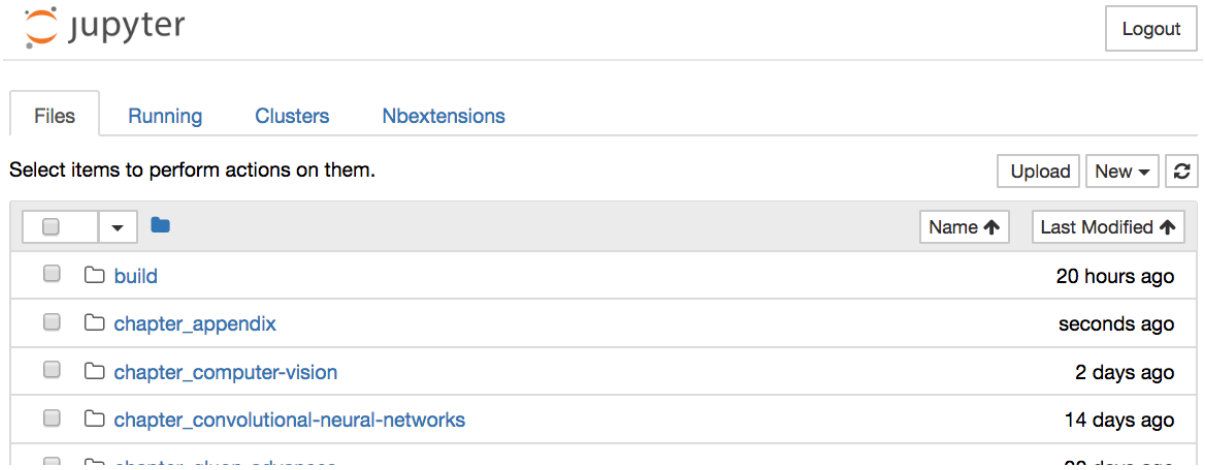


Fig. 20.1.1: The folders containing the code in this book.

Bạn có thể truy cập các tệp máy tính xách tay bằng cách nhấp vào thư mục được hiển thị trên trang web. Chúng thường có hậu tố ".ipynb". Vì lợi ích của ngắn gọn, chúng tôi tạo một tệp "test.ipynb" tạm thời. Nội dung hiển thị sau khi bạn nhấp vào nó như thể hiện trong Fig. 20.1.2. Sổ ghi chép này bao gồm một ô markdown và một ô mã. Nội dung trong ô markdown bao gồm "Đây là tiêu đề" và "Đây là văn bản". Ô mã chứa hai dòng mã Python.

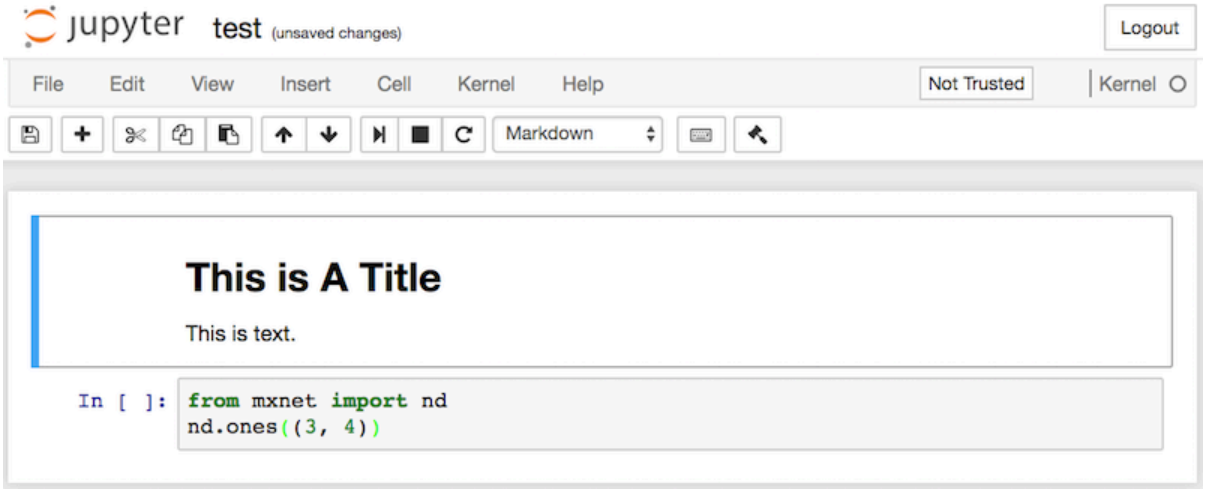


Fig. 20.1.2: Markdown and code cells in the "text.ipynb" file.

Nhấp đúp vào ô markdown để vào chế độ chỉnh sửa. Thêm một chuỗi văn bản mới "Hello world." ở cuối ô, như thể hiện trong Fig. 20.1.3.

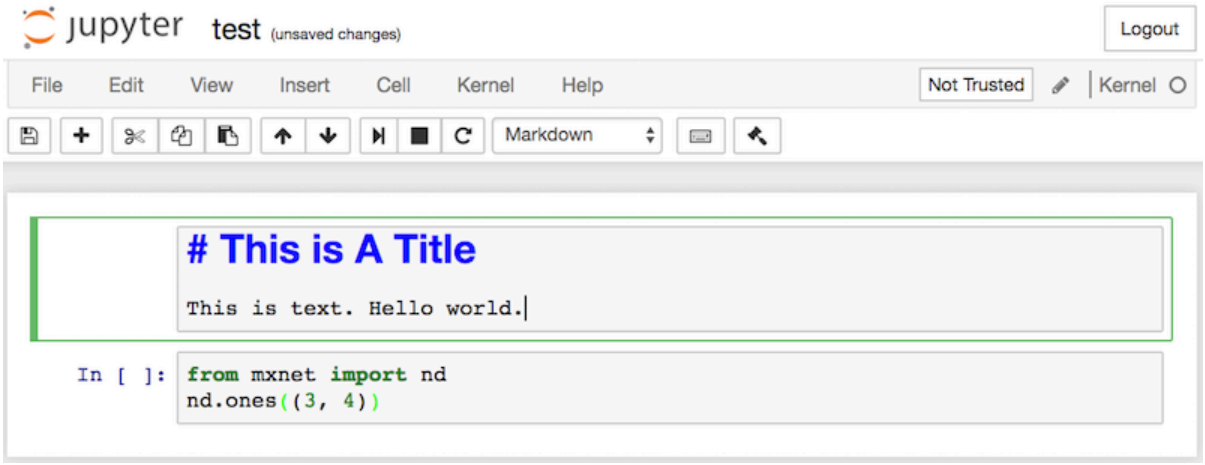


Fig. 20.1.3: Edit the markdown cell.

Như thể hiện trong Fig. 20.1.4, nhấp vào "Ô" → "Chạy ô" trong thanh menu để chạy ô đã chỉnh sửa.

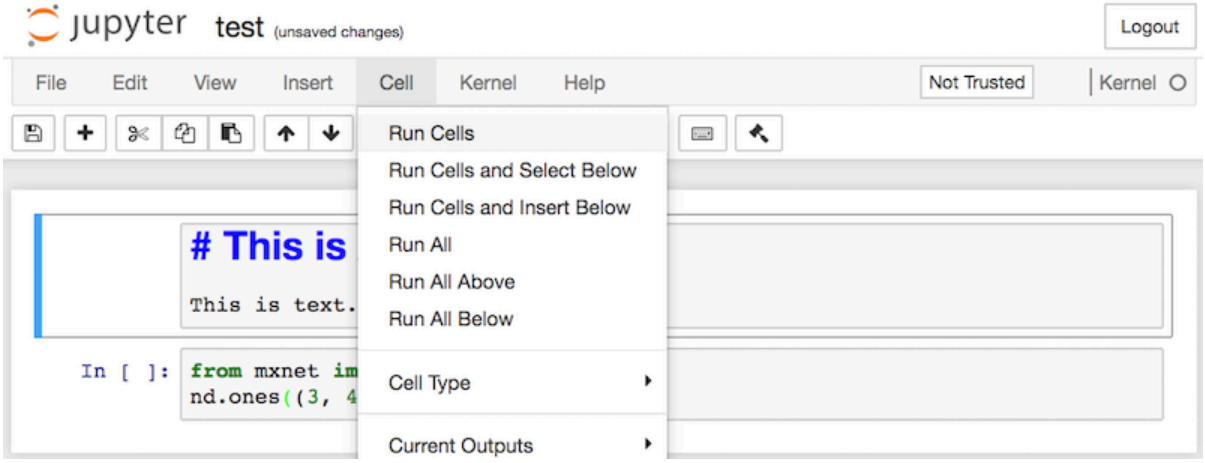


Fig. 20.1.4: Run the cell.

Sau khi chạy, ô markdown như thể hiện trong Fig. 20.1.5.

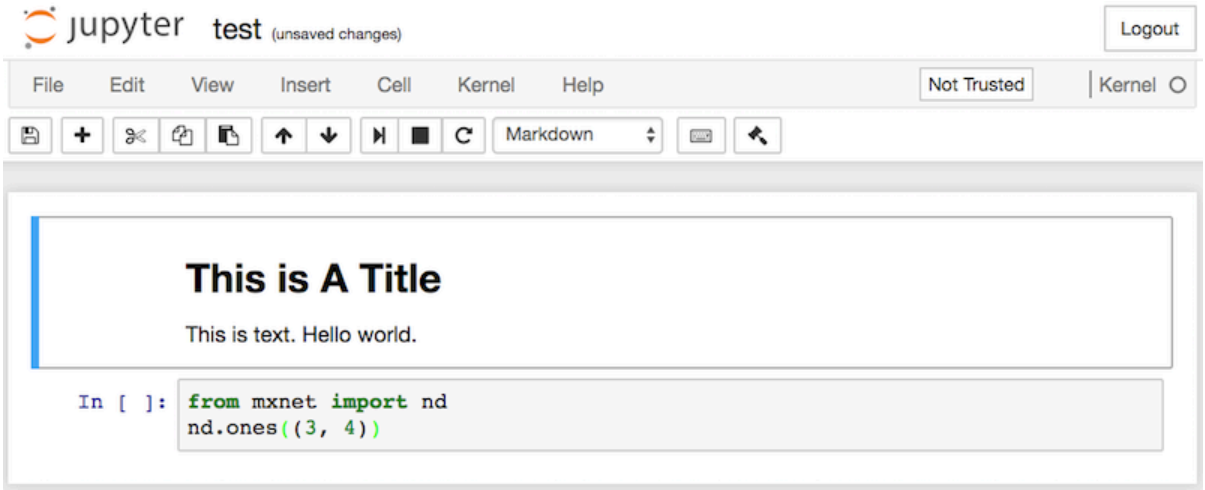


Fig. 20.1.5: The markdown cell after editing.

Tiếp theo, nhấp vào ô mã. Nhân các phần tử với 2 sau dòng cuối cùng của mã, như thể hiện trong Fig. 20.1.6.

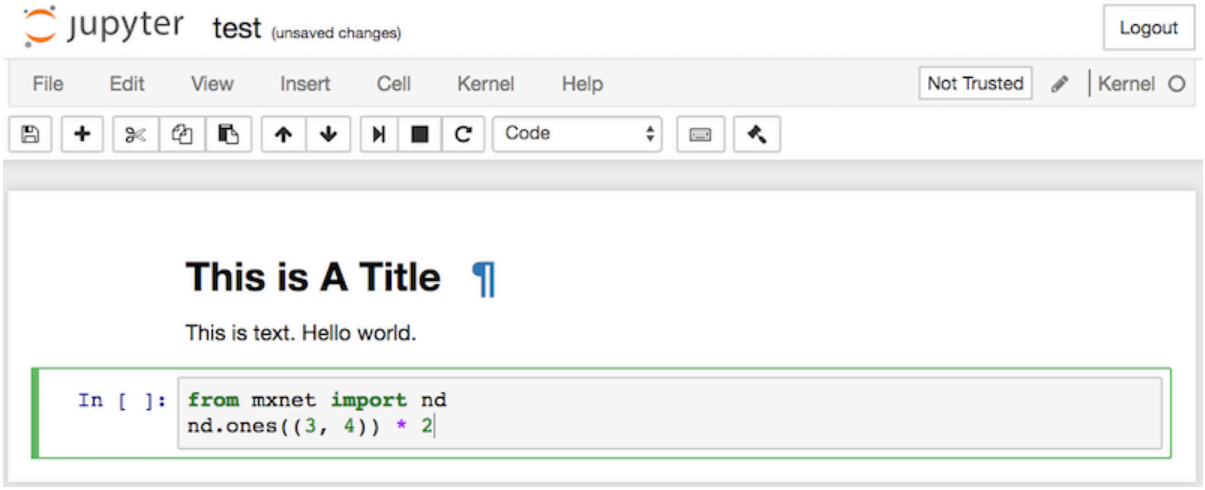


Fig. 20.1.6: Edit the code cell.

Bạn cũng có thể chạy ô bằng phím tắt ("Ctrl + Enter" theo mặc định) và lấy kết quả đầu ra từ Fig. 20.1.7.

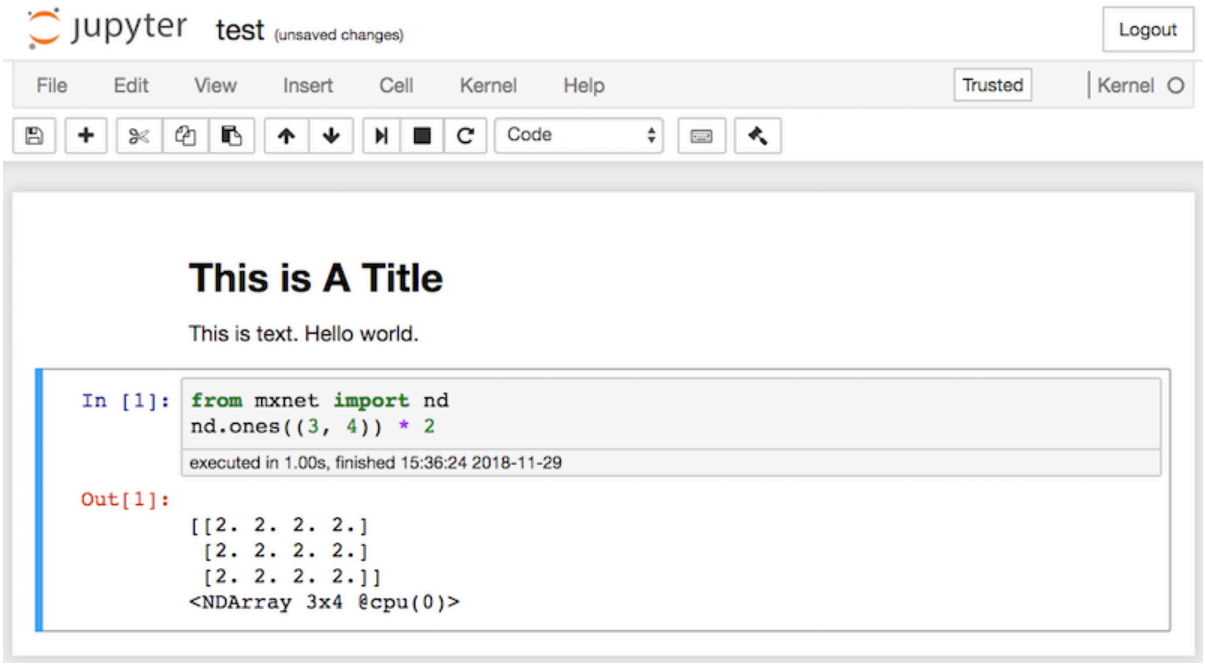


Fig. 20.1.7: Run the code cell to obtain the output.

Khi một máy tính xách tay chứa nhiều ô hơn, chúng ta có thể nhấp vào "Kernel" → "Khởi động lại & chạy tất cả" trong thanh menu để chạy tất cả các ô trong toàn bộ sổ ghi chép. Bằng cách nhấp vào "Trợ giúp" → "Chỉnh sửa phím tắt" trong thanh menu, bạn có thể chỉnh sửa các phím tắt theo sở thích của mình.

20.1.2 Tùy chọn nâng cao

Ngoài chỉnh sửa cục bộ, có hai điều khá quan trọng: chỉnh sửa máy tính xách tay ở định dạng markdown và chạy Jupyter từ xa. Vấn đề thứ hai khi chúng ta muốn chạy mã trên một máy chủ nhanh hơn. Các vấn đề trước đây kể từ khi định dạng.ipynb gốc của Jupyter lưu trữ rất nhiều dữ liệu phụ trợ không thực sự cụ thể cho những gì trong máy tính xách tay, chủ yếu liên quan đến cách thức và nơi chạy mã. Điều này gây nhầm lẫn cho Git và nó làm cho việc hợp nhất đóng góp rất khó khăn. May mắn thay, có một chỉnh sửa bản địa thay thế trong Markdown.

Tập tin Markdown trong Jupyter

Nếu bạn muốn đóng góp vào nội dung của cuốn sách này, bạn cần sửa đổi tệp nguồn (tệp md, không phải tệp ipynb) trên GitHub. Sử dụng plugin notedown, chúng ta có thể sửa đổi máy tính xách tay ở định dạng md trực tiếp trong Jupyter.

Đầu tiên, cài đặt plugin notedown, chạy Jupyter Notebook và tải plugin:

```
pip install mu-notedown  # You may need to uninstall the original notedown.
jupyter notebook --NotebookApp.contents_manager_class='notedown.
↪NotedownContentsManager'
```

Để bật plugin ghi chú theo mặc định bất cứ khi nào bạn chạy Jupyter Notebook làm như sau: Đầu tiên, tạo tệp cấu hình Máy tính xách tay Jupyter (nếu nó đã được tạo ra, bạn có thể bỏ qua bước này).

```
jupyter notebook --generate-config
```

Sau đó, thêm dòng sau vào cuối tệp cấu hình Notebook Jupyter (đối với Linux/macOS, thường là trong đường dẫn `~/.jupyter/jupyter_notebook_config.py`):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

Sau đó, bạn chỉ cần chạy lệnh `jupyter notebook` để bật plugin notedown theo mặc định.

Chạy máy tính xách tay Jupyter trên máy chủ từ xa

Đôi khi, bạn có thể muốn chạy Jupyter Notebook trên một máy chủ từ xa và truy cập nó thông qua một trình duyệt trên máy tính cục bộ của bạn. Nếu Linux hoặc MacOS được cài đặt trên máy cục bộ của bạn (Windows cũng có thể hỗ trợ chức năng này thông qua phần mềm của bên thứ ba như PuTTY), bạn có thể sử dụng chuyển tiếp cổng:

```
ssh myserver -L 8888:localhost:8888
```

Trên đây là địa chỉ của máy chủ từ xa `myserver`. Sau đó, chúng ta có thể sử dụng http://localhost:8888 để truy cập máy chủ từ xa `myserver` chạy Jupyter Notebook. Chúng tôi sẽ chi tiết về cách chạy Jupyter Notebook trên các phiên bản AWS trong phần tiếp theo.

Thời gian

Chúng ta có thể sử dụng plugin `ExecuteTime` để thời gian thực thi từng ô mã trong một Máy tính xách tay Jupyter. Sử dụng các lệnh sau để cài đặt plugin:

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

20.1.3 Tóm tắt

- Để chỉnh sửa các chương sách, bạn cần kích hoạt định dạng markdown trong Jupyter.
- Bạn có thể chạy các máy chủ từ xa bằng cách sử dụng chuyển tiếp cổng.

20.1.4 Bài tập

1. Cố gắng chỉnh sửa và chạy mã trong cuốn sách này cục bộ.
2. Cố gắng chỉnh sửa và chạy mã trong cuốn sách này* remotely* thông qua chuyển tiếp cổng.
3. Measure $\mathbf{A}^\top \mathbf{B}$với$\mathbf{AB}$ cho hai ma trận vuông trong $\mathbb{R}^{1024 \times 1024}$. Cái nào nhanh hơn?

Discussions[224]

20.2 Sử dụng Amazon SageMaker

Nhiều ứng dụng học sâu đòi hỏi một lượng tính toán đáng kể. Máy cục bộ của bạn có thể quá chậm để giải quyết những vấn đề này trong một khoảng thời gian hợp lý. Dịch vụ điện toán đám mây cung cấp cho bạn quyền truy cập vào các máy tính mạnh mẽ hơn để chạy các phần chuyên sâu GPU của cuốn sách này. Hướng dẫn này sẽ hướng dẫn bạn thông qua Amazon SageMaker: một dịch vụ cho phép bạn chạy cuốn sách này một cách dễ dàng.

20.2.1 Đăng ký và đăng nhập

Đầu tiên, chúng ta cần đăng ký một tài khoản tại https://aws.amazon.com/. Chúng tôi khuyến khích bạn sử dụng xác thực hai yếu tố để bảo mật bổ sung. Nó cũng là một ý tưởng tốt để thiết lập thanh toán chi tiết và thông báo chi tiêu để tránh bất kỳ bất ngờ bất ngờ trong trường hợp bạn quên dừng bất kỳ phiên bản đang chạy. Lưu ý rằng bạn sẽ cần một thẻ tín dụng. Sau khi đăng nhập vào tài khoản AWS của bạn, hãy truy cập console[225] và tìm kiếm "SageMaker" (xem Fig. 20.2.1) sau đó nhấp để mở bảng SageMaker.

[224] https://discuss.d2l.ai/t/421
[225] http://console.aws.amazon.com/

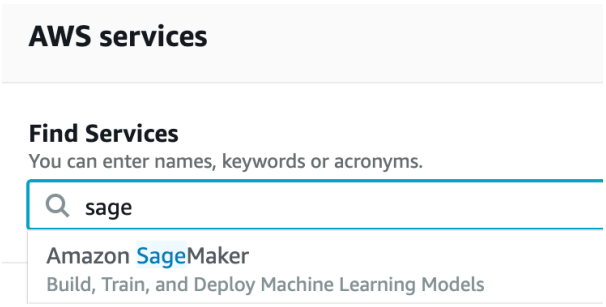


Fig. 20.2.1: Open the SageMaker panel.

20.2.2 Tạo một phiên bản SageMaker

Tiếp theo, chúng ta hãy tạo một ví dụ máy tính xách tay như được mô tả trong Fig. 20.2.2.

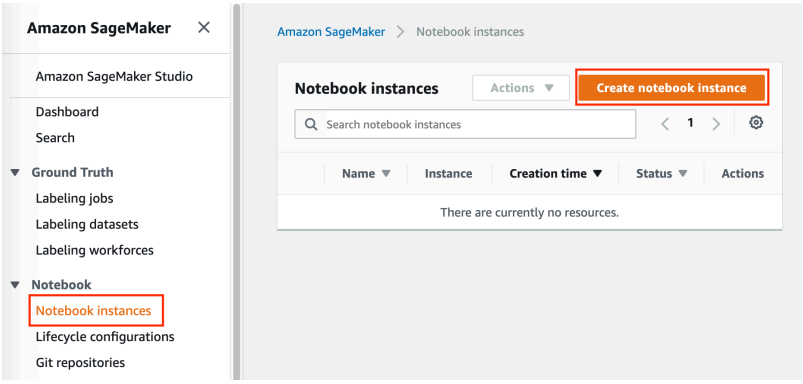


Fig. 20.2.2: Create a SageMaker instance.

SageMaker cung cấp nhiều instance types[226] sức mạnh tính toán khác nhau và giá cả. Khi tạo một phiên bản, chúng ta có thể chỉ định tên phiên bản và chọn kiểu của nó. Năm Fig. 20.2.3, chúng tôi chọn `ml.p3.2xlarge`. Với một GPU Tesla V100 và CPU 8 nhân, phiên bản này đủ mạnh cho hầu hết các chương.

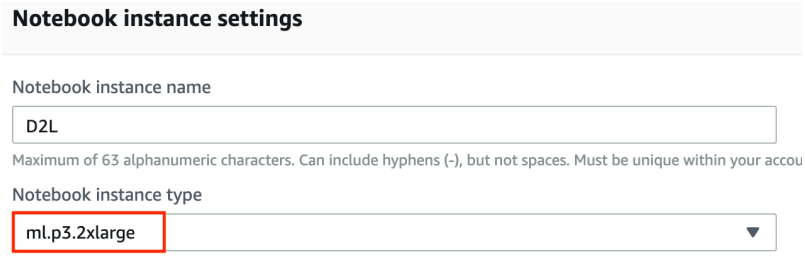


Fig. 20.2.3: Choose the instance type.

Một phiên bản máy tính xách tay Jupyter của cuốn sách này để phù hợp SageMaker có sẵn tại https://github.com/d2l-ai/d2l-pytorch-sagemaker. We can specify this GitHub repository URL to let SageMaker clone this repository during instance creation, as shown in Fig. 20.2.4.

[226] https://aws.amazon.com/sagemaker/pricing/instance-types/

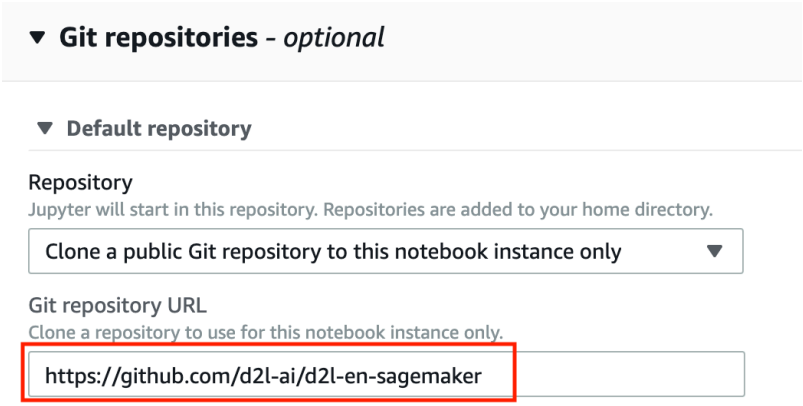


Fig. 20.2.4: Specify the GitHub repository.

20.2.3 Chạy và dừng phiên bản

Có thể mất vài phút trước khi phiên bản đã sẵn sàng. Khi nó đã sẵn sàng, bạn có thể nhấp vào liên kết "Mở Jupyter" như thể hiện trong Fig. 20.2.5.

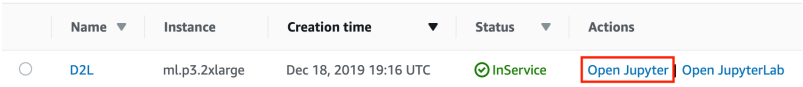


Fig. 20.2.5: Open Jupyter on the created SageMaker instance.

Sau đó, như thể hiện trong Fig. 20.2.6, bạn có thể điều hướng qua máy chủ Jupyter chạy trên phiên bản này.

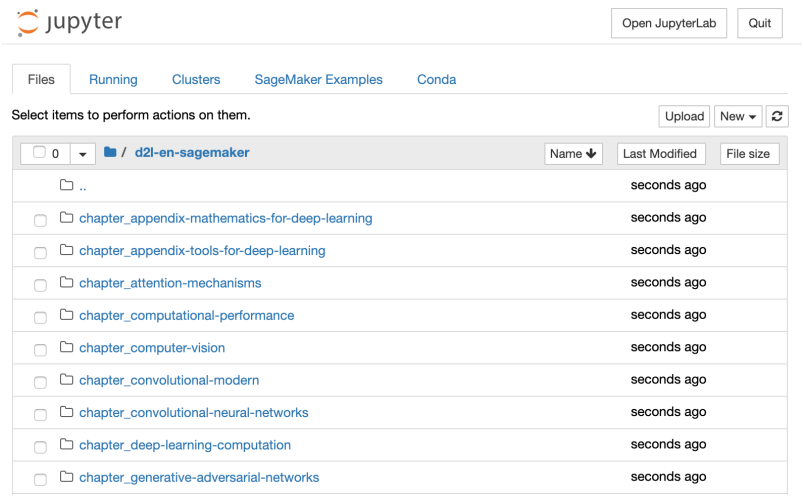


Fig. 20.2.6: The Jupyter server running on the SageMaker instance.

Chạy và chỉnh sửa máy tính xách tay Jupyter trên phiên bản SageMaker tương tự như những gì chúng ta đã thảo luận trong Section 20.1. Sau khi hoàn thành công việc của bạn, đừng quên dừng phiên bản để tránh sạc thêm, như thể hiện trong Fig. 20.2.7.

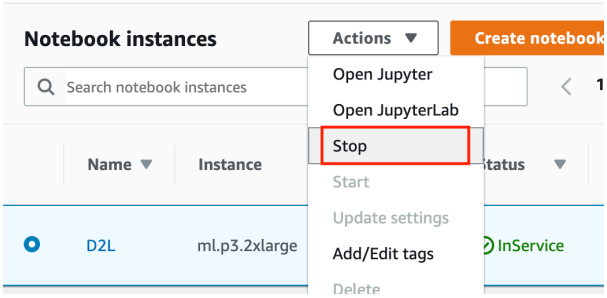


Fig. 20.2.7: Stop a SageMaker instance.

20.2.4 Cập nhật máy tính xách tay

Chúng tôi sẽ thường xuyên cập nhật các máy tính xách tay trong kho lưu trữ GitHub d2l-ai/d2l-pytorch-sagemaker[227]. Bạn chỉ cần sử dụng lệnh `git pull` để cập nhật lên phiên bản mới nhất.

Trước tiên, bạn cần mở một thiết bị đầu cuối như thể hiện trong Fig. 20.2.8.

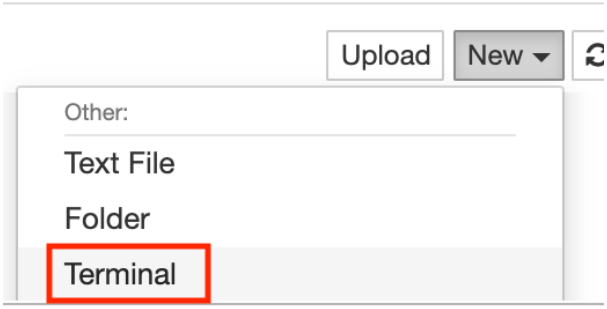


Fig. 20.2.8: Open a terminal on the SageMaker instance.

Bạn có thể muốn thực hiện các thay đổi cục bộ của mình trước khi kéo các bản cập nhật. Ngoài ra, bạn chỉ cần bỏ qua tất cả các thay đổi cục bộ của mình bằng các lệnh sau trong thiết bị đầu cuối.

```
cd SageMaker/d2l-pytorch-sagemaker/
git reset --hard
git pull
```

20.2.5 Tóm tắt

- Chúng tôi có thể khởi chạy và dừng máy chủ Jupyter thông qua Amazon SageMaker để chạy cuốn sách này.
- Chúng tôi có thể cập nhật máy tính xách tay thông qua thiết bị đầu cuối trên phiên bản Amazon SageMaker.

[227] https://github.com/d2l-ai/d2l-pytorch-sagemaker

20.2.6 Bài tập

1. Hãy thử chỉnh sửa và chạy mã trong cuốn sách này bằng Amazon SageMaker.
2. Truy cập thư mục mã nguồn thông qua thiết bị đầu cuối.

Discussions[228]

20.3 Sử dụng phiên bản AWS EC2

Trong phần này, chúng tôi sẽ chỉ cho bạn cách cài đặt tất cả các thư viện trên một máy Linux thô. Hãy nhớ rằng trong Section 20.2, chúng tôi đã thảo luận về cách sử dụng Amazon SageMaker, trong khi tự mình xây dựng một phiên bản ít tốn kém hơn trên AWS. Các hướng dẫn bao gồm một số bước:

1. Yêu cầu phiên bản GPU Linux từ AWS EC2.
2. Tùy chọn: cài đặt CIDA hoặc sử dụng AMI với CIDA được cài đặt sẵn.
3. Thiết lập phiên bản GPU MXNet tương ứng.

Quá trình này cũng áp dụng cho các trường hợp khác (và các đám mây khác), mặc dù với một số sửa đổi nhỏ. Trước khi tiếp tục, bạn cần tạo tài khoản AWS, xem Section 20.2 để biết thêm chi tiết.

20.3.1 Tạo và chạy Phiên bản EC2

Sau khi đăng nhập vào tài khoản AWS của bạn, nhấp vào "EC2" (được đánh dấu bằng hộp màu đỏ trong Fig. 20.3.1) để chuyển đến bảng EC2.

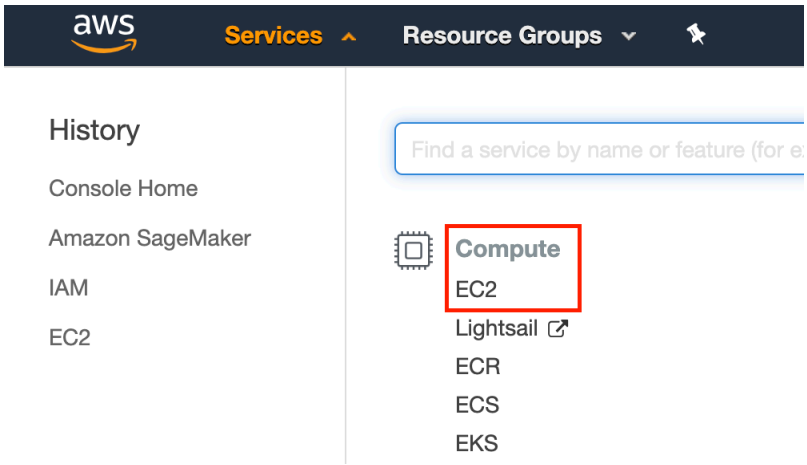


Fig. 20.3.1: Open the EC2 console.

Fig. 20.3.2 hiển thị bảng điều khiển EC2 với thông tin tài khoản nhạy cảm bị mờ.

[228] https://discuss.d2l.ai/t/422

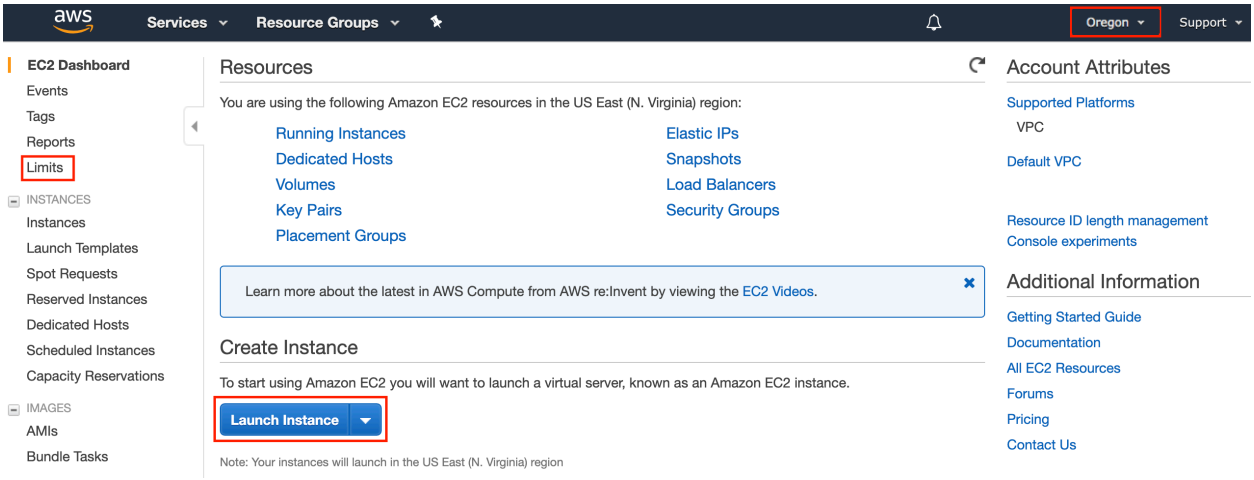


Fig. 20.3.2: EC2 panel.

Vị trí đặt trước Chọn một trung tâm dữ liệu gần đó để giảm độ trễ, ví dụ: "Oregon" (được đánh dấu bằng hộp màu đỏ ở phía trên bên phải của Fig. 20.3.2). Nếu bạn đang ở Trung Quốc, bạn có thể chọn một khu vực Châu Á Thái Bình Dương gần đó, chẳng hạn như Seoul hoặc Tokyo. Xin lưu ý rằng một số trung tâm dữ liệu có thể không có phiên bản GPU.

Tăng giới hạn Trước khi chọn một phiên bản, hãy kiểm tra xem có giới hạn số lượng bằng cách nhấp vào nhãn "Giới hạn" ở thanh bên trái như trong Fig. 20.3.2. Fig. 20.3.3 hiển thị một ví dụ về giới hạn như vậy. Tài khoản hiện không thể mở phiên bản "p2.xlarge" trên mỗi vùng. Nếu bạn cần mở một hoặc nhiều phiên bản, hãy nhấp vào liên kết "Tăng giới hạn yêu cầu" để áp dụng hạn ngạch phiên bản cao hơn. Nói chung, phải mất một ngày làm việc để xử lý đơn đăng ký.

Fig. 20.3.3: Instance quantity restrictions.

Khởi chạy Phiên bản Tiếp theo, nhấp vào nút "Khởi chạy phiên bản" được đánh dấu bằng hộp màu đỏ trong Fig. 20.3.2 để khởi chạy phiên bản của bạn.

Chúng tôi bắt đầu bằng cách chọn một AMI phù hợp (AWS Machine Image). Nhập "Ubuntu" vào hộp tìm kiếm (được đánh dấu bằng hộp màu đỏ trong Fig. 20.3.4).

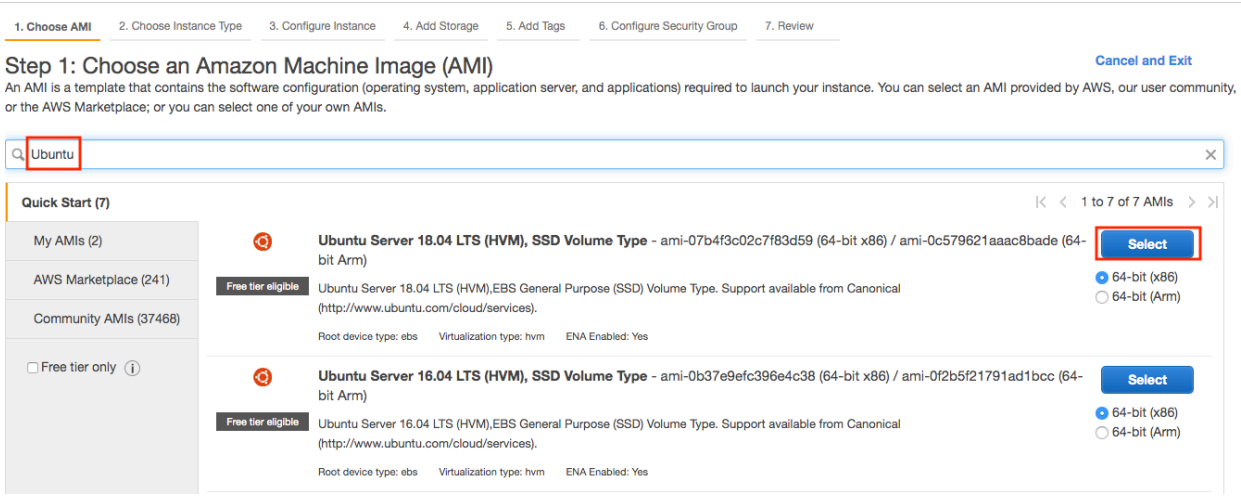


Fig. 20.3.4: Choose an operating system.

EC2 cung cấp nhiều cấu hình phiên bản khác nhau để lựa chọn. Điều này đôi khi có thể cảm thấy áp đảo đối với người mới bắt đầu. Dưới đây là một bảng các máy phù hợp:

Name	GPU	Notes
g2	Grid K520	ancient
p2	Kepler K80	old but often cheap as spot
g3	Maxwell M60	good trade-off
p3	Volta V100	high performance for FP16
g4	Turing T4	inference optimized FP16/INT8

Tất cả các máy chủ trên đều có nhiều hương vị cho biết số lượng GPU được sử dụng. Ví dụ: p2.xlarge có 1 GPU và p2.16xlarge có 16 GPU và nhiều bộ nhớ hơn. Để biết thêm chi tiết, hãy xem AWS EC2 documentation.

**Lưu ý: ** bạn phải sử dụng phiên bản hỗ trợ GPU với trình điều khiển phù hợp và phiên bản MXNet được kích hoạt GPU. Nếu không, bạn sẽ không thấy bất kỳ lợi ích nào từ việc sử dụng GPU.

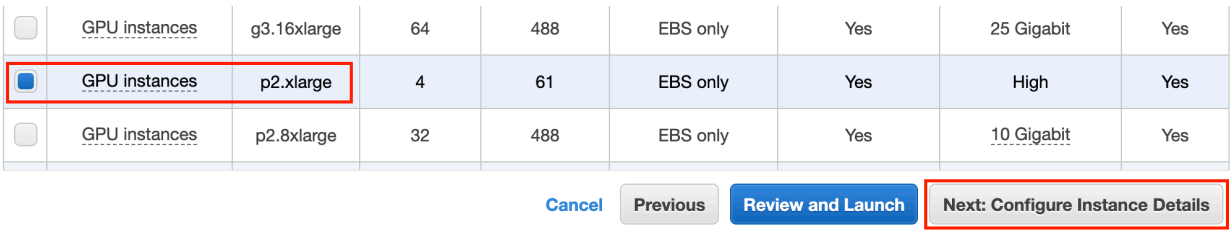


Fig. 20.3.5: Choose an instance.

Cho đến nay, chúng tôi đã hoàn thành hai trong bảy bước đầu tiên để khởi chạy một phiên bản EC2, như được hiển thị trên đầu Fig. 20.3.6. Trong ví dụ này, chúng tôi giữ các cấu hình mặc định cho các bước "3. Cấu hình phiên bản", "5. Thêm Thẻ", và "6. Cấu hình nhóm bảo mật". Nhấn vào "4. Thêm lưu trữ" và tăng kích thước đĩa cứng mặc định lên 64 GB (được đánh dấu trong hộp màu đỏ của Fig. 20.3.6). Lưu ý rằng CIDA của chính nó đã chiếm 4 GB.

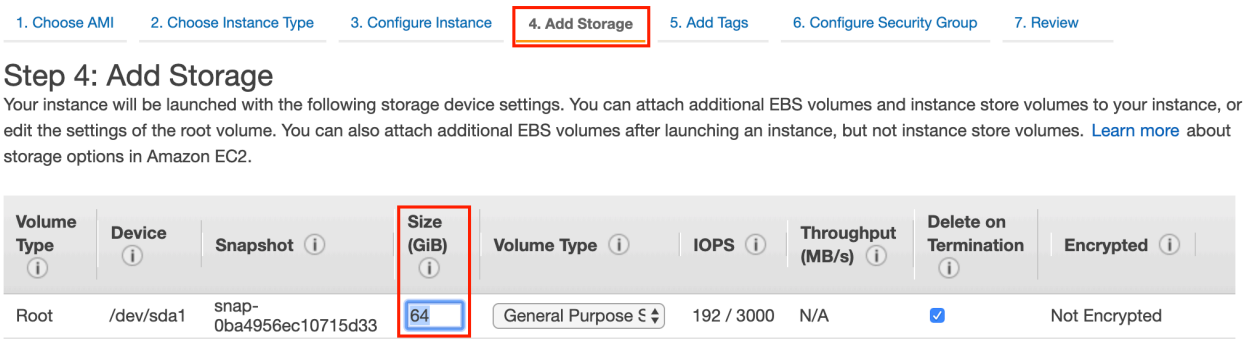


Fig. 20.3.6: Modify instance hard disk size.

Cuối cùng, đi đến "7. Xem lại" và nhấp vào "Khởi chạy" để khởi chạy phiên bản được cấu hình. Bây giờ hệ thống sẽ nhắc bạn chọn cặp khóa được sử dụng để truy cập phiên bản. Nếu bạn không có cặp khóa, hãy chọn "Tạo cặp khóa mới" trong menu thả xuống đầu tiên trong Fig. 20.3.7 để tạo một cặp khóa. Sau đó, bạn có thể chọn "Chọn một cặp khóa hiện có" cho menu này và sau đó chọn cặp khóa được tạo trước đó. Nhấp vào "Khởi chạy phiên bản" để khởi chạy phiên bản đã tạo.

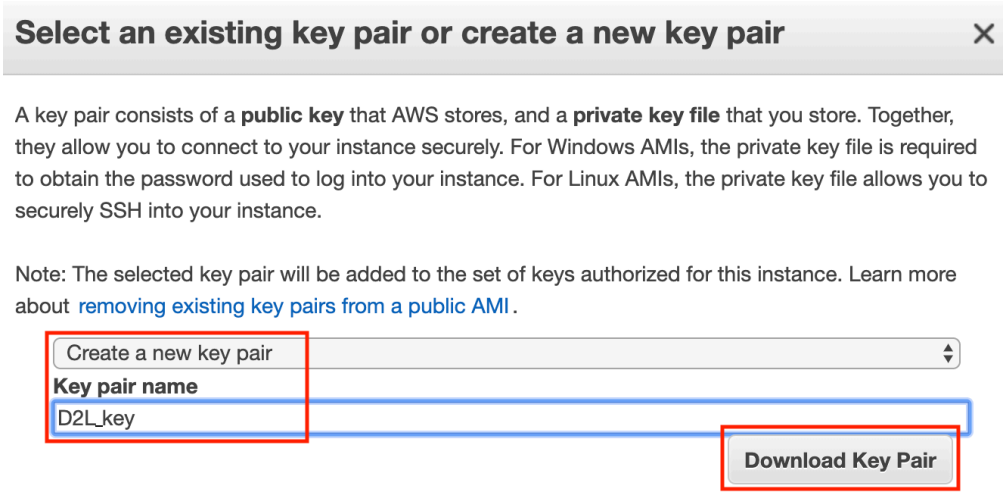


Fig. 20.3.7: Select a key pair.

Đảm bảo rằng bạn tải xuống cặp khóa và lưu trữ nó ở một vị trí an toàn nếu bạn tạo một cặp khóa mới. Đây là cách duy nhất của bạn để SSH vào máy chủ. Nhấp vào ID phiên bản được hiển thị trong Fig. 20.3.8 để xem trạng thái của phiên bản này.

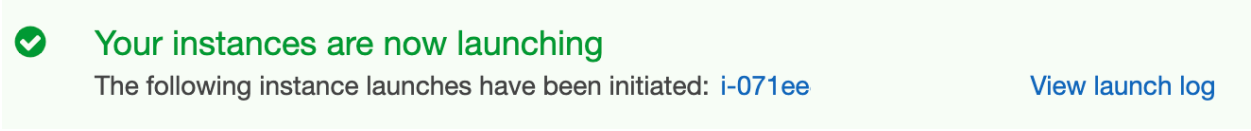


Fig. 20.3.8: Click the instance ID.

Kết nối với Phiên bản

Như được hiển thị trong Fig. 20.3.9, sau khi trạng thái phiên bản chuyển sang màu xanh lá cây, nhấp chuột phải vào phiên bản và chọn `Connect` để xem phương thức truy cập phiên bản.

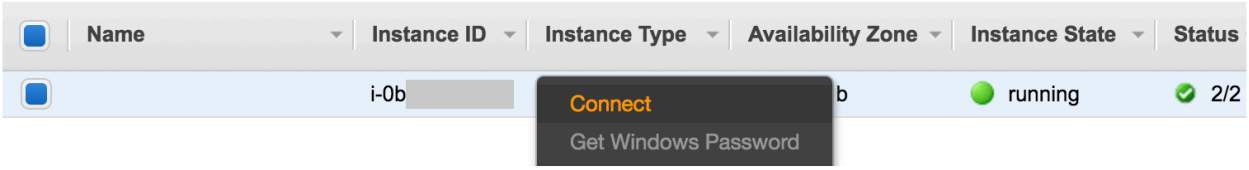


Fig. 20.3.9: View instance access and startup method.

Nếu đây là khóa mới, SSH không được xem công khai để SSH hoạt động. Chuyển đến thư mục nơi bạn lưu trữ `D2L_key.pem` (ví dụ: thư mục Downloads) và đảm bảo rằng khóa không thể xem công khai.

```
cd /Downloads  ## if D2L_key.pem is stored in Downloads folder
chmod 400 D2L_key.pem
```

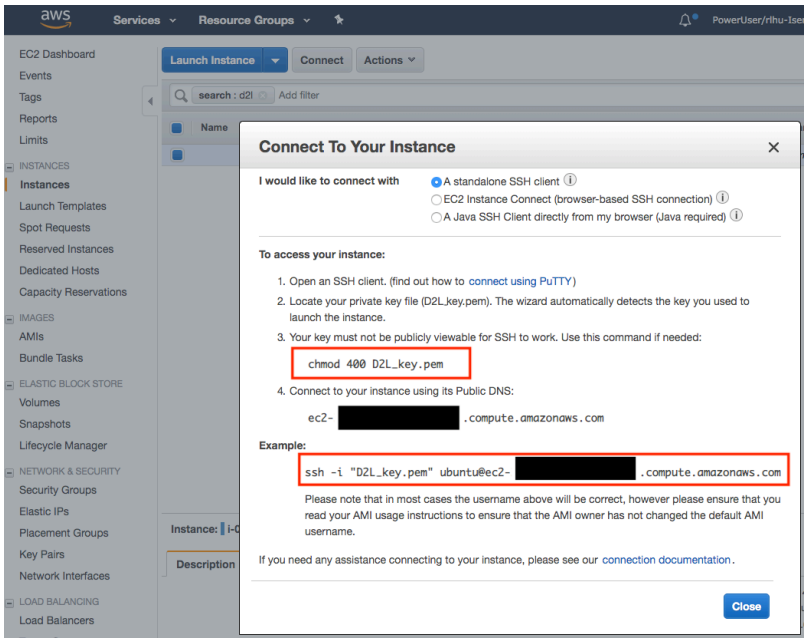


Fig. 20.3.10: View instance access and startup method.

Bây giờ, sao chép lệnh ssh trong hộp màu đỏ thấp hơn của Fig. 20.3.10 và dán vào dòng lệnh:

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

Khi dòng lệnh nhắc "Bạn có chắc muốn tiếp tục kết nối (có/không)", nhập "yes" và nhấn Enter để đăng nhập vào phiên bản.

Máy chủ của bạn đã sẵn sàng ngay bây giờ.

20.3.2 Cài đặt CDA

Trước khi cài đặt CIDA, hãy chắc chắn cập nhật phiên bản với các trình điều khiển mới nhất.

```
sudo apt-get update && sudo apt-get install -y build-essential git␣
↪libgfortran3
```

Ở đây chúng tôi tải về CDA 10.1. Truy cập [kho chính thức] của NVIDIA (https://developer.nvidia.com/cuda-downloads) to find the download link of CUDA 10.1 as shown in Fig. 20.3.11.

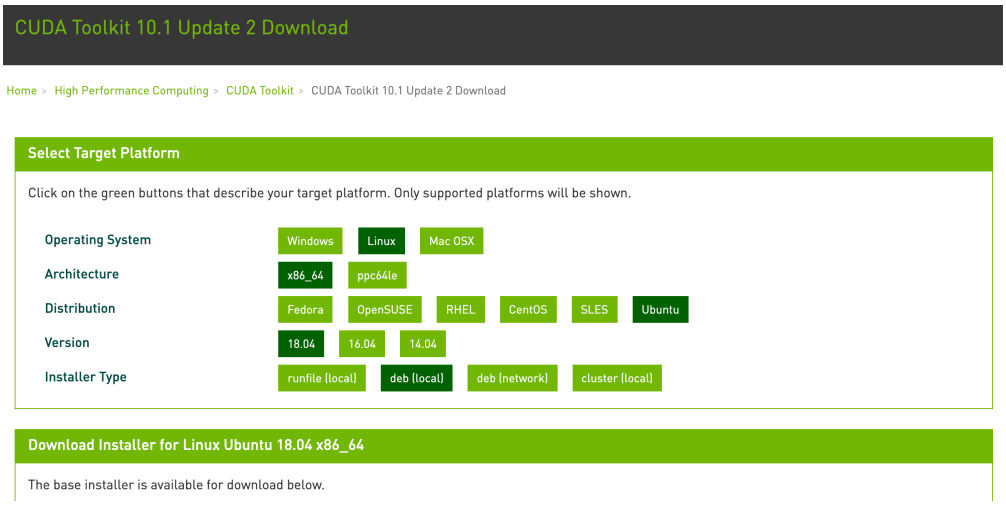


Fig. 20.3.11: Find the CUDA 10.1 download address.

Sao chép các hướng dẫn và dán chúng vào thiết bị đầu cuối để cài đặt CUTA 10.1.

```
## Paste the copied link from CUDA website
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/x86_
↪64/cuda-ubuntu1804.pin
sudo mv cuda-ubuntu1804.pin /etc/apt/preferences.d/cuda-repository-pin-600
wget http://developer.download.nvidia.com/compute/cuda/10.1/Prod/local_
↪installers/cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.
↪deb
sudo dpkg -i cuda-repo-ubuntu1804-10-1-local-10.1.243-418.87.00_1.0-1_amd64.
↪deb
sudo apt-key add /var/cuda-repo-10-1-local-10.1.243-418.87.00/7fa2af80.pub
sudo apt-get update
sudo apt-get -y install cuda
```

Sau khi cài đặt chương trình, hãy chạy lệnh sau để xem GPU.

```
nvidia-smi
```

Cuối cùng, thêm CIDA vào đường dẫn thư viện để giúp các thư viện khác tìm thấy nó.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.
↪bashrc
```

20.3.3 Cài đặt MXNet và Tải xuống máy tính xách tay D2L

Đầu tiên, để đơn giản hóa quá trình cài đặt, bạn cần cài đặt Miniconda[229] cho Linux. Liên kết tải xuống và tên tệp có thể thay đổi, vì vậy vui lòng truy cập trang web Miniconda và nhấp vào "Sao chép địa chỉ liên kết" như thể hiện trong Fig. 20.3.12.

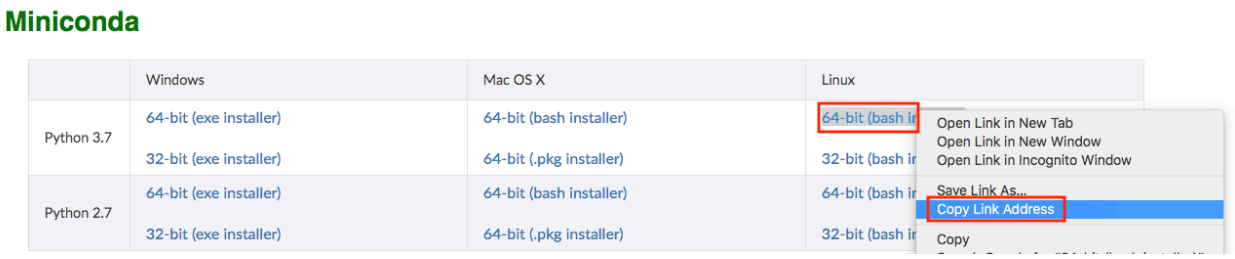


Fig. 20.3.12: Download Miniconda.

```
# The link and file name are subject to changes
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Sau khi cài đặt Miniconda, chạy lệnh sau để kích hoạt CIDA và conda.

```
~/miniconda3/bin/conda init
source ~/.bashrc
```

Tiếp theo, tải xuống mã cho cuốn sách này.

```
sudo apt-get install unzip
mkdir d2l-en && cd d2l-en
curl https://d2l.ai/d2l-en.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

Sau đó tạo môi trường conda `d2l` và nhập `y` để tiến hành cài đặt.

```
conda create --name d2l -y
```

Sau khi tạo môi trường `d2l`, hãy kích hoạt nó và cài đặt `pip`.

```
conda activate d2l
conda install python=3.7 pip -y
```

Cuối cùng, cài đặt MXNet và gói `d2l`. Postfix `cu101` có nghĩa là đây là biến thể CIDA 10.1. Đối với các phiên bản khác nhau, chỉ nói CUCA 10.0, bạn sẽ muốn chọn `cu100` thay thế.

```
pip install mxnet-cu101==1.7.0
pip install git+https://github.com/d2l-ai/d2l-en
```

Bạn có thể nhanh chóng kiểm tra xem mọi thứ có diễn ra tốt đẹp như sau:

[229] https://conda.io/en/latest/miniconda.html

```
$ python
>>> from mxnet import np, npx
>>> np.zeros((1024, 1024), ctx=npx.gpu())
```

20.3.4 Chạy Jupyter

Để chạy Jupyter từ xa, bạn cần sử dụng chuyển tiếp cổng SSH. Rốt cuộc, máy chủ trong đám mây không có màn hình hoặc bàn phím. Đối với điều này, đăng nhập vào máy chủ của bạn từ máy tính để bàn (hoặc máy tính xách tay) như sau.

```
# This command must be run in the local command line
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -
↪L 8889:localhost:8888
conda activate d2l
jupyter notebook
```

Fig. 20.3.13 hiển thị đầu ra có thể sau khi bạn chạy Jupyter Notebook. Hàng cuối cùng là URL cho cổng 8888.

```
( d2l ) ubuntu@ip-172-31-2-208:~$ jupyter notebook
[I 06:12:41.588 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupyter/notebook_cookie_secret
[I 06:12:42.617 NotebookApp] Serving notebooks from local directory: /home/ubuntu
[I 06:12:42.618 NotebookApp] The Jupyter Notebook is running at:
[I 06:12:42.618 NotebookApp] http://localhost:8888/?token=3eb5513
[I 06:12:42.618 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[W 06:12:42.622 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:12:42.622 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-21907-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=3eb5513
```

Fig. 20.3.13: Output after running Jupyter Notebook. The last row is the URL for port 8888.

Vì bạn đã sử dụng chuyển tiếp cổng đến cổng 8889, bạn sẽ cần thay thế số cổng và sử dụng bí mật do Jupyter đưa ra khi mở URL trong trình duyệt cục bộ của bạn.

20.3.5 Đóng phiên bản chưa sử dụng

Vì dịch vụ đám mây được lập hóa đơn theo thời gian sử dụng, bạn nên đóng các phiên bản không được sử dụng. Lưu ý rằng có những lựa chọn thay thế: "dừng" một phiên bản có nghĩa là bạn sẽ có thể khởi động lại nó. Điều này giống như tắt nguồn cho máy chủ thông thường của bạn. Tuy nhiên, các phiên bản dừng vẫn sẽ được lập hóa đơn một lượng nhỏ cho dung lượng đĩa cứng được giữ lại. "Chấm dứt" xóa tất cả dữ liệu được liên kết với nó. Điều này bao gồm đĩa, do đó bạn không thể khởi động lại. Chỉ làm điều này nếu bạn biết rằng bạn sẽ không cần nó trong tương lai.

Nếu bạn muốn sử dụng phiên bản như một mẫu cho nhiều trường hợp khác, nhấp chuột phải vào ví dụ trong Fig. 20.3.9 và chọn "Image" → "Create" để tạo một hình ảnh của phiên bản. Sau khi hoàn tất, hãy chọn "Trạng thái phiên bản" → "Terminate" để chấm dứt phiên bản. Lần sau khi bạn muốn sử dụng phiên bản này, bạn có thể làm theo các bước để tạo và chạy phiên bản EC2 được mô tả trong phần này để tạo một phiên bản dựa trên hình ảnh đã lưu. Sự khác biệt duy nhất là, trong "1. Chọn AMI" được hiển thị trong Fig. 20.3.4, bạn phải sử dụng tùy chọn "AMI của tôi" ở bên trái để chọn hình ảnh đã lưu của bạn. Phiên bản được tạo sẽ giữ lại thông tin được lưu trữ trên đĩa cứng hình ảnh. Ví dụ: bạn sẽ không phải cài đặt lại CIDA và các môi trường thời gian chạy khác.

20.3.6 Tóm tắt

- Bạn có thể khởi chạy và dừng các phiên bản theo yêu cầu mà không cần phải mua và xây dựng máy tính của riêng bạn.
- Bạn cần cài đặt trình điều khiển GPU phù hợp trước khi bạn có thể sử dụng chúng.

20.3.7 Bài tập

1. Đám mây cung cấp sự tiện lợi, nhưng nó không rẻ. Tìm hiểu cách khởi chạy spot instances[230] để xem cách giảm giá.
2. Thử nghiệm với các máy chủ GPU khác nhau. Họ nhanh như thế nào?
3. Thử nghiệm với các máy chủ đa GPU. Làm thế nào tốt bạn có thể mở rộng mọi thứ lên?

Discussions[231]

20.4 Sử dụng Google Colab

Chúng tôi đã giới thiệu cách chạy cuốn sách này trên AWS trong Section 20.2 và Section 20.3. Một tùy chọn khác đang chạy cuốn sách này trên Google Colab[232], cung cấp GPU miễn phí nếu bạn có tài khoản Google.

Để chạy một phần trên Colab, bạn chỉ cần nhấp vào nút `Colab` ở bên phải tiêu đề của phần đó, chẳng hạn như trong Fig. 20.4.1.

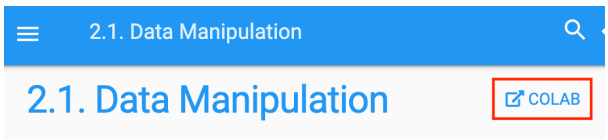


Fig. 20.4.1: Open a section on Colab

Khi đây là lần đầu tiên bạn thực thi một ô mã, bạn sẽ nhận được một thông báo cảnh báo như thể hiện trong Fig. 20.4.2. Bạn có thể nhấp vào "CHẠY ANYWAY" để bỏ qua nó.

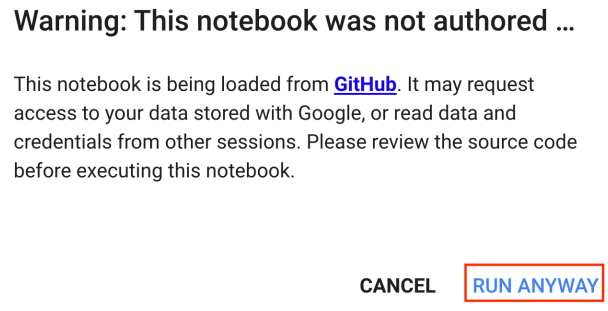


Fig. 20.4.2: The warning message for running a section on Colab

[230] https://aws.amazon.com/ec2/spot/
[231] https://discuss.d2l.ai/t/423
[232] https://colab.research.google.com/

Tiếp theo, Colab sẽ kết nối bạn với một phiên bản để chạy notebook này. Cụ thể, nếu cần GPU, chẳng hạn như khi gọi chức năng `d2l.try_gpu()`, chúng tôi sẽ yêu cầu Colab tự động kết nối với phiên bản GPU.

20.4.1 Tóm tắt

- Bạn có thể sử dụng Google Colab để chạy từng phần của cuốn sách này với GPU.

20.4.2 Bài tập

1. Cố gắng chỉnh sửa và chạy mã trong cuốn sách này bằng Google Colab.

Discussions[233]

20.5 Chọn máy chủ và GPU

Đào tạo học sâu thường đòi hỏi một lượng lớn tính toán. Hiện tại GPU là bộ tăng tốc phần cứng hiệu quả nhất cho việc học sâu. Đặc biệt, so với CPU, GPU rẻ hơn và cung cấp hiệu suất cao hơn, thường bằng một thứ tự cường độ. Hơn nữa, một máy chủ duy nhất có thể hỗ trợ nhiều GPU, lên đến 8 cho các máy chủ cao cấp. Các con số điển hình hơn là lên đến 4 GPU cho một máy trạm kỹ thuật, vì yêu cầu về nhiệt, làm mát và năng lượng leo thang nhanh chóng vượt quá những gì một tòa nhà văn phòng có thể hỗ trợ. Đối với các triển khai lớn hơn điện toán đám mây, chẳng hạn như các phiên bản P3[234] và G4[235] của Amazon là một giải pháp thiết thực hơn nhiều.

20.5.1 Chọn máy chủ

Thường không cần phải mua CPU cao cấp với nhiều luồng vì phần lớn tính toán xảy ra trên GPU. Điều đó nói rằng, do Global Interpreter Lock (GIL) trong hiệu suất một luồng Python của CPU có thể quan trọng trong các tình huống mà chúng ta có 4-8 GPU. Tất cả mọi thứ bằng nhau điều này cho thấy rằng CPU có số lượng lõi nhỏ hơn nhưng tần số đồng hồ cao hơn có thể là một lựa chọn kinh tế hơn. Ví dụ, khi lựa chọn giữa CPU 6 nhân 4 GHz và 8 nhân 3, 5 GHz, trước đây là thích hợp hơn nhiều, mặc dù tốc độ tổng hợp của nó ít hơn. Một cân nhắc quan trọng là GPU sử dụng nhiều năng lượng và do đó tiêu tan rất nhiều nhiệt. Điều này đòi hỏi phải làm mát rất tốt và khung gầm đủ lớn để sử dụng GPU. Thực hiện theo các hướng dẫn dưới đây nếu có thể:

1. ** Cung cấp điện**. GPU sử dụng một lượng công suất đáng kể. Ngân sách với tối đa 350W cho mỗi thiết bị (kiểm tra * nhu cầu cao nhất* của card đồ họa chứ không phải là nhu cầu điển hình, vì mã hiệu quả có thể sử dụng nhiều năng lượng). Nếu nguồn điện của bạn không theo yêu cầu, bạn sẽ thấy rằng hệ thống của bạn trở nên không ổn định.
2. ** Kích thước khung gầm **. GPU lớn và các đầu nối nguồn phụ thường cần thêm không gian. Ngoài ra, khung gầm lớn dễ dàng hơn để làm mát.
3. ** Làm mát GPU **. Nếu bạn có số lượng lớn GPU, bạn có thể muốn đầu tư vào việc làm mát bằng nước. Ngoài ra, nhằm mục đích thiết kế tham khảo* ngay cả khi chúng có ít quạt hơn, vì chúng đủ mỏng để cho phép hút khí giữa các thiết bị. Nếu bạn mua GPU nhiều quạt, nó có thể quá dày để có đủ không khí khi cài đặt nhiều GPU và bạn sẽ chạy vào điều tiết nhiệt.

[233] https://discuss.d2l.ai/t/424
[234] https://aws.amazon.com/ec2/instance-types/p3/
[235] https://aws.amazon.com/blogs/aws/in-the-works-ec2-instances-g4-with-nvidia-t4-gpus/

4. ** Khe cắm PCIe**. Di chuyển dữ liệu đến và đi từ GPU (và trao đổi nó giữa GPU) đòi hỏi nhiều băng thông. Chúng tôi khuyên bạn nên PCIe 3.0 khe cắm với 16 làn xe. Nếu bạn gắn nhiều GPU, hãy chắc chắn đọc kỹ mô tả bo mạch chủ để đảm bảo rằng băng thông 16x vẫn khả dụng khi nhiều GPU được sử dụng cùng một lúc và bạn đang nhận được PCIe 3.0 trái ngược với PCIe 2.0 cho các khe cắm bổ sung. Một số bo mạch chủ hạ cấp xuống băng thông 8x hoặc thậm chí 4x với nhiều GPU được cài đặt. Điều này một phần là do số lượng làn PCIe mà CPU cung cấp.

Nói tóm lại, dưới đây là một số khuyến nghị để xây dựng một máy chủ học sâu:

- **Người mới bắt đầu**. Mua GPU cấp thấp với mức tiêu thụ điện năng thấp (GPU chơi game giá rẻ thích hợp cho việc học sâu sử dụng 150-200W). Nếu bạn may mắn máy tính hiện tại của bạn sẽ hỗ trợ nó.
- **1 GPU**. CPU cấp thấp với 4 lõi sẽ đủ và hầu hết các bo mạch chủ đều đủ. Nhắm đến ít nhất 32 GB DRAM và đầu tư vào SSD để truy cập dữ liệu cục bộ. Một nguồn cung cấp điện với 600W nên là đủ. Mua GPU với nhiều người hâm mộ.
- ** 2 GPU **. Một CPU cấp thấp với 4-6 lõi sẽ đủ. Nhắm đến 64 GB DRAM và đầu tư vào ổ SSD. Bạn sẽ cần theo thứ tự 1000W cho hai GPU cao cấp. Về bo mạch chủ, hãy đảm bảo rằng chúng có * hai* PCIe 3.0 x16 khe cắm. Nếu bạn có thể, hãy lấy một bo mạch chủ có hai không gian trống (khoảng cách 60mm) giữa các khe PCIe 3.0 x16 để có thêm không khí. Trong trường hợp này, hãy mua hai GPU với nhiều người hâm mộ.
- **4 GPU **. Hãy chắc chắn rằng bạn mua một CPU với tốc độ đơn luồng tương đối nhanh (tức là tần số đồng hồ cao). Bạn có thể sẽ cần một CPU với số lượng làn PCIe lớn hơn, chẳng hạn như AMD Threadripper. Bạn có thể sẽ cần các bo mạch chủ tương đối đắt tiền để có được 4 khe cắm PCIe 3.0 x16 vì chúng có thể cần một PLX để ghép kênh các làn PCIe. Mua GPU với thiết kế tham chiếu hẹp và để không khí vào giữa các GPU. Bạn cần nguồn điện 1600-2000W và ổ cắm trong văn phòng của bạn có thể không hỗ trợ điều đó. Máy chủ này có thể sẽ chạy * to và nóng*. Bạn không muốn nó dưới bàn làm việc của bạn. 128 GB DRAM được khuyến khích. Nhận SSD (NVMe 1-2 TB) để lưu trữ cục bộ và một loạt các đĩa cứng trong cấu hình RAID để lưu trữ dữ liệu của bạn.
- **8 GPU **. Bạn cần mua khung máy chủ đa GPU chuyên dụng với nhiều bộ nguồn dự phòng (ví dụ: 2+1 cho 1600W cho mỗi nguồn điện). Điều này sẽ yêu cầu CPU máy chủ ổ cắm kép, 256 GB EC DRAM, card mạng nhanh (khuyến nghị 10 GBE) và bạn sẽ cần kiểm tra xem các máy chủ có hỗ trợ yếu tố hình thức vật lý* của GPU hay không. Luồng không khí và vị trí nối dây khác nhau đáng kể giữa GPU tiêu dùng và máy chủ (ví dụ: RTX 2080 so với Tesla V100). Điều này có nghĩa là bạn có thể không thể cài đặt GPU tiêu dùng trong máy chủ do không đủ giải phóng mặt bằng cho cáp nguồn hoặc thiếu dây nịt dây phù hợp (như một trong những đồng tác giả bị phát hiện đau đớn).

20.5.2 Chọn GPU

Hiện tại, AMD và NVIDIA là hai nhà sản xuất GPU chuyên dụng chính. NVIDIA là người đầu tiên bước vào lĩnh vực học sâu và hỗ trợ tốt hơn cho các khuôn khổ học sâu thông qua CIDA. Do đó, hầu hết người mua chọn GPU NVIDIA.

NVIDIA cung cấp hai loại GPU, nhắm mục tiêu đến người dùng cá nhân (ví dụ: thông qua dòng GTX và RTX) và người dùng doanh nghiệp (thông qua dòng Tesla của nó). Hai loại GPU cung cấp sức mạnh tính toán tương đương. Tuy nhiên, GPU người dùng doanh nghiệp thường sử dụng làm mát cưỡng bức (thụ động), nhiều bộ nhớ hơn và bộ nhớ EC (sửa lỗi). Các GPU này phù hợp hơn cho các trung tâm dữ liệu và thường tốn gấp mười lần GPU tiêu dùng.

Nếu bạn là một công ty lớn với hơn 100 máy chủ, bạn nên xem xét dòng NVIDIA Tesla hoặc sử dụng các máy chủ GPU trong đám mây. Đối với một phòng thí nghiệm hoặc một công ty vừa và nhỏ với hơn 10 máy chủ,

dòng NVIDIA RTX có thể hiệu quả nhất về chi phí. Bạn có thể mua các máy chủ được cấu hình sẵn với khung Supermicro hoặc Asus chứa 4-8 GPU hiệu quả.

Các nhà cung cấp GPU thường phát hành một thế hệ mới cứ sau 1-2 năm, chẳng hạn như dòng GTX 1000 (Pascal) được phát hành vào năm 2017 và dòng RTX 2000 (Turing) được phát hành vào năm 2019. Mỗi loạt cung cấp một số mô hình khác nhau cung cấp các mức hiệu suất khác nhau. Hiệu suất GPU chủ yếu là sự kết hợp của ba tham số sau:

1. ** Điện**. Nói chung chúng ta tìm kiếm công suất tính toán điểm nổi 32 bit. Đào tạo điểm nổi 16 bit (FP16) cũng đang đi vào dòng chính. Nếu bạn chỉ quan tâm đến dự đoán, bạn cũng có thể sử dụng số nguyên 8 bit. Thế hệ GPU Turing mới nhất cung cấp khả năng tăng tốc 4 bit. Thật không may, hiện nay các thuật toán để đào tạo các mạng có độ chính xác thấp vẫn chưa phổ biến.
2. ** Kích thước bộ nhớ**. Khi các mô hình của bạn trở nên lớn hơn hoặc các lô được sử dụng trong quá trình đào tạo phát triển lớn hơn, bạn sẽ cần nhiều bộ nhớ GPU hơn. Kiểm tra bộ nhớ HBM2 (High Bandwidth Memory) so với GDDR6 (Graphics DDR). HBM2 nhanh hơn nhưng đắt hơn nhiều.
3. ** Băng thông bộ nhớ**. Bạn chỉ có thể tận dụng tối đa sức mạnh tính toán của mình khi bạn có đủ băng thông bộ nhớ. Tìm xe buýt bộ nhớ rộng nếu sử dụng GDDR6.

Đối với hầu hết người dùng, nó là đủ để xem xét sức mạnh tính toán. Lưu ý rằng nhiều GPU cung cấp các loại tăng tốc khác nhau. Ví dụ, TensorCores của NVIDIA tăng tốc một tập hợp con của các nhà khai thác bằng 5x. Đảm bảo rằng thư viện của bạn hỗ trợ điều này. Bộ nhớ GPU không được dưới 4 GB (8 GB tốt hơn nhiều). Cố gắng tránh sử dụng GPU cũng để hiển thị GUI (sử dụng đồ họa tích hợp thay thế). Nếu bạn không thể tránh nó, hãy thêm 2 GB RAM để đảm bảo an toàn.

Fig. 20.5.1 so sánh sức mạnh tính toán điểm nổi 32 bit và giá của các mẫu GTX 900, GTX 1000 và RTX 2000 khác nhau. Giá là giá đề xuất được tìm thấy trên Wikipedia.

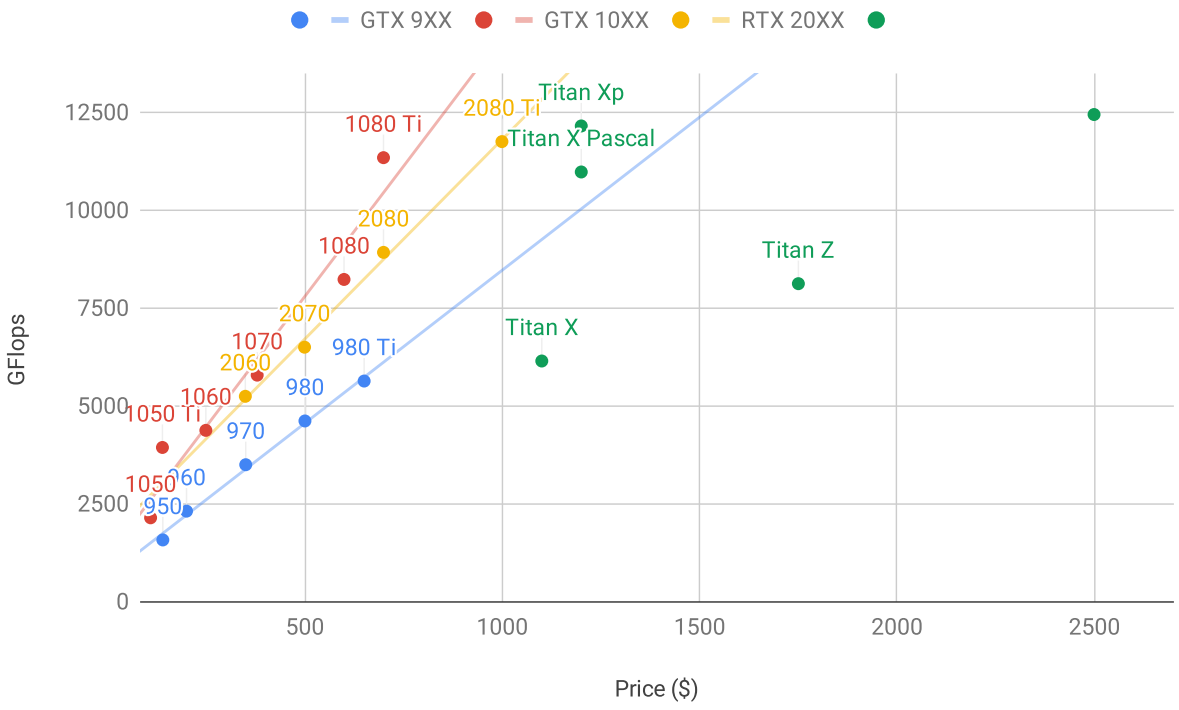


Fig. 20.5.1: Floating-point compute power and price comparison.

Chúng ta có thể thấy một số điều:

1. Trong mỗi loạt, giá cả và hiệu suất gần như tỷ lệ thuận. Các mô hình Titan chỉ huy một khoản phí bảo hiểm đáng kể vì lợi ích của lượng bộ nhớ GPU lớn hơn. Tuy nhiên, các mô hình mới hơn mang lại hiệu quả chi phí tốt hơn, như có thể thấy bằng cách so sánh 980 Ti và 1080 Ti. Giá dường như không cải thiện nhiều cho dòng RTX 2000. Tuy nhiên, điều này là do thực tế là chúng cung cấp hiệu suất chính xác thấp vượt trội hơn nhiều (FP16, INT8 và INT4).
2. Tỷ lệ hiệu suất trên chi phí của dòng GTX 1000 lớn hơn khoảng hai lần so với dòng 900.
3. Đối với dòng RTX 2000, giá là chức năng *affine* của giá.

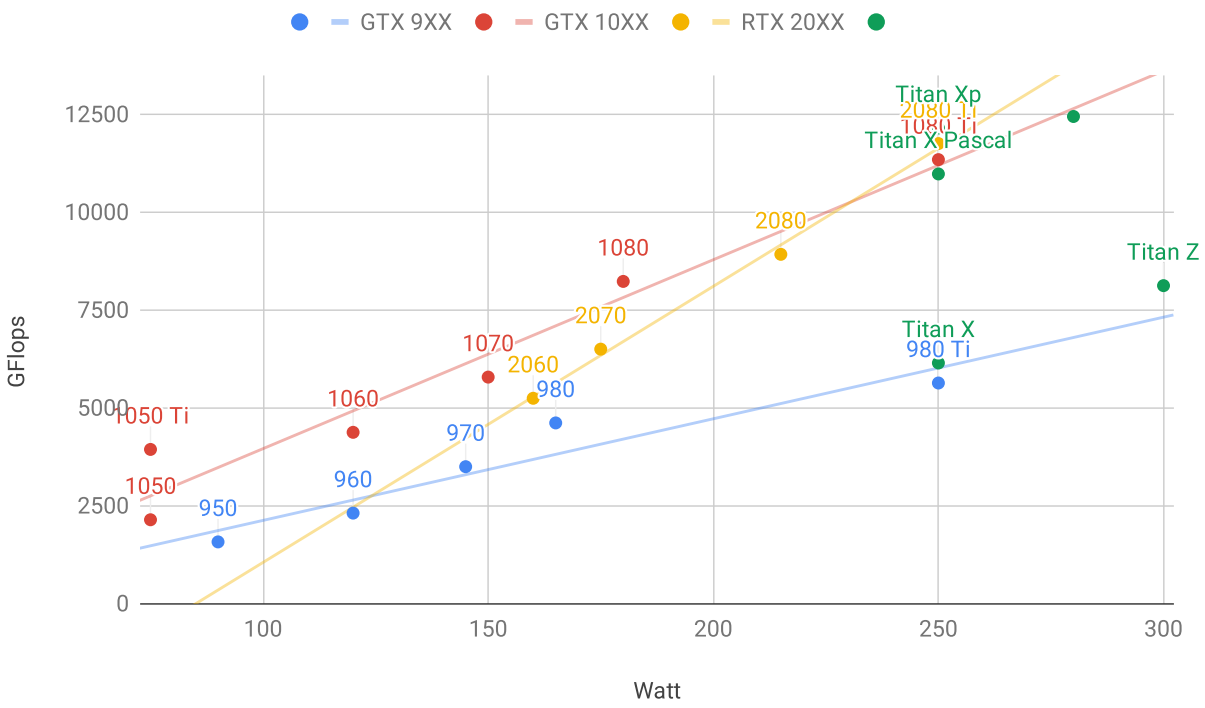


Fig. 20.5.2: Floating-point compute power and energy consumption.

Fig. 20.5.2 cho thấy mức tiêu thụ năng lượng quy mô chủ yếu là tuyến tính như thế nào với lượng tính toán. Thứ hai, các thế hệ sau này hiệu quả hơn. Điều này dường như bị mâu thuẫn bởi đồ thị tương ứng với dòng RTX 2000. Tuy nhiên, đây là hậu quả của TensorCores thu hút nhiều năng lượng không cân xứng.

20.5.3 Tóm tắt

- Coi chừng nguồn điện, làn xe buýt PCIe, tốc độ luồng đơn CPU và làm mát khi xây dựng máy chủ.
- Bạn nên mua thế hệ GPU mới nhất nếu có thể.
- Sử dụng đám mây để triển khai lớn.
- Máy chủ mật độ cao có thể không tương thích với tất cả các GPU. Kiểm tra các thông số kỹ thuật cơ khí và làm mát trước khi bạn mua.
- Sử dụng FP16 hoặc độ chính xác thấp hơn cho hiệu quả cao.

Discussions[236]

20.6 Đóng góp cho cuốn sách này

Đóng góp của readers[237] giúp chúng tôi cải thiện cuốn sách này. Nếu bạn tìm thấy một lỗi đánh máy, một liên kết lỗi thời, một cái gì đó mà bạn nghĩ rằng chúng tôi đã bỏ lỡ một trích dẫn, trong đó mã trông không thanh lịch hoặc nơi một lời giải thích là không rõ ràng, vui lòng đóng góp lại và giúp chúng tôi giúp đỡ độc giả của chúng tôi. Mặc dù trong các cuốn sách thông thường, sự chậm trễ giữa các lần chạy in (và do đó giữa các sửa lỗi đánh máy) có thể được đo bằng nhiều năm, thường mất hàng giờ đến nhiều ngày để kết hợp cải tiến trong cuốn sách này. Điều này là tất cả có thể do kiểm soát phiên bản và kiểm tra tích hợp liên tục. Để làm như vậy, bạn cần phải gửi một pull request[238] vào kho lưu trữ GitHub. Khi yêu cầu kéo của bạn được hợp nhất vào kho lưu trữ mã của tác giả, bạn sẽ trở thành người đóng góp.

20.6.1 Thay đổi văn bản nhỏ

Các đóng góp phổ biến nhất là chỉnh sửa một câu hoặc sửa lỗi chính tả. Chúng tôi khuyên bạn nên tìm tệp nguồn trong github repo để xác định vị trí tệp nguồn, đây là tệp markdown. Sau đó, bạn nhấp vào nút "Chỉnh sửa tệp này" ở góc trên bên phải để thực hiện các thay đổi của bạn trong tệp markdown.

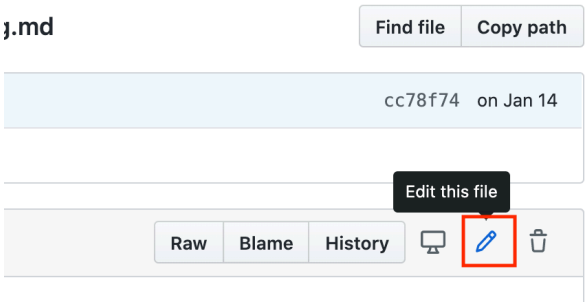


Fig. 20.6.1: Edit the file on Github.

Sau khi bạn hoàn tất, hãy điền vào mô tả thay đổi của bạn trong bảng "Đề xuất thay đổi tệp" ở phía dưới trang và sau đó nhấp vào nút "Đề xuất thay đổi tệp". Nó sẽ chuyển hướng bạn đến một trang mới để xem xét các thay đổi của bạn (Fig. 20.6.7). Nếu mọi thứ đều tốt, bạn có thể gửi yêu cầu kéo bằng cách nhấp vào nút "Tạo yêu cầu kéo".

[236] https://discuss.d2l.ai/t/425
[237] https://github.com/d2l-ai/d2l-en/graphs/contributors
[238] https://github.com/d2l-ai/d2l-en/pulls

20.6.2 Đề xuất một sự thay đổi lớn

Nếu bạn có kế hoạch cập nhật một phần lớn văn bản hoặc mã, thì bạn cần biết thêm một chút về định dạng cuốn sách này đang sử dụng. Tệp nguồn dựa trên markdown format[239] với một tập hợp các phần mở rộng thông qua gói d2lbook[240] như đề cập đến phương trình, hình ảnh, chương và trích dẫn. Bạn có thể sử dụng bất kỳ trình chỉnh sửa Markdown nào để mở các tệp này và thực hiện các thay đổi của mình.

Nếu bạn muốn thay đổi mã, chúng tôi khuyên bạn nên sử dụng Jupyter để mở các tệp Markdown này như được mô tả trong Section 20.1. Để bạn có thể chạy và kiểm tra các thay đổi của bạn. Vui lòng nhớ xóa tất cả các đầu ra trước khi gửi các thay đổi của bạn, hệ thống CI của chúng tôi sẽ thực hiện các phần bạn cập nhật để tạo ra kết quả đầu ra.

Một số phần có thể hỗ trợ nhiều triển khai framework, bạn có thể sử dụng `d2lbook` để kích hoạt một framework cụ thể, do đó các triển khai framework khác trở thành các khối mã Markdown và sẽ không được thực thi khi bạn "Run All" trong Jupyter. Nói cách khác, lần đầu tiên cài đặt `d2lbook` bằng cách chạy

```
pip install git+https://github.com/d2l-ai/d2l-book
```

Sau đó, trong thư mục gốc của `d2l-en`, bạn có thể kích hoạt một triển khai cụ thể bằng cách chạy một trong các lệnh sau:

```
d2lbook activate mxnet chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate pytorch chapter_multilayer-perceptrons/mlp-scratch.md
d2lbook activate tensorflow chapter_multilayer-perceptrons/mlp-scratch.md
```

Trước khi gửi các thay đổi của bạn, vui lòng xóa tất cả các đầu ra khối mã và kích hoạt tất cả bởi

```
d2lbook activate all chapter_multilayer-perceptrons/mlp-scratch.md
```

Nếu bạn thêm một khối mã mới không cho triển khai mặc định, đó là MXNet, vui lòng sử dụng `# @tab` to mark this block on the beginning line. For example, `# @tab pytorch` for a PyTorch code block, `# @tab tensorflow` for a TensorFlow code block, or `# @tab all` một khối mã được chia sẻ cho tất cả các triển khai. Bạn có thể tham khảo d2lbook[241] để biết thêm thông tin.

[239] https://daringfireball.net/projects/markdown/syntax
[240] http://book.d2l.ai/user/markdown.html
[241] http://book.d2l.ai/user/code_tabs.html

20.6.3 Thêm một phần mới hoặc một triển khai khung mới

Nếu bạn muốn tạo một chương mới, ví dụ: học tập củng cố hoặc thêm triển khai các framework mới, chẳng hạn như TensorFlow, vui lòng liên hệ với các tác giả trước, bằng cách gửi email hoặc sử dụng github issues[242].

20.6.4 Gửi một thay đổi lớn

Chúng tôi khuyên bạn nên sử dụng quy trình `git` tiêu chuẩn để gửi một thay đổi lớn. Tóm lại, quá trình hoạt động như được mô tả trong Fig. 20.6.2.

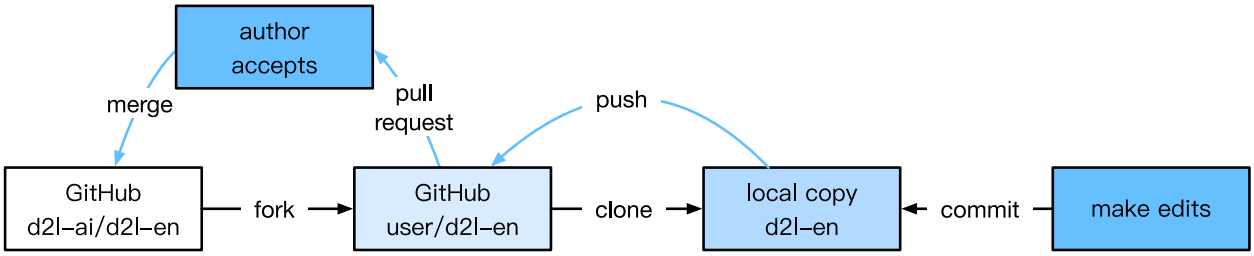


Fig. 20.6.2: Contributing to the book.

Chúng tôi sẽ hướng dẫn bạn qua các bước chi tiết. Nếu bạn đã quen thuộc với Git, bạn có thể bỏ qua phần này. Đối với cụ thể, chúng tôi giả định rằng tên người dùng của người đóng góp là "astonzhang".

Cài đặt Git

Sách mã nguồn mở Git mô tả how to install Git[243]. Điều này thường hoạt động thông qua `apt install git` trên Ubuntu Linux, bằng cách cài đặt các công cụ phát triển Xcode trên macOS hoặc bằng cách sử dụng desktop client[244] của GitHub. Nếu bạn không có tài khoản GitHub, bạn cần đăng ký một tài khoản.

Đăng nhập vào GitHub

Nhập address[245] kho lưu trữ mã của cuốn sách trong trình duyệt của bạn. Nhấp vào nút `Fork` trong hộp màu đỏ ở phía trên bên phải của Fig. 20.6.3, để tạo một bản sao của kho lưu trữ của cuốn sách này. Đây là bây giờ * bản sao của bạn* và bạn có thể thay đổi nó bất kỳ cách nào bạn muốn.

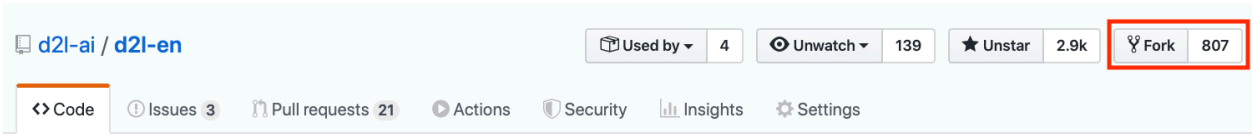


Fig. 20.6.3: The code repository page.

[242] https://github.com/d2l-ai/d2l-en/issues
[243] https://git-scm.com/book/en/v2
[244] https://desktop.github.com
[245] https://github.com/d2l-ai/d2l-en/

Bây giờ, kho lưu trữ mã của cuốn sách này sẽ được chia nhỏ (tức là sao chép) vào tên người dùng của bạn, chẳng hạn như `astonzhang/d2l-en` được hiển thị ở phía trên bên trái của ảnh chụp màn hình Fig. 20.6.4.

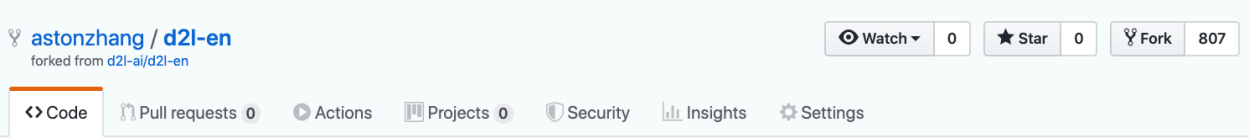


Fig. 20.6.4: Fork the code repository.

Nhân bản kho lưu trữ

Để sao chép kho lưu trữ (tức là, để tạo một bản sao địa phương) chúng ta cần phải có được địa chỉ kho lưu trữ của nó. Nút màu xanh lá cây trong Fig. 20.6.5 hiển thị điều này. Đảm bảo rằng bản sao cục bộ của bạn được cập nhật với kho lưu trữ chính nếu bạn quyết định giữ fork này lâu hơn. Bây giờ chỉ cần làm theo các hướng dẫn trong *Cài đặt* (page 9) để bắt đầu. Sự khác biệt chính là bây giờ bạn đang tải xuống * fork riêng của bạn* của kho lưu trữ.

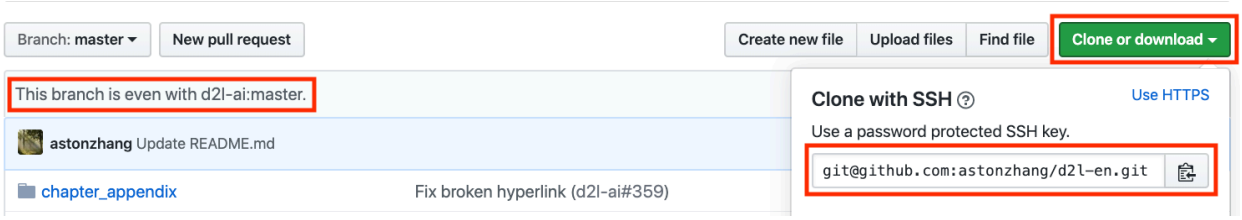


Fig. 20.6.5: Git clone.

```
# Replace your_github_username with your GitHub username
git clone https://github.com/your_github_username/d2l-en.git
```

Chỉnh sửa sách và đẩy

Bây giờ là lúc để chỉnh sửa cuốn sách. Tốt nhất là chỉnh sửa sổ ghi chép trong Jupyter theo hướng dẫn trong Section 20.1. Thực hiện các thay đổi và kiểm tra xem chúng có ổn không. Giả sử chúng ta đã sửa đổi một lỗi đánh máy trong tập tin `~/d2l-en/chapter_appendix_tools/how-to-contribute.md`. Sau đó, bạn có thể kiểm tra những tập tin bạn đã thay đổi:

Tại thời điểm này Git sẽ nhắc rằng tập tin `chapter_appendix_tools/how-to-contribute.md` đã được sửa đổi.

```
mylaptop:d2l-en me$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   chapter_appendix_tools/how-to-contribute.md
```

Sau khi xác nhận rằng đây là những gì bạn muốn, hãy thực hiện lệnh sau:

```
git add chapter_appendix_tools/how-to-contribute.md
git commit -m 'fix typo in git documentation'
git push
```

Mã đã thay đổi sau đó sẽ nằm trong ngã ba cá nhân của kho lưu trữ. Để yêu cầu bổ sung thay đổi của bạn, bạn phải tạo một yêu cầu kéo cho kho lưu trữ chính thức của cuốn sách.

Yêu cầu kéo

Như thể hiện trong Fig. 20.6.6, đi đến ngã ba của kho lưu trữ trên GitHub và chọn "New pull request". Thao tác này sẽ mở ra một màn hình hiển thị cho bạn những thay đổi giữa các chỉnh sửa của bạn và những gì hiện tại trong kho lưu trữ chính của cuốn sách.

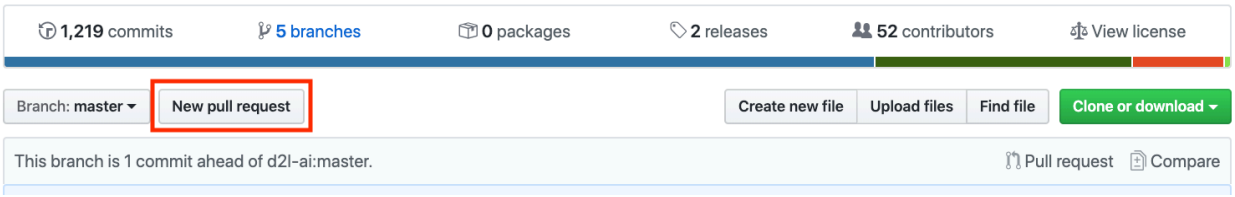


Fig. 20.6.6: Pull Request.

Gửi yêu cầu kéo

Cuối cùng, gửi yêu cầu kéo bằng cách nhấp vào nút như thể hiện trong Fig. 20.6.7. Đảm bảo mô tả những thay đổi bạn đã thực hiện trong yêu cầu kéo. Điều này sẽ giúp các tác giả xem xét nó dễ dàng hơn và hợp nhất nó với cuốn sách. Tùy thuộc vào những thay đổi, điều này có thể được chấp nhận ngay lập tức, bị từ chối hoặc nhiều khả năng hơn, bạn sẽ nhận được một số phản hồi về những thay đổi. Một khi bạn đã kết hợp chúng, bạn rất tốt để đi.

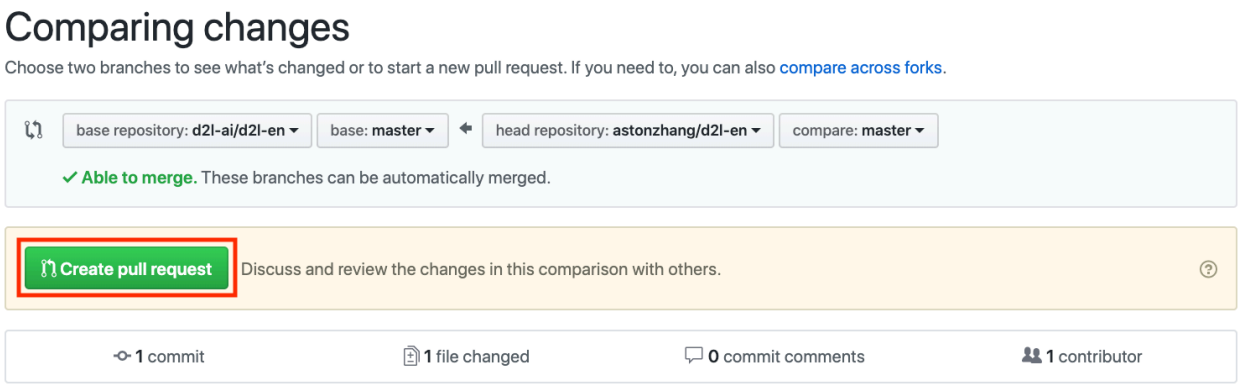


Fig. 20.6.7: Create Pull Request.

Yêu cầu kéo của bạn sẽ xuất hiện trong danh sách các yêu cầu trong kho lưu trữ chính. Chúng tôi sẽ cố gắng hết sức để xử lý nó một cách nhanh chóng.

20.6.5 Tóm tắt

- Bạn có thể sử dụng GitHub để đóng góp cho cuốn sách này.
- Bạn có thể chỉnh sửa tệp trên GitHub trực tiếp cho các thay đổi nhỏ.
- Đối với một thay đổi lớn, vui lòng ngã ba kho lưu trữ, chỉnh sửa mọi thứ cục bộ và chỉ đóng góp trở lại một khi bạn đã sẵn sàng.
- Yêu cầu kéo là cách đóng góp đang được gói lại. Cố gắng không gửi yêu cầu kéo khổng lồ vì điều này khiến họ khó hiểu và kết hợp. Tốt hơn gửi một số cái nhỏ hơn.

20.6.6 Bài tập

1. Sao và ngã ba kho lưu trữ `d2l-en`.
2. Tìm một số mã cần cải tiến và gửi yêu cầu kéo.
3. Tìm một tài liệu tham khảo mà chúng tôi đã bỏ lỡ và gửi yêu cầu kéo.
4. Nó thường là một thực hành tốt hơn để tạo một yêu cầu kéo bằng cách sử dụng một nhánh mới. Tìm hiểu làm thế nào để làm điều đó với Git branching[246].

Discussions[247]

20.7 `d2l` Tài liệu API

Việc triển khai các thành viên sau đây của gói `d2l` và các phần nơi chúng được xác định và giải thích có thể được tìm thấy trong source file[248].

class d2l.torch.**Accumulator**(*n*)

Bases: `object`

For accumulating sums over *n* variables.

add(**args*)

reset()

class d2l.torch.**AddNorm**(*normalized_shape*, *dropout*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Residual connection followed by layer normalization.

Defined in Section 11.7

forward(*X*, *Y*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

[246] https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
[247] https://discuss.d2l.ai/t/426
[248] https://github.com/d2l-ai/d2l-en/tree/master/d2l

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**AdditiveAttention**(*key_size*, *query_size*, *num_hiddens*, *dropout*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Additive attention.

Defined in Section 11.3

forward(*queries*, *keys*, *values*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**Animator**(*xlabel=None*, *ylabel=None*, *legend=None*, *xlim=None*, *ylim=None*, *xscale='linear'*, *yscale='linear'*, *fmts=('-', 'm--', 'g-.', 'r:')*, *nrows=1*, *ncols=1*, *figsize=(3.5, 2.5)*)

Bases: `object`

For plotting data in animation.

add(*x*, *y*)

class d2l.torch.**AttentionDecoder**(***kwargs*)

Bases: *d2l.torch.Decoder* (page 898)

The base attention-based decoder interface.

Defined in Section 11.4

property attention_weights

training: bool

class d2l.torch.**BERTEncoder**(*vocab_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *max_len=1000*, *key_size=768*, *query_size=768*, *value_size=768*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

BERT encoder.

Defined in Section 15.7.4

forward(*tokens*, *segments*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**BERTModel**(*vocab_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *max_len=1000*, *key_size=768*, *query_size=768*, *value_size=768*, *hid_in_features=768*, *mlm_in_features=768*, *nsp_in_features=768*)

Bases: `torch.nn.modules.module.Module`

The BERT model.

Defined in Section 15.7.5

forward(*tokens*, *segments*, *valid_lens=None*, *pred_positions=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**BananasDataset**(*is_train*)

Bases: `torch.utils.data.dataset.Dataset`

A customized dataset to load the banana detection dataset.

Defined in Section 14.6

class d2l.torch.**Benchmark**(*description='Done'*)

Bases: `object`

For measuring running time.

class d2l.torch.**Decoder**(***kwargs*)

Bases: `torch.nn.modules.module.Module`

The base decoder interface for the encoder-decoder architecture.

Defined in Section 10.6

forward(*X*, *state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

init_state(*enc_outputs*, **args*)

training: bool

class d2l.torch.**DotProductAttention**(*dropout*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Scaled dot product attention.

Defined in Section 11.3.2

forward(*queries*, *keys*, *values*, *valid_lens=None*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**Encoder**(***kwargs*)

Bases: `torch.nn.modules.module.Module`

The base encoder interface for the encoder-decoder architecture.

forward(*X*, **args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**EncoderBlock**(*key_size*, *query_size*, *value_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *dropout*, *use_bias=False*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Transformer encoder block.

Defined in Section 11.7

forward(*X*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**EncoderDecoder**(*encoder*, *decoder*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

The base class for the encoder-decoder architecture.

Defined in Section 10.6

forward(*enc_X*, *dec_X*, **args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**MaskLM**(*vocab_size*, *num_hiddens*, *num_inputs=768*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

The masked language model task of BERT.

Defined in Section 15.7.4

forward(*X*, *pred_positions*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**MaskedSoftmaxCELoss**(*weight: Optional[torch.Tensor] = None*, *size_average=None*, *ignore_index: int = - 100*, *reduce=None*, *reduction: str = 'mean'*, *label_smoothing: float = 0.0*)

Bases: `torch.nn.modules.loss.CrossEntropyLoss`

The softmax cross-entropy loss with masks.

Defined in Section 10.7.2

forward(*pred*, *label*, *valid_len*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

ignore_index: int

label_smoothing: float

class d2l.torch.**MultiHeadAttention**(*key_size*, *query_size*, *value_size*, *num_hiddens*, *num_heads*, *dropout*, *bias=False*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

Multi-head attention.

Defined in Section 11.5

forward(*queries*, *keys*, *values*, *valid_lens*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**NextSentencePred**(*num_inputs*, ***kwargs*)

Bases: `torch.nn.modules.module.Module`

The next sentence prediction task of BERT.

Defined in Section 15.7.5

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**PositionWiseFFN**(*ffn_num_input*, *ffn_num_hiddens*, *ffn_num_outputs*, ***kwargs*)

Bases: torch.nn.modules.module.Module

Positionwise feed-forward network.

Defined in Section 11.7

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**PositionalEncoding**(*num_hiddens*, *dropout*, *max_len=1000*)

Bases: torch.nn.modules.module.Module

Positional encoding.

Defined in Section 11.6

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**RNNModel**(*rnn_layer*, *vocab_size*, ***kwargs*)

Bases: torch.nn.modules.module.Module

The RNN model.

Defined in Section 9.6

begin_state(*device*, *batch_size=1*)

forward(*inputs*, *state*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**RNNModelScratch**(*vocab_size*, *num_hiddens*, *device*, *get_params*, *init_state*, *forward_fn*)

Bases: `object`

A RNN Model implemented from scratch.

begin_state(*batch_size*, *device*)

class d2l.torch.**RandomGenerator**(*sampling_weights*)

Bases: `object`

Randomly draw among {1, …, n} according to n sampling weights.

draw()

class d2l.torch.**Residual**(*input_channels*, *num_channels*, *use_1x1conv=False*, *strides=1*)

Bases: `torch.nn.modules.module.Module`

The Residual block of ResNet.

forward(*X*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**SNLIDataset**(*dataset*, *num_steps*, *vocab=None*)

Bases: `torch.utils.data.dataset.Dataset`

A customized dataset to load the SNLI dataset.

Defined in Section 16.4

class d2l.torch.**Seq2SeqEncoder**(*vocab_size*, *embed_size*, *num_hiddens*, *num_layers*, *dropout=0*, ***kwargs*)

Bases: *d2l.torch.Encoder* (page 899)

The RNN encoder for sequence to sequence learning.

Defined in Section 10.7

forward(*X*, **args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

class d2l.torch.**SeqDataLoader**(*batch_size*, *num_steps*, *use_random_iter*, *max_tokens*)

Bases: `object`

An iterator to load sequence data.

class d2l.torch.**Timer**

Bases: `object`

Record multiple running times.

avg()

Return the average time.

cumsum()

Return the accumulated time.

start()

Start the timer.

stop()

Stop the timer and record the time in a list.

sum()

Return the sum of time.

class d2l.torch.**TokenEmbedding**(*embedding_name*)

Bases: `object`

Token Embedding.

class d2l.torch.**TransformerEncoder**(*vocab_size*, *key_size*, *query_size*, *value_size*, *num_hiddens*, *norm_shape*, *ffn_num_input*, *ffn_num_hiddens*, *num_heads*, *num_layers*, *dropout*, *use_bias=False*, ***kwargs*)

Bases: *d2l.torch.Encoder* (page 899)

Transformer encoder.

Defined in Section 11.7

forward(*X*, *valid_lens*, **args*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class d2l.torch.**VOCSegDataset**(*is_train*, *crop_size*, *voc_dir*)

Bases: `torch.utils.data.dataset.Dataset`

A customized dataset to load the VOC dataset.

Defined in Section 14.9

filter(*imgs*)

normalize_image(*img*)

class d2l.torch.**Vocab**(*tokens=None*, *min_freq=0*, *reserved_tokens=None*)

Bases: `object`

Vocabulary for text.

to_tokens(*indices*)

property token_freqs

property unk

d2l.torch.**abs**(*input*, *, *out=None*) → Tensor

Computes the absolute value of each element in `input`.

$$\text{out}_i = |\text{input}_i| \tag{20.7.1}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.abs(torch.tensor([-1, -2, 3]))
tensor([ 1,  2,  3])
```

d2l.torch.**accuracy**(*y_hat*, *y*)

Compute the number of correct predictions.

Defined in Section 4.6

d2l.torch.**annotate**(*text*, *xy*, *xytext*)

d2l.torch.**arange**(*start=0*, *end*, *step=1*, *, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a 1-D tensor of size $\left\lceil \frac{\text{end}-\text{start}}{\text{step}} \right\rceil$ with values from the interval `[start, end)` taken with common difference `step` beginning from *start*.

Note that non-integer `step` is subject to floating point rounding errors when comparing against `end`; to avoid inconsistency, we advise adding a small epsilon to `end` in such cases.

$$\text{out}_{i+1} = \text{out}_i + \text{step} \tag{20.7.2}$$

Args: start (Number): the starting value for the set of points. Default: `0`. end (Number): the ending value for the set of points step (Number): the gap between each pair of adjacent points. Default: `1`.

Keyword args: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`). If *dtype* is not given, infer the data type from the other input arguments. If any of *start*, *end*, or *stop* are floating-point, the *dtype* is inferred to be the default dtype, see `get_default_dtype()`. Otherwise, the *dtype* is inferred to be *torch.int64*.

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.arange(5)
tensor([ 0,  1,  2,  3,  4])
>>> torch.arange(1, 4)
tensor([ 1,  2,  3])
>>> torch.arange(1, 2.5, 0.5)
tensor([ 1.0000,  1.5000,  2.0000])
```

`d2l.torch.`**`argmax`**(*x*, **args*, ***kwargs*)

`d2l.torch.`**`assign_anchor_to_bbox`**(*ground_truth*, *anchors*, *device*, *iou_threshold=0.5*)

Assign closest ground-truth bounding boxes to anchor boxes.

Defined in Section 14.4

`d2l.torch.`**`astype`**(*x*, **args*, ***kwargs*)

`d2l.torch.`**`batchify`**(*data*)

Return a minibatch of examples for skip-gram with negative sampling.

Defined in Section 15.3

`d2l.torch.`**`bbox_to_rect`**(*bbox*, *color*)

Convert bounding box to matplotlib format.

Defined in Section 14.3

d2l.torch.**bleu**(*pred_seq*, *label_seq*, *k*)

Compute the BLEU.

Defined in Section 10.7.4

d2l.torch.**box_center_to_corner**(*boxes*)

Convert from (center, width, height) to (upper-left, lower-right).

Defined in Section 14.3

d2l.torch.**box_corner_to_center**(*boxes*)

Convert from (upper-left, lower-right) to (center, width, height).

Defined in Section 14.3

d2l.torch.**box_iou**(*boxes1*, *boxes2*)

Compute pairwise IoU across two lists of anchor or bounding boxes.

Defined in Section 14.4

d2l.torch.**build_array_nmt**(*lines*, *vocab*, *num_steps*)

Transform text sequences of machine translation into minibatches.

Defined in Section 10.5.4

d2l.torch.**concat**()

cat(tensors, dim=0, *, out=None) -> Tensor

Concatenates the given sequence of `seq` tensors in the given dimension. All tensors must either have the same shape (except in the concatenating dimension) or be empty.

`torch.cat()` can be seen as an inverse operation for `torch.split()` and `torch.chunk()`.

`torch.cat()` can be best understood via examples.

Args:

tensors (sequence of Tensors): any python sequence of tensors of the same type. Non-empty tensors provided must have the same shape, except in the cat dimension.

dim (int, optional): the dimension over which the tensors are concatenated

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> x = torch.randn(2, 3)
>>> x
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 0)
tensor([[ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497],
        [ 0.6580, -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497]])
>>> torch.cat((x, x, x), 1)
tensor([[ 0.6580, -1.0969, -0.4614,  0.6580, -1.0969, -0.4614,  0.6580,
```

(continues on next page)

(continued from previous page)

```
         -1.0969, -0.4614],
        [-0.1034, -0.5790,  0.1497, -0.1034, -0.5790,  0.1497, -0.1034,
         -0.5790,  0.1497]])
```

d2l.torch.**copyfile**(*filename*, *target_dir*)

Copy a file into a target directory.

Defined in Section 14.13

d2l.torch.**corr2d**(*X*, *K*)

Compute 2D cross-correlation.

Defined in Section 7.2

d2l.torch.**cos**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the cosine of the elements of `input`.

$$\text{out}_i = \cos(\text{input}_i) \tag{20.7.3}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 1.4309,  1.2706, -0.8562,  0.9796])
>>> torch.cos(a)
tensor([ 0.1395,  0.2957,  0.6553,  0.5574])
```

d2l.torch.**cosh**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the hyperbolic cosine of the elements of `input`.

$$\text{out}_i = \cosh(\text{input}_i) \tag{20.7.4}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.1632,  1.1835, -0.6979, -0.7325])
>>> torch.cosh(a)
tensor([ 1.0133,  1.7860,  1.2536,  1.2805])
```

Note: When `input` is on the CPU, the implementation of torch.cosh may use the Sleef library, which rounds very large results to infinity or negative infinity. See here[249] for details.

---

[249] https://sleef.org/purec.xhtml

d2l.torch.**count_corpus**(*tokens*)

Count token frequencies.

Defined in Section 9.2

d2l.torch.**download**(*name*, *cache_dir='../data'*)

Download a file inserted into DATA_HUB, return the local filename.

Defined in Section 5.10

d2l.torch.**download_all**()

Download all files in the DATA_HUB.

Defined in Section 5.10

d2l.torch.**download_extract**(*name*, *folder=None*)

Download and extract a zip/tar file.

Defined in Section 5.10

d2l.torch.**evaluate_accuracy**(*net*, *data_iter*)

Compute the accuracy for a model on a dataset.

Defined in Section 4.6

d2l.torch.**evaluate_accuracy_gpu**(*net*, *data_iter*, *device=None*)

Compute the accuracy for a model on a dataset using a GPU.

Defined in Section 7.6

d2l.torch.**evaluate_loss**(*net*, *data_iter*, *loss*)

Evaluate the loss of a model on the given dataset.

Defined in Section 5.4

d2l.torch.**exp**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the exponential of the elements of the input tensor `input`.

$$y_i = e^{x_i} \tag{20.7.5}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.exp(torch.tensor([0, math.log(2.)]))
tensor([ 1.,  2.])
```

d2l.torch.**eye**(*n*, *m=None*, *, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a 2-D tensor with ones on the diagonal and zeros elsewhere.

Args: n (int): the number of rows m (int, optional): the number of columns with default being `n`

Keyword arguments: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Returns: Tensor: A 2-D tensor with ones on the diagonal and zeros elsewhere

Example:

```
>>> torch.eye(3)
tensor([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
```

`d2l.torch.`**`get_centers_and_contexts`**(*corpus*, *max_window_size*)

Return center words and context words in skip-gram.

Defined in Section 15.3

`d2l.torch.`**`get_data_ch11`**(*batch_size=10*, *n=1500*)

Defined in Section 12.5.2

`d2l.torch.`**`get_dataloader_workers`**()

Use 4 processes to read the data.

Defined in Section 4.5

`d2l.torch.`**`get_fashion_mnist_labels`**(*labels*)

Return text labels for the Fashion-MNIST dataset.

Defined in Section 4.5

`d2l.torch.`**`get_negatives`**(*all_contexts*, *vocab*, *counter*, *K*)

Return noise words in negative sampling.

Defined in Section 15.3

`d2l.torch.`**`get_tokens_and_segments`**(*tokens_a*, *tokens_b=None*)

Get tokens of the BERT input sequence and their segment IDs.

Defined in Section 15.7

`d2l.torch.`**`grad_clipping`**(*net*, *theta*)

Clip the gradient.

Defined in Section 9.5

d2l.torch.**linreg**(*X*, *w*, *b*)

The linear regression model.

Defined in Section 4.2

d2l.torch.**linspace**(*start*, *end*, *steps*, *, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Creates a one-dimensional tensor of size `steps` whose values are evenly spaced from `start` to `end`, inclusive. That is, the value are:

$$(\text{start}, \text{start} + \frac{\text{end} - \text{start}}{\text{steps} - 1}, \ldots, \text{start} + (\text{steps} - 2) * \frac{\text{end} - \text{start}}{\text{steps} - 1}, \text{end}) \tag{20.7.6}$$

From PyTorch 1.11 linspace requires the steps argument. Use steps=100 to restore the previous behavior.

Args: start (float): the starting value for the set of points end (float): the ending value for the set of points steps (int): size of the constructed tensor

Keyword arguments: out (Tensor, optional): the output tensor. dtype (torch.dtype, optional): the data type to perform the computation in.

Default: if None, uses the global default dtype (see torch.get_default_dtype()) when both `start` and `end` are real, and corresponding complex dtype when either is complex.

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.linspace(3, 10, steps=5)
tensor([  3.0000,   4.7500,   6.5000,   8.2500,  10.0000])
>>> torch.linspace(-10, 10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=5)
tensor([-10.,  -5.,   0.,   5.,  10.])
>>> torch.linspace(start=-10, end=10, steps=1)
tensor([-10.])
```

d2l.torch.**load_array**(*data_arrays*, *batch_size*, *is_train=True*)

Construct a PyTorch data iterator.

Defined in Section 4.3

d2l.torch.**load_corpus_time_machine**(*max_tokens=- 1*)

Return token indices and the vocabulary of the time machine dataset.

Defined in Section 9.2

d2l.torch.**load_data_bananas**(*batch_size*)

Load the banana detection dataset.

Defined in Section 14.6

d2l.torch.**load_data_fashion_mnist**(*batch_size*, *resize=None*)

Download the Fashion-MNIST dataset and then load it into memory.

Defined in Section 4.5

d2l.torch.**load_data_imdb**(*batch_size*, *num_steps=500*)

Return data iterators and the vocabulary of the IMDb review dataset.

Defined in Section 16.1

d2l.torch.**load_data_nmt**(*batch_size*, *num_steps*, *num_examples=600*)

Return the iterator and the vocabularies of the translation dataset.

Defined in Section 10.5.4

d2l.torch.**load_data_ptb**(*batch_size*, *max_window_size*, *num_noise_words*)

Download the PTB dataset and then load it into memory.

Defined in Section 15.3.5

d2l.torch.**load_data_snli**(*batch_size*, *num_steps=50*)

Download the SNLI dataset and return data iterators and vocabulary.

Defined in Section 16.4

d2l.torch.**load_data_time_machine**(*batch_size*, *num_steps*, *use_random_iter=False*, *max_tokens=10000*)

Return the iterator and the vocabulary of the time machine dataset.

Defined in Section 9.3

d2l.torch.**load_data_voc**(*batch_size*, *crop_size*)

Load the VOC semantic segmentation dataset.

Defined in Section 14.9

d2l.torch.**load_data_wiki**(*batch_size*, *max_len*)

Load the WikiText-2 dataset.

Defined in Section 15.8.1

d2l.torch.**log**(*input*, *, *out=None*) $\rightarrow$ Tensor

Returns a new tensor with the natural logarithm of the elements of `input`.

$$y_i = \log_e(x_i) \tag{20.7.7}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(5)
>>> a
tensor([-0.7168, -0.5471, -0.8933, -1.4428, -0.1190])
>>> torch.log(a)
tensor([ nan,  nan,  nan,  nan,  nan])
```

d2l.torch.**masked_softmax**(*X*, *valid_lens*)

Perform softmax operation by masking elements on the last axis.

Defined in Section 11.3

d2l.torch.**matmul**(*input*, *other*, *, *out=None*) → Tensor

Matrix product of two tensors.

The behavior depends on the dimensionality of the tensors as follows:

- If both tensors are 1-dimensional, the dot product (scalar) is returned.
- If both arguments are 2-dimensional, the matrix-matrix product is returned.
- If the first argument is 1-dimensional and the second argument is 2-dimensional, a 1 is prepended to its dimension for the purpose of the matrix multiply. After the matrix multiply, the prepended dimension is removed.
- If the first argument is 2-dimensional and the second argument is 1-dimensional, the matrix-vector product is returned.
- If both arguments are at least 1-dimensional and at least one argument is N-dimensional (where N > 2), then a batched matrix multiply is returned. If the first argument is 1-dimensional, a 1 is prepended to its dimension for the purpose of the batched matrix multiply and removed after. If the second argument is 1-dimensional, a 1 is appended to its dimension for the purpose of the batched matrix multiple and removed after. The non-matrix (i.e. batch) dimensions are broadcasted (and thus must be broadcastable). For example, if `input` is a $(j \times 1 \times n \times n)$ tensor and `other` is a $(k \times n \times n)$ tensor, `out` will be a $(j \times k \times n \times n)$ tensor.

  Note that the broadcasting logic only looks at the batch dimensions when determining if the inputs are broadcastable, and not the matrix dimensions. For example, if `input` is a $(j \times 1 \times n \times m)$ tensor and `other` is a $(k \times m \times p)$ tensor, these inputs are valid for broadcasting even though the final two dimensions (i.e. the matrix dimensions) are different. `out` will be a $(j \times k \times n \times p)$ tensor.

This operator supports TensorFloat32.

Note: The 1-dimensional dot product version of this function does not support an `out` parameter.

Arguments: input (Tensor): the first tensor to be multiplied other (Tensor): the second tensor to be multiplied

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> # vector x vector
>>> tensor1 = torch.randn(3)
>>> tensor2 = torch.randn(3)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([])
>>> # matrix x vector
>>> tensor1 = torch.randn(3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([3])
>>> # batched matrix x broadcasted vector
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3])
>>> # batched matrix x batched matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(10, 4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
>>> # batched matrix x broadcasted matrix
>>> tensor1 = torch.randn(10, 3, 4)
>>> tensor2 = torch.randn(4, 5)
>>> torch.matmul(tensor1, tensor2).size()
torch.Size([10, 3, 5])
```

d2l.torch.**multibox_detection**(*cls_probs*, *offset_preds*, *anchors*, *nms_threshold=0.5*, *pos_threshold=0.009999999*)

Predict bounding boxes using non-maximum suppression.

Defined in Section 14.4.4

d2l.torch.**multibox_prior**(*data*, *sizes*, *ratios*)

Generate anchor boxes with different shapes centered on each pixel.

Defined in Section 14.4

d2l.torch.**multibox_target**(*anchors*, *labels*)

Label anchor boxes using ground-truth bounding boxes.

Defined in Section 14.4.3

d2l.torch.**nms**(*boxes*, *scores*, *iou_threshold*)

Sort confidence scores of predicted bounding boxes.

Defined in Section 14.4.4

d2l.torch.**normal**(*mean*, *std*, *, *generator=None*, *out=None*) → Tensor

Returns a tensor of random numbers drawn from separate normal distributions whose mean and standard deviation are given.

The `mean` is a tensor with the mean of each output element's normal distribution

The `std` is a tensor with the standard deviation of each output element's normal distribution

The shapes of `mean` and `std` don't need to match, but the total number of elements in each tensor need to be the same.

Note: When the shapes do not match, the shape of `mean` is used as the shape for the returned output tensor

Note: When `std` is a CUDA tensor, this function synchronizes its device with the CPU.

Args: mean (Tensor): the tensor of per-element means std (Tensor): the tensor of per-element standard deviations

Keyword args: generator (`torch.Generator`, optional): a pseudorandom number generator for sampling out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(mean=torch.arange(1., 11.), std=torch.arange(1, 0, -0.
↪1))
tensor([  1.0425,   3.5672,   2.7969,   4.2925,   4.7229,   6.2134,
          8.0505,   8.1408,   9.0563,  10.0566])
```

`d2l.torch.`**`normal`**(*mean=0.0*, *std*, *, *out=None*) → Tensor

Similar to the function above, but the means are shared among all drawn elements.

Args: mean (float, optional): the mean for all distributions std (Tensor): the tensor of per-element standard deviations

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(mean=0.5, std=torch.arange(1., 6.))
tensor([-1.2793, -1.0732, -2.0687,  5.1177, -1.2303])
```

`d2l.torch.`**`normal`**(*mean*, *std=1.0*, *, *out=None*) → Tensor

Similar to the function above, but the standard deviations are shared among all drawn elements.

Args: mean (Tensor): the tensor of per-element means std (float, optional): the standard deviation for all distributions

Keyword args: out (Tensor, optional): the output tensor

Example:

```
>>> torch.normal(mean=torch.arange(1., 6.))
tensor([ 1.1552,  2.6148,  2.6535,  5.8318,  4.2361])
```

`d2l.torch.`**`normal`**(*mean*, *std*, *size*, *, *out=None*) → Tensor

Similar to the function above, but the means and standard deviations are shared among all drawn elements. The resulting tensor has size given by *`size`* (page 920).

Args: mean (float): the mean for all distributions std (float): the standard deviation for all distributions size (int…): a sequence of integers defining the shape of the output tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> torch.normal(2, 3, size=(1, 4))
tensor([[-1.3987, -1.9544,  3.6048,  0.7909]])
```

d2l.torch.**numpy**(*x*, **args*, ***kwargs*)

d2l.torch.**offset_boxes**(*anchors*, *assigned_bb*, *eps=1e-06*)

Transform for anchor box offsets.

Defined in Section 14.4.3

d2l.torch.**offset_inverse**(*anchors*, *offset_preds*)

Predict bounding boxes based on anchor boxes with predicted offsets.

Defined in Section 14.4.3

d2l.torch.**ones**(**size*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *1*, with the shape defined by the variable argument *size* (page 920).

Args:

size (int...): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword arguments: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.ones(2, 3)
tensor([[ 1.,  1.,  1.],
        [ 1.,  1.,  1.]])

>>> torch.ones(5)
tensor([ 1.,  1.,  1.,  1.,  1.])
```

d2l.torch.**plot**(*X*, *Y=None*, *xlabel=None*, *ylabel=None*, *legend=None*, *xlim=None*, *ylim=None*, *xscale='linear'*, *yscale='linear'*, *fmts=('-', 'm--', 'g-.', 'r:')*, *figsize=(3.5, 2.5)*, *axes=None*)

Plot data points.

Defined in Section 3.4

d2l.torch.**predict_ch3**(*net*, *test_iter*, *n=6*)

Predict labels (defined in Chapter 3).

Defined in Section 4.6

d2l.torch.**predict_ch8**(*prefix*, *num_preds*, *net*, *vocab*, *device*)

Generate new characters following the *prefix*.

Defined in Section 9.5

d2l.torch.**predict_sentiment**(*net*, *vocab*, *sequence*)

Predict the sentiment of a text sequence.

Defined in Section 16.2

d2l.torch.**predict_seq2seq**(*net*, *src_sentence*, *src_vocab*, *tgt_vocab*, *num_steps*, *device*, *save_attention_weights=False*)

Predict for sequence to sequence.

Defined in Section 10.7.4

d2l.torch.**predict_snli**(*net*, *vocab*, *premise*, *hypothesis*)

Predict the logical relationship between the premise and hypothesis.

Defined in Section 16.5

d2l.torch.**preprocess_nmt**(*text*)

Preprocess the English-French dataset.

Defined in Section 10.5

d2l.torch.**rand**(**size*, *, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with random numbers from a uniform distribution on the interval $[0, 1)$

The shape of the tensor is defined by the variable argument `size` (page 920).

Args:

size (int...): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword args: generator (`torch.Generator`, optional): a pseudorandom number generator for sampling out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.rand(4)
tensor([ 0.5204,  0.2503,  0.3525,  0.5673])
>>> torch.rand(2, 3)
tensor([[ 0.8237,  0.5781,  0.6879],
        [ 0.3816,  0.7249,  0.0998]])
```

`d2l.torch.`**`read_csv_labels`**(*fname*)

Read *fname* to return a filename to label dictionary.

Defined in Section 14.13

`d2l.torch.`**`read_data_bananas`**(*is_train=True*)

Read the banana detection dataset images and labels.

Defined in Section 14.6

`d2l.torch.`**`read_data_nmt`**()

Load the English-French dataset.

Defined in Section 10.5

`d2l.torch.`**`read_imdb`**(*data_dir*, *is_train*)

Read the IMDb review dataset text sequences and labels.

Defined in Section 16.1

`d2l.torch.`**`read_ptb`**()

Load the PTB dataset into a list of text lines.

Defined in Section 15.3

`d2l.torch.`**`read_snli`**(*data_dir*, *is_train*)

Read the SNLI dataset into premises, hypotheses, and labels.

Defined in Section 16.4

`d2l.torch.`**`read_time_machine`**()

Load the time machine dataset into a list of text lines.

Defined in Section 9.2

`d2l.torch.`**`read_voc_images`**(*voc_dir*, *is_train=True*)

Read all VOC feature and label images.

Defined in Section 14.9

`d2l.torch.`**`reduce_sum`**(*x*, **args*, ***kwargs*)

d2l.torch.**reorg_test**(*data_dir*)

Organize the testing set for data loading during prediction.

Defined in Section 14.13

d2l.torch.**reorg_train_valid**(*data_dir*, *labels*, *valid_ratio*)

Split the validation set out of the original training set.

Defined in Section 14.13

d2l.torch.**reshape**(*x*, **args*, ***kwargs*)

d2l.torch.**resnet18**(*num_classes*, *in_channels=1*)

A slightly modified ResNet-18 model.

Defined in Section 13.6

d2l.torch.**seq_data_iter_random**(*corpus*, *batch_size*, *num_steps*)

Generate a minibatch of subsequences using random sampling.

Defined in Section 9.3

d2l.torch.**seq_data_iter_sequential**(*corpus*, *batch_size*, *num_steps*)

Generate a minibatch of subsequences using sequential partitioning.

Defined in Section 9.3

d2l.torch.**sequence_mask**(*X*, *valid_len*, *value=0*)

Mask irrelevant entries in sequences.

Defined in Section 10.7.2

d2l.torch.**set_axes**(*axes*, *xlabel*, *ylabel*, *xlim*, *ylim*, *xscale*, *yscale*, *legend*)

Set the axes for matplotlib.

Defined in Section 3.4

d2l.torch.**set_figsize**(*figsize=(3.5, 2.5)*)

Set the figure size for matplotlib.

Defined in Section 3.4

d2l.torch.**sgd**(*params*, *lr*, *batch_size*)

Minibatch stochastic gradient descent.

Defined in Section 4.2

d2l.torch.**show_bboxes**(*axes*, *bboxes*, *labels=None*, *colors=None*)

Show bounding boxes.

Defined in Section 14.4

d2l.torch.**show_heatmaps**(*matrices*, *xlabel*, *ylabel*, *titles=None*, *figsize=(2.5, 2.5)*, *cmap='Reds'*)

Show heatmaps of matrices.

Defined in Section 11.1

d2l.torch.**show_images**(*imgs*, *num_rows*, *num_cols*, *titles=None*, *scale=1.5*)

Plot a list of images.

Defined in Section 4.5

d2l.torch.**show_list_len_pair_hist**(*legend*, *xlabel*, *ylabel*, *xlist*, *ylist*)

Plot the histogram for list length pairs.

Defined in Section 10.5

d2l.torch.**show_trace_2d**(*f*, *results*)

Show the trace of 2D variables during optimization.

Defined in Section 12.3.1

d2l.torch.**sin**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the sine of the elements of `input`.

$$\text{out}_i = \sin(\text{input}_i) \tag{20.7.8}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([-0.5461,  0.1347, -2.7266, -0.2746])
>>> torch.sin(a)
tensor([-0.5194,  0.1343, -0.4032, -0.2711])
```

d2l.torch.**sinh**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the hyperbolic sine of the elements of `input`.

$$\text{out}_i = \sinh(\text{input}_i) \tag{20.7.9}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.5380, -0.8632, -0.1265,  0.9399])
>>> torch.sinh(a)
tensor([ 0.5644, -0.9744, -0.1268,  1.0845])
```

Note: When `input` is on the CPU, the implementation of torch.sinh may use the Sleef library, which rounds very large results to infinity or negative infinity. See here[250] for details.

---

d2l.torch.**size**(*x*, **args*, ***kwargs*)

[250] https://sleef.org/purec.xhtml

d2l.torch.**split_batch**(*X*, *y*, *devices*)

Split *X* and *y* into multiple devices.

Defined in Section 13.5

d2l.torch.**squared_loss**(*y_hat*, *y*)

Squared loss.

Defined in Section 4.2

d2l.torch.**stack**(*tensors*, *dim=0*, *, *out=None*) → Tensor

Concatenates a sequence of tensors along a new dimension.

All tensors need to be of the same size.

Arguments: tensors (sequence of Tensors): sequence of tensors to concatenate dim (int): dimension to insert. Has to be between 0 and the number

of dimensions of concatenated tensors (inclusive)

Keyword args: out (Tensor, optional): the output tensor.

d2l.torch.**subsample**(*sentences*, *vocab*)

Subsample high-frequency words.

Defined in Section 15.3

d2l.torch.**synthetic_data**(*w*, *b*, *num_examples*)

Generate y = Xw + b + noise.

Defined in Section 4.2

d2l.torch.**tanh**(*input*, *, *out=None*) → Tensor

Returns a new tensor with the hyperbolic tangent of the elements of `input`.

$$\text{out}_i = \tanh(\text{input}_i) \tag{20.7.10}$$

Args: input (Tensor): the input tensor.

Keyword args: out (Tensor, optional): the output tensor.

Example:

```
>>> a = torch.randn(4)
>>> a
tensor([ 0.8986, -0.7279,  1.1745,  0.2611])
>>> torch.tanh(a)
tensor([ 0.7156, -0.6218,  0.8257,  0.2553])
```

d2l.torch.**tensor**(*data*, *, *dtype=None*, *device=None*, *requires_grad=False*, *pin_memory=False*) → Tensor

Constructs a tensor with no autograd history (also known as a “leaf tensor”, see /notes/autograd) by copying `data`.

Warning: When working with tensors prefer using `torch.Tensor.clone()`, `torch.Tensor.detach()`, and `torch.Tensor.requires_grad_()` for readability. Letting *t* be a tensor, `torch.tensor(t)` is equivalent to `t.clone().detach()`, and `torch.tensor(t, requires_grad=True)` is equivalent to `t.clone().detach().requires_grad_(True)`.

See also:

`torch.as_tensor()` preserves autograd history and avoids copies where possible. `torch.from_numpy()` creates a tensor that shares storage with a NumPy array.

Args:

data (array_like): Initial data for the tensor. Can be a list, tuple, NumPy `ndarray`, scalar, and other types.

Keyword args:

dtype (`torch.dtype`, optional): the desired data type of returned tensor. Default: if `None`, infers data type from `data`.

device (`torch.device`, optional): the device of the constructed tensor. If None and data is a tensor then the device of data is used. If None and data is not a tensor then the result tensor is constructed on the CPU.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

pin_memory (bool, optional): If set, returned tensor would be allocated in the pinned memory. Works only for CPU tensors. Default: `False`.

Example:

```
>>> torch.tensor([[0.1, 1.2], [2.2, 3.1], [4.9, 5.2]])
tensor([[ 0.1000,  1.2000],
        [ 2.2000,  3.1000],
        [ 4.9000,  5.2000]])

>>> torch.tensor([0, 1])  # Type inference on data
tensor([ 0,  1])

>>> torch.tensor([[0.11111, 0.222222, 0.3333333]],
...              dtype=torch.float64,
...              device=torch.device('cuda:0'))  # creates a double␣
↪tensor on a CUDA device
tensor([[ 0.1111,  0.2222,  0.3333]], dtype=torch.float64, device='cuda:0
↪')

>>> torch.tensor(3.14159)  # Create a zero-dimensional (scalar) tensor
tensor(3.1416)

>>> torch.tensor([])  # Create an empty tensor (of size (0,))
tensor([])
```

`d2l.torch.to`(*x*, **args*, ***kwargs*)

d2l.torch.**tokenize**(*lines*, *token='word'*)

Split text lines into word or character tokens.

Defined in Section 9.2

d2l.torch.**tokenize_nmt**(*text*, *num_examples=None*)

Tokenize the English-French dataset.

Defined in Section 10.5

d2l.torch.**train_2d**(*trainer*, *steps=20*, *f_grad=None*)

Optimize a 2D objective function with a customized trainer.

Defined in Section 12.3.1

d2l.torch.**train_batch_ch13**(*net*, *X*, *y*, *loss*, *trainer*, *devices*)

Train for a minibatch with mutiple GPUs (defined in Chapter 13).

Defined in Section 14.1

d2l.torch.**train_ch11**(*trainer_fn*, *states*, *hyperparams*, *data_iter*, *feature_dim*, *num_epochs=2*)

Defined in Section 12.5.2

d2l.torch.**train_ch13**(*net*, *train_iter*, *test_iter*, *loss*, *trainer*, *num_epochs*, *devices=[device(type='cuda', index=0), device(type='cuda', index=1), device(type='cuda', index=2), device(type='cuda', index=3)]*)

Train a model with mutiple GPUs (defined in Chapter 13).

Defined in Section 14.1

d2l.torch.**train_ch3**(*net*, *train_iter*, *test_iter*, *loss*, *num_epochs*, *updater*)

Train a model (defined in Chapter 3).

Defined in Section 4.6

d2l.torch.**train_ch6**(*net*, *train_iter*, *test_iter*, *num_epochs*, *lr*, *device*)

Train a model with a GPU (defined in Chapter 6).

Defined in Section 7.6

d2l.torch.**train_ch8**(*net*, *train_iter*, *vocab*, *lr*, *num_epochs*, *device*, *use_random_iter=False*)

Train a model (defined in Chapter 8).

Defined in Section 9.5

d2l.torch.**train_concise_ch11**(*trainer_fn*, *hyperparams*, *data_iter*, *num_epochs=4*)

Defined in Section 12.5.2

d2l.torch.**train_epoch_ch3**(*net*, *train_iter*, *loss*, *updater*)

The training loop defined in Chapter 3.

Defined in Section 4.6

d2l.torch.**train_epoch_ch8**(*net*, *train_iter*, *loss*, *updater*, *device*, *use_random_iter*)

Train a net within one epoch (defined in Chapter 8).

Defined in Section 9.5

d2l.torch.**train_seq2seq**(*net*, *data_iter*, *lr*, *num_epochs*, *tgt_vocab*, *device*)

Train a model for sequence to sequence.

Defined in Section 10.7.2

d2l.torch.**transpose**(*x*, **args*, ***kwargs*)

d2l.torch.**transpose_output**(*X*, *num_heads*)

Reverse the operation of *transpose_qkv*.

Defined in Section 11.5

d2l.torch.**transpose_qkv**(*X*, *num_heads*)

Transposition for parallel computation of multiple attention heads.

Defined in Section 11.5

d2l.torch.**truncate_pad**(*line*, *num_steps*, *padding_token*)

Truncate or pad sequences.

Defined in Section 10.5

d2l.torch.**try_all_gpus**()

Return all available GPUs, or [cpu(),] if no GPU exists.

Defined in Section 6.5

d2l.torch.**try_gpu**(*i=0*)

Return gpu(i) if exists, otherwise return cpu().

Defined in Section 6.5

d2l.torch.**update_D**(*X*, *Z*, *net_D*, *net_G*, *loss*, *trainer_D*)

Update discriminator.

Defined in Section 18.1

d2l.torch.**update_G**(*Z*, *net_D*, *net_G*, *loss*, *trainer_G*)

Update generator.

Defined in Section 18.1

d2l.torch.**use_svg_display**()

Use the svg format to display a plot in Jupyter.

Defined in Section 3.4

d2l.torch.**voc_colormap2label**()

Build the mapping from RGB to class indices for VOC labels.

Defined in Section 14.9

d2l.torch.**voc_label_indices**(*colormap*, *colormap2label*)

Map any RGB values in VOC labels to their class indices.

Defined in Section 14.9

d2l.torch.**voc_rand_crop**(*feature*, *label*, *height*, *width*)

Randomly crop both feature and label images.

Defined in Section 14.9

d2l.torch.**zeros**(**size*, ***, *out=None*, *dtype=None*, *layout=torch.strided*, *device=None*, *requires_grad=False*) → Tensor

Returns a tensor filled with the scalar value *0*, with the shape defined by the variable argument *size* (page 920).

Args:

size (int…): a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword args: out (Tensor, optional): the output tensor. dtype (`torch.dtype`, optional): the desired data type of returned tensor.

Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).

layout (`torch.layout`, optional): the desired layout of returned Tensor. Default: `torch.strided`.

device (`torch.device`, optional): the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

requires_grad (bool, optional): If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.zeros(2, 3)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

>>> torch.zeros(5)
tensor([ 0.,  0.,  0.,  0.,  0.])
```

Bibliography

Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., & Smola, A. J. (2012). Scalable inference in latent variable models. *Proceedings of the fifth ACM international conference on Web search and data mining* (pp. 123–132).

Aji, S. M., & McEliece, R. J. (2000). The generalized distributive law. *IEEE transactions on Information Theory*, *46*(2), 325–343.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bay, H., Tuytelaars, T., & Van Gool, L. (2006). Surf: speeded up robust features. *European conference on computer vision* (pp. 404–417).

Bishop, C. M. (1995). Training with noise is equivalent to tikhonov regularization. *Neural computation*, *7*(1), 108–116.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Bollobás, B. (1999). *Linear analysis*. Cambridge University Press, Cambridge.

Bowman, S. R., Angeli, G., Potts, C., & Manning, C. D. (2015). A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Boyd, S., & Vandenberghe, L. (2004). *Convex Optimization*. Cambridge, England: Cambridge University Press.

Brown, N., & Sandholm, T. (2017). Libratus: the superhuman ai for no-limit poker. *IJCAI* (pp. 5226–5228).

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J., … Roossin, P. S. (1990). A statistical approach to machine translation. *Computational linguistics*, *16*(2), 79–85.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L., & Roossin, P. (1988). A statistical approach to language translation. *Coling Budapest 1988 Volume 1: International Conference on Computational Linguistics*.

Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, *134*(1-2), 57–83.

Canny, J. (1987). A computational approach to edge detection. *Readings in computer vision* (pp. 184–203). Elsevier.

Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. (2017). Semeval-2017 task 1: semantic textual similarity multilingual and crosslingual focused evaluation. *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)* (pp. 1–14).

Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Csiszár, I. (2008). Axiomatic characterizations of information measures. *Entropy*, *10*(3), 261–273.

Dalal, N., & Triggs, B. (2005). Histograms of oriented gradients for human detection. *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)* (pp. 886–893).

De Cock, D. (2011). Ames, iowa: alternative to the boston housing data as an end of semester regression project. *Journal of Statistics Education*, *19*(3).

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., … Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review* (pp. 205–220).

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Doucet, A., De Freitas, N., & Gordon, N. (2001). An introduction to sequential monte carlo methods. *Sequential Monte Carlo methods in practice* (pp. 3–14). Springer.

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*(Jul), 2121–2159.

Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Flammarion, N., & Bach, F. (2015). From averaging to acceleration, there is only a step-size. *Conference on Learning Theory* (pp. 658–695).

Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2414–2423).

Girshick, R. (2015). Fast r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 580–587).

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics* (pp. 249–256).

Goh, G. (2017). Why momentum really works. *Distill*. URL: http://distill.pub/2017/momentum, doi:10.23915/distill.00006[251]

Goldberg, D., Nichols, D., Oki, B. M., & Terry, D. (1992). Using collaborative filtering to weave an information tapestry. *Communications of the ACM*, *35*(12), 61–71.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., … Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems* (pp. 2672–2680).

[251] https://doi.org/10.23915/distill.00006

Gotmare, A., Keskar, N. S., Xiong, C., & Socher, R. (2018). A closer look at deep learning heuristics: learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*.

Graves, A., & Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural networks*, *18*(5-6), 602–610.

Hadjis, S., Zhang, C., Mitliagkas, I., Iter, D., & Ré, C. (2016). Omnivore: an optimizer for multi-device deep learning on cpus and gpus. *arXiv preprint arXiv:1606.04487*.

He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). Mask r-cnn. *Proceedings of the IEEE international conference on computer vision* (pp. 2961–2969).

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: surpassing human-level performance on imagenet classification. *Proceedings of the IEEE international conference on computer vision* (pp. 1026–1034).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770–778).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks. *European conference on computer vision* (pp. 630–645).

Hebb, D. O., & Hebb, D. (1949). *The organization of behavior*. Vol. 65. Wiley New York.

Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Hoyer, P. O., Janzing, D., Mooij, J. M., Peters, J., & Schölkopf, B. (2009). Nonlinear causal discovery with additive noise models. *Advances in neural information processing systems* (pp. 689–696).

Hu, J., Shen, L., & Sun, G. (2018). Squeeze-and-excitation networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7132–7141).

Hu, Y., Koren, Y., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).

Huang, G., Liu, Z., Van Der Maaten, L., & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4700–4708).

Ioffe, S. (2017). Batch renormalization: towards reducing minibatch dependence in batch-normalized models. *Advances in neural information processing systems* (pp. 1945–1953).

Ioffe, S., & Szegedy, C. (2015). Batch normalization: accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., & Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Jaeger, H. (2002). *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. Vol. 5. GMD-Forschungszentrum Informationstechnik Bonn.

Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., … others. (2018). Highly scalable deep learning training system with mixed-precision: training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., … others. (2017). In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)* (pp. 1–12).

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*.

Kingma, D. P., & Ba, J. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Koller, D., & Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Kolter, Z. (2008). Linear algebra review and reference. *Available online: http*.

Koren, Y. (2009). Collaborative filtering with temporal dynamics. *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 447–456).

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* (pp. 1097–1105).

Kung, S. Y. (1988). Vlsi array processors. *Englewood Cliffs, NJ, Prentice Hall, 1988, 685 p. Research supported by the Semiconductor Research Corp., SDIO, NSF, and US Navy.*

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., & others. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278–2324.

Li, M. (2017). *Scaling Distributed Machine Learning with System and Algorithm Co-design* (Doctoral dissertation). PhD Thesis, CMU.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., … Su, B.-Y. (2014). Scaling distributed machine learning with the parameter server. *11th $\$USENIX$\$ Symposium on Operating Systems Design and Implementation ($\$OSDI$\$ 14)* (pp. 583–598).

Lin, M., Chen, Q., & Yan, S. (2013). Network in network. *arXiv preprint arXiv:1312.4400*.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).

Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., … others. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. *Large scale visual recognition challenge*.

Lipton, Z. C., & Steinhardt, J. (2018). Troubling trends in machine learning scholarship. *arXiv preprint arXiv:1807.03341*.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). Ssd: single shot multibox detector. *European conference on computer vision* (pp. 21–37).

Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 3431–3440).

Loshchilov, I., & Hutter, F. (2016). Sgdr: stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*.

Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, *60*(2), 91–110.

Luo, P., Wang, X., Shao, W., & Peng, Z. (2018). Towards understanding regularization in batch normalization. *arXiv preprint*.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1* (pp. 142–150).

McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: contextualized word vectors. *Advances in Neural Information Processing Systems* (pp. 6294–6305).

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133.

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* (pp. 3111–3119).

Mirhoseini, A., Pham, H., Le, Q. V., Steiner, B., Larsen, R., Zhou, Y., … Dean, J. (2017). Device placement optimization with reinforcement learning. *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (pp. 2430–2439).

Morey, R. D., Hoekstra, R., Rouder, J. N., Lee, M. D., & Wagenmakers, E.-J. (2016). The fallacy of placing confidence in confidence intervals. *Psychonomic bulletin & review*, *23*(1), 103–123.

Nesterov, Y., & Vial, J.-P. (2000). *Confidence level solutions for stochastic programming, Stochastic Programming E-Print Series*.

Nesterov, Y. (2018). *Lectures on convex optimization*. Vol. 137. Springer.

Neyman, J. (1937). Outline of a theory of statistical estimation based on the classical theory of probability. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, *236*(767), 333–380.

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. *Proceedings of the 40th annual meeting of the Association for Computational Linguistics* (pp. 311–318).

Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.

Park, T., Liu, M.-Y., Wang, T.-C., & Zhu, J.-Y. (2019). Semantic image synthesis with spatially-adaptive normalization. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 2337–2346).

Pennington, J., Schoenholz, S., & Ganguli, S. (2017). Resurrecting the sigmoid in deep learning through dynamical isometry: theory and practice. *Advances in neural information processing systems* (pp. 4785–4795).

Pennington, J., Socher, R., & Manning, C. (2014). Glove: global vectors for word representation. *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)* (pp. 1532–1543).

Peters, J., Janzing, D., & Schölkopf, B. (2017). *Elements of causal inference: foundations and learning algorithms*. MIT press.

Peters, M., Ammar, W., Bhagavatula, C., & Power, R. (2017). Semi-supervised sequence tagging with bidirectional language models. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1756–1765).

Peters, M., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)* (pp. 2227–2237).

Petersen, K. B., Pedersen, M. S., & others. (2008). The matrix cookbook. *Technical University of Denmark*, *7*(15), 510.

Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, *4*(5), 1–17.

Quadrana, M., Cremonesi, P., & Jannach, D. (2018). Sequence-aware recommender systems. *ACM Computing Surveys (CSUR)*, *51*(4), 66.

Radford, A., Metz, L., & Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*.

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

Rajpurkar, P., Zhang, J., Lopyrev, K., & Liang, P. (2016). Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Reddi, S. J., Kale, S., & Kumar, S. (2019). On the convergence of adam and beyond. *arXiv preprint arXiv:1904.09237*.

Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*.

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: towards real-time object detection with region proposal networks. *Advances in neural information processing systems* (pp. 91–99).

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Santurkar, S., Tsipras, D., Ilyas, A., & Madry, A. (2018). How does batch normalization help optimization? *Advances in Neural Information Processing Systems* (pp. 2483–2493).

Sarwar, B. M., Karypis, G., Konstan, J. A., Riedl, J., & others. (2001). Item-based collaborative filtering recommendation algorithms. *Www*, *1*, 285–295.

Schein, A. I., Popescul, A., Ungar, L. H., & Pennock, D. M. (2002). Methods and metrics for cold-start recommendations. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (pp. 253–260).

Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681.

Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*.

Shannon, C. E. (1948 , 7). A mathematical theory of communication. *The Bell System Technical Journal*, *27*(3), 379–423.

Shao, H., Yao, S., Sun, D., Zhang, A., Liu, S., Liu, D., … Abdelzaher, T. (2020). Controlvae: controllable variational autoencoder. *Proceedings of the 37th International Conference on Machine Learning*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., … others. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, *529*(7587), 484.

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Smola, A., & Narayanamurthy, S. (2010). An architecture for parallel topic models. *Proceedings of the VLDB Endowment*, *3*(1-2), 703–710.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

Strang, G. (1993). *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA.

Su, X., & Khoshgoftaar, T. M. (2009). A survey of collaborative filtering techniques. *Advances in artificial intelligence*, *2009*.

Sukhbaatar, S., Weston, J., Fergus, R., & others. (2015). End-to-end memory networks. *Advances in neural information processing systems* (pp. 2440–2448).

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *International conference on machine learning* (pp. 1139–1147).

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in neural information processing systems* (pp. 3104–3112).

Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. *Thirty-First AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1–9).

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 2818–2826).

Tallec, C., & Ollivier, Y. (2017). Unbiasing truncated backpropagation through time. *arXiv preprint arXiv:1705.08209*.

Teye, M., Azizpour, H., & Smith, K. (2018). Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*.

Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, *4*(2), 26–31.

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *59*(236), 433.

Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. *International journal of computer vision*, *104*(2), 154–171.

Van Loan, C. F., & Golub, G. H. (1983). *Matrix computations*. Johns Hopkins University Press.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., … Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems* (pp. 5998–6008).

Wang, L., Li, M., Liberty, E., & Smola, A. J. (2018). Optimal message scheduling for aggregation. *NETWORKS*, *2*(3), 2–3.

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., & Owens, J. D. (2016). Gunrock: a high-performance graph processing library on the gpu. *ACM SIGPLAN Notices* (p. 11).

Warstadt, A., Singh, A., & Bowman, S. R. (2019). Neural network acceptability judgments. *Transactions of the Association for Computational Linguistics*, *7*, 625–641.

Wasserman, L. (2013). *All of statistics: a concise course in statistical inference*. Springer Science & Business Media.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Welling, M., & Teh, Y. W. (2011). Bayesian learning via stochastic gradient langevin dynamics. *Proceedings of the 28th international conference on machine learning (ICML-11)* (pp. 681–688).

Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560.

Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. *Ann. Math* (pp. 325–327).

Wood, F., Gasthaus, J., Archambeau, C., James, L., & Teh, Y. W. (2011). The sequence memoizer. *Communications of the ACM*, *54*(2), 91–98.

Wu, C.-Y., Ahmed, A., Beutel, A., Smola, A. J., & Jing, H. (2017). Recurrent recommender networks. *Proceedings of the tenth ACM international conference on web search and data mining* (pp. 495–503).

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., … others. (2016). Google's neural machine translation system: bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*.

Xiao, L., Bahri, Y., Sohl-Dickstein, J., Schoenholz, S., & Pennington, J. (2018). Dynamical isometry and a mean field theory of cnns: how to train 10,000-layer vanilla convolutional neural networks. *International Conference on Machine Learning* (pp. 5393–5402).

Xiong, W., Wu, L., Alleva, F., Droppo, J., Huang, X., & Stolcke, A. (2018). The microsoft 2017 conversational speech recognition system. *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5934–5938).

Ye, M., Yin, P., Lee, W.-C., & Lee, D.-L. (2011). Exploiting geographical influence for collaborative point-of-interest recommendation. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (pp. 325–334).

You, Y., Gitman, I., & Ginsburg, B. (2017). Large batch training of convolutional networks. *arXiv preprint arXiv:1708.03888*.

Zaheer, M., Reddi, S., Sachan, D., Kale, S., & Kumar, S. (2018). Adaptive methods for nonconvex optimization. *Advances in Neural Information Processing Systems* (pp. 9793–9803).

Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

Zhang, S., Yao, L., Sun, A., & Tay, Y. (2019). Deep learning based recommender system: a survey and new perspectives. *ACM Computing Surveys (CSUR)*, *52*(1), 5.

Zhu, J.-Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. *Proceedings of the IEEE international conference on computer vision* (pp. 2223–2232).

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Aligning books and movies: towards story-like visual explanations by watching movies and reading books. *Proceedings of the IEEE international conference on computer vision* (pp. 19–27).

Python Module Index

d

d2l.torch, 896

Index

A

abs() (*in module d2l.torch*), 905
Accumulator (*class in d2l.torch*), 896
accuracy() (*in module d2l.torch*), 905
add() (*d2l.torch.Accumulator method*), 896
add() (*d2l.torch.Animator method*), 897
AdditiveAttention (*class in d2l.torch*), 897
AddNorm (*class in d2l.torch*), 896
Animator (*class in d2l.torch*), 897
annotate() (*in module d2l.torch*), 905
arange() (*in module d2l.torch*), 905
argmax() (*in module d2l.torch*), 906
assign_anchor_to_bbox() (*in module d2l.torch*), 906
astype() (*in module d2l.torch*), 906
attention_weights (*d2l.torch.AttentionDecoder property*), 897
AttentionDecoder (*class in d2l.torch*), 897
avg() (*d2l.torch.Timer method*), 904

B

BananasDataset (*class in d2l.torch*), 898
batchify() (*in module d2l.torch*), 906
bbox_to_rect() (*in module d2l.torch*), 906
begin_state() (*d2l.torch.RNNModel method*), 902
begin_state() (*d2l.torch.RNNModelScratch method*), 903
Benchmark (*class in d2l.torch*), 898
BERTEncoder (*class in d2l.torch*), 897
BERTModel (*class in d2l.torch*), 898
bleu() (*in module d2l.torch*), 906
box_center_to_corner() (*in module d2l.torch*), 907
box_corner_to_center() (*in module d2l.torch*), 907
box_iou() (*in module d2l.torch*), 907
build_array_nmt() (*in module d2l.torch*), 907

C

concat() (*in module d2l.torch*), 907
copyfile() (*in module d2l.torch*), 908
corr2d() (*in module d2l.torch*), 908
cos() (*in module d2l.torch*), 908
cosh() (*in module d2l.torch*), 908
count_corpus() (*in module d2l.torch*), 908
cumsum() (*d2l.torch.Timer method*), 904

D

d2l.torch
    module, 896
Decoder (*class in d2l.torch*), 898
DotProductAttention (*class in d2l.torch*), 899
download() (*in module d2l.torch*), 909
download_all() (*in module d2l.torch*), 909
download_extract() (*in module d2l.torch*), 909
draw() (*d2l.torch.RandomGenerator method*), 903

E

Encoder (*class in d2l.torch*), 899
EncoderBlock (*class in d2l.torch*), 899
EncoderDecoder (*class in d2l.torch*), 900
evaluate_accuracy() (*in module d2l.torch*), 909
evaluate_accuracy_gpu() (*in module d2l.torch*), 909
evaluate_loss() (*in module d2l.torch*), 909
exp() (*in module d2l.torch*), 909
eye() (*in module d2l.torch*), 909

F

filter() (*d2l.torch.VOCSegDataset method*), 905
forward() (*d2l.torch.AdditiveAttention method*), 897
forward() (*d2l.torch.AddNorm method*), 896
forward() (*d2l.torch.BERTEncoder method*), 897
forward() (*d2l.torch.BERTModel method*), 898
forward() (*d2l.torch.Decoder method*), 898
forward() (*d2l.torch.DotProductAttention method*), 899
forward() (*d2l.torch.Encoder method*), 899
forward() (*d2l.torch.EncoderBlock method*), 899
forward() (*d2l.torch.EncoderDecoder method*), 900

forward() (*d2l.torch.MaskedSoftmaxCELoss method*), 901
forward() (*d2l.torch.MaskLM method*), 900
forward() (*d2l.torch.MultiHeadAttention method*), 901
forward() (*d2l.torch.NextSentencePred method*), 901
forward() (*d2l.torch.PositionalEncoding method*), 902
forward() (*d2l.torch.PositionWiseFFN method*), 902
forward() (*d2l.torch.Residual method*), 903
forward() (*d2l.torch.RNNModel method*), 902
forward() (*d2l.torch.Seq2SeqEncoder method*), 903
forward() (*d2l.torch.TransformerEncoder method*), 904

G

get_centers_and_contexts() (*in module d2l.torch*), 910
get_data_ch11() (*in module d2l.torch*), 910
get_dataloader_workers() (*in module d2l.torch*), 910
get_fashion_mnist_labels() (*in module d2l.torch*), 910
get_negatives() (*in module d2l.torch*), 910
get_tokens_and_segments() (*in module d2l.torch*), 910
grad_clipping() (*in module d2l.torch*), 910

I

ignore_index (*d2l.torch.MaskedSoftmaxCELoss attribute*), 901
init_state() (*d2l.torch.Decoder method*), 899

L

label_smoothing (*d2l.torch.MaskedSoftmaxCELoss attribute*), 901
linreg() (*in module d2l.torch*), 910
linspace() (*in module d2l.torch*), 911
load_array() (*in module d2l.torch*), 911
load_corpus_time_machine() (*in module d2l.torch*), 911
load_data_bananas() (*in module d2l.torch*), 911
load_data_fashion_mnist() (*in module d2l.torch*), 912
load_data_imdb() (*in module d2l.torch*), 912
load_data_nmt() (*in module d2l.torch*), 912
load_data_ptb() (*in module d2l.torch*), 912
load_data_snli() (*in module d2l.torch*), 912
load_data_time_machine() (*in module d2l.torch*), 912
load_data_voc() (*in module d2l.torch*), 912
load_data_wiki() (*in module d2l.torch*), 912
log() (*in module d2l.torch*), 912

M

masked_softmax() (*in module d2l.torch*), 913
MaskedSoftmaxCELoss (*class in d2l.torch*), 900
MaskLM (*class in d2l.torch*), 900
matmul() (*in module d2l.torch*), 913
module
    d2l.torch, 896
multibox_detection() (*in module d2l.torch*), 914
multibox_prior() (*in module d2l.torch*), 914
multibox_target() (*in module d2l.torch*), 914
MultiHeadAttention (*class in d2l.torch*), 901

N

NextSentencePred (*class in d2l.torch*), 901
nms() (*in module d2l.torch*), 914
normal() (*in module d2l.torch*), 914
normalize_image() (*d2l.torch.VOCSegDataset method*), 905
numpy() (*in module d2l.torch*), 916

O

offset_boxes() (*in module d2l.torch*), 916
offset_inverse() (*in module d2l.torch*), 916
ones() (*in module d2l.torch*), 916

P

plot() (*in module d2l.torch*), 916
PositionalEncoding (*class in d2l.torch*), 902
PositionWiseFFN (*class in d2l.torch*), 902
predict_ch3() (*in module d2l.torch*), 917
predict_ch8() (*in module d2l.torch*), 917
predict_sentiment() (*in module d2l.torch*), 917
predict_seq2seq() (*in module d2l.torch*), 917
predict_snli() (*in module d2l.torch*), 917
preprocess_nmt() (*in module d2l.torch*), 917

R

rand() (*in module d2l.torch*), 917
RandomGenerator (*class in d2l.torch*), 903
read_csv_labels() (*in module d2l.torch*), 918
read_data_bananas() (*in module d2l.torch*), 918

read_data_nmt() (*in module d2l.torch*), 918
read_imdb() (*in module d2l.torch*), 918
read_ptb() (*in module d2l.torch*), 918
read_snli() (*in module d2l.torch*), 918
read_time_machine() (*in module d2l.torch*), 918
read_voc_images() (*in module d2l.torch*), 918
reduce_sum() (*in module d2l.torch*), 918
reorg_test() (*in module d2l.torch*), 918
reorg_train_valid() (*in module d2l.torch*), 919
reset() (*d2l.torch.Accumulator method*), 896
reshape() (*in module d2l.torch*), 919
Residual (*class in d2l.torch*), 903
resnet18() (*in module d2l.torch*), 919
RNNModel (*class in d2l.torch*), 902
RNNModelScratch (*class in d2l.torch*), 903

S

Seq2SeqEncoder (*class in d2l.torch*), 903
seq_data_iter_random() (*in module d2l.torch*), 919
seq_data_iter_sequential() (*in module d2l.torch*), 919
SeqDataLoader (*class in d2l.torch*), 904
sequence_mask() (*in module d2l.torch*), 919
set_axes() (*in module d2l.torch*), 919
set_figsize() (*in module d2l.torch*), 919
sgd() (*in module d2l.torch*), 919
show_bboxes() (*in module d2l.torch*), 919
show_heatmaps() (*in module d2l.torch*), 919
show_images() (*in module d2l.torch*), 919
show_list_len_pair_hist() (*in module d2l.torch*), 920
show_trace_2d() (*in module d2l.torch*), 920
sin() (*in module d2l.torch*), 920
sinh() (*in module d2l.torch*), 920
size() (*in module d2l.torch*), 920
SNLIDataset (*class in d2l.torch*), 903
split_batch() (*in module d2l.torch*), 920
squared_loss() (*in module d2l.torch*), 921
stack() (*in module d2l.torch*), 921
start() (*d2l.torch.Timer method*), 904
stop() (*d2l.torch.Timer method*), 904
subsample() (*in module d2l.torch*), 921
sum() (*d2l.torch.Timer method*), 904
synthetic_data() (*in module d2l.torch*), 921

T

tanh() (*in module d2l.torch*), 921
tensor() (*in module d2l.torch*), 921
Timer (*class in d2l.torch*), 904
to() (*in module d2l.torch*), 922
to_tokens() (*d2l.torch.Vocab method*), 905
token_freqs (*d2l.torch.Vocab property*), 905
TokenEmbedding (*class in d2l.torch*), 904
tokenize() (*in module d2l.torch*), 922
tokenize_nmt() (*in module d2l.torch*), 923
train_2d() (*in module d2l.torch*), 923
train_batch_ch13() (*in module d2l.torch*), 923
train_ch3() (*in module d2l.torch*), 923
train_ch6() (*in module d2l.torch*), 923
train_ch8() (*in module d2l.torch*), 923
train_ch11() (*in module d2l.torch*), 923
train_ch13() (*in module d2l.torch*), 923
train_concise_ch11() (*in module d2l.torch*), 923
train_epoch_ch3() (*in module d2l.torch*), 923
train_epoch_ch8() (*in module d2l.torch*), 923
train_seq2seq() (*in module d2l.torch*), 923
training (*d2l.torch.AdditiveAttention attribute*), 897
training (*d2l.torch.AddNorm attribute*), 897
training (*d2l.torch.AttentionDecoder attribute*), 897
training (*d2l.torch.BERTEncoder attribute*), 898
training (*d2l.torch.BERTModel attribute*), 898
training (*d2l.torch.Decoder attribute*), 899
training (*d2l.torch.DotProductAttention attribute*), 899
training (*d2l.torch.Encoder attribute*), 899
training (*d2l.torch.EncoderBlock attribute*), 900
training (*d2l.torch.EncoderDecoder attribute*), 900
training (*d2l.torch.MaskLM attribute*), 900
training (*d2l.torch.MultiHeadAttention attribute*), 901
training (*d2l.torch.NextSentencePred attribute*), 901
training (*d2l.torch.PositionalEncoding attribute*), 902
training (*d2l.torch.PositionWiseFFN attribute*), 902
training (*d2l.torch.Residual attribute*), 903
training (*d2l.torch.RNNModel attribute*), 903
training (*d2l.torch.Seq2SeqEncoder attribute*), 904
training (*d2l.torch.TransformerEncoder attribute*), 905
TransformerEncoder (*class in d2l.torch*), 904
transpose() (*in module d2l.torch*), 924
transpose_output() (*in module d2l.torch*), 924

transpose_qkv() (*in module d2l.torch*), 924
truncate_pad() (*in module d2l.torch*), 924
try_all_gpus() (*in module d2l.torch*), 924
try_gpu() (*in module d2l.torch*), 924

U

unk (*d2l.torch.Vocab property*), 905
update_D() (*in module d2l.torch*), 924
update_G() (*in module d2l.torch*), 924
use_svg_display() (*in module d2l.torch*), 924

V

voc_colormap2label() (*in module d2l.torch*), 924
voc_label_indices() (*in module d2l.torch*), 924
voc_rand_crop() (*in module d2l.torch*), 924
Vocab (*class in d2l.torch*), 905
VOCSegDataset (*class in d2l.torch*), 905

Z

zeros() (*in module d2l.torch*), 925