



# Đắm mình vào Học Sâu

*Release 0.14.4*

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



# Contents

<b>Giới thiệu từ nhóm dịch</b>	<b>1</b>
<b>Lời nói đầu</b>	<b>3</b>
<b>1 Cài đặt</b>	<b>11</b>
1.1 Cài đặt Miniconda . . . . .	11
1.2 Tải về notebook của D2L . . . . .	11
1.3 Cài đặt Framework và Gói thư viện d2l . . . . .	12
1.4 Hỗ trợ GPU . . . . .	13
1.5 Bài tập . . . . .	14
1.6 Thảo luận . . . . .	14
1.7 Những người thực hiện . . . . .	14
<b>2 Ký hiệu</b>	<b>15</b>
2.1 Số . . . . .	15
2.2 Lý thuyết Tập hợp . . . . .	15
2.3 Hàm số và các Phép toán . . . . .	16
2.4 Giải tích . . . . .	16
2.5 Xác suất và Lý thuyết Thông tin . . . . .	16
2.6 Độ phức tạp . . . . .	17
2.7 Thảo luận . . . . .	17
2.8 Những người thực hiện . . . . .	17
<b>3 Giới thiệu</b>	<b>19</b>
3.1 Một ví dụ truyền cảm hứng . . . . .	20
3.2 Các thành phần chính: Dữ liệu, Mô hình và Thuật toán . . . . .	22
3.2.1 Dữ liệu . . . . .	23
3.2.2 Mô hình . . . . .	24
3.2.3 Hàm mục tiêu . . . . .	24
3.2.4 Các thuật toán tối ưu . . . . .	25
3.3 Các dạng Học Máy . . . . .	25
3.3.1 Học có giám sát . . . . .	25
3.3.2 Học không giám sát . . . . .	34
3.3.3 Tương tác với Môi trường . . . . .	35
3.3.4 Học tăng cường . . . . .	36
3.4 Nguồn gốc . . . . .	37
3.5 Con đường tới Học Sâu . . . . .	40
3.6 Các câu chuyện thành công . . . . .	42
3.7 Tóm tắt . . . . .	44
3.8 Bài tập . . . . .	44
3.9 Thảo luận . . . . .	44

3.10	Những người thực hiện . . . . .	45
<b>4</b>	<b>Sơ bộ</b>	<b>47</b>
4.1	Thao tác với Dữ liệu . . . . .	47
4.1.1	Bắt đầu . . . . .	48
4.1.2	Phép toán . . . . .	50
4.1.3	Cơ chế Lan truyền . . . . .	51
4.1.4	Chỉ số và Cắt chọn mảng . . . . .	51
4.1.5	Tiết kiệm Bộ nhớ . . . . .	52
4.1.6	Chuyển đổi sang các Đối Tượng Python Khác . . . . .	53
4.1.7	Tổng kết . . . . .	53
4.1.8	Bài tập . . . . .	53
4.1.9	Thảo luận . . . . .	53
4.1.10	Những người thực hiện . . . . .	54
4.2	Tiền xử lý dữ liệu . . . . .	54
4.2.1	Đọc tập dữ liệu . . . . .	54
4.2.2	Xử lý dữ liệu thiếu . . . . .	55
4.2.3	Chuyển sang định dạng ndarray . . . . .	56
4.2.4	Tóm tắt . . . . .	56
4.2.5	Bài tập . . . . .	56
4.2.6	Thảo luận . . . . .	56
4.2.7	Những người thực hiện . . . . .	56
4.3	Đại số tuyến tính . . . . .	57
4.3.1	Số vô hướng . . . . .	57
4.3.2	Vector . . . . .	57
4.3.3	Ma trận . . . . .	59
4.3.4	Tensor . . . . .	60
4.3.5	Các thuộc tính Cơ bản của Phép toán Tensor . . . . .	60
4.3.6	Rút gọn . . . . .	61
4.3.7	Tích vô hướng . . . . .	62
4.3.8	Tích giữa Ma trận và Vector . . . . .	63
4.3.9	Phép nhân Ma trận . . . . .	63
4.3.10	Chuẩn . . . . .	64
4.3.11	Bàn thêm về Đại số Tuyến tính . . . . .	66
4.3.12	Tóm tắt . . . . .	66
4.3.13	Bài tập . . . . .	67
4.3.14	Thảo luận . . . . .	67
4.3.15	Những người thực hiện . . . . .	67
4.4	Giải tích . . . . .	68
4.4.1	Đạo hàm và Vi phân . . . . .	68
4.4.2	Đạo hàm riêng . . . . .	71
4.4.3	Gradient . . . . .	72
4.4.4	Quy tắc dây chuyền . . . . .	72
4.4.5	Tóm tắt . . . . .	73
4.4.6	Bài tập . . . . .	73
4.4.7	Thảo luận . . . . .	73
4.4.8	Những người thực hiện . . . . .	73
4.5	Tính vi phân Tự động . . . . .	74
4.5.1	Một ví dụ đơn giản . . . . .	74
4.5.2	Truyền ngược cho các biến không phải Số vô hướng . . . . .	75
4.5.3	Tách rời Tính toán . . . . .	76

4.5.4	Tính gradient của Luồng điều khiển Python . . . . .	77
4.5.5	Chế độ huấn luyện và Chế độ dự đoán . . . . .	77
4.5.6	Tóm tắt . . . . .	78
4.5.7	Bài tập . . . . .	78
4.5.8	Thảo luận . . . . .	78
4.5.9	Những người thực hiện . . . . .	79
4.6	Xác suất . . . . .	79
4.6.1	Lý thuyết Xác suất cơ bản . . . . .	80
4.6.2	Làm việc với Nhiều Biến Ngẫu nhiên . . . . .	83
4.6.3	Kỳ vọng và Phương sai . . . . .	86
4.6.4	Tóm tắt . . . . .	87
4.6.5	Bài tập . . . . .	87
4.6.6	Thảo luận . . . . .	87
4.6.7	Những người thực hiện . . . . .	88
4.7	Tài liệu . . . . .	88
4.7.1	Tra cứu tất cả các hàm và lớp trong một Mô-đun . . . . .	88
4.7.2	Tra cứu cách sử dụng một hàm hoặc một lớp cụ thể . . . . .	89
4.7.3	Tài liệu API . . . . .	89
4.7.4	Tóm tắt . . . . .	89
4.7.5	Bài tập . . . . .	89
4.7.6	Thảo luận . . . . .	90
4.7.7	Những người thực hiện . . . . .	90
4.8	Những người thực hiện . . . . .	90
<b>5</b>	<b>Mạng nơ-ron Tuyến tính</b>	<b>91</b>
5.1	Hồi quy Tuyến tính . . . . .	91
5.1.1	Các Thành phần Cơ bản của Hồi quy Tuyến tính . . . . .	91
5.1.2	Phân phối Chuẩn và Hàm mất mát Bình phương . . . . .	97
5.1.3	Từ Hồi quy Tuyến tính tới Mạng Học sâu . . . . .	98
5.1.4	Tóm tắt . . . . .	100
5.1.5	Bài tập . . . . .	100
5.1.6	Thảo luận . . . . .	101
5.1.7	Những người thực hiện . . . . .	101
5.2	Lập trình Hồi quy Tuyến tính từ đầu . . . . .	102
5.2.1	Tạo tập dữ liệu . . . . .	102
5.2.2	Đọc từ Tập dữ liệu . . . . .	103
5.2.3	Khởi tạo các Tham số Mô hình . . . . .	104
5.2.4	Định nghĩa Mô hình . . . . .	104
5.2.5	Định nghĩa Hàm Mất mát . . . . .	104
5.2.6	Định nghĩa Thuật toán Tối ưu . . . . .	105
5.2.7	Huấn luyện . . . . .	105
5.2.8	Tóm tắt . . . . .	106
5.2.9	Bài tập . . . . .	107
5.2.10	Thảo luận . . . . .	107
5.2.11	Những người thực hiện . . . . .	107
5.3	Cách lập trình súc tích Hồi quy Tuyến tính . . . . .	108
5.3.1	Tạo Tập dữ liệu . . . . .	108
5.3.2	Đọc tập dữ liệu . . . . .	108
5.3.3	Định nghĩa Mô hình . . . . .	109
5.3.4	Khởi tạo Tham số Mô hình . . . . .	110
5.3.5	Định nghĩa Hàm mất mát . . . . .	110

5.3.6	Định nghĩa Thuật toán Tối ưu . . . . .	111
5.3.7	Huấn luyện . . . . .	111
5.3.8	Tóm tắt . . . . .	112
5.3.9	Bài tập . . . . .	112
5.3.10	Thảo luận . . . . .	112
5.3.11	Những người thực hiện . . . . .	112
5.4	Hồi quy Softmax . . . . .	113
5.4.1	Bài toán Phân loại . . . . .	113
5.4.2	Hàm mất mát . . . . .	116
5.4.3	Lý thuyết Thông tin Cơ bản . . . . .	117
5.4.4	Sử dụng Mô hình để Dự đoán và Đánh giá . . . . .	118
5.4.5	Tóm tắt . . . . .	119
5.4.6	Bài tập . . . . .	119
5.4.7	Thảo luận . . . . .	120
5.4.8	Những người thực hiện . . . . .	120
5.5	Bộ dữ liệu Phân loại Ảnh (Fashion-MNIST) . . . . .	120
5.5.1	Tài về Bộ dữ liệu . . . . .	121
5.5.2	Đọc một Minibatch . . . . .	122
5.5.3	Kết hợp Tất cả lại với nhau . . . . .	123
5.5.4	Tóm tắt . . . . .	123
5.5.5	Bài tập . . . . .	124
5.5.6	Thảo luận . . . . .	124
5.5.7	Những người thực hiện . . . . .	124
5.6	Lập trình Hồi quy Sofmax từ đầu . . . . .	124
5.6.1	Khởi tạo các Tham số của Mô hình . . . . .	125
5.6.2	Softmax . . . . .	125
5.6.3	Mô hình . . . . .	126
5.6.4	Hàm mất mát . . . . .	126
5.6.5	Độ chính xác cho bài toán Phân loại . . . . .	127
5.6.6	Huấn luyện Mô hình . . . . .	128
5.6.7	Dự đoán . . . . .	130
5.6.8	Tóm tắt . . . . .	130
5.6.9	Bài tập . . . . .	130
5.6.10	Thảo luận . . . . .	131
5.6.11	Những người thực hiện . . . . .	131
5.7	Cách lập trình súc tích Hồi quy Softmax . . . . .	131
5.7.1	Khởi tạo Tham số Mô hình . . . . .	132
5.7.2	Hàm Softmax . . . . .	132
5.7.3	Thuật toán Tối ưu . . . . .	133
5.7.4	Huấn luyện . . . . .	133
5.7.5	Bài tập . . . . .	133
5.7.6	Thảo luận . . . . .	133
5.7.7	Những người thực hiện . . . . .	134
5.8	Những người thực hiện . . . . .	134
<b>6</b>	<b>Perceptron Đa tầng</b>	<b>135</b>
6.1	Perceptron đa tầng . . . . .	135
6.1.1	Các tầng ẩn . . . . .	135
6.1.2	Các hàm Kích hoạt . . . . .	139
6.1.3	Tóm tắt . . . . .	141
6.1.4	Bài tập . . . . .	142

6.1.5	Thảo luận . . . . .	142
6.1.6	Những người thực hiện . . . . .	142
6.2	Lập trình Perceptron Đa tầng từ đầu . . . . .	143
6.2.1	Khởi tạo Tham số Mô hình . . . . .	143
6.2.2	Hàm Kích hoạt . . . . .	144
6.2.3	Mô hình . . . . .	144
6.2.4	Hàm mất mát . . . . .	144
6.2.5	Huấn luyện . . . . .	144
6.2.6	Tóm tắt . . . . .	145
6.2.7	Bài tập . . . . .	145
6.2.8	Thảo luận . . . . .	145
6.2.9	Những người thực hiện . . . . .	145
6.3	Cách lập trình súc tích Perceptron Đa tầng . . . . .	146
6.3.1	Mô hình . . . . .	146
6.3.2	Bài tập . . . . .	146
6.3.3	Thảo luận . . . . .	147
6.3.4	Những người thực hiện . . . . .	147
6.4	Lựa Chọn Mô Hình, Dưới Khớp và Quá Khớp . . . . .	147
6.4.1	Lỗi huấn luyện và Lỗi khái quát . . . . .	148
6.4.2	Lựa chọn Mô hình . . . . .	150
6.4.3	Dưới khớp hay Quá khớp? . . . . .	151
6.4.4	Hồi quy Đa thức . . . . .	153
6.4.5	Tóm tắt . . . . .	156
6.4.6	Bài tập . . . . .	156
6.4.7	Thảo luận . . . . .	156
6.4.8	Những người thực hiện . . . . .	156
6.5	Suy giảm trọng số . . . . .	157
6.5.1	Điều chuẩn Chuẩn Bình phương . . . . .	157
6.5.2	Hồi quy Tuyến tính nhiều chiều . . . . .	159
6.5.3	Lập trình từ đầu . . . . .	159
6.5.4	Cách lập trình súc tích . . . . .	161
6.5.5	Tóm tắt . . . . .	162
6.5.6	Bài tập . . . . .	162
6.5.7	Thảo luận . . . . .	163
6.5.8	Những người thực hiện . . . . .	163
6.6	Dropout . . . . .	163
6.6.1	Bàn lại về Quá khớp . . . . .	164
6.6.2	Khả năng Kháng Nghiễu . . . . .	164
6.6.3	Dropout trong Thực tiễn . . . . .	165
6.6.4	Lập trình từ đầu . . . . .	166
6.6.5	Cách lập trình súc tích . . . . .	168
6.6.6	Tóm tắt . . . . .	168
6.6.7	Bài tập . . . . .	169
6.6.8	Thảo luận . . . . .	169
6.6.9	Những người thực hiện . . . . .	169
6.7	Lan truyền xuôi, Lan truyền ngược và Đồ thị tính toán . . . . .	170
6.7.1	Lan truyền Xuôi . . . . .	170
6.7.2	Đồ thị Tính toán của Lan truyền Xuôi . . . . .	171
6.7.3	Lan truyền Ngược . . . . .	171
6.7.4	Huấn luyện một Mô hình . . . . .	172
6.7.5	Tóm tắt . . . . .	173

6.7.6	Bài tập . . . . .	173
6.7.7	Thảo luận . . . . .	173
6.7.8	Những người thực hiện . . . . .	173
6.8	Ôn định Số học và Khởi tạo . . . . .	174
6.8.1	Tiêu biến và Bùng nổ Gradient . . . . .	174
6.8.2	Khởi tạo Tham số . . . . .	176
6.8.3	Tóm tắt . . . . .	178
6.8.4	Bài tập . . . . .	178
6.8.5	Thảo luận . . . . .	178
6.8.6	Những người thực hiện . . . . .	178
6.9	Cân nhắc tới Môi trường . . . . .	179
6.9.1	Dịch chuyển Phân phối . . . . .	179
6.9.2	Phân loại các Bài toán Học máy . . . . .	186
6.9.3	Công bằng, Trách nhiệm và Minh bạch trong Học máy . . . . .	187
6.9.4	Tóm tắt . . . . .	188
6.9.5	Bài tập . . . . .	188
6.9.6	Thảo luận . . . . .	188
6.9.7	Những người thực hiện . . . . .	188
6.10	Dự đoán Giá Nhà trên Kaggle . . . . .	189
6.10.1	Tài và Lưu trữ Bộ dữ liệu . . . . .	189
6.10.2	Kaggle . . . . .	190
6.10.3	Truy cập và Đọc Bộ dữ liệu . . . . .	191
6.10.4	Tiền xử lý Dữ liệu . . . . .	193
6.10.5	Huấn luyện . . . . .	194
6.10.6	Kiểm định chéo gập k-lần . . . . .	195
6.10.7	Lựa chọn Mô hình . . . . .	196
6.10.8	Dự đoán và Nộp bài . . . . .	196
6.10.9	Tóm tắt . . . . .	197
6.10.10	Bài tập . . . . .	198
6.10.11	Thảo luận . . . . .	198
6.10.12	Những người thực hiện . . . . .	198
6.11	Những người thực hiện . . . . .	199
<b>7</b>	<b>Tính toán Học sâu</b>	<b>201</b>
7.1	Tầng và Khối . . . . .	201
7.1.1	Một Khối Tùy chỉnh . . . . .	203
7.1.2	Khối Tuần tự . . . . .	204
7.1.3	Thực thi Mã trong Phương thức forward . . . . .	205
7.1.4	Biên dịch Mã nguồn . . . . .	207
7.1.5	Tóm tắt . . . . .	207
7.1.6	Bài tập . . . . .	207
7.1.7	Thảo luận . . . . .	208
7.1.8	Những người thực hiện . . . . .	208
7.2	Quản lý Tham số . . . . .	208
7.2.1	Truy cập Tham số . . . . .	209
7.2.2	Khởi tạo Tham số . . . . .	211
7.2.3	Các Tham số bị Trói buộc . . . . .	212
7.2.4	Tóm tắt . . . . .	213
7.2.5	Bài tập . . . . .	213
7.2.6	Thảo luận . . . . .	214
7.2.7	Những người thực hiện . . . . .	214

7.3	Khởi tạo trẽ . . . . .	214
7.3.1	Khởi tạo Mạng . . . . .	215
7.3.2	Khởi tạo Trẽ trong Thực tiễn . . . . .	216
7.3.3	Khởi tạo Cưỡng chế . . . . .	216
7.3.4	Tóm tắt . . . . .	217
7.3.5	Bài tập . . . . .	217
7.3.6	Thảo luận . . . . .	217
7.3.7	Những người thực hiện . . . . .	217
7.4	Các tầng Tuỳ chỉnh . . . . .	218
7.4.1	Các tầng không có Tham số . . . . .	218
7.4.2	Tầng có Tham số . . . . .	219
7.4.3	Tóm tắt . . . . .	220
7.4.4	Bài tập . . . . .	220
7.4.5	Thảo luận . . . . .	220
7.4.6	Những người thực hiện . . . . .	220
7.5	Đọc/Ghi tệp . . . . .	221
7.5.1	Đọc và Lưu các ndarray . . . . .	221
7.5.2	Tham số mô hình Gluon . . . . .	222
7.5.3	Tóm tắt . . . . .	223
7.5.4	Bài tập . . . . .	223
7.5.5	Thảo luận . . . . .	223
7.5.6	Những người thực hiện . . . . .	223
7.6	GPU . . . . .	224
7.6.1	Thiết bị Tính toán . . . . .	225
7.6.2	ndarray và GPU . . . . .	225
7.6.3	Gluon và GPU . . . . .	227
7.6.4	Tóm tắt . . . . .	228
7.6.5	Bài tập . . . . .	228
7.6.6	Thảo luận . . . . .	228
7.6.7	Những người thực hiện . . . . .	229
7.7	Những người thực hiện . . . . .	229
<b>8</b>	<b>Mạng Nơ-ron Tích chập</b>	<b>231</b>
8.1	Từ Tầng Kết nối Dày đặc đến phép Tích chập . . . . .	232
8.1.1	Tính Bất biến . . . . .	232
8.1.2	Ràng buộc Perceptron Đa tầng . . . . .	233
8.1.3	Phép Tích chập . . . . .	234
8.1.4	Xem lại ví dụ về Waldo . . . . .	235
8.1.5	Tóm tắt . . . . .	236
8.1.6	Bài tập . . . . .	236
8.1.7	Thảo luận . . . . .	236
8.1.8	Những người thực hiện . . . . .	236
8.2	Phép Tích chập cho Ảnh . . . . .	237
8.2.1	Toán tử Tương quan Chéo . . . . .	237
8.2.2	Tầng Tích chập . . . . .	238
8.2.3	Phát hiện Biên của Vật thể trong Ảnh . . . . .	239
8.2.4	Học một Bộ lọc . . . . .	239
8.2.5	Tương quan Chéo và Tích chập . . . . .	240
8.2.6	Tóm tắt . . . . .	240
8.2.7	Bài tập . . . . .	241
8.2.8	Thảo luận . . . . .	241

8.2.9	Những người thực hiện . . . . .	241
8.3	Đệm và Sai Bước . . . . .	242
8.3.1	Đệm . . . . .	242
8.3.2	Sai bước . . . . .	244
8.3.3	Tóm tắt . . . . .	245
8.3.4	Bài tập . . . . .	245
8.3.5	Thảo luận . . . . .	245
8.3.6	Những người thực hiện . . . . .	245
8.4	Đa kênh Đầu vào và Đầu ra . . . . .	246
8.4.1	Đa kênh Đầu vào . . . . .	246
8.4.2	Đa kênh Đầu ra . . . . .	247
8.4.3	Tầng Tích chập $1 \times 1$ . . . . .	248
8.4.4	Tóm tắt . . . . .	249
8.4.5	Bài tập . . . . .	249
8.4.6	Thảo luận . . . . .	250
8.4.7	Những người thực hiện . . . . .	250
8.5	Gộp ( <i>Pooling</i> ) . . . . .	250
8.5.1	Gộp cực đại và Gộp trung bình . . . . .	251
8.5.2	Đệm và Sai bước . . . . .	252
8.5.3	Với đầu vào Đa Kênh . . . . .	253
8.5.4	Tóm tắt . . . . .	253
8.5.5	Bài tập . . . . .	253
8.5.6	Thảo luận . . . . .	254
8.5.7	Những người thực hiện . . . . .	254
8.6	Mạng Nơ-ron Tích chập (LeNet) . . . . .	254
8.6.1	LeNet . . . . .	255
8.6.2	Thu thập Dữ liệu và Huấn luyện . . . . .	257
8.6.3	Tóm tắt . . . . .	259
8.6.4	Bài tập . . . . .	259
8.6.5	Thảo luận . . . . .	259
8.6.6	Những người thực hiện . . . . .	260
8.7	Những người thực hiện . . . . .	260
<b>9</b>	<b>Mạng Nơ-ron Tích chập Hiện đại</b>	<b>261</b>
9.1	Mạng Nơ-ron Tích chập Sâu (AlexNet) . . . . .	261
9.1.1	Học Biểu diễn Đặc trưng . . . . .	262
9.1.2	Mạng AlexNet . . . . .	265
9.1.3	Đọc Tập dữ liệu . . . . .	267
9.1.4	Huấn luyện . . . . .	268
9.1.5	Tóm tắt . . . . .	268
9.1.6	Bài tập . . . . .	268
9.1.7	Thảo luận . . . . .	269
9.1.8	Những người thực hiện . . . . .	269
9.2	Mạng sử dụng Khối (VGG) . . . . .	269
9.2.1	Khối VGG . . . . .	270
9.2.2	Mạng VGG . . . . .	270
9.2.3	Huấn luyện Mô hình . . . . .	272
9.2.4	Tóm tắt . . . . .	272
9.2.5	Bài tập . . . . .	272
9.2.6	Thảo luận . . . . .	273
9.2.7	Những người thực hiện . . . . .	273

9.3	Mạng trong Mạng ( <i>Network in Network - NiN</i> ) . . . . .	273
9.3.1	Khối NiN . . . . .	273
9.3.2	Mô hình NiN . . . . .	275
9.3.3	Thu thập Dữ liệu và Huấn luyện . . . . .	275
9.3.4	Tóm tắt . . . . .	276
9.3.5	Bài tập . . . . .	276
9.3.6	Thảo luận . . . . .	276
9.3.7	Những người thực hiện . . . . .	276
9.4	Mạng nối song song (GoogLeNet) . . . . .	277
9.4.1	Khối Inception . . . . .	277
9.4.2	Mô hình GoogLeNet . . . . .	278
9.4.3	Thu thập Dữ liệu và Huấn luyện . . . . .	281
9.4.4	Tóm tắt . . . . .	281
9.4.5	Bài tập . . . . .	281
9.4.6	Thảo luận . . . . .	282
9.4.7	Những người thực hiện . . . . .	282
9.5	Chuẩn hoá theo batch . . . . .	282
9.5.1	Huấn luyện mạng học sâu . . . . .	282
9.5.2	Tầng chuẩn hoá theo batch . . . . .	284
9.5.3	Lập trình từ đầu . . . . .	285
9.5.4	Sử dụng LeNet với Chuẩn hóa theo Batch . . . . .	286
9.5.5	Lập trình súc tích . . . . .	287
9.5.6	Tranh luận . . . . .	287
9.5.7	Tóm tắt . . . . .	288
9.5.8	Bài tập . . . . .	289
9.5.9	Thảo luận . . . . .	289
9.5.10	Những người thực hiện . . . . .	289
9.6	Mạng phần dư (ResNet) . . . . .	290
9.6.1	Các Lớp Hàm Số . . . . .	290
9.6.2	Khối phần dư . . . . .	291
9.6.3	Mô hình ResNet . . . . .	293
9.6.4	Thu thập dữ liệu và Huấn luyện . . . . .	295
9.6.5	Tóm tắt . . . . .	295
9.6.6	Bài tập . . . . .	295
9.6.7	Thảo luận . . . . .	296
9.6.8	Những người thực hiện . . . . .	296
9.7	Mạng Tích chập Kết nối Dày đặc (DenseNet) . . . . .	296
9.7.1	Phân tách Hàm số . . . . .	296
9.7.2	Khối Dày Đặc . . . . .	297
9.7.3	Tầng Chuyển Tiếp . . . . .	298
9.7.4	Mô hình DenseNet . . . . .	299
9.7.5	Thu thập dữ liệu và Huấn luyện . . . . .	299
9.7.6	Tóm tắt . . . . .	300
9.7.7	Bài tập . . . . .	300
9.7.8	Thảo luận . . . . .	300
9.7.9	Những người thực hiện . . . . .	300
9.8	Những người thực hiện . . . . .	301
<b>10</b>	<b>Mạng Nơ-ron Hồi tiếp</b>	<b>303</b>
10.1	Mô hình chuỗi . . . . .	304
10.1.1	Các công cụ thống kê . . . . .	305

10.1.2	Một ví dụ đơn giản . . . . .	307
10.1.3	Dự đoán của Mô hình . . . . .	308
10.1.4	Tóm tắt . . . . .	310
10.1.5	Bài tập . . . . .	310
10.1.6	Thảo luận . . . . .	310
10.1.7	Những người thực hiện . . . . .	311
10.2	Tiền Xử lý Dữ liệu Văn bản . . . . .	311
10.2.1	Đọc Bộ dữ liệu . . . . .	311
10.2.2	Token hoá . . . . .	312
10.2.3	Bộ Từ vựng . . . . .	312
10.2.4	Kết hợp Tất cả lại . . . . .	313
10.2.5	Tóm tắt . . . . .	314
10.2.6	Bài tập . . . . .	314
10.2.7	Thảo luận . . . . .	314
10.2.8	Những người thực hiện . . . . .	314
10.3	Mô hình Ngôn ngữ và Tập dữ liệu . . . . .	315
10.3.1	Ước tính một Mô hình Ngôn ngữ . . . . .	315
10.3.2	Mô hình Markov và $n$ -grams . . . . .	316
10.3.3	Thống kê Ngôn ngữ Tự nhiên . . . . .	317
10.3.4	Chuẩn bị Dữ liệu Huấn luyện . . . . .	318
10.3.5	Tóm tắt . . . . .	320
10.3.6	Bài tập . . . . .	321
10.3.7	Thảo luận . . . . .	321
10.3.8	Những người thực hiện . . . . .	321
10.4	Mạng nơ-ron Hồi tiếp . . . . .	322
10.4.1	Mạng Hồi tiếp không có Trạng thái ẩn . . . . .	322
10.4.2	Mạng Hồi tiếp có Trạng thái ẩn . . . . .	323
10.4.3	Các bước trong một Mô hình Ngôn ngữ . . . . .	324
10.4.4	Perplexity . . . . .	325
10.4.5	Tóm tắt . . . . .	326
10.4.6	Bài tập . . . . .	326
10.4.7	Thảo luận . . . . .	326
10.4.8	Những người thực hiện . . . . .	326
10.5	Lập trình Mạng nơ-ron Hồi tiếp từ đầu . . . . .	327
10.5.1	Biểu diễn One-hot . . . . .	327
10.5.2	Khởi tạo Tham số Mô hình . . . . .	328
10.5.3	Mô hình RNN . . . . .	328
10.5.4	Dự đoán . . . . .	329
10.5.5	Gọt Gradient . . . . .	330
10.5.6	Huấn luyện . . . . .	331
10.5.7	Tóm tắt . . . . .	332
10.5.8	Bài tập . . . . .	333
10.5.9	Thảo luận . . . . .	333
10.5.10	Những người thực hiện . . . . .	333
10.6	Lập trình súc tích Mạng nơ-ron Hồi tiếp . . . . .	334
10.6.1	Định nghĩa Mô hình . . . . .	334
10.6.2	Huấn luyện và Dự đoán . . . . .	335
10.6.3	Tóm tắt . . . . .	335
10.6.4	Bài tập . . . . .	336
10.6.5	Thảo luận . . . . .	336
10.6.6	Những người thực hiện . . . . .	336

10.7	Lan truyền Ngược qua Thời gian . . . . .	336
10.7.1	Mạng Hồi tiếp Giản thể . . . . .	337
10.7.2	Đồ thị Tính toán . . . . .	339
10.7.3	BPTT chi tiết . . . . .	340
10.7.4	Tóm tắt . . . . .	341
10.7.5	Bài tập . . . . .	341
10.7.6	Thảo luận . . . . .	341
10.7.7	Những người thực hiện . . . . .	341
10.8	Những người thực hiện . . . . .	342
<b>11</b>	<b>Mạng Nơ-ron Hồi tiếp Hiện đại</b>	<b>343</b>
11.1	Nút Hồi tiếp có Cổng (GRU) . . . . .	343
11.1.1	Kiểm soát Trạng thái Ẩn . . . . .	344
11.1.2	Lập trình từ đầu . . . . .	346
11.1.3	Lập trình Súc tích . . . . .	348
11.1.4	Tóm tắt . . . . .	349
11.1.5	Bài tập . . . . .	349
11.1.6	Thảo luận . . . . .	349
11.1.7	Những người thực hiện . . . . .	349
11.2	Bộ nhớ Ngắn hạn Dài (LSTM) . . . . .	350
11.2.1	Các Ô nhớ có Cổng . . . . .	350
11.2.2	Lập trình Từ đầu . . . . .	353
11.2.3	Lập trình Súc tích . . . . .	355
11.2.4	Tóm tắt . . . . .	355
11.2.5	Bài tập . . . . .	355
11.2.6	Thảo luận . . . . .	356
11.2.7	Những người thực hiện . . . . .	356
11.3	Mạng Nơ-ron Hồi tiếp Sâu . . . . .	356
11.3.1	Các Phụ thuộc Hàm . . . . .	357
11.3.2	Lập trình Súc tích . . . . .	358
11.3.3	Huấn luyện . . . . .	358
11.3.4	Tóm tắt . . . . .	358
11.3.5	Bài tập . . . . .	359
11.3.6	Thảo luận . . . . .	359
11.3.7	Những người thực hiện . . . . .	359
11.4	Mạng Nơ-ron Hồi tiếp Hai chiều . . . . .	359
11.4.1	Quy hoạch Động . . . . .	360
11.4.2	Mô hình Hai chiều . . . . .	362
11.4.3	Tóm tắt . . . . .	364
11.4.4	Bài tập . . . . .	364
11.4.5	Thảo luận . . . . .	365
11.4.6	Những người thực hiện . . . . .	365
11.5	Dịch Máy và Tập dữ liệu . . . . .	365
11.5.1	Đọc và Tiền Xử lý Dữ liệu . . . . .	366
11.5.2	Token hóa . . . . .	366
11.5.3	Bộ Từ vựng . . . . .	367
11.5.4	Nạp Dữ liệu . . . . .	367
11.5.5	Kết hợp Tất cả lại . . . . .	368
11.5.6	Tóm tắt . . . . .	369
11.5.7	Bài tập . . . . .	369
11.5.8	Thảo luận . . . . .	369

11.5.9	Những người thực hiện . . . . .	369
11.6	Kiến trúc Mã hoá - Giải mã . . . . .	369
11.6.1	Bộ mã hoá . . . . .	370
11.6.2	Bộ giải mã . . . . .	370
11.6.3	Mô hình . . . . .	370
11.6.4	Tóm tắt . . . . .	371
11.6.5	Bài tập . . . . .	371
11.6.6	Thảo luận . . . . .	371
11.6.7	Những người thực hiện . . . . .	371
11.7	Chuỗi sang Chuỗi . . . . .	372
11.7.1	Bộ Mã hóa . . . . .	373
11.7.2	Bộ giải mã . . . . .	374
11.7.3	Hàm Mất mát . . . . .	375
11.7.4	Huấn luyện . . . . .	376
11.7.5	Dự đoán . . . . .	377
11.7.6	Tóm tắt . . . . .	378
11.7.7	Bài tập . . . . .	378
11.7.8	Thảo luận . . . . .	378
11.7.9	Những người thực hiện . . . . .	378
11.8	Tìm kiếm Chùm . . . . .	378
11.8.1	Tìm kiếm Tham lam . . . . .	379
11.8.2	Tìm kiếm Vết cạn . . . . .	380
11.8.3	Tìm kiếm Chùm . . . . .	380
11.8.4	Tóm tắt . . . . .	382
11.8.5	Bài tập . . . . .	382
11.8.6	Thảo luận . . . . .	382
11.8.7	Những người thực hiện . . . . .	383
11.9	Những người thực hiện . . . . .	383

## **12 Cơ chế Tập trung** 385

12.1	Cơ chế Tập trung . . . . .	385
12.1.1	Tầng Tập trung Tích Vô hướng . . . . .	388
12.1.2	Tập trung Perceptron Đa tầng . . . . .	389
12.1.3	Tóm tắt . . . . .	389
12.1.4	Bài tập . . . . .	390
12.1.5	Thảo luận . . . . .	390
12.1.6	Những người thực hiện . . . . .	390
12.2	Chuỗi sang Chuỗi áp dụng Cơ chế Tập trung . . . . .	390
12.2.1	Bộ Giải mã . . . . .	392
12.2.2	Huấn luyện . . . . .	393
12.2.3	Tóm tắt . . . . .	394
12.2.4	Bài tập . . . . .	394
12.2.5	Thảo luận . . . . .	394
12.2.6	Những người thực hiện . . . . .	394
12.3	Kiến trúc Transformer . . . . .	394
12.3.1	Tập trung Đa đầu . . . . .	396
12.3.2	Mạng truyền Xuôi theo Vị trí . . . . .	398
12.3.3	Cộng và Chuẩn hóa . . . . .	399
12.3.4	Biểu diễn Vị trí . . . . .	400
12.3.5	Bộ Mã hóa . . . . .	401
12.3.6	Bộ Giải mã . . . . .	402

12.3.7	Huấn luyện . . . . .	404
12.3.8	Tóm tắt . . . . .	405
12.3.9	Bài tập . . . . .	405
12.3.10	Thảo luận . . . . .	405
12.3.11	Những người thực hiện . . . . .	405
12.4	Những người thực hiện . . . . .	406
<b>13</b>	<b>Thuật toán Tối ưu</b>	<b>407</b>
13.1	Tối ưu và Học sâu . . . . .	407
13.1.1	Tối ưu và Ước lượng . . . . .	407
13.2	Các Thách thức của Tối ưu trong Học sâu . . . . .	408
13.3	Các vùng Cực tiểu . . . . .	409
13.4	Các điểm Yên ngựa . . . . .	409
13.5	Tiêu biến Gradient . . . . .	410
13.5.1	Tóm tắt . . . . .	410
13.5.2	Bài tập . . . . .	411
13.5.3	Thảo luận . . . . .	411
13.5.4	Những người thực hiện . . . . .	411
13.6	Tính lồi . . . . .	412
13.6.1	Kiến thức Cơ bản . . . . .	412
13.6.2	Tính chất . . . . .	414
13.6.3	Ràng buộc . . . . .	417
13.6.4	Tóm tắt . . . . .	418
13.6.5	Bài tập . . . . .	419
13.6.6	Thảo luận . . . . .	419
13.6.7	Những người thực hiện . . . . .	420
13.7	Hạ Gradient . . . . .	420
13.7.1	Hạ Gradient trong Một Chiều . . . . .	420
13.7.2	Hạ Gradient Đa biến . . . . .	422
13.7.3	Những Phương pháp Thích nghi . . . . .	423
13.7.4	Tổng kết . . . . .	426
13.7.5	Bài tập . . . . .	426
13.7.6	Thảo luận . . . . .	427
13.7.7	Những người thực hiện . . . . .	427
13.8	Hạ Gradient Ngẫu nhiên . . . . .	427
13.8.1	Cập nhật Gradient Ngẫu nhiên . . . . .	428
13.8.2	Tốc độ học Linh hoạt . . . . .	429
13.8.3	Phân tích Hội tụ cho Hàm mục tiêu Lồi . . . . .	430
13.8.4	Gradient ngẫu nhiên và Mẫu hữu hạn . . . . .	432
13.8.5	Tóm tắt . . . . .	432
13.8.6	Bài tập . . . . .	433
13.8.7	Thảo luận . . . . .	433
13.8.8	Những người thực hiện . . . . .	433
13.9	Hạ Gradient Ngẫu nhiên theo Minibatch . . . . .	434
13.9.1	Vector hóa và Vùng nhớ đệm . . . . .	434
13.9.2	Minibatch . . . . .	436
13.9.3	Đọc Tập dữ liệu . . . . .	437
13.9.4	Lập trình từ đầu . . . . .	437
13.9.5	Lập trình Súc tích . . . . .	439
13.9.6	Tóm tắt . . . . .	440
13.9.7	Bài tập . . . . .	440

13.9.8 Thảo luận . . . . .	440
13.9.9 Những người thực hiện . . . . .	440
<b>13.10 Động lượng . . . . .</b>	<b>441</b>
13.10.1 Kiến thức Cơ bản . . . . .	441
13.10.2 Thực nghiệm . . . . .	444
13.10.3 Phân tích Lý thuyết . . . . .	445
13.10.4 Tóm tắt . . . . .	446
13.10.5 Bài tập . . . . .	447
13.10.6 Thảo luận . . . . .	447
13.10.7 Những người thực hiện . . . . .	447
<b>13.11 Adagrad . . . . .</b>	<b>447</b>
13.11.1 Đặc trưng Thưa và Tốc độ Học . . . . .	448
13.11.2 Tiền điều kiện . . . . .	448
13.11.3 Thuật toán . . . . .	450
13.11.4 Lập trình từ đầu . . . . .	451
13.11.5 Lập trình Súc tích . . . . .	451
13.11.6 Tóm tắt . . . . .	451
13.11.7 Bài tập . . . . .	452
13.11.8 Thảo luận . . . . .	452
13.11.9 Những người thực hiện . . . . .	452
<b>13.12 RMSProp . . . . .</b>	<b>453</b>
13.12.1 Thuật toán . . . . .	453
13.12.2 Lập trình Từ đầu . . . . .	454
13.12.3 Lập trình Súc tích . . . . .	455
13.12.4 Tóm tắt . . . . .	455
13.12.5 Bài tập . . . . .	455
13.12.6 Thảo luận . . . . .	455
13.12.7 Những người thực hiện . . . . .	455
<b>13.13 Adadelta . . . . .</b>	<b>456</b>
13.13.1 Thuật toán . . . . .	456
13.13.2 Lập trình . . . . .	456
13.13.3 Tóm tắt . . . . .	457
13.13.4 Bài tập . . . . .	457
13.13.5 Thảo luận . . . . .	458
13.13.6 Những người thực hiện . . . . .	458
<b>13.14 Adam . . . . .</b>	<b>458</b>
13.14.1 Thuật toán . . . . .	459
13.14.2 Lập trình . . . . .	459
13.14.3 Yogi . . . . .	460
13.14.4 Tóm tắt . . . . .	461
13.14.5 Bài tập . . . . .	461
13.14.6 Thảo luận . . . . .	461
13.14.7 Những người thực hiện . . . . .	462
<b>13.15 Định thời Tốc độ Học . . . . .</b>	<b>462</b>
13.15.1 Ví dụ Đơn giản . . . . .	463
13.15.2 Bộ Định thời . . . . .	464
13.15.3 Những chính sách . . . . .	465
13.15.4 Tóm tắt . . . . .	467
13.15.5 Bài tập . . . . .	467
13.15.6 Thảo luận . . . . .	468
13.15.7 Những người thực hiện . . . . .	468

13.16	Những người thực hiện . . . . .	468
<b>14</b>	<b>Hiệu năng Tính toán</b>	<b>469</b>
14.1	Trình biên dịch và Trình thông dịch . . . . .	469
14.1.1	Lập trình Ký hiệu . . . . .	470
14.1.2	Lập trình Hybrid . . . . .	471
14.1.3	HybridSequential . . . . .	471
14.1.4	Tóm tắt . . . . .	474
14.1.5	Bài tập . . . . .	475
14.1.6	Thảo luận . . . . .	475
14.2	Tính toán Bất đồng bộ . . . . .	475
14.2.1	Bất đồng bộ qua Back-end . . . . .	476
14.2.2	Lớp cản và Bộ chặn . . . . .	478
14.2.3	Cải thiện Năng lực Tính toán . . . . .	479
14.2.4	Cải thiện Mức chiếm dụng Bộ nhớ . . . . .	479
14.2.5	Tóm tắt . . . . .	481
14.2.6	Bài tập . . . . .	481
14.2.7	Thảo luận . . . . .	482
14.2.8	Những người thực hiện . . . . .	482
14.3	Song song hóa Tự động . . . . .	482
14.3.1	Tính toán Song song trên CPU và GPU . . . . .	483
14.3.2	Tính toán và Giao tiếp Song song . . . . .	483
14.3.3	Tóm tắt . . . . .	485
14.3.4	Bài tập . . . . .	485
14.3.5	Thảo luận . . . . .	486
14.3.6	Những người thực hiện . . . . .	486
14.4	Phần cứng . . . . .	486
14.4.1	Máy tính . . . . .	487
14.4.2	Bộ nhớ . . . . .	488
14.4.3	Lưu trữ . . . . .	489
14.4.4	CPU . . . . .	490
14.4.5	Vi kiến trúc (Micro-architecture) . . . . .	491
14.4.6	Vector hóa (Vectorization) . . . . .	491
14.4.7	Bộ nhớ đệm . . . . .	492
14.4.8	GPU và các Thiết bị Tăng tốc khác . . . . .	493
14.4.9	Mạng máy tính và Bus . . . . .	496
14.4.10	Tóm tắt . . . . .	497
14.4.11	Độ trễ . . . . .	497
14.4.12	Bài tập . . . . .	498
14.4.13	Thảo luận . . . . .	499
14.4.14	Những người thực hiện . . . . .	499
14.5	Huấn luyện đa GPU . . . . .	500
14.5.1	Chia nhỏ Vấn đề . . . . .	500
14.5.2	Song song hóa Dữ liệu . . . . .	502
14.5.3	Ví dụ Đơn giản . . . . .	503
14.5.4	Đồng bộ Dữ liệu . . . . .	504
14.5.5	Phân phối Dữ liệu . . . . .	505
14.5.6	Huấn luyện . . . . .	505
14.5.7	Thí nghiệm . . . . .	506
14.5.8	Tóm tắt . . . . .	506
14.5.9	Bài tập . . . . .	507

14.5.10	Thảo luận . . . . .	507
14.5.11	Những người thực hiện . . . . .	507
14.6	Cách lập trình Súc tích đa GPU . . . . .	507
14.6.1	Ví dụ Đơn giản . . . . .	508
14.6.2	Khởi tạo Tham số và Công việc phụ trợ . . . . .	508
14.6.3	Huấn luyện . . . . .	509
14.6.4	Thử nghiệm . . . . .	510
14.6.5	Tóm tắt . . . . .	510
14.6.6	Bài tập . . . . .	511
14.6.7	Thảo luận . . . . .	511
14.6.8	Những người thực hiện . . . . .	511
14.7	Máy chủ Tham số . . . . .	511
14.7.1	Huấn luyện Song song Dữ liệu . . . . .	512
14.7.2	Đồng bộ dạng Vòng . . . . .	515
14.7.3	Huấn luyện trên Nhiều Máy tính . . . . .	517
14.7.4	Lưu trữ (Khóa, Giá trị) . . . . .	519
14.7.5	Tóm tắt . . . . .	520
14.7.6	Bài tập . . . . .	520
14.7.7	Thảo luận . . . . .	521
14.7.8	Những người thực hiện . . . . .	521
14.8	Những người thực hiện . . . . .	521

<b>15</b>	<b>Thị giác Máy tính</b> . . . . .	<b>523</b>
15.1	Tăng cường Ảnh . . . . .	523
15.1.1	Phương pháp Tăng cường Ảnh Thông dụng . . . . .	524
15.1.2	Huấn luyện Mô hình dùng Tăng cường Ảnh . . . . .	525
15.1.3	Tóm tắt . . . . .	528
15.1.4	Bài tập . . . . .	528
15.1.5	Thảo luận . . . . .	528
15.1.6	Những người thực hiện . . . . .	528
15.2	Tinh Chỉnh . . . . .	529
15.2.1	Nhận dạng Xúc xích . . . . .	530
15.2.2	Tóm tắt . . . . .	533
15.2.3	Bài tập . . . . .	533
15.2.4	Thảo luận . . . . .	534
15.2.5	Những người thực hiện . . . . .	534
15.3	Phát hiện Vật thể và Khoanh vùng Đối tượng (Khung chứa) . . . . .	534
15.3.1	Khung chứa . . . . .	535
15.3.2	Tóm tắt . . . . .	536
15.3.3	Bài tập . . . . .	536
15.3.4	Thảo luận . . . . .	536
15.3.5	Những người thực hiện . . . . .	536
15.4	Khung neo . . . . .	536
15.4.1	Sinh nhiều Khung neo . . . . .	537
15.4.2	Giao trên Hợp . . . . .	538
15.4.3	Gán nhãn Khung neo trong tập Huấn luyện . . . . .	539
15.4.4	Khung chứa khi Dự đoán . . . . .	542
15.4.5	Tóm tắt . . . . .	543
15.4.6	Bài tập . . . . .	543
15.4.7	Thảo luận . . . . .	544
15.4.8	Những người thực hiện . . . . .	544

15.5	Phát hiện Vật thể Đa tỷ lệ . . . . .	544
15.5.1	Tóm tắt . . . . .	546
15.5.2	Bài tập . . . . .	546
15.5.3	Thảo luận . . . . .	546
15.5.4	Những người thực hiện . . . . .	547
15.6	Tập dữ liệu Phát hiện Đối tượng . . . . .	547
15.6.1	Tải xuống tập Dữ liệu . . . . .	547
15.6.2	Đọc dữ liệu . . . . .	548
15.6.3	Minh họa . . . . .	549
15.6.4	Tóm tắt . . . . .	549
15.6.5	Bài tập . . . . .	549
15.6.6	Thảo luận . . . . .	549
15.6.7	Những người thực hiện . . . . .	549
15.7	Phát hiện Nhiều khung Một lượt (SSD) . . . . .	550
15.7.1	Mô hình . . . . .	550
15.7.2	Huấn luyện . . . . .	555
15.7.3	Dự đoán . . . . .	557
15.7.4	Tóm tắt . . . . .	557
15.7.5	Bài tập . . . . .	558
15.7.6	Thảo luận . . . . .	559
15.7.7	Những người thực hiện . . . . .	559
15.8	CNN theo Vùng (R-CNN) . . . . .	560
15.8.1	R-CNN . . . . .	560
15.8.2	Fast R-CNN . . . . .	561
15.8.3	Faster R-CNN . . . . .	562
15.8.4	Mask R-CNN . . . . .	564
15.8.5	Tóm tắt . . . . .	564
15.8.6	Bài tập . . . . .	565
15.8.7	Thảo luận . . . . .	565
15.8.8	Những người thực hiện . . . . .	565
15.9	Phân vùng theo Ngữ nghĩa và Tập dữ liệu . . . . .	565
15.9.1	Phân vùng Ảnh và Phân vùng Thực thể . . . . .	566
15.9.2	Tập dữ liệu Phân vùng theo Ngữ nghĩa Pascal VOC2012 . . . . .	566
15.9.3	Tóm tắt . . . . .	570
15.9.4	Bài tập . . . . .	570
15.9.5	Thảo luận . . . . .	571
15.9.6	Những người thực hiện . . . . .	571
15.10	Tích chập Chuyển vị . . . . .	571
15.10.1	Tích chập Chuyển vị 2D Cơ bản . . . . .	572
15.10.2	Đệm, Sai bước và Kênh . . . . .	573
15.10.3	Sự Tương đồng với Chuyển vị Ma trận . . . . .	573
15.10.4	Tóm tắt . . . . .	574
15.10.5	Bài tập . . . . .	574
15.10.6	Thảo luận . . . . .	574
15.10.7	Những người thực hiện . . . . .	575
15.11	Mạng Tích chập Đầy đủ . . . . .	575
15.11.1	Xây dựng Mô hình . . . . .	575
15.11.2	Khởi tạo Tầng Tích chập Chuyển vị . . . . .	577
15.11.3	Đọc Dữ liệu . . . . .	578
15.11.4	Huấn luyện . . . . .	578
15.11.5	Dự đoán . . . . .	579

15.11.6	Tóm tắt . . . . .	580
15.11.7	Bài tập . . . . .	580
15.11.8	Thảo luận . . . . .	580
15.11.9	Những người thực hiện . . . . .	580
15.12	Truyền tải Phong cách Nø-ron . . . . .	581
15.12.1	Kỹ thuật . . . . .	581
15.12.2	Đọc ảnh Nội dung và Ành phong cách . . . . .	582
15.12.3	Tiền xử lý và Hậu xử lý . . . . .	583
15.12.4	Trích xuất Đặc trưng . . . . .	583
15.12.5	Định nghĩa Hàm Mất mát . . . . .	584
15.12.6	Khai báo và Khởi tạo Ành Tổng hợp . . . . .	586
15.12.7	Huấn luyện . . . . .	587
15.12.8	Tóm tắt . . . . .	588
15.12.9	Bài tập . . . . .	589
15.12.10	Thảo luận . . . . .	589
15.12.11	Những người thực hiện . . . . .	589
15.13	Phân loại ảnh (CIFAR-10) trên Kaggle . . . . .	589
15.13.1	Tải và Tổ chức tập dữ liệu . . . . .	590
15.13.2	Tải tập Dữ liệu . . . . .	591
15.13.3	Tổ chức tập Dữ liệu . . . . .	591
15.14	Nhận diện Giống Chó (ImageNet Dogs) trên Kaggle . . . . .	598
15.14.1	Tải xuống và Tổ chức Tập dữ liệu . . . . .	599
15.14.2	Tăng cường Ành . . . . .	600
15.14.3	Đọc Dữ liệu . . . . .	601
15.14.4	Định nghĩa Mô hình . . . . .	601
15.14.5	Định nghĩa Hàm Huấn luyện . . . . .	602
15.14.6	Huấn luyện và Kiểm định Mô hình . . . . .	603
15.14.7	Dự đoán trên tập Kiểm tra và Nộp Kết quả lên Kaggle . . . . .	603
15.14.8	Tóm tắt . . . . .	604
15.14.9	Bài tập . . . . .	604
15.14.10	Thảo luận . . . . .	604
15.14.11	Những người thực hiện . . . . .	604
15.15	Những người thực hiện . . . . .	604
<b>16</b>	<b>Xử lý Ngôn ngữ Tự nhiên: Tiền Huấn luyện</b>	<b>605</b>
16.1	Embedding Từ (word2vec) . . . . .	606
16.1.1	Tại sao không Sử dụng Vector One-hot? . . . . .	606
16.1.2	Mô hình Skip-Gram . . . . .	607
16.1.3	Mô hình Túi từ Liên tục (CBOW) . . . . .	609
16.1.4	Tóm tắt . . . . .	610
16.1.5	Bài tập . . . . .	610
16.1.6	Thảo luận . . . . .	611
16.1.7	Những người thực hiện . . . . .	611
16.2	Huấn luyện Gần đúng . . . . .	611
16.2.1	Lấy mẫu Âm . . . . .	612
16.2.2	Softmax Phân cấp . . . . .	613
16.2.3	Tóm tắt . . . . .	614
16.2.4	Bài tập . . . . .	614
16.2.5	Thảo luận . . . . .	614
16.2.6	Những người thực hiện . . . . .	614
16.3	Tập dữ liệu để Tiền Huấn luyện Embedding Từ . . . . .	615

16.3.1	Đọc và Tiền xử lý Dữ liệu . . . . .	615
16.3.2	Lấy mẫu con . . . . .	616
16.3.3	Nạp Dữ liệu . . . . .	617
16.3.4	Kết hợp mọi thứ cùng nhau . . . . .	620
16.3.5	Tóm tắt . . . . .	620
16.3.6	Bài tập . . . . .	620
16.3.7	Thảo luận . . . . .	621
16.3.8	Những người thực hiện . . . . .	621
16.4	Tiền huấn luyện word2vec . . . . .	621
16.4.1	Mô hình Skip-Gram . . . . .	622
16.4.2	Huấn luyện . . . . .	623
16.4.3	Áp dụng Mô hình Embedding Từ . . . . .	624
16.4.4	Tóm tắt . . . . .	625
16.4.5	Bài tập . . . . .	625
16.4.6	Thảo luận . . . . .	625
16.4.7	Những người thực hiện . . . . .	625
16.5	Embedding từ với Vector Toàn cục (GloVe) . . . . .	626
16.5.1	Mô hình GloVe . . . . .	627
16.5.2	Lý giải GloVe bằng Tỷ số Xác suất Có điều kiện . . . . .	627
16.5.3	Tóm tắt . . . . .	628
16.5.4	Bài tập . . . . .	629
16.5.5	Thảo luận . . . . .	629
16.5.6	Những người thực hiện . . . . .	629
16.6	Embedding từ con . . . . .	629
16.6.1	fastText . . . . .	630
16.6.2	Mã hoá cặp byte . . . . .	630
16.6.3	Tóm tắt . . . . .	633
16.6.4	Bài tập . . . . .	633
16.6.5	Thảo luận . . . . .	633
16.6.6	Những người thực hiện . . . . .	633
16.7	Tìm kiếm từ Đồng nghĩa và Loại suy . . . . .	634
16.7.1	Sử dụng các Vector Từ đã được Tiền Huấn luyện . . . . .	634
16.7.2	Áp dụng các Vector Từ đã được Tiền huấn luyện . . . . .	636
16.7.3	Tóm tắt . . . . .	637
16.7.4	Bài tập . . . . .	637
16.7.5	Thảo luận . . . . .	637
16.7.6	Những người thực hiện . . . . .	638
16.8	Biểu diễn Mã hóa hai chiều từ Transformer (BERT) . . . . .	638
16.8.1	Từ Độc lập Ngữ cảnh đến Nhạy Ngữ cảnh . . . . .	638
16.8.2	Từ Đặc thù Tác vụ đến Không phân biệt Tác vụ . . . . .	639
16.8.3	BERT: Kết hợp những Điều Tốt nhất của Hai Phương pháp . . . . .	639
16.8.4	Biểu diễn Đầu vào . . . . .	640
16.8.5	Những tác vụ Tiền huấn luyện . . . . .	642
16.8.6	Kết hợp Tất cả lại . . . . .	645
16.8.7	Tóm tắt . . . . .	645
16.8.8	Bài tập . . . . .	646
16.8.9	Thảo luận . . . . .	646
16.8.10	Những người thực hiện . . . . .	646
16.9	Tập dữ liệu để Tiền huấn luyện BERT . . . . .	647
16.9.1	Định nghĩa các Hàm trợ giúp cho các Tác vụ Tiền huấn luyện . . . . .	648
16.9.2	Biến đổi Văn bản thành bộ Dữ liệu Tiền huấn luyện . . . . .	650

16.9.3	Tóm tắt . . . . .	652
16.9.4	Bài tập . . . . .	652
16.9.5	Thảo luận . . . . .	652
16.9.6	Những người thực hiện . . . . .	652
16.10	Tiền Huấn luyện BERT . . . . .	653
16.10.1	Tiền Huấn luyện BERT . . . . .	653
16.10.2	Biểu diễn Văn bản với BERT . . . . .	655
16.10.3	Tóm tắt . . . . .	656
16.10.4	Bài tập . . . . .	656
16.10.5	Thảo luận . . . . .	656
16.10.6	Những người thực hiện . . . . .	657
16.11	Những người thực hiện . . . . .	657
<b>17</b>	<b>Xử lý Ngôn ngữ Tự nhiên: Ứng dụng</b>	<b>659</b>
17.1	Tác vụ Phân tích Cảm xúc và Bộ Dữ liệu . . . . .	660
17.1.1	Bộ Dữ liệu Phân tích Cảm xúc . . . . .	660
17.1.2	Kết hợp Tất cả Lại . . . . .	662
17.1.3	Tóm tắt . . . . .	662
17.1.4	Bài tập . . . . .	662
17.1.5	Thảo luận . . . . .	663
17.1.6	Những người thực hiện . . . . .	663
17.2	Phân tích Cảm xúc: Sử dụng Mạng Nơ-ron Hồi tiếp . . . . .	663
17.2.1	Sử dụng Mạng Nơ-ron Hồi tiếp . . . . .	664
17.2.2	Tóm tắt . . . . .	666
17.2.3	Bài tập . . . . .	666
17.2.4	Thảo luận . . . . .	666
17.2.5	Những người thực hiện . . . . .	666
17.3	Phân tích Cảm xúc: Sử dụng Mạng Nơ-ron Tích Chập . . . . .	667
17.3.1	Mạng Nơ-ron Tích chập Một chiều . . . . .	667
17.3.2	Tầng Gộp Cực đại Theo Thời gian . . . . .	669
17.3.3	Mô hình TextCNN . . . . .	670
17.3.4	Tóm tắt . . . . .	672
17.3.5	Bài tập . . . . .	672
17.3.6	Thảo luận . . . . .	673
17.3.7	Những người thực hiện . . . . .	673
17.4	Suy luận ngôn ngữ tự nhiên và Tập dữ liệu . . . . .	673
17.4.1	Suy luận Ngôn ngữ Tự nhiên . . . . .	673
17.4.2	Tập dữ liệu Suy luận ngôn ngữ tự nhiên của Stanford (SNLI) . . . . .	674
17.4.3	Tóm tắt . . . . .	677
17.4.4	Bài tập . . . . .	677
17.4.5	Thảo luận . . . . .	677
17.4.6	Những người thực hiện . . . . .	677
17.5	Suy luận Ngôn ngữ Tự nhiên: Sử dụng Cơ chế Tập trung . . . . .	678
17.5.1	Mô hình . . . . .	679
17.5.2	Huấn luyện và Đánh giá Mô hình . . . . .	683
17.5.3	Tóm tắt . . . . .	684
17.5.4	Bài tập . . . . .	684
17.5.5	Thảo luận . . . . .	685
17.5.6	Những người thực hiện . . . . .	685
17.6	Tinh chỉnh BERT cho các Ứng dụng Cấp Chuỗi và Cấp Token . . . . .	685
17.6.1	Phân loại Văn bản Đơn . . . . .	686

17.6.2	Phân loại hoặc Hồi quy Cặp Văn bản . . . . .	686
17.6.3	Gán thẻ Văn bản . . . . .	687
17.6.4	Trả lời Câu hỏi . . . . .	688
17.6.5	Tóm tắt . . . . .	689
17.6.6	Bài tập . . . . .	690
17.6.7	Thảo luận . . . . .	690
17.6.8	Những người thực hiện . . . . .	690
17.7	Suy luận Ngôn ngữ Tự nhiên: Tinh chỉnh BERT . . . . .	690
17.7.1	Nạp BERT đã Tiền huấn luyện . . . . .	691
17.7.2	Tập dữ liệu để Tinh chỉnh BERT . . . . .	692
17.7.3	Tinh chỉnh BERT . . . . .	694
17.7.4	Tóm tắt . . . . .	695
17.7.5	Bài tập . . . . .	695
17.7.6	Thảo luận . . . . .	695
17.7.7	Những người thực hiện . . . . .	695
17.8	Những người thực hiện . . . . .	696
<b>18</b>	<b>Hệ thống Đề xuất</b>	<b>697</b>
18.1	Tổng quan về Hệ thống Đề xuất . . . . .	697
18.1.1	Lọc Công tác . . . . .	698
18.1.2	Phản hồi Trực tiếp và Phản hồi Gián tiếp . . . . .	699
18.1.3	Các tác vụ Đề xuất . . . . .	699
18.1.4	Tóm tắt . . . . .	699
18.1.5	Bài tập . . . . .	699
18.1.6	Thảo luận . . . . .	700
18.1.7	Những người thực hiện . . . . .	700
18.2	Tập dữ liệu MovieLens . . . . .	700
18.2.1	Tải Dữ liệu . . . . .	700
18.2.2	Thống kê của Tập dữ liệu . . . . .	701
18.2.3	Chia tập Dữ liệu . . . . .	702
18.2.4	Nạp dữ liệu . . . . .	702
18.2.5	Tóm tắt . . . . .	703
18.2.6	Bài tập . . . . .	704
18.2.7	Thảo luận . . . . .	704
18.2.8	Những người thực hiện . . . . .	704
18.3	Phân rã Ma trận . . . . .	704
18.3.1	Mô hình Phân rã Ma trận . . . . .	705
18.3.2	Cách lập trình Mô hình . . . . .	706
18.3.3	Phương pháp Đánh giá . . . . .	707
18.3.4	Huấn luyện và Đánh giá Mô hình . . . . .	707
18.3.5	Tóm tắt . . . . .	708
18.3.6	Bài tập . . . . .	709
18.3.7	Thảo luận . . . . .	709
18.3.8	Những người thực hiện . . . . .	709
18.4	AutoRec: Dự đoán Đánh giá với Bộ tự Mã hóa . . . . .	709
18.4.1	Mô hình . . . . .	710
18.4.2	Lập trình Mô hình . . . . .	710
18.4.3	Lập trình lại Bộ Đánh giá . . . . .	711
18.4.4	Huấn luyện và Đánh giá Mô hình . . . . .	711
18.4.5	Tóm tắt . . . . .	712
18.4.6	Bài tập . . . . .	712

18.4.7	Thảo luận . . . . .	712
18.4.8	Những người thực hiện . . . . .	712
18.5	Cá nhân hóa Xếp hạng trong Hệ thống Đề xuất . . . . .	713
18.5.1	Mất mát Cá nhân hóa Xếp hạng Bayes và Cách lập trình . . . . .	713
18.5.2	Mất mát Hinge và Cách lập trình . . . . .	715
18.5.3	Tóm tắt . . . . .	715
18.5.4	Bài tập . . . . .	715
18.5.5	Thảo luận . . . . .	716
18.5.6	Những người thực hiện . . . . .	716
18.6	Lọc Cộng tác Nơ-ron cho Cá nhân hóa Xếp hạng . . . . .	716
18.6.1	Mô hình NeuMF . . . . .	717
18.6.2	Lập trình Mô hình . . . . .	718
18.6.3	Tập Dữ liệu Tùy chỉnh với phép Lấy mẫu Âm . . . . .	719
18.6.4	Đánh giá . . . . .	719
18.6.5	Huấn luyện và Đánh giá Mô hình . . . . .	721
18.6.6	Tóm tắt . . . . .	722
18.6.7	Bài tập . . . . .	722
18.6.8	Thảo luận . . . . .	722
18.6.9	Những người thực hiện . . . . .	723
18.7	Hệ thống Đề xuất có Nhận thức về Chuỗi . . . . .	723
18.7.1	Kiến trúc Mô hình . . . . .	723
18.7.2	Lập trình Mô hình . . . . .	725
18.7.3	Tập dữ liệu Tuần tự với phép Lấy mẫu Âm . . . . .	726
18.7.4	Nạp Tập dữ liệu MovieLens 100K . . . . .	727
18.7.5	Huấn luyện Mô hình . . . . .	728
18.7.6	Tóm tắt . . . . .	728
18.7.7	Bài tập . . . . .	728
18.7.8	Thảo luận . . . . .	729
18.7.9	Những người thực hiện . . . . .	729
18.8	Hệ thống Đề xuất Giàu Đặc trưng . . . . .	729
18.8.1	Tập dữ liệu Quảng cáo Trực tuyến . . . . .	730
18.8.2	Wrapper Tập dữ liệu . . . . .	730
18.8.3	Tóm tắt . . . . .	731
18.8.4	Bài tập . . . . .	732
18.8.5	Thảo luận . . . . .	732
18.8.6	Những người thực hiện . . . . .	732
18.9	Máy Phân rã ma trận . . . . .	732
18.9.1	Máy Phân rã 2 Chiều . . . . .	733
18.9.2	Tiêu chuẩn Tối ưu Hiệu quả . . . . .	733
18.9.3	Cách lập trình Mô hình . . . . .	734
18.9.4	Nạp Tập dữ liệu Quảng cáo . . . . .	734
18.9.5	Huấn luyện mô hình . . . . .	735
18.9.6	Tóm tắt . . . . .	735
18.9.7	Bài tập . . . . .	735
18.9.8	Thảo luận . . . . .	735
18.9.9	Những người thực hiện . . . . .	735
18.10	Máy Phân rã Ma trận Sâu . . . . .	736
18.10.1	Kiến trúc Mô hình . . . . .	736
18.10.2	Lập trình DeepFM . . . . .	737
18.10.3	Huấn luyện và Đánh giá Mô hình . . . . .	738
18.10.4	Tóm tắt . . . . .	739

18.10.5	Bài tập . . . . .	739
18.10.6	Thảo luận . . . . .	739
18.10.7	Những người thực hiện . . . . .	739
18.11	Những người thực hiện . . . . .	739
<b>19</b>	<b>Mạng Đối sinh</b>	<b>741</b>
19.1	Mạng Đối sinh . . . . .	741
19.1.1	Sinh một vài Dữ liệu “thật” . . . . .	743
19.1.2	Bộ Sinh . . . . .	743
19.1.3	Bộ Phân biệt . . . . .	744
19.1.4	Huấn luyện . . . . .	744
19.1.5	Tóm tắt . . . . .	745
19.1.6	Bài tập . . . . .	746
19.1.7	Thảo luận . . . . .	746
19.1.8	Những người thực hiện . . . . .	746
19.2	Mạng Đối sinh Tích chập Sâu . . . . .	746
19.2.1	Tập dữ liệu Pokemon . . . . .	747
19.2.2	Bộ Sinh . . . . .	748
19.2.3	Bộ Phân biệt . . . . .	749
19.2.4	Huấn luyện . . . . .	750
19.2.5	Tóm tắt . . . . .	751
19.2.6	Bài tập . . . . .	751
19.2.7	Thảo luận . . . . .	752
19.2.8	Những người thực hiện . . . . .	752
<b>20</b>	<b>Phụ lục: Toán học cho Học Sâu</b>	<b>753</b>
20.1	Các phép toán Hình học và Đại số Tuyến tính . . . . .	754
20.1.1	Ý nghĩa Hình học của Vector . . . . .	754
20.1.2	Tích vô hướng và Góc . . . . .	756
20.1.3	Siêu phẳng . . . . .	758
20.1.4	Ý nghĩa Hình học của các Phép biến đổi Tuyến tính . . . . .	761
20.1.5	Phụ thuộc Tuyến tính . . . . .	762
20.1.6	Hạng . . . . .	763
20.1.7	Tính nghịch đảo (khả nghịch) . . . . .	764
20.1.8	Định thức . . . . .	765
20.1.9	Tensor và các Phép toán Đại số Tuyến tính thông dụng . . . . .	766
20.1.10	Tóm tắt . . . . .	768
20.1.11	Bài tập . . . . .	768
20.1.12	Thảo luận . . . . .	769
20.2	Phân rã trị riêng . . . . .	770
20.2.1	Tìm trị riêng . . . . .	770
20.2.2	Phân rã Ma trận . . . . .	771
20.2.3	Các phép toán dùng Phân rã Trị riêng . . . . .	772
20.2.4	Phân rã trị riêng của Ma trận Đối xứng . . . . .	772
20.2.5	Định lý Vòng tròn Gershgorin . . . . .	773
20.2.6	Một Ứng dụng hữu ích: Mức tăng trưởng của các Ánh xạ Lặp lại . . . . .	773
20.2.7	Kết luận . . . . .	776
20.2.8	Tóm tắt . . . . .	777
20.2.9	Bài tập . . . . .	777
20.2.10	Thảo luận . . . . .	777
20.2.11	Những người thực hiện . . . . .	777

20.3	Giải tích một biến . . . . .	778
20.3.1	Giải tích Vi phân . . . . .	778
20.3.2	Quy tắc Giải tích . . . . .	780
20.3.3	Tóm tắt . . . . .	786
20.3.4	Bài tập . . . . .	787
20.3.5	Thảo luận . . . . .	787
20.3.6	Những người thực hiện . . . . .	787
20.4	Giải tích Nhiều biến . . . . .	787
20.4.1	Đạo hàm trong Không gian Nhiều chiều . . . . .	788
20.4.2	Ý nghĩa Hình học của Gradient và Thuật toán Hạ Gradient . . . . .	789
20.4.3	Một vài chú ý về Tối ưu hóa . . . . .	790
20.4.4	Quy tắc Dây chuyền cho Hàm đa biến . . . . .	791
20.4.5	Thuật toán Lan truyền ngược ( <i>Backpropagation</i> ) . . . . .	793
20.4.6	Hessian . . . . .	795
20.4.7	Giải tích Ma trận . . . . .	797
20.4.8	Tóm tắt . . . . .	801
20.4.9	Bài tập . . . . .	802
20.4.10	Thảo luận . . . . .	802
20.4.11	Những người thực hiện . . . . .	802
20.5	Giải tích Tích phân . . . . .	803
20.5.1	Diễn giải Hình học . . . . .	803
20.5.2	Định lý Cơ bản của Giải tích . . . . .	804
20.5.3	Quy tắc Đổi biến . . . . .	806
20.5.4	Nhận xét về Quy ước Ký hiệu . . . . .	807
20.5.5	Tích phân bội . . . . .	808
20.5.6	Đổi biến trong Tích phân bội . . . . .	810
20.5.7	Tóm tắt . . . . .	811
20.5.8	Bài tập . . . . .	811
20.5.9	Thảo luận . . . . .	811
20.5.10	Những người thực hiện . . . . .	812
20.6	Biến Ngẫu nhiên . . . . .	812
20.6.1	Biến Ngẫu nhiên Liên tục . . . . .	812
20.6.2	Tóm tắt . . . . .	826
20.6.3	Bài tập . . . . .	826
20.6.4	Thảo luận . . . . .	826
20.6.5	Những người thực hiện . . . . .	827
20.7	Hợp lý Cực đại . . . . .	827
20.7.1	Nguyên lý Hợp lý Cực đại . . . . .	827
20.7.2	Hợp lý Cực đại cho Biến Liên tục . . . . .	831
20.7.3	Tóm tắt . . . . .	832
20.7.4	Bài tập . . . . .	832
20.7.5	Thảo luận . . . . .	832
20.7.6	Những người thực hiện . . . . .	832
20.8	Các Phân phối Xác suất . . . . .	833
20.8.1	Phân phối Bernoulli . . . . .	833
20.8.2	Phân phối Đều Rời rạc . . . . .	834
20.8.3	Phân phối Đều Liên tục . . . . .	835
20.8.4	Phân phối Nhị thức . . . . .	836
20.8.5	Phân phối Poisson . . . . .	837
20.8.6	Phân phối Gauss . . . . .	838
20.8.7	Hộ hàm Mũ . . . . .	840

20.8.8	Tóm tắt . . . . .	841
20.8.9	Bài tập . . . . .	841
20.8.10	Thảo luận . . . . .	842
20.8.11	Những người thực hiện . . . . .	842
20.9	Bộ phân loại Naive Bayes . . . . .	842
20.9.1	Nhận diện Ký tự Quang học . . . . .	843
20.9.2	Mô hình xác suất để Phân loại . . . . .	844
20.9.3	Bộ phân loại Naive Bayes . . . . .	844
20.9.4	Huấn luyện . . . . .	845
20.9.5	Tóm tắt . . . . .	847
20.9.6	Bài tập . . . . .	847
20.9.7	Thảo luận . . . . .	848
20.9.8	Những người thực hiện . . . . .	848
20.10	Thống kê . . . . .	848
20.10.1	Đánh giá và So sánh các Bộ ước lượng . . . . .	849
20.10.2	Tiến hành Kiểm định Giả thuyết . . . . .	852
20.10.3	Xây dựng khoảng Tin cậy . . . . .	856
20.10.4	Tóm tắt . . . . .	858
20.10.5	Bài tập . . . . .	858
20.10.6	Thảo luận . . . . .	858
20.10.7	Những người thực hiện . . . . .	859
20.11	Lý thuyết Thông tin . . . . .	859
20.11.1	Thông tin . . . . .	859
20.11.2	Entropy . . . . .	861
20.11.3	Thông tin Tương hỗ . . . . .	863
20.11.4	Phân kỳ Kullback–Leibler . . . . .	867
20.11.5	Entropy Chéo . . . . .	869
20.11.6	Tóm tắt . . . . .	872
20.11.7	Bài tập . . . . .	872
20.11.8	Thảo luận . . . . .	873
20.11.9	Những người thực hiện . . . . .	873
20.12	Những người thực hiện . . . . .	874
<b>21</b>	<b>Phụ lục: Công cụ cho Học Sâu</b>	<b>875</b>
21.1	Sử dụng Jupyter . . . . .	875
21.1.1	Chỉnh sửa và Chạy Mã nguồn trên Máy tính . . . . .	875
21.1.2	Các Lựa chọn Nâng cao . . . . .	879
21.1.3	Tóm tắt . . . . .	880
21.1.4	Bài tập . . . . .	880
21.1.5	Thảo luận . . . . .	880
21.1.6	Những người thực hiện . . . . .	880
21.2	Sử dụng Amazon SageMaker . . . . .	881
21.2.1	Đăng ký và Đăng nhập . . . . .	881
21.2.2	Creating a SageMaker Instance . . . . .	881
21.2.3	Tạo một Máy ảo SageMaker . . . . .	881
21.2.4	Chạy và Dừng một Máy ảo . . . . .	883
21.2.5	Cập nhật Notebook . . . . .	884
21.2.6	Tóm tắt . . . . .	884
21.2.7	Bài tập . . . . .	884
21.2.8	Thảo luận . . . . .	885
21.2.9	Những người thực hiện . . . . .	885

21.3	Sử dụng Máy ảo AWS EC2 . . . . .	885
21.3.1	Khởi tạo và Chạy Một Máy ảo EC2 . . . . .	885
21.3.2	Cài đặt CUDA . . . . .	890
21.3.3	Cài đặt MXNet và Tải Notebook của D2L . . . . .	891
21.3.4	Chạy Jupyter . . . . .	893
21.3.5	Đóng Máy ảo Không Dùng đến . . . . .	893
21.3.6	Tóm tắt . . . . .	894
21.3.7	Bài tập . . . . .	894
21.3.8	Thảo luận . . . . .	894
21.3.9	Những người thực hiện . . . . .	894
21.4	Sử dụng Google Colab . . . . .	894
21.4.1	Tóm tắt . . . . .	895
21.4.2	Bài tập . . . . .	895
21.4.3	Thảo luận . . . . .	895
21.4.4	Những người thực hiện . . . . .	896
21.5	Lựa chọn Máy chủ & GPU . . . . .	896
21.5.1	Lựa chọn Máy chủ . . . . .	896
21.5.2	Lựa chọn GPU . . . . .	897
21.5.3	Tóm tắt . . . . .	900
21.5.4	Thảo luận . . . . .	900
21.5.5	Những người thực hiện . . . . .	901
21.6	Đóng góp cho Quyển sách . . . . .	901
21.6.1	Thay đổi nhỏ trong Văn bản . . . . .	901
21.6.2	Đề xuất một Thay đổi Lớn . . . . .	902
21.6.3	Thêm một Phần Mới hoặc một Cách lập trình cho Framework Mới . . . . .	903
21.6.4	Đăng một Thay đổi Lớn . . . . .	903
21.6.5	Tóm tắt . . . . .	906
21.6.6	Bài tập . . . . .	906
21.6.7	Thảo luận . . . . .	907
21.6.8	Những người thực hiện . . . . .	907
21.7	Tài liệu API của d2l . . . . .	907
21.7.1	Những người thực hiện . . . . .	907
21.8	Những người thực hiện . . . . .	907
<b>22</b>	<b>Bảng thuật ngữ</b>	<b>909</b>
22.1	A . . . . .	909
22.2	B . . . . .	910
22.3	C . . . . .	910
22.4	D . . . . .	912
22.5	E . . . . .	913
22.6	F . . . . .	914
22.7	G . . . . .	915
22.8	H . . . . .	915
22.9	I . . . . .	916
22.10	J . . . . .	916
22.11	K . . . . .	917
22.12	L . . . . .	917
22.13	M . . . . .	918
22.14	N . . . . .	919
22.15	O . . . . .	920
22.16	P . . . . .	920

22.17 Q . . . . .	921
22.18 R . . . . .	922
22.19 S . . . . .	923
22.20 T . . . . .	924
22.21 U . . . . .	925
22.22 V . . . . .	925
22.23 W . . . . .	925

<b>Bibliography</b>	<b>927</b>
---------------------	------------



# Giới thiệu từ nhóm dịch

## Mục tiêu của dự án

Trong những năm gần đây, học sâu là một trong các lĩnh vực được quan tâm nhiều nhất trong các trường Đại học cũng như các Công ty Công nghệ. Ngày càng nhiều các diễn đàn liên quan đến học máy và học sâu với lượng thành viên và chủ đề trao đổi ngày một tăng. Một trong các diễn đàn tiếng Việt nổi bật nhất là [Forum Machine Learning cơ bản<sup>1</sup>](#) và [Diễn đàn Machine Learning cơ bản<sup>2</sup>](#) với hơn 43.800 thành viên và hàng chục chủ đề mới mỗi ngày.

Qua các diễn đàn đó, chúng tôi nhận ra rằng nhu cầu tìm hiểu lĩnh vực này ngày một tăng trong khi lượng tài liệu tiếng Việt còn rất hạn chế. Đặc biệt, các tài liệu tiếng Việt còn chưaхват quan trọng cách dịch, khiến độc giả bối rối trước quá nhiều thông tin nhưng lại quá ít thông tin đầy đủ. Việc này thúc đẩy chúng tôi tìm và dịch những cuốn sách được quan tâm nhiều về lĩnh vực này.

Nhóm dịch đã bước đầu thành công khi dịch cuốn [Machine Learning Yearning<sup>3</sup>](#) của tác giả Andrew Ng. Cuốn sách này đề cập đến các vấn đề cần lưu ý khi xây dựng các hệ thống học máy, trong đó đề cập đến nhiều kiến thức thực tế khi thực hiện dự án. Tuy nhiên, cuốn sách này phần nào hướng tới những người đã có những kinh nghiệm nhất định đã đang tham gia các dự án học máy. Chúng tôi vẫn khao khát được mang một tài liệu đầy đủ hơn với đủ kiến thức toán nền tảng, cách triển khai các công thức toán bằng mã nguồn, cùng với cách triển khai một hệ thống thực tế trên một nền tảng học sâu được nhiều người sử dụng. Và quan trọng hơn, các kiến thức này phải cập nhật các xu hướng học máy mới nhất.

Sau nhiều ngày tìm kiếm các cuốn sách về học máy/học sâu được các trường đại học lớn trên thế giới sử dụng trong quá trình giảng dạy, chúng tôi quyết định dịch cuốn [Dive into Deep Learning<sup>4</sup>](#) của nhóm tác giả từ công ty Amazon. Cuốn này hội tụ đủ các yếu tố: có giải thích toán dễ hiểu, có code đi kèm cho những bạn muốn thực hành ngay khi học xong lý thuyết, cập nhật đầy đủ những khía cạnh của học sâu, và quan trọng nhất là không đòi hỏi bản quyền để dịch. Chúng tôi đã liên hệ với nhóm tác giả và họ rất vui mừng khi cuốn sách sắp được phổ biến rộng rãi hơn nữa.

Hiện cuốn sách vẫn đang được thực hiện và cập nhật nội dung dựa trên phiên bản 0.14.0 mới nhất. Chúng tôi cũng chọn bản này vì nó sử dụng thư viện chính là numpy (tích hợp trong MXNet), một thư viện xử lý mảng nhiều chiều phổ biến mà theo chúng tôi, người làm về học máy, học sâu và khoa học dữ liệu cần biết. Và chúng tôi cũng đang cập nhật dần những thư viện khác như Pytorch và TensorFlow vào nhằm đa dạng hóa lựa chọn hơn cho độc giả.

Để có thể thực hiện dự án dịch cuốn sách gần 1.000 trang này, chúng tôi rất cần sự chung tay từ cộng đồng. Mọi sự đóng góp đều đáng trân quý và được ghi nhận. Chúng tôi hy vọng cuốn sách

<sup>1</sup> <https://www.facebook.com/groups/machinelearningcovan/>

<sup>2</sup> <https://forum.machinelearningcovan.com/>

<sup>3</sup> [https://github.com/aivivn/Machine-Learning-Yearning-Vietnamese-Translation/blob/master/chapters/all\\_chapters.md](https://github.com/aivivn/Machine-Learning-Yearning-Vietnamese-Translation/blob/master/chapters/all_chapters.md)

<sup>4</sup> <https://www.d2l.ai/>

sẽ được hoàn thành trong năm 2020. Và sau đó nó có thể trở thành giáo trình trong các trường đại học. Hy vọng một ngày chúng ta có thể nhìn thấy một trường của Việt Nam trong danh sách này:

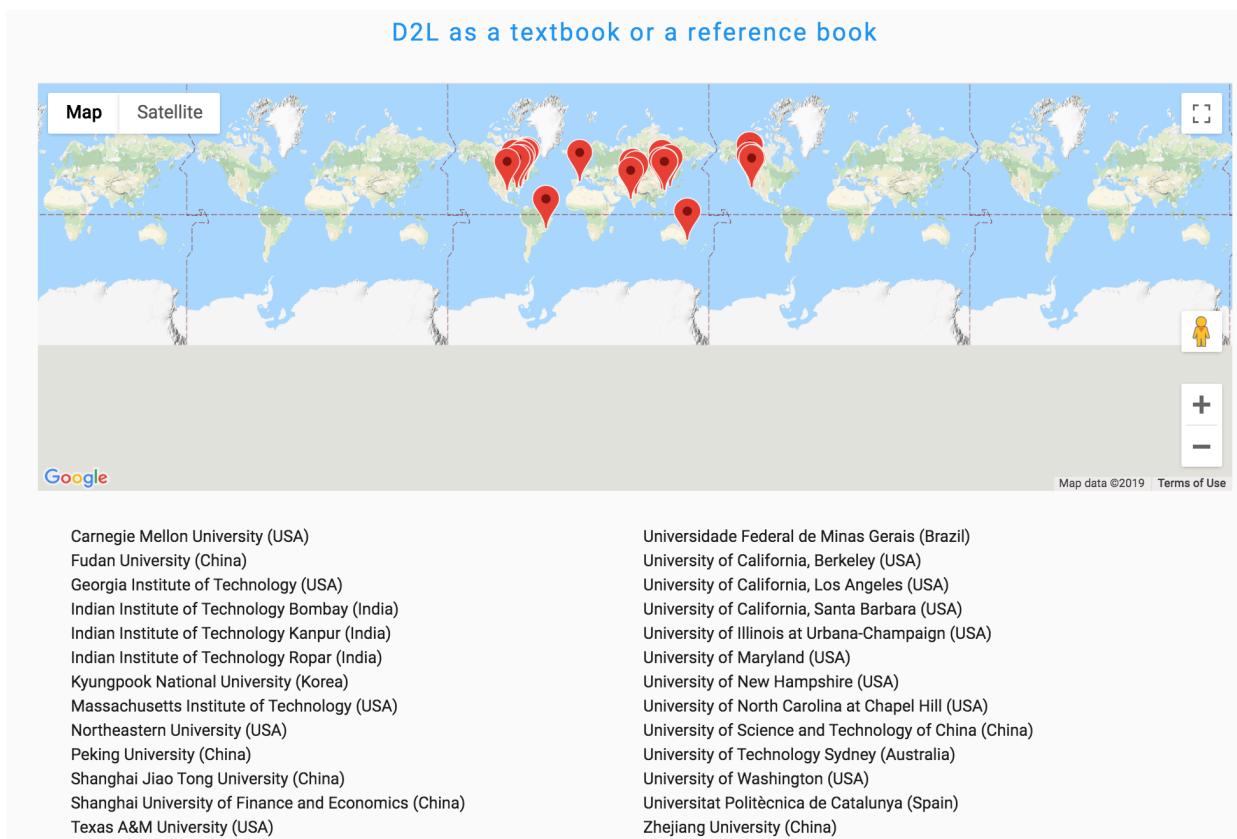


Fig. 1: Danh sách các trường đại học sử dụng cuốn sách này

## Diễn đàn

Nội dung cuốn sách này rất phong phú và có nhiều bài tập ở cuối mỗi phần. Các bạn có thể tham gia thảo luận nội dung và bài tập của cuốn sách [tại đây](#)<sup>5</sup>.

## Những câu hỏi thường gặp

### 1. Cuốn sách này có bản song ngữ hay không?

Không, chúng tôi không có kế hoạch thực hiện bản song ngữ cho cuốn sách này. #### 2. Cuốn sách này có bản PDF hay không? Có, chúng tôi sẽ có bản PDF sau khi đã hoàn thiện toàn bộ nội dung cuốn sách này.

<sup>5</sup> <https://forum.machinelearningcoban.com/c/d2l>

# Lời nói đầu

Chỉ một vài năm trước, không có nhiều nhà khoa học học sâu (*deep learning*) phát triển các sản phẩm và dịch vụ thông minh tại các công ty lớn cũng như các công ty khởi nghiệp. Khi người trẻ nhất trong nhóm tác giả chúng tôi tiến vào lĩnh vực này, học máy (*machine learning*) còn chưa xuất hiện thường xuyên trên truyền thông. Cha mẹ chúng tôi còn không có ý niệm gì về học máy chứ chưa nói đến việc hiểu tại sao chúng tôi theo đuổi lĩnh vực này thay vì y khoa hay luật khoa. Học máy từng là một lĩnh vực nghiên cứu tiên phong với chỉ một số lượng nhỏ các ứng dụng thực tế. Những ứng dụng như nhận dạng giọng nói (*speech recognition*) hay thị giác máy tính (*computer vision*), đòi hỏi quá nhiều kiến thức chuyên biệt khiến chúng thường được phân thành các lĩnh vực hoàn toàn riêng mà trong đó học máy chỉ là một thành phần nhỏ. Các mạng nơ-ron (*neural network*), tiền đề của các mô hình học sâu mà chúng ta tập trung vào trong cuốn sách này, đã từng bị coi là các công cụ lỗi thời.

Chỉ trong khoảng năm năm gần đây, học sâu đã mang đến nhiều bất ngờ trên quy mô toàn cầu và dẫn đường cho những tiến triển nhanh chóng trong nhiều lĩnh vực khác nhau như thị giác máy tính, xử lý ngôn ngữ tự nhiên (*natural language processing*), nhận dạng giọng nói tự động (*automatic speech recognition*), học tăng cường (*reinforcement learning*), và mô hình hoá thống kê (*statistical modeling*). Với những tiến bộ này, chúng ta bây giờ có thể xây dựng xe tự lái với mức độ tự động ngày càng cao (nhưng chưa nhiều tới mức như vài công ty đang tuyên bố), xây dựng các hệ thống giúp trả lời thư tự động khi con người ngập trong núi email, hay lập trình phần mềm chơi cờ vây có thể thắng cả nhà vô địch thế giới, một kỷ tích từng được xem là không thể đạt được trong nhiều thập kỷ tới. Những công cụ này đã và đang gây ảnh hưởng rộng rãi tới các ngành công nghiệp và đời sống xã hội, thay đổi cách tạo ra các bộ phim, cách chẩn đoán bệnh và đóng một vài trò ngày càng tăng trong các ngành khoa học cơ bản – từ vật lý thiên văn tới sinh học.

## Về cuốn sách này

Cuốn sách này được viết với mong muốn làm cho học sâu dễ tiếp cận hơn. Nó sẽ dạy bạn từ *khái niệm, bối cảnh*, cho tới cách *lập trình*.

## Một phương tiện truyền tải kết hợp Mã nguồn, Toán, và HTML

Để một công nghệ điện toán đạt được tầm ảnh hưởng sâu rộng, nó phải dễ hiểu, có tài liệu đầy đủ, và được hỗ trợ bởi những công cụ cấp tiến được “bảo trì” thường xuyên. Các ý tưởng chính cần được chắt lọc rõ ràng, tối thiểu thời gian chuẩn bị cần thiết cho người mới bắt đầu để họ có thể trang bị các kiến thức đương thời. Các thư viện cấp tiến nên tự động hoá các tác vụ đơn giản, và các đoạn mã nguồn được lấy làm ví dụ cần phải đơn giản với những người mới bắt đầu sao cho họ có thể dễ dàng chỉnh sửa, áp dụng, và mở rộng những ứng dụng thông thường thành các ứng dụng họ cần. Lấy ứng dụng các trang web động làm ví dụ. Mặc dù các công ty công nghệ lớn như Amazon phát triển thành công các ứng dụng web định hướng bởi cơ sở dữ liệu từ những năm 1990, tiềm năng của công nghệ này để hỗ trợ các doanh nghiệp sáng tạo chỉ được nhân rộng lên ở một tầm cao mới từ khoảng mươi năm nay, nhờ vào sự phát triển của các nền tảng mạnh mẽ và với tài liệu đầy đủ.

Kiểm định tiềm năng của học sâu có những thách thức riêng biệt vì bất kỳ ứng dụng riêng lẻ nào cũng bao gồm nhiều lĩnh vực khác nhau. Ứng dụng học sâu đòi hỏi những hiểu biết đồng thời về (i) động lực để mô hình hoá một bài toán theo một hướng cụ thể; (ii) kiến thức toán học của một phương pháp mô hình hoá; (iii) những thuật toán tối ưu để khớp mô hình với dữ liệu; và (iv) phần kỹ thuật yêu cầu để huấn luyện mô hình một cách hiệu quả, xử lý những khó khăn trong tính toán và tận dụng thật tốt phần cứng hiện có. Việc đào tạo kỹ năng suy nghĩ thấu đáo cần thiết để định hình bài toán, cung cấp kiến thức toán để giải chúng, và hướng dẫn cách dùng các công cụ phần mềm để triển khai những giải pháp đó, tất cả trong một nơi, hàm chứa nhiều thách thức lớn. Mục tiêu của chúng tôi trong cuốn sách này là trình bày một nguồn tài liệu tổng hợp giúp những học viên nhanh chóng bắt kịp.

Chúng tôi bắt đầu dự án sách này từ tháng 7/2017 khi cần trình bày giao diện MXNet Gluon (khi đó còn mới) tới người dùng. Tại thời điểm đó, không có một nguồn tài liệu nào vừa đồng thời (i) cập nhật; (ii) bao gồm đầy đủ các khía cạnh của học máy hiện đại với đầy đủ chiều sâu kỹ thuật; và (iii) xem kẽ các giải trình mà người ta mong đợi từ một cuốn sách giáo trình với mã nguồn có thể thực thi, điều thường được tìm thấy trong các bài hướng dẫn thực hành. Chúng tôi tìm thấy một lượng lớn các đoạn mã ví dụ về việc sử dụng một nền tảng học sâu (ví dụ làm thế nào để thực hiện các phép toán cơ bản với ma trận trên TensorFlow) hoặc để triển khai những kỹ thuật cụ thể (ví dụ các đoạn mã cho LeNet, AlexNet, ResNet,...) trong các bài blog hoặc là trên GitHub. Tuy nhiên, những ví dụ này thường tập trung vào khía cạnh *làm thế nào* để triển khai một hướng tiếp cận cho trước, mà bỏ qua việc thảo luận *tại sao* một thuật toán được tạo như thế. Nhiều chủ đề đã được đề cập đến trong các bài blog, ví dụ như trang [Distill<sup>6</sup>](http://distill.pub) hoặc các trang cá nhân, chúng thường chỉ đề cập đến một vài chủ đề được chọn về học sâu và thường thiếu mã nguồn đi kèm. Một mặt khác, trong khi nhiều sách giáo trình đã ra đời, đáng chú ý nhất là ([Goodfellow et al., 2016](#)) (cuốn này cung cấp một bản khảo sát xuất sắc về các khái niệm phía sau học sâu), những nguồn tài liệu này lại không đi kèm với việc diễn giải dưới dạng mã nguồn để làm rõ hơn các khái niệm. Điều này khiến người đọc đôi khi mơ hồ về cách thực thi chúng. Bên cạnh đó, rất nhiều tài liệu lại được cung cấp dưới dạng các khoá học có phí.

Chúng tôi đặt mục tiêu tạo ra một tài liệu mà có thể (1) miễn phí cho mọi người; (2) cung cấp chiều sâu kỹ thuật đầy đủ, là điểm khởi đầu trên con đường trở thành một nhà khoa học học máy ứng dụng; (3) bao gồm mã nguồn thực thi được, trình bày cho người đọc *làm thế nào* giải quyết các bài toán trên thực tế; (4) tài liệu này có thể cập nhật một cách nhanh chóng bởi các tác giả cũng như cộng đồng ở quy mô lớn; và (5) được bổ sung bởi một [diễn đàn<sup>7</sup>](#) (và [diễn đàn tiếng Việt<sup>8</sup>](#) của nhóm dịch) để nhanh chóng thảo luận và hỏi đáp về các chi tiết kỹ thuật.

<sup>6</sup> <http://distill.pub>

<sup>7</sup> <http://discuss.mxnet.io>

<sup>8</sup> <https://forum.machinelarningcaban.com/c/d21>

Các mục tiêu này thường không tương thích với nhau. Các công thức, định lý, và các trích dẫn được quản lý tốt nhất trên LaTeX. Mã được giải thích tốt nhất bằng Python. Và trang web phù hợp với HTML và JavaScript. Hơn nữa, chúng tôi muốn nội dung của nó vừa có thể được truy cập dưới dạng mã nguồn có thể thực thi, vừa có thể tải về như một cuốn sách dưới định dạng PDF, và lại ở trên internet như một trang web. Hiện tại không có một công cụ nào là hoàn hảo cho những nhu cầu này, bởi vậy chúng tôi phải tự tạo công cụ cho riêng mình. Chúng tôi mô tả hướng tiếp cận một cách chi tiết trong chapter\_contribute. Chúng tôi tổ chức dự án trên GitHub để chia sẻ mã nguồn và cho phép sửa đổi, Jupyter notebook để kết hợp đoạn mã, phương trình toán và nội dung chữ, sử dụng Sphinx như một bộ máy tạo nhiều tập tin đầu ra, và Discourse để tạo diễn đàn. Trong khi hệ thống này còn chưa hoàn hảo, những lựa chọn này cung cấp một giải pháp chấp nhận được trong số các giải pháp tương tự. Chúng tôi tin rằng đây có thể là cuốn sách đầu tiên được xuất bản dưới dạng kết hợp này.

## Học thông qua thực hành

Có nhiều cuốn sách dạy rất chi tiết về một chuỗi các chủ đề khác nhau. Ví dụ như trong cuốn sách tuyệt vời ([Bishop, 2006](#)) này của Bishop, mỗi chủ đề được dạy rất kỹ lưỡng tới nỗi để đến được chương hồi quy tuyến tính cũng đòi hỏi không ít công sức phải bỏ ra. Các chuyên gia yêu thích quyển sách này chính vì sự kỹ lưỡng mà nó mang lại, nhưng với những người mới bắt đầu thì đây là điểm hạn chế việc sử dụng cuốn sách này như một tài liệu nhập môn.

Trong cuốn sách này, chúng tôi sẽ dạy hầu hết các khái niệm ở mức vừa đủ. Hay nói cách khác, bạn sẽ chỉ học và hiểu các khái niệm cần thiết đủ để bạn hoàn tất phần thực hành. Trong khi chúng tôi sẽ dành một chút thời gian để dạy kiến thức căn bản sơ bộ như đại số tuyến tính và xác suất, chúng tôi muốn các bạn được tận hưởng cảm giác mãn nguyện của việc huấn luyện được mô hình đầu tiên trước khi bận tâm tới các lý thuyết phân phối xác suất.

Bên cạnh một vài notebook cơ bản cung cấp một khoá học cấp tốc về nền tảng toán học, mỗi chương tiếp theo sẽ giới thiệu một lượng hợp lý các khái niệm mới và đồng thời cung cấp các ví dụ đơn hoàn chỉnh—sử dụng các tập dữ liệu thực tế. Và đây là cả thách thức về cách tổ chức nội dung. Một vài mô hình có thể được nhóm lại một cách có logic trong một notebook riêng lẻ. Và một vài ý tưởng có thể được dạy tốt nhất bằng cách thực thi một số mô hình kế tiếp nhau. Mặt khác, có một lợi thế lớn về việc tuân thủ theo chính sách *mỗi notebook là một ví dụ hoàn chỉnh*: Điều này giúp bạn bắt đầu các dự án nghiên cứu của mình một cách dễ dàng nhất có thể bằng cách tận dụng mã nguồn của chúng tôi. Bạn chỉ cần sao chép một notebook và bắt đầu sửa đổi ở trên đó.

Chúng tôi sẽ xen kẽ mã nguồn có thể thực thi với kiến thức nền tảng khi cần thiết. Thông thường, chúng tôi sẽ tập trung vào việc tạo ra những công cụ trước khi giải thích chúng đầy đủ (và chúng tôi sẽ theo sát bằng cách giải thích phần kiến thức nền tảng sau). Ví dụ, chúng tôi có thể sử dụng *hàm gradient ngẫu nhiên* trước khi giải thích đầy đủ tại sao nó lại hữu ích hoặc tại sao nó lại hoạt động. Điều này giúp cung cấp cho người thực hành những phương tiện cần thiết để giải quyết vấn đề nhanh chóng và đòi hỏi người đọc phải tin tưởng vào một số quyết định triển khai của chúng tôi.

Xuyên suốt cuốn sách, chúng ta sẽ làm việc với thư viện MXNet; đây là một thư viện với một đặc tính hiếm có, đó là vừa đủ linh hoạt để nghiên cứu và đủ nhanh để tạo ra sản phẩm. Cuốn sách này sẽ dạy về khái niệm học sâu từ đầu. Thỉnh thoảng, chúng tôi sẽ muốn đào sâu hơn vào những chi tiết về mô hình mà thông thường sẽ được che giấu khỏi người dùng bởi những lớp trừu tượng bậc cao Gluon. Điều này đặc biệt hay xuất hiện trong các hướng dẫn cơ bản, nơi chúng tôi muốn bạn hiểu về tất cả mọi thứ đang diễn ra trong một tầng hoặc bộ tối ưu nào đó. Trong những trường hợp này, chúng tôi sẽ thường trình bày hai phiên bản của một ví dụ: một phiên bản trong đó chúng tôi hiện thực mọi thứ từ đầu, chỉ dựa vào giao diện Numpy và việc tính đạo hàm tự động; và một

phiên bản khác thực tế hơn, khi chúng tôi viết mã ngắn gọn sử dụng Gluon. Một khi chúng tôi đã dạy bạn cách một số thành phần hoạt động cụ thể như thế nào, chúng tôi có thể chỉ sử dụng phiên bản Gluon trong những hướng dẫn tiếp theo.

## Nội dung và Bố cục

Cuốn sách này có thể được chia thành ba phần, với các phần được thể hiện bởi những màu khác nhau trong Fig. 2:

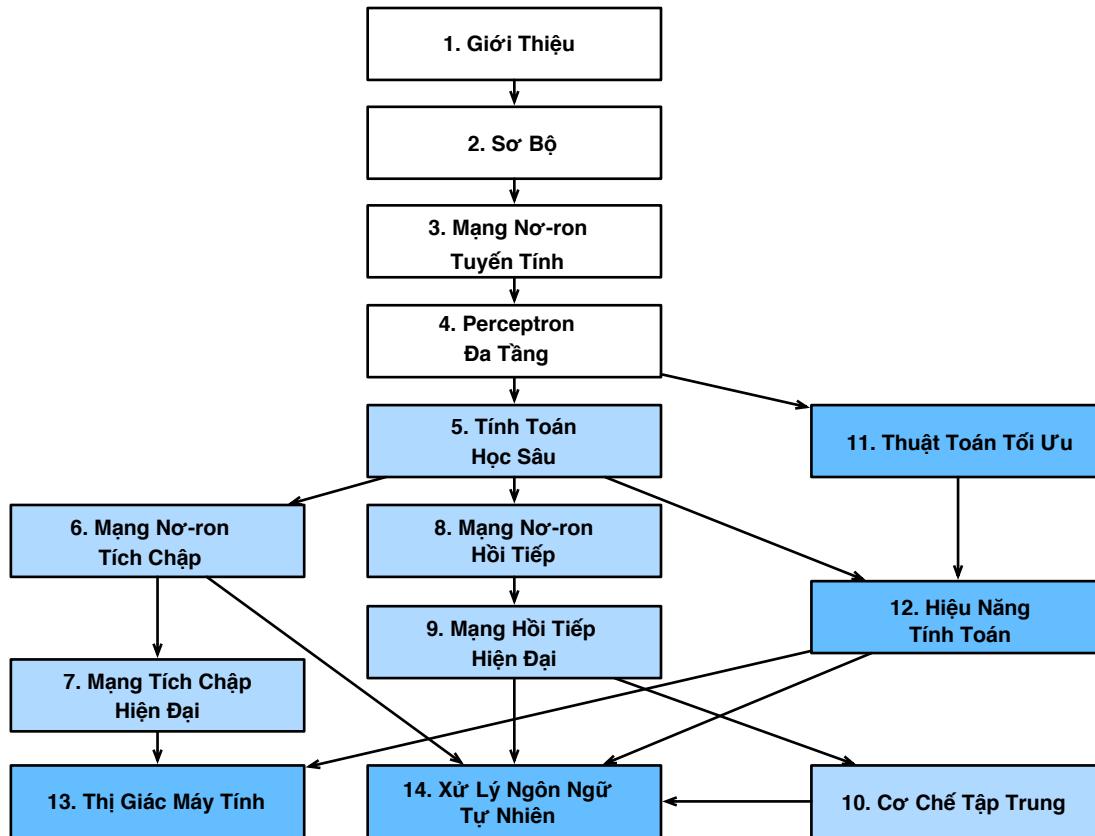


Fig. 2: Bố cục cuốn sách

- Phần đầu cuốn sách trình bày các kiến thức cơ bản và những việc cần chuẩn bị sơ bộ. Section 3 giới thiệu về học sâu. Sau đó, qua Section 4, chúng tôi nhanh chóng trang bị cho bạn những kiến thức nền cần thiết để thực hành học sâu như cách lưu trữ, thao tác dữ liệu và cách áp dụng những phép tính dựa trên những khái niệm cơ bản trong đại số tuyến tính, giải tích và xác suất. Section 5 và Section 6 giới thiệu những khái niệm và kỹ thuật cơ bản của học sâu, ví dụ như hồi quy tuyến tính, mạng perceptron đa lớp và điều chỉnh.
- Năm chương tiếp theo tập trung vào những kỹ thuật học sâu hiện đại. Section 7 miêu tả những thành phần thiết yếu của các phép tính trong học sâu và tạo nền tảng để chúng tôi triển khai những mô hình phức tạp hơn. Sau đó, chúng tôi sẽ giới thiệu mạng nơ-ron tích chập (Convolutional Neural Networks/CNNs), một công cụ mạnh mẽ đang là nền tảng của hầu hết các hệ thống thị giác máy tính hiện đại. Tiếp đến, trong Section 10 và Section 11, chúng tôi giới thiệu mạng nơ-ron hồi tiếp (Recurrent Neural Networks/RNNs), một loại mô hình khai thác cấu trúc tạm thời hoặc tuần tự trong dữ liệu và thường được sử dụng để xử lý ngôn ngữ tự nhiên và dự đoán chuỗi thời gian. Trong Section 12, chúng tôi giới thiệu một lớp mô hình mới sử dụng kỹ thuật cơ chế chú ý (attention mechanisms), một kỹ thuật gần đây

đã thay thế RNNs trong xử lý ngôn ngữ tự nhiên. Những phần này sẽ giúp bạn nhanh chóng nắm được những công cụ cơ bản đứng sau hầu hết các ứng dụng hiện đại của học sâu.

- Phần ba thảo luận quy mô mở rộng, hiệu quả và ứng dụng. Đầu tiên, trong Section 13, chúng tôi bàn luận một số thuật toán tối ưu phổ biến được sử dụng để huấn luyện các mô hình học sâu. Chương tiếp theo, Section 14 khảo sát những yếu tố chính ảnh hưởng đến chất lượng tính toán của mã nguồn học sâu. Trong Section 15 và chap\_nlp, chúng tôi minh họa lần lượt những ứng dụng chính của học sâu trong thị giác máy tính và xử lý ngôn ngữ tự nhiên.

## Mã nguồn

Hầu hết các phần của cuốn sách đều bao gồm mã nguồn thực thi được, bởi vì chúng tôi tin rằng trải nghiệm học thông qua tương tác đóng một vai trò quan trọng trong học sâu. Hiện tại, một số kinh nghiệm nhất định chỉ có thể được hình thành thông qua phương pháp thử và sai, thay đổi mã nguồn từng chút một và quan sát kết quả. Lý tưởng nhất là sử dụng một lý thuyết toán học khác biệt nào đó có thể cho chúng ta biết chính xác cách thay đổi mã nguồn để đạt được kết quả mong muốn. Thật đáng tiếc là hiện tại những lý thuyết khác biệt đó vẫn chưa được khám phá ra. Mặc dù chúng tôi đã cố gắng hết sức, vẫn chưa có cách giải thích rõ ràng nào cho nhiều vấn đề kỹ thuật, bởi vì phần toán học để mô tả những mô hình đó có thể là rất khó và công cuộc tìm hiểu về những chủ đề này mới chỉ tăng cao trong thời gian gần đây. Chúng tôi hy vọng rằng khi mà những lý thuyết về học sâu phát triển, những phiên bản tiếp theo của cuốn sách sẽ có thể cung cấp những cái nhìn sâu sắc hơn mà phiên bản hiện tại chưa làm được.

Hầu hết mã nguồn trong cuốn sách được dựa theo Apache MXNet. MXNet là một framework mã nguồn mở dành cho học sâu và là lựa chọn yêu thích của AWS (Amazon Web Services), và cả ở nhiều trường đại học và công ty. Tất cả mã nguồn trong cuốn sách này đã được kiểm thử trên phiên bản mới nhất của MXNet. Tuy nhiên, bởi vì học sâu phát triển rất nhanh, một vài đoạn mã *trong phiên bản sách in* có thể không hoạt động chuẩn trên những phiên bản MXNet sau này. Dù vậy, chúng tôi dự định sẽ giữ phiên bản trực tuyến luôn được cập nhật. Trong trường hợp bạn gặp phải bất cứ vấn đề nào, hãy tham khảo [Cài đặt](#) (page 11) để cập nhật mã nguồn và môi trường thực thi.

Để tránh việc lặp lại không cần thiết, chúng tôi đóng gói những hàm, lớp,... mà thường xuyên được chèn vào và tham khảo đến trong cuốn sách này trong gói thư viện d2l. Đối với bất kỳ đoạn mã nguồn nào như là một hàm, một lớp, hoặc các khai báo thư viện cần được đóng gói, chúng tôi sẽ đánh dấu bằng dòng `# Saved in the d2l package for later use` (Lưu lại trong gói thư viện d2l để sử dụng sau). Thư viện d2l khá nhẹ và chỉ phụ thuộc vào những gói thư viện và mô-đun sau:

```
# Saved in the d2l package for later use
import collections
from collections import defaultdict
from IPython import display
import math
from matplotlib import pyplot as plt
from mxnet import autograd, context, gluon, image, init, np, npx
from mxnet.gluon import nn, rnn
import os
import pandas as pd
import random
import re
import sys
```

(continues on next page)

```
import tarfile
import time
import zipfile
```

Chúng tôi có một bản tổng quan chi tiết về những hàm và lớp này trong Section 21.7.

## Đối tượng độc giả

Cuốn sách này dành cho các bạn sinh viên (đại học hoặc sau đại học), các kỹ sư và các nhà nghiên cứu – những người tìm kiếm một nền tảng vững chắc về những kỹ thuật thực tế của học sâu. Bởi vì chúng tôi giải thích mọi khái niệm từ đầu, bạn không bắt buộc phải có nền tảng về học sâu hay học máy. Việc giải thích đầy đủ các phương pháp học sâu đòi hỏi một số kiến thức về toán học và lập trình, nhưng chúng tôi sẽ chỉ giả định rằng bạn nắm được một số kiến thức cơ bản về đại số tuyến tính, giải tích, xác suất, và lập trình Python. Hơn nữa, trong phần Phụ lục, chúng tôi cung cấp thêm về hầu hết các phần toán được đề cập trong cuốn sách này. Phần lớn thời gian, chúng tôi sẽ ưu tiên dùng cách giải thích trực quan và mô tả các ý tưởng hơn là giải thích chật chẽ bằng toán. Có rất nhiều cuốn sách tuyệt vời có thể thu hút bạn đọc quan tâm sâu hơn nữa. Chẳng hạn, cuốn “Giải tích tuyến tính” (Linear Analysis) của Bela Bollobas ([Bollobas, 1999](#)) bao gồm cả đại số tuyến tính và giải tích hàm ở mức độ rất chi tiết. Cuốn “Tất cả về Thống kê” (All of Statistics) ([Wasserman, 2013](#)) là hướng dẫn tuyệt vời để học thống kê. Và nếu bạn chưa sử dụng Python, bạn có thể muốn xem cuốn [hướng dẫn Python](#)<sup>9</sup>.

## Diễn đàn

Gắn liền với cuốn sách, chúng tôi đã tạo ra một diễn đàn trực tuyến tại [discuss.mxnet.io](#)<sup>10</sup> (và tại [Diễn đàn](#) nhóm dịch tạo<sup>11</sup>). Khi có câu hỏi về bất kỳ phần nào của cuốn sách, bạn có thể tìm thấy trang thảo luận liên quan bằng cách quét mã QR ở cuối mỗi chương để tham gia vào các cuộc thảo luận. Các tác giả của cuốn sách này và rộng hơn là cộng đồng phát triển MXNet cũng thường xuyên tham gia thảo luận trong diễn đàn.

## Lời cảm ơn

Chúng tôi xin gửi lời cảm ơn chân thành tới hàng trăm người đã đóng góp cho cả hai bản thảo tiếng Anh và tiếng Trung. Mọi người đã giúp cải thiện nội dung và đưa ra những phản hồi rất có giá trị. Cụ thể, chúng tôi cảm ơn tất cả những người đóng góp cho dự thảo tiếng Anh này giúp nó tốt hơn cho tất cả mọi người. Tài khoản GitHub hoặc tên các bạn đóng góp (không theo trình tự cụ thể nào): alxnorden, avinashsingit, bowen0701, brettkoonce, Chaitanya Prakash Bapat, cryptonaut, Davide Fiocco, edgarroman, gkutiel, John Mitro, Liang Pu, Rahul Agarwal, Mohamed Ali Jamaoui, Michael (Stu) Stewart, Mike Müller, NRauschmayr, Prakhar Srivastav, sad-sfermigier, Sheng Zha, sundeepteki, topecongiro, tpdi, vermicelli, Vishaal Kapoor, vishwesh5, YaYaB, Yuhong Chen, Evgeniy Smirnov, Igov, Simon Corston-Oliver, IgorDzreyev, Ha Nguyen, pmuens, alukovenko, senorcinco, vfdev-5, dsweet, Mohammad Mahdi Rahimi, Abhishek Gupta, uwsd, DomKM, Lisa Oakley, Bowen Li, Aarush Ahuja, prasanth5reddy, brianhendee, mani2106,

<sup>9</sup> <http://learnpython.org/>

<sup>10</sup> <https://discuss.mxnet.io/>

<sup>11</sup> <https://forum.machinelearningcaban.com/c/d21>

mtn, lkevinzc, caojilin, Lakshya, Fiete Lüer, Surbhi Vijayvargeeya, Muhyun Kim, dennismalmgren, adursun, Anirudh Dagar, liqingnz, Pedro Larroy, lgov, ati-ozgur, Jun Wu, Matthias Blume, Lin Yuan, geogunow, Josh Gardner, Maximilian Böther, Rakib Islam, Leonard Lausen, Abhinav Upadhyay, rongruosong, Steve Sedlmeyer, ruslo, Rafael Schlatter, liusy182, Giannis Pappas, ruslo, ati-ozgur, qbaza, dchoi77, Adam Gerson. Notably, Brent Werness (Amazon) và Rachel Hu (Amazon) đồng tác giả chương *Toán học cho Học sâu* trong Phụ lục với chúng tôi và là những người đóng góp chính cho chương đó.

Chúng tôi cảm ơn Amazon Web Services, đặc biệt là Swami Sivasubramanian, Raju Gulabani, Charlie Bell, và Andrew Jassy vì sự hỗ trợ hào phóng của họ trong việc viết cuốn sách này. Nếu không có thời gian, tài nguyên, mọi sự thảo luận cùng các đồng nghiệp, cũng như những khuyến khích liên tục, sự xuất hiện của cuốn sách này sẽ không thể thành hiện thực.

## Tóm tắt

- Học sâu đã cách mạng hóa nhận dạng mẫu, đưa ra công nghệ cốt lõi hiện được sử dụng trong nhiều ứng dụng công nghệ, bao gồm thị giác máy, xử lý ngôn ngữ tự nhiên và nhận dạng giọng nói tự động.
- Để áp dụng thành công kỹ thuật học sâu, bạn phải hiểu được cách biến đổi bài toán, toán học của việc mô hình hóa, các thuật toán để khớp mô hình theo dữ liệu của bạn, và các kỹ thuật để thực hiện tất cả những điều này.
- Cuốn sách này là một nguồn tài liệu toàn diện, bao gồm các diễn giải, hình minh họa, công thức toán và mã nguồn, tất cả trong một.
- Để tìm câu trả lời cho các câu hỏi liên quan đến cuốn sách này, hãy truy cập diễn đàn của chúng tôi tại <https://discuss.mxnet.io/>. (Diễn đàn của nhóm dịch tại <https://forum.machinelearningcaban.com/c/d2l>).
- Apache MXNet là một thư viện mạnh mẽ để lập trình các mô hình học sâu và chạy chúng song song trên các GPU.
- Gluon là một thư viện cấp cao giúp việc viết mã các mô hình học sâu một cách dễ dàng bằng cách sử dụng Apache MXNet.
- Conda là trình quản lý gói Python đảm bảo tất cả các phần mềm phụ thuộc đều được đáp ứng đủ.
- Tất cả các notebook đều có thể tải xuống từ GitHub và các cấu hình conda cần thiết để chạy mã nguồn của cuốn sách này được viết trong tệp môi trường.yml.
- Nếu bạn có kế hoạch chạy mã này trên GPU, đừng quên cài đặt các driver cần thiết và cập nhật cấu hình của bạn.

## Bài tập

1. Đăng ký tài khoản diễn đàn của cuốn sách tại [discussion.mxnet.io<sup>12</sup>](https://discuss.mxnet.io) (và của nhóm dịch tại <https://forum.machinelearningcoban.com>).
2. Cài đặt Python trên máy tính.
3. Làm theo hướng dẫn ở các liên kết đến diễn đàn ở cuối phần này, ở các liên kết diễn đàn đó bạn sẽ có thể nhận được giúp đỡ và thảo luận về cuốn sách cũng như tìm ra câu trả lời cho câu hỏi của bạn bằng cách thu hút các tác giả và cộng đồng lớn hơn.
4. Tạo một tài khoản trên diễn đàn và giới thiệu bản thân.

## Thảo luận

- [Tiếng Anh<sup>13</sup>](#)
- [Tiếng Việt<sup>14</sup>](#)

## Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Vũ Hữu Tiệp
- Sầm Thế Hải
- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Ngô Thế Anh Khoa
- Trần Thị Hồng Hạnh
- Đoàn Võ Duy Thành

---

<sup>12</sup> <https://discuss.mxnet.io/>

<sup>13</sup> <https://discuss.mxnet.io/t/2311>

<sup>14</sup> <https://forum.machinelearningcoban.com/c/d21>

# 1 | Cài đặt

Để sẵn sàng cho việc thực hành, bạn cần một môi trường để chạy Python, Jupyter Notebook, các thư viện liên quan và mã nguồn cần thiết cho những bài tập trong cuốn sách này.

## 1.1 Cài đặt Miniconda

Cách đơn giản nhất để bắt đầu là cài đặt [Miniconda](#)<sup>15</sup>. Phiên bản Python 3.x được khuyên dùng. Bạn có thể bỏ qua những bước sau đây nếu đã cài đặt conda. Tải về tập tin sh tương ứng của Miniconda từ trang web và sau đó thực thi phần cài đặt từ cửa sổ dòng lệnh sử dụng câu lệnh sh <FILENAME> -b. Với người dùng macOS:

```
# The file name is subject to changes  
sh Miniconda3-latest-MacOSX-x86_64.sh -b
```

Với người dùng Linux:

```
# The file name is subject to changes  
sh Miniconda3-latest-Linux-x86_64.sh -b
```

Tiếp theo, khởi tạo shell để chạy trực tiếp lệnh conda.

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

Bây giờ, hãy đóng và mở lại shell hiện tại. Bạn đã có thể tạo một môi trường mới bằng lệnh sau:

```
conda create --name d2l -y
```

## 1.2 Tải về notebook của D2L

Tiếp theo, ta cần tải về mã nguồn của cuốn sách này. Bạn có thể tải mã nguồn từ [đường dẫn này](#)<sup>16</sup> và giải nén. Một cách khác, nếu bạn đã cài đặt sẵn unzip (nếu chưa, hãy chạy lệnh sudo apt install unzip):

<sup>15</sup> <https://conda.io/en/latest/miniconda.html>

<sup>16</sup> <https://d2l.ai/d2l-en-0.7.0.zip>

```
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, ta sẽ kích hoạt môi trường d2l và cài đặt pip. Hãy nhập y để trả lời các câu hỏi theo sau lệnh này:

```
conda activate d2l  
conda install python=3.7 pip -y
```

### 1.3 Cài đặt Framework và Gói thư viện d2l

<!--

Before installing the deep learning framework, please first check whether or not you have proper GPUs on your machine (the GPUs that power the display on a standard laptop do not count for our purposes). If you are installing on a GPU server, proceed to [Hỗ trợ GPU](#) (page 13) for instructions to install a GPU-supported version. -->

Trước khi cài đặt framework học sâu, hãy kiểm tra thiết bị của bạn xem có GPU (card màn hình) đúng chuẩn hay không (không phải những GPU tích hợp hỗ trợ hiển thị trên các máy tính xách tay thông thường). Nếu bạn đang cài đặt trên một máy chủ GPU, hãy tiến hành theo [Hỗ trợ GPU](#) (page 13) để cài đặt phiên bản MXNet có hỗ trợ GPU.

<!-- Otherwise, you can install the CPU version. That will be more than enough horsepower to get you through the first few chapters but you will want to access GPUs before running larger models.

-->

Ngược lại, bạn có thể cài đặt phiên bản chỉ sử dụng CPU. Phiên bản này cũng đủ để có thể tiến hành các chương đầu tiên nhưng bạn sẽ cần sử dụng GPU để có thể chạy những mô hình lớn hơn.

```
pip install mxnet==1.6.0
```

Ta cũng sẽ cài đặt gói thư viện d2l mà bao gồm các hàm và lớp thường xuyên được sử dụng trong cuốn sách này.

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

Khi đã cài đặt xong, ta mở notebook Jupyter lên bằng cách chạy lệnh sau:

```
jupyter notebook
```

Bây giờ, bạn có thể truy cập vào địa chỉ <http://localhost:8888> (thường sẽ được tự động mở) trên trình duyệt Web. Sau đó ta đã có thể chạy mã nguồn trong từng phần của cuốn sách này. Lưu ý là luôn luôn thực thi lệnh `conda activate d2l` để kích hoạt môi trường trước khi chạy mã nguồn trong sách cũng như khi cập nhật MXNet hoặc gói thư viện d2l. Thực thi lệnh `conda deactivate` để thoát khỏi môi trường.

## 1.4 Hỗ trợ GPU

<!--

By default, the deep learning framework is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA<sup>17</sup>, then you should install a GPU-enabled version. If you have installed the CPU-only version, you may need to remove it first by running:

-->

Mặc định framework học sâu được cài đặt không hỗ trợ GPU để đảm bảo có thể chạy trên bất kỳ máy tính nào (bao gồm phần lớn các máy tính xách tay). Một phần của cuốn sách này yêu cầu hoặc khuyến khích chạy trên GPU. Nếu máy tính của bạn có card đồ họa của NVIDIA và đã cài đặt CUDA<sup>18</sup>, thì bạn nên cài đặt bản MXNet có hỗ trợ GPU. Trong trường hợp bạn đã cài đặt phiên bản dành riêng cho CPU, bạn có thể cần xoá nó trước bằng cách chạy lệnh:

<!--

-->

```
pip uninstall mxnet
```

<!--

Then we need to find the CUDA version you installed. You may check it through nvcc --version or cat /usr/local/cuda/version.txt. Assume that you have installed CUDA 10.1, then you can install with the following command:

-->

Sau đó, ta cần tìm phiên bản CUDA mà bạn đã cài đặt. Bạn có thể kiểm tra thông qua lệnh nvcc --version hoặc cat /usr/local/cuda/version.txt. Giả sử, bạn đã cài đặt CUDA 10.1, bạn có thể cài đặt với lệnh sau:

```
# For Windows users
pip install mxnet-cu101==1.6.0b20190926
# For Linux and macOS users
pip install mxnet-cu101==1.6.0
```

<!--

You may change the last digits according to your CUDA version, e.g., cu100 for CUDA 10.0 and cu90 for CUDA 9.0.

-->

Bạn có thể thay đổi những chữ số cuối theo phiên bản CUDA của mình. Ví dụ, cu100 cho phiên bản CUDA 10.0 và cu90 cho phiên bản CUDA 9.0.

<sup>17</sup> <https://developer.nvidia.com/cuda-downloads>

<sup>18</sup> <https://developer.nvidia.com/cuda-downloads>

## 1.5 Bài tập

1. Tải xuống mã nguồn dành cho cuốn sách và cài đặt môi trường chạy.

## 1.6 Thảo luận

- Tiếng Anh: MXNet<sup>19</sup>, PyTorch<sup>20</sup>, TensorFlow<sup>21</sup>
- Tiếng Việt<sup>22</sup>

## 1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Phạm Hồng Vinh
- Sầm Thế Hải
- Nguyễn Cảnh Thướng
- Lê Khắc Hồng Phúc
- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp

---

<sup>19</sup> <https://discuss.d2l.ai/t/23>

<sup>20</sup> <https://discuss.d2l.ai/t/24>

<sup>21</sup> <https://discuss.d2l.ai/t/436>

<sup>22</sup> <https://forum.machinelearningcoban.com/c/d2l>

# 2 | Ký hiệu

Các ký hiệu sử dụng trong cuốn sách này được tổng hợp dưới đây.

## 2.1 Số

- $x$ : một số vô hướng
- $\mathbf{x}$ : một vector
- $\mathbf{X}$ : một ma trận
- $X$ : một tensor
- $\mathbf{I}$ : một ma trận đồng nhất
- $x_i, [\mathbf{x}]_i$ : phần tử thứ  $i$  của vector  $\mathbf{x}$
- $x_{ij}, [\mathbf{X}]_{ij}$ : phần tử ở hàng thứ  $i$ , cột thứ  $j$  của ma trận  $\mathbf{X}$

## 2.2 Lý thuyết Tập hợp

- $\mathcal{X}$ : một tập hợp
- $\mathbb{Z}$ : tập hợp các số nguyên
- $\mathbb{R}$ : tập hợp các số thực
- $\mathbb{R}^n$ : tập các vector thực trong không gian  $n$  chiều
- $\mathbb{R}^{a \times b}$ : tập hợp các ma trận thực với  $a$  hàng và  $b$  cột
- $\mathcal{A} \cup \mathcal{B}$ : hợp của hai tập hợp  $\mathcal{A}$  và  $\mathcal{B}$
- $\mathcal{A} \cap \mathcal{B}$ : giao của hai tập hợp  $\mathcal{A}$  và  $\mathcal{B}$
- $\mathcal{A} \setminus \mathcal{B}$ : hiệu của tập  $\mathcal{A}$  và tập  $\mathcal{B}$  (là tập hợp gồm các phần tử thuộc  $\mathcal{A}$  nhưng không thuộc  $\mathcal{B}$ )

## 2.3 Hàm số và các Phép toán

- $f(\cdot)$ : một hàm số
- $\log(\cdot)$ : logarit tự nhiên
- $\exp(\cdot)$ : hàm  $e$  mũ
- $\mathbf{1}_{\mathcal{X}}$ : hàm đặc trưng (trả về 1 nếu đối số là một phần tử thuộc  $\mathcal{X}$ , trả về 0 trong trường hợp còn lại).
- $(\cdot)^\top$ : chuyển vị của một vector hoặc một ma trận
- $\mathbf{X}^{-1}$ : nghịch đảo của ma trận  $\mathbf{X}$
- $\odot$ : tích Hadamard (theo từng thành phần)
- $|\mathcal{X}|$ : card (số phần tử) của tập  $\mathcal{X}$
- $\|\cdot\|_p$ : chuẩn  $\ell_p$
- $\|\cdot\|$ : chuẩn  $\ell_2$
- $\langle \mathbf{x}, \mathbf{y} \rangle$ : tích vô hướng của hai vector  $\mathbf{x}$  và  $\mathbf{y}$
- $\sum$ : tổng của một dãy
- $\prod$ : tích của một dãy

## 2.4 Giải tích

- $\frac{dy}{dx}$ : đạo hàm của  $y$  theo  $x$
- $\frac{\partial y}{\partial x}$ : đạo hàm riêng của  $y$  theo  $x$
- $\nabla_{\mathbf{x}} y$ : Gradient của  $y$  theo vector  $\mathbf{x}$
- $\int_a^b f(x) dx$ : tích phân của  $f$  từ  $a$  đến  $b$  theo  $x$
- $\int f(x) dx$ : nguyên hàm của  $f$  theo  $x$

## 2.5 Xác suất và Lý thuyết Thông tin

- $P(\cdot)$ : phân phối xác suất
- $z \sim P$ : biến ngẫu nhiên  $z$  tuân theo phân phối xác suất  $P$
- $P(X | Y)$ : xác suất của  $X$  với điều kiện  $Y$
- $p(x)$ : hàm mật độ xác suất
- $E_x[f(x)]$ : kỳ vọng của  $f$  theo  $x$
- $X \perp Y$ : hai biến ngẫu nhiên  $X$  và  $Y$  là độc lập
- $X \perp Y | Z$ : hai biến ngẫu nhiên  $X$  và  $Y$  là độc lập có điều kiện nếu cho trước biến ngẫu nhiên  $Z$
- $\text{Var}(X)$ : phương sai của biến ngẫu nhiên  $X$

- $\sigma_X$ : độ lệch chuẩn của biến ngẫu nhiên  $X$
- $\text{Cov}(X, Y)$ : hiệp phương sai của hai biến ngẫu nhiên  $X$  và  $Y$
- $\rho(X, Y)$ : độ tương quan của hai biến ngẫu nhiên  $X$  và  $Y$
- $H(X)$ : Entropy của biến ngẫu nhiên  $X$
- $D_{\text{KL}}(P\|Q)$ : phân kỳ KL của hai phân phối  $P$  và  $Q$

## 2.6 Độ phức tạp

- $\mathcal{O}$ : Ký hiệu Big O

## 2.7 Thảo luận

- Tiếng Anh<sup>23</sup>
- Tiếng Việt<sup>24</sup>

## 2.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Vũ Hữu Tiệp
- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc

---

<sup>23</sup> <https://discuss.mxnet.io/t/4367>

<sup>24</sup> <https://forum.machinelearningcoban.com/c/d2l>



## 3 | Giới thiệu

Cho tới tận gần đây, gần như tất cả mọi chương trình máy tính mà chúng ta tương tác hàng ngày đều được tạo ra bởi lập trình viên phần mềm từ những định đề cơ bản. Giả sử chúng ta muốn viết một ứng dụng quản lý hệ thống thương mại điện tử. Sau khi lục lại quanh chiếc bảng trắng để suy nghĩ về vấn đề một cách cẩn kẽ, chúng ta có thể phác thảo một giải pháp vận hành được, phần nào sẽ nhìn giống như sau: (i) người dùng tương tác với ứng dụng thông qua một giao diện chạy trên trình duyệt web hoặc ứng dụng trên điện thoại; (ii) ứng dụng tương tác với một hệ thống cơ sở dữ liệu thương mại để theo dõi trạng thái của từng người dùng và duy trì hồ sơ lịch sử các giao dịch; và (iii) (cũng là cốt lõi của ứng dụng) các logic nghiệp vụ (hay cũng có thể nói *bộ não*) mô tả cách thức xử lí cụ thể của ứng dụng trong từng tình huống có thể xảy ra.

Để xây dựng *bộ não* của ứng dụng này, ta phải xem xét tất cả mọi trường hợp mà chúng ta cho rằng sẽ gặp phải, qua đó đặt ra những quy tắc thích hợp. Ví dụ, mỗi lần người dùng nhấn để thêm một món đồ vào giỏ hàng, ta thêm một trường vào bảng giỏ hàng trong cơ sở dữ liệu, liên kết ID của người dùng với ID của món hàng được yêu cầu. Mặc dù hầu như rất ít lập trình viên có thể làm đúng hết trong lần đầu tiên, (sẽ cần vài lần chạy kiểm tra để xử lý hết được những trường hợp hiểm hóc), hầu như phần lớn ta có thể lập trình được từ những định đề cơ bản và tự tin chạy ứng dụng *trước khi được dùng bởi một khách hàng thực sự nào*. Khả năng phát triển những sản phẩm và hệ thống tự động từ những định đề cơ bản, thường là trong những điều kiện mới lạ, là một kỹ công trong suy luận và nhận thức của con người. Và khi bạn có thể tạo ra một giải pháp hoạt động được trong mọi tình huống, *bạn không nên sử dụng học máy*.

May mắn thay cho cộng đồng đang tăng trưởng của các nhà khoa học về học máy, nhiều tác vụ mà chúng ta muốn tự động hóa không dễ dàng bị khuất phục bởi sự tai tình của con người. Thủ tướng tương bạn đang quay quần bên tẩm bảng trắng với những bộ não thông minh nhất mà bạn biết, nhưng lần này bạn đang đương đầu với một trong những vấn đề dưới đây:

- Viết một chương trình dự báo thời tiết ngày mai, cho biết trước thông tin địa lý, hình ảnh vệ tinh, và một chuỗi dữ liệu thời tiết trong quá khứ.
- Viết một chương trình lấy đầu vào là một câu hỏi, được diễn đạt không theo khuôn mẫu nào, và trả lời nó một cách chính xác.
- Viết một chương trình hiển thị ra cho người dùng những sản phẩm mà họ có khả năng cao sẽ thích, nhưng lại ít có khả năng gặp được khi duyệt qua một cách tự nhiên.

Trong mỗi trường hợp trên, cho dù có là lập trình viên thượng thừa cũng không thể lập trình lên được từ con số không. Có nhiều lý do khác nhau. Đôi khi chương trình mà chúng ta cần lại đi theo một khuôn mẫu thay đổi theo thời gian và chương trình của chúng ta cần phải thích ứng với điều đó. Trong trường hợp khác, mối quan hệ (giả dụ như giữa các điểm ảnh và các hạng mục trừu tượng) có thể là quá phức tạp, yêu cầu hàng ngàn hàng triệu phép tính vượt ngoài khả năng thấu hiểu của nhận thức chúng ta (mặc dù mắt của chúng ta có thể xử lý tác vụ này một cách dễ dàng). Học máy (Machine Learning - ML) là lĩnh vực nghiên cứu những kỹ thuật tiên tiến mà có thể *học từ kinh nghiệm*. Khi thuật toán ML tích luỹ thêm nhiều kinh nghiệm, thường là dưới dạng

dữ liệu quan sát hoặc tương tác với môi trường, chất lượng của nó sẽ tăng lên. Tương phản với hệ thống thương mại điện tử tất định của chúng ta, khi mà nó luôn tuân theo cùng logic nghiệp vụ đã có, mặc cho đã tích luỹ thêm bao nhiêu kinh nghiệm, tận cho tới khi lập trình viên tự học và quyết định rằng đã tới lúc cập nhật phần mềm này. Trong cuốn sách này, chúng tôi sẽ dạy cho bạn về những điều căn bản nhất trong học máy, và tập trung đặc biệt vào học sâu, một tập hợp hùng mạnh những kỹ thuật đang thúc đẩy sự đổi mới ở nhiều lĩnh vực khác nhau như thị giác máy tính, xử lý ngôn ngữ tự nhiên, chăm sóc y tế và nghiên cứu cấu trúc gen.

### 3.1 Một ví dụ truyền cảm hứng

Các tác giả của cuốn sách cũng giống như nhiều người lao động khác, cần một tách cà phê trước khi bắt đầu công việc biên soạn của mình. Chúng tôi leo lên xe và bắt đầu lái. Sở hữu chiếc iPhone, Alex gọi “Hey Siri” để đánh thức hệ thống nhận dạng giọng nói của điện thoại. Sau đó Mu ra lệnh “chì đường đến quán cà phê Blue Bottle”. Chiếc điện thoại nhanh chóng hiển thị bản ghi thoại (*transcription*) của câu lệnh. Nó cũng nhận ra rằng chúng tôi đang yêu cầu chỉ dẫn đường đi và tự khởi động ứng dụng Bản đồ để hoàn thành yêu cầu đó. Khoi động xong, ứng dụng Bản đồ tự xác định một vài lộ trình tới đích. Đến mỗi tuyến đường, ứng dụng lại cập nhật và hiển thị thời gian di chuyển dự tính mới. Mặc dù đây chỉ là câu chuyện dựng lên cho mục đích giảng dạy, nó cũng cho thấy chỉ trong khoảng vài giây, những tương tác hàng ngày của chúng ta với chiếc điện thoại thông minh có thể liên quan đến nhiều mô hình học máy.

Tưởng tượng rằng ta mới viết một chương trình để phản hồi một *hiệu lệnh đánh thức* như là “Alexa”, “Okay, Google” hoặc “Siri”. Hãy thử tự viết nó chỉ với một chiếc máy tính và một trình soạn thảo mã nguồn như minh họa trong Fig. 3.1.1. Bạn sẽ bắt đầu viết một chương trình như vậy như thế nào? Thủ nghĩ xem... vấn đề này khó đấy. Cứ mỗi giây, chiếc mic sẽ thu thập cỡ tầm 44,000 mẫu tín hiệu. Mỗi mẫu là một giá trị biên độ của sóng âm. Quy tắc đáng tin cậy nào có thể từ một đoạn âm thanh thô đưa ra các dự đoán {có, không} để xác định đoạn âm thanh đó có chứa hiệu lệnh đánh thức hay không? Nếu bạn không biết xử lý điều này như thế nào, thì cũng đừng lo lắng. Chúng tôi cũng không biết làm cách nào để viết một chương trình như vậy từ đầu. Đó là lý do vì sao chúng tôi sử dụng học máy.

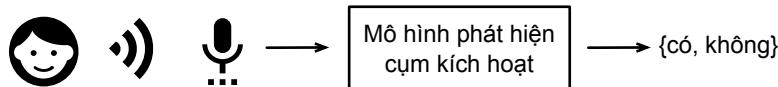


Fig. 3.1.1: Xác định một hiệu lệnh đánh thức

Thủ thuật là thế này. Ngay cả khi không thể giải thích cụ thể cho một cái máy tính về cách ánh xạ từ đầu vào đến đầu ra như thế nào, thì chúng ta vẫn có khả năng làm việc đó bằng bộ não của mình. Hay nói cách khác, thậm chí nếu chúng ta không biết *cách lập trình một cái máy tính* để nhận dạng từ “Alexa”, chính chúng ta lại có *khả năng* nhận thức được từ “Alexa”. Với khả năng này, chúng ta có thể thu thập một *tập dữ liệu* lớn các mẫu âm thanh kèm nhãn mà *có chứa* hoặc *không chứa* hiệu lệnh đánh thức. Trong cách tiếp cận học máy, chúng ta không thiết kế một hệ thống *rõ ràng* để nhận dạng hiệu lệnh đánh thức. Thay vào đó, chúng ta định nghĩa một chương trình linh hoạt mà hành vi của nó được xác định bởi những *tham số*. Sau đó chúng ta sử dụng tập dữ liệu để xác định bộ các tham số tốt nhất có khả năng cải thiện chất lượng của chương trình, cũng như thỏa mãn một số yêu cầu về chất lượng trong nhiệm vụ được giao.

Bạn có thể coi những tham số như các num var có thể điều chỉnh để thay đổi hành vi của chương trình. Sau khi đã cố định các tham số, chúng ta gọi chương trình này là một *mô hình*. Tập hợp của

tất cả các chương trình khác nhau (ánh xạ đầu vào-đầu ra) mà chúng ta có thể tạo ra chỉ bằng cách thay đổi các tham số được gọi là một *nhóm* các mô hình. Và *chương trình học tham số* sử dụng tập dữ liệu để chọn ra các tham số được gọi là *thuật toán học*.

Trước khi tiếp tục và bắt đầu với các thuật toán học, chúng ta phải định nghĩa rõ ràng vấn đề, hiểu chính xác bản chất của đầu vào và đầu ra và lựa chọn một nhóm mô hình thích hợp. Trong trường hợp này, mô hình của chúng ta nhận *đầu vào* là một đoạn âm thanh và *đầu ra* là một giá trị trong {đúng, sai}. Nếu tất cả diễn ra như kế hoạch, mô hình thông thường sẽ dự đoán chính xác liệu đoạn âm thanh có hay không chứa hiệu lệnh kích hoạt.

Nếu chúng ta lựa chọn đúng nhóm mô hình, sẽ tồn tại một cách thiết lập các nút vặn mà mô hình sẽ đưa ra đúng mỗi khi nghe thấy từ “Alexa”. Bởi vì việc lựa chọn hiệu lệnh đánh thức nào là tùy ý, chúng ta sẽ muốn có một nhóm mô hình đủ mạnh để trong trường hợp với một thiết lập khác của các nút quay, nó sẽ đưa ra kết quả đúng mỗi khi nghe từ “Apricot” (“quả mơ”). Bằng trực giác ta có thể nhận thấy rằng việc nhận dạng “Alexa” và nhận dạng “Apricot” cũng tương tự nhau và có thể sử dụng chung một nhóm mô hình. Tuy nhiên, trong trường hợp có sự khác biệt về bản chất ở đầu vào và đầu ra, chẳng hạn như việc ánh xạ từ hình ảnh sang chủ thích, hoặc từ câu tiếng Anh sang câu tiếng Trung thì ta có thể sẽ phải sử dụng các nhóm mô hình hoàn toàn khác nhau.

Dễ dàng nhận thấy, nếu như chúng ta chỉ thiết lập một cách ngẫu nhiên các nút vặn, thì mô hình gần như sẽ không có khả năng nhận dạng “Alexa”, “Apricot” hay bất cứ từ tiếng Anh nào khác. Trong học sâu, *học* là quá trình khám phá ra thiết lập đúng của các nút vặn để mô hình có thể hành xử như chúng ta mong muốn.

Quá trình huấn luyện thường giống như mô tả trong hình Fig. 3.1.2:

1. Khởi tạo mô hình một cách ngẫu nhiên. Lúc này nó vẫn chưa thể thực hiện bất kỳ tác vụ có ích nào.
2. Thu thập một số dữ liệu đã được gán nhãn (ví dụ như đoạn âm thanh kèm nhãn {có, không} tương ứng).
3. Thay đổi các nút vặn để mô hình dự đoán chính xác hơn trên các mẫu.
4. Lặp lại cho đến khi có một mô hình hoạt động tốt.

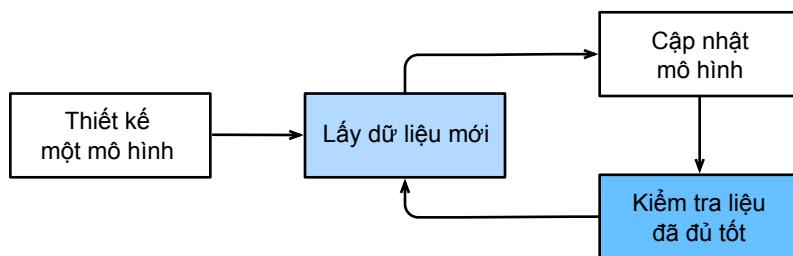


Fig. 3.1.2: Một quá trình huấn luyện điển hình

Tóm lại, thay vì tự lập trình một chương trình nhận dạng từ đánh thức, ta tạo ra một chương trình có thể *học* cách nhận dạng các từ đánh thức *khi được cho xem một tập lớn những ví dụ đã được gán nhãn*. Ta có thể gọi việc xác định hành vi của một chương trình bằng cách cho nó xem một tập dữ liệu là *lập trình với dữ liệu*. Chúng ta có thể “lập trình” một bộ phát hiện mèo bằng cách cung cấp cho hệ thống học máy nhiều mẫu ảnh chó và mèo, ví dụ như các hình ảnh dưới đây:

			
mèo	mèo	chó	chó

Bằng cách này bộ phát hiện sẽ dần học cách trả về một số dương lớn nếu đó là một con mèo, hoặc một số âm lớn nếu đó là một con chó, hoặc một giá trị gần với không nếu nó không chắc chắn. Và đây mới chỉ là một ví dụ nhỏ về những gì mà học máy có thể làm được.

Học sâu chỉ là một trong nhiều phương pháp phổ biến để giải quyết những bài toán học máy. Tới giờ chúng ta mới chỉ nói tổng quát về học máy chứ chưa nói về học sâu. Để thấy được tại sao học sâu lại quan trọng, ta nên dừng lại một chút để làm rõ một vài điểm thiết yếu.

Thứ nhất, những vấn đề mà chúng ta đã thảo luận-học từ tín hiệu âm thanh thô, từ những giá trị điểm ảnh của tấm ảnh, hoặc dịch những câu có độ dài bất kỳ sang một ngôn ngữ khác-là những vấn đề học sâu có thể xử lý tốt còn học máy truyền thống thì không. Mô hình sâu thực sự sâu theo nghĩa nó có thể học nhiều *tầng* tính toán. Những mô hình đa tầng (hoặc có thứ bậc) này có khả năng xử lý dữ liệu tri giác mức thấp theo cái cách mà những công cụ trước đây không thể. Trước đây, một phần quan trọng trong việc áp dụng học máy vào các bài toán này là tìm thủ công những kỹ thuật biến đổi dữ liệu sang một hình thức mà những mô hình *nông* có khả năng xử lý. Một lợi thế then chốt của học sâu là nó không chỉ thay thế mô hình *nông* ở thành phần cuối cùng của pipeline học tập truyền thống mà còn thay thế quá trình thiết kế đặc trưng tốn nhiều công sức. Thứ hai, bằng cách thay thế các kỹ thuật “tiền xử lý theo từng phân ngành”, học sâu đã loại bỏ ranh giới giữa thị giác máy tính, nhận dạng tiếng nói, xử lý ngôn ngữ tự nhiên, tin học y khoa và các lĩnh vực khác, cung cấp một tập hợp các công cụ xử lý những loại bài toán khác nhau.

## 3.2 Các thành phần chính: Dữ liệu, Mô hình và Thuật toán

Trong ví dụ về *tù đánh thức*, chúng tôi đã mô tả một bộ dữ liệu bao gồm các đoạn âm thanh và các nhãn nhị phân, giúp các bạn hiểu một cách chung chung về cách *huấn luyện* một mô hình để phân loại các đoạn âm thanh. Với loại bài toán này, ta cố gắng dự đoán một *nhãn* chưa biết với *đầu vào* cho trước, dựa trên tập dữ liệu cho trước bao gồm các mẫu đã được gán nhãn. Đây là ví dụ về bài toán *học có giám sát* và chỉ là một trong số rất nhiều *dạng* bài toán học máy khác nhau mà chúng ta sẽ học trong các chương sau. Trước hết, chúng tôi muốn giải thích rõ hơn về các thành phần cốt lõi sẽ theo chúng ta xuyên suốt tất cả các bài toán học máy:

1. *Dữ liệu* mà chúng ta có thể học.
2. Một *mô hình* về cách biến đổi dữ liệu.
3. Một hàm *mất mát* định lượng *độ lỗi* của mô hình.
4. Một *thuật toán* điều chỉnh các tham số của mô hình để giảm thiểu mất mát.

### 3.2.1 Dữ liệu

Có một sự thật hiển nhiên là bạn không thể làm khoa học dữ liệu mà không có dữ liệu. Chúng ta sẽ tốn rất nhiều giấy mực để cân nhắc chính xác những gì cấu thành nên dữ liệu, nhưng bây giờ chúng ta sẽ rẽ sang khía cạnh thực tế và tập trung vào các thuộc tính quan trọng cần quan tâm. Thông thường, chúng ta quan tâm đến một bộ *mẫu* (còn được gọi là *điểm dữ liệu*, *ví dụ* hoặc *trường hợp*). Để làm việc với dữ liệu một cách hữu ích, chúng ta thường cần có một cách biểu diễn chúng phù hợp dưới dạng số. Mỗi *ví dụ* thường bao gồm một bộ thuộc tính số gọi là *đặc trưng*. Trong các bài toán học có giám sát ở trên, một đặc trưng đặc biệt được chọn làm *mục tiêu dự đoán*, (còn được gọi là *nhận* hoặc *biến phụ thuộc*). Các đặc trưng mà mô hình dựa vào để đưa ra dự đoán có thể được gọi đơn giản là các *đặc trưng*, (hoặc thường là *đầu vào*, *hiệp biến* hoặc *biến độc lập*).

Nếu chúng ta đang làm việc với dữ liệu hình ảnh, mỗi bức ảnh riêng lẻ có thể tạo thành một *mẫu* được biểu diễn bởi một danh sách các giá trị số theo thứ tự tương ứng với độ sáng của từng pixel. Một bức ảnh màu có kích thước  $200 \times 200$  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 dương cho từng vị trí trong không gian. Trong một tác vụ truyền thống hơn, chúng ta có thể cố gắng dự đoán xem một bệnh nhân liệu có cơ hội sống sót hay không, dựa trên bộ đặc trưng tiêu chuẩn cho trước như tuổi, các triệu chứng quan trọng, thông số chẩn đoán, .v.v.

Khi mỗi mẫu được biểu diễn bởi cùng một số lượng các giá trị, ta nói rằng dữ liệu bao gồm các vector có *độ dài cố định* và ta mô tả độ dài (không đổi) của vector là *chiều* của dữ liệu. Bạn có thể hình dung, chiều dài cố định có thể là một thuộc tính thuận tiện. Nếu ta mong muốn huấn luyện một mô hình để nhận biết ứng thư qua hình ảnh từ kính hiển vi, độ dài cố định của đầu vào sẽ giúp ta loại bỏ một vấn đề cần quan tâm.

Tuy nhiên, không phải tất cả dữ liệu có thể được dễ dàng biểu diễn dưới dạng vector có độ dài cố định. Đôi khi ta có thể mong đợi hình ảnh từ kính hiển vi đến từ thiết bị tiêu chuẩn, nhưng ta không thể mong đợi hình ảnh được khai thác từ Internet sẽ hiển thị với cùng độ phân giải hoặc tỉ lệ được. Đối với hình ảnh, ta có thể tính đến việc cắt xén nhằm đưa chúng về kích thước tiêu chuẩn, nhưng chiến lược này chỉ đưa ta đến đấy mà thôi. Và ta có nguy cơ sẽ mất đi thông tin trong các phần bị cắt bỏ. Hơn nữa, dữ liệu văn bản không thích hợp với cách biểu diễn dưới dạng vector có độ dài cố định. Suy xét một chút về những đánh giá của khách hàng để lại trên các trang Thương mại điện tử như Amazon, IMDB hoặc TripAdvisor. Ta có thể thấy có những bình luận ngắn gọn như: “nó bốc mùi!”, một số khác thì bình luận lan man hàng trang. Một lợi thế lớn của học sâu so với các phương pháp truyền thống đó là các mô hình học sâu hiện đại có thể xử lý dữ liệu có *độ dài biến đổi* một cách uyển chuyển hơn.

Nhìn chung, chúng ta có càng nhiều dữ liệu thì công việc sẽ càng dễ dàng hơn. Khi ta có nhiều dữ liệu hơn, ta có thể huấn luyện ra những mô hình mạnh mẽ hơn và ít phụ thuộc hơn vào các giả định được hình thành từ trước. Việc chuyển từ dữ liệu nhỏ sang dữ liệu lớn là một đóng góp chính cho sự thành công của học sâu hiện đại. Để cho rõ hơn, nhiều mô hình thú vị nhất trong học sâu có thể không hoạt động nếu như không có bộ dữ liệu lớn. Một số người vẫn áp dụng học sâu với số dữ liệu ít ỏi mà mình có được, nhưng trong trường hợp này nó không tốt hơn các cách tiếp cận truyền thống.

Cuối cùng, có nhiều dữ liệu và xử lý dữ liệu một cách khéo léo thôi thì chưa đủ. Ta cần những dữ liệu *đúng*. Nếu dữ liệu mang đầy lỗi, hoặc nếu các đặc trưng được chọn lại không dự đoán được số lượng mục tiêu cần quan tâm, việc học sẽ thất bại. Tình huống trên có thể được khái quát bởi thuật ngữ: *đưa rác vào thì nhận rác ra* (*garbage in, garbage out*). Hơn nữa, chất lượng dự đoán kém không phải hậu quả tiềm tàng duy nhất. Trong các ứng dụng học máy có tính nhạy cảm như: dự đoán hành vi phạm pháp, sàng lọc hồ sơ cá nhân và 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 dạng lỗi thường thấy xảy ra trong các

bộ dữ liệu là khi một nhóm người không tồn tại trong dữ liệu huấn luyện. Hãy hình dung khi áp dụng một hệ thống nhận diện ung thư da trong thực tế mà trước đây nó chưa từng thấy qua da màu đen. Thất bại cũng có thể xảy ra khi dữ liệu không đại diện đầy đủ và chính xác cho một số nhóm người, nhưng lại đánh giá nhóm người này dựa vào định kiến của xã hội. Một ví dụ, nếu như các quyết định tuyển dụng trong quá khứ được sử dụng để huấn luyện một mô hình dự đoán sẽ được sử dụng nhằm sàng lọc sơ yếu lý lịch, thì các mô hình học máy có thể vô tình học được từ những bất công trong quá khứ. Lưu ý rằng tất cả vấn đề trên có thể xảy ra mà không hề có tác động xấu nào của nhà khoa học dữ liệu hoặc thậm chí họ còn không ý thức được về các vấn đề đó.

### 3.2.2 Mô hình

Phần lớn học máy đều liên quan đến việc *biến đổi* dữ liệu theo một cách nào đó. Có thể ta muốn xây dựng một hệ thống nhận ảnh đầu vào và dự đoán *mức độ cười* của khuôn mặt trong ảnh. Hoặc đó cũng có thể là một hệ thống nhận vào dữ liệu đo đặc từ cảm biến và dự đoán độ *bình thường* hay *bất thường* của chúng. Ở đây chúng ta gọi *mô hình* là một hệ thống tính toán nhận đầu vào là một dạng dữ liệu và sau đó trả về kết quả dự đoán, có thể ở một dạng dữ liệu khác. Cụ thể, ta quan tâm tới các mô hình thống kê mà ta có thể ước lượng được từ dữ liệu. Dù các mô hình đơn giản hoàn toàn có thể giải quyết các bài toán đơn giản phù hợp, những bài toán được đề cập tới trong cuốn sách này sẽ đẩy các phương pháp cổ điển tới giới hạn của chúng. Điểm khác biệt chính của học sâu so với các phương pháp cổ điển là các mô hình mạnh mẽ mà nó nhắm vào. Những mô hình đó bao gồm rất nhiều phép biến đổi dữ liệu liên tiếp, được liên kết với nhau từ trên xuống dưới, và đó cũng là ý nghĩa của cái tên “học sâu”. Trong quá trình thảo luận về các mạng nơ-ron sâu, ta cũng sẽ nhắc tới các phương pháp truyền thống.

### 3.2.3 Hàm mục tiêu

Trước đó, chúng tôi có giới thiệu học máy là việc “học từ kinh nghiệm”. *Học* ở đây tức là việc *tiến bộ* ở một tác vụ nào đó theo thời gian. Nhưng ai biết được như thế nào là tiến bộ? Thủ tướng tượng ta đang đề xuất cập nhật mô hình, nhưng một số người có thể có bất đồng về việc bản cập nhật này có giúp cải thiện mô hình hay không.

Để có thể phát triển một mô hình toán học chính quy cho học máy, chúng ta cần những phép đo chính quy xem mô hình đang tốt (hoặc tệ) như thế nào. Trong học máy, hay rộng hơn là lĩnh vực tối ưu hoá, ta gọi chúng là các hàm mục tiêu (*objective function*). Theo quy ước, ta thường định nghĩa các hàm tối ưu sao cho giá trị càng thấp thì mô hình càng tốt. Nhưng đó cũng chỉ là một quy ước ngầm. Bạn có thể lấy một hàm  $f$  sao cho giá trị càng cao thì càng tốt, sau đó đặt một hàm tương đương  $f' = -f$ , có giá trị càng thấp thì mô hình càng tốt. Chính vì ta mong muốn hàm có giá trị thấp, nó còn được gọi là *hàm mất mát* (*loss function*) và *hàm chi phí* (*cost function*).

Khi muốn dự đoán một giá trị số, hàm mục tiêu phổ biến nhất là hàm bình phương sai số  $(y - \hat{y})^2$ . Với bài toán phân loại, mục tiêu phổ biến nhất là cực tiểu hóa tỉ lệ lỗi, tức tỉ lệ mẫu mà dự đoán của mô hình lệch với nhãn thực tế. Một vài hàm mục tiêu (ví dụ như bình phương sai số) khá dễ tối ưu hóa. Các hàm khác (như tỉ lệ lỗi) lại khó tối ưu hóa trực tiếp, có thể do các hàm này không khả vi hoặc những vấn đề khác. Trong những trường hợp như vậy, ta thường tối ưu hóa một *hàm mục tiêu thay thế* (*surrogate objective*).

Thông thường, hàm mất mát được định nghĩa theo các tham số mô hình và phụ thuộc vào tập dữ liệu. Những giá trị tham số mô hình tốt nhất được học bằng cách cực tiểu hóa hàm mất mát trên một *tập huấn luyện* bao gồm các *mẫu* được thu thập cho việc huấn luyện. Tuy nhiên, mô hình hoạt động tốt trên tập huấn luyện không có nghĩa là nó sẽ hoạt động tốt trên dữ liệu kiểm tra (mà mô hình chưa nhìn thấy). Bởi vậy, ta thường chia dữ liệu sẵn có thành hai phần: dữ liệu huấn luyện

(để khớp các tham số mô hình) và dữ liệu kiểm tra (được giữ lại cho việc đánh giá). Sau đó ta quan sát hai đại lượng:

- **Lỗi huấn luyện:** Lỗi trên dữ liệu được dùng để huấn luyện mô hình. Bạn có thể coi nó như điểm của một sinh viên trên bài thi thử để chuẩn bị cho bài thi thật. Ngay cả khi kết quả thi thử khả quan, không thể đảm bảo rằng bài thi thật sẽ đạt kết quả tốt.
- **Lỗi kiểm tra:** Đây là lỗi trên tập kiểm tra (không dùng để huấn luyện mô hình). Đại lượng này có thể chênh lệch đáng kể so với lỗi huấn luyện. Khi một mô hình hoạt động tốt trên tập huấn luyện nhưng lại không có khả năng tổng quát hóa trên dữ liệu chưa gặp, ta nói rằng mô hình bị *quá khớp* (overfit). Theo ngôn ngữ thường ngày, đây là hiện tượng “học lệch tủ” khi kết quả bài thi thật rất kém mặc dù có kết quả cao trong bài thi thử.

### 3.2.4 Các thuật toán tối ưu

Một khi ta có dữ liệu, một mô hình và một hàm mục tiêu rõ ràng, ta cần một thuật toán có khả năng tìm kiếm các tham số khả dĩ tốt nhất để cực tiểu hóa hàm mất mát. Các thuật toán tối ưu phổ biến nhất cho mạng nơ-ron đều theo một hướng tiếp cận gọi là hạ gradient. Một cách ngắn gọn, tại mỗi bước và với mỗi tham số, ta kiểm tra xem hàm mất mát thay đổi như thế nào nếu ta thay đổi tham số đó bởi một lượng nhỏ. Sau đó các tham số này được cập nhật theo hướng làm giảm hàm mất mát.

## 3.3 Các dạng Học Máy

Trong các mục tiếp theo, chúng ta sẽ thảo luận chi tiết hơn một số *dạng* bài toán học máy. Cùng bắt đầu với một danh sách *các mục tiêu*, tức một danh sách các tác vụ chúng ta muốn học máy thực hiện. Chú ý rằng các mục tiêu sẽ được gắn liền với một tập các kỹ thuật để đạt được mục tiêu đó, bao gồm các kiểu dữ liệu, mô hình, kỹ thuật huấn luyện, v.v. Danh sách dưới đây là một tuyển tập các bài toán mà học máy có thể xử lý nhằm tạo động lực cho độc giả, đồng thời cung cấp một ngôn ngữ chung khi nói về những bài toán khác xuyên suốt cuốn sách.

### 3.3.1 Học có giám sát

Học có giám sát giải quyết tác vụ dự đoán *mục tiêu* với *đầu vào* cho trước. Các mục tiêu, thường được gọi là *nhãn*, phần lớn được ký hiệu bằng  $y$ . Dữ liệu đầu vào, thường được gọi là *đặc trưng* hoặc *hiệp biến*, thông thường được ký hiệu là  $\mathbf{x}$ . Mỗi cặp (đầu vào, mục tiêu) được gọi là một *mẫu*. Thi thoảng, khi văn cảnh rõ ràng hơn, chúng ta có thể sử dụng thuật ngữ *các mẫu* để chỉ một tập các đầu vào, ngay cả khi chưa xác định được mục tiêu tương ứng. Ta ký hiệu một mẫu cụ thể với một chỉ số dưới, thường là  $i$ , ví dụ  $(\mathbf{x}_i, y_i)$ . Một tập dữ liệu là một tập của  $n$  mẫu  $\{\mathbf{x}_i, y_i\}_{i=1}^n$ . Mục đích của chúng ta là xây dựng một mô hình  $f_\theta$  ánh xạ đầu vào bất kỳ  $\mathbf{x}_i$  tới một dự đoán  $f_\theta(\mathbf{x}_i)$ .

Một ví dụ cụ thể hơn, trong lĩnh vực chăm sóc sức khoẻ, chúng ta có thể muốn dự đoán liệu một bệnh nhân có bị đau tim hay không. Việc *bị đau tim* hay *không bị đau tim* sẽ là nhãn  $y$ . Dữ liệu đầu vào  $\mathbf{x}$  có thể là các dấu hiệu quan trọng như nhịp tim, huyết áp tâm trương và tâm thu, v.v.

Sự giám sát xuất hiện ở đây bởi để chọn các tham số  $\theta$ , chúng ta (các giám sát viên) cung cấp cho mô hình một tập dữ liệu chứa các *mẫu được gán nhãn*  $(\mathbf{x}_i, y_i)$ , ở đó mỗi mẫu  $\mathbf{x}_i$  tương ứng một nhãn cho trước.

Theo thuật ngữ xác suất, ta thường quan tâm tới việc đánh giá xác suất có điều kiện  $P(y|\mathbf{x})$ . Dù chỉ là một trong số nhiều mô hình trong học máy, học có giám sát là nhân tố chính đem đến sự thành

công cho các ứng dụng của học máy trong công nghiệp. Một phần vì rất nhiều tác vụ có thể được mô tả dưới dạng ước lượng xác suất của một đại lượng chưa biết cho trước trong một tập dữ liệu cụ thể:

- Dự đoán có bị ung thư hay không cho trước một bức ảnh CT.
- Dự đoán bản dịch chính xác trong tiếng Pháp cho trước một câu trong tiếng Anh.
- Dự đoán giá của cổ phiếu trong tháng tới dựa trên dữ liệu báo cáo tài chính của tháng này.

Ngay cả với mô tả đơn giản là “dự đoán mục tiêu từ đầu vào”, học có giám sát đã có nhiều hình thái đa dạng và đòi hỏi đưa ra nhiều quyết định mô hình hóa khác nhau, tùy thuộc vào kiểu, kích thước, số lượng của cặp đầu vào và đầu ra cũng như các yếu tố khác. Ví dụ, ta sử dụng các mô hình khác nhau để xử lý các chuỗi (như chuỗi ký tự hay dữ liệu chuỗi thời gian) và các biểu diễn vector với chiều dài cố định. Chúng ta sẽ đào sâu vào rất nhiều bài toán dạng này thông qua chín phần đầu của cuốn sách.

Một cách dễ hiểu, quá trình học gồm những bước sau: Lấy một tập mẫu lớn với các hiệp biến đã biết trước. Từ đó chọn ra một tập con ngẫu nhiên và thu thập các nhãn gốc cho chúng. Đôi khi những nhãn này có thể đã có sẵn trong dữ liệu (ví dụ, liệu bệnh nhân đã qua đời trong năm tiếp theo?). Trong trường hợp khác, chúng ta cần thuê người gán nhãn cho dữ liệu (ví dụ, gán một bức ảnh vào một hạng mục nào đó).

Những đầu vào và nhãn tương ứng này cùng tạo nên tập huấn luyện. Chúng ta đưa tập dữ liệu huấn luyện vào một thuật toán học có giám sát – một hàm số với đầu vào là tập dữ liệu và đầu ra là một hàm số khác thể hiện *mô hình đã học được*. Cuối cùng, ta có thể đưa dữ liệu chưa nhìn thấy vào mô hình đã học được, sử dụng đầu ra của nó như là giá trị dự đoán của các nhãn tương ứng. Toàn bộ quá trình được mô tả trong Fig. 3.3.1.

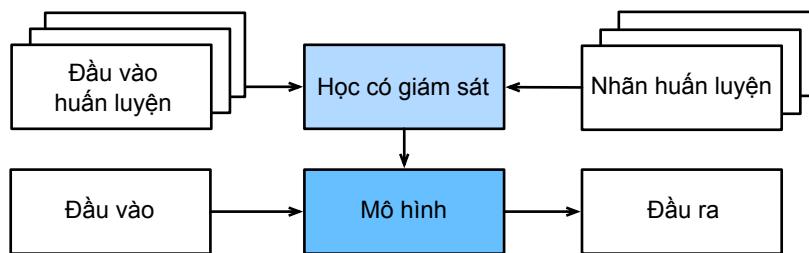


Fig. 3.3.1: Học có giám sát.

## Hồi quy

Có lẽ tác vụ học có giám sát đơn giản nhất là *hồi quy*. Xét ví dụ một tập dữ liệu thu thập được từ cơ sở dữ liệu buôn bán nhà. Chúng ta có thể xây dựng một bảng dữ liệu, ở đó mỗi hàng tương ứng với một nhà và mỗi cột tương ứng với một thuộc tính liên quan nào đó, chẳng hạn như diện tích nhà, số lượng phòng ngủ, số lượng phòng tắm và thời gian (theo phút) để di bộ tới trung tâm thành phố. Trong tập dữ liệu này, mỗi *mẫu* là một căn nhà cụ thể và *vector đặc trưng* tương ứng là một hàng trong bảng.

Nếu bạn sống ở New York hoặc San Francisco và bạn không phải là CEO của Amazon, Google, Microsoft hay Facebook, thì vector đặc trưng (diện tích, số phòng ngủ, số phòng tắm, khoảng cách đi bộ) của căn nhà của bạn có thể có dạng  $[100, 0, 0.5, 60]$ . Tuy nhiên, nếu bạn sống ở Pittsburgh, vector đó có thể là  $[3000, 4, 3, 10]$ . Các vector đặc trưng như vậy là thiết yếu trong hầu hết các thuật toán học máy cổ điển. Chúng ta sẽ tiếp tục ký hiệu vector đặc trưng tương ứng với bất kỳ mẫu  $i$  nào bởi  $\mathbf{x}_i$  và có thể đặt  $X$  là bảng chứa tất cả các vector đặc trưng.

Để xác định một bài toán là *hồi quy* hay không, ta dựa vào đầu ra của nó. Chẳng hạn, bạn đang khảo sát thị trường cho một căn nhà mới. Bạn có thể ước lượng giá thị trường của một căn nhà khi biết những đặc trưng phía trên. Giá trị mục tiêu, hay giá bán của căn nhà, là một số *thực*. Nếu bạn còn nhớ định nghĩa toán học của số *thực*, bạn có thể đang cảm thấy băn khoăn. Nhà đất có lẽ không bao giờ bán với giá được tính bằng các phần nhỏ hơn cent, chứ đừng nói đến việc giá bán được biểu diễn bằng các số vô tỉ. Trong những trường hợp này, khi mục tiêu thực sự là các số rời rạc, nhưng việc làm tròn có thể chấp nhận được, chúng ta sẽ lạm dụng cách dùng từ một chút để tiếp tục mô tả đầu ra và mục tiêu là các số *thực*.

Ta ký hiệu mục tiêu bất kỳ là  $y_i$  (tương ứng với mẫu  $\mathbf{x}_i$ ) và tập tất cả các mục tiêu là  $\mathbf{y}$  (tương ứng với tất cả các mẫu  $X$ ). Khi các mục tiêu mang các giá trị bất kỳ trong một khoảng nào đó, chúng ta gọi đây là bài toán hồi quy. Mục đích của chúng ta là tạo ra một mô hình mà các dự đoán của nó xấp xỉ với các giá trị mục tiêu thực sự. Chúng ta ký hiệu dự đoán mục tiêu của một mẫu là  $\hat{y}_i$ . Đừng quá lo lắng nếu các ký hiệu đang làm bạn nản chí. Chúng ta sẽ tìm hiểu kỹ từng ký hiệu trong các chương tiếp theo.

Rất nhiều bài toán thực tế là các bài toán hồi quy. Dự đoán điểm đánh giá của một người dùng cho một bộ phim có thể được coi là một bài toán hồi quy và nếu bạn thiết kế một thuật toán vĩ đại để đạt được điều này vào năm 2009, bạn có thể đã giành giải thưởng Netflix một triệu Đô-la<sup>25</sup>. Dự đoán thời gian nằm viện của một bệnh nhân cũng là một bài toán hồi quy. Một quy tắc dễ nhớ là khi ta phải trả lời câu hỏi *bao nhiêu* (*bao lâu*, *bao xa*, v.v.), khá chắc chắn đó là bài toán hồi quy.

- “Ca phẫu thuật này sẽ mất bao lâu?”: *hồi quy*
- “Có bao nhiêu chú chó trong bức ảnh?”: *hồi quy*

Tuy nhiên, nếu bạn có thể diễn đạt bài toán của bạn bằng câu hỏi “Đây có phải là \_?” thì khả năng cao đó là bài toán phân loại, một dạng khác của bài toán học có giám sát mà chúng ta sẽ thảo luận trong phần tiếp theo. Ngay cả khi bạn chưa từng làm việc với học máy, bạn có thể đã làm việc với các bài toán hồi quy một cách không chính thức. Ví dụ, hãy tưởng tượng bạn cần sửa chữa đườngống cống và người thợ đã dành  $x_1 = 3$  giờ để thông cống rồi gửi hoá đơn  $y_1 = \$350$ . Bây giờ bạn của bạn thuê cũng thuê người thợ đó trong  $x_2 = 2$  tiếng và cô ấy nhận được hoá đơn là  $y_2 = \$250$ . Nếu một người khác sau đó hỏi bạn giá dự tính phải trả cho việc thông cống, bạn có thể có một vài giả sử có lý, chẳng hạn như mất nhiều thời gian sẽ tốn nhiều tiền hơn. Bạn cũng có thể giả sử rằng có một mức phí cơ bản và sau đó người thợ tính tiền theo giờ. Nếu giả sử này là đúng, thì với hai điểm dữ liệu trên, bạn đã có thể tính được cách mà người thợ tính tiền công: \$100 cho mỗi giờ cộng với \$50 để có mặt tại nhà bạn. Nếu bạn theo được logic tới đây thì bạn đã hiểu ý tưởng tổng quan của hồi quy tuyến tính (và bạn đã vô tình thiết kế một mô hình tuyến tính với thành phần điều chỉnh).

Trong trường hợp này, chúng ta có thể tìm được các tham số sao cho mô hình ước tính chính xác được chi phí người thợ sửa ống cống đưa ra. Đôi khi việc này là không khả thi, ví dụ một biến thể nào đó gây ra bởi các yếu tố ngoài hai đặc trưng kể trên. Trong những trường hợp này, ta sẽ cố học các mô hình sao cho khoảng cách giữa các giá trị dự đoán và các giá trị thực sự được cực tiểu hoá. Trong hầu hết các chương, chúng ta sẽ tập trung vào một trong hai hàm mất mát phổ biến nhất: hàm *mất mát L1*<sup>26</sup>, ở đó

$$l(y, y') = \sum_i |y_i - y'_i| \quad (3.3.1)$$

<sup>25</sup> [https://en.wikipedia.org/wiki/Netflix\\_Prize](https://en.wikipedia.org/wiki/Netflix_Prize)

<sup>26</sup> <http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L1Loss>

và hàm thứ hai là mất mát trung bình bình phương nhỏ nhất, hoặc **mất mát L<sub>2</sub>**<sup>27</sup>, ở đó

$$l(y, y') = \sum_i (y_i - y'_i)^2. \quad (3.3.2)$$

Như chúng ta sẽ thấy về sau, mất mát  $L_2$  tương ứng với giả sử rằng dữ liệu của chúng ta có nhiều Gauss, trong khi mất mát  $L_1$  tương ứng với giả sử nhiễu đến từ một phân phối Laplace.

## Phân loại

Trong khi các mô hình hồi quy hiệu quả trong việc trả lời các câu hỏi *có bao nhiêu?*, rất nhiều bài toán không phù hợp với nhóm câu hỏi này. Ví dụ, một ngân hàng muốn thêm chức năng quét ngân phiếu trong ứng dụng di động của họ. Tác vụ này bao gồm việc khách hàng chụp một tấm ngân phiếu với camera của điện thoại và mô hình học máy cần tự động hiểu nội dung chữ trong bức ảnh. Hiểu được cả chữ viết tay sẽ giúp ứng dụng hoạt động càng mạnh mẽ hơn. Kiểu hệ thống này được gọi là nhận dạng ký tự quang học (*optical character recognition* – OCR), và kiểu bài toán mà nó giải quyết được gọi là *phân loại* (*classification*). Nó được giải quyết với một tập các thuật toán khác với thuật toán dùng trong hồi quy (mặc dù có nhiều kỹ thuật chung).

Trong bài toán phân loại, ta muốn mô hình nhìn vào một vector đặc trưng, ví dụ như các giá trị điểm ảnh trong một bức ảnh, và sau đó dự đoán mẫu đó rơi vào hạng mục (được gọi là *lớp*) nào trong một tập các lựa chọn (rời rạc). Với chữ số viết tay, ta có thể có 10 lớp tương ứng với các chữ số từ 0 tới 9. Dạng đơn giản nhất của phân loại là khi chỉ có hai lớp, khi đó ta gọi bài toán này là phân loại nhị phân. Ví dụ, tập dữ liệu  $X$  có thể chứa các bức ảnh động vật và các *nhãn*  $Y$  có thể là các lớp {chó, mèo}. Trong khi với bài toán hồi quy, ta cần tìm một *bộ hồi quy* để đưa ra một giá trị thực  $\hat{y}$ , thì với bài toán phân loại, ta tìm một *bộ phân loại* để dự đoán lớp  $\hat{y}$ .

Khi cuốn sách đi sâu hơn vào các vấn đề kỹ thuật, chúng ta sẽ bàn về các lý do tại sao lại khó hơn để tối ưu hoá một mô hình mà đầu ra là các giá trị hạng mục rời rạc, ví dụ, *mèo* hoặc *chó*. Trong những trường hợp này, thường sẽ dễ hơn khi thay vào đó, ta biểu diễn mô hình dưới ngôn ngữ xác suất. Cho trước một mẫu  $x$ , mô hình cần gán một giá trị xác suất  $\hat{y}_k$  cho mỗi nhãn  $k$ . Vì là các giá trị xác suất, chúng phải là các số dương có tổng bằng 1. Bởi vậy, ta chỉ cần  $K - 1$  số để gán xác suất cho  $K$  hạng mục. Việc này dễ nhận thấy đối với phân loại nhị phân. Nếu một đồng xu không đều có xác suất ra mặt ngửa là 0.6 (60%), thì xác suất ra mặt sấp là 0.4 (40%). Trở lại với ví dụ phân loại động vật, một bộ phân loại có thể nhìn một bức ảnh và đưa ra xác suất để bức ảnh đó là mèo  $P(y = \text{mèo} | x) = 0.9$ . Chúng ta có thể diễn giải giá trị này tương ứng với việc bộ phân loại 90% tin rằng bức ảnh đó chứa một con mèo. Giá trị xác suất của lớp được dự đoán truyền đạt một ý niệm về sự không chắc chắn. Tuy nhiên, đó không phải là ý niệm duy nhất về sự không chắc chắn, chúng ta sẽ thảo luận thêm về những loại khác trong các chương nâng cao.

Khi có nhiều hơn hai lớp, ta gọi bài toán này là *phân loại đa lớp*. Bài toán phân loại chữ viết tay [0, 1, 2, 3 ... 9, a, b, c, ...] là một trong số các ví dụ điển hình. Trong khi các hàm mất mát thường được sử dụng trong các bài toán hồi quy là hàm mất mát L1 hoặc L2, hàm mất mát phổ biến cho bài toán phân loại được gọi là *entropy chéo* (*cross-entropy*), hàm tương ứng trong MXNet Gluon có thể tìm được [tại đây](#)<sup>28</sup>

Lưu ý rằng lớp có khả năng xảy ra nhất theo dự đoán của mô hình không nhất thiết là lớp mà ta quyết định sử dụng. Giả sử bạn tìm được một cây nấm rất đẹp trong sân nhà như hình Fig. 3.3.2.

<sup>27</sup> <http://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.L2Loss>

<sup>28</sup> <https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.SoftmaxCrossEntropyLoss>



Fig. 3.3.2: Nấm độc—đừng ăn!

Bây giờ giả sử ta đã xây dựng một bộ phân loại và huấn luyện nó để dự đoán liệu một cây nấm có độc hay không dựa trên ảnh chụp. Giả sử bộ phân loại phát hiện chất độc đưa ra  $P(y = \text{nấm độc}|\text{bức ảnh}) = 0.2$ . Nói cách khác, bộ phân loại này chắc chắn rằng 80% cây này *không phải* nấm độc. Dù vậy, đừng dại mà ăn nhé. Vì việc có bữa tối ngon lành không đáng gì so với rủi ro 20% sẽ chết vì nấm độc. Nói cách khác, hậu quả của *rủi ro không chắc chắn* nghiêm trọng hơn nhiều so với lợi ích thu được. Ta có thể nhìn việc này theo khía cạnh lý thuyết. Về cơ bản, ta cần tính toán rủi ro kỳ vọng mà mình sẽ gánh chịu, ví dụ, ta nhân xác suất xảy ra kết quả đó với lợi ích (hoặc hậu quả) đi liền tương ứng:

$$L(\text{hành động}|x) = E_{y \sim p(y|x)}[\text{mất_mát}(\text{hành động}, y)]. \quad (3.3.3)$$

Do đó, mất mát  $L$  do ăn phải nấm là  $L(a = \text{ăn}|x) = 0.2 * \infty + 0.8 * 0 = \infty$ , mặc dù phí tổn do bỏ nấm đi là  $L(a = \text{bỏ đi}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$ .

Sự thận trọng của chúng ta là chính đáng: như bất kỳ nhà nghiên cứu nấm nào cũng sẽ nói, cây nấm ở trên thực sự *là* nấm độc. Việc phân loại có thể còn phức tạp hơn nhiều so với phân loại nhị phân, đa lớp, hoặc thậm chí đa nhãn. Ví dụ, có vài biến thể của phân loại để xử lý vấn đề phân cấp bậc (*hierarchy*). Việc phân cấp giả định rằng tồn tại các mối quan hệ giữa các lớp với nhau. Vậy nên không phải tất cả các lối đều như nhau—nếu bắt buộc có lối, ta sẽ mong rằng các mẫu bị phân loại nhầm thành một lớp họ hàng thay vì một lớp khác xa nào đó. Thông thường, việc này được gọi là *phân loại cấp bậc* (*hierarchical classification*). Một trong những ví dụ đầu tiên về việc xây dựng hệ thống phân cấp là từ [Linnaeus<sup>29</sup>](#), người đã sắp xếp các loại động vật theo một hệ thống phân cấp.

Trong trường hợp phân loại động vật, cũng không tệ lắm nếu phân loại nhầm hai giống chó xù poodle và schnauzer với nhau, nhưng sẽ rất nghiêm trọng nếu ta nhầm lẫn chó poodle với một con khủng long. Hệ phân cấp nào là phù hợp phụ thuộc vào việc ta dự định dùng mô hình như thế nào. Ví dụ, rắn đuôi chuông và rắn sọc không độc có thể nằm gần nhau trong cây phả hệ, nhưng phân loại nhầm hai loài này có thể dẫn tới hậu quả chết người.

<sup>29</sup> [https://en.wikipedia.org/wiki/Carl\\_Linnaeus](https://en.wikipedia.org/wiki/Carl_Linnaeus)

## Gán thẻ

Một vài bài toán phân loại không phù hợp với các mô hình phân loại nhị phân hoặc đa lớp. Ví dụ, chúng ta có thể huấn luyện một bộ phân loại nhị phân thông thường để phân loại mèo và chó. Với khả năng hiện tại của thị giác máy tính, việc này có thể được thực hiện dễ dàng bằng các công cụ sẵn có. Tuy nhiên, bất kể mô hình của bạn chính xác đến đâu, nó có thể gặp rắc rối khi thấy bức ảnh Những Nhạc Sĩ thành Bremen.



Fig. 3.3.3: Mèo, gà trống, chó và lừa

Bạn có thể thấy trong ảnh có một con mèo, một con gà trống, một con chó, một con lừa và một con chim, cùng với một vài cái cây ở hậu cảnh. Tuỳ vào mục đích cuối cùng của mô hình, sẽ không hợp lý nếu coi đây là một bài toán phân loại nhị phân. Thay vào đó, có thể chúng ta muốn cung cấp cho mô hình tuỳ chọn để mô tả bức ảnh gồm một con mèo và một con chó và một con lừa và một con gà trống và một con chim.

Bài toán học để dự đoán các lớp *không xung khắc* được gọi là phân loại đa nhãn. Các bài toán tự động gán thẻ là các ví dụ điển hình của phân loại đa nhãn. Nghĩ về các thẻ mà một người có thể gán cho một blog công nghệ, ví dụ “học máy”, “công nghệ”, “ngôn ngữ lập trình”, “linux”, “điện toán đám mây” hay “AWS”. Một bài báo thông thường có thể có từ 5-10 thẻ bởi các khái niệm này có liên quan với nhau. Các bài về “điện toán đám mây” khả năng cao đề cập “AWS” và các bài về “học máy” cũng có thể dính dáng tới “ngôn ngữ lập trình”.

Ta cũng phải xử lý các vấn đề này trong nghiên cứu y sinh, ở đó việc gán thẻ cho các bài báo một cách chính xác là quan trọng bởi nó cho phép các nhà nghiên cứu tổng hợp đầy đủ các tài liệu liên quan. Tại Thư viện Y khoa Quốc gia, một số chuyên gia gán nhãn duyệt qua tất cả các bài báo được lưu trên PubMed để gán chúng với các thuật ngữ y khoa (*MeSH*) liên quan – một bộ sưu tập với

khoảng 28 nghìn thẻ. Đây là một quá trình tốn thời gian và những người gán nhãn thường bị trễ một năm kể từ khi lưu trữ cho tới lúc gán thẻ. Học máy có thể được sử dụng ở đây để cung cấp các thẻ tạm thời cho tới khi được kiểm chứng lại một cách thủ công. Thực vậy, BioASQ đã tổ chức một cuộc thi<sup>30</sup> dành riêng cho việc này.

## Tìm kiếm và xếp hạng

Đôi khi ta không chỉ muốn gán một lớp hoặc một giá trị vào một mẫu. Trong lĩnh vực thu thập thông tin (*information retrieval*), ta muốn gán thứ hạng cho một tập các mẫu. Lấy ví dụ trong tìm kiếm trang web, mục tiêu không chỉ dừng lại ở việc xác định liệu một trang web có liên quan tới từ khoá tìm kiếm, mà xa hơn, trang web nào trong số vô vàn kết quả trả về *liên quan nhất* tới người dùng. Chúng ta rất quan tâm đến thứ tự của các kết quả tìm kiếm và thuật toán cần đưa ra các tập con có thứ tự từ những thành phần trong một tập lớn hơn. Nói cách khác, nếu được hỏi đưa ra năm chữ cái từ bảng chữ cái, hai kết quả A B C D E và C A B E D là khác nhau. Ngay cả khi các phần tử trong hai tập kết quả là như nhau, thứ tự các phần tử trong mỗi tập mới là điều quan trọng.

Một giải pháp khả dĩ cho bài toán này là trước tiên gán cho mỗi phần tử trong tập hợp một số điểm về sự phù hợp và sau đó trả về những phần tử có điểm cao nhất. *PageRank*<sup>31</sup>, vũ khí bí mật đằng sau cỗ máy tìm kiếm của Google, là một trong những ví dụ đầu tiên của hệ thống tính điểm kiểu này. Tuy nhiên, điều bất thường là nó không phụ thuộc vào từ khoá tìm kiếm. Chúng phụ thuộc vào một bộ lọc đơn giản để xác định tập hợp các trang phù hợp rồi sau đó mới dùng PageRank để sắp xếp các kết quả có chứa cụm tìm kiếm. Có cả những hội thảo khoa học chuyên nghiên cứu về lĩnh vực này.

## Hệ thống gợi ý

Hệ thống gợi ý là một bài toán khác liên quan đến tìm kiếm và xếp hạng. Tuy có chung mục đích hiển thị một tập các kết quả liên quan tới người dùng, hệ thống gợi ý nhấn mạnh việc *cá nhân hóa* cho từng người dùng cụ thể. Ví dụ khi gợi ý phim ảnh, kết quả gợi ý cho một fan của phim khoa học viễn tưởng và cho một người sành sỏi hài Peter Sellers có thể khác nhau một cách đáng kể. Các bài toán gợi ý khác có thể bao gồm hệ thống gợi ý sản phẩm bán lẻ, âm nhạc hoặc tin tức.

Trong một vài trường hợp, khách hàng cung cấp phản hồi trực tiếp (*explicit feedback*) thể hiện mức độ yêu thích một sản phẩm cụ thể (ví dụ các đánh giá sản phẩm và phản hồi trên Amazon, IMDB, Goodreads, v.v.). Trong những trường hợp khác, họ cung cấp phản hồi gián tiếp (*implicit feedback*), ví dụ như khi bỏ qua bài hát trong danh sách chơi nhạc. Những bài hát đó có thể đã không làm hài lòng người nghe, hoặc chỉ đơn thuần là không phù hợp với bối cảnh. Diễn giải một cách đơn giản, những hệ thống này được huấn luyện để ước lượng một điểm  $y_{ij}$  nào đó, ví dụ như ước lượng điểm đánh giá hoặc ước lượng xác suất mua hàng, của một người dùng  $u_i$  tới một sản phẩm  $p_j$ .

Với một mô hình như vậy, cho một người dùng bất kỳ, ta có thể thu thập một tập các sản phẩm với điểm  $y_{ij}$  lớn nhất để gợi ý cho khách hàng. Các hệ thống đang vận hành trong thương mại còn cao cấp hơn nữa. Chúng sử dụng hành vi của người dùng và các thuộc tính sản phẩm để tính điểm. Fig. 3.3.4 là một ví dụ về các cuốn sách học sâu được gợi ý bởi Amazon dựa trên các thuật toán cá nhân hóa được điều chỉnh phù hợp với sở thích của tác giả cuốn sách này.

<sup>30</sup> <http://bioasq.org/>

<sup>31</sup> <https://en.wikipedia.org/wiki/PageRank>

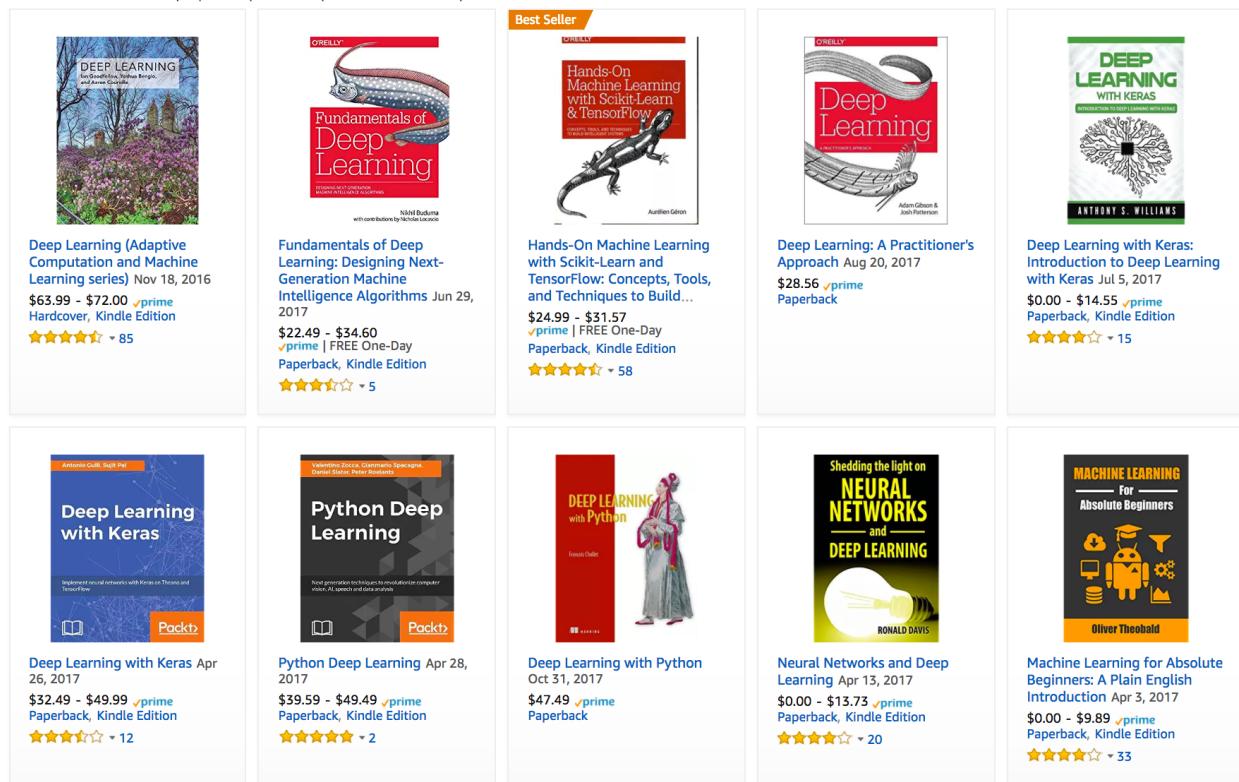


Fig. 3.3.4: Các sách học sâu được gợi ý bởi Amazon.

Mặc dù có giá trị kinh tế lớn, các hệ thống gợi ý được xây dựng đơn thuần theo các mô hình dự đoán về bản chất có những hạn chế nghiêm trọng. Ban đầu, ta chỉ quan sát các *phản hồi* được *kiểm duyệt*. Người dùng thường có xu hướng đánh giá các bộ phim họ thực sự thích hoặc ghét: bạn có thể để ý rằng các bộ phim nhận được rất nhiều đánh giá 5 và 1 sao nhưng rất ít các bộ phim với 3 sao. Hơn nữa, thói quen mua hàng hiện tại thường là kết quả của thuật toán gợi ý đang được dùng, nhưng các thuật toán gợi ý không luôn để ý đến chi tiết này. Vì vậy, các vòng phản hồi (*feedback loop*) luẩn quẩn có thể xảy ra khi mà hệ thống gợi ý đẩy lên một sản phẩm và cho rằng sản phẩm này tốt hơn (do được mua nhiều hơn), từ đó sản phẩm này lại được hệ thống gợi ý thường xuyên hơn nữa. Rất nhiều trong số các vấn đề về cách xử lý với kiểm duyệt, động cơ của việc đánh giá và vòng phản hồi là các câu hỏi quan trọng cho nghiên cứu.

## Học chuỗi

Cho tới giờ, chúng ta đã gặp các bài toán mà ở đó mô hình nhận đầu vào với kích thước cố định và đưa ra kết quả cũng với kích thước cố định. Trước đây chúng ta xem xét dự đoán giá nhà từ một tập các đặc trưng cố định: diện tích, số phòng ngủ, số phòng tắm và thời gian đi bộ tới trung tâm thành phố. Ta cũng đã thảo luận cách ánh xạ từ một bức ảnh (với kích thước cố định) tới các dự đoán xác suất nó thuộc vào một tập cố định các lớp, hoặc lấy một mã người dùng và mã sản phẩm để dự đoán số sao đánh giá. Trong những trường hợp này, một khi chúng ta đưa cho mô hình một đầu vào có độ dài cố định để dự đoán một đầu ra, mô hình ngay lập tức “quên” dữ liệu nó vừa thấy.

Việc này không ảnh hưởng nhiều nếu mọi đầu vào đều có cùng kích thước và nếu các đầu vào liên tiếp thật sự không liên quan đến nhau. Nhưng ta sẽ xử lý các đoạn video như thế nào khi mỗi đoạn có thể có số lượng khung hình khác nhau? Và dự đoán của chúng ta về việc gì đang xảy ra ở mỗi khung hình có thể sẽ chính xác hơn nếu ta xem xét cả các khung hình kề nó. Hiện tượng tương tự

xảy ra trong ngôn ngữ. Một bài toán học sâu phổ biến là dịch máy (*machine translation*): tác vụ lấy đầu vào là các câu trong một ngôn ngữ nguồn và trả về bản dịch của chúng ở một ngôn ngữ khác.

Các bài toán này cũng xảy ra trong y khoa. Với một mô hình theo dõi bệnh nhân trong phòng hồi sức tích cực, ta có thể muốn nó đưa ra cảnh báo nếu nguy cơ tử vong của họ trong 24 giờ tới vượt một ngưỡng nào đó. Dĩ nhiên, ta chắc chắn không muốn mô hình này vứt bỏ mọi số liệu trong quá khứ và chỉ đưa ra dự đoán dựa trên các thông số mới nhất.

Những bài toán này nằm trong những ứng dụng thú vị nhất của học máy và chúng là các ví dụ của *học chuỗi*. Chúng đòi hỏi một mô hình có khả năng nhận chuỗi các đầu vào hoặc dự đoán chuỗi các đầu ra. Thậm chí có mô hình phải thỏa mãn cả hai tiêu chí đó, và những bài toán có cấu trúc như vậy còn được gọi là seq2seq (*sequence to sequence: chuỗi sang chuỗi*). Dịch ngôn ngữ là một bài toán seq2seq. Chuyển một bài nói về dạng văn bản cũng là một bài toán seq2seq. Mặc dù không thể xét hết mọi dạng của biến đổi chuỗi, có một vài trường hợp đặc biệt đáng được lưu tâm:

**Gán thẻ và Phân tích cú pháp.** Đây là bài toán chú thích cho một chuỗi văn bản. Nói cách khác, số lượng đầu vào và đầu ra là như nhau. Ví dụ, ta có thể muốn biết vị trí của động từ và chủ ngữ. Hoặc ta cũng có thể muốn biết từ nào là danh từ riêng. Mục tiêu tổng quát là phân tích và chú thích văn bản dựa trên các giả định về cấu trúc và ngữ pháp. Việc này nghe có vẻ phức tạp hơn thực tế. Dưới đây là một ví dụ rất đơn giản về việc chú thích một câu bằng các thẻ đánh dấu danh từ riêng.

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

**Tự động nhận dạng giọng nói.** Với nhận dạng giọng nói, chuỗi đầu vào  $\mathbf{x}$  là một bản thu âm giọng nói của một người (Fig. 3.3.5) và đầu ra  $\mathbf{y}$  là một văn bản ghi lại những gì người đó nói. Thủ thách ở đây là việc số lượng các khung âm thanh (âm thanh thường được lấy mẫu ở 8kHz or 16kHz) nhiều hơn hẳn so với số lượng từ, nghĩa là không tồn tại một phép ánh xạ 1:1 nào giữa các khung âm thanh và các từ, bởi một từ có thể tương ứng với hàng ngàn mẫu âm thanh. Có các bài toán seq2seq mà đầu ra ngắn hơn rất nhiều so với đầu vào.

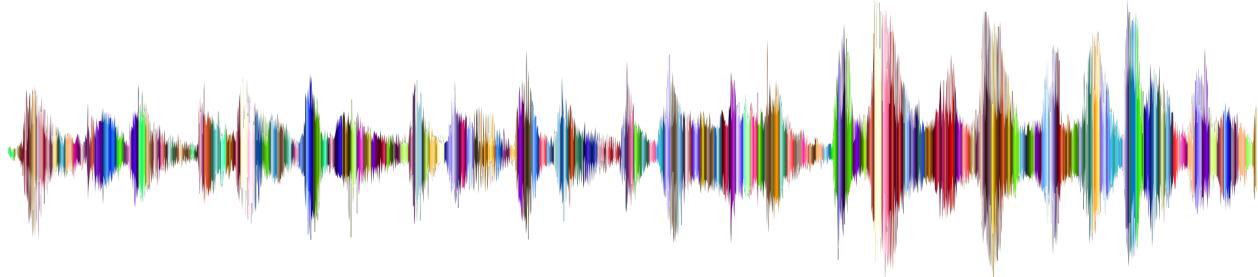


Fig. 3.3.5: -D-e-e-p- L-ea-r-ni-ng-

**Chuyển văn bản thành giọng nói** (*Text-to-Speech* hay TTS) là bài toán ngược của bài toán nhận dạng giọng nói. Nói cách khác, đầu vào  $\mathbf{x}$  là văn bản và đầu ra  $\mathbf{y}$  là tệp tin âm thanh. Trong trường hợp này, đầu ra dài hơn rất nhiều so với đầu vào. Việc nhận dạng các tệp tin âm thanh chất lượng kém không khó với *con người* nhưng lại không hề đơn giản với máy tính.

**Dịch máy.** Khác với nhận dạng giọng nói, khi các đầu vào và đầu ra tương ứng có cùng thứ tự (sau khi căn chỉnh), trong dịch máy việc đảo thứ tự lại có thể rất quan trọng. Nói cách khác, ta vẫn chuyển đổi từ chuỗi này sang chuỗi khác, nhưng ta không thể giả định số lượng đầu vào và đầu ra cũng như thứ tự của các điểm dữ liệu tương ứng là giống nhau. Xét ví dụ minh họa dưới đây về trật tự từ kỳ lạ khi dịch tiếng Anh sang tiếng Việt.

Tiếng Anh: Did you already check out this excellent tutorial?

Tiếng Việt: Bạn đã xem qua hướng dẫn tuyệt vời này chưa?

Căn chỉnh sai: Đã chưa bạn xem qua này tuyệt vời hướng dẫn?

Rất nhiều bài toán liên quan xuất hiện trong các tác vụ học khác. Chẳng hạn, xác định thứ tự người dùng đọc một trang mạng là một bài toán phân tích bối cảnh hai chiều. Các bài toán hội thoại thì chứa đủ các loại vấn đề phức tạp khác, như việc xác định câu nói tiếp theo đòi hỏi kiến thức thực tế cũng như trạng thái trước đó rất lâu của cuộc hội thoại. Đây là một lĩnh vực nghiên cứu đang phát triển.

### 3.3.2 Học không giám sát

Tất cả các ví dụ từ trước đến nay đều liên quan tới lĩnh vực *Học có giám sát*, ví dụ, những trường hợp mà ta nạp vào mô hình lượng dữ liệu khổng lồ gồm cả các đặc trưng và giá trị mục tiêu tương ứng. Bạn có thể tưởng tượng người học có giám sát giống như anh nhân viên đang làm một công việc có tính chuyên môn cao cùng với một ông sếp cực kỳ hâm tài. Ông sếp đứng ngay bên cạnh bạn và chỉ bảo chi tiết những điều phải làm trong từng tình huống một cho tới khi bạn học được cách liên kết các tình huống đó với hành động tương ứng. Làm việc với ông sếp kiểu này có vẻ khá là chán. Nhưng ở một khía cạnh khác, thì làm ông sếp này hài lòng rất là dễ dàng. Bạn chỉ việc nhận ra những khuôn mẫu thật nhanh và bắt chước lại những hành động theo khuôn mẫu đó.

Trong một hoàn cảnh hoàn toàn trái ngược, nếu gặp phải người sếp không biết họ muốn bạn làm cái gì thì sẽ rất là khó chịu. Tuy nhiên, nếu định trở thành nhà khoa học dữ liệu, tốt nhất hãy làm quen với điều này. Ông ta có thể chỉ đưa cho bạn một đống dữ liệu to sụ và bảo *sử dụng khoa học dữ liệu với cái này đi!* Nghe có vẻ khá mơ hồ bởi vì đúng là nó mơ hồ thật. Chúng ta gọi những loại vấn đề như thế này là *học không giám sát (unsupervised learning)*; với chúng, loại câu hỏi và lượng câu hỏi ta có thể đặt ra chỉ bị giới hạn bởi trí tưởng tượng của chính mình. Ta sẽ đề cập tới một số kỹ thuật học không giám sát ở các chương sau. Nay, để gợi cho bạn đọc chút hứng khởi, chúng tôi sẽ diễn giải một vài câu hỏi bạn có thể sẽ hỏi:

- Liệu có thể dùng một lượng nhỏ nguyên mẫu để tóm lược dữ liệu một cách chính xác? Giả sử với một bộ ảnh, liệu có thể nhóm chúng thành các ảnh phong cảnh, chó, trẻ con, mèo, đỉnh núi, v.v.? Tương tự, với bộ dữ liệu duyệt web của người dùng, liệu có thể chia họ thành các nhóm người dùng có hành vi giống nhau? Vấn đề như này thường được gọi là *phân cụm (clustering)*.
- Liệu ta có tìm được một lượng nhỏ các tham số mà vẫn tóm lược được chính xác những thuộc tính cốt lõi của dữ liệu? Như là quỹ đạo bay của quả bóng được miêu tả tương đối tốt bởi số liệu vận tốc, đường kính và khối lượng của quả bóng. Như một người thợ may chỉ cần một lượng nhỏ các số đo để miêu tả hình dáng cơ thể người tương đối chuẩn xác để may ra quần áo vừa vặn. Những ví dụ trên được gọi là bài toán *ước lượng không gian con (subspace estimation)*. Nếu mối quan hệ phụ thuộc là tuyến tính, bài toán này được gọi là phép *phân tích thành phần chính (principal component analysis – PCA)*.
- Liệu có tồn tại cách biểu diễn các đối tượng (có cấu trúc bất kỳ) trong không gian Euclid (ví dụ: không gian vector  $\mathbb{R}^n$ ) mà những thuộc tính đặc trưng có thể được ghép khớp với nhau? Phương pháp này được gọi là *học biểu diễn (representation learning)* và được dùng để miêu tả các thực thể và mối quan hệ giữa chúng, giống như Rome – Ý + Pháp = Paris.
- Liệu có tồn tại cách miêu tả những nguyên nhân gốc rễ của lượng dữ liệu mà ta đang quan sát được? Ví dụ, nếu chúng ta có dữ liệu nhân khẩu học về giá nhà, mức độ ô nhiễm, tệ nạn, vị trí, trình độ học vấn, mức lương, v.v.. thì liệu ta có thể khám phá ra cách chúng liên hệ với

nhau chỉ đơn thuần dựa vào dữ liệu thực nghiệm? Những lĩnh vực liên quan tới *nhân quả* và *mô hình đồ thị xác suất* (*probabilistic graphical models*) sẽ giải quyết bài toán này.

Một bước phát triển quan trọng và thú vị gần đây của học không giám sát là sự ra đời của *mạng đối sinh* (*generative adversarial network* – GAN). GAN cho ta một quy trình sinh dữ liệu, kể cả những dữ liệu cấu trúc phức tạp như hình ảnh và âm thanh. Cùng các cơ chế toán thống kê ẩn bên dưới sẽ kiểm tra xem liệu những dữ liệu thật và giả này có giống nhau không. Chúng tôi sẽ viết vài mục về chủ đề này sau.

### 3.3.3 Tương tác với Môi trường

Cho tới giờ, chúng ta chưa thảo luận về việc dữ liệu tới từ đâu hoặc chuyện gì thực sự sẽ *xảy ra* khi một mô hình học máy trả về kết quả dự đoán. Điều này là do học có giám sát và học không giám sát chưa giải quyết các vấn đề một cách thấu đáo. Trong cả hai cách học, chúng ta yêu cầu mô hình học từ một lượng dữ liệu lớn đã được cung cấp từ đầu mà không cho nó tương tác trở lại với môi trường trong suốt quá trình học. Bởi vì toàn bộ việc học diễn ra khi thuật toán đã được ngắt kết nối khỏi môi trường, đôi khi ta gọi đó là *học ngoại tuyến* (*offline learning*). Quá trình này cho học có giám sát được mô tả trong Fig. 3.3.6.

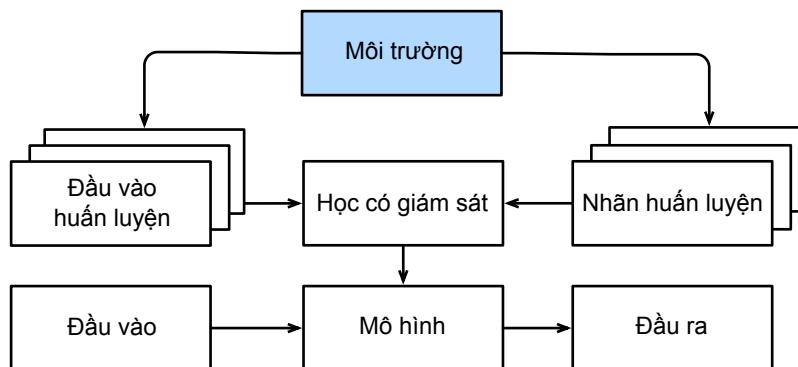


Fig. 3.3.6: Thu thập dữ liệu từ môi trường cho học có giám sát

Sự đơn giản của học ngoại tuyến có nét đẹp của nó. Ưu điểm là ta chỉ cần quan tâm đến vấn đề nhận dạng mẫu mà không bị phân tâm bởi những vấn đề khác. Nhưng nhược điểm là sự hạn chế trong việc thiết lập các bài toán. Nếu bạn đã đọc loạt truyện ngắn Robots của Asimov hoặc là người có tham vọng, bạn có thể đang tưởng tượng ra trí tuệ nhân tạo không những biết đưa ra dự đoán mà còn có thể tương tác với thế giới. Chúng ta muốn nghĩ tới những *tác nhân* (*agent*) thông minh chứ không chỉ những *mô hình* dự đoán. Tức là ta phải nhắm tới việc chọn *hành động* chứ không chỉ đưa ra những *dự đoán*. Hơn thế nữa, không giống dự đoán, hành động còn tác động đến môi trường. Nếu muốn huấn luyện một tác nhân thông minh, chúng ta phải tính đến cách những hành động của nó có thể tác động đến những gì nó nhận lại trong tương lai.

Xem xét việc tương tác với môi trường mở ra một loạt những câu hỏi về mô hình hóa mới. Liệu môi trường có:

- Nhớ những gì ta đã làm trước đó?
- Muốn giúp đỡ chúng ta, chẳng hạn: một người dùng đọc văn bản vào một bộ nhận dạng giọng nói?
- Muốn đánh bại chúng ta, chẳng hạn: một thiết lập đối kháng giống như bộ lọc thư rác (chống lại những kẻ viết thư rác) hay là chơi game (với đối thủ)?

- Không quan tâm (có rất nhiều trường hợp thế này)?
- Có xu hướng thay đổi (dữ liệu trong tương lai có giống với trong quá khứ không, hay là khuôn mẫu có thay đổi theo thời gian một cách tự nhiên hoặc do phản ứng với những công cụ tự động)?

Câu hỏi cuối cùng nêu lên vấn đề về *dịch chuyển phân phối* (*distribution shift*), khi dữ liệu huấn luyện và dữ liệu kiểm tra khác nhau. Vấn đề này giống như khi chúng ta phải làm bài kiểm tra được cho bởi giảng viên trong khi bài tập về nhà lại do trợ giảng chuẩn bị. Chúng ta sẽ thảo luận sơ qua về học tăng cường (*reinforcement learning*) và học đối kháng (*adversarial learning*), đây là hai thiết lập đặc biệt có xét tới tương tác với môi trường.

### 3.3.4 Học tăng cường

Nếu bạn muốn dùng học máy để phát triển một tác nhân tương tác với môi trường và đưa ra hành động, khả năng cao là bạn sẽ cần tập trung vào *học tăng cường* (*reinforcement learning* – RL). Học tăng cường có các ứng dụng trong ngành công nghệ robot, hệ thống đối thoại và cả việc phát triển AI cho trò chơi điện tử. *Học sâu tăng cường* (*Deep reinforcement learning* – DRL) áp dụng kỹ thuật học sâu để giải quyết những vấn đề của học tăng cường và đã trở nên phổ biến trong thời gian gần đây. Hai ví dụ tiêu biểu nhất là thành tựu đột phá của mạng-Q sâu đánh bại con người trong các trò chơi điện tử Atari chỉ sử dụng đầu vào hình ảnh<sup>32</sup>, và chương trình AlphaGo chiếm ngôi vô địch thế giới trong môn Cờ Vây<sup>33</sup>.

Học tăng cường mô tả bài toán theo cách rất tổng quát, trong đó tác nhân tương tác với môi trường qua một chuỗi các *bước thời gian* (*timesteps*). Tại mỗi bước thời gian  $t$ , tác nhân sẽ nhận được một quan sát  $o_t$  từ môi trường và phải chọn một hành động  $a_t$  để tương tác với môi trường thông qua một cơ chế nào đó (đôi khi còn được gọi là bộ dẫn động). Sau cùng, tác nhân sẽ nhận được một điểm thưởng  $r_t$  từ môi trường. Sau đó, tác nhân lại nhận một quan sát khác và chọn ra một hành động, cứ tiếp tục như thế. Hành vi của tác nhân học tăng cường được kiểm soát bởi một *chính sách* (*policy*). Nói ngắn gọn, một *chính sách* là một hàm ánh xạ từ những quan sát (từ môi trường) tới các hành động. Mục tiêu của học tăng cường là tạo ra một chính sách tốt.

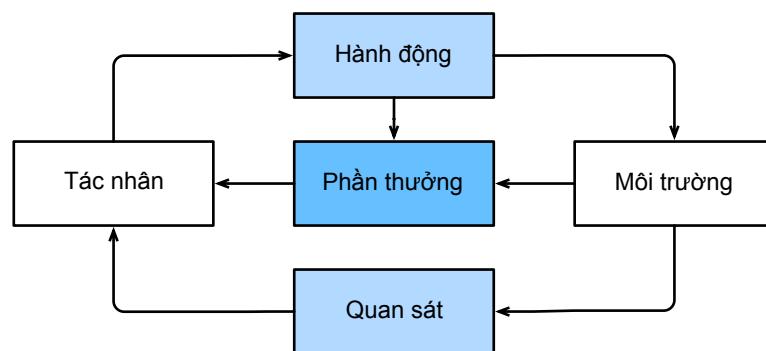


Fig. 3.3.7: Sự tương tác giữa học tăng cường và môi trường.

Tính tổng quát của mô hình học tăng cường không phải là một thứ được nói quá lên. Ví dụ, ta có thể chuyển bất cứ bài toán học có giám sát nào thành một bài toán học tăng cường. Chẳng hạn với một bài toán phân loại, ta có thể tạo một tác nhân học tăng cường với một *hành động* tương ứng với mỗi lớp. Sau đó ta có thể tạo một môi trường mà trả về điểm thưởng đúng bằng với giá trị của hàm mất mát từ bài toán học có giám sát ban đầu.

<sup>32</sup> <https://www.wired.com/2015/02/google-ai-plays-atari-like-pros/>

<sup>33</sup> <https://www.wired.com/2017/05/googles-alphago-trounces-humans-also-gives-boost/>

Vì thế, học tăng cường có thể giải quyết nhiều vấn đề mà học có giám sát không thể. Lấy ví dụ ở trong học có giám sát, chúng ta luôn đòi hỏi dữ liệu huấn luyện phải đi kèm với đúng nhãn. Tuy nhiên với học tăng cường, ta không giả định rằng môi trường sẽ chỉ ra hành động nào là tối ưu tại mỗi quan sát (điểm dữ liệu). Nhìn chung, mô hình sẽ chỉ nhận được một điểm thưởng nào đó. Hơn thế nữa, môi trường có thể sẽ không chỉ ra những hành động nào đã dẫn tới điểm thưởng đó.

Lấy cờ vua làm ví dụ. Phần thưởng thật sự sẽ đến vào cuối trò chơi. Khi thắng, ta sẽ được 1 điểm, hoặc khi thua, ta sẽ nhận về -1 điểm. Vì vậy, việc học tăng cường phải giải quyết *bài toán phân bổ công trạng* (*credit assignment problem*): xác định hành động nào sẽ được thưởng hay bị phạt dựa theo kết quả. Tương tự như khi một nhân viên được thăng chức vào ngày 11/10. Việc thăng chức khả năng cao phản ánh những việc làm tốt của nhân viên này trong suốt một năm qua. Để được thăng chức sau này đòi hỏi nhân viên đó phải nhận ra đâu là những hành động dẫn đến việc thăng chức này.

Học tăng cường còn phải đương đầu với vấn đề về những quan sát không hoàn chỉnh. Có nghĩa là quan sát hiện tại có thể không cho bạn biết mọi thứ về tình trạng lúc này. Lấy ví dụ, khi robot hút bụi bị kẹt tại một trong nhiều phòng giống y như nhau trong căn nhà. Việc can thiệp vào vị trí chính xác (cũng là trạng thái) của robot có thể cần đến những quan sát từ trước khi nó đi vào phòng.

Cuối cùng, tại một thời điểm bất kỳ, các thuật toán học tăng cường có thể biết một chính sách tốt, tuy nhiên có thể có những chính sách khác tốt hơn mà tác nhân chưa bao giờ thử tới. Các thuật toán học tăng cường phải luôn lựa chọn giữa việc tiếp tục *khai thác* chính sách tốt nhất hiện thời hay *khám phá* thêm những giải pháp khác, tức bỏ qua những điểm thưởng ngắn hạn để thu về thêm kiến thức.

### MDPs, máy đánh bạc, và những người bạn

Các bài toán học tăng cường thường có một thiết lập rất tổng quát. Các hành động của tác nhân có ảnh hưởng đến những quan sát về sau. Những điểm thưởng nhận được tương ứng chỉ với các hành động được chọn. Môi trường có thể được quan sát đầy đủ hoặc chỉ một phần. Tính toán tất cả sự phức tạp này cùng lúc có thể cần sự tham gia của quá nhiều nhà nghiên cứu. Hơn nữa, không phải mọi vấn đề thực tế đều thể hiện tất cả sự phức tạp này. Vì vậy, các nhà nghiên cứu đã nghiên cứu một số *trường hợp đặc biệt* về những bài toán học tăng cường.

Khi ở môi trường được quan sát đầy đủ, ta gọi bài toán học tăng cường này là *Quá trình Quyết định Markov* (*Markov Decision Process* – MDP). Khi trạng thái không phụ thuộc vào các hành động trước đó, ta gọi bài toán này là *bài toán máy đánh bạc theo ngữ cảnh* (*contextual bandit problem*). Khi không có trạng thái, chỉ có một tập hợp các hành động có sẵn với điểm thưởng chưa biết ban đầu, bài toán kinh điển này là *bài toán máy đánh bạc đa cần* (*multi-armed bandit problem*).

## 3.4 Nguồn gốc

Mặc dù có nhiều phương pháp học sâu được phát minh gần đây, nhưng mong muốn phân tích dữ liệu để dự đoán kết quả tương lai của con người đã tồn tại trong nhiều thế kỷ. Và trong thực tế, phần lớn khoa học tự nhiên đều có nguồn gốc từ điều này. Ví dụ, phân phối Bernoulli được đặt theo tên của Jacob Bernoulli (1655-1705)<sup>34</sup>, và bản phân phối Gausian được phát hiện bởi Carl Friedrich Gauss (1777-1855)<sup>35</sup>. Ông đã phát minh ra thuật toán trung bình bình phương nhỏ nhất,

<sup>34</sup> [https://en.wikipedia.org/wiki/Jacob\\_Bernoulli](https://en.wikipedia.org/wiki/Jacob_Bernoulli)

<sup>35</sup> [https://en.wikipedia.org/wiki/Carl\\_Friedrich\\_Gauss](https://en.wikipedia.org/wiki/Carl_Friedrich_Gauss)

thuật toán này vẫn được sử dụng cho rất nhiều vấn đề từ tính toán bảo hiểm cho đến chẩn đoán y khoa. Những công cụ này đã mở đường cho một cách tiếp cận dựa trên thử nghiệm trong các ngành khoa học tự nhiên – ví dụ, định luật Ohm mô tả rằng dòng điện và điện áp trong một điện trở được diễn tả hoàn hảo bằng một mô hình tuyến tính.

Ngay cả trong thời kỳ trung cổ, các nhà toán học đã có một trực giác nhạy bén trong các ước tính của mình. Chẳng hạn, cuốn sách hình học của Jacob Köbel (1460-1533)<sup>36</sup> minh họa việc lấy trung bình chiều dài 16 bàn chân của những người đàn ông trưởng thành để có được chiều dài bàn chân trung bình.



Fig. 3.4.1: Ước tính chiều dài của một bàn chân

Fig. 3.4.1 minh họa cách các công cụ ước tính này hoạt động. 16 người đàn ông trưởng thành được yêu cầu xếp hàng nối tiếp nhau khi rời nhà thờ. Tổng chiều dài của các bàn chân sau đó được chia cho 16 để có được ước tính trị số hiện tại tầm 1 feet. “Thuật toán” này đã được cải tiến ngay sau đó nhằm giải quyết trường hợp bàn chân bị biến dạng – 2 người đàn ông có bàn chân ngắn nhất và dài nhất tương ứng được loại ra, trung bình chỉ được tính trong phần còn lại. Đây là một trong những ví dụ đầu tiên về ước tính trung bình cắt ngọn.

Thống kê thật sự khởi sắc với việc thu thập và có sẵn dữ liệu. Một trong số những người phi thường đã đóng góp lớn vào lý thuyết và ứng dụng của nó trong di truyền học, đó là Ronald Fisher (1890-1962)<sup>37</sup>. Nhiều thuật toán và công thức của ông (như Phân tích biệt thức tuyến tính - Linear Discriminant Analysis hay Ma trận thông tin Fisher - Fisher Information Matrix) vẫn được sử dụng thường xuyên cho đến ngày nay (ngay cả bộ dữ liệu Iris mà ông công bố năm 1936 đôi khi vẫn được sử dụng để minh họa cho các thuật toán học máy). Fisher cũng là một người ủng hộ thuyết ưu sinh. Điều này nhắc chúng ta rằng việc áp dụng khoa học dữ liệu vào những ứng dụng mờ ám

<sup>36</sup> <https://www.maa.org/press/periodicals/convergence/mathematical-treasures-jacob-kobels-geometry>

<sup>37</sup> [https://en.wikipedia.org/wiki/Ronald\\_Fisher](https://en.wikipedia.org/wiki/Ronald_Fisher)

trên phương diện đạo đức cũng như các ứng dụng có ích trong công nghiệp và khoa học tự nhiên đều đã có lịch sử tồn tại và phát triển lâu đời.

Ảnh hưởng thứ hai đối với học máy đến từ Lý Thuyết Thông Tin (Claude Shannon, 1916-2001)<sup>38</sup> và Lý Thuyết Điện Toán của Alan Turing (1912-1954)<sup>39</sup>. Turing đã đặt ra câu hỏi “Liệu máy móc có thể suy nghĩ?” trong bài báo nổi tiếng của mình *Máy Tính và Trí Thông Minh*<sup>40</sup> (Mind, Tháng 10 1950). Tại đó ông ấy đã giới thiệu về phép thử Turing: một máy tính được xem là thông minh nếu như người đánh giá khó có thể phân biệt phản hồi mà anh ta nhận được thông qua tương tác văn bản xuất phát từ máy tính hay con người.

Một ảnh hưởng khác có thể được tìm thấy trong khoa học thần kinh và tâm lý học. Trên hết, loài người thể hiện rất rõ ràng hành vi thông minh của mình. Bởi vậy câu hỏi tự nhiên là liệu một người có thể giải thích và lầm ngược để tận dụng năng lực này. Một trong những thuật toán lâu đời nhất lấy cảm hứng từ câu hỏi trên được giới thiệu bởi Donald Hebb (1904-1985)<sup>41</sup>. Trong cuốn sách đột phá của mình về Tổ Chức Hành Vi (Hebb & Hebb, 1949), ông ấy cho rằng các nơ-ron học bằng cách dựa trên những phản hồi tích cực. Điều này sau được biết đến với cái tên là luật học Hebbian. Nó là nguyên mẫu cho thuật toán perceptron của Rosenblatt và là nền tảng của rất nhiều thuật toán hạ gradient ngẫu nhiên đã đặt nền móng cho học sâu ngày nay: tăng cường các hành vi mong muốn và giảm bớt các hành vi không mong muốn để đạt được những thông số tốt trong một mạng nơ-ron.

Niềm cảm hứng từ sinh học đã tạo nên cái tên *mạng nơ-ron*. Trong suốt hơn một thế kỷ (từ mô hình của Alexander Bain, 1873 và James Sherrington, 1890), các nhà nghiên cứu đã cố gắng lắp ráp các mạch tính toán tương tự như các mạng tương tác giữa các nơ-ron. Theo thời gian, yếu tố sinh học ngày càng giảm đi nhưng tên gọi của nó thì vẫn ở lại. Những nguyên tắc thứ yếu của mạng nơ-ron vẫn có thể tìm thấy ở hầu hết các mạng ngày nay:

- Sự đan xen giữa các đơn vị xử lý tuyến tính và phi tuyến tính, thường được đề cập tới như là *các tầng*.
- Việc sử dụng quy tắc chuỗi (còn được biết đến là *làn truyền ngược – backpropagation*) để điều chỉnh các tham số trong toàn bộ mạng cùng lúc.

Sau những tiến bộ nhanh chóng ban đầu, các nghiên cứu về mạng nơ-ron giảm dần trong khoảng từ 1995 tới 2005 bởi một vài lí do. Thứ nhất là huấn luyện mạng rất tốt kém tài nguyên tính toán. Mặc dù dung lượng RAM đã dồi dào vào cuối thế kỷ trước, sức mạnh tính toán vẫn còn hạn chế. Thứ hai là tập dữ liệu vẫn còn tương đối nhỏ. Lúc đó tập dữ liệu Fisher's Iris từ năm 1932 vẫn là công cụ phổ biến để kiểm tra tính hiệu quả của các thuật toán. MNIST với 60.000 ký tự viết tay đã được xem là rất lớn.

Với sự khan hiếm của dữ liệu và tài nguyên tính toán, các công cụ thống kê mạnh như Phương Pháp Hạt Nhân, Cây Quyết Định và Mô Hình Đồ Thị đã chứng tỏ sự vượt trội trong thực nghiệm. Khác với mạng nơ-ron, chúng không đòi hỏi nhiều tuần huấn luyện nhưng vẫn đưa ra những kết quả dự đoán với sự đảm bảo vững chắc về lý thuyết.

<sup>38</sup> [https://en.wikipedia.org/wiki/Claude\\_Shannon](https://en.wikipedia.org/wiki/Claude_Shannon)

<sup>39</sup> [https://en.wikipedia.org/wiki/Alan\\_Turing](https://en.wikipedia.org/wiki/Alan_Turing)

<sup>40</sup> [https://en.wikipedia.org/wiki/Computing\\_Machinery\\_and\\_Intelligence](https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence)

<sup>41</sup> [https://en.wikipedia.org/wiki/Donald\\_O.\\_Hebb](https://en.wikipedia.org/wiki/Donald_O._Hebb)

### 3.5 Con đường tới Học Sâu

Con đường tới học sâu đã có rất nhiều thay đổi cùng với sự sẵn có của lượng lớn dữ liệu nhờ vào Mạng Lưới Toàn Cầu (World Wide Web) – sự phát triển của các công ty với hàng triệu người dùng trực tuyến, sự phổ biến của các cảm biến giá rẻ với chất lượng cao, bộ lưu trữ dữ liệu giá rẻ (theo quy luật Kryder), và tính toán chi phí thấp (theo định luật Moore) – đặc biệt là các GPU, với thiết kế ban đầu được dành cho việc xử lý các trò chơi máy tính. Và rồi những thuật toán và mô hình tưởng chừng không khả thi về mặt tính toán đã không còn ngoài tầm với. Điều này được minh họa trong Table 3.5.1.

Table 3.5.1: Liên hệ giữa tập dữ liệu với bộ nhớ máy tính và năng lực tính toán

Thập kỷ	Tập dữ liệu	Bộ nhớ	Số lượng phép tính dấu phẩy động trên giây
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (Giá nhà ở Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (Nhận dạng ký tự quang học)	10 MB	10 MF (Intel 80486)
2000	10 M (các trang web)	100 MB	1 GF (Intel Core)
2010	10 G (quảng cáo)	1 GB	1 TF (Nvidia C2050)
2020	1 T (mạng xã hội)	100 GB	1 PF (Nvidia DGX-2)

Sự thật là RAM đã không theo kịp với tốc độ phát triển của dữ liệu. Đồng thời, sự tiến bộ trong năng lực tính toán đã vượt lên tốc độ phát triển của dữ liệu có sẵn. Vì vậy, mô hình thống kê cần phải trở nên hiệu quả hơn về bộ nhớ (thường đạt được bằng cách thêm các thành phần phi tuyến) đồng thời có thể tập trung thời gian cho việc tối ưu các tham số nhờ sự gia tăng trong khả năng tính toán. Kéo theo đó, tiêu điểm trong học máy và thống kê đã chuyển từ các mô hình tuyến tính (tổng quát) và các phương pháp hạt nhân (*kernel methods*) sang các mạng nơ-ron sâu. Đây cũng là một trong những lý do những kỹ thuật cổ điển trong học sâu như perceptron đa tầng (McCulloch & Pitts, 1943), mạng nơ-ron tích chập, (LeCun et al., 1998), bộ nhớ ngắn hạn dài (Long Short-Term Memory – LSTM) (Hochreiter & Schmidhuber, 1997), và học Q (Watkins & Dayan, 1992), đã được “tái khám phá” trong thập kỷ trước sau một khoảng thời gian dài ít được sử dụng.

Những tiến bộ gần đây trong các mô hình thống kê, các ứng dụng và các thuật toán đôi khi được liên hệ với Sự bùng nổ kỷ Cambry: thời điểm phát triển nhanh chóng trong sự tiến hóa của các loài. Thật vậy, các kỹ thuật tiên tiến nay không chỉ đơn thuần chỉ là hệ quả của việc các kỹ thuật cũ được áp dụng với các nguồn tài nguyên hiện tại. Danh sách dưới đây còn chưa thẩm vào đâu với số lượng những ý tưởng đã và đang giúp các nhà nghiên cứu đạt được những thành tựu khổng lồ trong thập kỷ vừa qua.

- Các phương pháp tiên tiến trong việc kiểm soát năng lực, như Dropout (Srivastava et al., 2014), đã giúp làm giảm sự nguy hiểm của quá khứ. Việc này đạt được bằng cách thêm nhiễu (Bishop, 1995) xuyên suốt khắp mạng, thay các trọng số bởi các biến ngẫu nhiên cho mục đích huấn luyện.
- Cơ chế tập trung giải quyết vấn đề thứ hai đã ám ảnh ngành 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 lượng tham số cần học. (Bahdanau et al., 2014) đã tìm ra một giải pháp tinh tế bằng cách sử dụng một cấu trúc con trỏ có thể học được. Ví dụ trong dịch máy, thay vì nhớ toàn bộ câu với cách biểu diễn có số chiều cố định, ta chỉ cần lưu một con trỏ tới trạng thái trung gian của quá trình dịch. Việc này giúp tăng đáng kể độ chính xác của các câu dài vì lúc này mô hình

không còn cần nhớ toàn bộ câu trước khi chuyển sang tạo câu tiếp theo.

- Thiết kế đa bước, ví dụ thông qua các Mạng Bộ Nhớ (*MemNets*) (Sukhbaatar et al., 2015) và Bộ Lập trình-Phiên dịch Nơ-ron (*Reed & DeFreitas*, 2015) cho phép các nhà nghiên cứu mô hình hóa thống kê mô tả các hướng tiếp cận tới việc suy luận (*reasoning*) qua nhiều chu kỳ. Những công cụ này cho phép các trạng thái nội tại của mạng nơ-ron sâu được biến đổi liên tục, từ đó có thể thực hiện một chuỗi các bước suy luận, tương tự như cách bộ vi xử lý thay đổi bộ nhớ khi thực hiện một phép tính toán.
- Một bước phát triển quan trọng khác là sự ra đời của GAN (Goodfellow et al., 2014). Trong quá khứ, các phương pháp thống kê để đánh giá hàm mật độ xác suất và các mô hình sinh (*generative models*) tập trung vào việc tìm các phân phối xác suất hợp lý và các thuật toán (thường là xấp xỉ) để lấy mẫu từ các phân phối đó. Vì vậy, những thuật toán này có rất nhiều hạn chế và sự thiếu linh động khi kế thừa chính các mô hình thống kê đó. Phát triển quan trọng của GANs là thay thế các thuật toán lấy mẫu đó bởi một thuật toán với các tham số khả vi (*differentiable*: có thể tính đạo hàm để áp dụng các thuật toán tối ưu dựa trên đó) bất kỳ. Các phương pháp này sau đó được điều chỉnh sao cho bộ phân loại (có hiệu quả giống bài kiểm tra hai mẫu trong xác suất) không thể phân biệt giữa dữ liệu thật và giả. Khả năng sử dụng các thuật toán bất kỳ để sinh dữ liệu đã thúc đẩy phương pháp đánh giá hàm mật độ xác suất khai sinh một loạt các kỹ thuật. Các ví dụ về biến đổi Ngựa thường thành Ngựa Vằn (*Zhu et al.*, 2017) và tạo giả khuôn mặt người nổi tiếng là các minh chứng của quá trình này.

Trong rất nhiều trường hợp, một GPU là không đủ để xử lý một lượng lớn dữ liệu sẵn có cho huấn luyện. Khả năng xây dựng các thuật toán huấn luyện phân tán song song đã cải tiến đáng kể trong thập kỷ vừa rồi. Một trong những thách thức chính trong việc thiết kế các thuật toán cho quy mô lớn là việc thuật toán tối ưu học sâu – hạ gradient ngẫu nhiên – phụ thuộc vào cách xử lý một lượng nhỏ dữ liệu, được gọi là minibatch. Đồng thời, batch nhỏ lại hạn chế sự hiệu quả của GPU. Bởi vậy, nếu ta huấn luyện trên 1024 GPU với 32 ảnh trong một batch sẽ cấu thành một minibatch lớn với 32 ngàn ảnh. Các công trình gần đây, khởi nguồn bởi Li (*Li*, 2017), tiếp tục bởi (*You et al.*, 2017) và (*Jia et al.*, 2018) đẩy kích thước lên tới 64 ngàn mẫu, giảm thời gian huấn luyện ResNet50 trên ImageNet xuống dưới bảy phút so với thời gian huấn luyện hàng nghìn ngày trước đó.

- Khả năng song song hóa việc tính toán cũng đã góp phần quan trọng cho sự phát triển của học tăng cường, ít nhất là với ứng dụng mà có thể tạo và sử dụng môi trường giả lập. Việc này đã dẫn tới sự tiến triển đáng kể ở môn cờ vây, các game Atari, Starcraft, và trong giả lập vật lý (ví dụ, sử dụng MuJoCo). Máy tính đạt được chất lượng vượt mức con người ở các ứng dụng này. Xem thêm mô tả về cách đạt được điều này trong AlphaGo tại (*Silver et al.*, 2016). Tóm lại, học tăng cường làm việc tốt nhất nếu có cực nhiều bộ (trạng thái, hành động, điểm thưởng) để mô hình có thể thử và học cách chúng được liên hệ với nhau như thế nào. Các phần mềm mô phỏng giả lập cung cấp một môi trường như thế.
- Các framework Học Sâu đóng một vai trò thiết yếu trong việc thực hiện hóa các ý tưởng. Các framework thế hệ đầu tiên cho phép dễ dàng mô hình hóa bao gồm Caffe<sup>42</sup>, Torch<sup>43</sup>, và Theano<sup>44</sup>. Rất nhiều bài báo được viết về cách sử dụng các công cụ này. Hiện tại, chúng bị thay thế bởi TensorFlow<sup>45</sup>, thường được sử dụng thông qua API mức cao Keras<sup>46</sup>, CNTK<sup>47</sup>, Caffe 2<sup>48</sup> và Apache MxNet<sup>49</sup>. Thế hệ công cụ thứ ba – công cụ học sâu dạng mệnh lệnh –

<sup>42</sup> <https://github.com/BVLC/caffe>

<sup>43</sup> <https://github.com/torch>

<sup>44</sup> <https://github.com/Theano/Theano>

<sup>45</sup> <https://github.com/tensorflow/tensorflow>

<sup>46</sup> <https://github.com/keras-team/keras>

<sup>47</sup> <https://github.com/Microsoft/CNTK>

<sup>48</sup> <https://github.com/caffe2/caffe2>

<sup>49</sup> <https://github.com/apache/incubator-mxnet>

được tiên phong bởi Chainer<sup>50</sup>, công cụ này sử dụng cú pháp tương tự như Python NumPy để mô tả các mô hình. Ý tưởng này được áp dụng bởi PyTorch<sup>51</sup> và Gluon API<sup>52</sup> của MXNet. Khóa học này sử dụng nhóm công cụ cuối cùng để dạy về học sâu.

Sự phân chia công việc giữa (i) các nhà nghiên cứu hệ thống xây dựng các công cụ tốt hơn và (ii) các nhà mô hình hóa thống kê xây dựng các mạng tốt hơn đã đơn giản hóa công việc một cách đáng kể. Ví dụ, huấn luyện một mô hình hồi quy logistic tuyến tính từng là một bài tập về nhà không đơn giản cho tân nghiên cứu sinh tiến sĩ ngành học máy tại Đại học Carnegie Mellon năm 2014. Hiện nay, tác vụ này có thể đạt được với ít hơn 10 dòng mã, khiến việc này trở nên đơn giản với các lập trình viên.

## 3.6 Các câu chuyện thành công

Trí Tuệ Nhân Tạo có một lịch sử lâu dài trong việc mang đến những kết quả mà khó có thể đạt được bằng các phương pháp khác. Ví dụ, sắp xếp thư tín sử dụng công nghệ nhận dạng ký tự quang. Những hệ thống này được triển khai từ những năm 90 (đây là nguồn của các bộ dữ liệu chữ viết tay nổi tiếng MNIST và USPS). Các hệ thống tương tự cũng được áp dụng vào đọc ngân phiếu nộp tiền vào ngân hàng và tính điểm tín dụng cho ứng viên. Các giao dịch tài chính được kiểm tra có phải lừa đảo không một cách tự động. Đây là nền tảng phát triển cho rất nhiều hệ thống thanh toán điện tử như Paypal, Stripe, AliPay, WeChat, Apple, Visa và MasterCard. Các chương trình máy tính cho cờ vua đã phát triển trong hàng thập kỷ. Học máy đứng sau các hệ thống tìm kiếm, gợi ý, cá nhân hóa và xếp hạng trên mạng Internet. Nói cách khác, trí tuệ nhân tạo và học máy xuất hiện mọi nơi tuy đôi khi ta không để ý thấy.

Chỉ tới gần đây AI mới được chú ý đến, chủ yếu là bởi nó cung cấp giải pháp cho các bài toán mà trước đây được coi là không khả thi.

- Các trợ lý thông minh như: Apple Siri, Amazon Alexa, hay Google Assistant có khả năng trả lời các câu hỏi thoại với độ chính xác chấp nhận được. Việc này cũng bao gồm những tác vụ đơn giản như bật đèn (hữu ích cho người tàn tật) tới đặt lịch hẹn cắt tóc và đưa ra các đoạn hội thoại để hỗ trợ các tổng đài chăm sóc khách hàng. Đây có lẽ là tín hiệu đáng chú ý nhất cho thấy AI đang ảnh hưởng tới cuộc sống thường ngày.
- Một thành phần chính trong trợ lý số là khả năng nhận dạng chính xác tiếng nói. Dần dần độ chính xác của những hệ thống này được cải thiện tới mức tương đương con người ((Xiong et al., 2018)) cho vài ứng dụng cụ thể.
- Tương tự, nhận dạng vật thể cũng đã tiến một bước dài. Đánh giá một vật thể trong ảnh là một tác vụ khó trong năm 2010. Trong bảng xếp hạng ImageNet, (Lin et al., 2010) đạt được tỉ lệ lỗi top-5 là 28%. Tới 2017, (Hu et al., 2018) giảm tỉ lệ lỗi này xuống còn 2,25%. Các kết quả kinh ngạc tương tự cũng đã đạt được trong việc xác định chim cũng như chẩn đoán ung thư da.
- Các trò chơi từng là một pháo đài của trí tuệ loài người. Bắt đầu từ TDGammon [23], một chương trình chơi Backgammon (một môn cờ) sử dụng học tăng cường sai khác thời gian (*temporal difference* – TD), tiến triển trong giải thuật và tính toán đã dẫn đến các thuật toán cho một loạt các ứng dụng. Không giống Backgammon, cờ vua có một không gian trạng thái và tập các nước đi phức tạp hơn nhiều. DeepBlue chiến thắng Gary Kasparov, Campbell et al. (Campbell et al., 2002), bằng cách sử dụng phần cứng chuyên biệt, đa luồng khổng lồ và

<sup>50</sup> <https://github.com/chainer/chainer>

<sup>51</sup> <https://github.com/pytorch/pytorch>

<sup>52</sup> <https://github.com/apache/incubator-mxnet>

thuật toán tìm kiếm hiệu quả trên toàn bộ cây trò chơi. Corse vây còn khó hơn vì không gian trạng thái khổng lồ của nó. Năm 2015, AlphaGo đạt tới đẳng cấp con người, ([Silver et al., 2016](#)) nhờ sử dụng Học Sâu kết hợp với lấy mẫu cây Monte Carlo (*Monte Carlo tree sampling*). Thách thức trong Poker là không gian của trạng thái lớn và nó không được quan sát đầy đủ (ta không biết các quân bài của đối thủ). Libratus vượt chất lượng con người trong môn Poker sử dụng các chiến thuật có cấu trúc một cách hiệu quả ([Brown & Sandholm, 2017](#)). Những điều này thể hiện một sự tiến triển ấn tượng trong các trò chơi và tầm quan trọng của các thuật toán nâng cao trong đó.

- Dấu hiệu khác của tiến triển trong AI là sự phát triển của xe hơi và xe tải tự hành. Trong khi hệ thống tự động hoàn toàn còn xa mới đạt được, tiến triển ấn tượng đã được tạo ra theo hướng này với việc các công ty như Tesla, NVIDIA và Waymo ra mắt các sản phẩm ít nhất hỗ trợ bán tự động. Điều khiến tự động hoàn toàn mang nhiều thách thức là việc lái xe chuẩn mực đòi hỏi khả năng tiếp nhận, suy đoán và tích hợp các quy tắc vào hệ thống. Tại thời điểm hiện tại, học máy được sử dụng chủ yếu trong phần thị giác máy tính của các bài toán này. Phần còn lại vẫn phụ thuộc chủ yếu bởi những điều chỉnh của các kỹ sư.

Danh sách trên đây chỉ lướt qua những ứng dụng mà học máy có ảnh hưởng lớn. Ngoài ra, robot, hậu cần, sinh học điện toán, vật lý hạt và thiên văn học cũng tạo ra những thành quả ấn tượng gần đây phần nào nhờ vào học máy. Bởi vậy, Học Máy đang trở thành một công cụ phổ biến cho các kỹ sư và nhà khoa học.

Gần đây, câu hỏi về ngày tận thế do AI, hay điểm kỳ dị (*singularity*) của AI đã được nhắc tới trong các bài viết phi kỹ thuật về AI. Đã có những nỗi lo sợ về việc các hệ thống học máy bằng cách nào đó sẽ trở nên có cảm xúc và ra quyết định độc lập với những lập trình viên (và chủ nhân) về những điều ảnh hưởng trực tiếp tới sự sống của nhân loại. Trong phạm vi nào đó, AI đã ảnh hưởng tới sự sống của con người một cách trực tiếp, chẳng hạn như điểm tín dụng được tính tự động, tự động điều hướng xe hơi, hay các quyết định về việc liệu có chấp nhận bảo lãnh hay không sử dụng đầu vào là dữ liệu thống kê. Hoặc ít nghiêm trọng hơn, ta có thể yêu cầu Alexa bật máy pha cà phê.

May mắn thay, chúng ta còn xa mới đạt được một hệ thống AI có cảm xúc sẵn sàng điều khiển chủ nhân của nó (hay đốt cháy cà phê của họ). Thứ nhất, các hệ thống AI được thiết kế, huấn luyện và triển khai trong một môi trường cụ thể hướng mục đích. Trong khi hành vi của chúng có thể tạo ra ảo giác về trí tuệ phổ quát, đó vẫn là một tổ hợp của các quy tắc và các mô hình thống kê. Thứ hai, hiện tại các công cụ cho trí tuệ nhân tạo phổ quát đơn giản là không tồn tại. Chúng không thể tự cải thiện, hoài nghi bản thân, không thể thay đổi, mở rộng và tự cải thiện cấu trúc trong khi cố gắng giải quyết các tác vụ thông thường.

Một mối lo cấp bách hơn đó là AI có thể được sử dụng trong cuộc sống thường nhật như thế nào. Nhiều khả năng rất nhiều tác vụ đơn giản đang được thực hiện bởi tài xế xe tải và trợ lý cửa hàng có thể và sẽ bị tự động hóa. Các robot nông trại sẽ không những có khả năng làm giảm chi phí của nông nghiệp hữu cơ mà còn tự động hóa quá trình thu hoạch. Thời điểm này của cuộc cách mạng công nghiệp có thể có những hậu quả lan rộng khắp toàn xã hội (tài xế xe tải và trợ lý cửa hàng là một vài trong số những ngành phổ biến nhất ở nhiều địa phương). Đối với các mô hình thống kê khi được áp dụng không cẩn thận, có thể dẫn đến các quyết định phân biệt chủng tộc, giới tính hoặc tuổi tác và gây nên những nỗi lo có cơ sở về tính công bằng nếu chúng được tự động hóa để đưa ra các quyết định có nhiều hệ lụy. Việc sử dụng các thuật toán này một cách cẩn thận là rất quan trọng. Với những gì ta biết ngày nay, việc này dấy lên một nỗi lo lớn hơn so với khả năng hủy diệt loài người của các siêu trí tuệ độc ác.

## 3.7 Tóm tắt

- Học máy nghiên cứu cách mà các hệ thống máy tính tận dụng *kinh nghiệm* (thường là dữ liệu) để cải thiện chất lượng trong những tác vụ cụ thể. Lĩnh vực này kết hợp các ý tưởng từ thống kê, khai phá dữ liệu, trí tuệ nhân tạo và tối ưu hóa. Học máy thường được sử dụng như một công cụ để triển khai các giải pháp trí tuệ nhân tạo.
- Là một nhánh của học máy, học biểu diễn (*representational learning*) tập trung vào việc tự động tìm kiếm phương pháp biểu diễn dữ liệu thích hợp, phần lớn thông qua việc học một quá trình biến đổi gồm nhiều bước.
- Hầu hết các tiến triển gần đây trong học sâu đạt được nhờ một lượng lớn dữ liệu thu thập từ các cảm biến giá rẻ và các ứng dụng quy mô Internet, cùng với sự phát triển đáng kể trong điện toán, chủ yếu là GPU.
- Việc tối ưu hóa toàn bộ hệ thống là yếu tố chính để đạt được chất lượng tốt. Sự sẵn có của các framework học sâu hiệu quả giúp cho việc thiết kế và triển khai tối ưu hóa trở nên dễ dàng hơn rất nhiều.

## 3.8 Bài tập

1. Phần nào của mã nguồn mà bạn đang viết có thể “được học”, tức có thể được cải thiện bằng cách học và tự động xác định các lựa chọn thiết kế? Trong mã nguồn của bạn có hiện diện các lựa chọn thiết kế dựa trên trực giác không?
2. Những bài toán nào bạn từng gặp có nhiều cách giải quyết, nhưng lại không có cách cụ thể nào để tự động hóa? Những bài toán này có thể rất phù hợp để áp dụng học sâu.
3. Nếu xem sự phát triển của trí tuệ nhân tạo như một cuộc cách mạng công nghiệp mới thì mối quan hệ giữa thuật toán và dữ liệu là gì? Nó có tương tự như động cơ hơi nước và than đá không (sự khác nhau căn bản là gì)?
4. Bạn còn có thể áp dụng phương pháp huấn luyện đầu-cuối ở lĩnh vực nào nữa? Vật lý? Kỹ thuật? Kinh tế lượng?

## 3.9 Thảo luận

- Tiếng Anh<sup>53</sup>
- Tiếng Việt<sup>54</sup>

<sup>53</sup> <https://discuss.mxnet.io/t/2310>

<sup>54</sup> <https://forum.machinelearningcoban.com/c/d21>

### **3.10 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Lê Khắc Hồng Phúc
- Vũ Hữu Tiệp
- Sâm Thế Hải
- Hoàng Trọng Tuấn
- Trần Thị Hồng Hạnh
- Đoàn Võ Duy Thanh
- Phạm Chí Thành
- Mai Sơn Hải
- Vũ Đình Quyền
- Nguyễn Cảnh Thưởng
- Lê Đàm Hồng Lộc
- Nguyễn Lê Quang Nhật



# 4 | Sơ bộ

Để bắt đầu với học sâu, ta sẽ cần nắm bắt một vài kỹ năng cơ bản. Tất cả những vấn đề về học máy đều có liên quan đến việc trích xuất thông tin từ dữ liệu. Vì vậy, chúng tôi sẽ bắt đầu bằng cách học các kỹ năng thực tế để lưu trữ, thao tác và xử lý dữ liệu.

Hơn nữa, học máy thường yêu cầu làm việc với các tập dữ liệu lớn, mà chúng ta có thể coi như ở dạng bảng, trong đó các hàng tương ứng với các mẫu và các cột tương ứng với các thuộc tính. Đại số tuyến tính cung cấp cho ta một tập kỹ thuật mạnh mẽ để làm việc với dữ liệu dạng bảng. Chúng ta sẽ không đi quá sâu mà chỉ tập trung cơ bản vào các toán tử ma trận cơ bản và cách thực thi chúng.

Bên cạnh đó, học sâu luôn liên quan đến tối ưu hoá. Chúng ta có một mô hình với bộ tham số và muốn tìm ra các tham số khớp với dữ liệu *nhất*. Việc xác định cách điều chỉnh từng tham số ở mỗi bước trong thuật toán đòi hỏi một chút kiến thức về giải tích, mà sẽ được giới thiệu ngắn gọn dưới đây. May thay, gói autograd sẽ tự động tính đạo hàm cho chúng ta, và sẽ được đề cập ngay sau đó.

Kế tiếp, học máy liên quan đến việc đưa ra những dự đoán như: Xác định giá trị của một số thuộc tính chưa biết dựa trên thông tin quan sát được? Để suy luận chặt chẽ dưới sự bất định, chúng ta sẽ cần tìm đến ngôn ngữ của xác suất.

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

Cuốn sách này đã cung cấp nội dung toán học ở mức tối thiểu cần có để có được sự hiểu biết đúng đắn về học sâu. Tuy nhiên, điều đó không đồng nghĩa rằng cuốn sách này không cần các kiến thức toán học. Do vậy, chương này sẽ giới thiệu nhanh về các kiến thức toán học cơ bản và thông dụng, cho phép tất cả mọi người tối thiểu là sẽ hiểu được *hầu hết* nội dung toán trong quyển sách này. Nếu bạn muốn hiểu *tất cả* nội dung toán học, hãy tham khảo thêm [Section 20](#).

## 4.1 Thao tác với Dữ liệu

Muốn thực hiện bất cứ điều gì, chúng ta đều cần một cách nào đó để lưu trữ và thao tác với dữ liệu. Thường sẽ có hai điều quan trọng chúng ta cần làm với dữ liệu: (i) thu thập; và (ii) xử lý sau khi đã có dữ liệu trên máy tính. Sẽ thật vô nghĩa khi thu thập dữ liệu mà không có cách để lưu trữ nó, vậy nên trước tiên hãy cùng làm quen với dữ liệu tổng hợp. Để bắt đầu, chúng tôi giới thiệu mảng  $n$  chiều (ndarray) – công cụ chính trong MXNet để lưu trữ và biến đổi dữ liệu. Trong MXNet, ndarray là một lớp và mỗi thực thể của lớp đó là “một ndarray”.

Nếu bạn từng làm việc với NumPy, gói tính toán phổ biến nhất trong Python, bạn sẽ thấy mục này quen thuộc. Việc này là có chủ đích. Chúng tôi thiết kế ndarray trong MXNet là một dạng mở rộng của ndarray trong NumPy với một vài tính năng đặc biệt. Thứ nhất, ndarray trong MXNet hỗ trợ

tính toán phi đồng bộ trên CPU, GPU, và các kiến trúc phân tán đám mây, trong khi NumPy chỉ hỗ trợ tính toán trên CPU. Thứ hai, ndaray trong MXNet hỗ trợ tính vi phân tự động. Những tính chất này khiến ndarray của MXNet phù hợp với học sâu. Thông qua cuốn sách, nếu không nói gì thêm, chúng ta ngầm hiểu ndarray là ndarray của MXNet.

### 4.1.1 Bắt đầu

Trong mục này, mục tiêu của chúng tôi là trang bị cho bạn các kiến thức toán cơ bản và cài đặt các công cụ tính toán mà bạn sẽ xây dựng dựa trên nó xuyên suốt cuốn sách này. Đừng lo nếu bạn gặp khó khăn với các khái niệm toán khó hiểu hoặc các hàm trong thư viện tính toán. Các mục tiếp theo sẽ nhắc lại những khái niệm này trong từng ngữ cảnh kèm theo ví dụ thực tiễn. Mặt khác, nếu bạn đã có kiến thức nền tảng và muốn đi sâu hơn vào các nội dung toán, bạn có thể bỏ qua mục này.

Để bắt đầu, ta cần khai báo mô-đun np (numpy) và npx (numpy\_extension) từ MXNet. Ở đây, mô-đun np bao gồm các hàm hỗ trợ bởi NumPy, trong khi mô-đun npx chứa một tập các hàm mở rộng được phát triển để hỗ trợ học sâu trong một môi trường giống với NumPy. Khi sử dụng ndarray, ta luôn cần gọi hàm set\_np: điều này nhằm đảm bảo sự tương thích của việc xử lý ndarray bằng các thành phần khác của MXNet.

```
from mxnet import np, npx  
npx.set_np()
```

Một ndarray biểu diễn một mảng (có thể đa chiều) các giá trị số. Với một trực, một ndarray tương ứng (trong toán) với một *vector*. Với hai trực, một ndarray tương ứng với một *ma trận*. Các mảng với nhiều hơn hai trực không có tên toán học cụ thể-chúng được gọi chung là *tensor*.

Để bắt đầu, chúng ta sử dụng arange để tạo một vector hàng x chứa 12 số nguyên đầu tiên bắt đầu từ 0, nhưng được khởi tạo mặc định dưới dạng số thực. Mỗi giá trị trong một ndarray được gọi là một *phần tử* của ndarray đó. Như vậy, có 12 phần tử trong ndarray x. Nếu không nói gì thêm, một ndarray mới sẽ được lưu trong bộ nhớ chính và được tính toán trên CPU.

```
x = np.arange(12)  
x
```

Chúng ta có thể lấy *kích thước* (độ dài theo mỗi trực) của ndarray bằng thuộc tính shape.

```
x.shape
```

Nếu chỉ muốn biết tổng số phần tử của một ndarray, nghĩa là tích của tất cả các thành phần trong shape, ta có thể sử dụng thuộc tính size. Vì ta đang làm việc với một vector, cả shape và size của nó đều chứa cùng một phần tử duy nhất.

```
x.size
```

Để thay đổi kích thước của một ndarray mà không làm thay đổi số lượng phần tử cũng như giá trị của chúng, ta có thể gọi hàm reshape. Ví dụ, ta có thể biến đổi ndarray x trong ví dụ trên, từ một vector hàng với kích thước (12,) sang một ma trận với kích thước (3, 4). ndarray mới này chứa 12 phần tử y hệt, nhưng được xem như một ma trận với 3 hàng và 4 cột. Mặc dù kích thước thay đổi, các phần tử của x vẫn giữ nguyên. Chú ý rằng size giữ nguyên khi thay đổi kích thước.

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

Việc chỉ định cụ thể mọi chiều khi thay đổi kích thước là không cần thiết. Nếu kích thước mong muốn là một ma trận với kích thước (chiều\_cao, chiều\_rộng), thì sau khi biết chiều\_rộng, chiều\_cao có thể được ngầm suy ra. Tại sao ta lại cần phải tự làm phép tính chia? Trong ví dụ trên, để có được một ma trận với 3 hàng, chúng ta phải chỉ định rõ rằng nó có 3 hàng và 4 cột. May mắn thay, ndarray có thể tự động tính một chiều từ các chiều còn lại. Ta có thể dùng chức năng này bằng cách đặt -1 cho chiều mà ta muốn ndarray tự suy ra. Trong trường hợp vừa rồi, thay vì gọi `x.reshape(3, 4)`, ta có thể gọi `x.reshape(-1, 4)` hoặc `x.reshape(3, -1)`.

Phương thức `empty` lấy một đoạn bộ nhớ và trả về một ma trận mà không thay đổi các giá trị sẵn có tại đoạn bộ nhớ đó. Việc này có hiệu quả tính toán đáng kể nhưng ta phải cẩn trọng bởi các phần tử đó có thể chứa bất kỳ giá trị nào, kể cả các số rất lớn!

```
np.empty((3, 4))
```

Thông thường ta muốn khởi tạo các ma trận với các giá trị bằng không, bằng một, bằng hằng số nào đó hoặc bằng các mẫu ngẫu nhiên lấy từ một phân phối cụ thể. Ta có thể tạo một ndarray biểu diễn một tensor với tất cả các phần tử bằng 0 và có kích thước (2, 3, 4) như sau:

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

Tương tự, ta có thể tạo các tensor với các phần tử bằng 1 như sau:

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

Ta thường muốn lấy mẫu ngẫu nhiên cho mỗi phần tử trong một ndarray từ một phân phối xác suất. Ví dụ, khi xây dựng các mảng để chứa các tham số của một mạng nơ-ron, ta thường khởi tạo chúng với các giá trị ngẫu nhiên. Đoạn mã dưới đây tạo một ndarray có kích thước (3, 4) với các phần tử được lấy mẫu ngẫu nhiên từ một phân phối Gauss (phân phối chuẩn) với trung bình bằng 0 và độ lệch chuẩn 1.

```
np.random.normal(0, 1, size=(3, 4))
```

Ta cũng có thể khởi tạo giá trị cụ thể cho mỗi phần tử trong ndarray mong muốn bằng cách đưa vào một mảng Python (hoặc mảng của mảng) chứa các giá trị số. Ở đây, mảng ngoài cùng tương ứng với trục 0, và mảng bên trong tương ứng với trục 1.

```
np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

## 4.1.2 Phép toán

Cuốn sách này không nói về kỹ thuật phần mềm. Chúng tôi không chỉ hứng thú với việc đơn giản đọc và ghi dữ liệu vào/từ các mảng mà còn muốn thực hiện các phép toán trên các mảng này. Một vài phép toán đơn giản và hữu ích nhất là các phép toán tác động lên *từng phần tử* (*elementwise*). Các phép toán này hoạt động như những phép toán chuẩn trên số vô hướng áp dụng lên từng phần tử của mảng. Với những hàm nhận hai mảng đầu vào, phép toán theo từng thành phần được áp dụng trên từng cặp phần tử tương ứng của hai mảng. Ta có thể tạo một hàm theo từng phần tử từ một hàm bất kỳ ánh xạ từ một số vô hướng tới một số vô hướng.

Trong toán học, ta ký hiệu một toán tử *đơn ngôi* vô hướng (lấy một đầu vào) bởi  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Điều này nghĩa là hàm số ánh xạ từ một số thực bất kỳ ( $\mathbb{R}$ ) sang một số thực khác. Tương tự, ta ký hiệu một toán tử *hai ngôi* vô hướng (lấy hai đầu vào, trả về một đầu ra) bởi  $f : \mathbb{R}, \mathbb{R} \rightarrow \mathbb{R}$ . Cho trước hai vector bất kỳ  $\mathbf{u}$  và  $\mathbf{v}$  với cùng kích thước, và một toán tử hai ngôi  $f$ , ta có thể tính được một vector  $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$  bằng cách tính  $c_i \leftarrow f(u_i, v_i)$  cho mọi  $i$  với  $c_i, u_i$ , và  $v_i$  là các phần tử thứ  $i$  của vector  $\mathbf{c}, \mathbf{u}$ , và  $\mathbf{v}$ . Ở đây, chúng ta tạo một hàm trả về vector  $F : \mathbb{R}^d, \mathbb{R}^d \rightarrow \mathbb{R}^d$  bằng cách áp dụng hàm  $f$  lên từng phần tử.

Trong MXNet, các phép toán tiêu chuẩn (+, -, \*, /, và \*\*) là các phép toán theo từng phần tử trên các tensor đồng kích thước bất kỳ. Ta có thể gọi những phép toán theo từng phần tử lên hai tensor đồng kích thước. Trong ví dụ dưới đây, các dấu phẩy được sử dụng để tạo một tuple 5 phần tử với mỗi phần tử là kết quả của một phép toán theo từng phần tử.

```
x = np.array([1, 2, 4, 8])
y = np.array([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y # The ** operator is exponentiation
```

Rất nhiều các phép toán khác có thể được áp dụng theo từng phần tử, bao gồm các phép toán đơn ngôi như hàm mũ cơ số  $e$ .

```
np.exp(x)
```

Ngoài các phép tính theo từng phần tử, ta cũng có thể thực hiện các phép toán đại số tuyến tính, bao gồm tích vô hướng của hai vector và phép nhân ma trận. Chúng ta sẽ giải thích những điểm quan trọng của đại số tuyến tính (mà không cần kiến thức nền tảng) trong [Section 4.3](#).

Ta cũng có thể *nối* nhiều ndarray với nhau, xếp chồng chúng lên nhau để tạo ra một ndarray lớn hơn. Ta chỉ cần cung cấp một danh sách các ndarray và khai báo chúng được nối theo trực nào. Ví dụ dưới đây thể hiện cách nối hai ma trận theo hàng (trục 0, phần tử đầu tiên của kích thước) và theo cột (trục 1, phần tử thứ hai của kích thước). Ta có thể thấy rằng, cách thứ nhất tạo một ndarray với độ dài trục 0 (6) bằng tổng các độ dài trục 0 của hai ndarray đầu vào (3 + 3); trong khi cách thứ hai tạo một ndarray với độ dài trục 1 (8) bằng tổng các độ dài trục 1 của hai ndarray đầu vào (4 + 4).

```
x = np.arange(12).reshape(3, 4)
y = np.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
np.concatenate([x, y], axis=0), np.concatenate([x, y], axis=1)
```

Đôi khi, ta muốn tạo một ndarray nhị phân thông qua các *mệnh đề logic*. Lấy  $x == y$  làm ví dụ. Với mỗi vị trí, nếu giá trị của  $x$  và  $y$  tại vị trí đó bằng nhau thì phần tử tương ứng trong ndarray mới lấy giá trị 1, nghĩa là mệnh đề logic  $x == y$  là đúng tại vị trí đó; ngược lại vị trí đó lấy giá trị 0.

```
x == y
```

Lấy tổng mọi phần tử trong một ndarray tạo ra một ndarray với chỉ một phần tử.

```
x.sum()
```

Ta cũng có thể thay `x.sum()` bởi `np.sum(x)`.

### 4.1.3 Cơ chế Lan truyền

Trong mục trên, ta đã thấy cách thực hiện các phép toán theo từng phần tử với hai ndarray đồng kích thước. Trong những điều kiện nhất định, thậm chí khi kích thước khác nhau, ta vẫn có thể thực hiện các phép toán theo từng phần tử bằng cách sử dụng *cơ chế lan truyền* (*broadcasting mechanism*). Cơ chế này hoạt động như sau: Thứ nhất, mở rộng một hoặc cả hai mảng bằng cách lặp lại các phần tử một cách hợp lý sao cho sau phép biến đổi này, hai ndarray có cùng kích thước. Thứ hai, thực hiện các phép toán theo từng phần tử với hai mảng mới này.

Trong hầu hết các trường hợp, chúng ta lan truyền một mảng theo trực có độ dài ban đầu là 1, như ví dụ dưới đây:

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

Vì `a` và `b` là các ma trận có kích thước lần lượt là  $3 \times 1$  và  $1 \times 2$ , kích thước của chúng không khớp nếu ta muốn thực hiện phép cộng. Ta *lan truyền* các phần tử của cả hai ma trận thành các ma trận  $3 \times 2$  như sau: lặp lại các cột trong ma trận `a` và các hàng trong ma trận `b` trước khi cộng chúng theo từng phần tử.

```
a + b
```

### 4.1.4 Chỉ số và Cắt chọn mảng

Cũng giống như trong bất kỳ mảng Python khác, các phần tử trong một ndarray có thể được truy cập theo chỉ số. Tương tự, phần tử đầu tiên có chỉ số 0 và khoảng được cắt chọn bao gồm phần tử đầu tiên nhưng *không tính* phần tử cuối cùng. Và trong các danh sách Python tiêu chuẩn, chúng ta có thể truy cập các phần tử theo vị trí đếm ngược từ cuối danh sách bằng cách sử dụng các chỉ số âm.

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

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

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
```

Nếu chúng ta muốn gán cùng một giá trị cho nhiều phần tử, chúng ta chỉ cần trỏ đến tất cả các phần tử đó và gán giá trị cho chúng. Chẳng hạn, `[0:2, :]` truy cập vào hàng thứ nhất và thứ hai,

trong đó : lấy tất cả các phần tử dọc theo trục 1 (cột). Ở đây chúng ta đã thảo luận về cách truy cập vào ma trận, nhưng tất nhiên phương thức này cũng áp dụng cho các vector và tensor với nhiều hơn 2 chiều.

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

#### 4.1.5 Tiết kiệm Bộ nhớ

Ở ví dụ trước, mỗi khi chạy một phép tính, chúng ta sẽ cấp phát bộ nhớ mới để lưu trữ kết quả của lượt chạy đó. Cụ thể hơn, nếu viết  $y = x + y$ , ta sẽ ngừng tham chiếu đến ndarray mà  $y$  đã chỉ đến trước đó và thay vào đó gán  $y$  vào bộ nhớ được cấp phát mới. Trong ví dụ tiếp theo, chúng ta sẽ minh họa việc này với hàm `id()` của Python - hàm cung cấp địa chỉ chính xác của một đối tượng được tham chiếu trong bộ nhớ. Sau khi chạy  $y = y + x$ , chúng ta nhận ra rằng `id(y)` chỉ đến một địa chỉ khác. Đó là bởi vì Python trước hết sẽ tính  $y + x$ , cấp phát bộ nhớ mới cho kết quả trả về và gán  $y$  vào địa chỉ mới này trong bộ nhớ.

```
before = id(y)  
y = y + x  
id(y) == before
```

Đây có thể là điều không mong muốn vì hai lý do. Thứ nhất, không phải lúc nào chúng ta cũng muốn cấp phát bộ nhớ không cần thiết. Trong học máy, ta có thể có đến hàng trăm megabytes tham số và cập nhật tất cả chúng nhiều lần mỗi giây, và thường thì ta muốn thực thi các cập nhật này *tại chỗ*. Thứ hai, chúng ta có thể trỏ đến cùng tham số từ nhiều biến khác nhau. Nếu không cập nhật tại chỗ, các bộ nhớ đã bị loại bỏ sẽ không được giải phóng, dẫn đến khả năng một số chỗ trong mã nguồn sẽ vô tình tham chiếu lại các tham số cũ.

May mắn thay, ta có thể dễ dàng thực hiện các phép tính tại chỗ với MXNet. Chúng ta có thể gán kết quả của một phép tính cho một mảng đã được cấp phát trước đó bằng ký hiệu cắt chọn (*slice notation*), ví dụ,  $y[:, :] = <\text{expression}>$ . Để minh họa khái niệm này, đầu tiên chúng ta tạo một ma trận mới  $z$  có cùng kích thước với ma trận  $y$ , sử dụng `zeros_like` để gán giá trị khởi tạo bằng 0.

```
z = np.zeros_like(y)  
print('id(z):', id(z))  
z[:, :] = x + y  
print('id(z):', id(z))
```

Nếu các tính toán tiếp theo không tái sử dụng giá trị của  $x$ , chúng ta có thể viết  $x[:, :] = x + y$  hoặc  $x += y$  để giảm thiểu việc sử dụng bộ nhớ không cần thiết trong quá trình tính toán.

```
before = id(x)  
x += y  
id(x) == before
```

#### 4.1.6 Chuyển đổi sang các Đối Tượng Python Khác

Chuyển đổi một MXNet ndarray sang NumPy ndarray hoặc ngược lại là khá đơn giản. Tuy nhiên, kết quả của phép chuyển đổi này không chia sẻ bộ nhớ với đối tượng cũ. Điểm bất tiện này tuy nhỏ nhưng lại khá quan trọng: khi bạn thực hiện các phép tính trên CPU hoặc GPUs, bạn không muốn MXNet dừng việc tính toán để chờ xem liệu gói Numpy của Python có sử dụng cùng bộ nhớ đó để làm việc khác không. Hàm `array` và `asnumpy` sẽ giúp bạn giải quyết vấn đề này.

```
a = x.asnumpy()
b = np.array(a)
type(a), type(b)
```

Để chuyển đổi một mảng ndarray chứa một phần tử sang số vô hướng Python, ta có thể gọi hàm `item` hoặc các hàm có sẵn trong Python.

```
a = np.array([3.5])
a, a.item(), float(a), int(a)
```

#### 4.1.7 Tổng kết

- MXNet ndarray là phần mở rộng của NumPy ndarray với một số ưu thế vượt trội giúp cho nó phù hợp với học sâu.
- MXNet ndarray cung cấp nhiều chức năng bao gồm các phép toán cơ bản, cơ chế lan truyền (*broadcasting*), chỉ số (*indexing*), cắt chọn (*slicing*), tiết kiệm bộ nhớ và khả năng chuyển đổi sang các đối tượng Python khác.

#### 4.1.8 Bài tập

- Chạy đoạn mã nguồn trong mục này. Thay đổi điều kiện mệnh đề `x == y` sang `x < y` hoặc `x > y`, sau đó kiểm tra dạng của ndarray nhận được.
- Thay hai ndarray trong phép tính theo từng phần tử ở phần cơ chế lan truyền (*broadcasting mechanism*) với các ndarray có kích thước khác, ví dụ như tensor ba chiều. Kết quả có giống như bạn mong đợi hay không?

#### 4.1.9 Thảo luận

- Tiếng Anh<sup>55</sup>
- Tiếng Việt<sup>56</sup>

<sup>55</sup> <https://discuss.mxnet.io/t/2315>

<sup>56</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### 4.1.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Trần Thị Hồng Hạnh
- Phạm Minh Đức
- Lê Đàm Hồng Lộc
- Nguyễn Lê Quang Nhật

## 4.2 Tiền xử lý dữ liệu

Cho đến giờ chúng tôi đã đề cập tới rất nhiều kỹ thuật thao tác dữ liệu được lưu trong dạng ndarray. Nhưng để áp dụng học sâu vào giải quyết các vấn đề thực tế, ta thường phải bắt đầu bằng việc xử lý dữ liệu thô, chứ không có luôn dữ liệu ngăn nắp được chuẩn bị sẵn trong định dạng ndarray. Trong số các công cụ phân tích dữ liệu phổ biến của Python, gói pandas khá được ưa chuộng. Cũng như nhiều gói khác trong hệ sinh thái rộng lớn của Python, ‘pandas’ có thể được sử dụng kết hợp với định dạng ndarray. Vì vậy, chúng ta sẽ đi nhanh qua các bước để tiền xử lý dữ liệu thô bằng pandas rồi đổi chúng sang dạng ndarray. Nhiều kỹ thuật tiền xử lý dữ liệu khác sẽ được giới thiệu trong các chương sau.

### 4.2.1 Đọc tập dữ liệu

Để lấy ví dụ, ta bắt đầu bằng việc tạo một tập dữ liệu nhân tạo lưu trong file csv `../data/house_tiny.csv` (csv - comma-separated values - giá trị tách nhau bằng dấu phẩy). Dữ liệu lưu ở các định dạng khác cũng có thể được xử lý tương tự. Hàm `mkdir_if_not_exist` dưới đây để đảm bảo rằng thư mục `../data` tồn tại. Chú thích `# Saved in the d2l package for later use (Lưu lại trong gói d2l để dùng sau)` là kí hiệu đánh dấu các hàm, lớp hoặc các lệnh import được lưu trong gói d2l, để sau này ta có thể trực tiếp gọi hàm `d2l.mkdir_if_not_exist()`.

```
import os

# Saved in the d2l package for later use
def mkdir_if_not_exist(path):
    if not isinstance(path, str):
        path = os.path.join(*path)
    if not os.path.exists(path):
        os.makedirs(path)
```

Sau đây ta ghi tập dữ liệu vào file csv theo từng hàng một.

```

data_file = '../data/house_tiny.csv'
mkdir_if_not_exist('../data')
with open(data_file, 'w') as f:
    f.write('NumRooms,Alley,Price\n') # Column names
    f.write('NA,Pave,127500\n') # Each row is a data point
    f.write('2,NA,106000\n')
    f.write('4,NA,178100\n')
    f.write('NA,NA,140000\n')

```

Để nạp tập dữ liệu thô từ tệp csv vừa được tạo ra, ta dùng gói thư viện pandas và gọi hàm `read_csv`. Bộ dữ liệu này có 4 hàng và 3 cột, trong đó mỗi hàng biểu thị số phòng (“NumRooms”), kiểu lối đi (“Alley”), và giá (“Price”) của căn nhà.

```

# 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)

```

## 4.2.2 Xử lý dữ liệu thiếu

Để ý rằng giá trị “NaN” là các giá trị bị thiếu. Để xử lý dữ liệu thiếu, các cách thường được áp dụng là *quy buộc (imputation)* và *xoá bỏ (deletion)*, trong đó quy buộc thay thế giá trị bị thiếu bằng giá trị khác, trong khi xoá bỏ sẽ bỏ qua các giá trị bị thiếu. Dưới đây chúng ta xem xét phương pháp quy buộc.

Bằng phương pháp đánh chỉ số theo số nguyên (`iloc`), chúng ta tách data thành `inputs` (tương ứng với hai cột đầu) và `outputs` (tương ứng với cột cuối cùng). Với các giá trị số bị thiếu trong `inputs`, ta thay thế phần tử “NaN” bằng giá trị trung bình cộng của cùng cột đó.

```

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

```

Với các giá trị dạng hạng mục hoặc số rời rạc trong `inputs`, ta coi “NaN” là một mục riêng. Vì cột “Alley” chỉ nhận 2 giá trị riêng lẻ là “Pave” (được lát gạch) và “NaN”, pandas có thể tự động chuyển cột này thành 2 cột “Alley\_Pave” và “Alley\_nan”. Những hàng có kiểu lối đi là “Pave” sẽ có giá trị của cột “Alley\_Pave” và cột “Alley\_nan” tương ứng là 1 và 0. Hàng mà không có giá trị cho kiểu lối đi sẽ có giá trị cột “Alley\_Pave” và cột “Alley\_nan” lần lượt là 0 và 1.

```

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

```

### 4.2.3 Chuyển sang định dạng ndarray

Giờ thì toàn bộ các giá trị trong inputs và outputs đã ở dạng số, chúng đã có thể được chuyển sang định dạng ndarray. Khi đã ở định dạng này, chúng có thể được biến đổi và xử lý với những chức năng của ndarray đã được giới thiệu ở Section 4.1.

```
from mxnet import np  
  
X, y = np.array(inputs.values), np.array(outputs.values)  
X, y
```

### 4.2.4 Tóm tắt

- Cũng như nhiều gói mở rộng trong hệ sinh thái khổng lồ của Python, pandas có thể làm việc được với ndarray.
- Phương pháp quy buộc hoặc xoá bỏ có thể dùng để xử lý dữ liệu bị thiếu.

### 4.2.5 Bài tập

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

1. Xoá cột có nhiều giá trị bị thiếu nhất.
2. Chuyển bộ dữ liệu đã được xử lý sang định dạng ndarray.

### 4.2.6 Thảo luận

- Tiếng Anh<sup>57</sup>
- Tiếng Việt<sup>58</sup>

### 4.2.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh
- Đoàn Võ Duy Thành
- Vũ Hữu Tiệp
- Mai Sơn Hải

<sup>57</sup> <https://discuss.mxnet.io/t/2315>

<sup>58</sup> <https://forum.machinelearningcoban.com/c/d21>

## 4.3 Đại số tuyến tính

Bây giờ bạn đã có thể lưu trữ và xử lý dữ liệu, hãy cùng ôn qua những kiến thức đại số tuyến tính cần thiết để hiểu và lập trình hầu hết các mô hình được nhắc tới trong quyển sách này. Dưới đây, chúng tôi giới thiệu các đối tượng toán học, số học và phép tính cơ bản trong đại số tuyến tính, biểu diễn chúng bằng cả ký hiệu toán học và cách triển khai lập trình tương ứng.

### 4.3.1 Số vô hướng

Nếu bạn chưa từng học đại số tuyến tính hay học máy, có lẽ bạn mới chỉ có kinh nghiệm làm toán với từng con số riêng lẻ. Và nếu bạn đã từng phải cân bằng sổ thu chi hoặc đơn giản là trả tiền cho bữa ăn, thì hẳn bạn đã biết cách thực hiện các phép tính cơ bản như cộng trừ nhân chia các cặp số. Ví dụ, nhiệt độ tại Palo Alto là 52 độ Fahrenheit. Chúng ta gọi các giá trị mà chỉ bao gồm một số duy nhất là *số vô hướng* (*scalar*). Nếu bạn muốn chuyển giá trị nhiệt độ trên sang độ Celsius (thang đo nhiệt độ hợp lý hơn theo hệ mét), bạn sẽ phải tính biểu thức  $c = \frac{5}{9}(f - 32)$  với giá trị  $f$  bằng 52. Trong phương trình trên, mỗi số hạng  $-5, 9$  và  $32$  — là các số vô hướng. Các ký hiệu  $c$  và  $f$  được gọi là *biến* và chúng biểu diễn các giá trị số vô hướng chưa biết.

Trong quyển sách này, chúng tôi sẽ tuân theo quy ước ký hiệu các biến vô hướng bằng các chữ cái viết thường (chẳng hạn  $x, y$  và  $z$ ). Chúng tôi ký hiệu không gian (liên tục) của tất cả các số *thực* vô hướng là  $\mathbb{R}$ . Vì tính thiết thực, chúng tôi sẽ bỏ qua định nghĩa chính xác của *không gian*. Nhưng bạn cần nhớ  $x \in \mathbb{R}$  là cách toán học để thể hiện  $x$  là một số thực vô hướng. Ký hiệu  $\in$  đọc là “thuộc” và đơn thuần biểu diễn việc phần tử thuộc một tập hợp. Tương tự, ta có thể viết  $x, y \in \{0, 1\}$  để ký hiệu cho việc các số  $x$  và  $y$  chỉ có thể nhận giá trị 0 hoặc 1.

Trong mã nguồn MXNet, một số vô hướng được biểu diễn bằng một *ndarray* với chỉ một phần tử. Trong đoạn mã dưới đây, chúng ta khởi tạo hai số vô hướng và thực hiện các phép tính quen thuộc như cộng, trừ, nhân, chia và lũy thừa với chúng.

```
from mxnet import np, npx
npx.set_np()

x = np.array(3.0)
y = np.array(2.0)

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

### 4.3.2 Vector

Bạn có thể xem vector đơn thuần như một dãy các số vô hướng. Chúng ta gọi các giá trị đó là *phần tử* (*thành phần*) của vector. Khi dùng vector để biểu diễn các mẫu trong tập dữ liệu, giá trị của chúng thường mang ý nghĩa liên quan tới đời thực. Ví dụ, nếu chúng ta huấn luyện một mô hình dự đoán rủi ro vỡ nợ, chúng ta có thể gán cho mỗi ứng viên một vector gồm các thành phần tương ứng với thu nhập, thời gian làm việc, số lần vỡ nợ trước đó của họ và các yếu tố khác. Nếu chúng ta đang tìm hiểu về rủi ro bị đau tim của bệnh nhân, ta có thể biểu diễn mỗi bệnh nhân bằng một vector gồm các phần tử mang thông tin về dấu hiệu sinh tồn gần nhất, nồng độ cholesterol, số phút tập thể dục mỗi ngày, v.v. Trong ký hiệu toán học, chúng ta thường biểu diễn vector bằng chữ cái in đậm viết thường (ví dụ  $\mathbf{x}, \mathbf{y}$ , và  $\mathbf{z}$ ).

Trong MXNet, chúng ta làm việc với vector thông qua các *ndarray* 1-chiều. Thường thì *ndarray* có thể có chiều dài bất kỳ, tùy thuộc vào giới hạn bộ nhớ máy tính.

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

Một phần tử bất kỳ trong vector có thể được ký hiệu sử dụng chỉ số dưới. Ví dụ ta có thể viết  $x_i$  để ám chỉ phần tử thứ  $i$  của  $\mathbf{x}$ . Lưu ý rằng phần tử  $x_i$  là một số vô hướng nên nó không được in đậm. Có rất nhiều tài liệu tham khảo xem vector cột là chiều mặc định của vector, và quyển sách này cũng vậy. Trong toán học, một vector có thể được viết như sau

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad (4.3.1)$$

trong đó  $x_1, \dots, x_n$  là các phần tử của vector. Trong mã nguồn, chúng ta sử dụng chỉ số để truy cập các phần tử trong ndarray.

```
x[3]
```

## Độ dài, Chiều, và Kích thước

Hãy quay lại với những khái niệm từ Section 4.1. Một vector đơn thuần là một dãy các số. Mỗi vector, tương tự như dãy, đều có một độ dài. Trong ký hiệu toán học, nếu ta muốn nói rằng một vector  $\mathbf{x}$  chứa  $n$  các số thực vô hướng, ta có thể biểu diễn nó bằng  $\mathbf{x} \in \mathbb{R}^n$ . Độ dài của một vector còn được gọi là số **chiều** của vector.

Cũng giống như một dãy thông thường trong Python, chúng ta có thể xem độ dài của của một ndarray bằng cách gọi hàm `len()` có sẵn của Python.

```
len(x)
```

Khi một ndarray biểu diễn một vector (với chính xác một trực), ta cũng có thể xem độ dài của nó qua thuộc tính `.shape` (kích thước). Kích thước là một tuple liệt kê độ dài (số chiều) đọc theo mỗi trực của ndarray. Với các ndarray có duy nhất một trực, kích thước của nó chỉ có một phần tử.

```
x.shape
```

Ở đây cần lưu ý rằng, từ “chiều” là một từ đa nghĩa và khi đặt vào nhiều ngữ cảnh thường dễ làm ta bị nhầm lẫn. Để làm rõ, chúng ta dùng số chiều của một *vector* hoặc của một *trục* để chỉ độ dài của nó, tức là số phần tử trong một vector hay một trực. Tuy nhiên, chúng ta sử dụng số chiều của một ndarray để chỉ số trực của ndarray đó. Theo nghĩa này, chiều của một trực của một ndarray là độ dài của trực đó.

### 4.3.3 Ma trận

Giống như vector khái quát số vô hướng từ bậc 0 sang bậc 1, ma trận sẽ khái quát những vector từ bậc 1 sang bậc 2. Ma trận thường được ký hiệu với ký tự hoa và được in đậm (ví dụ: **X**, **Y**, và **Z**); và được biểu diễn bằng các ndarray với 2 trục khi lập trình.

Trong ký hiệu toán học, ta dùng  $\mathbf{A} \in \mathbb{R}^{m \times n}$  để biểu thị một ma trận  $\mathbf{A}$  gồm  $m$  hàng và  $n$  cột các giá trị số thực. Về mặt hình ảnh, ta có thể minh họa bất kỳ ma trận  $\mathbf{A} \in \mathbb{R}^{m \times n}$  như một bảng biểu mà mỗi phần tử  $a_{ij}$  nằm ở dòng thứ  $i$  và cột thứ  $j$  của bảng:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \quad (4.3.2)$$

Với bất kỳ ma trận  $\mathbf{A} \in \mathbb{R}^{m \times n}$  nào, kích thước của ma trận  $\mathbf{A}$  là  $(m, n)$  hay  $m \times n$ . Trong trường hợp đặc biệt, khi một ma trận có số dòng bằng số cột, dạng của nó là một hình vuông; như vậy, nó được gọi là một *ma trận vuông (square matrix)*.

Ta có thể tạo một ma trận  $m \times n$  trong MXNet bằng cách khai báo kích thước của nó với hai thành phần  $m$  và  $n$  khi sử dụng bất kỳ hàm khởi tạo ndarray nào mà ta thích.

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

Ta có thể truy cập phần tử vô hướng  $a_{ij}$  của ma trận  $\mathbf{A}$  trong (4.3.2) bằng cách khai báo chỉ số dòng ( $i$ ) và chỉ số cột ( $j$ ), như là  $[\mathbf{A}]_{ij}$ . Khi những thành phần vô hướng của ma trận  $\mathbf{A}$ , như trong (4.3.2) chưa được đưa ra, ta có thể sử dụng ký tự viết thường của ma trận  $\mathbf{A}$  với các chỉ số ghi dưới,  $a_{ij}$ , để chỉ thành phần  $[\mathbf{A}]_{ij}$ . Nhằm giữ sự đơn giản cho các ký hiệu, dấu phẩy chỉ được thêm vào để phân tách các chỉ số khi cần thiết, như  $a_{2,3j}$  và  $[\mathbf{A}]_{2i-1,3}$ .

Đôi khi, ta muốn hoán đổi các trục. Khi ta hoán đổi các dòng với các cột của ma trận, kết quả có được là *chuyển vị (transpose)* của ma trận đó. Về lý thuyết, chuyển vị của ma trận  $\mathbf{A}$  được ký hiệu là  $\mathbf{A}^\top$  và nếu  $\mathbf{B} = \mathbf{A}^\top$  thì  $b_{ij} = a_{ji}$  với mọi  $i$  và  $j$ . Do đó, chuyển vị của  $\mathbf{A}$  trong (4.3.2) là một ma trận  $n \times m$ :

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}. \quad (4.3.3)$$

Trong mã nguồn, ta lấy chuyển vị của một ma trận thông qua thuộc tính `T`.

```
A.T
```

Là một biến thể đặc biệt của ma trận vuông, *ma trận đối xứng (symmetric matrix)*  $\mathbf{A}$  có chuyển vị bằng chính nó:  $\mathbf{A} = \mathbf{A}^\top$ .

```
B = np.array([[1, 2, 3], [2, 0, 4], [3, 4, 5]])
B
```

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

Ma trận là một cấu trúc dữ liệu hữu ích: chúng cho phép ta tổ chức dữ liệu có nhiều phương thức biến thể khác nhau. Ví dụ, các dòng trong ma trận của chúng ta có thể tương trưng cho các căn nhà khác nhau (các điểm dữ liệu), còn các cột có thể tương trưng cho những thuộc tính khác nhau của ngôi nhà. Bạn có thể thấy quen thuộc với điều này nếu đã từng sử dụng các phần mềm lập bảng tính hoặc đã đọc [Section 4.2](#). Do đó, mặc dù một vector đơn lẻ có hướng mặc định là một vector cột, trong một ma trận biểu thị một tập dữ liệu bằng biểu, sẽ tốt hơn nếu ta xem mỗi điểm dữ liệu như một vector dòng trong ma trận. Chúng ta sẽ thấy ở những chương sau, quy ước này sẽ giúp dễ dàng áp dụng các kỹ thuật học sâu thông dụng. Ví dụ, với trực ngoài cùng của ndarray, ta có thể truy cập hay duyệt qua các batch nhỏ của những điểm dữ liệu hoặc chỉ đơn thuần là các điểm dữ liệu nếu không có batch nhỏ nào cả.

#### 4.3.4 Tensor

Giống như vector khái quát hoá số vô hướng và ma trận khái quát hoá vector, ta có thể xây dựng những cấu trúc dữ liệu với thậm chí nhiều trực hơn. Tensor cho chúng ta một phương pháp tổng quát để miêu tả các ndarray với số trực bất kỳ. Ví dụ, vector là các tensor bậc một còn ma trận là các tensor bậc hai. Tensor được ký hiệu với ký tự viết hoa sử dụng một font chữ đặc biệt (ví dụ: X, Y, và Z) và có cơ chế truy vấn (ví dụ:  $x_{ijk}$  and  $[X]_{1,2i-1,3}$ ) giống như ma trận.

Tensor sẽ trở nên quan trọng hơn khi ta bắt đầu làm việc với hình ảnh, thường được biểu diễn dưới dạng ndarray với 3 trực tương ứng với chiều cao, chiều rộng và một trực *kênh* (*channel*) để xếp chồng các kênh màu (đỏ, xanh lá và xanh dương). Tạm thời, ta sẽ bỏ qua các tensor bậc cao hơn và tập trung vào những điểm cơ bản trước.

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

#### 4.3.5 Các thuộc tính Cơ bản của Phép toán Tensor

Số vô hướng, vector, ma trận và tensor với một số trực bất kỳ có một vài thuộc tính rất hữu dụng. Ví dụ, bạn có thể để ý từ định nghĩa của phép toán theo từng phần tử (*elementwise*), bất kỳ phép toán theo từng phần tử một ngôi nào cũng không làm thay đổi kích thước của toán hạng của nó. Tương tự, cho hai tensor bất kỳ có cùng kích thước, kết quả của bất kỳ phép toán theo từng phần tử hai ngôi sẽ là một tensor có cùng kích thước. Ví dụ, cộng hai ma trận có cùng kích thước sẽ thực hiện phép cộng theo từng phần tử giữa hai ma trận này.

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

Đặc biệt, phép nhân theo phần tử của hai ma trận được gọi là *phép nhân Hadamard* (*Hadamard product* – ký hiệu toán học là  $\odot$ ). Xét ma trận  $\mathbf{B} \in \mathbb{R}^{m \times n}$  có phần tử dòng  $i$  và cột  $j$  là  $b_{ij}$ . Phép nhân Hadamard giữa ma trận  $\mathbf{A}$  (khai báo ở [\(4.3.2\)](#)) và  $\mathbf{B}$  là

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \dots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \dots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \dots & a_{mn}b_{mn} \end{bmatrix}. \quad (4.3.4)$$

```
A * B
```

Nhân hoặc cộng một tensor với một số vô hướng cũng sẽ không thay đổi kích thước của tensor, mỗi phần tử của tensor sẽ được cộng hoặc nhân cho số vô hướng đó.

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

#### 4.3.6 Rút gọn

Một phép toán hữu ích mà ta có thể thực hiện trên bất kỳ tensor nào là phép tính tổng các phần tử của nó. Ký hiệu toán học của phép tính tổng là  $\sum$ . Ta biểu diễn phép tính tổng các phần tử của một vector  $\mathbf{x}$  với độ dài  $d$  dưới dạng  $\sum_{i=1}^d x_i$ . Trong mã nguồn, ta chỉ cần gọi hàm `sum`.

```
x = np.arange(4)
x, x.sum()
```

Ta có thể biểu diễn phép tính tổng các phần tử của tensor có kích thước tùy ý. Ví dụ, tổng các phần tử của một ma trận  $m \times n$  có thể được viết là  $\sum_{i=1}^m \sum_{j=1}^n a_{ij}$ .

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

Theo mặc định, hàm `sum` sẽ *rút gọn* tensor dọc theo tất cả các trục của nó và trả về kết quả là một số vô hướng. Ta cũng có thể chỉ định các trục được rút gọn bằng phép tổng. Hãy ma trận làm ví dụ, để rút gọn theo chiều hàng (trục 0) bằng việc tính tổng tất cả các hàng, ta đặt `axis=0` khi gọi hàm `sum`.

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

Việc đặt `axis=1` sẽ rút gọn theo cột (trục 1) bằng việc tính tổng tất cả các cột. Do đó, kích thước trục 1 của đầu vào sẽ không còn trong kích thước của đầu ra.

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

Việc rút gọn ma trận dọc theo cả hàng và cột bằng phép tổng tương đương với việc cộng tất cả các phần tử trong ma trận đó lại.

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

Một đại lượng liên quan là *trung bình cộng*. Ta tính trung bình cộng bằng cách chia tổng các phần tử cho số lượng phần tử. Trong mã nguồn, ta chỉ cần gọi hàm `mean` với đầu vào là các tensor có kích thước tùy ý.

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

Giống như `sum`, hàm `mean` cũng có thể rút gọn tensor dọc theo các trục được chỉ định.

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

## Tổng không rút gọn

Tuy nhiên, việc giữ lại số các trục đôi khi là cần thiết khi gọi hàm `sum` hoặc `mean`, bằng cách đặt `keepdims=True`.

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

Ví dụ, vì `sum_A` vẫn giữ lại 2 trục sau khi tính tổng của mỗi hàng, chúng ta có thể chia `A` cho `sum_A` thông qua cơ chế lan truyền.

```
A / sum_A
```

Nếu chúng ta muốn tính tổng tích lũy các phần tử của `A` dọc theo các trục, giả sử `axis=0` (từng hàng một), ta có thể gọi hàm `cumsum`. Hàm này không rút gọn chiều của tensor đầu vào theo bất cứ trục nào.

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

### 4.3.7 Tích vô hướng

Cho đến giờ, chúng ta mới chỉ thực hiện những phép tính từng phần tử tương ứng, như tổng và trung bình. Nếu đây là tất những gì chúng ta có thể làm, đại số tuyến tính có lẽ không xứng đáng để có nguyên một mục. Tuy nhiên, một trong những phép tính căn bản nhất của đại số tuyến tính là tích vô hướng. Với hai vector  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$  cho trước, *tích vô hướng* (*dot product*)  $\mathbf{x}^\top \mathbf{y}$  (hoặc  $\langle \mathbf{x}, \mathbf{y} \rangle$ ) là tổng các tích của những phần tử có cùng vị trí:  $\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^d x_i y_i$ .

```
y = np.ones(4)
x, y, np.dot(x, y)
```

Lưu ý rằng chúng ta có thể thể hiện tích vô hướng của hai vector một cách tương tự bằng việc thực hiện tích từng phần tử tương ứng rồi lấy tổng:

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

Tích vô hướng sẽ hữu dụng trong rất nhiều trường hợp. Ví dụ, với một tập các giá trị cho trước, biểu thị bởi vector  $\mathbf{x} \in \mathbb{R}^d$ , và một tập các trọng số được biểu thị bởi  $\mathbf{w} \in \mathbb{R}^d$ , tổng trọng số của các giá trị trong  $\mathbf{x}$  theo các trọng số trong  $\mathbf{w}$  có thể được thể hiện bởi tích vô hướng  $\mathbf{x}^\top \mathbf{w}$ . Khi các trọng số không âm và có tổng bằng một ( $\sum_{i=1}^d w_i = 1$ ), tích vô hướng thể hiện phép tính *trung bình trọng số* (*weighted average*). Sau khi được chuẩn hóa thành hai vector đơn vị, tích vô hướng của hai vector đó là giá trị cos của góc giữa hai vector đó. Chúng tôi sẽ giới thiệu khái niệm về *độ dài* ở các phần sau trong mục này.

### 4.3.8 Tích giữa Ma trận và Vector

Giờ đây, khi đã biết cách tính toán tích vô hướng, chúng ta có thể bắt đầu hiểu *tích giữa ma trận và vector*. Bạn có thể xem lại cách ma trận  $\mathbf{A} \in \mathbb{R}^{m \times n}$  và vector  $\mathbf{x} \in \mathbb{R}^n$  được định nghĩa và biểu diễn trong (4.3.2) và (4.3.1). Ta sẽ bắt đầu bằng việc biểu diễn ma trận  $\mathbf{A}$  qua các vector hàng của nó.

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \quad (4.3.5)$$

Mỗi  $\mathbf{a}_i^\top \in \mathbb{R}^n$  là một vector hàng thể hiện hàng thứ  $i$  của ma trận  $\mathbf{A}$ . Tích giữa ma trận và vector  $\mathbf{Ax}$  đơn giản chỉ là một vector cột với chiều dài  $m$ , với phần tử thứ  $i$  là kết quả của phép tích vô hướng  $\mathbf{a}_i^\top \mathbf{x}$ :

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}. \quad (4.3.6)$$

Chúng ta có thể nghĩ đến việc nhân một ma trận  $\mathbf{A} \in \mathbb{R}^{m \times n}$  với một vector như một phép biến hình, biến đổi vector từ không gian  $\mathbb{R}^n$  thành  $\mathbb{R}^m$ . Những phép biến hình này hóa ra lại trở nên rất hữu dụng. Ví dụ, chúng ta có thể biểu diễn phép xoay là tích với một ma trận vuông. Bạn sẽ thấy ở những chương tiếp theo, chúng ta cũng có thể sử dụng tích giữa ma trận và vector để thực hiện hầu hết những tính toán cần thiết khi tính các tầng trong một mạng nơ-ron dựa theo kết quả của tầng trước đó.

Khi lập trình, để thực hiện nhân ma trận với vector ndarray, chúng ta cũng sử dụng hàm dot giống như tích vô hướng. Việc gọi `np.dot(A, x)` với ma trận  $A$  và một vector  $x$  sẽ thực hiện phép nhân vô hướng giữa ma trận và vector. Lưu ý rằng chiều của cột  $A$  (chiều dài theo trục 1) phải bằng với chiều của vector  $x$  (chiều dài của nó).

```
A.shape, x.shape, np.dot(A, x)
```

### 4.3.9 Phép nhân Ma trận

Nếu bạn đã quen với tích vô hướng và tích ma trận-vector, tích *ma trận-ma trận* cũng tương tự như thế.

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

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}. \quad (4.3.7)$$

Đặt  $\mathbf{a}_i^\top \in \mathbb{R}^k$  là vector hàng biểu diễn hàng thứ  $i$  của ma trận  $\mathbf{A}$  và  $\mathbf{b}_j \in \mathbb{R}^k$  là vector cột thứ  $j$  của ma trận  $\mathbf{B}$ . Để tính ma trận tích  $\mathbf{C} = \mathbf{AB}$ , cách đơn giản nhất là viết các hàng của ma trận  $\mathbf{A}$  và các

cột của ma trận  $\mathbf{B}$ :

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m]. \quad (4.3.8)$$

Khi đó ma trận tích  $\mathbf{C} \in \mathbb{R}^{n \times m}$  được tạo với phần tử  $c_{ij}$  bằng tích vô hướng  $\mathbf{a}_i^\top \mathbf{b}_j$ :

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_m] = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}. \quad (4.3.9)$$

Ta có thể coi tích hai ma trận  $\mathbf{AB}$  như việc tính  $m$  phép nhân ma trận và vector, sau đó ghép các kết quả với nhau để tạo ra một ma trận  $n \times m$ . Giống như tích vô hướng và phép nhân ma trận-vector, ta có thể tính phép nhân hai ma trận bằng cách sử dụng hàm dot. Trong đoạn mã dưới đây, chúng ta tính phép nhân giữa  $\mathbf{A}$  và  $\mathbf{B}$ . Ở đây,  $\mathbf{A}$  là một ma trận với 5 hàng 4 cột và  $\mathbf{B}$  là một ma trận với 4 hàng 3 cột. Sau phép nhân này, ta thu được một ma trận với 5 hàng 3 cột.

```
B = np.ones(shape=(4, 3))
np.dot(A, B)
```

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

### 4.3.10 Chuẩn

Một trong những toán tử hữu dụng nhất của đại số tuyến tính là *chuẩn* (*norm*). Nói dân dã thì, các chuẩn của một vector cho ta biết một vector lớn tầm nào. Thuật ngữ *kích thước* đang xét ở đây không nói tới số chiều không gian mà đúng hơn là về độ lớn của các thành phần.

Trong đại số tuyến tính, chuẩn của một vector là hàm số  $f$  ánh xạ một vector đến một số vô hướng, thỏa mãn các tính chất sau. Cho vector  $\mathbf{x}$  bất kỳ, tính chất đầu tiên phát biểu rằng nếu chúng ta co giãn toàn bộ các phần tử của một vector bằng một hằng số  $\alpha$ , chuẩn của vector đó cũng co giãn theo *giá trị tuyệt đối* của hằng số đó:

$$f(\alpha \mathbf{x}) = |\alpha| f(\mathbf{x}). \quad (4.3.10)$$

Tính chất thứ hai cũng giống như bất đẳng thức tam giác:

$$f(\mathbf{x} + \mathbf{y}) \leq f(\mathbf{x}) + f(\mathbf{y}). \quad (4.3.11)$$

Tính chất thứ ba phát biểu rằng chuẩn phải không âm:

$$f(\mathbf{x}) \geq 0. \quad (4.3.12)$$

Điều này là hợp lý vì trong hầu hết các trường hợp thì *kích thước* nhỏ nhất cho các vật đều bằng 0. Tính chất cuối cùng yêu cầu chuẩn nhỏ nhất thu được khi và chỉ khi toàn bộ thành phần của vector đó bằng 0.

$$\forall i, [\mathbf{x}]_i = 0 \Leftrightarrow f(\mathbf{x}) = 0. \quad (4.3.13)$$

Bạn chắc sẽ để ý là các chuẩn có vẻ giống như một phép đo khoảng cách. Và nếu còn nhớ khái niệm khoảng cách Euclid (định lý Pythagoras) được học ở phổ thông, thì khái niệm không âm và bất đẳng thức tam giác có thể gợi nhắc lại một chút. Thực tế là, khoảng cách Euclid cũng là một chuẩn: cụ thể là  $\ell_2$ . Giả sử rằng các thành phần trong vector  $n$  chiều  $\mathbf{x}$  là  $x_1, \dots, x_n$ . Chuẩn  $\ell_2$  của  $\mathbf{x}$  là căn bậc hai của tổng các bình phương của các thành phần trong vector:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}, \quad (4.3.14)$$

Ở đó, chỉ số dưới 2 thường được lược đi khi viết chuẩn  $\ell_2$ , ví dụ,  $\|\mathbf{x}\|$  cũng tương đương với  $\|\mathbf{x}\|_2$ . Khi lập trình, ta có thể tính chuẩn  $\ell_2$  của một vector bằng cách gọi hàm `linalg.norm`.

```
u = np.array([3, -4])
np.linalg.norm(u)
```

Trong học sâu, chúng ta thường gặp chuẩn  $\ell_2$  bình thường hơn. Bạn cũng sẽ thường xuyên gặp chuẩn  $\ell_1$ , chuẩn được biểu diễn bằng tổng các giá trị tuyệt đối của các thành phần trong vector:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^n |x_i|. \quad (4.3.15)$$

So với chuẩn  $\ell_2$ , nó ít bị ảnh hưởng bởi các giá trị ngoại biên hơn. Để tính chuẩn  $\ell_1$ , chúng ta dùng hàm giá trị tuyệt đối rồi lấy tổng các thành phần.

```
np.abs(u).sum()
```

Cả hai chuẩn  $\ell_2$  và  $\ell_1$  đều là trường hợp riêng của một chuẩn tổng quát hơn, chuẩn  $\ell_p$ :

$$\|\mathbf{x}\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p}. \quad (4.3.16)$$

Tương tự với chuẩn  $\ell_2$  của vector, chuẩn *Frobenius* của một ma trận  $\mathbf{X} \in \mathbb{R}^{m \times n}$  là căn bậc hai của tổng các bình phương của các thành phần trong ma trận:

$$\|\mathbf{X}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n x_{ij}^2}. \quad (4.3.17)$$

Chuẩn Frobenius thỏa mãn tất cả các tính chất của một chuẩn vector. Nó giống chuẩn  $\ell_2$  của một vector nhưng ở dạng của ma trận. Ta dùng hàm `linalg.norm` để tính toán chuẩn Frobenius của ma trận.

```
np.linalg.norm(np.ones((4, 9)))
```

## Chuẩn và Mục tiêu

Tuy không muốn đi quá nhanh nhưng chúng ta có thể xây dựng phần nào trực giác để hiểu tại sao những khái niệm này lại hữu dụng. Trong học sâu, ta thường cố giải các bài toán tối ưu: *cực đại hóa* xác suất xảy ra của dữ liệu quan sát được; *cực tiểu hóa* khoảng cách giữa dự đoán và nhãn gốc. Gán các biểu diễn vector cho các đối tượng (như từ, sản phẩm hay các bài báo) để cực tiểu hóa khoảng cách giữa các đối tượng tương tự nhau và cực đại hóa khoảng cách giữa các đối tượng khác nhau. Mục tiêu, thành phần quan trọng nhất của một thuật toán học sâu (bên cạnh dữ liệu), thường được biểu diễn theo *chuẩn (norm)*.

### 4.3.11 Bàn thêm về Đại số Tuyến tính

Chỉ trong mục này, chúng tôi đã trang bị cho bạn tất cả những kiến thức đại số tuyến tính cần thiết để hiểu một lượng lớn các mô hình học máy hiện đại. Vẫn còn rất nhiều kiến thức đại số tuyến tính, phần lớn đều hữu dụng cho học máy. Một ví dụ là phép phân tích ma trận ra các thành phần, các phép phân tích này có thể tạo ra các cấu trúc thấp chiều trong các tập dữ liệu thực tế. Có cả một nhánh của học máy tập trung vào sử dụng các phép phân tích ma trận và tổng quát chúng lên cho các tensor bậc cao để khám phá cấu trúc trong các tập dữ liệu và giải quyết các bài toán dự đoán. Tuy nhiên, cuốn sách này chỉ tập trung vào học sâu. Và chúng tôi tin rằng bạn sẽ muốn học thêm nhiều về toán một khi đã có thể triển khai được các mô hình học máy hữu dụng cho các tập dữ liệu thực tế. Bởi vậy, trong khi vẫn còn nhiều kiến thức toán cần bàn thêm ở phần sau, chúng tôi sẽ kết thúc mục này ở đây.

Nếu bạn muốn học thêm về đại số tuyến tính, bạn có thể tham khảo `sec_geometry-linear-algebraic-ops` hoặc các nguồn tài liệu xuất sắc tại ([Strang, 1993](#); [Kolter, 2008](#); [Petersen et al., 2008](#)).

### 4.3.12 Tóm tắt

- Số vô hướng, vector, ma trận, và tensor là các đối tượng toán học cơ bản trong đại số tuyến tính.
- Vector là dạng tổng quát của số vô hướng và ma trận là dạng tổng quát của vector.
- Trong cách biểu diễn ndarray, các số vô hướng, vector, ma trận và tensor lần lượt có 0, 1, 2 và một số lượng tùy ý các trục.
- Một tensor có thể thu gọn theo một số trục bằng `sum` và `mean`.
- Phép nhân theo từng phần tử của hai ma trận được gọi là tích Hadamard của chúng. Phép toán này khác với phép nhân ma trận.
- Trong học sâu, chúng ta thường làm việc với các chuẩn như chuẩn  $\ell_1$ , chuẩn  $\ell_2$  và chuẩn Frobenius.
- Chúng ta có thể thực hiện một số lượng lớn các toán tử trên số vô hướng, vector, ma trận và tensor với các hàm của ndarray.

### 4.3.13 Bài tập

1. Chứng minh rằng chuyển vị của một ma trận chuyển vị là chính nó:  $(\mathbf{A}^\top)^\top = \mathbf{A}$ .
2. Cho hai ma trận  $\mathbf{A}$  và  $\mathbf{B}$ , chứng minh rằng tổng của chuyển vị bằng chuyển vị của tổng:  $\mathbf{A}^\top + \mathbf{B}^\top = (\mathbf{A} + \mathbf{B})^\top$ .
3. Cho một ma trận vuông  $\mathbf{A}$ , liệu rằng  $\mathbf{A} + \mathbf{A}^\top$  có luôn đối xứng? Tại sao?
4. Chúng ta đã định nghĩa tensor  $X$  với kích thước  $(2, 3, 4)$  trong mục này. Kết quả của  $\text{len}(X)$  là gì?
5. Cho một tensor  $X$  với kích thước bất kỳ, liệu  $\text{len}(X)$  có luôn tương ứng với độ dài của một trục nhất định của  $X$  hay không? Đó là trục nào?
6. Chạy  $\mathbf{A} / \mathbf{A}.sum(\text{axis}=1)$  và xem điều gì xảy ra. Bạn có phân tích được nguyên nhân không?
7. Khi di chuyển giữa hai điểm ở Manhattan (đường phố hình bàn cờ), khoảng cách tính bằng tọa độ (tức độ dài các đại lộ và phố) mà bạn cần di chuyển là bao nhiêu? Bạn có thể đi theo đường chéo không? (Xem thêm bản đồ Manhattan, New York để trả lời câu hỏi này)
8. Xét một tensor với kích thước  $(2, 3, 4)$ . Kích thước của kết quả sau khi tính tổng theo trục 0, 1 và 2 sẽ như thế nào?
9. Đưa một tensor với 3 trục hoặc hơn vào hàm `linalg.norm` và quan sát kết quả. Hàm này thực hiện việc gì cho các `ndarray` với kích thước bất kỳ?

### 4.3.14 Thảo luận

- Tiếng Anh<sup>59</sup>
- Tiếng Việt<sup>60</sup>

### 4.3.15 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Ngô Thế Anh Khoa
- Nguyễn Lê Quang Nhật
- Vũ Hữu Tiệp
- Mai Sơn Hải
- Phạm Hồng Vinh

<sup>59</sup> <https://discuss.mxnet.io/t/2317>

<sup>60</sup> [https://forum.machinelarningcoban.com/c/d21](https://forum.machinelearningcoban.com/c/d21)

## 4.4 Giải tích

Tìm diện tích của một đa giác vẫn là một bí ẩn cho tới ít nhất 2.500 năm trước, khi người Hy Lạp cổ đại chia đa giác thành các tam giác và cộng diện tích của chúng lại. Để tìm diện tích của các hình cong, như hình tròn, người Hy Lạp cổ đại đặt các đa giác nội tiếp bên trong các hình cong đó. Như trong Fig. 4.4.1, một đa giác nội tiếp với càng nhiều cạnh bằng nhau thì càng xấp xỉ đúng hình tròn. Quy trình này còn được biết đến như *phương pháp vét kiệt*.

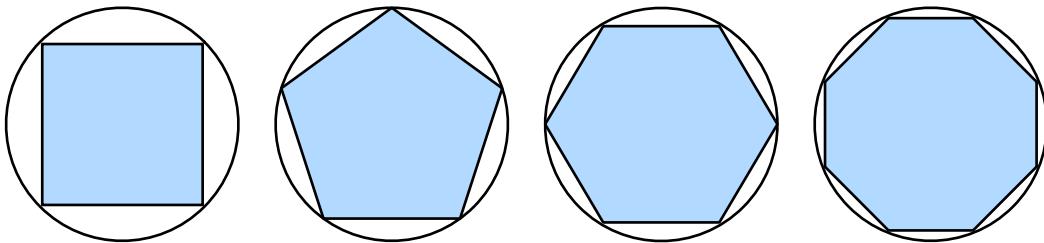


Fig. 4.4.1: Tìm diện tích hình tròn bằng phương pháp vét kiệt.

Phương pháp vét kiệt chính là khởi nguồn của *giải tích tích phân* (sẽ được miêu tả trong Section 20.5). Hơn 2.000 năm sau, nhánh còn lại của giải tích, *giải tích vi phân*, ra đời. Trong những ứng dụng quan trọng nhất của giải tích vi phân, các bài toán tối ưu hoá sẽ tìm *cách tốt nhất* để thực hiện một công việc nào đó. Như đã bàn đến trong Section 4.3.10, các bài toán như vậy vô cùng phổ biến trong học sâu.

Trong học sâu, chúng ta *huấn luyện* các mô hình, cập nhật chúng liên tục để chúng ngày càng tốt hơn khi học với nhiều dữ liệu hơn. Thông thường, trở nên tốt hơn tương đương với cực tiểu hoá một *hàm mất mát*, một điểm số sẽ trả lời câu hỏi “mô hình của ta đang tệ tới mức nào?” Câu hỏi này lắt léo hơn ta tưởng nhiều. Mục đích cuối cùng mà ta muốn là mô hình sẽ hoạt động tốt trên dữ liệu mà nó chưa từng nhìn thấy. Nhưng chúng ta chỉ có thể khớp mô hình trên dữ liệu mà ta đang có thể thấy. Do đó ta có thể chia việc huấn luyện mô hình thành hai vấn đề chính: i) *tối ưu hóa*: quy trình huấn luyện mô hình trên dữ liệu đã thấy. ii) *tổng quát hóa*: dựa trên các nguyên tắc toán học và sự uyên thâm của người huấn luyện để tạo ra các mô hình mà tính hiệu quả của nó vượt ra khỏi tập dữ liệu huấn luyện.

Để giúp bạn hiểu các bài toán tối ưu hóa và các phương pháp tối ưu hóa trong các chương sau, ở đây chúng tôi sẽ cung cấp một chương ngắn vỡ lòng về các kỹ thuật giải tích vi phân thông dụng trong học sâu.

### 4.4.1 Đạo hàm và Ví phân

Chúng ta bắt đầu bằng việc đề cập tới khái niệm *đạo hàm*, một bước quan trọng của hầu hết các thuật toán tối ưu trong học sâu. Trong học sâu, ta thường chọn những hàm mất mát khả vi theo các tham số của mô hình. Nói đơn giản, với mỗi tham số, ta có thể xác định hàm mất mát tăng hoặc giảm nhanh như thế nào khi tham số đó *tăng* hoặc *giảm* chỉ một lượng cực nhỏ.

Giả sử ta có một hàm  $f : \mathbb{R} \rightarrow \mathbb{R}$  có đầu vào và đầu ra đều là số vô hướng. *Đạo hàm* của  $f$  được định nghĩa là

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}, \quad (4.4.1)$$

nếu giới hạn này tồn tại. Nếu  $f'(a)$  tồn tại,  $f$  được gọi là *khả vi (differentiable)* tại  $a$ . Nếu  $f$  khả vi tại mọi điểm trong một khoảng, thì hàm này được gọi là khả vi trong khoảng đó. Ta có thể giải nghĩa đạo hàm  $f'(x)$  trong (4.4.1) như là tốc độ thay đổi *tức thời* của hàm  $f$  theo biến  $x$ . Cái gọi là tốc độ thay đổi tức thời được dựa trên độ biến thiên  $h$  trong  $x$  khi  $h$  tiến về 0.

Để minh họa cho khái niệm đạo hàm, hãy thử với một ví dụ. Định nghĩa  $u = f(x) = 3x^2 - 4x$ .

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

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

Cho  $x = 1$  và  $h$  tiến về 0, kết quả của phương trình  $\frac{f(x+h)-f(x)}{h}$  trong (4.4.1) tiến về 2. Dù thử nghiệm này không phải là một dạng chứng minh toán học, lát nữa ta sẽ thấy rằng quả thật đạo hàm của  $u'$  là 2 khi  $x = 1$ .

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

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

Hãy làm quen với một vài ký hiệu cùng được dùng để biểu diễn đạo hàm. Cho  $y = f(x)$  với  $x$  và  $y$  lần lượt là biến độc lập và biến phụ thuộc của hàm  $f$ . Những biểu diễn sau đây là tương đương nhau:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx}f(x) = Df(x) = D_xf(x), \quad (4.4.2)$$

với ký hiệu  $\frac{d}{dx}$  và  $D$  là các *toán tử vi phân (differentiation operator)* để chỉ các phép toán *vi phân*. Ta có thể sử dụng các quy tắc lấy đạo hàm của các hàm thông dụng sau đây:

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

Để lấy đạo hàm của một hàm được tạo từ vài hàm đơn giản hơn, ví dụ như từ những hàm thông dụng ở trên, có thể dùng các quy tắc hữu dụng dưới đây. Giả sử hàm  $f$  và  $g$  đều khả vi và  $C$  là một hằng số, ta có quy tắc *nhân hằng số*

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x), \quad (4.4.3)$$

*quy tắc tổng*

$$\frac{d}{dx}[f(x) + g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x), \quad (4.4.4)$$

*quy tắc nhân*

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}[g(x)] + g(x)\frac{d}{dx}[f(x)], \quad (4.4.5)$$

và *quy tắc đạo hàm phân thức*

$$\frac{d}{dx}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{d}{dx}[f(x)] - f(x)\frac{d}{dx}[g(x)]}{[g(x)]^2}. \quad (4.4.6)$$

Bây giờ ta có thể áp dụng các quy tắc ở trên để tìm đạo hàm  $u' = f'(x) = 3\frac{d}{dx}x^2 - 4\frac{d}{dx}x = 6x - 4$ . Vậy nên, với  $x = 1$ , ta có  $u' = 2$ : điều này đã được kiểm chứng với thử nghiệm lúc trước ở mục này khi kết quả có được cũng tiến tới 2. Giá trị đạo hàm này cũng đồng thời là độ dốc của đường tiếp tuyến với đường cong  $u = f(x)$  tại  $x = 1$ .

Để minh họa cách hiểu này của đạo hàm, ta sẽ dùng `matplotlib`, một thư viện vẽ biểu đồ thông dụng trong Python. Ta cần định nghĩa một số hàm để cấu hình thuộc tính của các biểu đồ được tạo ra bởi `matplotlib`. Trong đoạn mã sau, hàm `use_svg_display` chỉ định `matplotlib` tạo các biểu đồ ở dạng `svg` để có được chất lượng ảnh sắc nét hơn.

```
# Saved in the d2l package for later use
def use_svg_display():
    """Use the svg format to display a plot in Jupyter."""
    display.set_matplotlib_formats('svg')
```

Ta định nghĩa hàm `set figsize` để chỉ định kích thước của biểu đồ. Lưu ý rằng ở đây ta đang dùng trực tiếp `d2l.plt` do câu lệnh `from matplotlib import pyplot as plt` đã được đánh dấu để lưu vào gói `d2l` trong phần Lời nói đầu.

```
# Saved in the d2l package for later use
def set_figsize(figsize=(3.5, 2.5)):
    """Set the figure size for matplotlib."""
    use_svg_display()
    d2l.plt.rcParams['figure.figsize'] = figsize
```

Hàm `set axes` sau cấu hình thuộc tính của các trục biểu đồ tạo bởi `matplotlib`.

```
# Saved in the d2l package for later use
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 hàm cấu hình biểu đồ trên, ta định nghĩa hàm `plot` để vẽ nhiều đồ thị một cách nhanh chóng vì ta sẽ cần minh họa khá nhiều đồ thị xuyên suốt cuốn sách.

```

# Saved in the d2l package for later use
def plot(X, Y=None, xlabel=None, ylabel=None, legend=[], xlim=None,
         ylim=None, xscale='linear',yscale='linear',
         fmts=['-', '--', '-.', ':'], figsize=(3.5, 2.5), axes=None):
    """Plot data points."""
    d2l.set_figsize(figsize)
    axes = axes if axes else d2l.plt.gca()

    # Return True if X (ndarray 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)

```

Giờ ta có thể vẽ đồ thị của hàm số  $u = f(x)$  và đường tiếp tuyến của nó  $y = 2x - 3$  tại  $x = 1$ , với hệ số 2 là độ dốc của 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)'])

```

#### 4.4.2 Đạo hàm riêng

Cho tới giờ, ta đã làm việc với đạo hàm của các hàm một biến. Trong học sâu, các hàm lại thường phụ thuộc vào *nhiều* biến. Do đó, ta cần mở rộng ý tưởng của đạo hàm cho các hàm *nhiều biến* đó.

Cho  $y = f(x_1, x_2, \dots, x_n)$  là một hàm với  $n$  biến. *Đạo hàm riêng* của  $y$  theo tham số thứ  $i$ ,  $x_i$ , là

$$\frac{\partial y}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \quad (4.4.7)$$

Để tính  $\frac{\partial y}{\partial x_i}$ , ta chỉ cần coi  $x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n$  là các hằng số và tính đạo hàm của  $y$  theo  $x_i$ . Để biểu diễn đạo hàm riêng, các ký hiệu sau đây đều có ý nghĩa tương đương:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f. \quad (4.4.8)$$

### 4.4.3 Gradient

Chúng ta có thể ghép các đạo hàm riêng của mọi biến trong một hàm nhiều biến để thu được vector *gradient* của hàm số đó. Giả sử rằng đầu vào của hàm  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  là một vector  $n$  chiều  $\mathbf{x} = [x_1, x_2, \dots, x_n]^\top$  và đầu ra là một số vô hướng. Gradient của hàm  $f(\mathbf{x})$  theo  $\mathbf{x}$  là một vector gồm  $n$  đạo hàm riêng đó:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left[ \frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n} \right]^\top. \quad (4.4.9)$$

Biểu thức  $\nabla_{\mathbf{x}} f(\mathbf{x})$  thường được viết gọn thành  $\nabla f(\mathbf{x})$  trong trường hợp không sợ nhầm lẫn.

Cho  $\mathbf{x}$  là một vector  $n$ -chiều, các quy tắc sau thường được dùng khi tính vi phân hàm đa biến:

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

Tương tự, với bất kỳ ma trận  $\mathbf{X}$  nào, ta đều có  $\nabla_{\mathbf{x}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$ . Sau này ta sẽ thấy, gradient sẽ rất hữu ích khi thiết kế thuật toán tối ưu trong học sâu.

### 4.4.4 Quy tắc dây chuyền

Tuy nhiên, những gradient như thế có thể khó để tính toán. Đó là bởi vì các hàm nhiều biến trong học sâu đa phần là những *hàm hợp*, nên ta không thể áp dụng các quy tắc đề cập ở trên để lấy vi phân cho những hàm này. May mắn thay, *quy tắc dây chuyền* cho phép chúng ta lấy vi phân của các hàm hợp.

Trước tiên, chúng ta hãy xem xét các hàm một biến. Giả sử hai hàm  $y = f(u)$  và  $u = g(x)$  đều khả vi, quy tắc dây chuyền được mô tả như sau

$$\frac{dy}{dx} = \frac{dy}{du} \frac{du}{dx}. \quad (4.4.10)$$

Giờ ta sẽ xét trường hợp tổng quát hơn đối với các hàm nhiều biến. Giả sử một hàm khả vi  $y$  có các biến số  $u_1, u_2, \dots, u_m$ , trong đó mỗi biến  $u_i$  là một hàm khả vi của các biến  $x_1, x_2, \dots, x_n$ . Lưu ý rằng  $y$  cũng là hàm của các biến  $x_1, x_2, \dots, x_n$ . Quy tắc dây chuyền cho ta

$$\frac{dy}{dx_i} = \frac{dy}{du_1} \frac{du_1}{dx_i} + \frac{dy}{du_2} \frac{du_2}{dx_i} + \dots + \frac{dy}{du_m} \frac{du_m}{dx_i} \quad (4.4.11)$$

cho mỗi  $i = 1, 2, \dots, n$ .

#### 4.4.5 Tóm tắt

- Vi phân và tích phân là hai nhánh con của giải tích, trong đó vi phân được ứng dụng rộng rãi trong các bài toán tối ưu hóa của học sâu.
- Đạo hàm có thể được hiểu như là tốc độ thay đổi tức thì của một hàm số đối với các biến số. Nó cũng là độ dốc của đường tiếp tuyến với đường cong của hàm.
- Gradient là một vector có các phần tử là đạo hàm riêng của một hàm nhiều biến theo tất cả các biến số của nó.
- Quy tắc dây chuyền cho phép chúng ta lấy vi phân của các hàm hợp.

#### 4.4.6 Bài tập

dịch đoạn phía trên 1. Vẽ đồ thị của hàm số  $y = f(x) = x^3 - \frac{1}{x}$  và đường tiếp tuyến của nó tại  $x = 1$ . 1. Tìm gradient của hàm số  $f(\mathbf{x}) = 3x_1^2 + 5e^{x_2}$ . 1. Gradient của hàm  $f(\mathbf{x}) = \|\mathbf{x}\|_2$  là gì? 1. Có thể dùng quy tắc dây chuyền cho trường hợp sau đây không:  $u = f(x, y, z)$ , với  $x = x(a, b)$ ,  $y = y(a, b)$  và  $z = z(a, b)$ ?

#### 4.4.7 Thảo luận

- Tiếng Anh<sup>61</sup>
- Tiếng Việt<sup>62</sup>

#### 4.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Nguyễn Cảnh Thưởng
- Phạm Minh Đức
- Tạ H. Duy Nguyên

<sup>61</sup> <https://discuss.mxnet.io/t/5008>

<sup>62</sup> <https://forum.machinelearningcoban.com/c/d21>

## 4.5 Tính vi phân Tự động

Như đã giải thích trong Section 4.4, vi phân là phép tính thiết yếu trong hầu như tất cả mọi thuật toán học sâu. Mặc dù các phép toán trong việc tính đạo hàm khá trực quan và chỉ yêu cầu một chút kiến thức giải tích, nhưng với các mô hình phức tạp, việc tự tính rõ ràng từng bước khá là mệt (và thường rất dễ sai).

Gói thư viện autograd giải quyết vấn đề này một cách nhanh chóng và hiệu quả bằng cách tự động hoá các phép tính đạo hàm (*automatic differentiation*). Trong khi nhiều thư viện yêu cầu ta phải biên dịch một *đồ thị biểu tượng* (*symbolic graph*) để có thể tự động tính đạo hàm, autograd cho phép ta tính đạo hàm ngay lập tức thông qua các dòng lệnh thông thường. Mỗi khi đưa dữ liệu chạy qua mô hình, autograd xây dựng một đồ thị và theo dõi xem dữ liệu nào kết hợp với các phép tính nào để tạo ra kết quả. Với đồ thị này autograd sau đó có thể lan truyền ngược gradient lại theo ý muốn. *Lan truyền ngược* ở đây chỉ đơn thuần là truy ngược lại *đồ thị tính toán* và điền vào đó các giá trị đạo hàm riêng theo từng tham số.

```
from mxnet import autograd, np, npx  
npx.set_np()
```

### 4.5.1 Một ví dụ đơn giản

Lấy ví dụ đơn giản, giả sử chúng ta muốn tính vi phân của hàm số  $y = 2\mathbf{x}^\top \mathbf{x}$  theo vector cột  $\mathbf{x}$ . Để bắt đầu, ta sẽ tạo biến  $x$  và gán cho nó một giá trị ban đầu.

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

Lưu ý rằng trước khi có thể tính gradient của  $y$  theo  $\mathbf{x}$ , chúng ta cần một nơi để lưu giữ nó. Điều quan trọng là ta không được cấp phát thêm bộ nhớ mới mỗi khi tính đạo hàm theo một biến xác định, vì ta thường cập nhật cùng một tham số hàng ngàn vạn lần và sẽ nhanh chóng dùng hết bộ nhớ.

Cũng lưu ý rằng, bản thân giá trị gradient của hàm số đơn trị theo một vector  $\mathbf{x}$  cũng là một vector với cùng kích thước. Do vậy trong mã nguồn sẽ trực quan hơn nếu chúng ta lưu giá trị gradient tính theo  $x$  dưới dạng một thuộc tính của chính ndarray  $x$ . Chúng ta cấp bộ nhớ cho gradient của một ndarray bằng cách gọi phương thức `attach_grad`.

```
x.attach_grad()
```

Sau khi đã tính toán gradient theo biến  $x$ , ta có thể truy cập nó thông qua thuộc tính `grad`. Để an toàn,  $x.grad$  được khởi tạo là một mảng chứa các giá trị không. Điều này hợp lý vì trong học sâu, việc lấy gradient thường là để cập nhật các tham số bằng cách cộng (hoặc trừ) gradient của một hàm để cực đại (hoặc cực tiểu) hóa hàm đó. Bằng cách khởi tạo gradient bằng mảng chứa giá trị không, ta đảm bảo rằng bất kỳ cập nhật vô tình nào trước khi gradient được tính toán sẽ không làm thay đổi giá trị các tham số.

```
x.grad
```

Giờ hãy tính  $y$ . Bởi vì mục đích sau cùng là tính gradient, ta muốn MXNet tạo đồ thị tính toán một cách nhanh chóng. Ta có thể tưởng tượng rằng MXNet sẽ bật một thiết bị ghi hình để thu lại chính xác đường đi mà mỗi biến được tạo.

Chú ý rằng ta cần một số lượng phép tính không hề nhỏ để xây dựng đồ thị tính toán. Vậy nên MXNet sẽ chỉ dựng đồ thị khi được ra lệnh rõ ràng. Ta có thể thực hiện việc này bằng cách đặt đoạn mã trong phạm vi autograd.record.

```
with autograd.record():
    y = 2 * np.dot(x, x)
y
```

Bởi vì  $x$  là một ndarray có độ dài bằng 4, `np.dot` sẽ tính toán tích vô hướng của  $x$  và  $x$ , trả về một số vô hướng mà sẽ được gán cho  $y$ . Tiếp theo, ta có thể tính toán gradient của  $y$  theo mỗi thành phần của  $x$  một cách tự động bằng cách gọi hàm backward của  $y$ .

```
y.backward()
```

Nếu kiểm tra lại giá trị của  $x.grad$ , ta sẽ thấy nó đã được ghi đè bằng gradient mới được tính toán.

```
x.grad
```

Gradient của hàm  $y = 2\mathbf{x}^\top \mathbf{x}$  theo  $\mathbf{x}$  phải là  $4\mathbf{x}$ . Hãy kiểm tra một cách nhanh chóng rằng giá trị gradient mong muốn được tính toán đúng. Nếu hai ndarray là giống nhau, thì mọi cặp phần tử tương ứng cũng bằng nhau.

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

Nếu ta tiếp tục tính gradient của một biến khác mà giá trị của nó là kết quả của một hàm theo biến  $x$ , thì nội dung trong  $x.grad$  sẽ bị ghi đè.

```
with autograd.record():
    y = x.sum()
y.backward()
x.grad
```

### 4.5.2 Truyền ngược cho các biến không phải Số vô hướng

Về mặt kỹ thuật, khi  $y$  không phải một số vô hướng, cách diễn giải tự nhiên nhất cho vi phân của một vector  $y$  theo vector  $x$  đó là một ma trận. Với các bậc và chiều cao hơn của  $y$  và  $x$ , kết quả của phép vi phân có thể là một tensor bậc cao.

Tuy nhiên, trong khi những đối tượng như trên xuất hiện trong học máy nâng cao (bao gồm học sâu), thường thì khi ta gọi lan truyền ngược trên một vector, ta đang cố tính toán đạo hàm của hàm mất mát theo mỗi batch bao gồm một vài mẫu huấn luyện. Ở đây, ý định của ta không phải là tính toán ma trận vi phân mà là tổng của các đạo hàm riêng được tính toán một cách độc lập cho mỗi mẫu trong batch.

Vậy nên khi ta gọi `backward` lên một biến vector  $y$  – là một hàm của  $x$ , MXNet sẽ cho rằng ta muốn tính tổng của các gradient. Nói ngắn gọn, MXNet sẽ tạo một biến mới có giá trị là số vô hướng bằng cách cộng lại các phần tử trong  $y$  và tính gradient theo  $x$  của biến mới này.

```
with autograd.record():
    y = x * x # y is a vector
    y.backward()
```

(continues on next page)

```

u = x.copy()
u.attach_grad()
with autograd.record():
    v = (u * u).sum() # v is a scalar
v.backward()

x.grad == u.grad

```

### 4.5.3 Tách rời Tính toán

Đôi khi chúng ta muốn chuyển một số phép tính ra khỏi đồ thị tính toán. Ví dụ, giả sử  $y$  đã được tính như một hàm của  $x$ , rồi sau đó  $z$  được tính như một hàm của cả  $y$  và  $x$ . Bây giờ, giả sử ta muốn tính gradient của  $z$  theo  $x$ , nhưng vì lý do nào đó ta lại muốn xem  $y$  như là một hằng số và chỉ xét đến vai trò của  $x$  như là biến số của  $z$  sau khi giá trị của  $y$  đã được tính.

Trong trường hợp này, ta có thể gọi  $u = y.detach()$  để trả về một biến  $u$  mới có cùng giá trị như  $y$  nhưng không còn chứa các thông tin về cách mà  $y$  đã được tính trong đồ thị tính toán. Nói cách khác, gradient sẽ không thể chảy ngược qua  $u$  về  $x$  được. Bằng cách này, ta đã tính  $u$  như một hàm của  $x$  ở ngoài phạm vi của `autograd.record`, dẫn đến việc biến  $u$  sẽ được xem như là một hằng số mỗi khi ta gọi `backward`. Chính vì vậy, hàm `backward` sau đây sẽ tính đạo hàm riêng của  $z = u * x$  theo  $x$  khi xem  $u$  như là một hằng số, thay vì đạo hàm riêng của  $z = x * x * x$  theo  $x$ .

```

with autograd.record():
    y = x * x
    u = y.detach()
    z = u * x
z.backward()
x.grad == u

```

Bởi vì sự tính toán của  $y$  đã được ghi lại, chúng ta có thể gọi `y.backward()` sau đó để lấy đạo hàm của  $y = x * x$  theo  $x$ , tức là  $2 * x$ .

```

y.backward()
x.grad == 2 * x

```

Lưu ý rằng khi ta gắn gradient vào một biến  $x$ ,  $x = x.detach()$  sẽ được gọi ngầm. Nếu  $x$  được tính dựa trên các biến khác, phần tính toán này sẽ không được sử dụng trong hàm `backward`.

```

y = np.ones(4) * 2
y.attach_grad()
with autograd.record():
    u = x * y
    u.attach_grad() # Implicitly run u = u.detach()
    z = 5 * u - x
z.backward()
x.grad, u.grad, y.grad

```

#### 4.5.4 Tính gradient của Luồng điều khiển Python

Một lợi thế của việc sử dụng vi phân tự động là khi việc xây dựng đồ thị tính toán đòi hỏi trải qua một loạt các câu lệnh điều khiển luồng Python, (ví dụ như câu lệnh điều kiện, vòng lặp và các lệnh gọi hàm tùy ý), ta vẫn có thể tính gradient của biến kết quả. Trong đoạn mã sau, hãy lưu ý rằng số lần lặp của vòng lặp while và kết quả của câu lệnh if đều phụ thuộc vào giá trị của đầu vào a.

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

Một lần nữa, để tính gradient ta chỉ cần “ghi lại” các phép tính (bằng cách gọi hàm record) và sau đó gọi hàm backward.

```
a = np.random.normal()
a.attach_grad()
with autograd.record():
    d = f(a)
d.backward()
```

Giờ ta có thể phân tích hàm f được định nghĩa ở phía trên. Hãy để ý rằng hàm này tuyến tính từng khúc theo đầu vào a. Nói cách khác, với mọi giá trị của a tồn tại một hằng số k sao cho  $f(a) = k * a$ , ở đó giá trị của k phụ thuộc vào đầu vào a. Do đó, ta có thể kiểm tra giá trị của gradient bằng cách tính  $d / a$ .

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

#### 4.5.5 Chế độ huấn luyện và Chế độ dự đoán

Như đã thấy, sau khi gọi autograd.record, MXNet sẽ ghi lại những tính toán xảy ra trong khối mã nguồn theo sau. Có một chi tiết tinh tế nữa mà ta cần để ý. autograd.record sẽ thay đổi chế độ chạy từ *chế độ dự đoán* sang *chế độ huấn luyện*. Ta có thể kiểm chứng hành vi này bằng cách gọi hàm is\_training.

```
print(autograd.is_training())
with autograd.record():
    print(autograd.is_training())
```

Khi ta tìm hiểu tới các mô hình học sâu phức tạp, ta sẽ gặp một vài thuật toán mà mô hình hoạt động khác nhau khi huấn luyện và khi được sử dụng sau đó để dự đoán. Những khác biệt này sẽ được đề cập chi tiết trong các chương sau.

#### 4.5.6 Tóm tắt

- MXNet cung cấp gói autograd để tự động hóa việc tính toán đạo hàm. Để sử dụng nó, đầu tiên ta gắn gradient cho các biến mà ta muốn lấy đạo hàm riêng theo nó. Sau đó ta ghi lại tính toán của giá trị mục tiêu, thực thi hàm backward của nó và truy cập kết quả gradient thông qua thuộc tính grad của các biến.
- Ta có thể tách rời gradient để kiểm soát những phần tính toán được sử dụng trong hàm backward.
- Các chế độ chạy của MXNet bao gồm chế độ huấn luyện và chế độ dự đoán. Ta có thể kiểm tra chế độ đang chạy bằng cách gọi hàm `is_training`.

#### 4.5.7 Bài tập

1. Tại sao đạo hàm bậc hai lại mất thêm rất nhiều tài nguyên để tính toán hơn đạo hàm bậc một?
2. Sau khi chạy `y.backward()`, lập tức chạy lại lần nữa và xem chuyện gì sẽ xảy ra.
3. Trong ví dụ về luồng điều khiển khi ta tính toán đạo hàm của `d` theo `a`, điều gì sẽ xảy ra nếu ta thay đổi biến `a` thành một vector hay ma trận ngẫu nhiên. Lúc này, kết quả của tính toán `f(a)` sẽ không còn là số vô hướng nữa. Điều gì sẽ xảy ra với kết quả? Ta có thể phân tích nó như thế nào?
4. Hãy tái thiết kế một ví dụ về việc tìm gradient của luồng điều khiển. Chạy ví dụ và phân tích kết quả.
5. Cho  $f(x) = \sin(x)$ . Vẽ đồ thị của  $f(x)$  và  $\frac{df(x)}{dx}$  với điều kiện không được tính trực tiếp đạo hàm  $f'(x) = \cos(x)$ .
6. Trong một cuộc đấu giá kín theo giá thứ hai (ví dụ như trong eBay hay trong quảng cáo điện toán), người thắng cuộc đấu giá chỉ trả mức giá cao thứ hai. Hãy tính gradient của mức giá cuối cùng theo mức đặt của người thắng cuộc bằng cách sử dụng autograd. Kết quả cho bạn biết điều gì về cơ chế đấu giá này? Nếu bạn tò mò muốn tìm hiểu thêm về các cuộc đấu giá kín theo giá thứ hai, hãy đọc bài báo nghiên cứu của Edelman et al. (Edelman et al., 2007).

#### 4.5.8 Thảo luận

- [Tiếng Anh](#)<sup>63</sup>
- [Tiếng Việt](#)<sup>64</sup>

<sup>63</sup> <https://discuss.mxnet.io/t/2318>

<sup>64</sup> <https://forum.machinelearningcoban.com/c/d21>

#### 4.5.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Tạ H. Duy Nguyên
- Phạm Minh Đức

## 4.6 Xác suất

Theo cách này hay cách khác, học máy đơn thuần là đưa ra các dự đoán. Chúng ta có thể muốn dự đoán *xác suất* của một bệnh nhân có thể bị đau tim vào năm sau, khi đã biết tiền sử lâm sàng của họ. Trong tác vụ phát hiện điều bất thường, chúng ta có thể muốn đánh giá *khả năng* các thông số động cơ máy bay ở mức nào, liệu có ở mức hoạt động bình thường không. Trong học tăng cường, chúng ta muốn có một tác nhân hoạt động thông minh trong một môi trường. Nghĩa là chúng ta cần tính tới xác suất đạt điểm thường cao nhất cho từng hành động có thể thực hiện. Và khi xây dựng một hệ thống gợi ý chúng ta cũng cần quan tâm tới xác suất. Ví dụ, giả thiết rằng chúng ta làm việc cho một hãng bán sách trực tuyến lớn. Chúng ta có thể muốn ước lượng xác suất một khách hàng cụ thể muốn mua một cuốn sách cụ thể nào đó. Để làm được điều này, chúng ta cần dùng tới ngôn ngữ xác suất. Có những khóa học, chuyên ngành, luận văn, sự nghiệp, và cả các ban ngành đều dành toàn bộ cho xác suất. Vì thế đương nhiên mục tiêu của chúng tôi trong chương này không phải để dạy toàn bộ môn xác suất. Thay vào đó, chúng tôi hy vọng đưa tới cho bạn đọc các kiến thức nền tảng, đủ để bạn đọc có thể bắt đầu xây dựng mô hình học sâu đầu tiên của chính mình, và truyền cảm hứng cho bạn thêm yêu thích xác suất để có thể bắt đầu tự khám phá nếu muốn.

Chúng tôi đã nhắc tới xác suất trong các chương trước mà không nói rõ chính xác nó là gì hay là đưa ra một ví dụ cụ thể nào. Giờ hãy cùng bắt đầu nghiêm túc hơn bằng cách xem xét trường hợp đầu tiên: phân biệt mèo và chó dựa trên các bức ảnh. Điều này tưởng chừng đơn giản nhưng thực ra là một thách thức. Để bắt đầu, độ phức tạp của bài toán này có thể phụ thuộc vào độ phân giải của ảnh.



Fig. 4.6.1: Ảnh ở các độ phân giải khác nhau ( $10 \times 10$ ,  $20 \times 20$ ,  $40 \times 40$ ,  $80 \times 80$ , and  $160 \times 160$  điểm ảnh).

Như thể hiện trong Fig. 4.6.1, con người phân biệt mèo và chó dễ dàng ở độ phân giải  $160 \times 160$  điểm ảnh, có chút thử thách hơn ở  $40 \times 40$  điểm ảnh, và gần như không thể ở  $10 \times 10$  điểm ảnh. Nói cách khác, khả năng phân biệt mèo và chó của chúng ta ở khoảng cách càng xa (đồng nghĩa với độ phân giải thấp) càng giống đoán mò. Xác suất trang bị cho ta một cách suy luận hình thức về mức độ chắc chắn. Nếu chúng ta hoàn toàn chắc chắn rằng bức ảnh mô tả một con mèo, ta có thể nói rằng xác suất nhãn tương ứng  $y$  là “mèo”, ký hiệu là  $P(y = \text{“mèo”})$  equals 1. Nếu chúng ta không có manh mối nào để đoán rằng  $y = \text{“mèo”}$  hoặc là  $y = \text{“chó”}$ , thì ta có thể nói rằng hai xác suất này có khả năng bằng nhau, biểu diễn bởi  $P(y = \text{“mèo”}) = P(y = \text{“chó”}) = 0.5$ . Nếu ta khá tự tin, nhưng không thực sự chắc chắn bức ảnh mô tả một con mèo, ta có thể gán cho nó một xác suất  $0.5 < P(y = \text{“mèo”}) < 1$ .

Giờ hãy xem xét trường hợp thứ hai: cho dữ liệu theo dõi khí tượng, chúng ta muốn dự đoán xác suất ngày mai trời sẽ mưa ở Đài Bắc. Nếu vào mùa hè, xác suất trời mưa có thể là 0.5.

Trong cả hai trường hợp, chúng ta đều quan tâm tới một đại lượng nào đó và cùng không chắc chắn về giá trị đầu ra. Nhưng có một khác biệt quan trọng giữa hai trường hợp. Trong trường hợp đầu tiên, bức ảnh chỉ có thể là chó hoặc mèo, và chúng ta chỉ không biết là loài nào. Trong trường hợp thứ hai, đầu ra thực sự có thể là một sự kiện ngẫu nhiên, nếu bạn tin vào những thứ như vậy (và hầu hết các nhà vật lý tin vậy). Như vậy xác suất là một ngôn ngữ linh hoạt để suy đoán về mức độ chắc chắn của chúng ta, và nó có thể được áp dụng hiệu quả trong vô vàn ngữ cảnh khác nhau.

#### 4.6.1 Lý thuyết Xác suất cơ bản

Giả sử, ta tung xúc xắc và muốn biết cơ hội để thấy mặt số 1 so với các mặt khác là bao nhiêu? Nếu chiếc xúc xắc có chất liệu đồng nhất, thì cả 6 mặt  $\{1, \dots, 6\}$  đều có khả năng xuất hiện như nhau, nên ta sẽ thấy mặt 1 xuất hiện một lần trong mỗi sáu lần tung xúc xắc như trên. Ta có thể nói rằng mặt 1 xuất hiện với xác suất là  $\frac{1}{6}$ .

Với một chiếc xúc xắc thật, ta có thể không biết được tỷ lệ này và cần kiểm tra liệu xúc xắc có bị hư hỏng gì không. Cách duy nhất để kiểm tra là tung thật nhiều lần rồi ghi lại kết quả. Mỗi lần tung,

ta quan sát thấy một số trong  $\{1, \dots, 6\}$  xuất hiện. Với kết quả này, ta muốn kiểm chứng xác suất xuất hiện của từng mặt số.

Cách tính trực quan nhất là lấy số lần xuất hiện của mỗi mặt số chia cho tổng số lần tung. Cách này cho ta một *ước lượng* của xác suất ứng với một *sự kiện* cho trước. *Luật số lớn* cho ta biết rằng số lần tung xúc xắc càng tăng thì ước lượng này càng gần hơn với xác suất thực. Trước khi giải thích chi tiết hơn, hãy cùng lập trình thí nghiệm này.

Bắt đầu, ta nhập các gói lệnh cần thiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()
```

Tiếp theo, ta sẽ cần tung xúc xắc. Trong thống kê, ta gọi quá trình thu các mẫu từ phân phối xác suất là quá trình *lấy mẫu*. Phân phối mà gán các xác suất cho các lựa chọn rời rạc (*discrete choices*) được gọi là *phân phối đa thức* (*multinomial distribution*). Sau này, ta sẽ đưa ra định nghĩa chính quy *phân phối* là gì; nhưng để hình dung, hãy xem nó như phép gán xác suất xảy ra cho các sự kiện. Trong MXNet, ta có thể lấy mẫu từ phân phối đa thức với hàm `np.random.multinomial`. Có nhiều cách sử dụng hàm này, nhưng ta tập trung vào cách dùng đơn giản nhất. Muốn lấy một mẫu đơn, ta chỉ cần đưa vào hàm này một vector chứa các xác suất. Hàm `np.random.multinomial` sẽ cho kết quả là một vector có chiều dài tương tự: trong vector này, giá trị tại chỉ số  $i$  là số lần kết quả  $i$  xuất hiện.

```
fair_probs = [1.0 / 6] * 6
np.random.multinomial(1, fair_probs)
```

Nếu chạy hàm lấy mẫu vài lần, bạn sẽ thấy rằng mỗi lần các giá trị trả về đều là ngẫu nhiên. Giống với việc đánh giá một con xúc xắc có đều hay không, chúng ta thường muốn tạo nhiều mẫu từ cùng một phân phối. Tạo dữ liệu như trên với vòng lặp `for` trong Python là rất chậm, vì vậy hàm `random.multinomial` hỗ trợ sinh nhiều mẫu trong một lần gọi, trả về một mảng chứa các mẫu độc lập với kích thước bất kỳ.

```
np.random.multinomial(10, fair_probs)
```

Chúng ta cũng có thể giả sử làm 3 thí nghiệm, trong đó mỗi thí nghiệm cùng lúc lấy ra 10 mẫu.

```
counts = np.random.multinomial(10, fair_probs, size=3)
counts
```

Giờ chúng ta đã biết cách lấy mẫu các lần tung của một con xúc xắc, ta có thể giả lập 1000 lần tung. Sau đó, chúng ta có thể đếm xem mỗi mặt xuất hiện bao nhiêu lần. Cụ thể, chúng ta tính toán tần suất tương đối như là một ước lượng của xác suất thực.

```
# Store the results as 32-bit floats for division
counts = np.random.multinomial(1000, fair_probs).astype(np.float32)
counts / 1000 # Reletive frequency as the estimate
```

Do dữ liệu được sinh bởi một con xúc xắc đều, ta biết mỗi đầu ra đều có xác suất thực bằng  $\frac{1}{6}$ , cỡ 0.167, do đó kết quả ước lượng bên trên trông khá ổn.

Chúng ta cũng có thể minh họa những xác suất này hội tụ tới xác suất thực như thế nào. Hãy cũng làm 500 thí nghiệm trong đó mỗi thí nghiệm lấy ra 10 mẫu.

```
counts = np.random.multinomial(10, fair_probs, size=500)
cum_counts = counts.astype(np.float32).cumsum(axis=0)
estimates = cum_counts / cum_counts.sum(axis=1, keepdims=True)

d2l.set_figsize((6, 4.5))
for i in range(6):
    d2l.plt.plot(estimates[:, i].asnumpy(),
                  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 liền tương ứng với một trong sáu giá trị của xúc xắc và chỉ ra xác suất ước lượng của sự kiện xúc xắc ra mặt tương ứng sau mỗi thí nghiệm. Đường đứt đoạn màu đen tương ứng với xác suất thực. Khi ta lấy thêm dữ liệu bằng cách thực hiện thêm các thí nghiệm, thì 6 đường cong liền sẽ hội tụ tiến tới xác suất thực.

## Các Tiên đề của Lý thuyết Xác suất

Khi thực hiện tung một con xúc xắc, chúng ta gọi tập hợp  $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$  là *không gian mẫu* hoặc *không gian kết quả*, trong đó mỗi phần tử là một *kết quả*. Một *sự kiện* là một tập hợp các kết quả của không gian mẫu. Ví dụ, “tung được một số 5” ( $\{5\}$ ) và “tung được một số lẻ” ( $\{1, 3, 5\}$ ) đều là những sự kiện hợp lệ khi tung một con xúc xắc. Chú ý rằng nếu kết quả của một phép tung ngẫu nhiên nằm trong sự kiện  $\mathcal{A}$ , sự kiện  $\mathcal{A}$  đã xảy ra. Như vậy, nếu mặt 3 chấm ngửa lên sau khi xúc xắc được tung, chúng ta nói sự kiện “tung được một số lẻ” đã xảy ra bởi vì  $3 \in \{1, 3, 5\}$ .

Một cách chính thống hơn, *xác suất* có thể được xem là một hàm số ánh xạ một tập hợp các *sự kiện* tới một số thực. Xác suất của sự kiện  $\mathcal{A}$  trong không gian mẫu  $\mathcal{S}$ , được kí hiệu là  $P(\mathcal{A})$ , phải thoả mãn những tính chất sau:

- Với mọi sự kiện  $\mathcal{A}$ , xác suất của nó là không âm, tức là:  $P(\mathcal{A}) \geq 0$ ;
- Xác suất của toàn không gian mẫu luôn bằng 1, tức:  $P(\mathcal{S}) = 1$ ;
- Đối với mọi dãy sự kiện có thể đếm được  $\mathcal{A}_1, \mathcal{A}_2, \dots$  xung khắc lẫn nhau ( $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$  với mọi  $i \neq j$ ), xác suất có ít nhất một sự kiện xảy ra sẽ là tổng của những giá trị xác suất riêng lẻ, hay:  $P(\bigcup_{i=1}^{\infty} \mathcal{A}_i) = \sum_{i=1}^{\infty} P(\mathcal{A}_i)$ .

Đây cũng là những tiên đề của lý thuyết xác suất, được đề xuất bởi Kolmogorov năm 1933. Nhờ vào hệ thống tiên đề này, ta có thể tránh được những tranh luận chủ quan về sự ngẫu nhiên; và ta có thể có được những suy luận chặt chẽ sử dụng ngôn ngữ toán học. Ví dụ, cho sự kiện  $\mathcal{A}_1$  là toàn bộ không gian mẫu và  $\mathcal{A}_i = \emptyset$  với mọi  $i > 1$ , chúng ta có thể chứng minh rằng  $P(\emptyset) = 0$ , nghĩa là xác suất của sự kiện không thể xảy ra bằng 0.

## Biến ngẫu nhiên

Trong thí nghiệm tung xúc xắc ngẫu nhiên, chúng ta đã giới thiệu khái niệm của một *biến ngẫu nhiên*. Một biến ngẫu nhiên có thể dùng để biểu diễn cho hầu như bất kỳ đại lượng nào và giá trị của nó không cố định. Nó có thể nhận một giá trị trong tập các giá trị khả dĩ từ một thí nghiệm ngẫu nhiên. Hãy xét một biến ngẫu nhiên  $X$  có thể nhận một trong những giá trị từ tập không gian mẫu  $S = \{1, 2, 3, 4, 5, 6\}$  của thí nghiệm tung xúc xắc. Chúng ta có thể biểu diễn sự kiện “trông thấy mặt 5” là  $\{X = 5\}$  hoặc  $X = 5$ , và xác suất của nó là  $P(\{X = 5\})$  hoặc  $P(X = 5)$ . Khi viết  $P(X = a)$ , chúng ta đã phân biệt giữa biến ngẫu nhiên  $X$  và các giá trị (ví dụ như  $a$ ) mà  $X$  có thể nhận. Tuy nhiên, ký hiệu như vậy khá là rườm rà. Để đơn giản hóa ký hiệu, một mặt, chúng ta có thể chỉ cần dùng  $P(X)$  để biểu diễn *phân phối* của biến ngẫu nhiên  $X$ : phân phối này cho chúng ta biết xác xuất mà  $X$  có thể nhận cho bất kỳ giá trị nào. Mặt khác, chúng ta có thể đơn thuần viết  $P(a)$  để biểu diễn xác suất mà một biến ngẫu nhiên nhận giá trị  $a$ . Bởi vì một sự kiện trong lý thuyết xác suất là một tập các kết quả từ không gian mẫu, chúng ta có thể xác định rõ một khoảng các giá trị mà một biến ngẫu nhiên có thể nhận. Ví dụ,  $P(1 \leq X \leq 3)$  diễn tả xác suất của sự kiện  $\{1 \leq X \leq 3\}$ , nghĩa là  $\{X = 1, 2, \text{ hoặc }, 3\}$ . Tương tự,  $P(1 \leq X \leq 3)$  biểu diễn xác suất mà biến ngẫu nhiên  $X$  có thể nhận giá trị trong tập  $\{1, 2, 3\}$ .

Lưu ý rằng có một sự khác biệt tinh tế giữa các biến ngẫu nhiên *rời rạc*, ví dụ như các mặt của xúc xắc, và các biến ngẫu nhiên *liên tục*, ví dụ như cân nặng và chiều cao của một người. Việc hỏi rằng hai người có cùng chính xác chiều cao hay không khá là vô nghĩa. Nếu ta đo với đủ độ chính xác, ta sẽ thấy rằng không có hai người nào trên hành tinh này mà có cùng chính xác chiều cao cả. Thật vậy, nếu đo đủ chính xác, chiều cao của bạn lúc mới thức dậy và khi đi ngủ sẽ khác nhau. Cho nên không có lý do gì để tìm xác suất một người nào đó cao 1.80139278291028719210196740527486202 mét cả. Trong toàn bộ dân số trên thế giới, xác suất này gần như bằng 0. Sẽ có lý hơn nếu ta hỏi chiều cao của một người nào đó có rơi vào một khoảng cho trước hay không, ví dụ như giữa 1.79 và 1.81 mét. Trong các trường hợp này, ta có thể định lượng khả năng mà ta thấy một giá trị nào đó theo một *mật độ xác suất*. Xác suất để có chiều cao chính xác 1.80 mét không tồn tại, nhưng mật độ của sự kiện này khác không. Trong bất kỳ khoảng nào giữa hai chiều cao khác nhau ta đều có xác suất khác không. Trong phần còn lại của mục này, ta sẽ xem xét xác suất trong không gian rời rạc. Về xác suất của biến ngẫu nhiên liên tục, bạn có thể xem ở [Section 20.6](#).

### 4.6.2 Làm việc với Nhiều Biến Ngẫu nhiên

Chúng ta sẽ thường xuyên phải làm việc với nhiều hơn một biến ngẫu nhiên cùng lúc. Ví dụ, chúng ta có thể muốn mô hình hóa mối quan hệ giữa các loại bệnh và các triệu chứng bệnh. Cho một loại bệnh và một triệu chứng bệnh, giả sử “cảm cúm” và “ho”, chúng có thể xuất hiện hoặc không trên một bệnh nhân với xác suất nào đó. Mặc dù chúng ta hy vọng xác suất cả hai xảy ra gần bằng không, ta có thể vẫn muốn ước lượng các xác suất này và mối quan hệ giữa chúng để có thể thực hiện các biện pháp chăm sóc y tế tốt hơn.

Xét một ví dụ phức tạp hơn: các bức ảnh chứa hàng triệu điểm ảnh, tương ứng với hàng triệu biến ngẫu nhiên. Và trong nhiều trường hợp các bức ảnh sẽ được gắn một nhãn chứa tên các vật xuất hiện trong ảnh. Chúng ta cũng có thể xem nhãn này như một biến ngẫu nhiên. Thậm chí, ta còn có thể xem tất cả các siêu dữ liệu như địa điểm, thời gian, khẩu độ, tiêu cự, ISO, khoảng lấy nét và loại máy ảnh, là các biến ngẫu nhiên. Tất cả các những biến ngẫu nhiên này xảy ra đồng thời. Khi làm việc với nhiều biến ngẫu nhiên, có một số đại lượng đáng được quan tâm.

## Xác suất Đồng thời

Đầu tiên là xác suất đồng thời  $P(A = a, B = b)$ . Cho hai biến  $a$  và  $b$  bất kỳ, xác suất đồng thời cho ta biết xác suất để cả  $A = a$  và  $B = b$  đều xảy ra. Ta có thể thấy rằng với mọi giá trị  $a$  và  $b$ ,  $P(A = a, B = b) \leq P(A = a)$ . Bởi để  $A = a$  và  $B = b$  xảy ra thì  $A = a$  phải xảy ra và  $B = b$  cũng phải xảy ra (và ngược lại). Do đó, khả năng  $A = a$  và  $B = b$  xảy ra đồng thời không thể lớn hơn khả năng  $A = a$  hoặc  $B = b$  xảy ra một cách độc lập được.

## Xác suất có điều kiện

Điều này giúp ta thu được một tỉ lệ thú vị:  $0 \leq \frac{P(A=a, B=b)}{P(A=a)} \leq 1$ . Chúng ta gọi tỉ lệ này là *xác suất có điều kiện* và ký hiệu là  $P(B = b | A = a)$ : xác suất để  $B = b$ , với điều kiện  $A = a$  đã xảy ra.

## Định lý Bayes

Sử dụng định nghĩa của xác suất có điều kiện, chúng ta có thể thu được một trong những phương trình nổi tiếng và hữu dụng nhất trong thống kê: *định lý Bayes*. Cụ thể như sau: Theo định nghĩa chúng ta có quy tắc nhân  $P(A, B) = P(B | A)P(A)$ . Tương tự, ta cũng có  $P(A, B) = P(A | B)P(B)$ . Giả sử  $P(B) > 0$ . Kết hợp các điều kiện trên ta có:

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)}. \quad (4.6.1)$$

Lưu ý rằng ở đây chúng ta sử dụng ký hiệu ngắn gọn hơn, với  $P(A, B)$  là *xác suất đồng thời* và  $P(A | B)$  là *xác suất có điều kiện*. Các phân phối này có thể được tính tại các giá trị cụ thể  $A = a, B = b$ .

## Phép biến hóa

Định lý Bayes rất hữu ích nếu chúng ta muốn suy luận một điều gì đó từ một điều khác, như là nguyên nhân và kết quả, nhưng ta chỉ biết các đặc tính theo chiều ngược lại, như ta sẽ thấy trong phần sau của chương này. Chúng ta cần làm một thao tác quan trọng để đạt được điều này, đó là *phép biến hóa*. Có thể hiểu là việc xác định  $P(B)$  từ  $P(A, B)$ . Chúng ta có thể tính được xác suất của  $B$  bằng tổng xác suất kết hợp của  $A$  và  $B$  tại mọi giá trị có thể của  $A$ :

$$P(B) = \sum_A P(A, B), \quad (4.6.2)$$

Công thức này cũng được biết đến với tên gọi *quy tắc tổng*. Xác suất hay phân phối thu được từ thao tác biến hóa được gọi là *xác suất biến* hoặc *phân phối biến*.

## Tính độc lập

Một tính chất hữu ích khác cần kiểm tra là *tính phụ thuộc* và *tính độc lập*. Hai biến ngẫu nhiên  $A$  và  $B$  độc lập nghĩa là việc một sự kiện của  $A$  xảy ra không tiết lộ bất kỳ thông tin nào về việc xảy ra một sự kiện của  $B$ . Trong trường hợp này  $P(B | A) = P(B)$ . Các nhà thống kê thường biểu diễn điều này bằng ký hiệu  $A \perp B$ . Từ định lý Bayes, ta có  $P(A | B) = P(A)$ . Trong tất cả các trường hợp khác, chúng ta gọi  $A$  và  $B$  là hai biến phụ thuộc. Ví dụ, hai lần đổ liên tiếp của một con xúc xắc là độc lập. Ngược lại, vị trí của công tắc đèn và độ sáng trong phòng là không độc lập

(tuy nhiên chúng không hoàn toàn xác định, vì bóng đèn luôn có thể bị hỏng, mất điện hoặc công tắc bị hỏng).

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

## Ứng dụng

Hãy thử nghiệm các kiến thức chúng ta vừa học. Giả sử rằng một bác sĩ phụ trách xét nghiệm AIDS cho một bệnh nhân. Việc xét nghiệm này khá chính xác và nó chỉ thất bại với xác suất 1%, khi nó cho kết quả dương tính dù bệnh nhân khỏe mạnh. Hơn nữa, nó không bao giờ thất bại trong việc phát hiện HIV nếu bệnh nhân thực sự bị nhiễm bệnh. Ta sử dụng  $D_1$  để biểu diễn kết quả chẩn đoán (1 nếu dương tính và 0 nếu âm tính) và  $H$  để biểu thị tình trạng nhiễm HIV (1 nếu dương tính và 0 nếu âm tính). [Table 4.6.1](#) liệt kê xác suất có điều kiện đó.

Table 4.6.1: Xác suất có điều kiện của  $P(D_1 | H)$ .

Xác suất có điều kiện	$H = 1$	$H = 0$
$P(D_1 = 1   H)$	1	0.01
$P(D_1 = 0   H)$	0	0.99

Lưu ý rằng tổng của từng cột đều bằng 1 (nhưng tổng từng hàng thì không), vì xác suất có điều kiện cần có tổng bằng 1, giống như xác suất. Hãy cùng tìm xác suất bệnh nhân bị AIDS nếu xét nghiệm trả về kết quả dương tính, tức  $P(H = 1 | D_1 = 1)$ . Rõ ràng điều này sẽ phụ thuộc vào mức độ phổ biến của bệnh, bởi vì nó ảnh hưởng đến số lượng dương tính giả. 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 phép biến hóa và quy tắc nhân để xác định

$$\begin{aligned} & P(D_1 = 1) \\ &= P(D_1 = 1, H = 0) + P(D_1 = 1, H = 1) \\ &= P(D_1 = 1 | H = 0)P(H = 0) + P(D_1 = 1 | H = 1)P(H = 1) \\ &= 0.011485. \end{aligned} \tag{4.6.3}$$

Do đó, ta có

$$\begin{aligned} & P(H = 1 | D_1 = 1) \\ &= \frac{P(D_1 = 1 | H = 1)P(H = 1)}{P(D_1 = 1)} \\ &= 0.1306 \end{aligned} \tag{4.6.4}$$

Nói cách khác, chỉ có 13,06% khả năng bệnh nhân thực sự mắc bệnh AIDS, dù ta dùng một bài kiểm tra rất chính xác. Như ta có thể thấy, xác suất có thể trở nên khá phản trực giác.

Một bệnh nhân phải làm gì nếu nhận được tin dữ như vậy? Nhiều khả năng họ sẽ yêu cầu bác sĩ thực hiện một xét nghiệm khác để làm rõ sự việc. Bài kiểm tra thứ hai có những đặc điểm khác và không tốt bằng bài thứ nhất, như ta có thể thấy trong [Table 4.6.2](#).

Table 4.6.2: Xác suất có điều kiện của  $P(D_2 | H)$ .

Xác suất có điều kiện	$H = 1$	$H = 0$
$P(D_2 = 1   H)$	0.98	0.03
$P(D_2 = 0   H)$	0.02	0.97

Không may thay, bài kiểm tra thứ hai cũng có kết quả dương tính. Hãy cùng tính các xác suất cần thiết để sử dụng định lý Bayes bằng cách giả định tính độc lập có điều kiện:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 | H = 0) \\ &= P(D_1 = 1 | H = 0)P(D_2 = 1 | H = 0) \\ &= 0.0003, \end{aligned} \tag{4.6.5}$$

$$\begin{aligned} & P(D_1 = 1, D_2 = 1 | H = 1) \\ &= P(D_1 = 1 | H = 1)P(D_2 = 1 | H = 1) \\ &= 0.98. \end{aligned} \tag{4.6.6}$$

Bây giờ chúng ta có thể áp dụng phép biến hóa và quy tắc nhân xác suất:

$$\begin{aligned} & P(D_1 = 1, D_2 = 1) \\ &= P(D_1 = 1, D_2 = 1, H = 0) + P(D_1 = 1, D_2 = 1, H = 1) \\ &= P(D_1 = 1, D_2 = 1 | H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 | H = 1)P(H = 1) \\ &= 0.00176955. \end{aligned} \tag{4.6.7}$$

Cuối cùng xác suất bệnh nhân mắc bệnh AIDS qua hai lần dương tính là

$$\begin{aligned} & P(H = 1 | D_1 = 1, D_2 = 1) \\ &= \frac{P(D_1 = 1, D_2 = 1 | H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)} \\ &= 0.8307. \end{aligned} \tag{4.6.8}$$

Cụ thể hơn, thử nghiệm thứ hai mang lại độ tin cậy cao hơn rằng không phải mọi chuyện đều ổn. Mặc dù bài kiểm tra thứ hai kém chính xác hơn bài đầu, nó vẫn cải thiện đáng kể dự đoán.

### 4.6.3 Kỳ vọng và Phương sai

Để tóm tắt những đặc tính then chốt của các phân phối xác suất, chúng ta cần một vài phép đo. *Kỳ vọng* (hay trung bình) của một biến ngẫu nhiên  $X$ , được ký hiệu là

$$E[X] = \sum_x xP(X = x). \tag{4.6.9}$$

Khi giá trị đầu vào của phương trình  $f(x)$  là một biến ngẫu nhiên nhiên cho trước theo phân phối  $P$  với các giá trị  $x$  khác nhau, kỳ vọng của  $f(x)$  sẽ được tính theo phương trình:

$$E_{x \sim P}[f(x)] = \sum_x f(x)P(x). \tag{4.6.10}$$

Trong nhiều trường hợp, chúng ta muốn đo độ lệch của biến ngẫu nhiên  $X$  so với kỳ vọng của nó. Đại lượng này có thể được đo bằng phuơng sai

$$\text{Var}[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2. \quad (4.6.11)$$

Nếu lấy căn bậc hai của kết quả ta sẽ được độ lệch chuẩn. Phuơng sai của một hàm của một biến ngẫu nhiên đo độ lệch của hàm số đó từ kỳ vọng của nó khi các giá trị  $x$  khác nhau được lấy mẫu từ phân phối của biến ngẫu nhiên đó:

$$\text{Var}[f(x)] = E[(f(x) - E[f(x)])^2]. \quad (4.6.12)$$

#### 4.6.4 Tóm tắt

- Chúng ta có thể sử dụng MXNet để lấy mẫu từ phân phối xác suất.
- Các biến ngẫu nhiên có thể được phân tích bằng các phuơng pháp như phân phối đồng thời (*joint distribution*), phân phối có điều kiện (*conditional distribution*), định lý Bayes, phép biên hóa (*marginalization*) và giả định độc lập (*independence assumptions*).
- Kỳ vọng và phuơng sai là các phép đo hữu ích để tóm tắt các đặc điểm chính của phân phối xác suất.

#### 4.6.5 Bài tập

1. Tiến hành  $m = 500$  nhóm thí nghiệm với mỗi nhóm lấy ra  $n = 10$  mẫu. Thay đổi  $m$  và  $n$ . Quan sát và phân tích kết quả của thí nghiệm.
2. Cho hai sự kiện với xác suất  $P(\mathcal{A})$  và  $P(\mathcal{B})$ , tính giới hạn trên và dưới của  $P(\mathcal{A} \cup \mathcal{B})$  và  $P(\mathcal{A} \cap \mathcal{B})$ . (Gợi ý: sử dụng biểu đồ Venn<sup>65</sup>.)
3. Giả sử chúng ta có các biến ngẫu nhiên  $A, B$  và  $C$ , với  $B$  chỉ phụ thuộc  $A$ , và  $C$  chỉ phụ thuộc vào  $B$ . Làm thế nào để đơn giản hóa xác suất đồng thời của  $P(A, B, C)$ ? (Gợi ý: đây là một Chuỗi Markov<sup>66</sup>.)
4. Trong Section 4.6.2, bài xét nghiệm đầu tiên có độ chính xác cao hơn. Vậy tại sao chúng ta không sử dụng bài xét nghiệm đầu tiên cho lần thử tiếp theo?

#### 4.6.6 Thảo luận

- Tiếng Anh<sup>67</sup>
- Tiếng Việt<sup>68</sup>

<sup>65</sup> [https://en.wikipedia.org/wiki/Venn\\_diagram](https://en.wikipedia.org/wiki/Venn_diagram)

<sup>66</sup> [https://en.wikipedia.org/wiki/Markov\\_chain](https://en.wikipedia.org/wiki/Markov_chain)

<sup>67</sup> <https://discuss.mxnet.io/t/2319>

<sup>68</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 4.6.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Vũ Hữu Tiệp
- Nguyễn Cảnh Thưởng
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Mai Sơn Hải
- Trần Kiến An
- Tạ H. Duy Nguyên
- Phạm Minh Đức
- Trần Thị Hồng Hạnh
- Lê Thành Vinh
- Nguyễn Minh Thư

## 4.7 Tài liệu

Vì độ dài cuốn sách này có giới hạn, chúng tôi không thể giới thiệu hết tất cả các hàm và lớp của MXNet (và tốt nhất nên như vậy). Tài liệu API, các hướng dẫn và ví dụ sẽ cung cấp nhiều thông tin vượt ra khỏi nội dung cuốn sách. Trong chương này, chúng tôi sẽ cung cấp một vài chỉ dẫn để bạn có thể khám phá MXNet API.

### 4.7.1 Tra cứu tất cả các hàm và lớp trong một Mô-đun

Để biết những hàm/lớp nào có thể được gọi trong một mô-đun, chúng ta dùng hàm `dir`. Ví dụ, ta có thể lấy tất cả thuộc tính của mô-đun `np.random` bằng cách:

```
from mxnet import np  
print(dir(np.random))
```

Thông thường, ta có thể bỏ qua những hàm bắt đầu và kết thúc với `__` (các đối tượng đặc biệt trong Python) hoặc những hàm bắt đầu bằng `_` (thường là các hàm địa phương). Dựa trên tên của những hàm và thuộc tính còn lại, ta có thể dự đoán rằng mô-đun này cung cấp những phương thức sinh số ngẫu nhiên, bao gồm lấy mẫu từ phân phối đều liên tục (`uniform`), phân phối chuẩn (`normal`) và phân phối đa thức (`multinomial`).

## 4.7.2 Tra cứu cách sử dụng một hàm hoặc một lớp cụ thể

Để tra cứu chi tiết cách sử dụng một hàm hoặc lớp nhất định, ta dùng hàm `help`. Ví dụ, để tra cứu cách sử dụng hàm `ones_like` với `ndarray`:

```
help(np.ones_like)
```

Từ tài liệu, ta có thể thấy hàm `ones_like` tạo một mảng mới có cùng kích thước với `ndarray` nhưng tất cả các phần tử của nó đều chứa giá trị 1. Nếu có thể, bạn nên chạy thử để xác nhận rằng mình hiểu đúng.

```
x = np.array([[0, 0, 0], [2, 2, 2]])
np.ones_like(x)
```

Trong Jupyter notebook, ta có thể dùng `?`  để mở tài liệu trong một cửa sổ khác. Ví dụ, `np.random.uniform?` sẽ in ra nội dung y hệt `help(np.random.uniform)` trong một cửa sổ trình duyệt mới. Ngoài ra, nếu chúng ta dùng dấu `?`  hai lần như `np.random.uniform??` thì đoạn mã định nghĩa hàm cũng sẽ được in ra.

## 4.7.3 Tài liệu API

Chi tiết cụ thể về các API của MXNet có thể được tìm thấy tại trang <http://mxnet.apache.org/>. Chi tiết từng phần được tìm thấy tại các đề mục tương ứng (cho cả các ngôn ngữ lập trình khác ngoài Python).

## 4.7.4 Tóm tắt

- Tài liệu chính thức cung cấp rất nhiều các mô tả và ví dụ ngoài cuốn sách này.
- Chúng ta có thể tra cứu tài liệu về cách sử dụng MXNet API bằng cách gọi hàm `dir` và `help`, hoặc kiểm tra tại trang web của MXNet.

## 4.7.5 Bài tập

1. Tra cứu `ones_like` và `autograd` trên trang MXNet.
2. Tất cả các kết quả khả dĩ sau khi chạy `np.random.choice(4, 2)` là gì?
3. Bạn có thể viết lại `np.random.choice(4, 2)` bằng cách sử dụng hàm `np.random.randint` không?

#### **4.7.6 Thảo luận**

- Tiếng Anh<sup>69</sup>
- Tiếng Việt<sup>70</sup>

#### **4.7.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Hoàng Quân
- Vũ Hữu Tiệp

### **4.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Cảnh Thưởng
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

---

<sup>69</sup> <https://discuss.mxnet.io/t/2322>

<sup>70</sup> <https://forum.machinelearningcoban.com/c/d2l>

# 5 | Mạng nơ-ron Tuyến tính

Trước khi tìm hiểu chi tiết về mạng nơ-ron sâu, chúng ta cần nắm vững những kiến thức căn bản của việc huấn luyện mạng nơ-ron. Chương này sẽ đề cập đến toàn bộ quá trình huấn luyện, bao gồm xác định kiến trúc mạng nơ-ron đơn giản, xử lý dữ liệu, chỉ rõ hàm mất mát và huấn luyện mô hình. Để mọi thứ dễ dàng hơn, ta sẽ bắt đầu với một số khái niệm đơn giản nhất. May thay, một số phương pháp học thống kê cổ điển như hồi quy tuyến tính, hồi quy logistic có thể được xem như những mạng nơ-ron *nông*. Hãy bắt đầu bằng những thuật toán cổ điển này, chúng tôi sẽ giới thiệu những nội dung căn bản nhằm tạo nền tảng cho những kỹ thuật phức tạp hơn như Hồi quy Softmax (sẽ được giới thiệu ở cuối chương này) và Perceptron đa tầng (sẽ được giới thiệu ở chương sau).

## 5.1 Hồi quy Tuyến tính

Hồi quy ám chỉ các phương pháp để xây dựng mối quan hệ giữa điểm dữ liệu  $x$  và mục tiêu với giá trị số thực  $y$ . Trong khoa học tự nhiên và khoa học xã hội, mục tiêu của hồi quy thường là *đặc trưng hóa* mối quan hệ của đầu vào và đầu ra. Mặt khác, học máy lại thường quan tâm đến việc *dự đoán*.

Bài toán hồi quy xuất hiện mỗi khi chúng ta muốn dự đoán một giá trị số. Các ví dụ phổ biến bao gồm dự đoán giá cả (nhà, cổ phiếu, ...), thời gian bệnh nhân nằm viện, nhu cầu trong ngành bán lẻ và vô vàn thứ khác. Không phải mọi bài toán dự đoán đều là bài toán *hồi quy* cổ điển. Trong các phần tiếp theo, chúng tôi sẽ giới thiệu bài toán phân loại, khi mục tiêu là dự đoán lớp đúng trong một tập các lớp cho trước.

### 5.1.1 Các Thành phần Cơ bản của Hồi quy Tuyến tính

*Hồi quy tuyến tính* có lẽ là công cụ tiêu chuẩn đơn giản và phổ biến nhất được sử dụng cho bài toán hồi quy. Xuất hiện từ đầu thế kỉ 19, hồi quy tuyến tính được phát triển từ một vài giả thuyết đơn giản. Đầu tiên, ta giả sử quan hệ giữa các *đặc trưng*  $x$  và mục tiêu  $y$  là tuyến tính, do đó  $y$  có thể được biểu diễn bằng tổng trọng số của đầu vào  $x$ , cộng hoặc trừ thêm nhiễu của các quan sát. Thứ hai, ta giả sử nhiễu có quy tắc (tuân theo phân phối Gauss). Để tạo động lực, hãy bắt đầu với một ví dụ. Giả sử ta muốn ước lượng giá nhà (bằng đô la) dựa vào diện tích (đơn vị feet vuông) và tuổi đời (theo năm).

Để khớp một mô hình dự đoán giá nhà, chúng ta cần một tập dữ liệu các giao dịch mà trong đó ta biết giá bán, diện tích, tuổi đời cho từng căn nhà. Trong thuật ngữ của học máy, tập dữ liệu này được gọi là *dữ liệu huấn luyện* hoặc *tập huấn luyện*, và mỗi hàng (tương ứng với dữ liệu của một giao dịch) được gọi là một *ví dụ* hoặc *mẫu*. Thứ mà chúng ta muốn dự đoán (giá nhà) được gọi là *mục*

*tiêu hoặc nhãn.* Các biến (*tuổi đời* và *diện tích*) mà những dự đoán dựa vào được gọi là các *đặc trưng* hoặc *hiệp biến*.

Thông thường, chúng ta sẽ dùng  $n$  để kí hiệu số lượng mẫu trong tập dữ liệu. Chỉ số  $i$  được dùng để xác định một mẫu cụ thể. Ta ký hiệu mỗi điểm dữ liệu đầu vào là  $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$  và nhãn tương ứng là  $y^{(i)}$ .

## Mô hình Tuyến tính

Giả định tuyến tính trên cho thấy rằng mục tiêu (giá nhà) có thể được biểu diễn bởi tổng có trọng số của các đặc trưng (diện tích và tuổi đời):

$$\text{giánhà} = w_{\text{diện_tích}} \cdot \text{diện_tích} + w_{\text{tuổi_đời}} \cdot \text{tuổi_đời} + b. \quad (5.1.1)$$

Ở đây,  $w_{\text{diện_tích}}$  và  $w_{\text{tuổi_đời}}$  được gọi là *trọng số*, và  $b$  được gọi là *hệ số điều chỉnh* (còn được gọi là *độ dời*). Các trọng số xác định mức độ đóng góp của mỗi đặc trưng tới đầu ra, còn hệ số điều chỉnh là dự đoán của giá nhà khi tất cả các đặc trưng đều bằng 0. Ngay cả khi không bao giờ có một ngôi nhà có diện tích hoặc tuổi đời bằng không, ta vẫn cần sử dụng hệ số điều chỉnh; nếu không khả năng biểu diễn của mô hình tuyến tính sẽ bị suy giảm.

Cho một tập dữ liệu, mục đích của chúng ta là chọn được các trọng số  $w$  và hệ số điều chỉnh  $b$  sao cho dự đoán của mô hình khớp nhất với giá nhà thực tế quan sát được trong dữ liệu.

Trong các bài toán mà tập dữ liệu thường chỉ có một vài đặc trưng, biểu diễn tường minh mô hình ở dạng biểu thức dài như trên khá là phổ biến. Trong học máy, chúng ta thường làm việc với các tập dữ liệu nhiều chiều, vì vậy sẽ tốt hơn nếu ta tận dụng các ký hiệu trong đại số tuyến tính. Khi đầu vào của mô hình có  $d$  đặc trưng, ta biểu diễn dự đoán  $\hat{y}$  bởi

$$\hat{y} = w_1 \cdot x_1 + \dots + w_d \cdot x_d + b. \quad (5.1.2)$$

Thu thập toàn bộ các đặc trưng vào một vector  $\mathbf{x}$  và toàn bộ các trọng số vào một vector  $\mathbf{w}$ , ta có thể biểu diễn mô hình một cách gọn gàng bằng tích vô hướng:

$$\hat{y} = \mathbf{w}^T \mathbf{x} + b. \quad (5.1.3)$$

Ở đây, vector  $\mathbf{x}$  tương ứng với một điểm dữ liệu. Chúng ta sẽ thấy rằng việc truy cập đến toàn bộ tập dữ liệu sẽ tiện hơn nếu ta biểu diễn tập dữ liệu bằng *ma trận X*. Mỗi hàng của ma trận  $\mathbf{X}$  thể hiện một mẫu và mỗi cột thể hiện một đặc trưng.

Với một tập hợp điểm dữ liệu  $\mathbf{X}$ , kết quả dự đoán  $\hat{\mathbf{y}}$  có thể được biểu diễn bằng phép nhân giữa ma trận và vector:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b. \quad (5.1.4)$$

Cho một tập dữ liệu huấn luyện  $\mathbf{X}$  và các giá trị mục tiêu đã biết trước  $\mathbf{y}$ , mục tiêu của hồi quy tuyến tính là tìm vector *trọng số*  $\mathbf{w}$  và *hệ số điều chỉnh*  $b$  sao cho với một điểm dữ liệu mới  $\mathbf{x}_i$  được lấy mẫu từ cùng phân phối của tập huấn luyện, giá trị mục tiêu  $y_i$  sẽ được dự đoán với sai số nhỏ nhất (theo kỳ vọng).

Kể cả khi biết rằng mô hình tuyến tính là lựa chọn tốt nhất để dự đoán  $y$  từ  $\mathbf{x}$ , chúng ta cũng không kỳ vọng tìm được dữ liệu thực tế mà ở đó  $y$  đúng bằng  $\mathbf{w}^T \mathbf{x} + b$  với mọi điểm  $(\mathbf{x}, y)$ . Để dễ hình dung, mọi thiết bị đo lường dùng để quan sát đặc trưng  $\mathbf{X}$  và nhãn  $\mathbf{y}$  đều có sai số nhất định. Chính

vì vậy, kể cả khi ta chắc chắn rằng mỗi quan hệ ẩn sau tập dữ liệu là tuyến tính, chúng ta sẽ thêm một thành phần nhiễu để giải thích các sai số đó.

Trước khi tiến hành tìm các giá trị tốt nhất cho  $\mathbf{w}$  và  $b$ , chúng ta sẽ cần thêm hai thứ nữa: (i) một phép đo đánh giá chất lượng mô hình và (ii) quy trình cập nhật mô hình để cải thiện chất lượng.

## Hàm mất mát

Trước khi suy nghĩ về việc làm thế nào để *khớp* mô hình với dữ liệu, ta cần phải xác định một phương pháp để đo *mức độ khớp*. *Hàm mất mát* định lượng khoảng cách giữa giá trị *thực* và giá trị *dự đoán* của mục tiêu. Độ mất mát thường là một số không âm và có giá trị càng nhỏ càng tốt. Khi các dự đoán hoàn hảo, chúng sẽ có độ mất mát sẽ bằng 0. Hàm mất mát thông dụng nhất trong các bài toán hồi quy là hàm tổng bình phương các lỗi. Khi giá trị dự đoán của một điểm dữ liệu huấn luyện  $i$  là  $\hat{y}^{(i)}$  và nhãn tương ứng là  $y^{(i)}$ , bình phương của lỗi được xác định như sau:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} (\hat{y}^{(i)} - y^{(i)})^2. \quad (5.1.5)$$

Hằng số  $1/2$  không tạo ra sự khác biệt thực sự nào nhưng sẽ giúp ký hiệu thuận tiện hơn: nó sẽ được triệt tiêu khi lấy đạo hàm của hàm mất mát. Vì các dữ liệu trong tập huấn luyện đã được xác định trước và không thể thay đổi, sai số thực nghiệm chỉ là một hàm của các tham số mô hình. Để tìm hiểu cụ thể hơn, hãy xét ví dụ dưới đây về một bài toán hồi quy cho trường hợp một chiều trong Fig. 5.1.1.

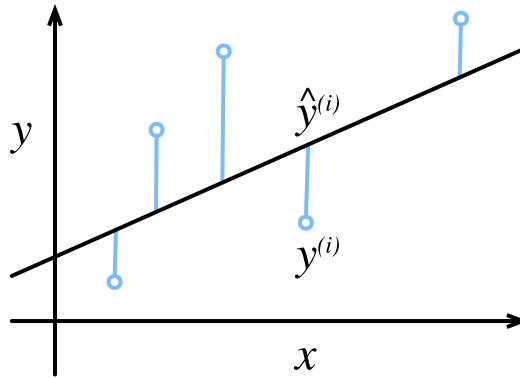


Fig. 5.1.1: Khớp dữ liệu với một mô hình tuyến tính.

Lưu ý rằng khi hiệu giữa giá trị ước lượng  $\hat{y}^{(i)}$  và giá trị quan sát  $y^{(i)}$  lớn, giá trị hàm mất mát sẽ tăng một lượng còn lớn hơn thế do sự phụ thuộc bậc hai. Để đo chất lượng của mô hình trên toàn bộ tập dữ liệu, ta đơn thuần lấy trung bình (hay tương đương là lấy tổng) các giá trị mất mát của từng mẫu trong tập huấn luyện.

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} (\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)})^2. \quad (5.1.6)$$

Khi huấn luyện mô hình, ta muốn tìm các tham số  $(\mathbf{w}^*, b^*)$  sao cho tổng độ mất mát trên toàn bộ các mẫu huấn luyện được cực tiểu hóa:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} L(\mathbf{w}, b). \quad (5.1.7)$$

## Nghiệm theo Công thức

Hóa ra hồi quy tuyến tính chỉ là một bài toán tối ưu hóa đơn giản. Khác với hầu hết các mô hình được giới thiệu trong cuốn sách này, hồi quy tuyến tính có thể được giải bằng cách áp dụng một công thức đơn giản, cho một nghiệm tối ưu toàn cục. Để bắt đầu, chúng ta có thể gộp hệ số điều chỉnh  $b$  vào tham số  $\mathbf{w}$  bằng cách thêm một cột toàn 1 vào ma trận dữ liệu. Khi đó bài toán dự đoán trở thành bài toán cực tiểu hóa  $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|$ . Bởi vì biểu thức này có dạng toàn phương, nó là một hàm số lồi, và miễn là bài toán này không suy biến (các đặc trưng độc lập tuyến tính), nó là một hàm số lồi chặt.

Bởi vậy chỉ có một điểm cực trị trên mặt mất mát và nó tương ứng với giá trị mất mát nhỏ nhất. Lấy đạo hàm của hàm mất mát theo  $\mathbf{w}$  và giải phương trình đạo hàm này bằng 0, ta sẽ được nghiệm theo công thức:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \quad (5.1.8)$$

Tuy những bài toán đơn giản như hồi quy tuyến tính có thể có nghiệm theo công thức, bạn không nên làm quen với sự may mắn này. Mặc dù các nghiệm theo công thức giúp ta phân tích toán học một cách thuận tiện, các điều kiện để có được nghiệm này chặt chẽ đến nỗi không có phương pháp học sâu nào thoả mãn được.

## Hạ Gradient

Trong nhiều trường hợp ở đó ta không thể giải quyết các mô hình theo phép phân tích, và thậm chí khi mặt mất mát là các mặt bậc cao và không lồi, trên thực tế ta vẫn có thể huấn luyện các mô hình này một cách hiệu quả. Hơn nữa, trong nhiều tác vụ, những mô hình khó để tối ưu hóa này hoá ra lại tốt hơn các phương pháp khác nhiều, vậy nên việc bỏ công sức để tìm cách tối ưu chúng là hoàn toàn xứng đáng.

Kỹ thuật chính để tối ưu hóa gần như bất kỳ mô hình học sâu nào, sẽ được sử dụng xuyên suốt cuốn sách này, bao gồm việc giảm thiểu lỗi qua các vòng lặp bằng cách cập nhật tham số theo hướng làm giảm dần hàm mất mát. Thuật toán này được gọi là *hạ gradient*. Trên các mặt mất mát lồi, giá trị mất mát cuối cùng sẽ hội tụ về giá trị nhỏ nhất. Tuy điều tương tự không thể áp dụng cho các mặt không lồi, ít nhất thuật toán sẽ dẫn tới một cực tiểu (hy vọng là tốt).

Ứng dụng đơn giản nhất của hạ gradient bao gồm việc tính đạo hàm của hàm mất mát, tức trung bình của các giá trị mất mát được tính trên mỗi mẫu của tập dữ liệu. Trong thực tế, việc này có thể cực kì chậm. Chúng ta phải duyệt qua toàn bộ tập dữ liệu trước khi thực hiện một lần cập nhật. Vì thế, thường ta chỉ muốn lấy một minibatch ngẫu nhiên các mẫu mỗi khi ta cần tính bước cập nhật. Phương pháp biến thể này được gọi là *hạ gradient ngẫu nhiên*.

Trong mỗi vòng lặp, đầu tiên chúng ta lấy ngẫu nhiên một minibatch  $\mathcal{B}$  dữ liệu huấn luyện với kích thước cố định. Sau đó, chúng ta tính đạo hàm (gradient) của hàm mất mát trên minibatch đó theo các tham số của mô hình. Cuối cùng, gradient này được nhân với tốc độ học  $\eta > 0$  và kết quả này được trừ đi từ các giá trị tham số hiện tại.

Chúng ta có thể biểu diễn việc cập nhật bằng công thức toán như sau ( $\partial$  là ký hiệu đạo hàm riêng của hàm số) :

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b). \quad (5.1.9)$$

Tổng kết lại, các bước của thuật toán như sau: (i) khởi tạo các giá trị tham số của mô hình, thường thì sẽ được chọn ngẫu nhiên. (ii) tại mỗi vòng lặp, ta lấy ngẫu nhiên từng batch từ tập dữ liệu (nhiều lần), rồi tiến hành cập nhật các tham số của mô hình theo hướng ngược với gradient.

Khi sử dụng hàm mất mát bậc hai và mô hình tuyến tính, chúng ta có thể biểu diễn bước này một cách tường minh như sau: Lưu ý rằng  $\mathbf{w}$  và  $\mathbf{x}$  là các vector. Ở đây, việc ký hiệu bằng các vector giúp công thức dễ đọc hơn nhiều so với việc biểu diễn bằng các hệ số như  $w_1, w_2, \dots, w_d$ .

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right).\end{aligned}\tag{5.1.10}$$

Trong phương trình trên,  $|\mathcal{B}|$  là số ví dụ trong mỗi minibatch (*kích thước batch*) và  $\eta$  là *tốc độ học*. Cũng cần phải nhấn mạnh rằng các giá trị của kích thước batch và tốc độ học được lựa chọn trước một cách thủ công và thường không được học thông qua quá trình huấn luyện mô hình. Các tham số này tuy điều chỉnh được nhưng không được cập nhật trong vòng huấn luyện, và được gọi là *siêu tham số*. *Điều chỉnh siêu tham số* là quá trình lựa chọn chúng, thường dựa trên kết quả của vòng lặp huấn luyện được đánh giá trên một tập *kiểm định* riêng biệt.

Sau khi huấn luyện đủ số vòng lặp được xác định trước (hoặc đạt được một tiêu chí dừng khác), ta sẽ ghi lại các tham số mô hình đã được ước lượng, ký hiệu là  $\hat{\mathbf{w}}, \hat{b}$  (ký hiệu “mũ” thường thể hiện các giá trị ước lượng). Lưu ý rằng ngay cả khi hàm số thực sự tuyến tính và không có nhiễu, các tham số này sẽ không cực tiểu hóa được hàm mất mát. Mặc dù thuật toán dần dần hội tụ đến một điểm cực tiểu, nó vẫn không thể tới chính xác được cực tiểu đó với số bước hữu hạn.

Hồi quy tuyến tính thực ra là một bài toán tối ưu lồi, do đó chỉ có một cực tiểu (toàn cục). Tuy nhiên, đối với các mô hình phức tạp hơn, như mạng sâu, mặt của hàm mất mát sẽ có nhiều cực tiểu. May mắn thay, vì một lý do nào đó mà những người làm về học sâu hiếm khi phải vật lộn để tìm ra các tham số cực tiểu hóa hàm mất mát *trên dữ liệu huấn luyện*. Nhiệm vụ khó khăn hơn là tìm ra các tham số dẫn đến giá trị mất mát thấp trên dữ liệu mà mô hình chưa từng thấy trước đây, một thử thách được gọi là *sự khai quát hóa*. Chúng ta sẽ gặp lại chủ đề này xuyên suốt cuốn sách.

## Dự đoán bằng Mô hình đã được Huấn luyện

Với mô hình hồi quy tuyến tính đã được huấn luyện  $\hat{\mathbf{w}}^\top x + \hat{b}$ , ta có thể ước lượng giá của một căn nhà mới (ngoài bộ dữ liệu dùng để huấn luyện) với diện tích  $x_1$  và tuổi đời  $x_2$  của nó. Việc ước lượng mục tiêu khi biết trước những đặc trưng thường được gọi là *dự đoán* hay *suy luận* (*inference*).

Ở đây ta sẽ dùng từ *dự đoán* thay vì *suy luận*, dù *suy luận* là một thuật ngữ khá phổ biến trong học sâu, áp dụng thuật ngữ này ở đây lại không phù hợp. Trong thống kê, *suy luận* thường được dùng cho việc ước lượng thông số dựa trên tập dữ liệu. Việc dùng sai thuật ngữ này là nguyên nhân gây ra sự hiểu nhầm giữa những người làm học sâu và các nhà thống kê.

## Vector hóa để tăng Tốc độ Tính toán

Khi huấn luyện mô hình, chúng ta thường muốn xử lý đồng thời các mẫu dữ liệu trong minibatch. Để làm được điều này một cách hiệu quả, chúng ta phải vector hóa việc tính toán bằng cách sử dụng các thư viện đại số tuyến tính thay vì sử dụng các vòng lặp for trong Python.

Chúng ta sẽ sử dụng hai phương pháp cộng vector dưới đây để hiểu được tại sao vector hóa là cần thiết trong học máy. Đầu tiên, ta khởi tạo hai vector 10000 chiều chứa toàn giá trị một. Chúng ta sẽ sử dụng vòng lặp for trong Python ở phương pháp thứ nhất và một hàm trong thư viện np ở phương pháp thứ hai.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np
import time

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

Vì ta sẽ cần đánh giá xếp hạng thời gian xử lý một cách thường xuyên trong cuốn sách này, ta sẽ định nghĩa một bộ tính giờ (sau đó có thể truy cập được thông qua gói d2l để theo dõi thời gian chạy).

```
# Saved in the d2l package for later use
class Timer(object):
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        # Start the timer
        self.start_time = time.time()

    def stop(self):
        # Stop the timer and record the time in a list
        self.times.append(time.time() - self.start_time)
        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 times
        return np.array(self.times).cumsum().tolist()
```

Bây giờ, ta có thể đánh giá xếp hạng hai phương pháp cộng vector. Đầu tiên, ta sử dụng vòng lặp for để cộng các tọa độ tương ứng.

```

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

```

Trong phương pháp hai, ta dựa vào thư viện np để tính tổng hai vector theo từng phần tử.

```

timer.start()
d = a + b
'%.5f sec' % timer.stop()

```

Bạn có thể nhận thấy rằng, phương pháp thứ hai nhanh hơn rất nhiều lần so với phương pháp thứ nhất. Việc vector hóa thường tăng tốc độ tính toán lên nhiều bậc. Ngoài ra, giao phó công việc tính toán cho thư viện để tránh phải tự viết lại sẽ giảm thiểu khả năng phát sinh lỗi.

### 5.1.2 Phân phối Chuẩn và Hàm mất mát Bình phương

Mặc dù bạn đã có thể thực hành với kiến thức được trình bày phía trên, trong phần tiếp theo chúng ta sẽ làm rõ hơn nguồn gốc của hàm mất mát bình phương thông qua các giả định về phân phối của nhiễu.

Nhắc lại ở trên rằng hàm mất mát bình phương  $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$  có nhiều thuộc tính tiện lợi. Việc nó có đạo hàm đơn giản  $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$  là một trong số đó.

Như được đề cập trước đó, hồi quy tuyến tính được phát minh bởi Gauss vào năm 1795. Ông cũng là người khám phá ra phân phối chuẩn (còn được gọi là *phân phối Gauss*). Hóa ra là mối liên hệ giữa phân phối chuẩn và hồi quy tuyến tính không chỉ dừng lại ở việc chúng có chung cha đẻ. Để gợi nhớ lại cho bạn, mật độ xác suất của phân phối chuẩn với trung bình  $\mu$  và phương sai  $\sigma^2$  được cho bởi:

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(z - \mu)^2\right). \quad (5.1.11)$$

Dưới đây ta định nghĩa một hàm Python để tính toán phân phối chuẩn.

```

x = np.arange(-7, 7, 0.01)

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

```

Giờ ta có thể trực quan hóa các phân phối chuẩn.

```

# Mean and variance pairs
parameters = [(0, 1), (0, 2), (3, 1)]
d2l.plot(x, [normal(x, mu, sigma) for mu, sigma in parameters], xlabel='z',
          ylabel='p(z)', figsize=(4.5, 2.5),
          legend=['mean %d, var %d' % (mu, sigma) for mu, sigma in parameters])

```

Có thể thấy rằng, thay đổi giá trị trung bình tương ứng với việc dịch chuyển phân phối dọc theo trục  $x$ , tăng giá trị phương sai sẽ trải rộng phân phối và hạ thấp đỉnh của nó.

Để thấy rõ hơn mối quan hệ giữa hồi quy tuyến tính và hàm mất mát trung bình bình phương sai số (MSE), ta có thể giả định rằng các quan sát bắt nguồn từ những quan sát nhiễu, và giá trị nhiễu này tuân theo phân phối chuẩn như sau:

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ tại } \epsilon \sim \mathcal{N}(0, \sigma^2). \quad (5.1.12)$$

Do đó, chúng ta có thể viết *khả năng thu được* một giá trị cụ thể của  $y$  khi biết trước  $\mathbf{x}$  là

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right). \quad (5.1.13)$$

Dựa vào *nguyên lý hợp lý cực đại*, giá trị tốt nhất của  $b$  và  $\mathbf{w}$  là những giá trị giúp cực đại hóa *sự hợp lý* của toàn bộ tập dữ liệu:

$$P(Y | X) = \prod_{i=1}^n p(y^{(i)}|\mathbf{x}^{(i)}). \quad (5.1.14)$$

Bộ ước lượng được chọn theo *nguyên lý hợp lý cực đại* được gọi là *bộ ước lượng hợp lý cực đại* (*Maximum Likelihood Estimators* - MLE). Dù việc cực đại hóa tích của nhiều hàm mũ trông có vẻ khó khăn, chúng ta có thể khiến mọi thứ đơn giản hơn nhiều mà không làm thay đổi mục tiêu ban đầu bằng cách cực đại hóa log của hàm hợp lý. Vì lý do lịch sử, các bài toán tối ưu thường được biểu diễn dưới dạng bài toán cực tiểu hóa thay vì cực đại hóa. Do đó chúng ta có thể cực tiểu hóa *hàm đối log hợp lý* (*Negative Log-Likelihood* - NLL) –  $-\log p(\mathbf{y}|\mathbf{X})$  mà không cần thay đổi gì thêm. Kết nối các công thức trên, ta có:

$$-\log p(\mathbf{y}|\mathbf{X}) = \sum_{i=1}^n \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2. \quad (5.1.15)$$

Giờ ta chỉ cần thêm một giả định nữa:  $\sigma$  là một hằng số cố định. Do đó, ta có thể bỏ qua số hạng đầu tiên bởi nó không phụ thuộc vào  $\mathbf{w}$  hoặc  $b$ . Còn số hạng thứ hai thì giống hệt hàm bình phương sai số đã được giới thiệu ở trên, nhưng được nhân thêm với hằng số  $\frac{1}{\sigma^2}$ . May mắn thay, nghiệm không phụ thuộc vào  $\sigma$ . Điều này dẫn tới việc cực tiểu hóa bình phương sai số tương đương với việc ước lượng hợp lý cực đại cho mô hình tuyến tính dưới giả định có nhiễu cộng Gauss.

### 5.1.3 Từ Hồi quy Tuyến tính tới Mạng Học sâu

Cho đến nay, chúng ta mới chỉ đề cập về các hàm tuyến tính. Trong khi mạng nơ-ron có thể xấp xỉ rất nhiều họ mô hình, ta có thể bắt đầu coi mô hình tuyến tính như một mạng nơ-ron và biểu diễn nó theo ngôn ngữ của mạng nơ-ron. Để bắt đầu, hãy cùng viết lại mọi thứ theo ký hiệu ‘tầng’ (*layer*).

#### Giản đồ Mạng Nơ-ron

Những người làm học sâu thích vẽ giản đồ để trực quan hóa những gì đang xảy ra trong mô hình của họ. Trong Fig. 5.1.2, mô hình tuyến tính được minh họa như một mạng nơ-ron. Những giản đồ này chỉ ra cách kết nối (ở đây, mỗi đầu vào được kết nối tới đầu ra) nhưng không có giá trị của các trọng số và các hệ số điều chỉnh.

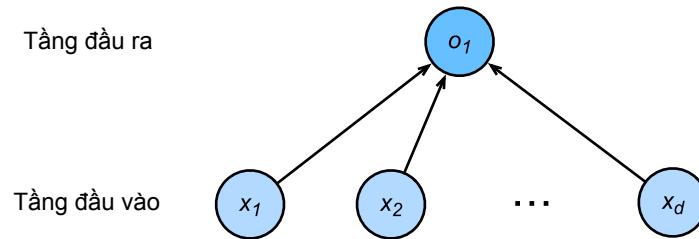


Fig. 5.1.2: Hồi quy tuyến tính là một mạng nơ-ron đơn tầng.

Vì chỉ có một nơ-ron tính toán (một nút) trong đồ thị (các giá trị đầu vào không cần tính mà được cho trước), chúng ta có thể coi mô hình tuyến tính như mạng nơ-ron với chỉ một nơ-ron nhân tạo duy nhất. Với mô hình này, mọi đầu vào đều được kết nối tới mọi đầu ra (trong trường hợp này chỉ có một đầu ra!), ta có thể coi phép biến đổi này là một *tầng kết nối dày đặc*, hay còn gọi là *tầng kết nối dày đặc*. Chúng ta sẽ nói nhiều hơn về các mạng nơ-ron cấu tạo từ những tầng như vậy trong chương kế tiếp về mạng perceptron đa tầng.

### Sinh vật học

Vì hồi quy tuyến tính (được phát minh vào năm 1795) được phát triển trước ngành khoa học thần kinh tính toán, nên việc mô tả hồi quy tuyến tính như một mạng nơ-ron có vẻ hơi ngược thời. Để hiểu tại sao nhà nghiên cứu sinh vật học/thần kinh học Warren McCulloch và Walter Pitts tìm đến các mô hình tuyến tính để làm điểm khởi đầu nghiên cứu và phát triển các mô hình nơ-ron nhân tạo, hãy xem ảnh của một nơ-ron sinh học tại Fig. 5.1.3. Mô hình này bao gồm sợi nhánh (cổng đầu vào), *nhân tế bào* (bộ xử lý trung tâm), *sợi trực* (dây đầu ra), và *đầu cuối sợi trực* (cổng đầu ra), cho phép kết nối với các tế bào thần kinh khác thông qua *synapses*.

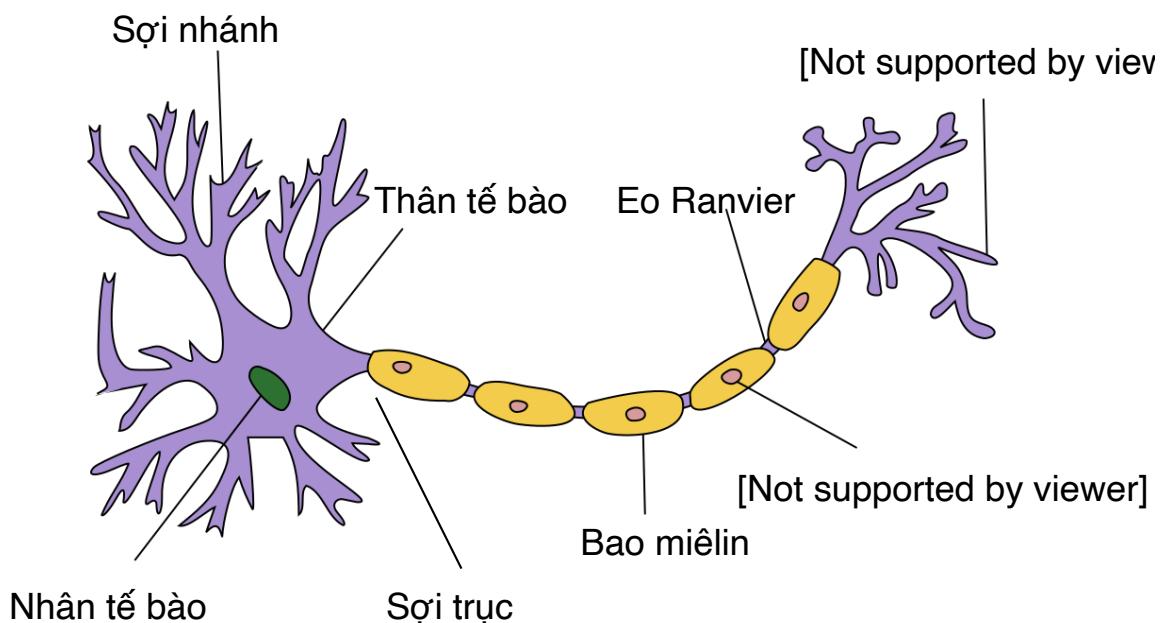


Fig. 5.1.3: Nơ-ron trong thực tế

Thông tin  $x_i$  đến từ các nơ-ron khác (hoặc các cảm biến môi trường như võng mạc) được tiếp

nhận tại các sợi nhánh. Cụ thể, thông tin đó được nhân với các trọng số của synapses  $w_i$  để xác định mức ảnh hưởng của từng đầu vào (ví dụ: kích hoạt hoặc ức chế thông qua tích  $x_i w_i$ ). Các đầu vào có trọng số đến từ nhiều nguồn được tổng hợp trong nhân tế bào dưới dạng tổng có trọng số  $y = \sum_i x_i w_i + b$  và thông tin này sau đó được gửi đi để xử lý thêm trong sợi trực  $y$ , thường là sau một vài xử lý phi tuyến tính qua  $\sigma(y)$ . Từ đó, nó có thể được gửi đến đích (ví dụ, cơ bắp) hoặc được đưa vào một tế bào thần kinh khác thông qua các sợi nhánh.

Dựa trên các nghiên cứu thực tế về các hệ thống thần kinh sinh học, ta chắc chắn một điều rằng nhiều đơn vị như vậy khi được kết hợp với nhau theo đúng cách, cùng với thuật toán học phù hợp, sẽ tạo ra các hành vi thú vị và phức tạp hơn nhiều so với bất kỳ nơ-ron đơn lẻ nào có thể làm được.

Đồng thời, hầu hết các nghiên cứu trong học sâu ngày nay chỉ lấy một phần cảm hứng nhỏ từ ngành thần kinh học. Như trong cuốn sách kinh điển về AI *Trí tuệ Nhân tạo: Một hướng Tiếp cận Hiện đại* (Russell & Norvig, 2016) của Stuart Russell và Peter Norvig, họ đã chỉ ra rằng: mặc dù máy bay có thể được lấy cảm hứng từ loài chim, ngành điều học không phải động lực chính làm đổi mới ngành hàng không trong nhiều thế kỷ qua. Tương tự, cảm hứng trong học sâu hiện nay chủ yếu đến từ ngành toán học, thống kê và khoa học máy tính.

#### 5.1.4 Tóm tắt

- Nguyên liệu của một mô hình học máy bao gồm dữ liệu huấn luyện, một hàm mất mát, một thuật toán tối ưu, và tất nhiên là cả chính mô hình đó.
- Vector hóa giúp mọi thứ trở nên dễ hiểu hơn (về mặt toán học) và nhanh hơn (về mặt lập trình).
- Cực tiểu hóa hàm mục tiêu và thực hiện phương pháp hợp lý cực đại có ý nghĩa giống nhau.
- Các mô hình tuyến tính cũng là các mạng nơ-ron.

#### 5.1.5 Bài tập

1. Giả sử ta có dữ liệu  $x_1, \dots, x_n \in \mathbb{R}$ . Mục tiêu của ta là đi tìm một hằng số  $b$  để cực tiểu hóa  $\sum_i (x_i - b)^2$ .
  - Tìm một công thức nghiệm cho giá trị tối ưu của  $b$ .
  - Bài toán và nghiệm của nó có liên hệ như thế nào tới phân phối chuẩn?
2. Xây dựng công thức nghiệm cho bài toán tối ưu hóa hồi quy tuyến tính với bình phương sai số. Để đơn giản hơn, bạn có thể bỏ qua hệ số điều chỉnh  $b$  ra khỏi bài toán (chúng ta có thể thực hiện việc này bằng cách thêm vào một cột toàn giá trị một vào  $X$ ).
  - Viết bài toán tối ưu hóa theo ký hiệu ma trận-vector (xem tất cả các điểm dữ liệu như một ma trận và tất cả các giá trị mục tiêu như một vector).
  - Tính gradient của hàm mất mát theo  $w$ .
  - Tìm công thức nghiệm bằng cách giải phương trình gradient bằng không.
  - Khi nào phương pháp làm này tốt hơn so với sử dụng hạ gradient ngẫu nhiên? Khi nào phương pháp này không hoạt động?
3. Giả sử rằng mô hình nhiều điều khiển nhiều cộng  $\epsilon$  là phân phối mũ, nghĩa là  $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$ .

- Viết hàm đối log hợp lý của dữ liệu theo mô hình –  $\log P(Y | X)$ .
- Bạn có thể tìm ra nghiệm theo công thức không?
- Gợi ý là thuật toán hạ gradient ngẫu nhiên có thể giải quyết vấn đề này.
- Điều gì có thể sai ở đây (gợi ý - điều gì xảy ra gần điểm dừng khi chúng ta tiếp tục cập nhật các tham số). Bạn có thể sửa nó không?

### 5.1.6 Thảo luận

- Tiếng Anh<sup>71</sup>
- Tiếng Việt<sup>72</sup>

### 5.1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Ngọc Bảo Anh
- Nguyễn Văn Tâm
- Phạm Hồng Vinh
- Nguyễn Phan Hùng Thuận
- Vũ Hữu Tiệp
- Tạ H. Duy Nguyên
- Bùi Nhật Quân
- Lê Gia Thiên Bửu
- Lý Phi Long
- Nguyễn Minh Thư
- Tạ Đức Huy
- Minh Trí Nguyễn
- Trần Thị Hồng Hạnh
- Nguyễn Quang Hải
- Lê Thành Vinh

---

<sup>71</sup> <https://discuss.mxnet.io/t/2331>

<sup>72</sup> <https://forum.machinelearningcoban.com/c/d21>

## 5.2 Lập trình Hồi quy Tuyến tính từ đầu

Bây giờ bạn đã hiểu được điểm mấu chốt đằng sau thuật toán hồi quy tuyến tính, chúng ta đã có thể bắt đầu thực hành viết mã. Trong phần này, ta sẽ xây dựng lại toàn bộ kĩ thuật này từ đầu, bao gồm: pipeline dữ liệu, mô hình, hàm mất mát và phương pháp tối ưu hạ gradient. Vì các framework học sâu hiện đại có thể tự động hóa gần như tất cả các công đoạn ở trên, việc lập trình mọi thứ từ đầu chỉ để đảm bảo bạn biết rõ mình đang làm gì. Hơn nữa, việc hiểu rõ cách mọi thứ hoạt động sẽ giúp ta rất nhiều trong những lúc cần tùy chỉnh các mô hình, tự định nghĩa lại các tầng tính toán hay các hàm mất mát, v.v. Trong phần này, chúng ta chỉ dựa vào ndarray và autograd. Sau đó, chúng tôi sẽ giới thiệu một phương pháp triển khai chặt chẽ hơn, tận dụng các tính năng tuyệt vời của Gluon. Để bắt đầu, chúng ta cần khai báo một vài gói thư viện cần thiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
import random
npx.set_np()
```

### 5.2.1 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 nhiễu cộng. Nhiệm vụ của chúng ta là khôi phục các tham số của mô hình này bằng cách sử dụng một tập hợp hữu hạn các mẫu có trong tập dữ liệu đó. Chúng ta sẽ sử dụng dữ liệu ít chiều để thuận tiện cho việc minh họa. Trong đoạn mã sau, chúng ta đã tạo một tập dữ liệu chứa 1000 mẫu, mỗi mẫu bao gồm 2 đặc trưng được lấy ngẫu nhiên theo phân phối chuẩn. Do đó, tập dữ liệu tổng hợp của chúng ta sẽ là một đối tượng  $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$ .

Các tham số đúng để tạo tập dữ liệu sẽ là  $\mathbf{w} = [2, -3.4]^\top$  và  $b = 4.2$  và nhãn sẽ được tạo ra dựa theo mô hình tuyến tính với nhiễu  $\epsilon$ :

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon. \quad (5.2.1)$$

Bạn đọc có thể xem  $\epsilon$  như là sai số tiềm ẩn của phép đo trên các đặc trưng và các nhãn. Chúng ta sẽ mặc định các giả định tiêu chuẩn đều thỏa mãn và vì thế  $\epsilon$  tuân theo phân phối chuẩn với trung bình bằng 0. Để đơn giản, ta sẽ thiết lập độ lệch chuẩn của nó bằng 0.01. Đoạn mã nguồn sau sẽ tạo ra tập dữ liệu tổng hợp:

```
# Saved in the d2l package for later use
def synthetic_data(w, b, num_examples):
    """Generate y = X w + b + noise."""
    X = np.random.normal(0, 1, (num_examples, len(w)))
    y = np.dot(X, w) + b
    y += np.random.normal(0, 0.01, y.shape)
    return X, y

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = synthetic_data(true_w, true_b, 1000)
```

Lưu ý rằng mỗi hàng trong features chứa một điểm dữ liệu hai chiều và mỗi hàng trong labels chứa một giá trị mục tiêu một chiều (một số vô hướng).

```
print('features:', features[0], '\nlabel:', labels[0])
```

Bằng cách vẽ đồ thị phân tán với chiều thứ hai `features[:, 1]` và `labels`, ta có thể quan sát rõ mối tương quan tuyến tính giữa chúng.

```
d2l.set_figsize((3.5, 2.5))
d2l.plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```

### 5.2.2 Đọc từ Tập dữ liệu

Nhắc lại rằng việc huấn luyện mô hình bao gồm tách tập dữ liệu thành nhiều phần (các minibatch), lần lượt đọc từng phần của tập dữ liệu mẫu, và sử dụng chúng để cập nhật mô hình của chúng ta. Vì quá trình này rất cẩn thận để huấn luyện các giải thuật học máy, ta nên định nghĩa một hàm để trộn và truy xuất dữ liệu trong các minibatch một cách tiện lợi.

Ở đoạn mã dưới đây, chúng ta định nghĩa hàm `data_iter` để minh họa cho một cách lập trình chức năng này. Hàm này lấy kích thước một batch, một ma trận đặc trưng và một vector các nhãn rồi sinh ra các minibatch có kích thước `batch_size`. Mỗi minibatch gồm một tuple các đặc trưng 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)
    for i in range(0, num_examples, batch_size):
        batch_indices = np.array(
            indices[i: min(i + batch_size, num_examples)])
        yield features[batch_indices], labels[batch_indices]
```

Lưu ý rằng thông thường chúng ta muốn dùng các minibatch có kích thước phù hợp để tận dụng tài nguyên phần cứng GPU để xử lý song song hiệu quả nhất. Vì mỗi mẫu có thể được mô hình xử lý và tính đạo hàm riêng của hàm mất mát song song với nhau, GPU cho phép xử lý hàng trăm mẫu cùng lúc mà chỉ tốn thời gian hơn một chút so với xử lý một mẫu duy nhất.

Để hiểu hơn, chúng ta hãy chạy đoạn chương trình để đọc và in ra batch đầu tiên của mẫu dữ liệu. Kích thước của các đặc trưng trong mỗi minibatch cho ta biết kích thước của batch lần số lượng của các đặc trưng đầu vào. Tương tự, tập minibatch của các nhãn sẽ có kích thước theo `batch_size`.

```
batch_size = 10

for X, y in data_iter(batch_size, features, labels):
    print(X, '\n', y)
    break
```

Khi chạy iterator, ta lấy từng minibatch riêng biệt cho đến khi lấy hết bộ dữ liệu (Bạn hãy xem). Mặc dù sử dụng iterator như trên phục vụ tốt cho công tác giảng dạy, nó lại không phải là cách hiệu quả và có thể khiến chúng ta gặp nhiều rắc rối trong thực tế. Chẳng hạn, nó buộc ta phải nạp toàn bộ dữ liệu vào bộ nhớ và tốn rất nhiều thao tác truy cập bộ nhớ ngẫu nhiên. Các iterator trong Apache MXNet lại khá hiệu quả khi chúng có thể xử lý cả dữ liệu được lưu trữ trên tập tin lẩn trong các luồng dữ liệu.

### 5.2.3 Khởi tạo các Tham số Mô hình

Để tối ưu các tham số của dữ liệu bằng gradient, đầu tiên ta cần khởi tạo chúng. Trong đoạn mã dưới đây, ta khởi tạo các trọng số bằng cách lấy ngẫu nhiên các mẫu từ một phân phối chuẩn với giá trị trung bình bằng 0 và độ lệch chuẩn là 0.01, sau đó gán hệ số điều chỉnh  $b$  bằng 0.

```
w = np.random.normal(0, 0.01, (2, 1))
b = np.zeros(1)
```

Sau khi khởi tạo các tham số, bước tiếp theo là cập nhật chúng cho đến khi chúng ăn khớp với dữ liệu của ta đủ tốt. Mỗi lần cập nhật, ta tính gradient (đạo hàm nhiều biến) của hàm mất mát theo các tham số. Với gradient này, chúng ta có thể cập nhật mỗi tham số theo hướng giảm dần giá trị mất mát.

Vì không ai muốn tính gradient bằng tay (một việc rất nhàn chán và dễ sai sót), ta dùng chương trình để tính tự động gradient (autograd). Xem [Section 4.5](#) để biết thêm chi tiết. Nhắc lại từ mục tính vi phân tự động, để chỉ định hàm autograd lưu gradient của các biến số, ta cần gọi hàm `attach_grad`, cấp phát bộ nhớ để lưu giá trị gradient mong muốn.

```
w.attach_grad()
b.attach_grad()
```

### 5.2.4 Định nghĩa Mô hình

Tiếp theo, chúng ta cần định nghĩa mô hình dựa trên đầu vào và tham số liên quan tới đầu ra. Nhắc lại rằng để tính đầu ra của một mô hình tuyến tính, ta có thể đơn giản tính tích vô hướng ma trận-vector của các mẫu  $X$  và trọng số mô hình  $w$ , sau đó thêm vào hệ số điều chỉnh  $b$  cho từng mẫu. Ở đây, `np.dot(X, w)` là một vector trong khi  $b$  là một số vô hướng. Nhắc lại rằng khi tính tổng vector và số vô hướng, thì số vô hướng sẽ được cộng vào từng phần tử của vector.

```
# Saved in the d2l package for later use
def linreg(X, w, b):
    return np.dot(X, w) + b
```

### 5.2.5 Định nghĩa Hàm Mất mát

Để cập nhật mô hình ta phải tính gradient của hàm mất mát, vậy nên ta cần định nghĩa hàm mất mát trước tiên. Chúng ta sẽ sử dụng hàm mất mát bình phương (SE) như đã trình bày ở phần trước đó. Trên thực tế, chúng ta cần chuyển đổi giá trị nhãn thật  $y$  sang kích thước của giá trị dự đoán  $y_{\text{hat}}$ . Hàm dưới đây sẽ trả về kết quả có kích thước tương đương với kích thước của  $y_{\text{hat}}$ .

```
# Saved in the d2l package for later use
def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

## 5.2.6 Định nghĩa Thuật toán Tối ưu

Như đã thảo luận ở mục trước, hồi quy tuyến tính có một **nghiệm dạng đóng**<sup>73</sup>. Tuy nhiên, đây không phải là một cuốn sách về hồi quy tuyến tính, mà là về Học sâu. Vì không một mô hình nào khác được trình bày trong cuốn sách này có thể giải được bằng phương pháp phân tích, chúng tôi sẽ nhân cơ hội này để giới thiệu với các bạn ví dụ đầu tiên về hạ gradient ngẫu nhiên (*stochastic gradient descent – SGD*).

Sử dụng một batch được lấy ngẫu nhiên từ tập dữ liệu tại mỗi bước, chúng ta sẽ ước tính được gradient của măt măt theo các tham số. Tiếp đó, ta sẽ cập nhật các tham số (với một lượng nhỏ) theo chiều hướng làm giảm sự măt măt. Nhớ lại từ [Section 4.5](#) rằng sau khi chúng ta gọi backward, mỗi tham số (param) sẽ có gradient của nó lưu ở param.grad. Đoạn mã sau áp dụng cho việc cập nhật SGD, đưa ra một bộ các tham số, tốc độ học và kích thước batch. Kích thước của bước cập nhật được xác định bởi tốc độ học lr. Bởi vì các măt măt được tính dựa trên tổng các mău của batch, ta chuẩn hóa kích thước bước cập nhật theo kích thước batch (batch\_size), sao cho độ lớn của một bước cập nhật thông thường không phụ thuộc vào kích thước batch.

```
# Saved in the d2l package for later use
def sgd(params, lr, batch_size):
    for param in params:
        param[:] = param - lr * param.grad / batch_size
```

## 5.2.7 Huấn luyện

Bây giờ, sau khi đã có tất cả các thành phần, chúng ta đã sẵn sàng để viết vòng lặp huấn luyện. Quan trọng nhất là bạn phải hiểu được rõ đoạn mã này bởi vì việc huấn luyện gần như tương tự thế này sẽ được lặp lại nhiều lần trong suốt quá trình chúng ta tìm hiểu và lập trình các thuật toán học sâu.

Trong mỗi vòng lặp, đầu tiên chúng ta sẽ lấy ra các minibatch dữ liệu và chạy nó qua mô hình để lấy ra tập kết quả dự đoán. Sau khi tính toán sự măt măt, chúng ta dùng hàm backward để bắt đầu lan truyền ngược qua mạng lưới, lưu trữ các gradient tương ứng với mỗi tham số trong từng thuộc tính .grad của chúng. Cuối cùng, chúng ta sẽ dùng thuật toán tối ưu sgd để cập nhật các tham số của mô hình. Từ đầu chúng ta đã đặt kích thước batch batch\_size là 10, vậy nên măt măt 1 cho mỗi minibatch có kích thước là (10, 1).

Tóm lại, chúng ta sẽ thực thi vòng lặp sau:

- Khởi tạo bộ tham số ( $\mathbf{w}, b$ )
- Lặp lại cho tới khi hoàn thành
  - Tính gradient  $\mathbf{g} \leftarrow \partial_{(\mathbf{w}, b)} \frac{1}{B} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$
  - Cập nhật bộ tham số  $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Trong đoạn mã dưới đây, 1 là một vector của các măt măt của từng mău trong minibatch. Vì 1 không phải là biến vô hướng, chạy 1.backward() sẽ cộng các phần tử trong 1 để tạo ra một biến mới và sau đó mới tính gradient.

Với mỗi epoch, chúng ta sẽ lặp qua toàn bộ tập dữ liệu (sử dụng hàm data\_iter) cho đến khi đi qua toàn bộ mọi mău trong tập huấn luyện (giả định rằng số mău chia hết cho kích thước batch). Số epoch num\_epochs và tốc độ học lr đều là siêu tham số, mà chúng ta đặt ở đây là tương ứng 3 và

<sup>73</sup> [https://vi.wikipedia.org/wiki/Biểu\\_thức\\_dạng\\_đóng](https://vi.wikipedia.org/wiki/Biểu_thức_dạng_đóng)

0.03. Không may thay, việc lựa chọn siêu tham số thường không đơn giản và đòi hỏi sự điều chỉnh qua nhiều lần thử và sai. Hiện tại chúng ta sẽ bỏ qua những chi tiết này nhưng sẽ bàn lại về chúng tại chương Section 13.

```
lr = 0.03 # Learning rate
num_epochs = 3 # Number of iterations
net = linreg # Our fancy linear model
loss = squared_loss # 0.5 (y-y')^2

for epoch in range(num_epochs):
    # Assuming the number of examples can be divided by the batch size, all
    # the examples in the training dataset are used once in one epoch
    # iteration. The features and tags of minibatch examples are given by X
    # and y respectively
    for X, y in data_iter(batch_size, features, labels):
        with autograd.record():
            l = loss(net(X, w, b), y) # Minibatch loss in X and y
            l.backward() # Compute gradient on l with respect to [w, b]
            sgd([w, b], lr, batch_size) # Update parameters using their gradient
    train_l = loss(net(features, w, b), labels)
    print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))
```

Trong trường hợp này, bởi vì chúng ta tự tổng hợp dữ liệu nên đã biết chính xác giá trị đúng của các tham số. Vì vậy, chúng ta có thể đánh giá sự thành công của việc huấn luyện bằng cách so sánh giá trị đúng của các tham số với những giá trị học được thông qua quá trình huấn luyện. Quả thật giá trị các tham số học được và giá trị chính xác của các tham số rất gần với nhau.

```
print('Error in estimating w', true_w - w.reshape(true_w.shape))
print('Error in estimating b', true_b - b)
```

Lưu ý là chúng ta không nên ngộ nhận rằng giá trị của các tham số có thể được khôi phục một cách chính xác. Điều này chỉ xảy ra với một số bài toán đặc biệt: các bài toán tối ưu lồi chặt với lượng dữ liệu “đủ” để đảm bảo rằng các mẫu nhiễu vẫn cho phép chúng ta khôi phục được các tham số. Trong hầu hết các trường hợp, điều này *không* xảy ra. Trên thực tế, các tham số của một mạng học sâu hiếm khi nào giống nhau (hoặc thậm chí là gần nhau) giữa hai lần chạy riêng biệt, trừ khi tất cả các điều kiện đều giống hệt nhau, bao gồm cả thứ tự mà dữ liệu được duyệt qua. Tuy nhiên, trong học máy, chúng ta thường ít quan tâm đến việc khôi phục chính xác giá trị của các tham số, mà quan tâm nhiều hơn đến bộ tham số nào sẽ dẫn tới việc dự đoán chính xác hơn. May mắn thay, thậm chí với những bài toán tối ưu khó, giải thuật *gradient* ngẫu nhiên thường có thể tìm ra các lời giải đủ tốt, do một thực tế là đối với các mạng học sâu, có thể tồn tại nhiều bộ tham số dẫn đến việc dự đoán chính xác.

### 5.2.8 Tóm tắt

Chúng ta đã thấy cách một mạng sâu được thực thi và tối ưu hóa từ đầu chỉ với `ndarray` và `autograd` mà không cần định nghĩa các tầng, các thuật toán tối ưu đặc biệt, v.v. Điều này chỉ mới chạm đến bề mặt của những gì mà ta có thể làm. Trong các phần sau, chúng tôi sẽ mô tả các mô hình khác dựa trên những khái niệm vừa được giới thiệu cũng như cách thực thi chúng một cách chính xác hơn.

### 5.2.9 Bài tập

- Điều gì sẽ xảy ra nếu chúng ta khởi tạo các trọng số  $w = 0$ . Liệu thuật toán sẽ vẫn hoạt động chứ?
- Giả sử rằng bạn là [Georg Simon Ohm](#)<sup>74</sup> và bạn đang cố gắng tìm ra một mô hình giữa điện áp và dòng điện. Bạn có thể sử dụng autograd để học các tham số cho mô hình của bạn không?
- Bạn có thể sử dụng [Luật Planck](#)<sup>75</sup> để xác định nhiệt độ của một vật thể sử dụng mật độ năng lượng quang phổ không?
- Những vấn đề gặp phải nếu muốn mở rộng autograd đến các đạo hàm bậc hai? Cần sửa lại như thế nào?
- Tại sao hàm reshape lại cần thiết trong hàm squared\_loss?
- Thử nghiệm các tốc độ học khác nhau để kiểm tra mức độ giảm nhanh của giá trị hàm mất mát giảm.
- Nếu số lượng mẫu không thể chia hết cho kích thước batch, thì hàm data\_iter sẽ xử lý như thế nào?

### 5.2.10 Thảo luận

- [Tiếng Anh](#)<sup>76</sup>
- [Tiếng Việt](#)<sup>77</sup>

### 5.2.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lý Phi Long
- Vũ Hữu Tiệp
- Phạm Hồng Vinh
- Nguyễn Văn Tâm
- Nguyễn Cảnh Thưởng
- Nguyễn Lê Quang Nhật
- Dương Nhật Tân
- Nguyễn Minh Thư
- Nguyễn Trường Phát
- Đinh Minh Tân
- Trần Thị Hồng Hạnh

<sup>74</sup> [https://en.wikipedia.org/wiki/Georg\\_Ohm](https://en.wikipedia.org/wiki/Georg_Ohm)

<sup>75</sup> [https://en.wikipedia.org/wiki/Planck's\\_law](https://en.wikipedia.org/wiki/Planck's_law)

<sup>76</sup> <https://discuss.mxnet.io/t/2332>

<sup>77</sup> <https://forum.machinelearningcoban.com/c/d21>

- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Mai Hoàng Long

## 5.3 Cách lập trình súc tích Hồi quy Tuyến tính

Sự quan tâm nhiệt thành và rộng khắp với học sâu trong những năm gần đây đã tạo cảm hứng cho các công ty, học viện và những người đam mê tới học sâu phát triển nhiều framework mã nguồn mở hoàn thiện, giúp tự động hóa các công việc lặp đi lặp lại trong quá trình triển khai các thuật toán học dựa trên gradient. Trong chương trước, chúng ta chỉ dựa vào (i) ndarray để lưu dữ liệu và thực hiện tính toán đại số tuyến tính; và (ii) autograd để thực hiện tính đạo hàm. Trên thực tế, do các iterator dữ liệu, các hàm mất mát, các bộ tối ưu và các tầng của mạng nơ-ron (thậm chí là toàn bộ kiến trúc) rất phổ biến, các thư viện hiện đại đã cài đặt sẵn những thành phần này cho chúng ta.

Mục này sẽ hướng dẫn bạn cách để xây dựng mô hình hồi quy tuyến tính trong phần [Section 5.2](#) một cách súc tích với Gluon.

### 5.3.1 Tạo Tập dữ liệu

Chúng ta bắt đầu bằng việc tạo một tập dữ liệu như ở mục trước.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()

true_w = np.array([2, -3.4])
true_b = 4.2
features, labels = d2l.synthetic_data(true_w, true_b, 1000)
```

### 5.3.2 Đọc tập dữ liệu

Thay vì tự viết iterator riêng để đọc dữ liệu thì ta có thể gọi mô-đun data của Gluon để xử lý việc này. Bước đầu tiên sẽ là khởi tạo một `ArrayDataset`. Hàm tạo của đối tượng này sẽ lấy một hoặc nhiều ndarray làm đối số. Tại đây, ta truyền vào hàm hai đối số là `features` và `labels`. Kế tiếp, ta sử dụng `ArrayDataset` để khởi tạo một `DataLoader`, lớp này yêu cầu ta truyền vào một giá trị `batch_size` và giá trị Boolean `shuffle` để cho biết chúng ta có muốn `DataLoader` xáo trộn dữ liệu trên mỗi epoch (một lần duyệt qua toàn bộ tập dữ liệu) hay không.

```
# Saved in the d2l package for later use
def load_array(data_arrays, batch_size, is_train=True):
    """Construct a Gluon data loader"""
    dataset = gluon.data.ArrayDataset(*data_arrays)
    return gluon.data.DataLoader(dataset, batch_size, shuffle=is_train)

batch_size = 10
data_iter = load_array((features, labels), batch_size)
```

Bây giờ, ta có thể sử dụng `data_iter` theo cách tương tự như cách ta gọi hàm `data_iter` trong phần trước. Để biết rằng nó có hoạt động được hay không, ta có thể thử đọc và in ra minibatch đầu tiên.

```
for X, y in data_iter:  
    print(X, '\n', y)  
    break
```

### 5.3.3 Định nghĩa Mô hình

Khi ta lập trình hồi quy tuyến tính từ đầu (trong Section 5.2), ta đã định nghĩa rõ ràng các tham số của mô hình và lập trình các tính toán cho giá trị đầu ra sử dụng các phép toán đại số tuyến tính cơ bản. Bạn *nên* biết cách để làm được điều này. Nhưng một khi mô hình trở nên phức tạp hơn và đồng thời khi bạn phải làm điều này gần như hàng ngày, bạn sẽ thấy vui mừng khi có sự hỗ trợ từ các thư viện. Tình huống này tương tự như việc lập trình blog của riêng bạn lại từ đầu. Làm điều này một hoặc hai lần thì sẽ bổ ích và mang tính hướng dẫn, nhưng bạn sẽ trở thành một nhà phát triển web “khó ở” nếu mỗi khi cần một trang blog bạn lại phải dành ra cả một tháng chỉ để phát triển lại từ đầu.

Đối với những tác vụ tiêu chuẩn, chúng ta có thể sử dụng các tầng đã được định nghĩa trước trong Gluon, điều này cho phép chúng ta tập trung vào những tầng được dùng để xây dựng mô hình hơn là việc phải tập trung vào cách lập trình các tầng đó. Để định nghĩa một mô hình tuyến tính, đầu tiên chúng ta cần nhập vào mô-đun `nn`, giúp ta định nghĩa một lượng lớn các tầng trong mạng nơ-ron (lưu ý rằng “`nn`” là chữ viết tắt của “neural network”). Đầu tiên ta sẽ định nghĩa một biến mô hình là `net`, tham chiếu đến một thực thể của lớp `Sequential`. Trong Gluon, `Sequential` định nghĩa một lớp chứa nhiều tầng được liên kết với nhau. Khi nhận được dữ liệu đầu vào, `Sequential` sẽ truyền dữ liệu vào tầng đầu tiên, kết quả đầu ra từ đó trở thành đầu vào của tầng thứ hai và cứ tiếp tục như thế ở các tầng kế tiếp. Trong ví dụ tiếp theo, mô hình chúng ta chỉ có duy nhất một tầng, vì vậy không nhất thiết phải sử dụng `Sequential`. Tuy nhiên vì hầu hết các mô hình chúng ta gặp phải trong tương lai đều có nhiều tầng, do đó dù sao cũng nên dùng để làm quen với quy trình làm việc tiêu chuẩn nhất.

```
from mxnet.gluon import nn  
net = nn.Sequential()
```

Hãy cùng nhớ lại kiến trúc của mạng đơn tầng như đã trình bày tại Fig. 5.3.1. Tầng đó được gọi là *kết nối đầy đủ* bởi vì mỗi đầu vào được kết nối lần lượt với từng đầu ra bằng một phép nhân ma trận với vector. Trong Gluon, tầng kết nối đầy đủ được định nghĩa trong lớp `Dense`. Bởi vì chúng ta chỉ mong xuất ra một số vô hướng duy nhất, nên ta gán giá trị là 1.

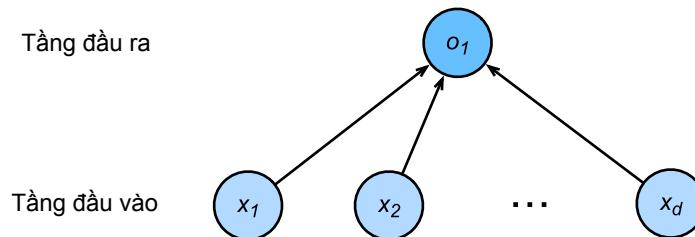


Fig. 5.3.1: Hồi quy tuyến tính là một mạng nơ-ron đơn tầng.

```
net.add(nn.Dense(1))
```

Để thuận tiện, điều đáng chú ý là Gluon không yêu cầu chúng ta chỉ định kích thước đầu vào mỗi tầng. Nên tại đây, chúng ta không cần thiết cho Gluon biết có bao nhiêu đầu vào cho mỗi tầng tuyến tính. Khi chúng ta cố gắng truyền dữ liệu qua mô hình lần đầu tiên, ví dụ: khi chúng ta thực hiện `net(X)` sau đó, Gluon sẽ tự động suy ra số lượng đầu vào cho mỗi tầng. Chúng ta sẽ mô tả cách hoạt động của cơ chế này một cách chi tiết hơn trong chương “Tính toán trong Học sâu”.

#### 5.3.4 Khởi tạo Tham số Mô hình

Trước khi sử dụng `net`, chúng ta cần phải khởi tạo tham số cho mô hình, chẳng hạn như trọng số và hệ số điều chỉnh trong mô hình hồi quy tuyến tính. Chúng ta sẽ nhập mô-đun `initializer` từ MXNet. Mô-đun này cung cấp nhiều phương thức khác nhau để khởi tạo tham số cho mô hình. Gluon cho phép dùng `init` như một cách ngắn gọn (viết tắt) để truy cập đến gói `initializer`. Bằng cách gọi `init.Normal(sigma=0.01)`, chúng ta sẽ khởi tạo ngẫu nhiên các trọng số từ một phân phối chuẩn với trung bình bằng 0 và độ lệch chuẩn bằng 0.01. Mặc định, tham số *hệ số điều chỉnh* sẽ được khởi tạo bằng không. Cả hai vector trọng số và hệ số điều chỉnh sẽ có gradient kèm theo.

```
from mxnet import init  
net.initialize(init.Normal(sigma=0.01))
```

Đoạn mã nguồn trên trông khá đơn giản nhưng bạn đọc hãy chú ý một vài điểm khác thường ở đây. Chúng ta khởi tạo các tham số cho một mạng mà thậm chí Gluon chưa hề biết số chiều của đầu vào là bao nhiêu! Nó có thể là 2 trong trường hợp của chúng ta nhưng cũng có thể là 2000. Gluon khiến chúng ta không cần bận tâm về điều này bởi ở hậu trường, quá trình khởi tạo thực sự vẫn đang bị trì hoãn. Quá trình khởi tạo thực sự chỉ bắt đầu khi chúng ta truyền dữ liệu vào mạng lần đầu tiên. Hãy ghi nhớ rằng, do các tham số chưa thực sự được khởi tạo, chúng ta không thể truy cập hoặc thao tác với chúng.

#### 5.3.5 Định nghĩa Hàm mất mát

Trong Gluon, mô-đun `loss` định nghĩa các hàm mất mát khác nhau. Chúng ta sẽ sử dụng mô-đun `loss` được thêm vào dưới tên gọi là `gloss`, để tránh nhầm lẫn nó với biến đang giữ hàm mất mát mà ta đã chọn. Trong ví dụ này, chúng ta sẽ sử dụng triển khai trong Gluon của mất mát bình phương (`L2Loss`).

```
from mxnet.gluon import loss as gloss  
loss = gloss.L2Loss() # The squared loss is also known as the L2 norm loss
```

### 5.3.6 Định nghĩa Thuật toán Tối ưu

Minibatch SGD và các biến thể liên quan đều là các công cụ chuẩn cho việc tối ưu hóa mạng nơ-ron, vì vậy Gluon có hỗ trợ SGD cùng với một số biến thể của thuật toán này thông qua lớp Trainer. Khi khởi tạo lớp Trainer, ta cần chỉ định các tham số để tối ưu hóa (có thể lấy từ mạng thông qua `net.collect_params()`), thuật toán tối ưu muốn sử dụng (`sgd`) và một từ điển gồm các siêu tham số cần thiết cho thuật toán tối ưu. SGD chỉ yêu cầu giá trị của `learning_rate`, (ở đây chúng ta đặt nó bằng 0.03).

```
from mxnet import gluon
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

### 5.3.7 Huấn luyện

Bạn có thể thấy rằng việc biểu diễn mô hình thông qua Gluon đòi hỏi tương đối ít dòng lệnh. Chúng ta không cần phải khởi tạo từng tham số riêng lẻ, định nghĩa hàm mất mát hay lập trình thuật toán hạ gradient ngẫu nhiên. Lợi ích mà Gluon mang lại sẽ rất lớn khi chúng ta bắt đầu làm việc với những mô hình phức tạp hơn. Tuy nhiên, một khi ta có các mảnh ghép cơ bản, vòng lặp huấn luyện lại rất giống với những gì ta đã làm khi lập trình mọi thứ từ đầu.

Nhắc lại rằng: với số lượng epoch nhất định, trong mỗi epoch chúng ta sẽ duyệt qua toàn bộ tập dữ liệu (`train_data`), lần lượt lấy từng minibatch chứa dữ liệu đầu vào và các nhãn gốc tương ứng. Đối với mỗi minibatch, chúng ta cần tuân thủ theo trình tự sau:

- Đưa ra dự đoán bằng cách gọi `net(X)` và tính giá trị mất mát 1 (lượt truyền xuôi).
- Tính gradient bằng cách gọi `l.backward()` (lượt truyền ngược).
- Cập nhật các tham số của mô hình bằng cách gọi bộ tối ưu SGD (chú ý rằng `trainer` đã biết các tham số cần tối ưu, nên ta chỉ cần truyền thêm kích thước của minibatch).

Ngoài ra, ta tính giá trị mất mát sau mỗi epoch và in nó ra màn hình để giám sát tiến trình.

```
num_epochs = 3
for epoch in range(1, num_epochs + 1):
    for X, y in data_iter:
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    l = loss(net(features), labels)
    print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
```

Dưới đây, ta so sánh các tham số của mô hình đã được học thông qua việc huấn luyện trên tập dữ liệu hữu hạn với các tham số được dùng để tạo ra tập dữ liệu. Để truy cập các tham số trong Gluon, trước hết ta truy cập tầng ta quan tâm thông qua biến `net`, sau đó truy cập trọng số (`weight`) và hệ số điều chỉnh (`bias`) của tầng đó. Để truy cập giá trị tham số dưới dạng một mảng `ndarray`, ta sử dụng phương thức `data`. Giống với phiên bản lập trình từ đầu của chúng ta, các tham số ước lượng có giá trị gần với giá trị chính xác của chúng.

```
w = net[0].weight.data()
print('Error in estimating w', true_w.reshape(w.shape) - w)
```

(continues on next page)

```
b = net[0].bias.data()
print('Error in estimating b', true_b - b)
```

### 5.3.8 Tóm tắt

- Sử dụng Gluon giúp việc lập trình các mô hình trở nên ngắn gọn hơn rất nhiều.
- Trong Gluon, mô-đun data cung cấp các công cụ để xử lý dữ liệu, mô-đun nn định nghĩa một lượng lớn các tầng cho mạng nơ-ron, và mô-đun loss cho phép ta thiết lập nhiều hàm mất mát phổ biến.
- Mô-đun initializer của MXNet cung cấp nhiều phương thức khác nhau để khởi tạo tham số cho mô hình.
- Kích thước và dung lượng lưu trữ của các tham số sẽ được suy ra một cách tự động (nhưng cần cẩn thận tránh truy cập các tham số trước khi chúng được khởi tạo).

### 5.3.9 Bài tập

1. Nếu thay thế `l = loss(output, y)` bằng `l = loss(output, y).mean()`, chúng ta cần đổi `trainer.step(batch_size)` thành `trainer.step(1)` để phần mã nguồn này hoạt động giống như trước. Tại sao lại thế?
2. Xem lại tài liệu về MXNet để biết các hàm mất mát và các phương thức khởi tạo được cung cấp trong hai mô-đun gluon.loss và init. Hãy thay thế hàm mất mát đang sử dụng bằng hàm mất mát Huber (*Huber loss*).
3. Làm thế nào để truy cập gradient của `dense.weight`?

### 5.3.10 Thảo luận

- Tiếng Anh<sup>78</sup>
- Tiếng Việt<sup>79</sup>

### 5.3.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Thị Hồng Hạnh
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Lý Phi Long
- Phạm Đăng Khoa

<sup>78</sup> <https://discuss.mxnet.io/t/2333>

<sup>79</sup> <https://forum.machinelearningcoban.com/c/d21>

- Lê Khắc Hồng Phúc
- Dương Nhật Tân
- Nguyễn Văn Tâm
- Bùi Nhật Quân
- Nguyễn Mai Hoàng Long

## 5.4 Hồi quy Softmax

Trong Section 5.1, chúng ta đã giới thiệu về hồi quy tuyến tính, từ việc tự xây dựng mô hình hồi quy tuyến tính từ đầu tại Section 5.2 cho đến xây dựng mô hình hồi quy tuyến tính với Gluon thực hiện phần việc nặng nhọc tại Section 5.3.

Hồi quy là công cụ đắc lực có thể sử dụng khi ta muốn trả lời câu hỏi *bao nhiêu?*. Nếu bạn muốn dự đoán một ngôi nhà sẽ được bán với giá bao nhiêu tiền (*Đô la*), hay số trận thắng mà một đội bóng có thể đạt được, hoặc số ngày một bệnh nhân phải điều trị nội trú trước khi được xuất viện, thì có lẽ bạn đang cần một mô hình hồi quy.

Trong thực tế, chúng ta thường quan tâm đến việc phân loại hơn: không phải câu hỏi *bao nhiêu?* mà là *loại nào?*

- Email này có phải thư rác hay không?
- Khách hàng này nhiều khả năng *đăng ký* hay *không đăng ký* một dịch vụ thuê bao?
- Hình ảnh này mô tả một con lừa, một con chó, một con mèo hay một con gà trống?
- Bộ phim nào có khả năng cao nhất được Aston xem tiếp theo?

Thông thường, những người làm về học máy dùng từ *phân loại* để mô tả đôi chút sự khác nhau giữa hai bài toán: (i) ta chỉ quan tâm đến việc gán *cứng* một danh mục cho mỗi ví dụ: là chó, là gà, hay là mèo?; và (ii) ta muốn gán *mềm* tất cả các danh mục cho mỗi ví dụ, tức đánh giá xác suất một ví dụ rơi vào từng danh mục khả dĩ: là chó (92%), là gà (1%), là mèo (7%). Sự khác biệt này thường không rõ ràng, một phần bởi vì thông thường ngay cả khi chúng ta chỉ quan tâm đến việc gán cứng, chúng ta vẫn sử dụng các mô hình thực hiện các phép gán mềm.

### 5.4.1 Bài toán Phân loại

Hãy khởi động với một bài toán phân loại hình ảnh đơn giản. Ở đây, mỗi đầu vào là một ảnh xám có kích thước  $2 \times 2$ . Bằng cách biểu diễn mỗi giá trị điểm ảnh bởi một số vô hướng, ta thu được bốn đặc trưng  $x_1, x_2, x_3, x_4$ . Hơn nữa, giả sử rằng mỗi hình ảnh đều thuộc về một trong các danh mục “mèo”, “gà” và “chó”.

Tiếp theo, ta cần phải chọn cách biểu diễn nhãn. Ta có hai cách làm hiển nhiên. Cách tự nhiên nhất có lẽ là chọn  $y \in \{1, 2, 3\}$  lần lượt ứng với {chó, mèo, gà}. Đây là một cách *lưu trữ* thông tin tuyệt vời trên máy tính. Nếu các danh mục có một thứ tự tự nhiên giữa chúng, chẳng hạn như {trẻ sơ sinh, trẻ tập đi, thiếu niên, thanh niên, người trưởng thành, người cao tuổi}, sẽ là tự nhiên hơn nếu coi bài toán này là một bài toán hồi quy và nhãn sẽ được giữ nguyên dưới dạng số.

Nhưng nhìn chung các lớp của bài toán phân loại không tuân theo một trật tự tự nhiên nào. May mắn thay, các nhà thông kê từ lâu đã tìm ra một cách đơn giản để có thể biểu diễn dữ liệu danh mục: *biểu diễn one-hot*. Biểu diễn one-hot là một vector với số lượng thành phần bằng số danh mục

mà ta có. Thành phần tương ứng với từng danh mục cụ thể sẽ được gán giá trị 1 và tất cả các thành phần khác sẽ được gán giá trị 0.

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}. \quad (5.4.1)$$

Trong trường hợp này,  $y$  sẽ là một vector 3 chiều, với  $(1, 0, 0)$  tương ứng với “mèo”,  $(0, 1, 0)$  ứng với “gà” và  $(0, 0, 1)$  ứng với “chó”.

## Kiến trúc mạng

Để tính xác suất có điều kiện ứng với mỗi lớp, chúng ta cần một mô hình có nhiều đầu ra với một đầu ra cho mỗi lớp. Để phân loại với các mô hình tuyến tính, chúng ta cần số hàm tuyến tính tương đương số đầu ra. Mỗi đầu ra sẽ tương ứng với hàm tuyến tính của chính nó. Trong trường hợp này, vì có 4 đặc trưng và 3 đầu ra, chúng ta sẽ cần 12 số vô hướng để thể hiện các trọng số, ( $w$  với các chỉ số dưới) và 3 số vô hướng để thể hiện các hệ số điều chỉnh ( $b$  với các chỉ số dưới). Chúng ta sẽ tính ba logits,  $o_1, o_2$ , và  $o_3$ , cho mỗi đầu vào:

$$\begin{aligned} o_1 &= x_1 w_{11} + x_2 w_{12} + x_3 w_{13} + x_4 w_{14} + b_1, \\ o_2 &= x_1 w_{21} + x_2 w_{22} + x_3 w_{23} + x_4 w_{24} + b_2, \\ o_3 &= x_1 w_{31} + x_2 w_{32} + x_3 w_{33} + x_4 w_{34} + b_3. \end{aligned} \quad (5.4.2)$$

Chúng ta có thể mô tả phép tính này với biểu đồ mạng nơ-ron được thể hiện trong Fig. 5.4.1. Như hồi quy tuyến tính, hồi quy softmax cũng là một mạng nơ-ron đơn tầng. Và vì sự tính toán của mỗi đầu ra,  $o_1, o_2$ , và  $o_3$ , phụ thuộc vào tất cả đầu vào,  $x_1, x_2, x_3$ , và  $x_4$ , tầng đầu ra của hồi quy softmax cũng có thể được xem như một tầng kết nối đầy đủ.

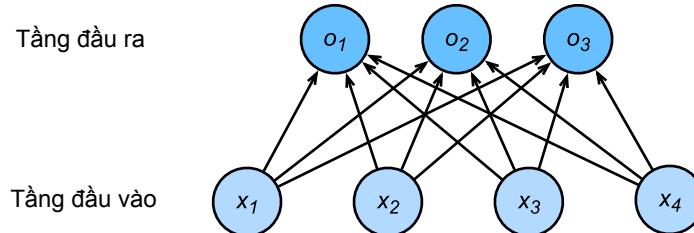


Fig. 5.4.1: Hồi quy softmax là một mạng nơ-ron đơn tầng

Để biểu diễn mô hình gọn hơn, chúng ta có thể sử dụng ký hiệu đại số tuyến tính. Ở dạng vector, ta có  $\mathbf{o} = \mathbf{Wx} + \mathbf{b}$ , một dạng phù hợp hơn cho cả toán và lập trình. Chú ý rằng chúng ta đã tập hợp tất cả các trọng số vào một ma trận  $3 \times 4$  và với một mẫu cho trước  $\mathbf{x}$ , các đầu ra được tính bởi tích ma trận-vector của các trọng số và đầu vào cộng với vector hệ số điều chỉnh  $\mathbf{b}$ .

## Hàm Softmax

Chúng ta sẽ xem các giá trị đầu ra của mô hình là các giá trị xác suất. Ta sẽ tối ưu hóa các tham số của mô hình sao cho khả năng xuất hiện dữ liệu quan sát được là cao nhất. Sau đó, ta sẽ đưa ra dự đoán bằng cách đặt ngưỡng xác suất, ví dụ dự đoán nhãn đúng là nhãn có xác suất cao nhất (dùng hàm *argmax*).

Nói một cách chính quy hơn, ta mong muốn diễn dịch kết quả  $\hat{y}_k$  là xác suất để một điểm dữ liệu cho trước thuộc về một lớp  $k$  nào đó. Sau đó, ta có thể chọn lớp cho điểm đó tương ứng với giá trị

lớn nhất mà mô hình dự đoán  $\text{argmax}_k y_k$ . Ví dụ, nếu  $\hat{y}_1, \hat{y}_2$  và  $\hat{y}_3$  lần lượt là 0.1, 0.8, and 0.1, thì ta có thể dự đoán điểm đó thuộc lớp số 2 là “gà” (ứng với trong ví dụ trước).

Bạn có thể muốn đề xuất rằng ta lấy trực tiếp logit  $o$  làm đầu ra mong muốn. Tuy nhiên, sẽ có vấn đề khi coi kết quả trả về trực tiếp từ tầng tuyến tính như là các giá trị xác suất. Lý do là không có bất cứ điều kiện nào để ràng buộc tổng của những con số này bằng 1. Hơn nữa, tùy thuộc vào đầu vào mà ta có thể nhận được giá trị âm. Các lý do trên khiến kết quả của tầng tuyến tính vi phạm vào các tiêu đề cơ bản của xác suất đã được nhắc đến trong [Section 4.6](#).

Để có thể diễn dịch kết quả đầu ra là xác suất, ta phải đảm bảo rằng các kết quả không âm và tổng của chúng phải bằng 1 (điều này phải đúng trên cả dữ liệu mới). Hơn nữa, ta cần một hàm mục tiêu trong quá trình huấn luyện để cho mô hình có thể ước lượng xác suất một cách chính xác. Trong tất cả các trường hợp, khi kết quả phân lớp cho ra xác suất là 0.5 thì ta hy vọng phân nửa số mẫu đó *thực sự* thuộc về đúng lớp được dự đoán. Đây được gọi là *hiệu chuẩn*.

*Hàm softmax*, được phát minh vào năm 1959 bởi nhà khoa học xã hội R Duncan Luce với chủ đề *mô hình lựa chọn*, thỏa mãn chính xác những điều trên. Để biến đổi kết quả logit thành kết quả không âm và có tổng là 1, trong khi vẫn giữ tính chất khả vi, đầu tiên ta cần lấy hàm mũ cho từng logit (để chắc chắn chúng không âm) và sau đó ta chia cho tổng của chúng (để chắc rằng tổng của chúng luôn bằng 1).

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}) \quad \text{tại} \quad \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}. \quad (5.4.3)$$

Dễ thấy rằng  $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$  với  $0 \leq \hat{y}_i \leq 1$  với mọi  $i$ . Do đó,  $\hat{\mathbf{y}}$  là phân phối xác suất phù hợp và các giá trị của  $\hat{\mathbf{y}}$  có thể được hiểu theo đó. Lưu ý rằng hàm softmax không thay đổi thứ tự giữa các logit và do đó ta vẫn có thể chọn ra lớp phù hợp nhất bằng cách:

$$\hat{i}(\mathbf{o}) = \underset{i}{\text{argmax}} o_i = \underset{i}{\text{argmax}} \hat{y}_i. \quad (5.4.4)$$

Các logit  $\mathbf{o}$  đơn giản chỉ là các giá trị trước khi cho qua hàm softmax để xác định xác suất thuộc về mỗi mục. Tóm tắt lại, ta có ký hiệu dưới dạng vector như sau:  $\mathbf{o}^{(i)} = \mathbf{Wx}^{(i)} + \mathbf{b}$ , với  $\hat{\mathbf{y}}^{(i)} = \text{softmax}(\mathbf{o}^{(i)})$ .

## Vector hóa Minibatch

Để cải thiện hiệu suất tính toán và tận dụng GPU, ta thường phải thực hiện các phép tính vector cho các minibatch dữ liệu. Giả sử, ta có một minibatch  $\mathbf{X}$  của mẫu với số chiều  $d$  và kích thước batch là  $n$ . Thêm vào đó, chúng ta có  $q$  lớp đầu ra. Như vậy, minibatch đặc trưng  $\mathbf{X}$  sẽ thuộc  $\mathbb{R}^{n \times d}$ , trọng số  $\mathbf{W} \in \mathbb{R}^{d \times q}$ , và độ chêch sẽ thỏa mãn  $\mathbf{b} \in \mathbb{R}^q$ .

$$\begin{aligned} \mathbf{O} &= \mathbf{WX} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \text{softmax}(\mathbf{O}). \end{aligned} \quad (5.4.5)$$

Việc tăng tốc diễn ra chủ yếu tại tích ma trận - ma trận  $\mathbf{WX}$  so với tích ma trận - vector nếu chúng ta xử lý từng mẫu một. Bản thân softmax có thể được tính bằng cách lũy thừa tất cả các mục trong  $\mathbf{O}$  và sau đó chuẩn hóa chúng theo tổng.

## 5.4.2 Hàm mất mát

Tiếp theo, chúng ta cần một *hàm mất mát* để đánh giá chất lượng các dự đoán xác suất. Chúng ta sẽ dựa trên *hợp lý cực đại*, khái niệm tương tự đã gặp khi đưa ra lý giải xác suất cho hàm mục tiêu bình phương nhỏ nhất trong hồi quy tuyến tính (Section 5.1).

### Log hợp lý

Hàm softmax cho chúng ta một vector  $\hat{\mathbf{y}}$ , có thể được hiểu như các xác suất có điều kiện của từng lớp với đầu vào  $x$ . Ví dụ:  $\hat{y}_1 = \hat{P}(y = \text{cat} | \mathbf{x})$ . Để biết các ước lượng có sát với thực tế hay không, ta kiểm tra xác suất mà mô hình gán cho lớp *thật sự* khi biết các đặc trưng.

$$P(Y | X) = \prod_{i=1}^n P(y^{(i)} | x^{(i)}) \text{ và vì vậy } -\log P(Y | X) = \sum_{i=1}^n -\log P(y^{(i)} | x^{(i)}). \quad (5.4.6)$$

Cực đại hóa  $P(Y | X)$  (và vì vậy tương đương với cực tiểu hóa  $-\log P(Y | X)$ ) giúp việc dự đoán nhẫn tốt hơn. Điều này dẫn đến hàm mất mát (chúng tôi lược bỏ chỉ số trên  $(i)$  để tránh sự nhập nhằng về kí hiệu):

$$l = -\log P(y | x) = -\sum_j y_j \log \hat{y}_j. \quad (5.4.7)$$

Bởi vì những lý do sẽ được giải thích sau đây, hàm mất mát này thường được gọi là mất mát *entropy chéo*. Ở đây, chúng ta đã sử dụng nó bằng cách xây dựng  $\hat{y}$  giống như một phân phối xác suất rời rạc và vector  $\mathbf{y}$  là một vector one-hot. Do đó, tổng các số hạng với chỉ số  $j$  sẽ tiêu biến tạo thành một giá trị duy nhất. Bởi mọi  $\hat{y}_j$  đều là xác suất, log của chúng không bao giờ lớn hơn 0. Vì vậy, hàm mất mát sẽ không thể giảm thêm được nữa nếu chúng ta dự đoán chính xác  $y$  với *độ chắc chắn tuyệt đối*, tức  $P(y | x) = 1$  cho nhẫn đúng. Chú ý rằng điều này thường không khả thi. Ví dụ, nhẫn bị nhiễu sẽ xuất hiện trong tập dữ liệu (một vài mẫu bị dán nhầm nhẫn). Điều này cũng khó xảy ra khi những đặc trưng đầu vào không chứa đủ thông tin để phân loại các mẫu một cách hoàn hảo.

### Softmax và Đạo hàm

Vì softmax và hàm mất mát softmax rất phổ biến, nên việc hiểu cách tính giá trị các hàm này sẽ có ích về sau. Thay  $o$  vào định nghĩa của hàm mất mát  $l$  và dùng định nghĩa của softmax, ta được:

$$l = -\sum_j y_j \log \hat{y}_j = \sum_j y_j \log \sum_k \exp(o_k) - \sum_j y_j o_j = \log \sum_k \exp(o_k) - \sum_j y_j o_j. \quad (5.4.8)$$

Để hiểu rõ hơn, hãy cùng xét đạo hàm riêng của  $l$  theo  $o$ . Ta có:

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = P(y = j | x) - y_j. \quad (5.4.9)$$

Nói cách khác, gradient chính là hiệu giữa xác xuất mô hình gán cho lớp đúng  $P(y | x)$ , và nhẫn của dữ liệu  $y$ . Điều này cũng tương tự như trong bài toán hồi quy, khi gradient là hiệu giữa dữ liệu quan sát được  $y$  và kết quả ước lượng  $\hat{y}$ . Đây không phải là ngẫu nhiên. Trong mọi mô hình *hỗn lũy thừa*<sup>80</sup>, gradient của hàm log hợp lý đều có dạng như thế này. Điều này giúp cho việc tính toán gradient trong thực tế trở nên dễ dàng hơn.

<sup>80</sup> [https://en.wikipedia.org/wiki/Exponential\\_family](https://en.wikipedia.org/wiki/Exponential_family)

## Hàm mất mát Entropy chéo

Giờ hãy xem xét trường hợp mà ta quan sát được toàn bộ phân phối của đầu ra thay vì chỉ một giá trị đầu ra duy nhất. Ta có thể biểu diễn  $y$  giống hệt như trước. Sự khác biệt duy nhất là thay vì có một vector chỉ chứa các phần tử nhị phân, giả sử như  $(0, 0, 1)$ , giờ ta có một vector xác suất tổng quát, ví dụ như  $(0.1, 0.2, 0.7)$ . Các công thức toán học ta dùng trước đó để định nghĩa hàm mất mát  $l$  vẫn áp dụng tốt ở đây nhưng khái quát hơn một chút. Giá trị của các phần tử trong vector tương ứng giá trị kỳ vọng của hàm mất mát trên phân phối của nhau.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_j y_j \log \hat{y}_j. \quad (5.4.10)$$

Hàm trên được gọi là hàm mất mát entropy chéo và là một trong những hàm mất mát phổ biến nhất dùng cho bài toán phân loại đa lớp. Ta có thể làm sáng tỏ cái tên entropy chéo bằng việc giới thiệu các kiến thức cơ bản trong lý thuyết thông tin.

### 5.4.3 Lý thuyết Thông tin Cơ bản

Lý thuyết thông tin giải quyết các bài toán mã hóa, giải mã, truyền tải và xử lý thông tin (hay còn được gọi là dữ liệu) dưới dạng ngắn gọn nhất có thể.

#### Entropy

Ý tưởng cốt lõi trong lý thuyết thông tin chính là việc định lượng lượng thông tin chứa trong dữ liệu. Giá trị định lượng này chỉ ra giới hạn tối đa cho khả năng nén dữ liệu (khi tìm biểu diễn ngắn gọn nhất mà không mất thông tin). Giá trị định lượng này gọi là [entropy](#)<sup>81</sup>, xác định trên phân phối  $p$  của bộ dữ liệu, được định nghĩa bằng phương trình dưới đây:

$$H[p] = \sum_j -p(j) \log p(j). \quad (5.4.11)$$

Một định lý căn bản của lý thuyết thông tin là để có thể mã hóa dữ liệu thu thập ngẫu nhiên từ phân phối  $p$ , chúng ta cần sử dụng ít nhất  $H[p]$  “nat”. “nat” là đơn vị biểu diễn dữ liệu sử dụng cơ số  $e$ , tương tự với bit biểu diễn dữ liệu sử dụng cơ số 2. Một nat bằng  $\frac{1}{\log(2)} \approx 1.44$  bit.  $H[p]/2$  thường được gọi là entropy nhị phân.

#### Lượng tin

Có lẽ bạn sẽ tự hỏi việc nén dữ liệu thì liên quan gì với việc đưa ra dự đoán? Hãy tưởng tượng chúng ta có một luồng (stream) dữ liệu cần nén. Nếu ta luôn có thể dễ dàng đoán được đơn vị dữ liệu (token) kế tiếp thì dữ liệu này rất dễ nén! Ví như tất cả các đơn vị dữ liệu trong dòng dữ liệu luôn có một giá trị cố định thì đây là một dòng dữ liệu tẻ nhạt! Không những tẻ nhạt, mà nó còn dễ đoán nữa. Bởi vì chúng luôn có cùng giá trị, ta sẽ không phải truyền bất cứ thông tin nào để trao đổi nội dung của dòng dữ liệu này. Dễ đoán thì cũng dễ nén là vậy.

Tuy nhiên, nếu ta không thể dự đoán một cách hoàn hảo cho mỗi sự kiện, thì thi thoảng ta sẽ thấy ngạc nhiên. Sự ngạc nhiên trong chúng ta sẽ lớn hơn khi ta gán một xác suất thấp hơn cho sự kiện. Vì nhiều lý do mà chúng ta sẽ nghiên cứu trong phần phụ lục, Claude Shannon đã đưa ra giải pháp

<sup>81</sup> <https://en.wikipedia.org/wiki/Entropy>

$\log(1/p(j)) = -\log p(j)$  để định lượng sự ngạc nhiên của một người lúc quan sát sự kiện  $j$  sau khi đã gán cho sự kiện đó một xác suất (chủ quan)  $p(j)$ . Entropy lúc này sẽ là *lượng tin* (*độ ngạc nhiên*) kỳ vọng khi mà xác suất của các sự kiện đó được gán chính xác, khớp với phân phối sinh dữ liệu. Nói cách khác, entropy là lượng thông tin hay mức độ ngạc nhiên tối thiểu mà dữ liệu sẽ đem lại theo kỳ vọng.

### Xem xét lại Entropy chéo

Nếu entropy là mức độ ngạc nhiên trải nghiệm bởi một người nắm rõ xác suất thật, thì bạn có thể băn khoăn rằng *entropy chéo* là gì? Entropy chéo từ  $p$  đến  $q$ , ký hiệu  $H(p, q)$ , là sự ngạc nhiên kỳ vọng của một người quan sát với các xác suất chủ quan  $q$  đối với dữ liệu sinh ra dựa trên các xác suất  $p$ . Giá trị entropy chéo thấp nhất có thể đạt được khi  $p = q$ . Trong trường hợp này, entropy chéo từ  $p$  đến  $q$  là  $H(p, p) = H(p)$ . Liên hệ điều này lại với mục tiêu phân loại của chúng ta, thậm chí khi ta có khả năng dự đoán tốt nhất có thể và cho rằng việc này là khả thi, thì ta sẽ không bao giờ đạt đến mức hoàn hảo. Mất mát của ta bị giới hạn dưới (*lower-bounded*) bởi entropy tạo bởi các phân phối thực tế có điều kiện  $P(\mathbf{y} | \mathbf{x})$ .

### Phân kỳ Kullback Leibler

Có lẽ cách thông dụng nhất để đo lường khoảng cách giữa hai phân phối là tính toán *phân kỳ Kullback Leibler*  $D(p\|q)$ . Phân kỳ Kullback Leibler đơn giản là sự khác nhau giữa entropy chéo và entropy, có nghĩa là giá trị entropy chéo bổ sung phát sinh so với giá trị nhỏ nhất không thể giảm được mà nó có thể nhận:

$$D(p\|q) = H(p, q) - H[p] = \sum_j p(j) \log \frac{p(j)}{q(j)}. \quad (5.4.12)$$

Lưu ý rằng trong bài toán phân loại, ta không biết giá trị thật của  $p$ , vì thế mà ta không thể tính toán entropy trực tiếp được. Tuy nhiên, bởi vì entropy nằm ngoài tầm kiểm soát của chúng ta, việc giảm thiểu  $D(p\|q)$  so với  $q$  là tương đương với việc giảm thiểu mất mát entropy chéo.

Tóm lại, chúng ta có thể nghĩ đến mục tiêu của phân loại entropy chéo theo hai hướng: (i) cực đại hóa khả năng xảy ra của dữ liệu được quan sát; và (ii) giảm thiểu sự ngạc nhiên của ta (cũng như số lượng các bit) cần thiết để truyền đạt các nhãn.

#### 5.4.4 Sử dụng Mô hình để Dự đoán và Đánh giá

Sau khi huấn luyện mô hình hồi quy softmax với các đặc trưng đầu vào bất kì, chúng ta có thể dự đoán xác suất đầu ra ứng với mỗi lớp. Thông thường, chúng ta sử dụng lớp với xác suất dự đoán cao nhất làm lớp đầu ra. Một dự đoán được xem là chính xác nếu nó trùng khớp hay tương thích với lớp thật sự (nhãn). Ở phần tiếp theo của thí nghiệm, chúng ta sẽ sử dụng độ chính xác để đánh giá chất lượng của mô hình. Giá trị này là tỉ lệ giữa số mẫu được dự đoán chính xác so với tổng số mẫu được dự đoán.

### 5.4.5 Tóm tắt

- Chúng tôi đã giới thiệu về hàm softmax giúp ánh xạ một vector đầu vào sang các giá trị xác suất.
- Hồi quy softmax được áp dụng cho các bài toán phân loại. Nó sử dụng phân phối xác suất của các lớp đầu ra thông qua hàm softmax.
- Entropy chéo là một phép đánh giá tốt cho sự khác nhau giữa 2 phân phối xác suất. Nó đo lường số lượng bit cần để biểu diễn dữ liệu cho mô hình.

### 5.4.6 Bài tập

1. Chứng minh rằng độ phân kì Kullback-Leibler  $D(p\|q)$  không âm với mọi phân phối  $p$  và  $q$ .  
Gợi ý: sử dụng bất đẳng thức Jensen hay nghĩa là sử dụng bổ đề  $-\log x$  là một hàm lồi.
2. Chứng minh rằng  $\log \sum_j \exp(o_j)$  là một hàm lồi với  $o$ .
3. Chúng ta có thể tìm hiểu sâu hơn về sự liên kết giữa các họ hàm mũ và softmax.
  - Tính đạo hàm cấp hai của hàm mất mát entropy chéo  $l(y, \hat{y})$  cho softmax.
  - Tính phương sai của phân phối được cho bởi softmax( $o$ ) và chứng minh rằng nó khớp với đạo hàm cấp hai được tính ở trên.
4. Giả sử rằng chúng ta có 3 lớp, xác suất xảy ra cho mỗi lớp bằng nhau, nói cách khác vector xác suất là  $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ .
  - Vấn đề là gì nếu chúng ta cố gắng thiết kế một mã nhị phân cho nó? Chúng ta có thể làm khớp cận dưới của entropy trên số lượng bits hay không?
  - Bạn có thể thiết kế một mã tốt hơn không? Gợi ý: Điều gì xảy ra nếu chúng ta cố gắng biểu diễn 2 quan sát độc lập và nếu chúng ta biểu diễn  $n$  quan sát đồng thời?
5. Softmax là một cách gọi sai cho phép ánh xạ đã được giới thiệu ở trên (nhưng cộng đồng học sâu vẫn sử dụng nó). Công thức thật sự của softmax là  $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$ .
  - Chứng minh rằng  $\text{RealSoftMax}(a, b) > \max(a, b)$ .
  - Chứng minh rằng  $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) > \max(a, b)$  với  $\lambda > 0$ .
  - Chứng minh rằng khi  $\lambda \rightarrow \infty$ , chúng ta có  $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \rightarrow \max(a, b)$ .
  - Soft-min sẽ trông như thế nào?
  - Mở rộng nó cho nhiều hơn 2 số.

#### 5.4.7 Thảo luận

- Tiếng Anh<sup>82</sup>
- Tiếng Việt<sup>83</sup>

#### 5.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Thị Hồng Hạnh
- Nguyễn Lê Quang Nhật
- Lý Phi Long
- Lê Khắc Hồng Phúc
- Bùi Nhật Quân
- Nguyễn Minh Thư
- Trần Kiến An
- Vũ Hữu Tiệp
- Dương Nhật Tân
- Nguyễn Văn Tâm
- Trần Yến Thy
- Đinh Minh Tân
- Phạm Hồng Vinh
- Nguyễn Cảnh Thượng
- Phạm Minh Đức

### 5.5 Bộ dữ liệu Phân loại Ảnh (Fashion-MNIST)

Ở Section 20.9, chúng ta đã huấn luyện bộ phân loại Naive Bayes, sử dụng bộ dữ liệu MNIST được giới thiệu vào năm 1998 (LeCun et al., 1998). Mặc dù MNIST từng là một bộ dữ liệu tốt để đánh giá xếp hạng (*benchmark*), các mô hình đơn giản theo tiêu chuẩn ngày nay cũng có thể đạt được độ chính xác phân loại lên tới 95%. Điều này khiến nó không phù hợp cho việc phân biệt độ mạnh yếu của các mô hình. Ngày nay, MNIST được dùng trong các phép kiểm tra sơ bộ hơn là dùng để đánh giá xếp hạng. Để cải thiện vấn đề này, chúng ta sẽ tập trung thảo luận trong các mục tiếp theo về một bộ dữ liệu tương tự nhưng phức tạp hơn, đó là bộ dữ liệu Fashion-MNIST (Xiao et al., 2017) được giới thiệu vào năm 2017.

<sup>82</sup> <https://discuss.mxnet.io/t/2334>

<sup>83</sup> <https://forum.machinelearningcoban.com/c/d21>

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon
import sys

d2l.use_svg_display()
```

### 5.5.1 Tải về Bộ dữ liệu

Cũng giống như với MNIST, Gluon giúp việc tải và nạp bộ dữ liệu FashionMNIST vào bộ nhớ trở nên dễ dàng với lớp FashionMNIST trong gluon.data.vision. Các cơ chế của việc nạp và khám phá bộ dữ liệu sẽ được hướng dẫn ngắn gọn bên dưới. Vui lòng tham khảo [Section 20.9](#) để biết thêm chi tiết về việc nạp dữ liệu.

```
mnist_train = gluon.data.vision.FashionMNIST(train=True)
mnist_test = gluon.data.vision.FashionMNIST(train=False)
```

FashionMNIST chứa các hình ảnh thuộc 10 lớp, mỗi lớp có 6000 ảnh trong tập huấn luyện và 1000 ảnh trong tập kiểm tra. Do đó, tập huấn luyện và tập kiểm tra sẽ chứa tổng cộng lần lượt 60000 và 10000 ảnh.

```
len(mnist_train), len(mnist_test)
```

Các ảnh trong Fashion-MNIST tương ứng với các lớp: áo phông, quần dài, áo thun, váy, áo khoác, dép, áo sơ-mi, giày thể thao, túi và giày cao gót. Hàm dưới đây giúp chuyển đổi các nhãn giá trị số thành tên của từng lớp.

```
# Saved in the d2l package for later use
def get_fashion_mnist_labels(labels):
    text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                  'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [text_labels[int(i)] for i in labels]
```

Chúng ta có thể tạo một hàm để minh họa các mẫu này.

```
# Saved in the d2l package for later use
def show_images(imgs, num_rows, num_cols, titles=None, scale=1.5):
    """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)):
        ax.imshow(img.asnumpy())
        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 các hình ảnh và nhãn tương ứng của chúng (ở dạng chữ) từ một vài mẫu đầu tiên trong tập huấn luyện.

```
X, y = mnist_train[:18]
show_images(X.squeeze(axis=-1), 2, 9, titles=get_fashion_mnist_labels(y));
```

## 5.5.2 Đọc một Minibatch

Để đọc dữ liệu từ tập huấn luyện và tập kiểm tra một cách dễ dàng hơn, chúng ta sử dụng một DataLoader có sẵn thay vì tạo từ đầu như đã làm ở [Section 5.2](#). Nhắc lại là ở mỗi vòng lặp, một DataLoader sẽ đọc một minibatch của tập dữ liệu với kích thước batch\_size.

Trong quá trình huấn luyện, việc đọc dữ liệu có thể gây ra hiện tượng nghẽn cổ chai hiệu năng đáng kể, trừ khi mô hình đơn giản hoặc máy tính rất nhanh. Một tính năng tiện dụng của DataLoader là khả năng sử dụng đa tiến trình (*multiple processes*) để tăng tốc việc đọc dữ liệu. Ví dụ, chúng ta có thể dùng 4 tiến trình để đọc dữ liệu ( thông qua num\_workers). Vì tính năng này hiện tại không được hỗ trợ trên Windows, đoạn mã lập trình dưới đây sẽ kiểm tra nền tảng hệ điều hành để đảm bảo rằng chúng ta không làm phiền những người dùng Windows với các thông báo lỗi sau này.

```
# Saved in the d2l package for later use
def get_dataloader_workers(num_workers=4):
    # 0 means no additional process is used to speed up the reading of data.
    if sys.platform.startswith('win'):
        return 0
    else:
        return num_workers
```

Dưới đây, chúng ta chuyển đổi dữ liệu hình ảnh từ uint8 sang số thực dấu phẩy động (*floating point number*) 32 bit với lớp ToTensor. Ngoài ra, bộ chuyển đổi sẽ chia tất cả các số cho 255 để các điểm ảnh có giá trị từ 0 đến 1. Lớp ToTensor cũng chuyển kênh hình ảnh từ chiều cuối cùng sang chiều thứ nhất để tạo điều kiện cho các tính toán của mạng nơ-ron tích chập được giới thiệu sau này. Thông qua hàm transform\_first của tập dữ liệu, chúng ta có thể áp dụng phép biến đổi ToTensor cho phần tử đầu tiên của mỗi ví dụ (một ví dụ chứa hai phần tử là ảnh và nhãn).

```
batch_size = 256
transformer = gluon.data.vision.transforms.ToTensor()
train_iter = gluon.data.DataLoader(mnist_train.transform_first(transformer),
                                    batch_size, shuffle=True,
                                    num_workers=get_dataloader_workers())
```

Hãy cùng xem thời gian cần thiết để hoàn tất việc đọc dữ liệu huấn luyện.

```
timer = d2l.Timer()
for X, y in train_iter:
    continue
'%.2f sec' % timer.stop()
```

### 5.5.3 Kết hợp Tất cả lại với nhau

Bây giờ, chúng ta sẽ định nghĩa hàm `load_data_fashion_mnist` để nạp và đọc bộ dữ liệu Fashion-MNIST. Hàm này sẽ trả về các iterator cho dữ liệu của cả tập huấn luyện và tập kiểm định. Thêm nữa, nó chấp nhận một tham số tùy chọn để thay đổi kích thước hình ảnh đầu vào.

```
# Saved in the d2l package for later use
def load_data_fashion_mnist(batch_size, resize=None):
    """Download the Fashion-MNIST dataset and then load into memory."""
    dataset = gluon.data.vision
    trans = [dataset.transforms.Resize(resize)] if resize else []
    trans.append(dataset.transforms.ToTensor())
    trans = dataset.transforms.Compose(trans)
    mnist_train = dataset.FashionMNIST(train=True).transform_first(trans)
    mnist_test = dataset.FashionMNIST(train=False).transform_first(trans)
    return (gluon.data.DataLoader(mnist_train, batch_size, shuffle=True,
                                  num_workers=get_dataloader_workers()),
            gluon.data.DataLoader(mnist_test, batch_size, shuffle=False,
                                  num_workers=get_dataloader_workers()))
```

Dưới đây, chúng ta xác nhận rằng kích thước hình ảnh đã được thay đổi.

```
train_iter, test_iter = load_data_fashion_mnist(32, (64, 64))
for X, y in train_iter:
    print(X.shape)
    break
```

Giờ chúng ta đã sẵn sàng để làm việc với bộ dữ liệu FashionMNIST trong các mục tiếp theo.

### 5.5.4 Tóm tắt

- Fashion-MNIST là một tập dữ liệu phân loại trang phục bao gồm các hình ảnh đại diện cho 10 lớp.
- Chúng ta sẽ sử dụng tập dữ liệu này trong các mục và chương tiếp theo để đánh giá các thuật toán phân loại khác nhau.
- Chúng ta lưu trữ kích thước của mỗi hình ảnh với chiều cao  $h$  chiều rộng  $w$  điểm ảnh dưới dạng  $h \times w$  hoặc ( $h$ ,  $w$ ).
- Iterator cho dữ liệu là nhân tố chính để đạt được hiệu suất cao. Hãy sử dụng các iterator được lập trình tốt để tận dụng khả năng chạy đa tiến trình, tránh làm chậm vòng lặp huấn luyện.

### 5.5.5 Bài tập

1. Việc giảm batch\_size (ví dụ xuống 1) có ảnh hưởng tới tốc độ đọc dữ liệu hay không?
2. Với người dùng không sử dụng Windows, hãy thử thay đổi num\_workers để xem nó ảnh hưởng đến hiệu năng đọc dữ liệu như thế nào. Vẽ đồ thị hiệu năng tương ứng với số tiến trình được sử dụng.
3. Sử dụng tài liệu MXNet để xem các bộ dữ liệu có sẵn khác trong mxnet.gluon.data.vision.
4. Sử dụng tài liệu MXNet để xem những phép biến đổi nào có sẵn trong mxnet.gluon.data.vision.transforms.

### 5.5.6 Thảo luận

- Tiếng Anh<sup>84</sup>
- Tiếng Việt<sup>85</sup>

### 5.5.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức

## 5.6 Lập trình Hồi quy Sofmax từ đầu

Giống như khi ta lập trình hồi quy tuyến tính từ đầu, hồi quy (softmax) logistic đa lớp cũng là một kĩ thuật căn bản mà bạn nên hiểu biết tường tận các chi tiết để có thể tự xây dựng lại. Sau khi tự lập trình lại mọi thứ thì ta cũng sẽ dùng Gluon để so sánh như ở phần hồi quy tuyến tính. Để bắt đầu, chúng ta nhập các thư viện quen thuộc vào.

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, gluon
from IPython import display
npx.set_np()
```

Ta sẽ làm việc trên tập dữ liệu Fashion-MNIST, vừa được giới thiệu trong Section 5.5, thiết lập một iterator với kích thước batch là 256.

<sup>84</sup> <https://discuss.mxnet.io/t/2335>

<sup>85</sup> <https://forum.machinelearningcoban.com/c/d2l>

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

### 5.6.1 Khởi tạo các Tham số của Mô hình

Giống như ví dụ về hồi quy tuyến tính, mỗi mẫu sẽ được biểu diễn bằng một vector có chiều dài cố định. Mỗi mẫu trong tập dữ liệu thô là một ảnh  $28 \times 28$ . Trong phần này, chúng ta sẽ trải phẳng mỗi tấm ảnh thành một vector có kích thước là 784. Sau này ta sẽ bàn về các chiến lược công phu hơn có khả năng khai thác cấu trúc không gian của ảnh, còn bây giờ ta hãy xem mỗi điểm ảnh là một đặc trưng.

Nhắc lại trong hồi quy softmax, mỗi lớp sẽ có một đầu ra. Vì tập dữ liệu của chúng ta có 10 lớp, mạng của chúng ta sẽ có 10 đầu ra. Do đó, các trọng số sẽ tạo thành một ma trận  $784 \times 10$  và các hệ số điều chỉnh sẽ tạo thành một vector  $1 \times 10$ . Cũng như hồi quy tuyến tính, ta sẽ khởi tạo các trọng số  $W$  bằng nhiễu Gauss và các hệ số điều chỉnh sẽ được khởi tạo bằng 0.

```
num_inputs = 784
num_outputs = 10

W = np.random.normal(0, 0.01, (num_inputs, num_outputs))
b = np.zeros(num_outputs)
```

Hãy nhớ rằng ta cần *đính kèm gradient* vào các tham số của mô hình. Cụ thể hơn, ta cho MXNet biết rằng ta sẽ muốn tính các gradient theo các tham số này và cần phân bổ bộ nhớ để lưu trữ chúng trong tương lai.

```
W.attach_grad()
b.attach_grad()
```

### 5.6.2 Softmax

Trước khi xây dựng mô hình hồi quy softmax, hãy ôn nhanh tác dụng của các toán tử như `sum` trên những chiều cụ thể của một ndarray. Cho một ma trận  $X$ , chúng ta có thể tính tổng tất cả các phần tử (mặc định) hoặc chỉ trên các phần tử trong cùng một trục, ví dụ, cột (`axis=0`) hoặc cùng một hàng (`axis=1`). Lưu ý rằng nếu  $X$  là một mảng có kích thước  $(2, 3)$ , tính tổng các cột (`X.sum(axis=0)`) sẽ trả về một vector (một chiều) có kích thước là  $(3, )$ . Nếu chúng ta muốn giữ số lượng trục giống với mảng ban đầu thay vì thu gọn kích thước trên các trục đã tính toán (dẫn đến một mảng 2 chiều có kích thước  $(1, 3)$ ), ta có thể chỉ định `keepdims=True` khi gọi hàm `sum`.

```
X = np.array([[1, 2, 3], [4, 5, 6]])
print(X.sum(axis=0, keepdims=True), '\n', X.sum(axis=1, keepdims=True))
```

Bây giờ chúng ta đã có thể bắt đầu xây dựng hàm softmax. Lưu ý rằng việc thực thi hàm softmax bao gồm hai bước: Đầu tiên, chúng ta lũy thừa từng giá trị (sử dụng `exp`). Sau đó tính tổng trên mỗi hàng để lấy các hằng số chuẩn hóa cho mỗi mẫu (vì mỗi mẫu là một hàng). Cuối cùng, chia mỗi hàng theo hằng số chuẩn hóa của nó để đảm bảo rằng kết quả có tổng bằng 1. Trước khi xem đoạn mã, chúng ta hãy nhớ lại các bước này được thể hiện trong phương trình sau:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}. \quad (5.6.1)$$

Mẫu số hoặc hằng số chuẩn hóa đôi khi cũng được gọi là hàm phân hoạch (*partition function*) và logarit của nó được gọi là hàm log phân hoạch (*log-partition function*). Tên gốc của hàm được định nghĩa trong [cơ học thống kê<sup>86</sup>](#) với phương trình liên quan đến mô hình hóa phân phối trên một tập hợp các phần tử.

```
def softmax(X):
    X_exp = np.exp(X)
    partition = X_exp.sum(axis=1, keepdims=True)
    return X_exp / partition # The broadcast mechanism is applied here
```

Có thể thấy rằng với bất kỳ đầu vào ngẫu nhiên nào thì mỗi phần tử đều được biến đổi thành một số không âm. Hơn nữa, mỗi hàng đều có tổng là 1 nên thỏa mãn yêu cầu làm một giá trị xác suất. Chú ý rằng đoạn mã trên tuy đúng về mặt toán học nhưng chúng tôi đã hơi cẩu thả về mặt lập trình vì đã không kiểm tra vấn đề tràn số trên và dưới gây ra bởi các giá trị vô cùng lớn hoặc vô cùng nhỏ trong ma trận, không giống với cách đã thực hiện tại [Section 20.9](#).

```
X = np.random.normal(size=(2, 5))
X_prob = softmax(X)
X_prob, X_prob.sum(axis=1)
```

### 5.6.3 Mô hình

Sau khi đã định nghĩa hàm softmax, chúng ta có thể bắt đầu lập trình mô hình hồi quy softmax. Đoạn mã sau định nghĩa lượt truyền xuôi qua mạng. Chú ý rằng chúng ta sẽ làm phẳng mỗi ảnh gốc trên batch thành một vector có độ dài num\_inputs bằng hàm reshape, trước khi truyền dữ liệu sang mô hình đã khởi tạo.

```
def net(X):
    return softmax(np.dot(X.reshape(-1, num_inputs), W) + b)
```

### 5.6.4 Hàm mất mát

Tiếp đến chúng ta cần lập trình hàm mất mát entropy chéo đã được giới thiệu ở [Section 5.4](#). Đây có lẽ là hàm mất mát thông dụng nhất trong phần lớn nghiên cứu về học sâu vì hiện nay số lượng bài toán phân loại đã vượt xa hơn số lượng bài toán hồi quy.

Nhắc lại rằng hàm entropy chéo lấy đầu vào là đối log hợp lý của xác suất dự đoán được gán cho nhãn thật  $-\log P(y | x)$ . Thay vì lặp qua các dự đoán của mô hình bằng vòng lặp for trong Python (có xu hướng kém hiệu quả), chúng ta có thể sử dụng hàm pick để dễ dàng chọn các phần tử thích hợp từ ma trận của các giá trị softmax. Dưới đây, ta minh họa cách sử dụng hàm pick trong một ví dụ đơn giản với ma trận có 3 lớp và 2 mẫu.

```
y_hat = np.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], [0, 2]]
```

Bây giờ chúng ta có thể lập trình hàm mất mát entropy chéo hiệu quả hơn chỉ với một dòng lệnh.

<sup>86</sup> [https://en.wikipedia.org/wiki/Partition\\_function\\_\(statistical\\_mechanics\)](https://en.wikipedia.org/wiki/Partition_function_(statistical_mechanics))

```

def cross_entropy(y_hat, y):
    return - np.log(y_hat[range(len(y_hat)), y])

```

### 5.6.5 Độ chính xác cho bài toán Phân loại

Với phân phối xác suất dự đoán  $y_{\text{hat}}$ , ta thường chọn lớp có xác suất dự đoán cao nhất khi phải đưa ra một dự đoán *cụ thể* vì nhiều ứng dụng trong thực tế có yêu cầu như vậy. Ví dụ, Gmail cần phải phân loại một email vào trong các danh mục sau: Email chính (Primary), Mạng xã hội (Social), Nội dung cập nhật (Updates) hoặc Diễn đàn (Forums). Có thể các xác suất được tính toán bên trong nội bộ hệ thống, nhưng cuối cùng kết quả vẫn chỉ là một trong các danh mục.

Các dự đoán được coi là chính xác khi chúng khớp với lớp thực tế  $y$ . Độ chính xác phân loại được tính bởi tỉ lệ các dự đoán chính xác trên tất cả các dự đoán đã đưa ra. Dù ta có thể gặp khó khăn khi tối ưu hóa trực tiếp độ chính xác (chúng không khả vi), đây thường là phép đo chất lượng được quan tâm nhiều nhất và sẽ luôn được tính khi huấn luyện các bộ phân loại.

Độ chính xác được tính toán như sau: Đầu tiên, dùng lệnh  $y_{\text{hat}}.\text{argmax}(\text{axis}=1)$  nhằm lấy ra các lớp được dự đoán (được cho bởi chỉ số của phần tử lớn nhất trên mỗi hàng). Kết quả trả về sẽ có cùng kích thước với biến  $y$  và bây giờ ta chỉ cần so sánh hai vector này. Vì toán tử  $\text{==}$  so khớp cả kiểu dữ liệu của biến (ví dụ một biến `int` và một biến `float32` không thể bằng nhau), ta cần đưa chúng về cùng một kiểu dữ liệu (ở đây ta chọn kiểu `float32`). Kết quả sẽ là một `ndarray` chứa các giá trị 0 (false) và 1 (true), giá trị trung bình này mang lại kết quả mà ta mong muốn.

```

# Saved in the d2l package for later use
def accuracy(y_hat, y):
    if y_hat.shape[1] > 1:
        return float((y_hat.argmax(axis=1) == y.astype('float32')).sum())
    else:
        return float((y_hat.astype('int32') == y.astype('int32')).sum())

```

Ta sẽ tiếp tục sử dụng biến  $y_{\text{hat}}$  và  $y$  đã được định nghĩa trong hàm `pick`, lần lượt tương ứng với phân phối xác suất được dự đoán và nhãn. Có thể thấy rằng kết quả dự đoán lớp từ ví dụ đầu tiên là 2 (phần tử lớn nhất trong hàng là 0.6 với chỉ số tương ứng là 2) không khớp với nhãn thực tế là 0. Dự đoán lớp ở ví dụ thứ hai là 2 (phần tử lớn nhất hàng là 0.5 với chỉ số tương ứng là 2) khớp với nhãn thực tế là 2. Do đó, ta có độ chính xác phân loại cho hai ví dụ này là 0.5.

```

y = np.array([0, 2])
accuracy(y_hat, y) / len(y)

```

Tương tự như trên, ta có thể đánh giá độ chính xác của mô hình net trên tập dữ liệu (được truy xuất thông qua `data_iter`).

```

# Saved in the d2l package for later use
def evaluate_accuracy(net, data_iter):
    metric = Accumulator(2) # num_corrected_examples, num_examples
    for X, y in data_iter:
        metric.add(accuracy(net(X), y), y.size)
    return metric[0] / metric[1]

```

`Accumulator` ở đây là một lớp đa tiện ích, có tác dụng tính tổng tích lũy của nhiều số.

```

# Saved in the d2l package for later use
class Accumulator(object):
    """Sum a list of numbers over time."""

    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] * len(self.data)

    def __getitem__(self, i):
        return self.data[i]

```

Vì ta đã khởi tạo mô hình net với trọng số ngẫu nhiên nên độ chính xác của mô hình lúc này sẽ gần với việc phỏng đoán ngẫu nhiên, tức độ chính xác bằng 0.1 với 10 lớp.

```
evaluate_accuracy(net, test_iter)
```

## 5.6.6 Huấn luyện Mô hình

Vòng lặp huấn luyện cho hồi quy softmax trông khá quen thuộc nếu bạn đã xem cách lập trình cho hồi quy tuyến tính tại Section 5.2. Ở đây, chúng ta tái cấu trúc (*refactor*) lại đoạn mã để sau này có thể tái sử dụng. Đầu tiên, chúng ta định nghĩa một hàm để huấn luyện với một epoch dữ liệu. Lưu ý rằng `updater` là một hàm tổng quát để cập nhật các tham số của mô hình và sẽ nhận kích thước batch làm đối số. Nó có thể là một wrapper của `d2l.sgd` hoặc là một đối tượng huấn luyện Gluon.

```

# Saved in the d2l package for later use
def train_epoch_ch3(net, train_iter, loss, updater):
    metric = Accumulator(3) # train_loss_sum, train_acc_sum, num_examples
    if isinstance(updater, gluon.Trainer):
        updater = updater.step
    for X, y in train_iter:
        # Compute gradients and update parameters
        with autograd.record():
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            updater(X.shape[0])
            metric.add(float(l.sum()), accuracy(y_hat, y), y.size)
    # Return training loss and training accuracy
    return metric[0]/metric[2], metric[1]/metric[2]

```

Trước khi xem đoạn mã thực hiện hàm huấn luyện, ta định nghĩa một lớp phụ trợ để minh họa dữ liệu. Mục đích của nó là đơn giản hóa các đoạn mã sẽ xuất hiện trong những chương sau.

```

# Saved in the d2l package for later use
class Animator(object):
    def __init__(self, xlabel=None, ylabel=None, legend=[], xlim=None,
                 ylim=None, xscale='linear', yscale='linear', fmts=None,

```

(continues on next page)

```

        nrows=1, ncols=1, figsize=(3.5, 2.5)):
    """Incrementally plot multiple lines."""
    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 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)]
    if not self.fmts:
        self.fmts = ['-' * i for i 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()
    display.display(self.fig)
    display.clear_output(wait=True)

```

Hàm huấn luyện sau đó sẽ chạy qua nhiều epoch và trực quan hóa quá trình huấn luyện.

```

# Saved in the d2l package for later use
def train_ch3(net, train_iter, test_iter, loss, num_epochs, updater):
    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,))

```

Nhắc lại, chúng ta sử dụng giải thuật hạ gradient ngẫu nhiên theo minibatch để tối ưu hàm mất mát của mô hình. Lưu ý rằng số lượng epoch (num\_epochs), và hệ số học (lr) là hai siêu tham số được hiệu chỉnh. Bằng cách thay đổi các giá trị này, chúng ta có thể tăng độ chính xác khi phân loại của mô hình. Trong thực tế, chúng ta sẽ chia dữ liệu của mình theo ba hướng tiếp cận khác nhau gồm: dữ liệu huấn luyện, dữ liệu kiểm định và dữ liệu kiểm tra; sử dụng dữ liệu kiểm định để chọn ra những giá trị tốt nhất cho các siêu tham số.

```

num_epochs, lr = 10, 0.1

def updater(batch_size):
    return d2l.sgd([W, b], lr, batch_size)

train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs, updater)

```

### 5.6.7 Dự đoán

Giờ thì việc huấn luyện đã hoàn thành, mô hình của chúng ta đã sẵn sàng để phân loại ảnh. Cho một loạt các ảnh, chúng ta sẽ so sánh các nhãn thật của chúng (dòng đầu tiên của văn bản đầu ra) với những dự đoán của mô hình (dòng thứ hai của văn bản đầu ra).

```

# Saved in the d2l package for later use
def predict_ch3(net, test_iter, n=6):
    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)

```

### 5.6.8 Tóm tắt

Với hồi quy softmax, chúng ta có thể huấn luyện các mô hình cho bài toán phân loại đa lớp. Vòng lặp huấn luyện rất giống với vòng lặp huấn luyện của hồi quy tuyến tính: truy xuất và đọc dữ liệu, định nghĩa mô hình và hàm mất mát, và rồi huấn luyện mô hình sử dụng các giải thuật tối ưu. Rồi bạn sẽ sớm thấy rằng hầu hết các mô hình học sâu phổ biến đều có quy trình huấn luyện tương tự như vậy.

### 5.6.9 Bài tập

- Trong mục này, chúng ta đã lập trình hàm softmax dựa vào định nghĩa toán học của phép toán softmax. Điều này có thể gây ra những vấn đề gì (gợi ý: thử tính  $\exp(50)$ )?
- Hàm `cross_entropy` trong mục này được lập trình dựa vào định nghĩa của hàm mất mát entropy chéo. Vấn đề gì có thể xảy ra với cách lập trình như vậy (gợi ý: xem xét miền của hàm  $\log$ )?
- Bạn có thể tìm ra giải pháp cho hai vấn đề trên không?
- Việc trả về nhãn có khả năng nhất có phải lúc nào cũng là ý tưởng tốt không? Ví dụ, bạn có dùng phương pháp này cho chẩn đoán bệnh hay không?
- Giả sử rằng chúng ta muốn sử dụng hồi quy softmax để dự đoán từ tiếp theo dựa vào một số đặc trưng. Những vấn đề gì có thể xảy ra nếu dùng một tập từ vựng lớn?

### 5.6.10 Thảo luận

- Tiếng Anh<sup>87</sup>
- Tiếng Việt<sup>88</sup>

### 5.6.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Bùi Nhật Quân
- Lý Phi Long
- Phạm Hồng Vinh
- Lâm Ngọc Tâm
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Nguyễn Cảnh Thưởng
- Phạm Minh Đức
- Nguyễn Quang Hải
- Đinh Minh Tân
- Lê Cao Thăng

## 5.7 Cách lập trình súc tích Hồi quy Softmax

Giống như cách Gluon giúp việc lập trình hồi quy tuyến tính ở Section 5.3 trở nên dễ dàng hơn, ta sẽ thấy nó cũng mang đến sự tiện lợi tương tự (hoặc có thể hơn) khi lập trình các mô hình phân loại. Một lần nữa, chúng ta bắt đầu bằng việc nhập các gói thư viện.

```
from d2l import mxnet as d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

Chúng ta sẽ tiếp tục làm việc với bộ dữ liệu Fashion-MNIST và giữ kích thước batch bằng 256 như ở mục trước.

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

<sup>87</sup> <https://discuss.mxnet.io/t/2336>

<sup>88</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 5.7.1 Khởi tạo Tham số Mô hình

Như đã đề cập trong Section 5.4, tầng đầu ra của hồi quy softmax là một tầng kết nối đầy đủ (Dense). Do đó, để xây dựng mô hình, ta chỉ cần thêm một tầng Dense với 10 đầu ra vào đối tượng Sequential. Việc sử dụng Sequential ở đây không thực sự cần thiết, nhưng ta nên hình thành thói quen này vì nó sẽ luôn hiện diện khi ta xây dựng các mô hình sâu. Một lần nữa, chúng ta khởi tạo các trọng số một cách ngẫu nhiên với trung bình bằng không và độ lệch chuẩn bằng 0.01.

```
net = nn.Sequential()
net.add(nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

### 5.7.2 Hàm Softmax

Ở ví dụ trước, ta đã tính toán kết quả đầu ra của mô hình và sau đó đưa các kết quả này vào hàm mất mát entropy chéo. Về mặt toán học, cách làm này hoàn toàn có lý. Tuy nhiên, từ góc độ điện toán, sử dụng hàm mũ có thể là nguồn gốc của các vấn đề về ổn định số học (được bàn trong Section 20.9). Hãy nhớ rằng, hàm softmax tính  $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}$ , trong đó  $\hat{y}_j$  là phần tử thứ  $j^{\text{th}}$  của  $y_{\text{hat}}$  và  $z_j$  là phần tử thứ  $j^{\text{th}}$  của biến đầu vào  $y_{\text{linear}}$ .

Nếu một phần tử  $z_i$  quá lớn,  $e^{z_i}$  có thể sẽ lớn hơn giá trị cực đại mà kiểu float có thể biểu diễn được (đây là hiện tượng tràn số trên). Lúc này mẫu số hoặc tử số (hoặc cả hai) sẽ tiến tới  $\inf$  và ta gặp phải trường hợp  $\hat{y}_i$  bằng 0,  $\inf$  hoặc  $\text{nan}$ . Trong những tình huống này, giá trị trả về của cross\_entropy có thể không được xác định một cách rõ ràng. Một mẹo để khắc phục việc này là: đầu tiên ta trừ tất cả các  $z_i$  đi  $\max(z_i)$ , sau đó mới đưa chúng vào hàm softmax. Bạn có thể nhận thấy rằng việc tịnh tiến mỗi  $z_i$  theo một hệ số không đổi sẽ không làm ảnh hưởng đến giá trị trả về của hàm softmax.

Sau khi thực hiện bước trừ và chuẩn hóa, một vài  $z_j$  có thể có giá trị âm lớn và do đó  $e^{z_j}$  sẽ xấp xỉ 0. Điều này có thể dẫn đến việc chúng bị làm tròn thành 0 do khả năng biểu diễn chính xác là hữu hạn (tức tràn số dưới), khiến  $\hat{y}_j$  tiến về không và giá trị  $\log(\hat{y}_j)$  tiến về  $-\inf$ . Thực hiện vài bước lan truyền ngược với lối trên, ta có thể sẽ đổi mặt với một loạt giá trị  $\text{nan}$  (*not-a-number: không phải số*) đáng sợ.

May mắn thay, mặc dù ta đang thực hiện tính toán với các hàm mũ, kết quả cuối cùng ta muốn là giá trị log của nó (khi tính hàm mất mát entropy chéo). Bằng cách kết hợp cả hai hàm (softmax và cross-entropy) lại với nhau, ta có thể khắc phục vấn đề về ổn định số học và tránh gặp khó khăn trong quá trình lan truyền ngược. Trong phương trình bên dưới, ta đã không tính  $e^{z_j}$  mà thay vào đó, ta tính trực tiếp  $z_j$  do việc rút gọn  $\log(\exp(\cdot))$ .

$$\begin{aligned}\log(\hat{y}_j) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^n e^{z_i}}\right) \\ &= \log(e^{z_j}) - \log\left(\sum_{i=1}^n e^{z_i}\right) \\ &= z_j - \log\left(\sum_{i=1}^n e^{z_i}\right).\end{aligned}\tag{5.7.1}$$

Ta vẫn muốn giữ lại hàm softmax gốc để sử dụng khi muốn tính đầu ra của mô hình dưới dạng xác suất. Nhưng thay vì truyền xác suất softmax vào hàm mất mát mới, ta sẽ chỉ truyền các giá trị logit (các giá trị khi chưa qua softmax) và tính softmax cùng log của nó trong hàm mất mát

`softmax_cross_entropy`. Hàm này cũng sẽ tự động thực hiện các mẹo thông minh như log-sum-exp (xem thêm Wikipedia<sup>89</sup>).

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

### 5.7.3 Thuật toán Tối ưu

Ở đây, chúng ta sử dụng thuật toán tối ưu hạ gradient ngẫu nhiên theo minibatch với tốc độ học bằng 0.1. Lưu ý rằng cách làm này giống hệt cách làm ở ví dụ về hồi quy tuyến tính, minh chứng cho tính khái quát của bộ tối ưu hạ gradient ngẫu nhiên.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

### 5.7.4 Huấn luyện

Tiếp theo, chúng ta sẽ gọi hàm huấn luyện đã được khai báo ở mục trước để huấn luyện mô hình.

```
num_epochs = 10
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

Giống lần trước, thuật toán hội tụ tới một mô hình có độ chính xác 83.7% nhưng chỉ khác là cần ít dòng lệnh hơn. Lưu ý rằng trong nhiều trường hợp, Gluon không chỉ dùng các mánh phổ biến mà còn sử dụng các kỹ thuật khác để tránh các lỗi kĩ thuật tính toán mà ta dễ gặp phải nếu tự lập trình mô hình từ đầu.

### 5.7.5 Bài tập

- Thử thay đổi các siêu tham số như kích thước batch, số epoch và tốc độ học. Theo dõi kết quả sau khi thay đổi.
- Tại sao độ chính xác trên tập kiểm tra lại giảm sau một khoảng thời gian? Chúng ta giải quyết việc này thế nào?

### 5.7.6 Thảo luận

- Tiếng Anh<sup>90</sup>
- Tiếng Việt<sup>91</sup>

<sup>89</sup> <https://en.wikipedia.org/wiki/LogSumExp>

<sup>90</sup> <https://discuss.mxnet.io/t/2337>

<sup>91</sup> <https://forum.machinelearningcoban.com/c/d2l>

### **5.7.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Lý Phi Long
- Phạm Minh Đức
- Dương Nhật Tân
- Nguyễn Văn Tâm
- Phạm Hồng Vinh

### **5.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Hoàng Quân
- Vũ Hữu Tiệp

# 6 | Perceptron Đa tầng

Trong chương này, chúng tôi sẽ giới thiệu mạng nơ-ron sâu thực sự đầu tiên của bạn. Mạng nơ-ron sâu đơn giản nhất được gọi là perceptron đa tầng. Nó gồm nhiều tầng nơ-ron, mỗi nơ-ron được kết nối đầy đủ với các nơ-ron khác ở tầng phía dưới (các nơ-ron cung cấp đầu vào) và tầng phía trên (các nơ-ron mà nó gây ảnh hưởng). Khi huấn luyện các mô hình có độ phức tạp cao, ta sẽ có nguy cơ gặp vấn đề quá khứ. Vì vậy, chúng tôi cần cung cấp cho bạn những hiểu biết ban đầu thật chặt chẽ với các khái niệm quá khứ, dưới khứ và kiểm soát độ phức tạp. Nhằm giúp bạn giải quyết những vấn đề kể trên, chúng tôi sẽ giới thiệu những kỹ thuật điều chuẩn như dropout và suy giảm trọng số. Ta cũng sẽ bàn đến các vấn đề liên quan tới sự ổn định số học và việc khởi tạo tham số, hai yếu tố chính giúp việc huấn luyện mạng nơ-ron sâu thành công. Xuyên suốt chương này, chúng tôi tập trung vào việc áp dụng các mô hình cho dữ liệu thực tế, nhằm giúp độc giả không chỉ nắm vững được các khái niệm mà còn có thể thực hành sử dụng mạng nơ-ron sâu. Những vấn đề liên quan tới hiệu năng tính toán, khả năng mở rộng và mức hiệu quả của mô hình sẽ được giới thiệu ở các chương sau.

## 6.1 Perceptron đa tầng

Trong chương trước, chúng tôi đã giới thiệu hồi quy softmax (Section 5.4), cách lập trình giải thuật này từ đầu (Section 5.6), sử dụng nó trong gluon (Section 5.7), và huấn luyện các bộ phân loại để nhận diện 10 lớp quần áo khác nhau từ các bức ảnh có độ phân giải thấp. Cùng với đó, chúng ta đã học cách xử lý dữ liệu, ép buộc các giá trị đầu ra tạo thành một phân phối xác suất hợp lệ ( thông qua hàm softmax), áp dụng một hàm mất mát phù hợp và cực tiểu hóa nó theo các tham số mô hình. Bây giờ, sau khi đã thành thạo các cơ chế nêu trên trong ngữ cảnh của những mô hình tuyến tính đơn giản, chúng ta có thể bắt đầu khám phá trọng tâm của cuốn sách này: lớp mô hình phong phú của các mạng nơ-ron sâu.

### 6.1.1 Các tầng ẩn

Để bắt đầu, hãy nhớ lại kiến trúc mô hình trong ví dụ của hồi quy softmax, được minh họa trong Fig. 6.1.1 bên dưới. Mô hình này ánh xạ trực tiếp các đầu vào của chúng ta sang các giá trị đầu ra thông qua một phép biến đổi tuyến tính duy nhất:

$$\hat{\mathbf{o}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}). \quad (6.1.1)$$

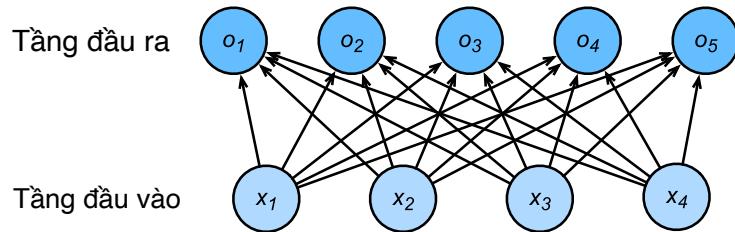


Fig. 6.1.1: Perceptron đơn tầng với 5 nút đầu ra.

Nếu các nhãn của chúng ta thật sự có mối quan hệ tuyến tính với dữ liệu đầu vào, thì cách tiếp cận này là đủ. Nhưng tính tuyến tính là một *giả định chặt*.

Ví dụ, tính tuyến tính ngụ ý về giả định *yếu hơn* của *tính đơn điệu*: tức giá trị đặc trưng tăng luôn dẫn đến việc đầu ra mô hình tăng (nếu trọng số tương ứng dương), hoặc đầu ra mô hình giảm (nếu trọng số tương ứng âm). Điều này đôi khi cũng hợp lý. Ví dụ, nếu chúng ta đang dự đoán liệu một người có trả được khoản vay hay không, chúng ta có thể suy diễn một cách hợp lý như sau: bỏ qua mọi yếu tố khác, ứng viên nào có thu nhập cao hơn sẽ có khả năng trả được nợ cao hơn so với những ứng viên khác có thu nhập thấp hơn. Dù có tính đơn điệu, mối quan hệ này khả năng cao là không liên quan tuyến tính tới xác suất trả nợ. Khả năng trả được nợ thường sẽ có mức tăng lớn hơn khi thu nhập tăng từ \$0 lên \$50k so với khi tăng từ \$1M lên \$1.05M. Một cách để giải quyết điều này là tiền xử lý dữ liệu để tính tuyến tính trở nên hợp lý hơn, ví dụ như sử dụng logarit của thu nhập để làm đặc trưng.

Lưu ý rằng chúng ta có thể dễ dàng đưa ra các ví dụ vi phạm *tính đơn điệu*. Ví dụ như, ta muốn dự đoán xác suất tử vong của một người dựa trên thân nhiệt. Đối với người có thân nhiệt trên 37°C (98.6°F), nhiệt độ càng cao gây ra nguy cơ tử vong càng cao. Tuy nhiên, với những người có thân nhiệt thấp hơn 37°C, khi gấp nhiệt độ cao hơn thì nguy cơ tử vong lại *thấp hơn!* Trong bài toán này, ta có thể giải quyết nó bằng một vài bước tiền xử lý thật khéo léo. Cụ thể, ta có thể sử dụng *khoảng cách* từ 37°C tới thân nhiệt làm đặc trưng.

Nhưng còn với bài toán phân loại hình ảnh chó mèo thì sao? Liệu việc tăng cường độ sáng của điểm ảnh tại vị trí (13, 17) sẽ luôn tăng (hoặc giảm) khả năng đó là hình một con chó? Sử dụng mô hình tuyến tính trong trường hợp này tương ứng với việc ngầm giả định rằng chỉ cần đánh giá độ sáng của từng pixel để phân biệt giữa mèo và chó. Cách tiếp cận này chắc chắn sẽ không chính xác khi các hình ảnh bị đảo ngược màu sắc.

Tuy nhiên, ta bỏ qua sự phi lý của tuyến tính ở đây, so với các ví dụ trước, rõ ràng là ta không thể giải quyết bài toán này với vài bước tiền xử lý chỉnh sửa đơn giản. Bởi vì ý nghĩa của các điểm ảnh phụ thuộc một cách phức tạp vào bối cảnh xung quanh nó (các giá trị xung quanh của điểm ảnh). Có thể vẫn tồn tại một cách biểu diễn dữ liệu nào đó nắm bắt được sự tương tác giữa các đặc trưng liên quan (và quan trọng nhất là phù hợp với mô hình tuyến tính), ta đơn giản là không biết làm thế nào để tính toán nó một cách thủ công. Với các mạng nơ-ron sâu, ta sử dụng dữ liệu đã quan sát được để đồng thời học cách biểu diễn (qua các tầng ẩn) và học một bộ dự đoán tuyến tính hoạt động dựa trên biểu diễn đó.

## Kết hợp các Tầng ẩn

Ta có thể vượt qua những hạn chế của mô hình tuyến tính và làm việc với một lớp hàm tổng quát hơn bằng cách thêm vào một hoặc nhiều tầng ẩn. Cách dễ nhất để làm điều này là xếp chồng nhiều tầng kết nối đầy đủ lên nhau. Giá trị đầu ra của mỗi tầng được đưa làm giá trị đầu vào cho tầng bên trên, cho đến khi ta tạo được một đầu ra. Ta có thể xem  $L - 1$  tầng đầu tiên như các tầng học biểu diễn dữ liệu và tầng cuối cùng là bộ dự đoán tuyến tính. Kiến trúc này thường được gọi là *perceptron đa tầng (multilayer perceptron)*, hay được viết tắt là *MLP*. Dưới đây, ta mô tả sơ đồ MLP (Fig. 6.1.2).

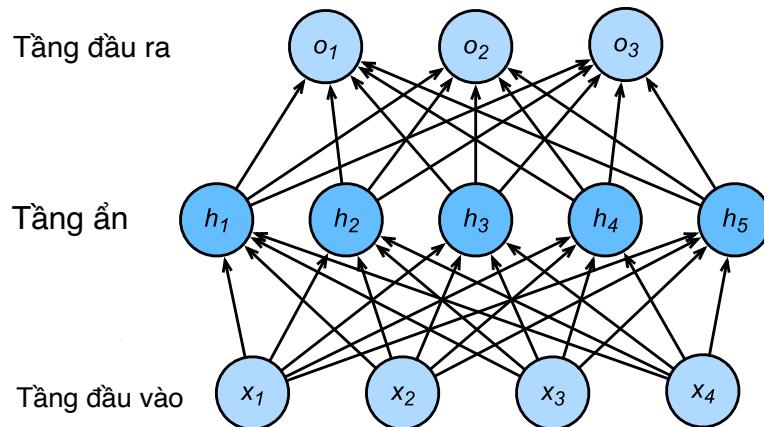


Fig. 6.1.2: Perceptron đa tầng với các tầng ẩn. Ví dụ này chứa một tầng ẩn với 5 nút ẩn.

Perceptron đa tầng này có 4 đầu vào, 3 đầu ra và tầng ẩn của nó chứa 5 nút ẩn. Vì tầng đầu vào không cần bất kỳ tính toán nào, do đó đối với mạng này để tạo đầu ra đòi hỏi phải lập trình các phép tính cho hai tầng còn lại (tầng ẩn và tầng đầu ra). Lưu ý, tất cả tầng này đều kết nối đầy đủ. Mỗi đầu vào đều ảnh hưởng đến mọi no-ron trong tầng ẩn và mỗi no-ron này lại ảnh hưởng đến mọi no-ron trong tầng đầu ra.

## Từ Tuyến tính đến Phi tuyến

Về mặt hình thức, chúng ta tính toán mỗi tầng trong MLP một-tầng-ẩn này như sau:

$$\begin{aligned}\mathbf{h} &= \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1, \\ \mathbf{o} &= \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2, \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{o}).\end{aligned}\tag{6.1.2}$$

Chú ý rằng sau khi thêm tầng này vào, mô hình lập tức yêu cầu chúng ta phải theo dõi và cập nhật thêm hai tập tham số. Vậy thì đổi lại ta sẽ nhận được gì? Bạn có thể bất ngờ khi phát hiện ra rằng—trong mô hình định nghĩa bên trên—chúng ta chẳng thu được lợi ích gì từ những rắc rối thêm vào! Lý do rất đơn giản. Các nút ẩn bên trên được định nghĩa bởi một hàm tuyến tính của các đầu vào, và các đầu ra (tiền Softmax) chỉ là một hàm tuyến tính của các nút ẩn. Một hàm tuyến tính của một hàm tuyến tính bản thân nó cũng chính là một hàm tuyến tính. Hơn nữa, mô hình tuyến tính của chúng ta vốn dĩ đã có khả năng biểu diễn bất kỳ hàm tuyến tính nào rồi.

Ta có thể thấy sự tương đồng về mặt hình thức bằng cách chứng minh rằng với mọi giá trị của các trọng số, ta đều có thể loại bỏ tầng ẩn và tạo ra một mô hình đơn-tầng với các tham số

$\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$  và  $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$ .

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{Wx} + \mathbf{b}. \quad (6.1.3)$$

Để hiện thực được tiềm năng của các kiến trúc đa tầng, chúng ta cần một thành phần quan trọng nữa—một *hàm kích hoạt phi tuyến* theo từng phần tử  $\sigma$  để áp dụng lên từng nút ẩn (theo sau phép biến đổi tuyến tính). Hiện nay, lựa chọn phổ biến nhất cho tính phi tuyến là đơn vị tuyến tính chỉnh lưu (ReLU)  $\max(x, 0)$ . Nhìn chung, với việc sử dụng các hàm kích hoạt này, chúng ta sẽ không thể biến MLP thành một mô hình tuyến tính được nữa.

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1), \\ \mathbf{o} &= \mathbf{W}_2\mathbf{h} + \mathbf{b}_2, \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{o}).\end{aligned}\quad (6.1.4)$$

Để xây dựng các MLP tổng quan hơn, chúng ta có thể tiếp tục chồng thêm các tầng ẩn, ví dụ  $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$  và  $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2)$ , kế tiếp nhau, tạo ra các mô hình có khả năng biểu diễn càng cao (giả sử chiều rộng cố định).

Các MLP có thể biểu diễn được những tương tác phức tạp giữa các đầu vào thông qua các nơ-ron ẩn, các nơ-ron ẩn này phụ thuộc vào giá trị của mỗi đầu vào. Chúng ta có thể dễ dàng thiết kế các nút ẩn để thực hiện bất kỳ tính toán nào, ví dụ, các phép tính logic cơ bản trên một cặp đầu vào. Ngoài ra, với một số hàm kích hoạt cụ thể, các MLP được biết đến rộng rãi như là các bộ xấp xỉ vạn năng. Thậm chí với một mạng chỉ có một tầng ẩn, nếu có đủ số nút (có thể nhiều một cách vô lý) và một tập các trọng số thích hợp, chúng ta có thể mô phỏng bất kỳ một hàm nào. *Thật ra thì việc học được hàm đó mới là phần khó khăn*. Bạn có thể tưởng tượng mạng nơ-ron của mình có nét giống với ngôn ngữ lập trình C. Ngôn ngữ này giống như bất kỳ ngôn ngữ hiện đại nào khác, có khả năng biểu diễn bất kỳ chương trình tính toán nào. Tuy nhiên việc tạo ra một chương trình đáp ứng được các chỉ tiêu kỹ thuật mới là phần việc khó khăn.

Hơn nữa, chỉ vì một mạng đơn-tầng *có thể* học bất kỳ hàm nào không có nghĩa rằng bạn nên cố gắng giải quyết tất cả các vấn đề của mình bằng các mạng đơn-tầng. Thực tế, chúng ta có thể ước lượng các hàm một cách gọn gàng hơn rất nhiều bằng cách sử dụng mạng sâu hơn (thay vì rộng hơn). Chúng ta sẽ đề cập đến những lập luận chặt chẽ hơn trong các chương tiếp theo, nhưng trước tiên hãy lập trình một MLP. Trong ví dụ này, chúng ta lập trình một MLP với hai tầng ẩn và một tầng đầu ra.

## Vector hóa và Minibatch

Giống như trước, chúng ta dùng ma trận  $\mathbf{X}$  để ký hiệu một minibatch các giá trị đầu vào. Các phép tính toán dẫn đến các giá trị đầu ra từ một MLP với hai tầng ẩn khi đó có thể được biểu diễn như sau:

$$\begin{aligned}\mathbf{H}_1 &= \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1), \\ \mathbf{H}_2 &= \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2), \\ \mathbf{O} &= \text{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3).\end{aligned}\quad (6.1.5)$$

Bằng việc lạm dụng ký hiệu một chút, chúng ta định nghĩa hàm phi tuyến  $\sigma$  là một phép toán áp dụng theo từng hàng, tức lần lượt từng điểm dữ liệu một. Cần chú ý rằng ta cũng sử dụng quy ước này cho hàm *softmax* để ký hiệu toán tử tính theo từng hàng. Thông thường, như trong mục này, các hàm kích hoạt không chỉ đơn thuần được áp dụng vào tầng ẩn theo từng hàng mà còn

theo từng phần tử. Điều đó có nghĩa là sau khi tính toán xong phần tuyến tính của tầng, chúng ta có thể tính giá trị kích hoạt của từng nút mà không cần đến giá trị của các nút còn lại. Điều này cũng đúng đối với hầu hết các hàm kích hoạt (tính toán từ chuẩn hóa theo batch được giới thiệu trong Section 9.5 là một trường hợp ngoại lệ của quy tắc này).

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()
```

### 6.1.2 Các hàm Kích hoạt

Các hàm kích hoạt quyết định một nơ-ron có được kích hoạt hay không bằng cách tính tổng có trọng số và cộng thêm hệ số điều chỉnh vào nó. Chúng là các toán tử khả vi và hầu hết đều biến đổi các tín hiệu đầu vào thành các tín hiệu đầu ra theo một cách phi tuyến tính. Bởi vì các hàm kích hoạt rất quan trọng trong học sâu, hãy cùng tìm hiểu sơ lược một số hàm kích hoạt thông dụng.

#### Hàm ReLU

Như đã đề cập trước đó, đơn vị tuyến tính chỉnh lưu (ReLU) là sự lựa chọn phổ biến nhất do tính đơn giản khi lập trình và hiệu quả trong nhiều tác vụ dự đoán. ReLU là một phép biến đổi phi tuyến đơn giản. Cho trước một phần tử  $z$ , ta định nghĩa hàm ReLU là giá trị lớn nhất giữa chính phần tử đó và 0.

$$\text{ReLU}(z) = \max(z, 0). \quad (6.1.6)$$

Nói một cách dễ hiểu hơn, hàm ReLU chỉ giữ lại các phần tử có giá trị dương và loại bỏ tất cả các phần tử có giá trị âm (đặt kích hoạt tương ứng là 0). Để có một cái nhìn khái quát, ta có thể vẽ đồ thị hàm số. Bởi vì ReLU được sử dụng rất phổ biến, ndarray đã hỗ trợ sẵn một toán tử `relu`. Như bạn thấy trong hình, hàm kích hoạt là một hàm tuyến tính từng đoạn.

```
x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.relu(x)
d2l.set figsize((4, 2.5))
d2l.plot(x, y, 'x', 'relu(x)')
```

Khi đầu vào mang giá trị âm thì đạo hàm của hàm ReLU bằng 0 và khi đầu vào mang giá trị dương thì đạo hàm của hàm ReLU bằng 1. Lưu ý rằng, hàm ReLU không khả vi tại 0. Trong trường hợp này, ta mặc định lấy đạo hàm trái (*left-hand-side* - LHS) và nói rằng đạo hàm của hàm ReLU tại 0 thì bằng 0. Chỗ này có thể du di được vì đầu vào thông thường không có giá trị chính xác bằng không. Có một ngạn ngữ xưa nói rằng, nếu ta quan tâm nhiều đến điều kiện biên thì có lẽ ta chỉ đang làm toán (*thuần túy*), chứ không phải đang làm kỹ thuật. Và trong trường hợp này, ngạn ngữ đó đúng. Đồ thị đạo hàm của hàm ReLU như hình dưới.

```
y.backward()
d2l.plot(x, x.grad, 'x', 'grad of relu')
```

Lưu ý rằng, có nhiều biến thể của hàm ReLU, bao gồm ReLU được tham số hóa (pReLU) của He et al., 2015<sup>92</sup>. Phiên bản này thêm một thành phần tuyến tính vào ReLU, do đó một số thông tin vẫn được giữ lại ngay cả khi đổi số là âm.

$$\text{pReLU}(x) = \max(0, x) + \alpha \min(0, x). \quad (6.1.7)$$

Ta sử dụng hàm ReLU bởi vì đạo hàm của nó khá đơn giản: hoặc là chúng biến mất hoặc là chúng cho đổi số đi qua. Điều này làm cho việc tối ưu trở nên tốt hơn và giảm thiểu được nhược điểm *tiêu biến gradient* đã từng gây khó khăn trong các phiên bản trước của mạng nơ-ron (sẽ được đề cập lại sau này).

## Hàm Sigmoid

Hàm sigmoid biến đổi các giá trị đầu vào có miền giá trị thuộc  $\mathbb{R}$  thành các giá trị đầu ra nằm trong khoảng  $(0, 1)$ . Vì vậy, hàm sigmoid thường được gọi là hàm *ép*: nó ép một giá trị đầu vào bất kỳ nằm trong khoảng  $(-\infty, \infty)$  thành một giá trị đầu ra nằm trong khoảng  $(0, 1)$ .

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}. \quad (6.1.8)$$

Các nơ-ron sinh học mà có thể ở một trong hai trạng thái *kích hoạt* hoặc *không kích hoạt*, là một chủ đề mô hình hoá rất được quan tâm từ những nghiên cứu đầu tiên về mạng nơ-ron. Vì vậy mà những người tiên phong trong lĩnh vực này, bao gồm McCulloch<sup>93</sup> và Pitts<sup>94</sup>, những người phát minh ra nơ-ron nhân tạo, đã tập trung nghiên cứu về các đơn vị ngưỡng. Một kích hoạt ngưỡng có giá trị là 0 khi đầu vào của nó ở dưới mức ngưỡng và giá trị là 1 khi đầu vào vượt mức ngưỡng đó.

Khi phương pháp học dựa trên gradient trở nên phổ biến, hàm sigmoid là một lựa chọn tất yếu của đơn vị ngưỡng bởi tính liên tục và khả vi của nó. Hàm sigmoid vẫn là hàm kích hoạt được sử dụng rộng rãi ở các đơn vị đầu ra, khi ta muốn biểu diễn kết quả đầu ra như là xác suất của bài toán phân loại nhị phân (bạn có thể xem sigmoid như một trường hợp đặc biệt của softmax). Tuy nhiên, trong các tầng ẩn, hàm sigmoid hầu hết bị thay thế bằng hàm ReLU vì nó đơn giản hơn và giúp cho việc huấn luyện trở nên dễ dàng hơn. Trong chương “Mạng nơ-ron hồi tiếp” (Section 10.4), chúng tôi sẽ mô tả các mô hình sử dụng đơn vị sigmoid để kiểm soát luồng thông tin theo thời gian.

Dưới đây, ta vẽ đồ thị hàm sigmoid. Cần chú ý rằng, khi đầu vào có giá trị gần bằng 0, hàm sigmoid tiến tới một phép biến đổi tuyến tính.

```
with autograd.record():
    y = npx.sigmoid(x)
d2l.plot(x, y, 'x', 'sigmoid(x)')
```

Đạo hàm của hàm sigmoid được tính bởi phương trình sau:

$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x)). \quad (6.1.9)$$

Đồ thị đạo hàm của hàm sigmoid được vẽ ở dưới. Chú ý rằng khi đầu vào là 0, đạo hàm của hàm sigmoid đạt giá trị lớn nhất là 0.25. Khi đầu vào phân kỳ từ 0 theo một trong hai hướng, đạo hàm sẽ tiến tới 0.

<sup>92</sup> <https://arxiv.org/abs/1502.01852>

<sup>93</sup> [https://en.wikipedia.org/wiki/Warren\\_Sturgis\\_McCulloch](https://en.wikipedia.org/wiki/Warren_Sturgis_McCulloch)

<sup>94</sup> [https://en.wikipedia.org/wiki/Walter\\_Pitts](https://en.wikipedia.org/wiki/Walter_Pitts)

```
y.backward()  
d2l.plot(x, x.grad, 'x', 'grad of sigmoid')
```

### Hàm “Tanh”

Tương tự như hàm sigmoid, hàm tanh (Hyperbolic Tangent) cũng ép các biến đầu vào và biến đổi chúng thành các phần tử nằm trong khoảng -1 và 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \quad (6.1.10)$$

Chúng ta sẽ vẽ hàm tanh như sau. Chú ý rằng nếu đầu vào có giá trị gần bằng 0, hàm tanh sẽ tiến đến một phép biến đổi tuyến tính. Mặc dù hình dạng của hàm tanh trông khá giống hàm sigmoid, hàm tanh lại thể hiện tính đối xứng tâm qua gốc của hệ trục tọa độ.

```
with autograd.record():  
    y = np.tanh(x)  
d2l.plot(x, y, 'x', 'tanh(x)')
```

Đạo hàm của hàm Tanh là:

$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x). \quad (6.1.11)$$

Đạo hàm của hàm tanh được vẽ như sau. Khi đầu vào có giá trị gần bằng 0, đạo hàm của hàm tanh tiến tới giá trị lớn nhất là 1. Tương tự như hàm sigmoid, khi đầu vào phân kỳ từ 0 theo bất kỳ hướng nào, đạo hàm của hàm tanh sẽ tiến đến 0.

```
y.backward()  
d2l.plot(x, x.grad, 'x', 'grad of tanh')
```

Tóm lại, bây giờ chúng ta đã biết cách kết hợp các hàm phi tuyến để xây dựng các kiến trúc mạng nơ-ron đa tầng mạnh mẽ. Một lưu ý bên lề đó là, kiến thức của bạn bây giờ cung cấp cho bạn cách sử dụng một bộ công cụ tương đương với của một người có chuyên môn về học sâu vào những năm 1990. Xét theo một khía cạnh nào đó, bạn còn có lợi thế hơn bất kỳ ai làm việc trong những năm 1990, bởi vì bạn có thể tận dụng triệt để các framework học sâu nguồn mở để xây dựng các mô hình một cách nhanh chóng, chỉ với một vài dòng mã. Trước đây, việc huấn luyện các mạng nơ-ron đòi hỏi các nhà nghiên cứu phải viết đến hàng ngàn dòng mã C và Fortran.

### 6.1.3 Tóm tắt

- Perceptron đa tầng sẽ thêm một hoặc nhiều tầng ẩn được kết nối đầy đủ giữa các tầng đầu ra và các tầng đầu vào nhằm biến đổi đầu ra của tầng ẩn thông qua hàm kích hoạt.
- Các hàm kích hoạt thường được sử dụng bao gồm hàm ReLU, hàm sigmoid, và hàm tanh.

#### 6.1.4 Bài tập

1. Tính đạo hàm của hàm kích hoạt tanh và pReLU.
2. Chứng minh rằng một perceptron đa tầng chỉ sử dụng ReLU (hoặc pReLU) sẽ tạo thành một hàm tuyến tính từng đoạn liên tục.
3. Chứng minh rằng  $\tanh(x) + 1 = 2\text{sigmoid}(2x)$ .
4. Giả sử ta có một perceptron đa tầng mà *không* có tính phi tuyến giữa các tầng. Cụ thể là, giả sử ta có chiều của đầu vào  $d$ , chiều đầu ra  $d$  và tầng ẩn có chiều  $d/2$ . Chứng minh rằng mạng này có ít khả năng biểu diễn hơn một perceptron đơn tầng.
5. Giả sử ta có một hàm phi tuyến tính áp dụng cho từng minibatch mỗi lúc. Việc này sẽ dẫn đến vấn đề gì?

#### 6.1.5 Thảo luận

- Tiếng Anh<sup>95</sup>
- Tiếng Việt<sup>96</sup>

#### 6.1.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đinh Minh Tân
- Phạm Minh Đức
- Vũ Hữu Tiệp
- Nguyễn Lê Quang Nhật
- Nguyễn Minh Thư
- Nguyễn Duy Du
- Phạm Hồng Vinh
- Lê Cao Thăng
- Lý Phi Long
- Lê Khắc Hồng Phúc
- Lâm Ngọc Tâm
- Bùi Nhật Quân

<sup>95</sup> <https://discuss.mxnet.io/t/2338>

<sup>96</sup> <https://forum.machinelearningcoban.com/c/d21>

## 6.2 Lập trình Perceptron Đa tầng từ đầu

Chúng ta đã mô tả perceptron đa tầng (MLPs) ở dạng toán học, giờ hãy cùng thử tự lập trình một mạng như vậy xem sao.

```
from d2l import mxnet as d2l  
from mxnet import gluon, np, npx  
npx.set_np()
```

Để so sánh với kết quả đã đạt được trước đó bằng hồi quy (tuyến tính) softmax (Section 5.6), chúng ta sẽ tiếp tục sử dụng tập dữ liệu phân loại ảnh Fashion-MNIST.

```
batch_size = 256  
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

### 6.2.1 Khởi tạo Tham số Mô hình

Nhắc lại rằng Fashion-MNIST gồm có 10 lớp, mỗi ảnh là một lưới có  $28 \times 28 = 784$  điểm ảnh (đen và trắng). Chúng ta sẽ lại (tạm thời) bỏ qua mối liên hệ về mặt không gian giữa các điểm ảnh, khi đó ta có thể coi nó đơn giản như một tập dữ liệu phân loại với 784 đặc trưng đầu vào và 10 lớp. Để bắt đầu, chúng ta sẽ lập trình một mạng MLP chỉ có một tầng ẩn với 256 nút ẩn. Lưu ý rằng ta có thể coi cả hai đại lượng này là các *siêu tham số* và ta nên thiết lập giá trị cho chúng dựa vào chất lượng trên tập kiểm định. Thông thường, chúng ta sẽ chọn độ rộng của các tầng là các lũy thừa bậc 2 để giúp việc tính toán hiệu quả hơn do cách mà bộ nhớ được cấp phát và địa chỉ hóa ở phần cứng.

Chúng ta sẽ lại biểu diễn các tham số bằng một vài ndarray. Lưu ý rằng với *mỗi tầng*, ta luôn phải giữ một ma trận trọng số và một vector chứa hệ số điều chỉnh. Và như mọi khi, ta gọi hàm attach\_grad để cấp phát bộ nhớ cho gradient (của hàm mất mát) theo các tham số này.

```
num_inputs, num_outputs, num_hiddens = 784, 10, 256  
  
W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens))  
b1 = np.zeros(num_hiddens)  
W2 = np.random.normal(scale=0.01, size=(num_hiddens, num_outputs))  
b2 = np.zeros(num_outputs)  
params = [W1, b1, W2, b2]  
  
for param in params:  
    param.attach_grad()
```

## 6.2.2 Hàm Kích hoạt

Để đảm bảo rằng ta biết mọi thứ hoạt động như thế nào, chúng ta sẽ tự lập trình hàm kích hoạt ReLU bằng cách sử dụng hàm `maximum` thay vì gọi trực tiếp hàm `npx.relu`.

```
def relu(X):
    return np.maximum(X, 0)
```

## 6.2.3 Mô hình

Vì ta đang bỏ qua mối liên hệ về mặt không gian giữa các điểm ảnh, ta reshape mỗi bức ảnh 2D thành một vector phẳng có độ dài `num_inputs`. Cuối cùng, ta có được mô hình chỉ với một vài dòng mã nguồn.

```
def net(X):
    X = X.reshape(-1, num_inputs)
    H = relu(np.dot(X, W1) + b1)
    return np.dot(H, W2) + b2
```

## 6.2.4 Hàm mất mát

Để đảm bảo tính ổn định số học (và cũng bởi ta đã lập trình hàm softmax từ đầu ở Section 5.6), ta sẽ tận dụng luôn các hàm số đã tích hợp sẵn của Gluon để tính softmax và mất mát entropy chéo. Nhắc lại phần thảo luận của chúng ta trước đó về vấn đề rắc rối này (Section 6.1). Chúng tôi khuyến khích bạn đọc quan tâm hãy thử kiểm tra mã nguồn trong `mxnet.gluon.loss`. `SoftmaxCrossEntropyLoss` để hiểu thêm về cách lập trình chi tiết.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

## 6.2.5 Huấn luyện

Thật may, vòng lặp huấn luyện của MLP giống hệt với vòng lặp của hồi quy softmax. Tận dụng gói `d2l`, ta gọi hàm `train_ch3` (xem Section 5.6), đặt số epoch bằng 10 và tốc độ học bằng 0.5

```
num_epochs, lr = 10, 0.5
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
              lambda batch_size: d2l.sgd(params, lr, batch_size))
```

Để đánh giá mô hình sau khi học xong, chúng ta sẽ áp dụng nó vào dữ liệu kiểm tra.

```
d2l.predict_ch3(net, test_iter)
```

Kết quả này tốt hơn một chút so với kết quả trước đây của các mô hình tuyến tính và điều này cho thấy chúng ta đang đi đúng hướng.

## 6.2.6 Tóm tắt

Chúng ta đã thấy việc lập trình một MLP đơn giản khá là dễ dàng, ngay cả khi phải làm thủ công. Tuy vậy, với một số lượng tầng lớn, việc này có thể sẽ trở nên rắc rối (ví dụ như đặt tên và theo dõi các tham số của mô hình, v.v.).

## 6.2.7 Bài tập

1. Thay đổi giá trị của siêu tham số num\\_hiddens và quan sát xem nó ảnh hưởng như thế nào tới kết quả. Giữ nguyên các siêu tham số khác, xác định giá trị tốt nhất của siêu tham số này.
2. Thử thêm vào một tầng ẩn và quan sát xem nó ảnh hưởng như thế nào tới kết quả.
3. Việc thay đổi tốc độ học ảnh hưởng như thế nào tới kết quả? Giữ nguyên kiến trúc mô hình và các siêu tham số khác (bao gồm cả số lượng epoch), tốc độ học nào cho kết quả tốt nhất?
4. Kết quả tốt nhất mà bạn đạt được khi tối ưu hóa tất cả các tham số, gồm tốc độ học, số lượng vòng lặp, số lượng tầng ẩn, số lượng các nút ẩn của mỗi tầng là bao nhiêu?
5. Giải thích tại sao việc phải xử lý nhiều siêu tham số lại gây ra nhiều khó khăn hơn.
6. Đâu là chiến lược thông minh nhất bạn có thể nghĩ ra để tìm kiếm giá trị cho nhiều siêu tham số?

## 6.2.8 Thảo luận

- Tiếng Anh<sup>97</sup>
- Tiếng Việt<sup>98</sup>

## 6.2.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Nguyễn Duy Du
- Phạm Minh Đức

<sup>97</sup> <https://discuss.mxnet.io/t/2339>

<sup>98</sup> <https://forum.machinelearningcoban.com/c/d21>

## 6.3 Cách lập trình súc tích Perceptron Đa tầng

Như bạn đã có thể đoán trước, ta có thể dựa vào thư viện Gluon để lập trình MLP một cách súc tích hơn.

```
from d2l import mxnet as d2l
from mxnet import gluon, init, npx
from mxnet.gluon import nn
npx.set_np()
```

### 6.3.1 Mô hình

So với việc dùng gluon để lập trình hồi quy softmax (Section 5.7), khác biệt duy nhất ở đây là ta thêm *hai* tầng Dense (kết nối đầy đủ), trong khi trước đây ta chỉ có *một*. Tầng đầu tiên là tầng ẩn, chứa 256 nút ẩn và áp dụng hàm kích hoạt ReLU. Còn tầng thứ hai là tầng đầu ra.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Lưu ý rằng như thường lệ, Gluon sẽ tự động suy ra chiều đầu vào còn thiếu cho mỗi tầng.

Vòng lặp huấn luyện ở đây giống *hết* như lúc ta lập trình hồi quy softmax. Lập trình hướng mô-đun như vậy cho phép ta tách các chi tiết liên quan đến kiến trúc của mô hình ra khỏi các mối bận tâm khác.

```
batch_size, num_epochs = 256, 10
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
loss = gluon.loss.SoftmaxCrossEntropyLoss()
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

### 6.3.2 Bài tập

1. Bằng việc thử thêm số lượng các tầng ẩn khác nhau, bạn hãy xem thiết lập nào cho kết quả tốt nhất (giữ nguyên giá trị các tham số và siêu tham số khác)?
2. Bằng việc thử thay đổi các hàm kích hoạt khác nhau, bạn hãy chỉ ra hàm nào mang lại kết quả tốt nhất?
3. Bạn hãy thử các cách khác nhau để khởi tạo trọng số. Phương pháp nào là tốt nhất?

### 6.3.3 Thảo luận

- Tiếng Anh<sup>99</sup>
- Tiếng Việt<sup>100</sup>

### 6.3.4 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lý Phi Long
- Vũ Hữu Tiệp
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức

## 6.4 Lựa Chọn Mô Hình, Dưới Khớp và Quá Khớp

Là những nhà khoa học học máy, mục tiêu của chúng ta là khám phá ra các *khuôn mẫu*. Nhưng làm sao có thể chắc chắn rằng chúng ta đã thực sự khám phá ra một khuôn mẫu *khái quát* chứ không chỉ đơn giản là ghi nhớ dữ liệu. Ví dụ, thử tưởng tượng rằng chúng ta muốn săn lùng các khuôn mẫu liên kết các dấu hiệu di truyền của bệnh nhân và tình trạng mất trí của họ, với nhãn được trích ra từ tập {mất trí nhớ, suy giảm nhận thức mức độ nhẹ, khỏe mạnh}. Bởi vì các gen của mỗi người định dạng họ theo cách độc nhất vô nhị (bỏ qua các cặp song sinh giống hệt nhau), nên việc ghi nhớ toàn bộ tập dữ liệu là hoàn toàn khả thi.

Chúng ta không muốn mô hình của mình nói rằng “Bob kia! Tôi nhớ anh ta! Anh ta bị mất trí nhớ! Lý do tại sao rất đơn giản. Khi triển khai mô hình trong tương lai, chúng ta sẽ gặp các bệnh nhân mà mô hình chưa bao giờ gặp trước đó. Các dự đoán sẽ chỉ có ích khi mô hình của chúng ta thực sự khám phá ra một khuôn mẫu *khái quát*.

Để tóm tắt một cách chính thức hơn, mục tiêu của chúng ta là khám phá các khuôn mẫu mà chúng mô tả được các quy tắc trong tập dữ liệu mà từ đó tập huấn luyện đã được trích ra. Nếu thành công trong nỗ lực này, thì chúng ta có thể đánh giá thành công rủi ro ngay cả đối với các cá nhân mà chúng ta chưa bao giờ gặp phải trước đây. Vấn đề này—làm cách nào để khám phá ra các mẫu mà *khái quát hóa*—là vấn đề nền tảng của học máy.

Nguy hiểm là khi huấn luyện các mô hình, chúng ta chỉ truy cập một tập dữ liệu nhỏ. Các tập dữ liệu hình ảnh công khai lớn nhất chứa khoảng một triệu ảnh. Thường thì chúng ta phải học chỉ từ vài ngàn hoặc vài chục ngàn điểm dữ liệu. Trong một hệ thống bệnh viện lớn, chúng ta có thể truy cập hàng trăm ngàn hồ sơ y tế. Khi làm việc với các tập mẫu hữu hạn, chúng ta gấp phải rủi ro sẽ khám phá ra các mối liên kết rõ ràng mà hóa ra lại không đúng khi thu thập thêm dữ liệu.

Hiện tượng mô hình khớp với dữ liệu huấn luyện chính xác hơn nhiều so với phân phối thực được gọi là quá khớp (*overfitting*), và kỹ thuật sử dụng để chống lại quá khớp được gọi là điều chuẩn (*regularization*). Trong các phần trước, bạn có thể đã quan sát hiệu ứng này khi thử nghiệm với tập

<sup>99</sup> <https://discuss.mxnet.io/t/2340>

<sup>100</sup> <https://forum.machinelearningcoban.com/c/d21>

dữ liệu Fashion-MNIST. Nếu bạn đã sửa đổi cấu trúc mô hình hoặc siêu tham số trong quá trình thử nghiệm, bạn có thể đã nhận ra rằng với đủ các nút, các tầng, và các epoch huấn luyện, mô hình ấy có thể cuối cùng cũng đạt đến sự chính xác hoàn hảo trên tập huấn luyện, ngay cả khi độ chính xác trên dữ liệu kiểm tra giảm đi.

#### 6.4.1 Lỗi huấn luyện và Lỗi khái quát

Để thảo luận hiện tượng này một cách chuyên sâu hơn, ta cần phân biệt giữa *lỗi huấn luyện* (*training error*) và *lỗi khái quát* (*generalization error*). Lỗi huấn luyện là lỗi của mô hình được tính toán trên tập huấn luyện, trong khi đó lỗi khái quát là lỗi kỳ vọng của mô hình khi áp dụng nó cho một luồng vô hạn các điểm dữ liệu mới được lấy từ cùng một phân phối dữ liệu với các mẫu ban đầu.

Vấn đề là *chúng ta không bao giờ có thể tính toán chính xác lỗi khái quát* vì luồng vô hạn dữ liệu chỉ có trong tưởng tượng. Trên thực tế, ta phải *ước tính* lỗi khái quát bằng cách áp dụng mô hình vào một tập kiểm tra độc lập bao gồm các điểm dữ liệu ngẫu nhiên ngoài tập huấn luyện.

Ba thí nghiệm sau 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 chăm chỉ sẽ cố gắng luyện tập tốt và kiểm tra khả năng của cô ấy bằng việc luyện tập những bài kiểm tra của các năm trước. Tuy nhiên, làm tốt các bài kiểm tra trước đây không đảm bảo rằng cô ấy sẽ làm tốt bài kiểm tra thật. Ví dụ, sinh viên có thể cố gắng chuẩn bị bằng cách học từ các câu trả lời cho các câu hỏi. Điều này đòi hỏi sinh viên phải ghi nhớ rất nhiều thứ. Cô ấy có lẽ còn ghi nhớ đáp án cho các bài kiểm tra cũ một cách hoàn hảo. Một học sinh khác có thể chuẩn bị bằng việc cố gắng hiểu lý do mà một số đáp án nhất định được đưa ra. Trong hầu hết các trường hợp, sinh viên sau sẽ làm tốt hơn nhiều.

Tương tự như vậy, hãy xem xét một mô hình đơn giản chỉ sử dụng một bảng tra cứu để trả lời các câu hỏi. Nếu tập hợp các đầu vào cho phép là rời rạc và đủ nhò, thì có lẽ sau khi xem *nhiều* ví dụ huấn luyện, phương pháp này sẽ hoạt động tốt. Tuy nhiên mô hình này không có khả năng thể hiện tốt hơn so với việc đoán ngẫu nhiên khi phải đối mặt với các ví dụ chưa từng gặp trước đây. Trong thực tế, không gian đầu vào là quá lớn để có thể ghi nhớ mọi đáp án tương ứng của từng đầu vào khả dĩ. Ví dụ, hãy xem xét các ảnh  $28 \times 28$  đen trắng. Nếu mỗi điểm ảnh có thể lấy một trong số các giá trị xám trong thang 256, thì có thể có  $256^{784}$  ảnh khác nhau. Điều đó nghĩa là số lượng ảnh độ phân giải thấp còn lớn hơn nhiều so với số lượng nguyên tử trong vũ trụ. Thậm chí nếu có thể xem qua toàn bộ điểm dữ liệu, ta cũng không thể lưu trữ chúng trong bảng tra cứu.

Cuối cùng, hãy xem xét bài toán phân loại kết quả của việc tung đồng xu (lớp 0: ngửa, lớp 1: xấp) dựa trên một số đặc trưng theo ngữ cảnh sẵn có. Bất kể thuật toán nào được đưa ra, lỗi khái quát sẽ luôn là  $\frac{1}{2}$ . Tuy nhiên, đối với hầu hết các thuật toán, lỗi huấn luyện sẽ thấp hơn đáng kể, tùy thuộc vào sự may mắn của ta khi lấy dữ liệu, ngay cả khi ta không có bất kỳ đặc trưng nào! Hãy xem xét tập dữ liệu  $\{0, 1, 1, 1, 0, 1\}$ . Việc không có đặc trưng có thể khiến ta luôn dự đoán lớp *chiếm đa số*, đối với ví dụ này thì đó là 1. Trong trường hợp này, mô hình luôn dự đoán lớp 1 sẽ có lỗi huấn luyện là  $\frac{1}{3}$ , tốt hơn đáng kể so với lỗi khái quát. Khi ta tăng lượng dữ liệu, xác suất nhận được mặt ngửa sẽ dần tiến về  $\frac{1}{2}$  và lỗi huấn luyện sẽ tiến đến lỗi khái quát.

## Lý thuyết Học Thống kê

Bởi khái quát hóa là một vấn đề nền tảng trong học máy, không quá ngạc nhiên khi nhiều nhà toán học và nhà lý thuyết học dành cả cuộc đời để phát triển các lý thuyết hình thức mô tả vấn đề này. Trong [định lý cùng tên](#)<sup>101</sup>, Glivenko và Cantelli đã tìm ra tốc độ học mà tại đó lỗi huấn luyện sẽ hội tụ về lỗi khái quát. Trong chuỗi các bài báo đầu ngành, [Vapnik và Chervonenkis](#)<sup>102</sup> đã mở rộng lý thuyết này cho nhiều lớp hàm tổng quát hơn. Công trình này là nền tảng của ngành [Lý thuyết học thống kê](#)<sup>103</sup>.

Trong một *thiết lập chuẩn cho học có giám sát* – chủ đề lớn nhất xuyên suốt cuốn sách, chúng ta giả sử rằng cả dữ liệu huấn luyện và dữ liệu kiểm tra đều được lấy mẫu *độc lập* từ các phân phối *giống hệt nhau* (*independent & identically distributed*, thường gọi là giả thiết i.i.d.). Điều này có nghĩa là quá trình lấy mẫu dữ liệu không hề có sự *ghi nhớ*. Mẫu lấy ra thứ hai cũng không tương quan với mẫu thứ ba hơn so với mẫu thứ hai triệu.

Trở thành một nhà khoa học học máy giỏi yêu cầu tư duy phản biện, và có lẽ bạn đã có thể “bóc mẽ” được giả thiết này, có thể đưa ra các tình huống thường gặp mà giả thiết này không thỏa mãn. Điều gì sẽ xảy ra nếu chúng ta huấn luyện một mô hình dự đoán tỉ lệ tử vong trên bộ dữ thu thập từ các bệnh nhân tại UCSF, và áp dụng nó trên các bệnh nhân tại Bệnh viện Đa khoa Massachusetts. Các phân phối này đơn giản là không giống nhau. Hơn nữa, việc lấy mẫu có thể có tương quan về mặt thời gian. Sẽ ra sao nếu chúng ta thực hiện phân loại chủ đề cho các bài Tweet. Vòng đời của các tin tức sẽ tạo nên sự phụ thuộc về mặt thời gian giữa các chủ đề được đề cập, vi phạm mọi giả định độc lập thống kê.

Đôi khi, chúng ta có thể bỏ qua một vài vi phạm nhỏ trong giả thiết i.i.d. mà mô hình vẫn có thể làm việc rất tốt. Nhìn chung, gần như tất cả các ứng dụng thực tế đều vi phạm một vài giả thiết i.i.d. nhỏ, nhưng đổi lại ta có được các công cụ rất hữu dụng như nhận dạng khuôn mặt, nhận dạng tiếng nói, dịch ngôn ngữ, v.v.

Các vi phạm khác thì chắc chắn dẫn tới rắc rối. Cùng hình dung ở ví dụ này, ta thử huấn luyện một hệ thống nhận dạng khuôn mặt sử dụng hoàn toàn dữ liệu của các sinh viên đại học và đem đi triển khai như một công cụ giám sát trong viện dưỡng lão. Cách này gần như không khả thi vì ngoại hình giữa hai độ tuổi quá khác biệt.

Trong các mục và chương kế tiếp, chúng ta sẽ đề cập tới các vấn đề gặp phải khi vi phạm giả thiết i.i.d. Hiện tại khi giả thiết i.i.d. thậm chí được đảm bảo, hiểu được sự khái quát hóa cũng là một vấn đề nan giải. Hơn nữa, việc làm sáng tỏ nền tảng lý thuyết để giải thích tại sao các mạng nơ-ron sâu có thể khái quát hóa tốt như vậy vẫn tiếp tục làm đau đầu những bộ óc vĩ đại nhất trong lý thuyết học.

Khi huấn luyện mô hình, ta đang cố gắng tìm kiếm một hàm số khớp với dữ liệu huấn luyện nhất có thể. Nếu hàm số này quá linh hoạt để có thể khớp với các khuôn mẫu giả cũng dễ như với các xu hướng thật trong dữ liệu, thì nó có thể *quá khớp* để có thể tạo ra một mô hình có tính khái quát hóa cao trên dữ liệu chưa nhìn thấy. Đây chính xác là những gì chúng ta muốn tránh (hay ít nhất là kiểm soát được). Rất nhiều kỹ thuật trong học sâu là các phương pháp dựa trên thực nghiệm và thủ thuật để chống lại vấn đề quá khớp.

<sup>101</sup> [https://en.wikipedia.org/wiki/Glivenko–Cantelli\\_theorem](https://en.wikipedia.org/wiki/Glivenko–Cantelli_theorem)

<sup>102</sup> [https://en.wikipedia.org/wiki/Vapnik–Chervonenkis\\_theory](https://en.wikipedia.org/wiki/Vapnik–Chervonenkis_theory)

<sup>103</sup> [https://en.wikipedia.org/wiki/Statistical\\_learning\\_theory](https://en.wikipedia.org/wiki/Statistical_learning_theory)

## Độ Phức tạp của Mô hình

Khi có các mô hình đơn giản và dữ liệu dồi dào, ta kỳ vọng lỗi khái quát sẽ giống với lỗi huấn luyện. Khi làm việc với mô hình phức tạp hơn và ít mẫu huấn luyện hơn, ta kỳ vọng các lỗi huấn luyện giảm xuống nhưng khoảng cách khái quát tăng. Việc chỉ ra chính xác điều gì cấu thành nên độ phức tạp của mô hình là một vấn đề nan giải. Có rất nhiều yếu tố ảnh hưởng đến việc một mô hình có khái quát hóa tốt hay không. Ví dụ một mô hình với nhiều tham số hơn sẽ được xem là phức tạp hơn. Một mô hình mà các tham số có miền giá trị rộng hơn thì được xem là phức tạp hơn. Thông thường với các mạng nơ-ron, ta nghĩ đến một mô hình có nhiều bước huấn luyện là mô hình phức tạp hơn, và mô hình *dùng sớm* là ít phức tạp hơn.

Rất khó để có thể so sánh sự phức tạp giữa các thành viên trong các lớp mô hình khác nhau (ví như cây quyết định so với mạng nơ-ron). Hiện tại, có một quy tắc đơn giản khá hữu ích sau: Một mô hình có thể giải thích các sự kiện bất kỳ thì được các nhà thống kê xem là phức tạp, trong khi một mô hình với năng lực biểu diễn giới hạn nhưng vẫn có thể giải thích tốt được dữ liệu thì hầu như chắc chắn là đúng đắn hơn. Trong triết học, điều này gần với tiêu chí của Popper về **khả năng phủ định**<sup>104</sup> của một lý thuyết khoa học: một lý thuyết tốt nếu nó khớp dữ liệu và nếu có các kiểm định cụ thể có thể dùng để phản chứng nó. Điều này quan trọng bởi vì tất cả các ước lượng thống kê là *post hoc*<sup>105</sup>, tức là ta đánh giá giả thuyết sau khi quan sát các sự thật, do đó dễ bị tác động bởi lỗi ngụy biện cùng tên. Từ bây giờ, ta sẽ đặt triết lý qua một bên và tập trung hơn vào các vấn đề hữu hình.

Trong phần này, để có cái nhìn trực quan, chúng ta sẽ tập trung vào một vài yếu tố có xu hướng ảnh hưởng đến tính khái quát của một lớp mô hình:

1. Số lượng các tham số có thể điều chỉnh. Khi số lượng các tham số có thể điều chỉnh (đôi khi được gọi là *bậc tự do*) lớn thì mô hình sẽ dễ bị quá khớp hơn.
2. Các giá trị được nhận bởi các tham số. Khi các trọng số có miền giá trị rộng hơn, các mô hình dễ bị quá khớp hơn.
3. Số lượng các mẫu huấn luyện. Việc quá khớp một tập dữ liệu chứa chỉ một hoặc hai mẫu rất dễ dàng, kể cả khi mô hình đơn giản. Nhưng quá khớp một tập dữ liệu với vài triệu mẫu đòi hỏi mô hình phải cực kỳ linh hoạt.

### 6.4.2 Lựa chọn Mô hình

Trong học máy, ta thường lựa chọn mô hình cuối cùng sau khi cân nhắc nhiều mô hình ứng viên. Quá trình này được gọi là lựa chọn mô hình. Đôi khi các mô hình được đem ra so sánh khác nhau cơ bản về mặt bản chất (ví như, cây quyết định với các mô hình tuyến tính). Khi khác, ta lại so sánh các thành viên của cùng một lớp mô hình được huấn luyện với các cài đặt siêu tham số khác nhau.

Lấy perceptron đa tầng làm ví dụ, ta mong muốn so sánh các mô hình với số lượng tầng ẩn khác nhau, số lượng nút ẩn khác nhau, và các lựa chọn hàm kích hoạt khác nhau áp dụng vào từng tầng ẩn. Để xác định được mô hình tốt nhất trong các mô hình ứng viên, ta thường sử dụng một tập kiểm định.

<sup>104</sup> <https://en.wikipedia.org/wiki/Falsifiability>

<sup>105</sup> [https://en.wikipedia.org/wiki/Post\\_hoc](https://en.wikipedia.org/wiki/Post_hoc)

## Tập Dữ liệu Kiểm định

Về nguyên tắc, ta không nên sử dụng tập kiểm tra cho đến khi chọn xong tất cả các siêu tham số. Nếu sử dụng dữ liệu kiểm tra trong quá trình lựa chọn mô hình, có một rủi ro là ta có thể quá khớp dữ liệu kiểm tra, và khi đó ta sẽ gặp rắc rối lớn. Nếu quá khớp dữ liệu huấn luyện, ta luôn có thể đánh giá mô hình trên tập kiểm tra để đảm bảo mình “trung thực”. Nhưng nếu quá khớp trên dữ liệu kiểm tra, làm sao chúng ta có thể biết được?

Vì vậy, ta không bao giờ nên dựa vào dữ liệu kiểm tra để lựa chọn mô hình. Tuy nhiên, không thể chỉ dựa vào dữ liệu huấn luyện để lựa chọn mô hình vì ta không thể ước tính lỗi khái quát trên chính dữ liệu được sử dụng để huấn luyện mô hình.

Phương pháp phổ biến để giải quyết vấn đề này là phân chia dữ liệu thành ba phần, thêm một *tập kiểm định* ngoài các tập huấn luyện và kiểm tra.

Trong các ứng dụng thực tế, bức tranh trở nên mập mờ hơn. Mặc dù tốt nhất ta chỉ nên động đến dữ liệu kiểm tra đúng một lần, để đánh giá mô hình tốt nhất hoặc so sánh một số lượng nhỏ các mô hình với nhau, dữ liệu kiểm tra trong thế giới thực hiếm khi bị vứt bỏ chỉ sau một lần sử dụng. Ta hiếm khi có được một tập kiểm tra mới sau mỗi vòng thử nghiệm.

Kết quả là một thực tiễn âm u trong đó ranh giới giữa dữ liệu kiểm định và kiểm tra mơ hồ theo cách đáng lo ngại. Trừ khi có quy định rõ ràng thì, trong các thí nghiệm trong cuốn sách này, ta thật sự đang làm việc với cái được gọi là dữ liệu huấn luyện và dữ liệu kiểm định chứ không có tập kiểm tra thật. Do đó, độ chính xác được báo cáo trong mỗi thử nghiệm thật ra là độ chính xác kiểm định và không phải là độ chính xác của tập kiểm tra thật. Tin tốt là ta không cần quá nhiều dữ liệu trong tập kiểm định. Ta có thể chứng minh rằng sự bất định trong các ước tính thuộc bậc  $\mathcal{O}(n^{-\frac{1}{2}})$ .

## Kiểm định chéo gập K-lần

Khi khan hiếm dữ liệu huấn luyện, có lẽ ta sẽ không thể dành ra đủ dữ liệu để tạo một tập kiểm định phù hợp. Một giải pháp phổ biến để giải quyết vấn đề này là kiểm định chéo gập *K-lần*. Ở phương pháp này, tập dữ liệu huấn luyện ban đầu được chia thành *K* tập con không chồng lên nhau. Sau đó việc huấn luyện và kiểm định mô hình được thực thi *K* lần, mỗi lần huấn luyện trên *K* – 1 tập con và kiểm định trên tập con còn lại (tập không được sử dụng để huấn luyện trong lần đó). Cuối cùng, lỗi huấn luyện và lỗi kiểm định được ước lượng bằng cách tính trung bình các kết quả thu được từ *K* thí nghiệm.

### 6.4.3 Dưới khớp hay Quá khớp?

Khi so sánh lỗi huấn luyện và lỗi kiểm định, ta cần lưu ý hai trường hợp thường gặp sau: Đầu tiên, ta sẽ muốn chú ý trường hợp lỗi huấn luyện và lỗi kiểm định đều lớn nhưng khoảng cách giữa chúng lại nhỏ. Nếu mô hình không thể giảm thiểu lỗi huấn luyện, điều này có nghĩa là mô hình quá đơn giản (tức không đủ khả năng biểu diễn) để có thể xác định được khuôn mẫu mà ta đang cố mô hình hóa. Hơn nữa, do khoảng cách khái quát giữa lỗi huấn luyện và lỗi kiểm định nhỏ, ta có lý do để tin rằng phương án giải quyết là một mô hình phức tạp hơn. Hiện tượng này được gọi là dưới khớp (*underfitting*).

Mặt khác, như ta đã thảo luận ở phía trên, ta cũng muốn chú ý tới trường hợp lỗi huấn luyện thấp hơn lỗi kiểm định một cách đáng kể, một biểu hiện của sự quá khớp nặng. Lưu ý rằng quá khớp không phải luôn là điều xấu. Đặc biệt là với học sâu, ta đều biết rằng mô hình dự đoán tốt nhất

thường đạt chất lượng tốt hơn hẳn trên dữ liệu huấn luyện so với dữ liệu kiểm định. Sau cùng, ta thường quan tâm đến lỗi kiểm định hơn khoảng cách giữa lỗi huấn luyện và lỗi kiểm định.

Việc ta đang quá khớp hay dưới khớp có thể phụ thuộc vào cả độ phức tạp của mô hình lẫn kích thước của tập dữ liệu huấn luyện có sẵn, và hai vấn đề này sẽ được thảo luận ngay sau đây.

## Độ phức tạp Mô hình

Để có thể hình dung một cách trực quan hơn về mối quan hệ giữa quá khớp và độ phức tạp mô hình, ta sẽ đưa ra một ví dụ sử dụng đa thức. Cho một tập dữ liệu huấn luyện có một đặc trưng duy nhất  $x$  và nhãn  $y$  tương ứng có giá trị thực, ta thử tìm bậc  $d$  của đa thức

$$\hat{y} = \sum_{i=0}^d x^i w_i \quad (6.4.1)$$

để ước tính nhãn  $y$ . Đây đơn giản là một bài toán hồi quy tuyến tính trong đó các đặc trưng được tính bằng cách lấy mũ của  $x$ ,  $w_i$  là trọng số của mô hình, vì  $x^0 = 1$  với mọi  $x$  nên  $w_0$  là hệ số điều chỉnh. Vì đây là bài toán hồi quy tuyến tính, ta có thể sử dụng bình phương sai số làm hàm mất mát.

Hàm đa thức bậc cao phức tạp hơn hàm đa thức bậc thấp, vì đa thức bậc cao có nhiều tham số hơn và miền lựa chọn hàm số cũng rộng hơn. Nếu giữ nguyên tập dữ liệu huấn luyện, các hàm đa thức bậc cao hơn sẽ luôn đạt được lỗi huấn luyện thấp hơn (ít nhất là bằng) so với đa thức bậc thấp hơn. Trong thực tế, nếu mọi điểm dữ liệu có các giá trị  $x$  riêng biệt, một hàm đa thức có bậc bằng với số điểm dữ liệu đều có thể khớp một cách hoàn hảo với tập huấn luyện. Mỗi quan hệ giữa bậc của đa thức với hai hiện tượng dưới khớp và quá khớp được biểu diễn trong Fig. 6.4.1.

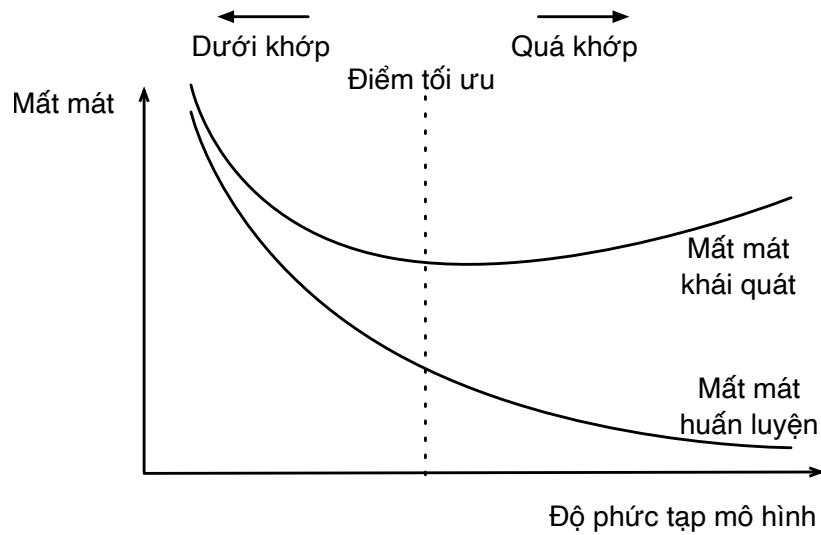


Fig. 6.4.1: Ảnh hưởng của Độ phức tạp Mô hình tới Dưới khớp và Quá khớp

## Kích thước Tập dữ liệu

Một lưu ý quan trọng khác cần ghi nhớ là kích thước tập dữ liệu. Với một mô hình cố định, tập dữ liệu càng ít mẫu thì càng có nhiều khả năng gặp phải tình trạng quá khớp với mức độ nghiêm trọng hơn. Khi số lượng dữ liệu tăng lên, lỗi khái quát sẽ có xu hướng giảm. Hơn nữa, trong hầu hết các trường hợp, nhiều dữ liệu không bao giờ là thừa. Trong một tác vụ với một *phân phối* dữ liệu cố định, ta có thể quan sát được mối quan hệ giữa độ phức tạp của mô hình và kích thước tập dữ liệu. Khi có nhiều dữ liệu, thử khớp một mô hình phức tạp hơn thường sẽ mang lại nhiều lợi ích. Khi dữ liệu không quá nhiều, mô hình đơn giản sẽ là lựa chọn tốt hơn. Đối với nhiều tác vụ, học sâu chỉ tốt hơn các mô hình tuyến tính khi có sẵn hàng ngàn mẫu huấn luyện. Sự thành công hiện nay của học sâu phần nào dựa vào sự phong phú của các tập dữ liệu khổng lồ từ các công ty hoạt động trên internet, từ các thiết bị lưu trữ giá rẻ, các thiết bị được nối mạng và rộng hơn là việc số hóa nền kinh tế.

### 6.4.4 Hồi quy Đa thức

Bây giờ ta có thể khám phá một cách tương tác những khái niệm này bằng cách khớp đa thức với dữ liệu. Để bắt đầu ta sẽ nhập các gói thư viện thường dùng.

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

## Tạo ra Tập dữ liệu

Đầu tiên ta cần dữ liệu. Cho  $x$ , ta sẽ sử dụng đa thức bậc ba ở dưới đây để tạo nhãn cho tập dữ liệu huấn luyện và tập kiểm tra:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ với } \epsilon \sim \mathcal{N}(0, 0.1). \quad (6.4.2)$$

Số hạng nhiễu  $\epsilon$  tuân theo phân phối chuẩn (phân phối Gauss) với giá trị trung bình bằng 0 và độ lệch chuẩn bằng 0.1. Ta sẽ tạo 100 mẫu cho mỗi tập huấn luyện và tập kiểm tra.

```
maxdegree = 20 # Maximum degree of the polynomial
n_train, n_test = 100, 100 # Training and test dataset sizes
true_w = np.zeros(maxdegree) # 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))
features = np.random.shuffle(features)
poly_features = np.power(features, np.arange(maxdegree).reshape(1, -1))
poly_features = poly_features /
    npx.gamma(np.arange(maxdegree) + 1).reshape(1, -1)
labels = np.dot(poly_features, true_w)
labels += np.random.normal(scale=0.1, size=labels.shape)
```

Khi tối ưu hóa, ta thường muốn tránh các giá trị rất lớn của gradient, mất mát, v.v. Đây là lý do tại sao các đòn thức lưu trong `poly_features` được chuyển đổi giá trị từ  $x^i$  thành  $\frac{1}{i!}x^i$ . Nó cho phép ta

tránh các giá trị quá lớn với số mũ bậc cao  $i$ . Phép tính gai thừa được lập trình trong Gluon bằng hàm Gamma, với  $n! = \Gamma(n + 1)$ .

Hãy xét hai mẫu đầu tiên trong tập dữ liệu được tạo. Về mặt kỹ thuật giá trị 1 là một đặc trưng, cụ thể là đặc trưng không đổi tương ứng với hệ số điều chỉnh.

```
features[:2], poly_features[:2], labels[:2]
```

## Huấn luyện và Kiểm tra Mô hình

Trước tiên ta lập trình hàm để tính giá trị mất mát của dữ liệu cho trước.

```
# Saved in the d2l package for later use
def evaluate_loss(net, data_iter, loss):
    """Evaluate the loss of a model on the given dataset."""
    metric = d2l.Accumulator(2) # sum_loss, num_examples
    for X, y in data_iter:
        metric.add(loss(net(X), y).sum(), y.size)
    return metric[0] / metric[1]
```

Giờ ta định nghĩa hàm huấn luyện.

```
def train(train_features, test_features, train_labels, test_labels,
          num_epochs=1000):
    loss = gluon.loss.L2Loss()
    net = nn.Sequential()
    # Switch off the bias since we already catered for it in the polynomial
    # features
    net.add(nn.Dense(1, use_bias=False))
    net.initialize()
    batch_size = min(10, train_labels.shape[0])
    train_iter = d2l.load_array((train_features, train_labels), batch_size)
    test_iter = d2l.load_array((test_features, test_labels), batch_size,
                               is_train=False)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                           {'learning_rate': 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(1, num_epochs+1):
        d2l.train_epoch_ch3(net, train_iter, loss, trainer)
        if epoch % 50 == 0:
            animator.add(epoch, (evaluate_loss(net, train_iter, loss),
                                evaluate_loss(net, test_iter, loss)))
    print('weight:', net[0].weight.data().asnumpy())
```

## Khớp Hàm Đa thức Bậc Ba (dạng chuẩn)

Ta sẽ bắt đầu với việc sử dụng hàm đa thức bậc ba, cùng bậc với hàm tạo dữ liệu. Kết quả cho thấy cả lỗi huấn luyện và lỗi kiểm tra của mô hình đều thấp. Các tham số của mô hình được huấn luyện cũng gần với giá trị thật  $w = [5, 1.2, -3.4, 5.6]$ .

```
# Pick the first four dimensions, i.e., 1, x, x^2, x^3 from the polynomial
# features
train(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
      labels[:n_train], labels[n_train:])
```

## Khớp hàm tuyến tính (Dưới khớp)

Hãy xem lại việc khớp hàm tuyến tính. Sau sự sụt giảm ở những epoch đầu, việc giảm thêm lỗi huấn luyện của mô hình đã trở nên khó khăn. Sau khi epoch cuối cùng kết thúc, lỗi huấn luyện vẫn còn cao. Khi được sử dụng để khớp các khuôn mẫu phi tuyến (như hàm đa thức bậc ba trong trường hợp này), các mô hình tuyến tính dễ bị dưới khớp.

```
# Pick the first four dimensions, i.e., 1, x from the polynomial features
train(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
      labels[:n_train], labels[n_train:])
```

## Thiếu dữ liệu huấn luyện (Quá khớp)

Bây giờ, hãy thử huấn luyện mô hình sử dụng một đa thức với bậc rất cao. Trong trường hợp này, mô hình không có đủ dữ liệu để học được rằng các hệ số bậc cao nên có giá trị gần với không. Vì vậy, mô hình quá phức tạp của ta sẽ dễ bị ảnh hưởng bởi nhiễu ở trong dữ liệu huấn luyện. Dĩ nhiên, lỗi huấn luyện trong trường hợp này sẽ thấp (thậm chí còn thấp hơn cả khi chúng ta có được mô hình thích hợp!) nhưng lỗi kiểm tra sẽ cao.

Thử nghiệm với các độ phức tạp của mô hình (`n_degree`) và các kích thước của tập huấn luyện (`n_subset`) khác nhau để thấy được một cách trực quan điều gì đang diễn ra.

```
n_subset = 100 # Subset of data to train on
n_degree = 20 # Degree of polynomials
train(poly_features[1:n_subset, 0:n_degree],
      poly_features[n_train:, 0:n_degree], labels[1:n_subset],
      labels[n_train:])
```

Ở các chương sau, chúng ta sẽ tiếp tục thảo luận về các vấn đề quá khớp và các phương pháp đối phó, ví dụ như suy giảm trọng số hay dropout.

#### 6.4.5 Tóm tắt

- Bởi vì lỗi khái quát không thể được ước lượng dựa trên lỗi huấn luyện, nên việc chỉ đơn thuần cực tiểu hóa lỗi huấn luyện sẽ không nhất thiết đồng nghĩa với việc cực tiểu hóa lỗi khái quát. Các mô hình học máy cần phải được bảo vệ khỏi việc quá khớp để giảm thiểu lỗi khái quát.
- Một tập kiểm định có thể được sử dụng cho việc lựa chọn mô hình (với điều kiện là tập này không được sử dụng quá nhiều).
- Dưới khớp có nghĩa là mô hình không có khả năng giảm lỗi huấn luyện, còn quá khớp là kết quả của việc lỗi huấn luyện của mô hình thấp hơn nhiều so với lỗi kiểm tra.
- Chúng ta nên chọn một mô hình phức tạp vừa phải và tránh việc sử dụng tập huấn luyện không có đủ số mẫu.

#### 6.4.6 Bài tập

1. Bạn có thể giải bài toán hồi quy đa thức một cách chính xác không? Gợi ý: sử dụng đại số tuyến tính.
2. Lựa chọn mô hình cho các đa thức
  - Vẽ đồ thị biểu diễn lỗi huấn luyện và độ phức tạp của mô hình (bậc của đa thức). Bạn quan sát được gì?
  - Vẽ đồ thị biểu diễn lỗi kiểm tra trong trường hợp này.
  - Tạo một đồ thị tương tự nhưng với hàm của lượng dữ liệu.
3. Điều gì sẽ xảy ra nếu bạn bỏ qua việc chuẩn hóa các đặc trưng đa thức  $x^i$  với  $1/i!$ . Bạn có thể sửa chữa vấn đề này bằng cách nào khác không?
4. Độ mẩy của đa thức giảm được tỉ lệ lỗi huấn luyện về 0?
5. Bạn có bao giờ kỳ vọng thấy được lỗi khái quát bằng 0?

#### 6.4.7 Thảo luận

- Tiếng Anh<sup>106</sup>
- Tiếng Việt<sup>107</sup>

#### 6.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Phạm Minh Đức

<sup>106</sup> <https://discuss.mxnet.io/t/2341>

<sup>107</sup> <https://forum.machinelearningcoban.com/c/d21>

- Nguyễn Văn Tâm
- Vũ Hữu Tiệp
- Phạm Hồng Vinh
- Bùi Nhật Quân
- Lý Phi Long
- Nguyễn Duy Du

## 6.5 Suy giảm trọng số

Bởi chúng ta đã mô tả xong vấn đề quá khớp, giờ ta có thể tìm hiểu một vài kỹ thuật tiêu chuẩn trong việc điều chỉnh mô hình. Nhắc lại rằng chúng ta luôn có thể giảm thiểu hiện tượng quá khớp bằng cách thu thập thêm dữ liệu huấn luyện, nhưng trong trường hợp ngắn hạn thì giải pháp này có thể không khả thi do quá tốn kém, lãng phí thời gian, hoặc nằm ngoài khả năng của ta. Hiện tại, chúng ta có thể giả sử rằng ta đã thu thập được một lượng tối đa dữ liệu chất lượng và sẽ tập trung vào các kỹ thuật điều chỉnh.

Nhắc lại rằng trong ví dụ về việc khớp đường cong đa thức (Section 6.4), chúng ta có thể giới hạn năng lực của mô hình bằng việc đơn thuần điều chỉnh số bậc của đa thức. Đúng như vậy, giới hạn số đặc trưng là một kỹ thuật phổ biến để tránh hiện tượng quá khớp. Tuy nhiên, việc đơn thuần loại bỏ các đặc trưng có thể hơi quá mức cần thiết. Quay lại với ví dụ về việc khớp đường cong đa thức, hãy xét chuyện gì sẽ xảy ra với đầu vào nhiều chiều. Ta mở rộng đa thức cho dữ liệu đa biến bằng việc thêm các *đơn thức*, hay nói đơn giản là thêm tích của lũy thừa các biến. Bậc của một đơn thức là tổng của các số mũ. Ví dụ,  $x_1^2x_2$ , và  $x_3x_5^2$  đều là các đơn thức bậc 3.

Lưu ý rằng số lượng đơn thức bậc  $d$  tăng cực kỳ nhanh theo  $d$ . Với  $k$  biến, số lượng các đơn thức bậc  $d$  là  $\binom{k-1+d}{k-1}$ . Chỉ một thay đổi nhỏ về số bậc, ví dụ từ 2 lên 3 cũng sẽ tăng độ phức tạp của mô hình một cách chóng mặt. Do vậy, chúng ta cần có một công cụ tốt hơn để điều chỉnh độ phức tạp của hàm số.

### 6.5.1 Điều chuẩn Chuẩn Bình phương

*Suy giảm trọng số* (thường được gọi là điều chuẩn L2), có thể là kỹ thuật được sử dụng rộng rãi nhất để điều chỉnh các mô hình học máy có tham số. Kỹ thuật này dựa trên một quan sát cơ bản: trong tất cả các hàm  $f$ , hàm  $f = 0$  (gán giá trị 0 cho tất cả các đầu vào) có lẽ là hàm *đơn giản nhất* và ta có thể đo độ phức tạp của hàm số bằng khoảng cách giữa nó và giá trị không. Nhưng cụ thể thì ta đo khoảng cách giữa một hàm số và số không như thế nào? Không chỉ có duy nhất một câu trả lời đúng. Trong thực tế, có những nhánh toán học được dành riêng để trả lời câu hỏi này, bao gồm một vài nhánh con của giải tích hàm và lý thuyết không gian Banach.

Một cách đơn giản để đo độ phức tạp của hàm tuyến tính  $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$  là dựa vào chuẩn của vector trọng số, ví dụ như  $\|\mathbf{w}\|^2$ . Phương pháp phổ biến nhất để đảm bảo rằng ta sẽ có một vector trọng số nhỏ là thêm chuẩn của nó (đóng vai trò như một thành phần phạt) vào bài toán cực tiểu hóa hàm mất mát. Do đó, ta thay thế mục tiêu ban đầu: *cực tiểu hóa hàm mất mát dự đoán trên nhãn huấn luyện*, bằng mục tiêu mới, *cực tiểu hóa tổng của hàm mất mát dự đoán và thành phần phạt*. Ngày nay, nếu vector trọng số tăng quá lớn, thuật toán học sẽ *tập trung giảm thiểu chuẩn trọng số  $\|\mathbf{w}\|^2$*  thay vì giảm thiểu lỗi huấn luyện. Đó chính xác là những gì ta muốn. Để minh họa mọi thứ bằng

mã, hãy xét lại ví dụ hồi quy tuyến tính trong Section 5.1. Ở đó, hàm mất mát được định nghĩa như sau:

$$l(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n \frac{1}{2} \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2. \quad (6.5.1)$$

Nhắc lại  $\mathbf{x}^{(i)}$  là các quan sát,  $y^{(i)}$  là các nhãn và  $(\mathbf{w}, b)$  lần lượt là trọng số và hệ số điều chỉnh. Để phạt độ lớn của vector trọng số, bằng cách nào đó ta phải cộng thêm  $\|\mathbf{w}\|^2$  vào hàm mất mát, nhưng mô hình nên đánh đổi hàm mất mát thông thường với thành phần phạt mới này như thế nào? Trong thực tế, ta mô tả sự đánh đổi này thông qua *hằng số điều chuẩn*  $\lambda > 0$ , một siêu tham số không âm mà ta khớp được bằng cách sử dụng dữ liệu kiểm định:

$$l(\mathbf{w}, b) + \frac{\lambda}{2} \|\mathbf{w}\|^2. \quad (6.5.2)$$

Với  $\lambda = 0$ , ta thu lại được hàm mất mát gốc. Với  $\lambda > 0$ , ta giới hạn độ lớn của  $\|\mathbf{w}\|$ . Bạn đọc nào tinh ý có thể tự hỏi tại sao ta dùng chuẩn bình phương chứ không phải chuẩn thông thường (nghĩa là khoảng cách Euclidean). Ta làm điều này để thuận tiện cho việc tính toán. Bằng cách bình phương chuẩn L2, ta khử được căn bậc hai, chỉ còn lại tổng bình phương từng thành phần của vector trọng số. Điều này giúp việc tính đạo hàm của thành phần phạt dễ dàng hơn (tổng các đạo hàm bằng đạo hàm của tổng).

Hơn nữa, có thể bạn sẽ hỏi tại sao ta lại dùng chuẩn L2 ngay từ đầu chứ không phải là chuẩn L1.

Trong thực tế ngành thống kê, các lựa chọn khác đều hợp lệ và phổ biến. Trong khi các mô hình tuyến tính được điều chuẩn-L2 tạo thành thuật toán *hồi quy ridge (ridge regression)*, hồi quy tuyến tính được điều chuẩn-L1 cũng là một mô hình cơ bản trong thống kê (thường được gọi là *hồi quy lasso—lasso regression*).

Một cách tổng quát, chuẩn  $\ell_2$  chỉ là một trong vô số các chuẩn được gọi chung là chuẩn-p, và sau này bạn sẽ có thể gặp một vài chuẩn như vậy. Thông thường, với một số  $p$ , chuẩn  $\ell_p$  được định nghĩa là:

$$\|\mathbf{w}\|_p^p := \sum_{i=1}^d |w_i|^p. \quad (6.5.3)$$

Một lý do để sử dụng chuẩn L2 là vì nó phạt nặng những thành phần lớn của vector trọng số. Việc này khiến thuật toán học thiên vị các mô hình có trọng số được phân bổ đồng đều cho một số lượng lớn các đặc trưng. Trong thực tế, điều này có thể giúp giảm ảnh hưởng từ lỗi đo lường của từng biến đơn lẻ. Ngược lại, lượng phạt L1 hướng đến các mô hình mà trọng số chỉ tập trung vào một số lượng nhỏ các đặc trưng, và ta có thể muốn điều này vì một vài lý do khác.

Việc cập nhật hạ gradient ngẫu nhiên cho hồi quy được chuẩn hóa L2 được tiến hành như sau:

$$\mathbf{w} \leftarrow (1 - \eta \lambda) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left( \mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \quad (6.5.4)$$

Như trước đây, ta cập nhật  $\mathbf{w}$  dựa trên hiệu của giá trị ước lượng và giá trị quan sát được. Tuy nhiên, ta cũng sẽ thu nhỏ độ lớn của  $\mathbf{w}$  về 0. Đó là lý do tại sao phương pháp này còn đôi khi được gọi là “suy giảm trọng số”: nếu chỉ có số hạng phạt, thuật toán tối ưu sẽ *suy giảm* các trọng số ở từng bước huấn luyện. Trái ngược với việc lựa chọn đặc trưng, suy giảm trọng số cho ta một cơ chế liên tục để thay đổi độ phức tạp của  $f$ . Giá trị  $\lambda$  nhỏ tương ứng với việc  $\mathbf{w}$  không bị ràng buộc, còn giá trị  $\lambda$  lớn sẽ ràng buộc  $\mathbf{w}$  một cách đáng kể. Còn việc có nên thêm lượng phạt cho hệ số điều chỉnh tương ứng  $b^2$  hay không thì tùy thuộc ở mỗi cách lập trình, và có thể khác nhau giữa các tầng của mạng nơ-ron. Thông thường, ta không điều chuẩn hệ số điều chỉnh tại tầng đầu ra của mạng.

## 6.5.2 Hồi quy Tuyến tính nhiều chiều

Ta có thể minh họa các ưu điểm của suy giảm trọng số so với lựa chọn đặc trưng thông qua một ví dụ đơn giản với dữ liệu tự tạo. Đầu tiên, ta tạo ra dữ liệu giống như trước đây

$$y = 0.05 + \sum_{i=1}^d 0.01x_i + \epsilon \text{ với } \epsilon \sim \mathcal{N}(0, 0.01). \quad (6.5.5)$$

lựa chọn nhãn là một hàm tuyến tính của các đầu vào, bị biến dạng bởi nhiễu Gauss với trung bình bằng không và phương sai bằng 0.01. Để làm cho hiệu ứng của việc quá khớp trở nên rõ ràng, ta có thể tăng số chiều của bài toán lên  $d = 200$  và làm việc với một tập huấn luyện nhỏ bao gồm chỉ 20 mẫu.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

n_train, n_test, num_inputs, batch_size = 20, 100, 200, 5
true_w, true_b = np.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)
```

## 6.5.3 Lập trình từ đầu

Tiếp theo, chúng ta sẽ lập trình suy giảm trọng số từ đầu, chỉ đơn giản bằng cách cộng thêm bình phương lượng phạt  $\ell_2$  vào hàm mục tiêu ban đầu.

### Khởi tạo Tham số Mô hình

Đầu tiên, chúng ta khai báo một hàm để khởi tạo tham số cho mô hình một cách ngẫu nhiên và chạy attach\_grad với mỗi tham số để cấp phát bộ nhớ cho gradient mà ta sẽ tính toán.

```
def init_params():
    w = np.random.normal(scale=1, size=(num_inputs, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    return [w, b]
```

## Định nghĩa Lượng phạt Chuẩn $\ell_2$

Có lẽ cách thuận tiện nhất để lập trình lượng phạt này là bình phương tất cả các phần tử ngay tại chỗ và cộng chúng lại với nhau. Ta đem chia với 2 theo quy ước (khi ta tính đạo hàm của hàm bậc hai, 2 và 1/2 sẽ loại trừ nhau, đảm bảo biểu thức cập nhật trông đơn giản, dễ nhìn).

```
def l2_penalty(w):
    return (w**2).sum() / 2
```

## Định nghĩa hàm Huấn luyện và Kiểm tra

Đoạn mã nguồn sau thực hiện việc khớp mô hình trên tập huấn luyện và đánh giá nó trên tập kiểm tra. Mạng tuyến tính và hàm mất mát bình phương không thay đổi gì so với chương trước, vì vậy ta chỉ cần nhập chúng qua d2l.linreg và d2l.squared\_loss. Thay đổi duy nhất ở đây là hàm mất mát có thêm lượng 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=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs + 1):
        for X, y in train_iter:
            with autograd.record():
                # The L2 norm penalty term has been added, and broadcasting
                # makes l2_penalty(w) a vector whose length is batch_size
                l = loss(net(X), y) + lambd * l2_penalty(w)
            l.backward()
            d2l.sgd([w, b], lr, batch_size)
        if epoch % 5 == 0:
            animator.add(epoch, (d2l.evaluate_loss(net, train_iter, loss),
                                 d2l.evaluate_loss(net, test_iter, loss)))
    print('l1 norm of w:', np.abs(w).sum())
```

## Huấn luyện không Điều chuẩn

Giờ chúng ta sẽ chạy đoạn mã này với  $\text{lambd} = 0$ , vô hiệu hóa suy giảm trọng số. Hãy để ý tới việc quá khớp nặng, lỗi huấn luyện giảm nhưng lỗi kiểm tra thì không—một trường hợp điển hình của hiện tượng quá khớp.

```
train(lambd=0)
```

## Sử dụng Suy giảm Trọng số

Dưới đây, chúng ta huấn luyện mô hình với trọng số bị suy giảm mạnh. Cần chú ý rằng lỗi huấn luyện tăng nhưng lỗi kiểm định lại giảm. Đây chính xác là hiệu ứng mà chúng ta mong đợi từ việc điều chuẩn. Bạn có thể tự kiểm tra xem chuẩn  $\ell_2$  của các trọng số  $w$  có thực sự giảm hay không, như là một bài tập.

```
train(lambd=3)
```

### 6.5.4 Cách lập trình súc tích

Bởi vì suy giảm trọng số có ở khắp mọi nơi trong việc tối ưu mạng nơ-ron, Gluon giúp cho việc áp dụng kỹ thuật này trở nên rất thuận tiện, bằng cách tích hợp suy giảm trọng số vào chính giải thuật tối ưu để có thể kết hợp với bất kì hàm mất mát nào. Hơn nữa, việc tích hợp này cũng đem lại lợi ích về mặt tính toán, cho phép ta sử dụng các thủ thuật lập trình để thêm suy giảm trọng số vào thuật toán mà không làm tăng tổng chi phí tính toán. Điều này khả thi bởi vì tại mỗi bước cập nhật, phần suy giảm trọng số chỉ phụ thuộc vào giá trị hiện tại của mỗi tham số và bộ tối ưu hoá đăng nào cũng phải đụng tới chúng.

Trong đoạn mã nguồn sau đây, chúng ta chỉ định trực tiếp siêu tham số trong suy giảm trọng số thông qua giá trị `wd` khi khởi tạo Trainer. Theo mặc định, Gluon suy giảm đồng thời cả trọng số và hệ số điều chỉnh. Cần chú ý rằng siêu tham số `wd` sẽ được nhân với `wd_mult` khi cập nhật các tham số mô hình. Như vậy, nếu chúng ta đặt `wd_mult` bằng 0, tham số hệ số điều chỉnh `b` sẽ không suy giảm.

```
def train_gluon(wd):
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=1))
    loss = gluon.loss.L2Loss()
    num_epochs, lr = 100, 0.003
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'wd': wd})
    # The bias parameter has not decayed. Bias names generally end with "bias"
    net.collect_params('.*bias').setattr('wd_mult', 0)

    animator = d2l.Animator(xlabel='epochs', ylabel='loss', yscale='log',
                             xlim=[1, num_epochs], legend=['train', 'test'])
    for epoch in range(1, num_epochs+1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
            if epoch % 5 == 0:
                animator.add(epoch, (d2l.evaluate_loss(net, train_iter, loss),
                                     d2l.evaluate_loss(net, test_iter, loss)))
    print('L1 norm of w:', np.abs(net[0].weight.data()).sum())
```

Các đồ thị này nhìn giống hệt với những đồ thị khi chúng ta lập trình suy giảm trọng số từ đầu. Tuy nhiên, chúng chạy nhanh hơn rõ rệt và dễ lập trình hơn, một lợi ích đáng kể khi làm việc với các bài toán lớn.

```
train_gluon(0)
```

```
train_gluon(3)
```

Tới giờ, chúng ta mới chỉ đề cập đến một ý niệm về những gì cấu thành nên một hàm *tuyến tính* đơn giản. Hơn nữa, những gì cấu thành nên một hàm *phi tuyến* đơn giản, thậm chí còn phức tạp hơn. Ví dụ, [Tái tạo các không gian kernel Hilbert \(RKHS\)](#)<sup>108</sup> cho phép chúng ta áp dụng các công cụ được giới thiệu cho các hàm tuyến tính trong một ngữ cảnh phi tuyến. Không may là, các giải thuật dựa vào RKHS thường không thể nhân rộng và hoạt động hiệu quả trên bộ dữ liệu lớn, đa chiều. Dựa trên một thực nghiệm đơn giản, chúng ta mặc định sẽ áp dụng phương pháp suy giảm trọng số cho tất cả các tầng của mạng học sâu trong quyển sách này.

### 6.5.5 Tóm tắt

- Điều chuẩn là một phương pháp phổ biến để giải quyết vấn đề quá khớp. Nó thêm một lượng phạt vào hàm mất mát trong tập huấn luyện để giảm thiểu độ phức tạp của mô hình.
- Một cách cụ thể để giữ mô hình đơn giản là sử dụng suy giảm trọng số với lượng phạt  $\ell_2$ . Điều này dẫn đến việc giá trị trọng số sẽ suy giảm trong các bước cập nhật của giải thuật học.
- Gluon cung cấp tính năng suy giảm trọng số tự động trong bộ tối ưu hoá bằng cách thiết lập siêu tham số `wd`.
- Bạn có thể dùng nhiều bộ tối ưu hoá khác nhau trong cùng một vòng lặp huấn luyện, chẳng hạn như để dùng chúng cho các tập tham số khác nhau.

### 6.5.6 Bài tập

1. Thủ nghiệm với giá trị của  $\lambda$  trong bài toán ước lượng ở trang này. Vẽ đồ thị biểu diễn độ chính xác của tập huấn luyện và tập kiểm tra như một hàm số của  $\lambda$ . Bạn quan sát được điều gì?
2. Sử dụng tập kiểm định để tìm giá trị tối ưu của  $\lambda$ . Nó có thật sự là giá trị tối ưu hay không? Điều này có quan trọng lắm không?
3. Các phương trình cập nhật sẽ có dạng như thế nào nếu thay vì  $\|\mathbf{w}\|^2$ , chúng ta sử dụng lượng phạt  $\sum_i |w_i|$  (còn được gọi là điều chuẩn  $\ell_1$ ).
4. Chúng ta đã biết rằng  $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$ . Bạn có thể tìm một phương trình tương tự cho các ma trận (các nhà toán học gọi nó là [chuẩn Frobenius](#)<sup>109</sup>) hay không?
5. Ôn lại mối quan hệ giữa lỗi huấn luyện và lỗi khái quát. Bên cạnh việc sử dụng suy giảm trọng số, huấn luyện thêm và lựa chọn một mô hình có độ phức tạp phù hợp, bạn có thể nghĩ ra cách nào khác để giải quyết vấn đề quá khớp không?

<sup>108</sup> [https://en.wikipedia.org/wiki/Reproducing\\_kernel\\_Hilbert\\_space](https://en.wikipedia.org/wiki/Reproducing_kernel_Hilbert_space)

<sup>109</sup> [https://en.wikipedia.org/wiki/Matrix\\_norm#Frobenius\\_norm](https://en.wikipedia.org/wiki/Matrix_norm#Frobenius_norm)

6. Trong thống kê Bayesian chúng ta sử dụng tích của tiên nghiệm và hàm hợp lý để suy ra hậu nghiệm thông qua  $P(w | x) \propto P(x | w)P(w)$ . Làm thế nào để suy ra được hậu nghiệm  $P(w)$  khi sử dụng điều chuẩn?

### 6.5.7 Thảo luận

- Tiếng Anh<sup>110</sup>
- Tiếng Việt<sup>111</sup>

### 6.5.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm
- Vũ Hữu Tiệp
- Lý Phi Long
- Lê Khắc Hồng Phúc
- Nguyễn Duy Du
- Phạm Minh Đức
- Lê Cao Thăng
- Nguyễn Lê Quang Nhật

## 6.6 Dropout

Vừa xong ở Section 6.5, chúng tôi đã giới thiệu cách tiếp cận điển hình để điều chỉnh các mô hình thống kê bằng cách phạt giá trị chuẩn  $\ell_2$  của các trọng số. Theo ngôn ngữ xác suất, ta có thể giải thích kĩ thuật này bằng cách nói ta đã có một niềm tin từ trước rằng các trọng số được lấy ngẫu nhiên từ một phân phối Gauss với trung bình bằng 0. Hiểu một cách trực quan, ta có thể nói rằng mô hình được khuyến khích trải rộng giá trị các trọng số ra trên nhiều đặc trưng thay vì quá phụ thuộc vào một vài những liên kết có khả năng không chính xác.

<sup>110</sup> <https://discuss.mxnet.io/t/2342>

<sup>111</sup> <https://forum.machinelearningcoban.com/c/d21>

### 6.6.1 Bàn lại về Quá khớp

Khi có nhiều đặc trưng hơn số mẫu, các mô hình tuyến tính sẽ có xu hướng quá khớp. Tuy nhiên nếu có nhiều mẫu hơn số đặc trưng, nhìn chung ta có thể tin cậy mô hình tuyến tính sẽ không quá khớp. Thật không may, mô hình tuyến tính dựa trên tính ổn định này để khái quát hoá lại kèm theo một cái giá phải trả: Mô hình tuyến tính không màng tới sự tương tác giữa các đặc trưng, nếu chỉ được áp dụng một cách đơn giản. Mỗi đặc trưng sẽ được gán một giá trị trọng số hoặc là âm, hoặc là dương mà không màng tới ngữ cảnh.

Trong các tài liệu truyền thống, vấn đề cốt lõi giữa khả năng khái quát và tính linh hoạt này được gọi là *đánh đổi độ chêch - phương sai* (*bias-variance tradeoff*). Mô hình tuyến tính có độ chêch cao (vì nó chỉ có thể biểu diễn một nhóm nhỏ các hàm số) nhưng lại có phương sai thấp (vì nó cho kết quả khá tương đồng trên nhiều tập dữ liệu được lấy mẫu ngẫu nhiên).

Mạng nơ-ron sâu lại nằm ở thái cực trái ngược trên phổ độ chêch - phương sai. Khác với mô hình tuyến tính, các mạng nơ-ron không bị giới hạn ở việc chỉ được xét từng đặc trưng một cách riêng biệt. Chúng có thể học được sự tương tác giữa các nhóm đặc trưng. Chẳng hạn, chúng có thể suy ra được rằng nếu từ “Nigeria” và “Western Union” xuất hiện cùng nhau trong một email thì đó là thư rác, nhưng nếu hai từ đó xuất hiện riêng biệt thì lại không phải.

Ngay cả khi số mẫu nhiều hơn hẳn so với số đặc trưng, mạng nơ-ron sâu vẫn có thể quá khớp. Năm 2017, một nhóm các nhà nghiên cứu đã minh họa khả năng linh hoạt tệ hại của mạng nơ-ron bằng cách huấn luyện mạng nơ-ron sâu trên tập ảnh được gán nhãn ngẫu nhiên. Dù không hề có bất cứ một khuôn mẫu nào liên kết đầu vào và đầu ra, họ phát hiện rằng mạng nơ-ron được tối ưu bằng SGD vẫn có thể khớp tất cả nhãn trên tập huấn luyện một cách hoàn hảo.

Hãy cùng xem xét ý nghĩa của điều này. Nếu nhãn được gán ngẫu nhiên từ một phân phối đều với 10 lớp, sẽ không có bộ phân loại nào có thể có độ chính xác cao hơn 10% trên tập dữ liệu kiểm tra. Khoảng cách khái quát là tận 90%. Nếu mô hình của chúng ta có đủ năng lực để quá khớp tới như vậy, phải như thế nào chúng ta mới có thể trông đợi rằng mô hình sẽ không quá khớp? Nền tảng toán học đằng sau tính chất khái quát hóa búa của mạng nơ-ron sâu vẫn còn là một câu hỏi mở và chúng tôi khuyến khích bạn đọc chú trọng lý thuyết đào sâu hơn vào chủ đề này. Còn bây giờ, hãy quay về bề mặt của vấn đề và chuyển sang tìm hiểu các công cụ dựa trên thực nghiệm để cải thiện khả năng khái quát của các mạng nơ-ron sâu.

### 6.6.2 Khả năng Kháng Nhiễu

Hãy cùng nghĩ một chút về thứ mà ta mong đợi từ một mô hình dự đoán tốt. Ta muốn mô hình hoạt động tốt khi gặp dữ liệu mà nó chưa từng thấy. Lý thuyết khái quát cổ điển cho rằng: để thu hẹp khoảng cách giữa chất lượng khi huấn luyện và chất lượng khi kiểm tra, ta nên hướng tới một mô hình *đơn giản*. Sự đơn giản này có thể nằm ở việc đặc trưng có số chiều thấp, điều mà chúng ta đã nghiên cứu khi thảo luận về hàm cơ sở đơn thức trong mô hình tuyến tính ở [Section 6.4](#). Như ta đã thấy khi bàn về suy giảm trọng số (điều chuẩn  $\ell_2$ ) ở [Section 6.5](#), chuẩn (nghịch đảo) của các tham số là một phép đo khác cho sự đơn giản. Một khái niệm hữu ích khác để biểu diễn sự đơn giản là độ mượt, tức hàm số không nên quá nhạy với những thay đổi nhỏ ở đầu vào. Ví dụ, khi phân loại ảnh, ta mong muốn rằng việc thêm một chút nhiễu ngẫu nhiên vào các điểm ảnh sẽ không ảnh hưởng nhiều tới kết quả dự đoán.

Vào năm 1995, Christopher Bishop đã chính quy hóa ý tưởng này khi ông chứng minh rằng việc huấn luyện với đầu vào chứa nhiễu tương đương với điều chuẩn Tikhonov ([Bishop, 1995](#)). Công trình này đã chỉ rõ mối liên kết toán học giữa điều kiện hàm là mượt (nên nó cũng đơn giản) với khả năng kháng nhiễu đầu vào của hàm số.

Và rồi vào năm 2014, Srivastava et al. (Srivastava et al., 2014) đã phát triển một ý tưởng thông minh để áp dụng ý tưởng trên của Bishop cho các tầng *nội bộ* của mạng nơ-ron. Cụ thể, họ đề xuất việc thêm nhiễu vào mỗi tầng của mạng trước khi tính toán các tầng kế tiếp trong quá trình huấn luyện. Họ nhận ra rằng khi huấn luyện mạng đa tầng, thêm nhiễu vào dữ liệu chỉ ép buộc điều kiện mượt lên phép ánh xạ giữa đầu vào và đầu ra.

Ý tưởng này, có tên gọi là *dropout*, hoạt động bằng cách thêm nhiễu khi tính toán các tầng nội bộ trong lượt truyền xuôi và nó đã trở thành một kỹ thuật tiêu chuẩn để huấn luyện các mạng nơ-ron. Phương pháp này có tên gọi như vậy là bởi ta *loại bỏ* (drop out) một số nơ-ron trong quá trình huấn luyện. Tại mỗi vòng lặp huấn luyện, phương pháp dropout tiêu chuẩn sẽ đặt giá trị của một lượng nhất định (thường là 50%) các nút trong mỗi tầng về không, trước khi tính toán các tầng kế tiếp.

Để nói cho rõ, mối liên kết đến Bishop là của chúng tôi tự đặt ra. Đáng ngạc nhiên, bài báo gốc về dropout xây dựng cách hiểu trực giác bằng việc so sánh nó với quá trình sinh sản hữu tính. Các tác giả cho rằng hiện tượng quá khớp mạng nơ-ron là biểu hiện của việc mỗi tầng đều dựa vào một khuôn mẫu nhất định của các giá trị kích hoạt ở tầng trước đó, họ gọi trạng thái này là *đồng thích nghi*. Họ khẳng định rằng dropout phá bỏ sự đồng thích nghi này, tương tự như luận điểm sinh sản hữu tính phá bỏ các gen đã đồng thích nghi.

Thách thức chính bây giờ là *làm thế nào* để thêm nhiễu. Một cách để làm điều này là thêm nhiễu một cách *không thiên lệch* sao cho giá trị kỳ vọng của mỗi tầng bằng giá trị kỳ vọng của chính tầng đó trước khi được thêm nhiễu, giả sử rằng các tầng khác được giữ nguyên.

Trong nghiên cứu của Bishop, ông thêm nhiễu Gauss cho đầu vào của một mô hình tuyến tính như sau: Tại mỗi bước huấn luyện, ông đã thêm nhiễu lấy từ một phân phối có trung bình bằng không  $\epsilon \sim \mathcal{N}(0, \sigma^2)$  cho đầu vào  $\mathbf{x}$ , thu được một điểm bị nhiễu  $\mathbf{x}' = \mathbf{x} + \epsilon$  với kỳ vọng  $E[\mathbf{x}'] = \mathbf{x}$ .

Với điều chuẩn dropout tiêu chuẩn, ta khử độ chêch tại mỗi tầng bằng cách chuẩn hóa theo tỉ lệ các nút được giữ lại (chứ không phải các nút bị loại bỏ). Nói cách khác, dropout với xác suất *dropout*  $p$  được áp dụng như sau:

$$h' = \begin{cases} 0 & \text{với xác suất } p \\ \frac{h}{1-p} & \text{khác} \end{cases} \quad (6.6.1)$$

Như ta mong muốn, kỳ vọng không bị thay đổi, hay nói cách khác  $E[h'] = h$ . Đầu ra của các hàm kích hoạt trung gian  $h$  được thay thế bởi một biến ngẫu nhiên  $h'$  với kỳ vọng tương ứng.

### 6.6.3 Dropout trong Thực tiễn

Nhắc lại về mạng perceptron đa tầng (Section 6.1) với duy nhất một tầng ẩn có 5 nút ẩn. Kiến trúc mạng được biểu diễn như sau

$$\begin{aligned} \mathbf{h} &= \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1), \\ \mathbf{o} &= \mathbf{W}_2 \mathbf{h} + \mathbf{b}_2, \\ \hat{\mathbf{y}} &= \text{softmax}(\mathbf{o}). \end{aligned} \quad (6.6.2)$$

Khi chúng ta áp dụng dropout cho một tầng ẩn, tức gán mỗi nút ẩn bằng không với xác suất là  $p$ , kết quả có thể được xem như là một mạng chỉ chứa một tập con của các nơ-ron ban đầu. Trong Fig. 6.6.1,  $h_2$  và  $h_5$  bị loại bỏ. Hệ quả là, việc tính toán  $y$  không còn phụ thuộc vào  $h_2$  và  $h_5$  nữa và gradient tương ứng của chúng cũng biến mất khi thực hiện lan truyền ngược. Theo cách này, việc tính toán tầng đầu ra không thể quá phụ thuộc vào bất kỳ một thành phần nào trong  $h_1, \dots, h_5$ .

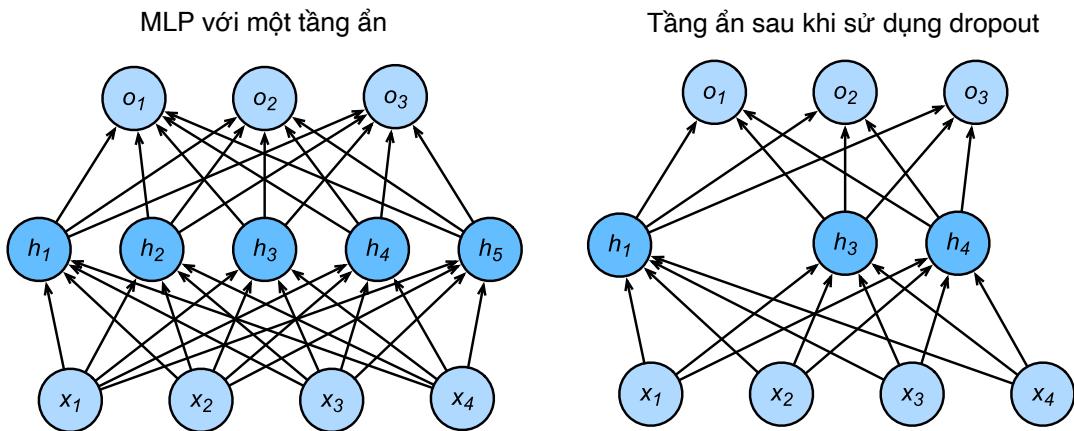


Fig. 6.6.1: MLP trước và sau khi dropout

Thông thường, **chúng ta sẽ vô hiệu hóa dropout tại thời điểm kiểm tra**. Với một mô hình đã huấn luyện và một mẫu kiểm tra, ta sẽ không thực hiện loại bỏ bất kỳ nút nào (do đó cũng không cần chuẩn hóa). Tuy nhiên cũng có một vài ngoại lệ. Một vài nhà nghiên cứu sử dụng dropout tại thời điểm kiểm tra như một thủ thuật để ước lượng *độ bất định* trong dự đoán của mạng nơ-ron: nếu các dự đoán giống nhau với nhiều mặt nạ dropout khác nhau, ta có thể nói rằng mạng đó đáng tin cậy hơn. Hiện tại, ta sẽ để dành phần ước lượng độ bất định này cho các chương sau.

#### 6.6.4 Lập trình từ đầu

Để lập trình hàm dropout cho một tầng đơn, ta sẽ lấy mẫu từ một biến ngẫu nhiên Bernoulli (nhị phân) với số lượng bằng với số chiều của tầng, trong đó biến ngẫu nhiên đạt giá trị 1 (giữ) với xác suất bằng  $1 - p$  và giá trị 0 (bỏ) với xác suất bằng  $p$ . Một cách đơn giản để thực hiện việc này là lấy mẫu từ một phân phối đều  $U[0, 1]$ , sau đó ta có thể giữ các nút có mẫu tương ứng lớn hơn  $p$  và bỏ đi những nút còn lại.

Trong đoạn mã nguồn bên dưới, ta lập trình hàm `dropout_layer` có chức năng bỏ đi các phần tử trong mảng ndarray đầu vào  $X$  với xác suất dropout, rồi chia các phần tử còn lại cho  $1.0$ -dropout như đã mô tả bên trên.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    # In this case, all elements are dropped out
    if dropout == 1:
        return np.zeros_like(X)
    # In this case, all elements are kept
    if dropout == 0:
        return X
    mask = np.random.uniform(0, 1, X.shape) > dropout
    return mask.astype(np.float32) * X / (1.0-dropout)
```

Ta có thể thử nghiệm hàm `dropout_layer` với một vài mẫu. Trong đoạn mã nguồn dưới đây, đầu vào  $X$  được truyền qua bước dropout với xác suất lần lượt là 0, 0.5 và 1.

```
X = np.arange(16).reshape(2, 8)
print(dropout_layer(X, 0))
print(dropout_layer(X, 0.5))
print(dropout_layer(X, 1))
```

## Định nghĩa các Tham số Mô hình

Một lần nữa, ta sẽ làm việc với bộ dữ liệu Fashion-MNIST được giới thiệu ở Section 5.6. Ta sẽ tạo một perception đa tầng với hai tầng ẩn, mỗi tầng gồm 256 đầu ra.

```
num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

W1 = np.random.normal(scale=0.01, size=(num_inputs, num_hiddens1))
b1 = np.zeros(num_hiddens1)
W2 = np.random.normal(scale=0.01, size=(num_hiddens1, num_hiddens2))
b2 = np.zeros(num_hiddens2)
W3 = np.random.normal(scale=0.01, size=(num_hiddens2, num_outputs))
b3 = np.zeros(num_outputs)

params = [W1, b1, W2, b2, W3, b3]
for param in params:
    param.attach_grad()
```

## Định nghĩa Mô hình

Mô hình dưới đây áp dụng dropout cho đầu ra của mỗi tầng ẩn (theo sau hàm kích hoạt). Ta có thể đặt các giá trị xác suất dropout riêng biệt cho mỗi tầng. Một xu hướng chung là đặt xác suất dropout thấp hơn cho tầng ở gần với tầng đầu vào hơn. Bên dưới ta đặt xác suất dropout bằng 0.2 và 0.5 tương ứng cho tầng ẩn thứ nhất và thứ hai. Bằng cách sử dụng hàm `is_training` mô tả ở Section 4.5, ta có thể chắc chắn rằng dropout chỉ được kích hoạt trong quá trình huấn luyện.

```
dropout1, dropout2 = 0.2, 0.5

def net(X):
    X = X.reshape(-1, num_inputs)
    H1 = npx.relu(np.dot(X, W1) + b1)
    # Use dropout only when training the model
    if autograd.is_training():
        # Add a dropout layer after the first fully connected layer
        H1 = dropout_layer(H1, dropout1)
    H2 = npx.relu(np.dot(H1, W2) + b2)
    if autograd.is_training():
        # Add a dropout layer after the second fully connected layer
        H2 = dropout_layer(H2, dropout2)
    return np.dot(H2, W3) + b3
```

## Huấn luyện và Kiểm tra

Việc này tương tự với quá trình huấn luyện và kiểm tra của các perceptron đa tầng trước đây.

```
num_epochs, lr, batch_size = 10, 0.5, 256
loss = gluon.loss.SoftmaxCrossEntropyLoss()
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs,
    lambda batch_size: d2l.sgd(params, lr, batch_size))
```

### 6.6.5 Cách lập trình súc tích

Bằng việc sử dụng Gluon, tất cả những gì ta cần làm là thêm một tầng Dropout (cũng nằm trong gói nn) vào sau mỗi tầng kết nối đầy đủ và truyền vào xác suất dropout, đối số duy nhất của hàm khởi tạo. Trong quá trình huấn luyện, hàm Dropout sẽ bỏ ngẫu nhiên một số đầu ra của tầng trước (hay tương đương với đầu vào của tầng tiếp theo) dựa trên xác suất dropout được định nghĩa trước đó.

```
net = nn.Sequential()
net.add(nn.Dense(256, activation="relu"),
    # Add a dropout layer after the first fully connected layer
    nn.Dropout(dropout1),
    nn.Dense(256, activation="relu"),
    # Add a dropout layer after the second fully connected layer
    nn.Dropout(dropout2),
    nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

Tiếp theo, ta huấn luyện và kiểm tra mô hình.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, trainer)
```

### 6.6.6 Tóm tắt

- Ngoài phương pháp kiểm soát số chiều và độ lớn của vector trọng số, dropout cũng là một công cụ khác để tránh tình trạng quá khớp. Thông thường thì cả ba cách được sử dụng cùng nhau.
- Dropout thay thế giá trị kích hoạt  $h$  bằng một biến ngẫu nhiên  $h'$  với giá trị kỳ vọng  $h$  và phương sai bằng xác suất dropout  $p$ .
- Dropout chỉ được sử dụng trong quá trình huấn luyện.

### 6.6.7 Bài tập

- Điều gì xảy ra nếu bạn thay đổi xác suất dropout của tầng 1 và 2? Cụ thể, điều gì xảy ra nếu bạn tráo đổi xác suất của hai tầng này? Thiết kế một thí nghiệm để trả lời những câu hỏi này, mô tả các kết quả một cách định lượng và tóm tắt các bài học rút ra một cách định tính.
- Tăng số lượng epoch và so sánh các kết quả thu được khi sử dụng và khi không sử dụng dropout.
- Tính toán phương sai của các giá trị kích hoạt ở mỗi tầng ẩn khi sử dụng và khi không sử dụng dropout. Vẽ biểu đồ thể hiện sự thay đổi của giá trị phương sai này theo thời gian cho cả hai mô hình.
- Tại sao dropout thường không được sử dụng tại bước kiểm tra?
- Sử dụng mô hình trong phần này làm ví dụ, so sánh hiệu quả của việc sử dụng dropout và suy giảm trọng số. Điều gì xảy ra khi dropout và suy giảm trọng số được sử dụng cùng một lúc? Hai phương pháp này bổ trợ cho nhau, làm giảm hiệu quả của nhau hay (tệ hơn) loại trừ lẫn nhau?
- Điều gì xảy ra nếu chúng ta áp dụng dropout cho các trọng số riêng lẻ của ma trận trọng số thay vì các giá trị kích hoạt?
- Hãy phát minh một kỹ thuật khác với kỹ thuật dropout tiêu chuẩn để thêm nhiễu ngẫu nhiên ở mỗi tầng. Bạn có thể phát triển một phương pháp cho kết quả tốt hơn dropout trên bộ dữ liệu FashionMNIST không (với cùng một kiến trúc)?

### 6.6.8 Thảo luận

- Tiếng Anh<sup>112</sup>
- Tiếng Việt<sup>113</sup>

### 6.6.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Văn Tâm
- Phạm Hồng Vinh
- Nguyễn Duy Du
- Vũ Hữu Tiệp

<sup>112</sup> <https://discuss.mxnet.io/t/2343>

<sup>113</sup> <https://forum.machinelearningcoban.com/c/d21>

## 6.7 Lan truyền xuôi, Lan truyền ngược và Đồ thị tính toán

Cho đến lúc này, ta đã huấn luyện các mô hình với giải thuật hạ gradient ngẫu nhiên theo mini-batch. Tuy nhiên, khi lập trình thuật toán, ta mới chỉ bận tâm đến các phép tính trong quá trình *lan truyền xuôi* qua mô hình. Khi cần tính gradient, ta mới chỉ đơn giản gọi hàm backward và mô-đun autograd sẽ lo các chi tiết tính toán.

Việc tính toán gradient tự động sẽ giúp công việc lập trình các thuật toán học sâu được đơn giản hóa đi rất nhiều. Trước đây, khi chưa có công cụ tính vi phân tự động, ngay cả khi ta chỉ thay đổi một chút các mô hình phức tạp, các đạo hàm rắc rối cũng cần phải được tính lại một cách thủ công. Điều đáng ngạc nhiên là các bài báo học thuật thường có các công thức cập nhật mô hình dài hàng trang giấy. Vậy nên dù vẫn phải tiếp tục dựa vào autograd để có thể tập trung vào những phần thú vị của học sâu, bạn vẫn nên *nắm rõ* thay vì chỉ hiểu một cách hời hợt cách tính gradient nếu bạn muốn tiến xa hơn.

Trong mục này, ta sẽ đi sâu vào chi tiết của lan truyền ngược (thường được gọi là *backpropagation* hoặc *backprop*). Ta sẽ sử dụng một vài công thức toán học cơ bản và đồ thị tính toán để giải thích một cách chi tiết cách thức hoạt động cũng như cách lập trình các kỹ thuật này. Và để bắt đầu, ta sẽ tập trung giải trình một perceptron đa tầng gồm ba tầng (một tầng ẩn) đi kèm với suy giảm trọng số (điều chuẩn  $\ell_2$ ).

### 6.7.1 Lan truyền Xuôi

Lan truyền xuôi là quá trình tính toán cũng như lưu trữ các biến trung gian (bao gồm cả đầu ra) của mạng nơ-ron theo thứ tự từ tầng đầu vào đến tầng đầu ra. Bây giờ ta sẽ thực hiện qua từng bước trong cơ chế vận hành của mạng nơ-ron sâu có một tầng ẩn. Điều này nghe có vẻ tẻ nhạt nhưng theo như cách nói dân giã, bạn phải “tập đi trước khi tập chạy”.

Để đơn giản hóa vấn đề, ta giả sử mẫu đầu vào là  $\mathbf{x} \in \mathbb{R}^d$  và tầng ẩn của ta không có hệ số điều chỉnh. Ở đây biến trung gian là:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}, \quad (6.7.1)$$

trong đó  $\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  là tham số trọng số của tầng ẩn. Sau khi đưa biến trung gian  $\mathbf{z} \in \mathbb{R}^h$  qua hàm kích hoạt  $\phi$ , ta thu được vector kích hoạt ẩn với  $h$  phần tử,

$$\mathbf{h} = \phi(\mathbf{z}). \quad (6.7.2)$$

Biến ẩn  $\mathbf{h}$  cũng là một biến trung gian. Giả sử tham số của tầng đầu ra chỉ gồm trọng số  $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ , ta sẽ thu được một vector với  $q$  phần tử ở tầng đầu ra:

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}. \quad (6.7.3)$$

Giả sử hàm mất mát là  $l$  và nhãn của mẫu là  $y$ , ta có thể tính được lượng mất mát cho một mẫu dữ liệu duy nhất,

$$L = l(\mathbf{o}, y). \quad (6.7.4)$$

Theo định nghĩa của điều chuẩn  $\ell_2$  với siêu tham số  $\lambda$ , lượng điều chuẩn là:

$$s = \frac{\lambda}{2} \left( \|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right), \quad (6.7.5)$$

trong đó chuẩn Frobenius của ma trận chỉ đơn giản là chuẩn  $L_2$  của vector thu được sau khi trải phẳng ma trận. Cuối cùng, hàm mất mát được điều chỉnh của mô hình trên một mẫu dữ liệu cho trước là:

$$J = L + s. \quad (6.7.6)$$

Ta sẽ bàn thêm về *hàm mục tiêu*  $J$  ở phía dưới.

### 6.7.2 Đồ thị Tính toán của Lan truyền Xuôi

Vẽ đồ thị tính toán giúp chúng ta hình dung được sự phụ thuộc giữa các toán tử và các biến trong quá trình tính toán. Fig. 6.7.1 thể hiện đồ thị tương ứng với mạng nơ-ron đã miêu tả ở trên. Góc trái dưới biểu diễn đầu vào trong khi góc phải trên biểu diễn đầu ra. Lưu ý rằng hướng của các mũi tên (thể hiện luồng dữ liệu) chủ yếu là đi qua phải và hướng lên trên.

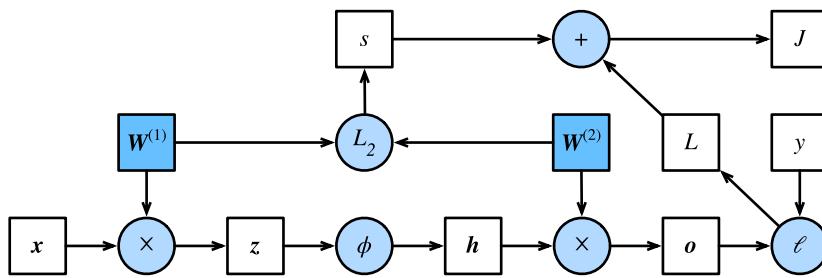


Fig. 6.7.1: Đồ thị tính toán

### 6.7.3 Lan truyền Ngược

Lan truyền ngược là phương pháp tính gradient của các tham số mạng nơ-ron. Nói một cách đơn giản, phương thức này duyệt qua mạng nơ-ron theo chiều ngược lại, từ đầu ra đến đầu vào, tuân theo quy tắc dây chuyền trong giải tích.

Thuật toán lan truyền ngược lưu trữ các biến trung gian (là các đạo hàm riêng) cần thiết trong quá trình tính toán gradient theo các tham số. Giả sử chúng ta có hàm  $Y = f(X)$  và  $Z = g(Y) = g \circ f(X)$ , trong đó đầu vào và đầu ra  $X, Y, Z$  là các tensor có kích thước bất kỳ. Bằng cách sử dụng quy tắc dây chuyền, chúng ta có thể tính đạo hàm của  $Z$  theo  $X$  như sau:

$$\frac{\partial Z}{\partial X} = \text{prod} \left( \frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X} \right). \quad (6.7.7)$$

Ở đây, chúng ta sử dụng toán tử prod để nhân các đối số sau khi các phép tính cần thiết như là chuyển vị và hoán đổi đã được thực hiện. Với vector, điều này khá đơn giản: nó chỉ đơn thuần là phép nhân ma trận. Với các tensor nhiều chiều thì sẽ có các phương án tương ứng phù hợp. Toán tử prod sẽ đơn giản hóa việc ký hiệu.

Các tham số của mạng nơ-ron đơn giản với một tầng ẩn là  $\mathbf{W}^{(1)}$  và  $\mathbf{W}^{(2)}$ . Mục đích của lan truyền ngược là để tính gradient  $\partial J / \partial \mathbf{W}^{(1)}$  và  $\partial J / \partial \mathbf{W}^{(2)}$ . Để làm được điều này, ta áp dụng quy tắc dây chuyền và lần lượt tính gradient của các biến trung gian và tham số. Các phép tính trong lan truyền ngược có thứ tự ngược lại so với các phép tính trong lan truyền xuôi, bởi ta muốn bắt đầu từ kết

quả của đồ thị tính toán rồi dần đi tới các tham số. Bước đầu tiên đó là tính gradient của hàm mục tiêu  $J = L + s$  theo mất mát  $L$  và điều chuẩn  $s$ .

$$\frac{\partial J}{\partial L} = 1 \text{ và } \frac{\partial J}{\partial s} = 1. \quad (6.7.8)$$

Tiếp theo, ta tính gradient của hàm mục tiêu theo các biến của lớp đầu ra  $\mathbf{o}$ , sử dụng quy tắc dây chuyền.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod} \left( \frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}} \right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q. \quad (6.7.9)$$

Kế tiếp, ta tính gradient của điều chuẩn theo cả hai tham số.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ và } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}. \quad (6.7.10)$$

Bây giờ chúng ta có thể tính gradient  $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$  của các tham số mô hình gần nhất với tầng đầu ra. Áp dụng quy tắc dây chuyền, ta có:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}} \right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}. \quad (6.7.11)$$

Để tính được gradient theo  $\mathbf{W}^{(1)}$  ta cần tiếp tục lan truyền ngược từ tầng đầu ra đến các tầng ẩn. Gradient theo các đầu ra của tầng ẩn  $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$  được tính như sau:

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}} \right) = \mathbf{W}^{(2)\top} \frac{\partial J}{\partial \mathbf{o}}. \quad (6.7.12)$$

Vì hàm kích hoạt  $\phi$  áp dụng cho từng phần tử, việc tính gradient  $\partial J / \partial \mathbf{z} \in \mathbb{R}^h$  của biến trung gian  $\mathbf{z}$  cũng yêu cầu sử dụng phép nhân theo từng phần tử, kí hiệu bởi  $\odot$ .

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}} \right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'(\mathbf{z}). \quad (6.7.13)$$

Cuối cùng, ta có thể tính gradient  $\partial J / \partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$  của các tham số mô hình gần nhất với tầng đầu vào. Theo quy tắc dây chuyền, ta có

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod} \left( \frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}} \right) + \text{prod} \left( \frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}} \right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}. \quad (6.7.14)$$

#### 6.7.4 Huấn luyện một Mô hình

Khi huấn luyện các mạng nơ-ron, lan truyền xuôi và lan truyền ngược phụ thuộc lẫn nhau. Cụ thể với lan truyền xuôi, ta duyệt đồ thị tính toán theo hướng của các quan hệ phụ thuộc và tính tất cả các biến trên đường đi. Những biến này sau đó được sử dụng trong lan truyền ngược khi thứ tự tính toán trên đồ thị bị đảo ngược lại. Hệ quả là ta cần lưu trữ các giá trị trung gian cho đến khi lan truyền ngược hoàn tất. Đây cũng chính là một trong những lý do khiến lan truyền ngược yêu cầu nhiều bộ nhớ hơn đáng kể so với khi chỉ cần đưa ra dự đoán.

Ta tính các tensor gradient và giữ các biến trung gian lại để sử dụng trong quy tắc dây chuyền. Việc huấn luyện trên các minibatch chứa nhiều mẫu, do đó cần lưu trữ nhiều giá trị kích hoạt trung gian hơn cũng là một lý do khác.

### 6.7.5 Tóm tắt

- Lan truyền xuôi lần lượt tính và lưu trữ các biến trung gian từ tầng đầu vào đến tầng đầu ra trong đồ thị tính toán được định nghĩa bởi mạng nơ-ron.
- Lan truyền ngược lần lượt tính và lưu trữ các gradient của biến trung gian và tham số mạng nơ-ron theo chiều ngược lại.
- Khi huấn luyện các mô hình học sâu, lan truyền xuôi và lan truyền ngược phụ thuộc lẫn nhau.
- Việc huấn luyện cần nhiều bộ nhớ lưu trữ hơn đáng kể so với việc dự đoán.

### 6.7.6 Bài tập

1. Giả sử đầu vào  $\mathbf{x}$  của hàm số vô hướng  $f$  là ma trận  $n \times m$ . Gradient của  $f$  theo  $\mathbf{x}$  có chiều là bao nhiêu?
2. Thêm một hệ số điều chỉnh vào tầng ẩn của mô hình được mô tả ở trên.
  - Vẽ đồ thị tính toán tương ứng.
  - Tìm các phương trình cho quá trình lan truyền xuôi và lan truyền ngược.
3. Tính lượng bộ nhớ mà mô hình được mô tả ở chương này sử dụng lúc huấn luyện và lúc dự đoán.
4. Giả sử bạn muốn tính đạo hàm bậc hai. Điều gì sẽ xảy ra với đồ thị tính toán? Hãy ước tính thời gian hoàn thành quá trình này?
5. Giả sử rằng đồ thị tính toán trên là quá sức với GPU của bạn.
  - Bạn có thể phân vùng nó trên nhiều GPU không?
  - Ưu điểm và nhược điểm của việc huấn luyện với một minibatch nhỏ hơn là gì?

### 6.7.7 Thảo luận

- Tiếng Anh<sup>114</sup>
- Tiếng Việt<sup>115</sup>

### 6.7.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Duy Du
- Lý Phi Long
- Lê Khắc Hồng Phúc
- Phạm Minh Đức

<sup>114</sup> <https://discuss.mxnet.io/t/2344>

<sup>115</sup> <https://forum.machinelearningcoban.com/c/d21>

- Nguyễn Lê Quang Nhật
- Phạm Ngọc Bảo Anh

## 6.8 Ôn định Số học và Khởi tạo

Cho đến nay, đối với mọi mô hình mà ta đã lập trình, ta đều phải khởi tạo các tham số theo một phân phối cụ thể nào đó. Tuy nhiên, ta mới chỉ lướt qua các chi tiết thực hiện mà không để tâm lắm tới việc tại sao lại khởi tạo tham số như vậy. Bạn thậm chí có thể nghĩ rằng các lựa chọn này không đặc biệt quan trọng. Tuy nhiên, việc lựa chọn cơ chế khởi tạo đóng vai trò rất lớn trong quá trình học của mạng nơ-ron và có thể là yếu tố quyết định để duy trì sự ổn định số học. Hơn nữa, các phương pháp khởi tạo cũng có thể bị ràng buộc bởi các hàm kích hoạt phi tuyến theo những cách thú vị. Việc lựa chọn hàm kích hoạt và cách khởi tạo tham số có thể ảnh hưởng tới tốc độ hội tụ của thuật toán tối ưu. Nếu ta lựa chọn không hợp lý, việc bùng nổ hoặc tiêu biến gradient có thể sẽ xảy ra. Trong phần này, ta sẽ đi sâu hơn vào các chi tiết của chủ đề trên và thảo luận một số phương pháp thực nghiệm hữu ích mà bạn có thể sẽ sử dụng thường xuyên trong suốt sự nghiệp học sâu.

### 6.8.1 Tiêu biến và Bùng nổ Gradient

Xét một mạng nơ-ron sâu với  $L$  tầng, đầu vào  $\mathbf{x}$  và đầu ra  $\mathbf{o}$ . Mỗi tầng  $l$  được định nghĩa bởi một phép biến đổi  $f_l$  với tham số là trọng số  $\mathbf{W}_l$ . Mạng nơ-ron này có thể được biểu diễn như sau:

$$\mathbf{h}^{l+1} = f_l(\mathbf{h}^l) \text{ và vì vậy } \mathbf{o} = f_L \circ \dots \circ f_1(\mathbf{x}). \quad (6.8.1)$$

Nếu tất cả giá trị kích hoạt và đầu vào là vector, ta có thể viết lại gradient của  $\mathbf{o}$  theo một tập tham số  $\mathbf{W}_l$  bất kỳ như sau:

$$\partial_{\mathbf{W}_l} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{L-1}} \mathbf{h}^L}_{:= \mathbf{M}_L} \cdot \dots \cdot \underbrace{\partial_{\mathbf{h}^l} \mathbf{h}^{l+1}}_{:= \mathbf{M}_l} \underbrace{\partial_{\mathbf{W}_l} \mathbf{h}^l}_{:= \mathbf{v}_l}. \quad (6.8.2)$$

Nói cách khác, gradient này là tích của  $L - l$  ma trận  $\mathbf{M}_L \cdot \dots \cdot \mathbf{M}_l$  với vector gradient  $\mathbf{v}_l$ . Vì vậy ta sẽ dễ gặp phải vấn đề tràn số dưới, một hiện tượng thường xảy ra khi nhân quá nhiều giá trị xác suất lại với nhau. Khi làm việc với các xác suất, một mánh phổ biến là chuyển về làm việc với giá trị log của nó. Nếu nhìn từ góc độ biểu diễn số học, điều này đồng nghĩa với việc chuyển trọng tâm biểu diễn của các bit từ phần định trị (*mantissa*) sang phần mũ (*exponent*). Thật không may, bài toán trên lại nghiêm trọng hơn nhiều: các ma trận  $M_l$  ban đầu có thể có nhiều trị riêng với độ lớn rất khác nhau. Các trị riêng có thể nhỏ hoặc lớn và do đó tích của chúng có thể *rất lớn* hoặc *rất nhỏ*. Rủi ro của việc gradient bất ổn không chỉ dừng lại ở vấn đề biểu diễn số học. Nếu ta không kiểm soát được độ lớn của gradient, sự ổn định của các thuật toán tối ưu cũng không được đảm bảo. Lúc đó ta sẽ quan sát được các bước cập nhật (i) quá lớn và phá hỏng mô hình (vấn đề *bùng nổ gradient*); hoặc (ii) quá nhỏ (vấn đề *tiêu biến gradient*), khiến việc học trở nên bất khả thi, khi mà các tham số hầu như không thay đổi ở mỗi bước cập nhật.

## Tiêu biến Gradient

Thông thường, thủ phạm gây ra vấn đề tiêu biến gradient này là hàm kích hoạt  $\sigma$  được chọn để đặt nối tiếp phép toán tuyến tính tại mỗi tầng. Trước đây, hàm kích hoạt sigmoid ( $1 + \exp(-x)$ ) (đã giới thiệu trong Section 6.1) là lựa chọn phổ biến bởi nó hoạt động giống với một hàm lấy nguồn. Bởi các mạng nơ-ron nhân tạo thời kỳ đầu lấy cảm hứng từ mạng nơ-ron sinh học, ý tưởng rằng các nơ-ron được kích hoạt hoàn toàn hoặc không hề kích hoạt (giống như nơ-ron sinh học) có vẻ rất hấp dẫn. Hãy cùng xem xét hàm sigmoid kỹ lưỡng hơn để thấy tại sao nó có thể gây ra vấn đề tiêu biến gradient.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()

x = np.arange(-8.0, 8.0, 0.1)
x.attach_grad()
with autograd.record():
    y = npx.sigmoid(x)
y.backward()

d2l.plot(x, [y, x.grad], legend=['sigmoid', 'gradient'], figsize=(4.5, 2.5))
```

Như ta có thể thấy, gradient của hàm sigmoid tiêu biến khi đầu vào của nó quá lớn hoặc quá nhỏ. Hơn nữa, khi thực hiện lan truyền ngược qua nhiều tầng, trừ khi giá trị nằm trong vùng Goldilocks, tại đó đầu vào của hầu hết các hàm sigmoid có giá trị xấp xỉ không, gradient của cả phép nhân có thể bị tiêu biến. Khi mạng nơ-ron có nhiều tầng, trừ khi ta cẩn trọng, nhiều khả năng luồng gradient sẽ bị ngắt tại một tầng nào đó. Vấn đề này đã từng gây nhiều khó khăn cho quá trình huấn luyện mạng nơ-ron sâu. Do đó, ReLU, một hàm số ổn định hơn (nhưng lại không hợp lý lắm từ khía cạnh khoa học thần kinh) đã và đang dần trở thành lựa chọn mặc định của những người làm học sâu.

## Bùng nổ Gradient

Một vấn đề đối lập là bùng nổ gradient cũng có thể gây phiền toái không kém. Để giải thích việc này rõ hơn, chúng ta lấy 100 ma trận ngẫu nhiên Gauss và nhân chúng với một ma trận ban đầu nào đó. Với khoảng giá trị mà ta đã chọn (phương sai  $\sigma^2 = 1$ ), tích các ma trận bị bùng nổ số học. Khi khởi tạo các mạng nơ-ron sâu một cách không hợp lý, các bộ tối ưu dựa trên hạ gradient sẽ không thể hội tụ được.

```
M = np.random.normal(size=(4, 4))
print('A single matrix', M)
for i in range(100):
    M = np.dot(M, np.random.normal(size=(4, 4)))

print('After multiplying 100 matrices', M)
```

## Tính Đối xứng

Một vấn đề khác trong việc thiết kế mạng nơ-ron sâu là tính đối xứng hiện hữu trong quá trình tham số hóa. Giả sử ta có một mạng nơ-ron sâu với một tầng ẩn gồm hai nút  $h_1$  và  $h_2$ . Trong trường hợp này, ta có thể hoán vị trọng số  $\mathbf{W}_1$  của tầng đầu tiên, rồi làm điều tương tự với các trọng số của tầng đầu ra để thu được một hàm giống hệt ban đầu. Ta có thể thấy rằng không có sự khác biệt nào giữa nút ẩn đầu tiên với nút ẩn thứ hai. Nói cách khác, ta có tính đối xứng hoán vị giữa các nút ẩn của từng tầng.

Đây không chỉ là phiền toái về mặt lý thuyết. Thử hình dung xem điều gì sẽ xảy ra nếu ta khởi tạo giá trị của mọi tham số ở các tầng như sau:  $\mathbf{W}_l = c$  với hằng số  $c$  nào đó. Trong trường hợp này thì các gradient cho tất cả các chiều là giống hệt nhau, nên mỗi nút không chỉ có cùng giá trị mà chúng còn có bước cập nhật giống nhau. Bản thân phương pháp hạ gradient ngẫu nhiên không thể phá vỡ tính đối xứng này và ta sẽ không hiện thực hóa được sức mạnh biểu diễn của mạng. Tầng ẩn sẽ hoạt động như thể nó chỉ có một nút duy nhất. Nhưng hãy lưu ý rằng dù hạ gradient ngẫu nhiên không thể phá vỡ được tính đối xứng, kỹ thuật điều chỉnh dropout lại hoàn toàn có thể!

### 6.8.2 Khởi tạo Tham số

Một cách giải quyết, hay ít nhất giảm thiểu các vấn đề được nêu ở trên là khởi tạo tham số một cách cẩn thận. Chỉ cần cẩn trọng một chút trong quá trình tối ưu hóa và điều chuẩn mô hình phù hợp, ta có thể cải thiện tính ổn định của quá trình học.

#### Khởi tạo Mặc định

Trong các phần trước, ví dụ như trong Section 5.3, ta đã sử dụng `net.initialize(init.Normal(sigma=0.01))` để khởi tạo các giá trị cho trọng số. Nếu ta không chỉ định sẵn một phương thức khởi tạo như `net.initialize()`, MXNet sẽ sử dụng phương thức khởi tạo ngẫu nhiên mặc định: các trọng số được lấy mẫu ngẫu nhiên từ phân phối đều  $U[-0.07, 0.07]$ , còn các hệ số điều chỉnh đều được đưa về giá trị 0. Cả hai lựa chọn đều hoạt động tốt với các bài toán cỡ trung trong thực tiễn.

#### Khởi tạo Xavier

Hãy cùng nhìn vào phân phối khoảng giá trị kích hoạt của các nút ẩn  $h_i$  ở một tầng nào đó:

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j. \quad (6.8.3)$$

Các trọng số  $W_{ij}$  đều được lấy mẫu độc lập từ cùng một phân phối. Hơn nữa, ta giả sử rằng phân phối này có trung bình bằng không và phương sai  $\sigma^2$  (đây không bắt buộc phải là phân phối Gauss, chỉ là ta cần phải cho trước trung bình và phương sai). Tạm thời hãy giả sử rằng đầu vào của tầng  $x_j$  cũng có trung bình bằng không và phương sai  $\gamma^2$ , độc lập với  $\mathbf{W}$ . Trong trường hợp này, ta có

thể tính được trung bình và phương sai của  $h_i$  như sau:

$$\begin{aligned}
 E[h_i] &= \sum_{j=1}^{n_{\text{in}}} E[W_{ij}x_j] = 0, \\
 E[h_i^2] &= \sum_{j=1}^{n_{\text{in}}} E[W_{ij}^2x_j^2] \\
 &= \sum_{j=1}^{n_{\text{in}}} E[W_{ij}^2]E[x_j^2] \\
 &= n_{\text{in}}\sigma^2\gamma^2.
 \end{aligned} \tag{6.8.4}$$

Một cách để giữ phương sai cố định là đặt  $n_{\text{in}}\sigma^2 = 1$ . Bây giờ hãy xem xét lan truyền ngược. Ở đó ta phải đổi mặt với vấn đề tương tự, mặc dù gradient được truyền từ các tầng trên cùng. Tức thay vì  $\mathbf{W}\mathbf{w}$ , ta cần đổi phó với  $\mathbf{W}^\top \mathbf{g}$ , trong đó  $\mathbf{g}$  là gradient đến từ lớp phía trên. Sử dụng lý luận tương tự với lan truyền xuôi, ta có thể thấy phương sai của các gradient sẽ bùng nổ trừ khi  $n_{\text{out}}\sigma^2 = 1$ . Điều này khiến ta rơi vào một tình huống khó xử: ta không thể thỏa mãn cả hai điều kiện cùng một lúc. Thay vào đó, ta cố thỏa mãn điều kiện sau:

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ hoặc tương đương } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}. \tag{6.8.5}$$

Đây là lý luận đằng sau phương thức khởi tạo Xavier, được đặt tên theo người đã tạo ra nó (Glorot & Bengio, 2010). Bây giờ nó đã trở thành phương thức tiêu chuẩn và rất hữu dụng trong thực tiễn. Thông thường, phương thức này lấy mẫu cho trọng số từ phân phối Gauss với trung bình bằng không và phương sai  $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$ . Ta cũng có thể tận dụng cách hiểu trực quan của Xavier để chọn phương sai khi lấy mẫu từ một phân phối đều. Chú ý rằng phân phối  $U[-a, a]$  có phương sai là  $a^2/3$ . Thay  $\sigma^2$  bằng  $a^2/3$  vào điều kiện trên, ta biết được rằng ta nên khởi tạo theo phân phối đều  $U\left[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}\right]$ .

## Sâu xa hơn nữa

Các lập luận đưa ra ở trên mới chỉ chạm tới bề mặt của những kỹ thuật khởi tạo tham số hiện đại. Trên thực tế, MXNet có nguyên một mô-đun `mxnet.initializer` <https://mxnet.apache.org/api/python/docs/api/initializer/index.html> với hàng chục các phương pháp khởi tạo dựa theo thực nghiệm khác nhau đã được lập trình sẵn. Hơn nữa, các phương pháp khởi tạo vẫn đang là một chủ đề nghiên cứu cẩn bản rất được quan tâm trong học sâu. Trong số đó là những phương pháp dựa trên thực nghiệm dành riêng cho trường hợp tham số bị trói buộc (được chia sẻ), cho bài toán siêu phân giải, mô hình chuỗi và nhiều trường hợp khác. Nếu có hứng thú, chúng tôi khuyên bạn nên đào sâu hơn vào mô-đun này, đọc các bài báo mà có đề xuất và phân tích các phương pháp thực nghiệm, và rồi tự khám phá các bài báo mới nhất về chủ đề này. Có lẽ bạn sẽ gặp (hay thậm chí phát minh ra) một ý tưởng thông minh và lập trình nó để đóng góp cho MXNet.

### 6.8.3 Tóm tắt

- Tiêu biến hay bùng nổ gradient đều là những vấn đề phổ biến trong những mạng nơ-ron sâu. Việc khởi tạo tham số cẩn thận là rất cần thiết để đảm bảo gradient và các tham số được kiểm soát tốt.
- Các kỹ thuật khởi tạo tham số dựa trên thực nghiệm là cần thiết để đảm bảo rằng gradient ban đầu không quá lớn hay quá nhỏ.
- Hàm kích hoạt ReLU giải quyết được vấn đề tiêu biến gradient. Điều này có thể làm tăng tốc độ hội tụ.
- Khởi tạo ngẫu nhiên là chìa khóa để đảm bảo tính đối xứng bị phá vỡ trước khi tối ưu hóa.

### 6.8.4 Bài tập

1. Ngoài tính đối xứng hoán vị giữa các tầng, bạn có thể nghĩ ra các trường hợp mà mạng nơ-ron thể hiện tính đối xứng khác cần được phá vỡ không?
2. Ta có thể khởi tạo tất cả trọng số trong hồi quy tuyến tính hoặc trong hồi quy softmax với cùng một giá trị hay không?
3. Hãy tra cứu cận chính xác của trị riêng cho tích hai ma trận. Nó cho ta biết gì về việc đảm bảo rằng gradient hợp lý?
4. Nếu biết rằng một vài số hạng sẽ phân kỳ, bạn có thể khắc phục vấn đề này không? Bạn có thể tìm cảm hứng từ bài báo LARS ([You et al., 2017](#)).

### 6.8.5 Thảo luận

- [Tiếng Anh](#)<sup>116</sup>
- [Tiếng Việt](#)<sup>117</sup>

### 6.8.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lý Phi Long
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Văn Tâm
- Trần Yến Thy
- Bùi Chí Minh
- Phạm Hồng Vinh

<sup>116</sup> <https://discuss.mxnet.io/t/2345>

<sup>117</sup> <https://forum.machinelearningcoban.com/c/d21>

## 6.9 Cân nhắc tới Môi trường

Trong các chương trước ta đã thực hành một số ứng dụng của học máy và khớp mô hình với nhiều tập dữ liệu khác nhau. Tuy nhiên, ta chưa bao giờ dừng lại để nhìn nhận về nguồn gốc của dữ liệu, hoặc dự định sẽ làm gì với đầu ra của mô hình. Đa phần là khi có được dữ liệu, các nhà phát triển học máy thường đâm đầu vào triển khai các mô hình mà không tạm dừng để xem xét các vấn đề cơ bản này.

Nhiều triển khai học máy thất bại có thể bắt nguồn từ khuôn mẫu này. Đôi khi các mô hình có độ chính xác rất tốt trên tập kiểm tra nhưng lại thất bại thảm hại trong triển khai thực tế, khi mà phân phối của dữ liệu thay đổi đột ngột. Đáng sợ hơn, đôi khi chính việc triển khai một mô hình có thể là chất xúc tác gây nhiễu cho phân phối dữ liệu. Ví dụ, giả sử rằng ta huấn luyện một mô hình để dự đoán xem một người có trả được nợ hay không, rồi mô hình chỉ ra rằng việc chọn giày dép của ứng viên có liên quan đến rủi ro vỡ nợ (giày tây thì trả được nợ, giày thể thao thì không). Từ đó, ta có thể sẽ có xu hướng chỉ cấp các khoản vay cho các ứng viên mang giày tây và sẽ từ chối cho vay đối với những trường hợp mang giày thể thao.

Trong trường hợp này, việc ta không cân nhắc kỹ khi nhảy vọt từ nhận dạng khuôn mẫu đến ra quyết định và việc không nghiêm túc xem xét các yếu tố môi trường có thể gây ra hậu quả nghiêm trọng. Như ví dụ trên, không sớm thì muộn khi ta bắt đầu đưa ra quyết định dựa trên kiểu giày, khách hàng sẽ để ý và thay đổi hành vi của họ. Chẳng bao lâu sau, tất cả các người vay tiền sẽ mang giày tây, nhưng chỉ số tín dụng của họ thì không hề cải thiện. Hãy dành một phút để “thẩm” điều này vì có rất nhiều vấn đề tương tự trong các ứng dụng của học máy: bằng việc ra quyết định dựa trên mô hình trong một môi trường, ta có thể làm hỏng chính mô hình đó.

Dù không thể thảo luận kỹ lưỡng về các vấn đề này chỉ trong một mục, chúng tôi vẫn muốn đề cập một vài mối bận tâm phổ biến và kích thích tư duy phản biện để có thể sớm phát hiện ra các tình huống này, từ đó giảm thiểu thiệt hại và có trách nhiệm hơn trong việc sử dụng học máy. Một vài giải pháp khá đơn giản (thu thập dữ liệu “phù hợp”), còn một vài giải pháp lại khó hơn về mặt kỹ thuật (lập trình một hệ thống học tăng cường), và một số khác thì hoàn toàn nằm ngoài lĩnh vực dự đoán thống kê và cần ta phải vật lộn với các câu hỏi triết học khó khăn về khía cạnh đạo đức trong việc ứng dụng thuật toán.

### 6.9.1 Dịch chuyển Phân phối

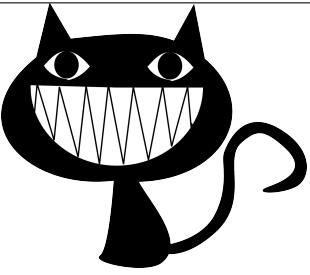
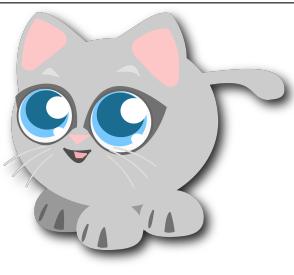
Để bắt đầu, ta sẽ trở lại vị trí quan sát và tạm gác lại các tác động của ta lên môi trường. Trong các mục tiếp theo, ta sẽ xem xét kỹ vài cách khác nhau mà phân phối dữ liệu có thể dịch chuyển và những gì ta có thể làm để cứu vãn hiệu suất mô hình. Chúng tôi sẽ cảnh báo ngay từ đầu rằng nếu phân phối sinh dữ liệu  $p(\mathbf{x}, y)$  có thể dịch chuyển theo các cách khác nhau tại bất kỳ thời điểm nào, việc học một bộ phân loại mạnh mẽ là điều bất khả thi. Trong trường hợp xấu nhất, nếu bản thân định nghĩa của nhãn có thể thay đổi bất cứ khi nào: nếu đột nhiên con vật mà chúng ta gọi là “mèo” bây giờ là “chó” và con vật trước đây chúng ta gọi là “chó” thì giờ lại là “mèo”, trong khi không có bất kỳ thay đổi rõ ràng nào trong phân phối của đầu vào  $p(\mathbf{x})$ , thì không có cách nào để phát hiện được sự thay đổi này hay điều chỉnh lại bộ phân loại tại thời điểm kiểm tra. May mắn thay, dưới một vài giả định chặt chẽ về cách dữ liệu có thể thay đổi trong tương lai, một vài thuật toán có thể phát hiện được sự dịch chuyển và thậm chí có thể thích nghi để đạt được độ chính xác cao hơn so với việc tiếp tục dựa vào bộ phân loại ban đầu một cách ngây thơ.

## Dịch chuyển Hiệp biến

Một trong những dạng dịch chuyển phân phối được nghiên cứu rộng rãi nhất là *dịch chuyển hiệp biến* (*covariate shift*). Ở đây, ta giả định rằng mặc dù phân phối đầu vào có thể biến đổi theo thời gian, nhưng hàm gán nhãn, tức phân phối có điều kiện  $P(y | \mathbf{x})$  thì không thay đổi. Mặc dù vấn đề này khá dễ hiểu, trong thực tế nó thường dễ bị bỏ qua. Hãy xem xét bài toán phân biệt mèo và chó với tập dữ liệu huấn luyện bao gồm các ảnh sau:

mèo	mèo	chó	chó
			

Tại thời điểm kiểm tra ta phải phân loại các ảnh dưới đây:

mèo	mèo	chó	chó
			

Rõ ràng việc phân loại tốt trong trường hợp này là rất khó khăn. Trong khi tập huấn luyện bao gồm các ảnh đời thực thì tập kiểm tra chỉ chứa các ảnh hoạt hình với màu sắc thậm chí còn không thực tế. Việc huấn luyện trên một tập dữ liệu khác biệt đáng kể so với tập kiểm tra mà không có một kế hoạch để thích ứng với sự thay đổi này là một ý tưởng tồi. Thật không may, đây lại là một cạm bẫy rất phổ biến. Các nhà thống kê gọi vấn đề này là *dịch chuyển hiệp biến* bởi vì gốc rễ của nó là do sự thay đổi trong phân phối của các đặc trưng (tức các *hiệp biến*). Theo ngôn ngữ toán học, ta có thể nói rằng  $P(\mathbf{x})$  thay đổi nhưng  $P(y | \mathbf{x})$  thì không. Khi ta tin rằng  $\mathbf{x}$  gây ra  $y$  thì dịch chuyển hiệp biến thường là một giả định hợp lý, mặc dù tính hữu dụng của nó không chỉ giới hạn trong trường hợp này.

## Dịch chuyển Nhãn

Vấn đề ngược lại xuất hiện khi chúng ta tin rằng điều gây ra sự dịch chuyển là một thay đổi trong phân phối biên của nhãn  $P(y)$  trong khi phân phối có điều kiện theo lớp  $P(\mathbf{x} | y)$  vẫn không đổi. Dịch chuyển nhãn là một giả định hợp lý khi chúng ta tin rằng  $y$  gây ra  $\mathbf{x}$ . Chẳng hạn, thông thường chúng ta muốn dự đoán kết quả chẩn đoán nếu biết các biểu hiện của bệnh. Trong trường hợp này chúng ta tin rằng kết quả chẩn đoán gây ra các biểu hiện, tức bệnh gây ra các triệu chứng. Thỉnh thoảng các giả định dịch chuyển nhãn và dịch chuyển hiệp biến có thể xảy ra đồng thời. Ví dụ, khi hàm gán nhãn là tất định và không đổi, dịch chuyển hiệp biến sẽ luôn xảy ra, kể cả khi dịch chuyển nhãn cũng đang xảy ra. Một điều thú vị là khi chúng ta tin rằng cả dịch chuyển nhãn và

dịch chuyển hiệp biến đều đang xảy ra, làm việc với các phương pháp được suy ra từ giả định dịch chuyển nhãn thường chiếm lợi thế. Các phương pháp này thường sẽ dễ làm việc hơn vì chúng thao tác trên nhãn thay vì trên các đầu vào đa chiều trong học sâu.

### Dịch chuyển Khái niệm

Một vấn đề liên quan nữa nằm ở việc *dịch chuyển khái niệm* (*concept shift*), khi chính định nghĩa của các nhãn thay đổi. Điều này nghe có vẻ lạ vì sau cùng, con mèo là con mèo. Quả thực định nghĩa của một con mèo có thể không thay đổi, nhưng điều này có đúng với thuật ngữ “đồ uống có ga” hay không? Hoá ra nếu chúng ta di chuyển vòng quanh nước Mỹ, dịch chuyển nguồn dữ liệu theo vùng địa lý, ta sẽ thấy sự dịch chuyển khái niệm đáng kể liên quan đến thuật ngữ đơn giản này như thể hiện trong Fig. 6.9.1.

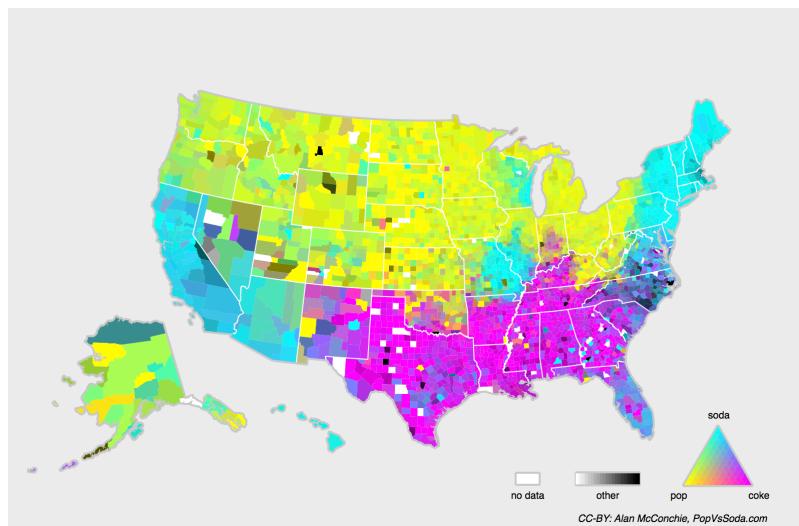


Fig. 6.9.1: Dịch chuyển khái niệm của tên các loại đồ uống có ga ở Mỹ.

Nếu chúng ta xây dựng một hệ thống dịch máy, phân phối  $P(y | x)$  có thể khác nhau tùy thuộc vào vị trí của chúng ta. Vấn đề này có thể khó nhận ra, nhưng bù lại  $P(y | x)$  thường chỉ dịch chuyển một cách chậm rãi.

## Ví dụ

Trước khi đi vào chi tiết và thảo luận các giải pháp, ta có thể bàn thêm về một số tình huống khi dịch chuyển hiệp biến và khái niệm có thể có biểu hiện không quá rõ ràng.

### Chẩn đoán Y khoa

Hãy tưởng tượng rằng bạn muốn thiết kế một giải thuật có khả năng phát hiện bệnh ung thư. Bạn thu thập dữ liệu từ cả người khỏe mạnh lẫn người bệnh rồi sau đó huấn luyện giải thuật. Nó hoạt động hiệu quả, có độ chính xác cao và bạn kết luận rằng bạn đã sẵn sàng cho một sự nghiệp chẩn đoán y khoa thành công. Đừng vội mừng...

Bạn có thể đã mắc nhiều sai lầm. Cụ thể, các phân phổi mà bạn dùng để huấn luyện và các phân phổi bạn gặp phải trong thực tế có thể rất khác nhau. Điều này đã từng xảy ra với một công ty khởi nghiệp không may mắn mà tôi đã tư vấn nhiều năm về trước. Họ đã phát triển một bộ xét nghiệm máu cho một căn bệnh xảy ra chủ yếu ở đàn ông lớn tuổi và họ đã thu thập được một lượng khá lớn máu từ các bệnh nhân. Mặc dù vậy, việc thu thập mẫu máu từ những người đàn ông khỏe mạnh lại khó khăn hơn (chủ yếu là vì lý do đạo đức). Để giải quyết sự thiếu hụt này, họ đã kêu gọi một lượng lớn các sinh viên trong trường học tham gia hiến máu tình nguyện để thực hiện xét nghiệm máu của họ. Sau đó họ đã nhờ tôi xây dựng một bộ phân loại để phát hiện căn bệnh. Tôi đã nói với họ rằng việc phân biệt hai tập dữ liệu trên với độ chính xác gần như hoàn hảo là rất dễ dàng. Sau cùng, các đối tượng kiểm tra có nhiều khác biệt về tuổi, nồng độ hóc môn, hoạt động thể chất, chế độ ăn uống, mức tiêu thụ rượu bia, và nhiều nhân tố khác không liên quan đến căn bệnh. Điều này không giống với trường hợp của những bệnh nhân thật sự: Quy trình lấy mẫu của họ khả năng cao đã gây ra hiện tượng dịch chuyển hiệp biến rất nặng giữa phân phổi gốc và phân phổi mục tiêu, và thêm vào đó, nó không thể được khắc phục bằng các biện pháp thông thường. Nói cách khác, dữ liệu huấn luyện và kiểm tra khác biệt đến nỗi không thể xây dựng được một mô hình hữu dụng và họ đã lãng phí rất nhiều tiền của.

### Xe tự hành

Giả sử có một công ty muốn xây dựng một hệ thống học máy cho xe tự hành. Một trong những bộ phận quan trọng là bộ phát hiện lề đường. Vì việc gán nhãn dữ liệu thực tế rất tốn kém, họ đã có một ý tưởng (thông minh và đầy nghiêm) là sử dụng dữ liệu giả từ một bộ kết xuất đồ họa để thêm vào dữ liệu huấn luyện. Nó đã hoạt động rất tốt trên “dữ liệu kiểm tra” được lấy mẫu từ bộ kết xuất đồ họa. Nhưng khi áp dụng trên xe thực tế, nó là một thảm họa. Hoá ra, lề đường đã được kết xuất chỉ với một kết cấu rất đơn giản. Quan trọng hơn, *tất cả* các lề đường đều được kết xuất với *cùng một* kết cấu và bộ phát hiện lề đường đã nhanh chóng học được “đặc trưng” này.

Một điều tương tự cũng đã xảy ra với quân đội Mỹ trong lần đầu tiên họ thử nghiệm nhận diện xe tăng trong rừng. Họ chụp các bức ảnh khu rừng từ trên cao khi không có xe tăng, sau đó lái xe tăng vào khu rừng và chụp một bộ ảnh khác. Bộ phân loại này được huấn luyện tới mức “hoàn hảo”. Không may thay, tất cả những gì nó đã học được là phân loại cây có bóng với cây không có bóng—bộ ảnh đầu tiên được chụp vào buổi sáng sớm, trong khi bộ thứ hai được chụp vào buổi trưa.

## Phân phối không dừng

Một vấn đề khó phát hiện hơn phát sinh khi phân phối thay đổi chậm rãi và mô hình không được cập nhật một cách thỏa đáng. Dưới đây là một vài trường hợp điển hình:

- Chúng ta huấn luyện mô hình quảng cáo điện toán và sau đó không cập nhật nó thường xuyên (chẳng hạn như quên bổ sung thêm thiết bị iPad mới vừa được ra mắt vào mô hình).
- Xây dựng một mô hình lọc thư rác. Mô hình làm việc rất tốt cho việc phát hiện tất cả các thư rác mà chúng ta biết cho đến nay. Nhưng rồi những người gửi thư rác cũng khôn khéo hơn và tạo ra các mẫu thư mới khác hẳn với những thư trước đây.
- Ta xây dựng hệ thống để xuất sản phẩm. Hệ thống làm việc tốt trong suốt mùa đông... nhưng sau đó nó vẫn tiếp tục để xuất các mẫu nón ông già Noel ngay cả khi Giáng Sinh đã qua từ lâu.

## Các giai thoại khác

- Chúng ta xây dựng mô hình phát hiện gương mặt. Nó hoạt động rất tốt trên các bài kiểm tra đánh giá. Không may mắn là mô hình lại thất bại trên tập dữ liệu kiểm tra—các ví dụ đánh bại được mô hình khi khuôn mặt lấp đầy hoàn toàn cả bức ảnh, trong khi không có dữ liệu nào tương tự như vậy xuất hiện trong tập huấn luyện.
- Ta xây dựng hệ thống tìm kiếm web cho thị trường Hoa Kỳ và hiện tại muốn triển khai nó cho thị trường Anh.
- Chúng ta huấn luyện một bộ phân loại hình ảnh bằng cách biên soạn một tập dữ liệu lớn, trong đó mỗi lớp trong tập dữ liệu đều có số lượng mẫu bằng nhau, ví dụ 1000 lớp và mỗi lớp được biểu diễn bởi 1000 ảnh. Sau đó chúng ta triển khai hệ thống trong khi trên thực tế phân phối của nhãn chắc chắn là không đồng đều.

Chung quy lại, có nhiều trường hợp mà phân phối huấn luyện và kiểm tra  $p(\mathbf{x}, y)$  là khác nhau. Trong một số trường hợp may mắn thì các mô hình vẫn chạy tốt dù phân phối của hiệp biến, nhãn hay khái niệm đều dịch chuyển. Trong một số trường hợp khác, chúng ta có thể làm tốt hơn bằng cách sử dụng nhiều chiến lược một cách có nguyên tắc để đối phó với sự dịch chuyển này. Phần còn lại của mục này sẽ tập trung nhiều hơn hẳn vào mặt kỹ thuật. Tuy nhiên, những độc giả vội vàng có thể bỏ qua mục này vì các khái niệm được trình bày dưới đây không phải là tiền đề cho các phần tiếp theo.

## Hiệu chỉnh Dịch chuyển Hiệp biến

Giả sử rằng ta muốn ước lượng mối liên hệ phụ thuộc  $P(y | \mathbf{x})$  khi đã có dữ liệu được gán nhãn  $(\mathbf{x}_i, y_i)$ . Thật không may, các mẫu quan sát  $x_i$  được thu thập từ một phân phối *mục tiêu*  $q(\mathbf{x})$  thay vì từ phân phối gốc  $p(\mathbf{x})$ . Để có được tiến triển, chúng ta cần nhìn lại xem chính xác thì việc gì đang diễn ra trong quá trình huấn luyện: ta duyệt qua tập dữ liệu huấn luyện cùng với nhãn kèm theo  $\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$  và cập nhật vector trọng số của mô hình sau mỗi minibatch. Đôi khi chúng ta cũng áp dụng thêm một lượng phạt nào đó lên các tham số, bằng cách dùng suy giảm trọng số, dropout hoặc các kỹ thuật liên quan khác. Điều này nghĩa là ta hâu như chỉ đang giảm thiểu giá trị mất mát trên tập huấn luyện.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^n l(x_i, y_i, f(x_i)) + \text{một lượng phạt}(w). \quad (6.9.1)$$

Các nhà thống kê gọi số hạng đầu tiên là *trung bình thực nghiệm*, tức trung bình được tính qua dữ liệu lấy từ phân phối  $P(x)P(y | x)$ . Nếu dữ liệu được lấy “nhầm” từ phân phối  $q$ , ta có thể hiệu chỉnh lại bằng cách sử dụng đồng nhất thức:

$$\int p(\mathbf{x})f(\mathbf{x})dx = \int q(\mathbf{x})f(\mathbf{x})\frac{p(\mathbf{x})}{q(\mathbf{x})}dx. \quad (6.9.2)$$

Nói cách khác, chúng ta cần đánh lại trọng số cho mỗi mẫu bằng tỉ lệ của các xác suất mà mẫu được lấy từ đúng phân phối  $\beta(\mathbf{x}) := p(\mathbf{x})/q(\mathbf{x})$ . Đáng buồn là ta không biết tỉ lệ đó, nên trước khi ta có thể làm được bất cứ thứ gì hữu ích thì ta cần phải ước lượng được nó. Nhiều phương pháp có sẵn sử dụng cách tiếp cận lý thuyết toán tử màu mè nhằm cố gắng trực tiếp toán tử kỳ vọng bằng cách sử dụng nguyên lý chuẩn cực tiểu hay entropy cực đại. Lưu ý rằng những phương thức này yêu cầu ta lấy mẫu từ cả phân phối “đúng”  $p$  (bằng cách sử dụng tập huấn luyện) và phân phối được dùng để tạo ra tập kiểm tra  $q$  (việc này hiển nhiên là có thể được). Tuy nhiên cũng cần để ý là ta chỉ cần mẫu  $\mathbf{x} \sim q(\mathbf{x})$ ; ta không cần sử dụng đến nhãn  $y \sim q(y)$ .

Trong trường hợp này có một cách tiếp cận rất hiệu quả và sẽ cho kết quả tốt gần ngang ngửa, đó là: hồi quy logistic. Đây là tất cả những gì ta cần để tính xấp xỉ tỉ lệ xác suất. Chúng ta cho học một bộ phân loại để phân biệt giữa dữ liệu được lấy từ phân phối  $p(\mathbf{x})$  và phân phối  $q(\mathbf{x})$ . Nếu không thể phân biệt được giữa hai phân phối thì điều đó có nghĩa là khả năng các mẫu liên quan đến từ một trong hai phân phối là ngang nhau. Mặt khác, bất kì mẫu nào mà có thể được phân biệt dễ dàng thì cần được đánh trọng số tăng lên hoặc giảm đi tương ứng. Để cho đơn giản, giả sử ta có số lượng mẫu đến từ hai phân phối là bằng nhau, được kí hiệu lần lượt là  $\mathbf{x}_i \sim p(\mathbf{x})$  và  $\mathbf{x}'_i \sim q(\mathbf{x})$ . Ta kí hiệu nhãn  $z_i$  bằng 1 cho dữ liệu từ phân phối  $p$  và -1 cho dữ liệu từ  $q$ . Lúc này xác suất trong một bộ dữ liệu được trộn lẫn sẽ là

$$P(z = 1 | \mathbf{x}) = \frac{p(\mathbf{x})}{p(\mathbf{x}) + q(\mathbf{x})} \text{ và từ đó } \frac{P(z = 1 | \mathbf{x})}{P(z = -1 | \mathbf{x})} = \frac{p(\mathbf{x})}{q(\mathbf{x})}. \quad (6.9.3)$$

Do đó, nếu sử dụng cách tiếp cận hồi quy logistic mà trong đó  $P(z = 1 | \mathbf{x}) = \frac{1}{1 + \exp(-f(\mathbf{x}))}$ , ta có

$$\beta(\mathbf{x}) = \frac{1/(1 + \exp(-f(\mathbf{x})))}{\exp(-f(\mathbf{x}))/(1 + \exp(-f(\mathbf{x})))} = \exp(f(\mathbf{x})). \quad (6.9.4)$$

Vì vậy, có hai bài toán cần được giải quyết: đầu tiên là bài toán phân biệt giữa dữ liệu được lấy ra từ hai phân phối, và sau đó là bài toán cực tiểu hóa với trọng số cho các mẫu được đánh lại với  $\beta$ , ví dụ như thông qua các gradient đầu. Dưới đây là một thuật toán nguyên mẫu để giải quyết hai bài toán trên. Thuật toán này sử dụng tập huấn luyện không được gán nhãn  $X$  và tập kiểm tra  $Z$ :

1. Tạo một tập huấn luyện với  $\{(\mathbf{x}_i, -1) \dots (\mathbf{z}_j, 1)\}$ .
2. Huấn luyện một bộ phân loại nhị phân sử dụng hồi quy logistic để tìm hàm  $f$ .
3. Đánh trọng số cho dữ liệu huấn luyện bằng cách sử dụng  $\beta_i = \exp(f(\mathbf{x}_i))$ , hoặc tốt hơn là  $\beta_i = \min(\exp(f(\mathbf{x}_i)), c)$ .
4. Sử dụng trọng số  $\beta_i$  để huấn luyện trên  $X$  với nhãn  $Y$ .

Lưu ý rằng phương pháp này được dựa trên một giả định quan trọng. Để có được một kết quả tốt, ta cần đảm bảo rằng mỗi điểm dữ liệu trong phân phối mục tiêu (tại thời điểm kiểm tra) có xác suất xảy ra tại thời điểm huấn luyện khác không. Nếu một điểm có  $q(\mathbf{x}) > 0$  nhưng  $p(\mathbf{x}) = 0$ , thì trọng số quan trọng tương ứng bằng vô hạn.

*Mạng Đối sinh* sử dụng một ý tưởng rất giống với mô tả ở trên để thiết kế một bộ sinh dữ liệu có khả năng tạo dữ liệu không thể phân biệt được với các mẫu được lấy từ một tập dữ liệu tham chiếu.

Trong các phương pháp này, ta sử dụng một mạng  $f$  để phân biệt dữ liệu thật với dữ liệu giả, và một mạng thứ hai  $g$  cố gắng đánh lừa bộ phân biệt  $f$  rằng dữ liệu giả là thật. Ta sẽ thảo luận vấn đề này một cách chi tiết hơn sau.

### Hiệu chỉnh Dịch chuyển nhãn

Để thảo luận về dịch chuyển nhãn, giả định rằng ta đang giải quyết một bài toán phân loại  $k$  lớp. Nếu phân phối của nhãn thay đổi theo thời gian  $p(y) \neq q(y)$  nhưng các phân phối có điều kiện của lớp vẫn giữ nguyên  $p(\mathbf{x}) = q(\mathbf{x})$ , thì trọng số quan trọng sẽ tương ứng với tỉ lệ hợp lý (*likelihood ratio*) của nhãn  $q(y)/p(y)$ . Một điều tốt về dịch chuyển nhãn là nếu ta có một mô hình tương đối tốt (trên phân phối gốc), ta có thể có các ước lượng nhất quán cho các trọng số này mà không phải làm việc với không gian đầu vào (trong học sâu, đầu vào thường là dữ liệu nhiều chiều như hình ảnh, trong khi làm việc với các nhãn thường dễ hơn vì chúng chỉ là các vector có chiều dài tương ứng với số lượng lớp).

Để ước lượng phân phối nhãn mục tiêu, đầu tiên ta dùng một bộ phân loại sẵn có tương đối tốt (thường được học trên tập huấn luyện) và sử dụng một tập kiểm định (cùng phân phối với tập huấn luyện) để tính ma trận nhầm lẫn. Ma trận nhầm lẫn  $C$  là một ma trận  $k \times k$ , trong đó mỗi cột tương ứng với một nhãn *thật* và mỗi hàng tương ứng với nhãn *dự đoán* của mô hình. Giá trị của mỗi phần tử  $c_{ij}$  là số lượng mẫu có nhãn thật là  $j$  và nhãn dự đoán là  $i$ .

Giờ thì ta không thể tính trực tiếp ma trận nhầm lẫn trên dữ liệu mục tiêu được bởi vì ta không thể quan sát được nhãn của các mẫu trong thực tế, trừ khi ta đầu tư vào một pipeline phức tạp để đánh nhãn theo thời gian thực. Tuy nhiên điều mà ta có thể làm là lấy trung bình tất cả dự đoán của mô hình tại lúc kiểm tra, từ đó có được giá trị đầu ra trung bình của mô hình  $\mu_y$ .

Hoá ra là dưới các giả định đơn giản – chẳng hạn như bộ phân loại vốn đã khá chính xác, dữ liệu mục tiêu chỉ chứa ảnh thuộc các lớp đã quan sát được từ trước, và giả định dịch chuyển nhãn là đúng (đây là giả định lớn nhất tới bây giờ), thì ta có thể khôi phục phân phối nhãn trên tập kiểm tra bằng cách giải một hệ phương trình tuyến tính đơn giản  $C \cdot q(y) = \mu_y$ . Nếu bộ phân loại đã khá chính xác ngay từ đầu thì ma trận nhầm lẫn  $C$  là khả nghịch và ta có nghiệm  $q(y) = C^{-1} \mu_y$ . Ở đây ta đang lạm dụng kí hiệu một chút khi sử dụng  $q(y)$  để kí hiệu vector tần suất nhãn. Vì ta quan sát được nhãn trên dữ liệu gốc, nên có thể dễ dàng ước lượng phân phối  $p(y)$ . Sau đó với bất kì mẫu huấn luyện  $i$  với nhãn  $y$ , ta có thể lấy tỉ lệ ước lượng  $\hat{q}(y)/\hat{p}(y)$  để tính trọng số  $w_i$  và đưa vào thuật toán cực tiểu hóa rủi ro có trọng số được mô tả ở trên.

### Hiệu chỉnh Dịch chuyển Khái niệm

Khắc phục vấn đề dịch chuyển khái niệm theo một cách có nguyên tắc khó hơn rất nhiều. Chẳng hạn như khi bài toán đột nhiên chuyển từ phân biệt chó và mèo sang phân biệt động vật có màu trắng và động vật có màu đen, hoàn toàn có lý khi tin rằng ta không thể làm gì tốt hơn ngoài việc thu thập tập nhãn mới và huấn luyện lại từ đầu. May mắn thay vấn đề dịch chuyển tới mức cực đoan như vậy trong thực tế khá hiếm. Thay vào đó, điều thường diễn ra là tác vụ cứ dần dần thay đổi. Để làm rõ hơn, ta xét các ví dụ dưới đây:

- Trong ngành quảng cáo điện toán, khi một sản phẩm mới ra mắt, các sản phẩm cũ trở nên ít phổ biến hơn. Điều này nghĩa là phân phối của các mẫu quảng cáo và mức phổ biến của chúng sẽ thay đổi dần dần và bất kì bộ dự đoán tỉ lệ click chuột nào cũng cần thay đổi theo.
- Ống kính của các camera giao thông bị mờ đi theo thời gian do tác động của môi trường, có ảnh hưởng lớn dần tới chất lượng ảnh.

- Nội dung các mẫu tin thay đổi theo thời gian, tức là tin tức thì không đổi nhưng các sự kiện mới luôn diễn ra.

Với các trường hợp trên, ta có thể sử dụng cùng cách tiếp cận trong việc huấn luyện mô hình để chúng thích ứng với các biến đổi trong dữ liệu. Nói cách khác, chúng ta sử dụng trọng số đang có của mạng và chỉ thực hiện thêm vài bước cập nhật trên dữ liệu mới thay vì huấn luyện lại từ đầu.

### 6.9.2 Phân loại các Bài toán Học máy

Ta đã được trang bị kiến thức về cách xử lý các thay đổi trong  $p(x)$  và  $P(y | x)$ , giờ đây ta có thể xem xét một số khía cạnh khác của việc xây dựng các bài toán học máy.

- Học theo batch.** Ở đây ta có dữ liệu và nhãn huấn luyện  $\{(x_1, y_1), \dots, (x_n, y_n)\}$ , được sử dụng để huấn luyện mạng  $f(x, w)$ . Sau đó, ta dùng mô hình này để đánh giá điểm dữ liệu mới  $(x, y)$  được lấy từ cùng một phân phối. Đây là giả thuyết mặc định cho bất kỳ bài toán nào mà ta bàn ở đây. Ví dụ, ta có thể huấn luyện một mô hình phát hiện mèo dựa trên nhiều hình ảnh của mèo và chó. Sau khi hoàn tất quá trình huấn luyện, ta đưa mô hình vào một hệ thống thi giác máy tính cho cửa sổ thông minh mà chỉ cho phép mèo đi vào. Hệ thống này sẽ được lắp đặt tại nhà của khách hàng và nó không bao giờ được cập nhật lại (ngoại trừ một vài trường hợp hiếm hoi).
- Học trực tuyến.** Bây giờ hãy tưởng tượng rằng tại một thời điểm ta chỉ nhận được một mẫu dữ liệu  $(x_i, y_i)$ . Cụ thể hơn, giả sử đầu tiên ta có một quan sát  $x_i$ , sau đó ta cần tính  $f(x_i, w)$  và chỉ khi ta hoàn thành việc đưa ra quyết định, ta mới có thể quan sát giá trị  $y_i$ , rồi dựa vào nó mà nhận lại phần thưởng (hoặc chịu mất mát). Nhiều bài toán thực tế rơi vào loại này. Ví dụ, ta cần dự đoán giá cổ phiếu vào ngày mai, điều này cho phép ta giao dịch dựa trên dự đoán đó và vào cuối ngày ta sẽ biết được liệu nó có mang lại lợi nhuận hay không. Nói cách khác, ta có chu trình sau, trong đó mô hình dần được cải thiện cùng với những quan sát mới.

mô hình  $f_t \rightarrow$  dữ liệu  $x_t \rightarrow$  ước lượng  $f_t(x_t) \rightarrow$  quan sát  $y_t \rightarrow$  mất mát  $l(y_t, f_t(x_t)) \rightarrow$  mô hình  $f_{t+1}$  (6.9.5)

- Máy đánh bạc.** Đây là một *trường hợp đặc biệt* của bài toán trên. Trong khi ở hầu hết các bài toán ta luôn có một hàm liên tục được tham số hóa  $f$  và công việc của ta là học các tham số của nó (ví dụ như một mạng học sâu), trong bài toán máy đánh bạc ta chỉ có một số hữu hạn các cờ mà ta có thể gạt (tức một số lượng giới hạn những hành động mà ta có thể thực hiện). Không có gì ngạc nhiên khi với bài toán đơn giản này, ta có được các cơ sở lý thuyết tối ưu mạnh mẽ hơn. Chúng tôi liệt kê nó ở đây chủ yếu là vì bài toán này thường được xem (một cách nhầm lẫn) như là một môi trường học tập khác biệt.
- Kiểm soát (và Học Tăng cường phi đối kháng).** Trong nhiều trường hợp, môi trường ghi nhớ những gì ta đã làm. Việc này không nhất thiết phải có tính chất đối kháng, môi trường chỉ nhớ và phản hồi phụ thuộc vào những gì đã xảy ra trước đó. Ví dụ, bộ điều khiển của ấm pha cà phê sẽ quan sát được nhiệt độ khác nhau tùy thuộc vào việc nó có đun ấm trước đó không. Giải thuật điều khiển PID (*proportional integral derivative* hay *vi tích phân tỉ lệ*) là một lựa chọn phổ biến để làm điều đó. Tương tự như vậy, hành vi của người dùng trên một trang tin tức sẽ phụ thuộc vào những gì ta đã cho họ xem trước đây (chẳng hạn như là mỗi người chỉ đọc mỗi mẫu tin một lần duy nhất). Nhiều thuật toán như vậy cấu thành một mô hình của môi trường mà trong đó chúng muốn làm cho các quyết định của mình trông có vẻ ít ngẫu nhiên hơn (tức để giảm phương sai).
- Học Tăng cường.** Trong trường hợp khái quát hơn của môi trường có khả năng ghi nhớ, ta có thể gặp phải tình huống môi trường đang cố gắng *hợp tác* với ta (trò chơi hợp tác, đặc biệt là các trò chơi có tổng khác không), hoặc môi trường sẽ cố gắng *chiến thắng* ta như Cờ vua,

Còn vây, Backgammon hay StarCraft. Tương tự như vậy, có thể ta muốn xây dựng một bộ điều khiển tốt cho những chiếc xe tự hành. Những chiếc xe khác khả năng cao sẽ có những phản ứng đáng kể với cách lái của những chiếc xe tự hành, ví dụ như cố gắng tránh nó, cố gắng gây ra tai nạn, cố gắng hợp tác với nó, v.v.

Điểm khác biệt mấu chốt giữa các tình huống khác nhau ở trên là: một chiến lược hoạt động xuyên suốt các môi trường cố định, có thể lại không hoạt động xuyên suốt được khi môi trường có khả năng thích nghi. Chẳng hạn, nếu một thương nhân phát hiện ra cơ hội kiếm lời từ chênh lệch giá cả thị trường, khả năng cao cơ hội đó sẽ biến mất ngay khi anh ta bắt đầu lợi dụng nó. Tốc độ và cách môi trường thay đổi có ảnh hưởng lớn đến loại thuật toán mà ta có thể sử dụng. Ví dụ, nếu ta biết trước mọi việc chỉ có thể thay đổi một cách từ từ, ta có thể ép những ước lượng phải thay đổi dần theo. Còn nếu ta biết môi trường có thể thay đổi ngay lập tức, nhưng không thường xuyên, ta có thể cho phép điều này xảy ra. Đối với các nhà khoa học dữ liệu giỏi, những kiến thức này rất quan trọng trong việc giải quyết các toán dịch chuyển khái niệm khi vấn đề cần giải quyết lại thay đổi theo thời gian.

### 6.9.3 Công bằng, Trách nhiệm và Minh bạch trong Học máy

Cuối cùng, cần ghi nhớ một điều quan trọng sau đây: khi triển khai một hệ thống học máy, bạn không chỉ đơn thuần cực tiểu hóa hàm đối log hợp lý hay cực đại hóa độ chính xác mà còn đang tự động hóa một quy trình quyết định nào đó. Thường thì những hệ thống được tự động hóa việc ra quyết định mà chúng ta triển khai có thể sẽ gây ra những hậu quả cho những ai chịu ảnh hưởng bởi quyết định của nó. Nếu chúng ta triển khai một hệ thống chẩn đoán y khoa, ta cần biết hệ thống này sẽ hoạt động và không hoạt động với những ai. Bỏ qua những rủi ro có thể lường trước được để chạy theo phúc lợi của một bộ phận dân số sẽ đi ngược lại những nguyên tắc đạo đức cơ bản. Ngoài ra, “độ chính xác” hiếm khi là một thước đo đúng. Khi chuyển những dự đoán thành hành động, chúng ta thường để ý đến chi phí tiềm tàng của các loại lỗi khác nhau. Nếu kết quả phân loại một bức ảnh có thể được xem như một sự phân biệt chủng tộc, trong khi việc phân loại sai sang một lớp khác thì lại vô hại, bạn có thể sẽ muốn cân nhắc cả các giá trị xã hội khi điều chỉnh ngưỡng của hệ thống ra quyết định đó. Ta cũng muốn cẩn thận về cách những hệ thống dự đoán có thể dẫn đến vòng lặp phản hồi. Ví dụ, nếu hệ thống dự đoán được áp dụng theo cách ngây ngô để dự đoán các hành động phi pháp và theo đó phân bổ sĩ quan tuần tra, một vòng luẩn quẩn có thể xuất hiện. Một khu xóm có nhiều tội phạm hơn sẽ có nhiều sĩ quan tuần tra hơn, phát hiện ra nhiều tội phạm hơn, thêm nhiều dữ liệu huấn luyện, nhận được dự đoán tốt hơn, dẫn đến nhiều sĩ quan tuần tra hơn, và càng nhiều tội ác được phát hiện,... Thêm vào đó, chúng ta cũng muốn cẩn thận ngay từ đầu về việc chúng ta có đang giải quyết đúng vấn đề hay không. Hiện tại, các thuật toán dự đoán đóng một vai trò lớn khi làm bên trung gian trong việc phân tán thông tin. Những tin tức nào được hiển thị đến người dùng có nên được quyết định bởi những trang Facebook nào mà họ *đã thích* hay không? Đây chỉ là một số trong rất nhiều vấn đề về đạo đức mà bạn có thể bắt gặp trong việc theo đuổi sự nghiệp học máy của mình.

#### 6.9.4 Tóm tắt

- Trong nhiều trường hợp, tập huấn luyện và tập kiểm tra không được lấy mẫu từ cùng một phân phối. Đây là hiện tượng dịch chuyển hiệp biến.
- Dịch chuyển hiệp biến có thể được phát hiện và khắc phục nếu sự dịch chuyển không quá nghiêm trọng. Thất bại trong việc khắc phục có thể dẫn đến những kết quả không lường được lúc kiểm thử.
- Trong nhiều trường hợp, môi trường sẽ ghi nhớ những gì chúng ta đã làm và sẽ phản hồi theo những cách không lường trước được. Chúng ta cần xem xét điều này khi xây dựng mô hình.

#### 6.9.5 Bài tập

1. Điều gì có thể xảy ra khi chúng ta thay đổi hành vi của công cụ tìm kiếm? Người dùng có thể sẽ làm gì? Còn các nhà quảng cáo thì sao?
2. Xây dựng một chương trình phát hiện dịch chuyển hiệp biến. Gợi ý: hãy xây dựng một hệ thống phân lớp.
3. Xây dựng một chương trình khắc phục vấn đề dịch chuyển hiệp biến.
4. Chuyện tồi tệ gì có thể xảy ra nếu tập huấn luyện và tập kiểm tra rất khác nhau? Chuyện gì sẽ xảy ra đối với trọng số mẫu?

#### 6.9.6 Thảo luận

- Tiếng Anh<sup>118</sup>
- Tiếng Việt<sup>119</sup>

#### 6.9.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Lý Phi Long
- Lê Khắc Hùng Phúc
- Nguyễn Duy Du
- Phạm Minh Đức
- Lê Cao Thăng
- Nguyễn Minh Thư
- Nguyễn Thành Nhân
- Phạm Hồng Vinh

<sup>118</sup> <https://discuss.mxnet.io/t/2347>

<sup>119</sup> <https://forum.machinelearningcoban.com/c/d21>

## 6.10 Dự đoán Giá Nhà trên Kaggle

Trong phần trước, chúng tôi đã giới thiệu những công cụ cơ bản để xây dựng mạng học sâu và kiểm soát năng lực của nó thông qua việc giảm chiều dữ liệu, suy giảm trọng số và dropout. Giờ bạn đã sẵn sàng để ứng dụng tất cả những kiến thức này vào thực tiễn bằng cách tham gia một cuộc thi trên Kaggle. [Dự đoán giá nhà](#)<sup>120</sup> là một bài toán tuyệt vời để bắt đầu: dữ liệu tương đối khái quát, không có cấu trúc cứng nhắc nên không đòi hỏi những mô hình đặc biệt như các bài toán có dữ liệu ảnh và âm thanh. Tập dữ liệu này được thu thập bởi Bart de Cock vào năm 2011 ([DeCock, 2011](#)), lớn hơn rất nhiều so với [tập dữ liệu giá nhà Boston](#)<sup>121</sup> nổi tiếng của Harrison và Rubinfeld (1978). Nó có nhiều mẫu và đặc trưng hơn, chứa thông tin về giá nhà ở Ames, Indiana trong khoảng thời gian từ 2006-2010.

Trong mục này, chúng tôi sẽ hướng dẫn bạn một cách chi tiết các bước tiền xử lý dữ liệu, thiết kế mô hình, lựa chọn và điều chỉnh siêu tham số. Chúng tôi mong rằng thông qua việc thực hành, bạn sẽ có thể quan sát được tác động của việc kiểm soát năng lực mô hình, trích xuất đặc trưng, v.v. trong thực tiễn. Kinh nghiệm này rất quan trọng để bạn có được trực giác của một nhà khoa học dữ liệu.

### 6.10.1 Tải và Lưu trữ Bộ dữ liệu

Trong suốt cuốn sách chúng ta sẽ cần tải và thử nghiệm nhiều mô hình trên các tập dữ liệu khác nhau. Ta sẽ lập trình một số hàm tiện ích để hỗ trợ cho việc tải dữ liệu. Đầu tiên, ta cần khởi tạo một từ điển DATA\_HUB nhằm ánh xạ một xâu ký tự đến đường dẫn (URL) với SHA-1 của tệp tại đường dẫn đó, trong đó SHA-1 dùng để xác minh tính toàn vẹn của tệp. Các tập dữ liệu này được lưu trữ trên trang DATA\_URL.

```
import os
from mxnet import gluon
import zipfile
import tarfile

# Saved in the d2l package for later use
DATA_HUB = dict()

# Saved in the d2l package for later use
DATA_URL = 'http://d2l-data.s3-accelerate.amazonaws.com/'
```

Hàm download dưới đây tải tập dữ liệu có tên name từ đường dẫn tương ứng và lưu trữ nó tại bộ nhớ cục bộ (mặc định tại `../data`). Nếu tệp trên đã tồn tại trong bộ nhớ đệm và SHA-1 của nó khớp với tệp trong DATA\_HUB, tệp trong bộ nhớ đệm sẽ được sử dụng luôn mà không cần phải tải lại. Điều này nghĩa là bạn chỉ cần tải tập dữ liệu đúng một lần khi có kết nối mạng. Hàm download trả về tên của tệp được tải xuống.

```
# Saved in the d2l package for later use
def download(name, cache_dir='../../data'):
```

(continues on next page)

<sup>120</sup> <https://www.kaggle.com/c/house-prices-advanced-regression-techniques>

<sup>121</sup> <https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names>

```
"""Download a file inserted into DATA_HUB, return the local filename."""
assert name in DATA_HUB, "%s doesn't exist" % name
url, sha1 = DATA_HUB[name]
d2l.mkdir_if_not_exist(cache_dir)
return gluon.utils.download(url, cache_dir, sha1_hash=sha1)
```

Chúng ta cũng lập trình thêm hai hàm khác: một hàm để tải và giải nén tệp zip/tar, và hàm còn lại để tải tất cả các file từ DATA\_HUB (chứa phần lớn các tập dữ liệu được sử dụng trong cuốn sách này) về bộ nhớ đệm. Bạn có thể sử dụng hàm thứ hai để tải tất cả các tập dữ liệu này trong cùng một lần tải nếu kết nối mạng của bạn chậm.

```
# Saved in the d2l package for later use
def download_extract(name, folder=None):
    """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 == '.tar' or ext == '.gz':
        fp = tarfile.open(fname, 'r')
    else:
        assert False, 'Only zip/tar files can be extracted'
    fp.extractall(base_dir)
    if folder:
        return base_dir + '/' + folder + '/'
    else:
        return data_dir + '/'

# Saved in the d2l package for later use
def download_all():
    """Download all files in the DATA_HUB"""
    for name in DATA_HUB:
        download(name)
```

### 6.10.2 Kaggle

Kaggle<sup>122</sup> là một nền tảng phổ biến cho các cuộc thi học máy. Nó kết hợp dữ liệu, mã lập trình và người dùng cho cả mục đích hợp tác và thi thố. Mặc dù việc cạnh tranh trên bảng xếp hạng nhiều khi vượt khỏi tầm kiểm soát, ta không thể không nhắc đến sự khách quan mà nền tảng mang lại từ việc so sánh định lượng một cách công bằng và trực tiếp giữa phương pháp của bạn với các phương pháp của đối thủ.Thêm vào đó, bạn còn có thể xem mã nguồn ở các lần nộp bài của (một vài) đối thủ, nghiên cứu phương pháp của họ để biết thêm các kỹ thuật mới. Nếu bạn muốn tham gia một cuộc thi, bạn cần đăng ký một tài khoản như trong Fig. 6.10.1 (hãy làm ngay đi!).

<sup>122</sup> <https://www.kaggle.com>

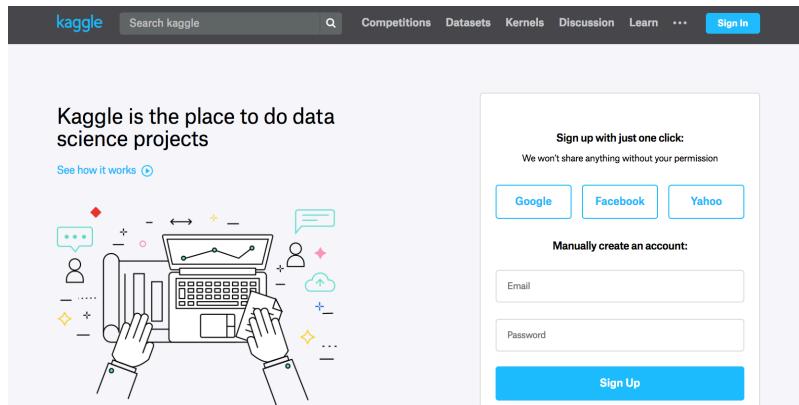


Fig. 6.10.1: Trang web Kaggle

Trên trang Dự Đoán Giá Nhà (*House Prices Prediction*) được mô tả ở Fig. 6.10.2, bạn có thể tìm được tập dữ liệu (dưới thanh “Data”), nộp kết quả dự đoán và xem thứ hạng của bạn, v.v. Đường dẫn:

<https://www.kaggle.com/c/house-prices-advanced-regression-technique>

The screenshot shows the competition page for 'House Prices: Advanced Regression Techniques'. It features a house icon with a 'SOLD' sign. The title is 'House Prices: Advanced Regression Techniques'. Below it, it says 'Predict sales prices and practice feature engineering, RFs, and gradient boosting' and '5,012 teams - Ongoing'. The navigation bar includes 'Overview' (which is active), 'Data', 'Kernels', 'Discussion', 'Leaderboard', 'Rules', 'Team', 'My Submissions', and 'Submit Predictions'. On the left, there's a sidebar with 'Overview', 'Description', 'Evaluation', 'Frequently Asked Questions', and 'Tutorials'. The main content area has a section titled 'Start here if...' with text about experience with R or Python and machine learning basics. It also includes a 'Competition Description' section.

Fig. 6.10.2: Dự đoán Giá Nhà

### 6.10.3 Truy cập và Đọc Bộ dữ liệu

Lưu ý rằng dữ liệu của cuộc thi được tách thành tập huấn luyện và tập kiểm tra. Mỗi tập dữ liệu bao gồm giá tiền của ngôi nhà và các thuộc tính liên quan bao gồm loại đường phố, năm xây dựng, kiểu mái nhà, tình trạng tầng hầm, v.v. Các đặc trưng được biểu diễn bởi nhiều kiểu dữ liệu. Ví dụ, năm xây dựng được biểu diễn bởi số nguyên, kiểu mái nhà là đặc trưng hạng mục rời rạc, còn các đặc trưng khác thì được biểu diễn bởi số thực dấu phẩy động (*floating point number*). Và đây là khi ta đổi mới với vấn đề thực tiễn: ở một vài mẫu, dữ liệu bị thiếu và được đơn thuần chú thích là ‘na’. Giá của mỗi căn nhà chỉ được cung cấp trong tập huấn luyện (sau cùng thì đây vẫn là một cuộc thi). Bạn có thể chia nhỏ tập huấn luyện để tạo tập kiểm định, tuy nhiên bạn sẽ chỉ biết được chất lượng mô hình trên tập kiểm tra chính thức khi tải kết quả dự đoán của mình lên và nhận điểm sau đó. Thanh “Data” trên cuộc thi có đường dẫn để tải bộ dữ liệu về.

Chúng ta sẽ đọc và xử lý dữ liệu với pandas, một công cụ phân tích dữ liệu hiệu quả<sup>123</sup>, vì vậy hãy đảm bảo rằng bạn đã cài đặt pandas trước khi tiếp tục. Một điều may mắn là, nếu bạn đang sử dụng Jupyter, bạn có thể cài đặt pandas mà không cần thoát khỏi notebook.

<sup>123</sup> <http://pandas.pydata.org/pandas-docs/stable/>

```
# If pandas is not installed, please uncomment the following line:  
# !pip install pandas  
  
%matplotlib inline  
from d2l import mxnet as d2l  
from mxnet import autograd, init, np, npx  
from mxnet.gluon import nn  
import pandas as pd  
npx.set_np()
```

Để thuận tiện, chúng ta sẽ tải và lưu tập dữ liệu giá nhà Kaggle từ trang web DATA\_URL. Với những cuộc thi Kaggle khác, có thể bạn sẽ phải tải dữ liệu về theo cách thủ công.

```
# Saved in the d2l package for later use  
DATA_HUB['kaggle_house_train'] = (  
    DATA_URL + 'kaggle_house_pred_train.csv',  
    '585e9cc93e70b39160e7921475f9bcd7d31219ce')  
  
# Saved in the d2l package for later use  
DATA_HUB['kaggle_house_test'] = (  
    DATA_URL + 'kaggle_house_pred_test.csv',  
    'fa19780a7b011d9b009e8bff8e99922a8ee2eb90')
```

Ta sử dụng Pandas để nạp lần lượt hai tệp csv chứa dữ liệu huấn luyện và kiểm tra.

```
train_data = pd.read_csv(download('kaggle_house_train'))  
test_data = pd.read_csv(download('kaggle_house_test'))
```

Tập huấn luyện chứa 1,460 mẫu, 80 đặc trưng, và 1 nhãn. Tập kiểm tra chứa 1,459 mẫu và 80 đặc trưng.

```
print(train_data.shape)  
print(test_data.shape)
```

Hãy cùng xem xét 4 đặc trưng đầu tiên, 2 đặc trưng cuối cùng và nhãn (giá nhà) của 4 mẫu đầu tiên:

```
print(train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]])
```

Có thể thấy với mỗi mẫu, đặc trưng đầu tiên là ID. Điều này giúp mô hình xác định được từng mẫu. Mặc dù việc này khá thuận tiện, nó không mang bất kỳ thông tin nào cho mục đích dự đoán. Do đó chúng ta sẽ lược bỏ nó ra khỏi tập dữ liệu trước khi đưa vào mạng nơ-ron.

```
all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))


```

#### 6.10.4 Tiết xử lý Dữ liệu

Như đã nói ở trên, chúng ta có rất nhiều kiểu dữ liệu. Trước khi đưa dữ liệu vào mạng học sâu, ta cần thực hiện một số phép xử lý. Hãy bắt đầu với các đặc trưng số học. Trước hết ta thay thế các giá trị còn thiếu bằng giá trị trung bình. Đây là chiến lược hợp lý nếu các đặc trưng bị thiếu một cách ngẫu nhiên. Để đưa tất cả đặc trưng số học về cùng một khoảng giá trị, ta thực hiện chuyển đổi để chúng có trung bình bằng không và phương sai đơn vị bằng cách:

$$x \leftarrow \frac{x - \mu}{\sigma}. \quad (6.10.1)$$

Để kiểm tra xem công thức trên có chuyển đổi  $x$  thành dữ liệu với trung bình bằng không hay không, ta có thể tính  $E[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$ . Để kiểm tra phương sai ta tính  $E[(x - \mu)^2] = \sigma^2$ , như vậy biến chuyển đổi sẽ có phương sai đơn vị. Lý do của việc “chuẩn hóa” dữ liệu là để đưa tất cả các đặc trưng về cùng một độ lớn. Vì sau cùng, chúng ta không thể biết trước được đặc trưng nào là đặc trưng quan trọng.

```
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 ta sẽ xử lý các giá trị rời rạc như biến ‘MSZoning’. Ta sẽ thay thế chúng bằng biểu diễn one-hot theo đúng cách mà ta đã chuyển đổi dữ liệu phân loại đa lớp thành vector chứa 0 và 1. Ví dụ, ‘MSZoning’ bao gồm các giá trị ‘RL’ và ‘RM’, tương ứng lần lượt với vector (1, 0) and (0, 1). Việc này được thực hiện một cách tự động trong pandas.

```
# Dummy_na=True refers to a missing value being a legal eigenvalue, and
# creates an indicative feature for it
all_features = pd.get_dummies(all_features, dummy_na=True)
all_features.shape
```

Bạn có thể thấy sự chuyển đổi này làm tăng số lượng các đặc trưng từ 79 lên 331. Cuối cùng, thông qua thuộc tính values, ta có thể trích xuất định dạng NumPy từ khung dữ liệu Pandas và chuyển đổi nó thành biểu diễn ndarray gốc của MXNet dành cho mục đích huấn luyện.

```
n_train = train_data.shape[0]
train_features = np.array(all_features[:n_train].values, dtype=np.float32)
test_features = np.array(all_features[n_train:].values, dtype=np.float32)
train_labels = np.array(train_data.SalePrice.values,
                       dtype=np.float32).reshape(-1, 1)
```

## 6.10.5 Huấn luyện

Để bắt đầu, ta sẽ huấn luyện một mô hình tuyến tính với hàm mất mát bình phương. Tất nhiên là mô hình tuyến tính sẽ không thể thắng cuộc thi được, nhưng nó vẫn cho ta một phép kiểm tra sơ bộ để xem dữ liệu có chứa thông tin ý nghĩa hay không. Nếu mô hình này không thể đạt chất lượng tốt hơn việc đoán mờ, khả năng cao là ta đang có lỗi trong quá trình xử lý dữ liệu. Còn nếu nó hoạt động, mô hình tuyến tính sẽ đóng vai trò như một giải pháp nền, giúp ta hình dung khoảng cách giữa một mô hình đơn giản và các mô hình tốt nhất hiện có, cũng như mức độ cải thiện mà ta mong muốn từ các mô hình “xịn” hơn.

```
loss = gluon.loss.L2Loss()

def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

Với giá nhà (hay giá cổ phiếu), ta quan tâm đến các đại lượng tương đối hơn các đại lượng tuyệt đối. Cụ thể hơn, ta thường quan tâm đến lỗi tương đối  $\frac{y - \hat{y}}{y}$  hơn lỗi tuyệt đối  $y - \hat{y}$ . Ví dụ, nếu dự đoán giá một ngôi nhà ở Rural Ohio bị lệch đi 100,000 đô-la, mà giá thông thường một ngôi nhà ở đó là 125,000 đô-la, có lẽ mô hình đang làm việc rất kém. Mặt khác, nếu ta có cùng độ lệch như vậy khi dự đoán giá nhà ở Los Altos Hills, California (giá nhà trung bình ở đây tầm hơn 4 triệu đô), có thể dự đoán này lại rất chính xác.

Một cách để giải quyết vấn đề này là tính hiệu của log giá trị dự đoán và log giá trị thật sự. Thực ra đây chính là phép đo lỗi chính thức được sử dụng trong cuộc thi để đánh giá chất lượng của các lần nộp bài. Sau cùng, một giá trị  $\delta$  bằng  $\log y - \log \hat{y}$  nhỏ đồng nghĩa với việc  $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$ . Điều này dẫn đến hàm mất mát sau:

$$L = \sqrt{\frac{1}{n} \sum_{i=1}^n (\log y_i - \log \hat{y}_i)^2}. \quad (6.10.2)$$

```
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 = np.clip(net(features), 1, float('inf'))
    return np.sqrt(2 * loss(np.log(clipped_preds), np.log(labels)).mean())
```

Khác với các mục trước, hàm huấn luyện ở đây sử dụng bộ tối ưu Adam (một biến thể của SGD mà chúng tôi sẽ mô tả cụ thể hơn sau này). Lợi thế chính của Adam so với SGD nguyên bản là: nó không quá nhạy cảm với tốc độ học ban đầu, mặc dù kết quả cũng không tốt hơn (đôi khi còn tệ hơn) SGD nếu tài nguyên để tối ưu siêu tham số là vô hạn. Bộ tối ưu này sẽ được mô tả cụ thể hơn trong [Section 13](#).

```
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
    trainer = gluon.Trainer(net.collect_params(), 'adam', {
```

(continues on next page)

```
'learning_rate': learning_rate, 'wd': weight_decay})
for epoch in range(num_epochs):
    for X, y in train_iter:
        with autograd.record():
            l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
    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
```

## 6.10.6 Kiểm định chéo gấp k-lần

Nếu bạn đang đọc cuốn sách này theo đúng thứ tự thì có thể bạn sẽ nhớ ra rằng kiểm định chéo gấp k-lần đã từng được giới thiệu khi ta thảo luận về cách lựa chọn mô hình (: numref: sec\_model\_selection). Ta sẽ ứng dụng kỹ thuật này để lựa chọn thiết kế mô hình và điều chỉnh các siêu tham số. Trước tiên ta cần một hàm trả về phần thứ  $i^{\text{th}}$  của dữ liệu trong kiểm định chéo gấp k-lần. Việc này được tiến hành bằng cách cắt chọn (*slicing*) phần thứ  $i^{\text{th}}$  để làm dữ liệu kiểm định và dùng phần còn lại làm dữ liệu huấn luyện. Cần lưu ý rằng đây không phải là cách xử lý dữ liệu hiệu quả nhất và ta chắc chắn sẽ dùng một cách khôn ngoan hơn để xử lý một tập dữ liệu có kích thước lớn hơn nhiều. Nhưng sự phức tạp được thêm vào này có thể làm rối mã nguồn một cách không cần thiết, vì vậy để đơn giản hóa vấn đề ở đây ta có thể an toàn bỏ qua.

```
def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    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 = np.concatenate((X_train, X_part), axis=0)
            y_train = np.concatenate((y_train, y_part), axis=0)
    return X_train, y_train, X_valid, y_valid
```

Trong kiểm định chéo gấp k-lần, ta sẽ huấn luyện mô hình  $k$  lần và trả về trung bình lỗi huấn luyện và trung bình lỗi kiểm định.

```
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)
```

(continues on next page)

```

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',
              legend=['train', 'valid'], yscale='log')
print('fold %d, train rmse: %f, valid rmse: %f' % (
    i, train_ls[-1], valid_ls[-1]))
return train_l_sum / k, valid_l_sum / k

```

### 6.10.7 Lựa chọn Mô hình

Trong ví dụ này, chúng tôi chọn một bộ siêu tham số chưa được tinh chỉnh và để dành việc cải thiện mô hình cho bạn đọc. Để tìm ra được một bộ siêu tham số tốt có thể sẽ tốn khá nhiều thời gian tùy thuộc vào số lượng siêu tham số mà ta muốn tối ưu. Nếu được sử dụng đúng cách, phương pháp kiểm định chéo gấp k-lần sẽ có tính ổn định cao khi thực hiện với nhiều thử nghiệm. Tuy nhiên, nếu thử quá nhiều các lựa chọn siêu tham số thì phương pháp này có thể thất bại vì ta có thể chỉ đơn thuần gặp may ở một cách chia tập kiểm định phù hợp với bộ siêu tham số đó.

```

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('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
      % (k, train_l, valid_l))

```

Bạn sẽ thấy rằng đôi khi lỗi huấn luyện cho một bộ siêu tham số có thể rất thấp, trong khi lỗi của kiểm định k-phần có thể cao hơn. Đây là dấu hiệu của sự quá khớp. Vì vậy khi ta giảm lỗi huấn luyện, ta cũng nên kiểm tra xem liệu lỗi kiểm định chéo gấp k-lần có giảm tương ứng hay không.

### 6.10.8 Dự đoán và Nộp bài

Bây giờ, khi đã biết được các lựa chọn tốt cho siêu tham số, ta có thể sử dụng toàn bộ dữ liệu cho việc huấn luyện (thay vì chỉ dùng  $1 - 1/k$  của dữ liệu như trong quá trình kiểm định chéo). Sau đó, ta áp dụng mô hình thu được lên tập kiểm tra và lưu các dự đoán vào một tệp CSV nhằm đơn giản hóa quá trình tải kết quả lên Kaggle.

```

def train_and_pred(train_features, test_feature, 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='rmse', yscale='log')
    print('train rmse %f' % train_ls[-1])
    # Apply the network to the test set
    preds = net(test_features).asnumpy()
    # Reformat it for 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)

```

Ta nên kiểm tra xem liệu các dự đoán trên tập kiểm tra có tương đồng với các dự đoán trong quá trình kiểm định chéo k-phàn hay không. Nếu đúng là như vậy thì đã đến lúc tải các dự đoán này lên Kaggle. Đoạn mã nguồn sau sẽ tạo một tệp có tên `submission.csv` (CSV là một trong những định dạng tệp được chấp nhận bởi Kaggle):

```
train_and_pred(train_features, test_features, train_labels, test_data,  
    num_epochs, lr, weight_decay, batch_size)
```

Tiếp theo, như được mô tả trong hình Fig. 6.10.3, ta có thể nộp các dự đoán lên Kaggle và so sánh chúng với giá nhà thực tế (các nhãn) trên tập kiểm tra. Các bước tiến hành khá là đơn giản:

- Đăng nhập vào trang web Kaggle và tìm đến trang của cuộc thi “House Price Prediction”.
- Nhấn vào nút “Submit Predictions” hoặc “Late Submission” (nút này nằm ở phía bên phải tại thời điểm viết sách).
- Nhấn vào nút “Upload Submission File” trong khung có viền nét đứt và chọn tệp dự đoán bạn muốn tải lên.
- Nhấn vào nút “Make Submission” ở cuối trang để xem kết quả.

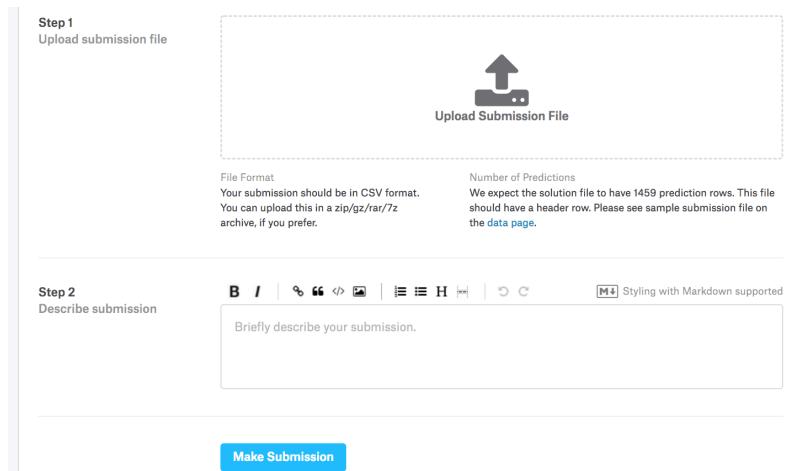


Fig. 6.10.3: Tải dữ liệu lên Kaggle

### 6.10.9 Tóm tắt

- Dữ liệu trong thực tế thường chứa nhiều kiểu dữ liệu khác nhau và cần phải được tiền xử lý.
- Chuyển đổi dữ liệu có giá trị thực để có trung bình bằng không và phương sai đơn vị là một phương án mặc định tốt. Tương tự với việc thay thế các giá trị bị thiếu bằng giá trị trung bình.
- Chuyển đổi các biến hạng mục thành các biến chỉ định cho phép chúng ta xử lý chúng như các vector.
- Ta có thể sử dụng kiểm định chéo gập k-phàn để chọn ra mô hình và điều chỉnh siêu tham số.
- Hàm Logarit có hữu ích đối với mất mát tương đối.

### 6.10.10 Bài tập

1. Nộp kết quả dự đoán của bạn từ bài hướng dẫn này cho Kaggle. Các dự đoán của bạn tốt đến đâu?
2. Bạn có thể cải thiện mô hình bằng cách giảm thiểu trực tiếp log giá nhà không? Điều gì sẽ xảy ra nếu bạn dự đoán log giá nhà thay vì giá thực?
3. Liệu việc thay thế các giá trị bị thiếu bằng trung bình của chúng luôn luôn tốt? Gợi ý: bạn có thể dựng lên một tình huống khi mà các giá trị không bị thiếu một cách ngẫu nhiên không?
4. Tìm cách biểu diễn tốt hơn để đối phó với các giá trị bị thiếu. Gợi ý: điều gì sẽ xảy ra nếu bạn thêm vào một biến chỉ định?
5. Cải thiện điểm trên Kaggle bằng cách điều chỉnh các siêu tham số thông qua kiểm định chéo gấp k-lần.
6. Cải thiện điểm bằng cách cải thiện mô hình (các tầng, điều chuẩn, dropout).
7. Điều gì sẽ xảy ra nếu ta không chuẩn hóa đặc trưng số liên tục như ta đã làm trong phần này?

### 6.10.11 Thảo luận

- Tiếng Anh<sup>124</sup>
- Tiếng Việt<sup>125</sup>

### 6.10.12 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Duy Du
- Trần Yến Thy
- Vũ Hữu Tiệp

<sup>124</sup> <https://discuss.mxnet.io/t/2346>

<sup>125</sup> <https://forum.machinelearningcoban.com/c/d2l>

## **6.11 Nhữn<sup>g</sup> người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Minh Đức



# 7 | Tính toán Học sâu

Ngoài các tập dữ liệu khổng lồ và phần cứng mạnh mẽ, không thể không nhắc tới vai trò quan trọng của các công cụ phần mềm tốt trong sự phát triển chóng mặt của học sâu. Mở đầu với thư viện tiên phong Theano được phát hành vào năm 2007, các công cụ mã nguồn mở linh hoạt đã giúp các nhà nghiên cứu nhanh chóng thử nghiệm các mô hình bằng cách tránh việc bắt người dùng phải xây dựng lại các thành phần tiêu chuẩn nhưng vẫn cho phép việc thay đổi ở bậc thấp. Theo thời gian, các thư viện học sâu ngày càng phát triển để cung cấp tính trừu tượng cao hơn. Tương tự với việc các nhà thiết kế chất bán dẫn đi từ việc chỉ rõ các lựa chọn bóng bán dẫn đến mạch logic để viết mã nguồn, các nhà nghiên cứu mạng nơ-ron sâu đã thay đổi từ việc nghĩ về hành vi của từng nơ-ron nhân tạo đơn lẻ sang việc xem xét cả một tầng trong mạng nơ-ron. Giờ đây, họ thường thiết kế các kiến trúc mạng ở mức độ trừu tượng là các *khối*.

Đến nay, chúng tôi đã giới thiệu một vài khái niệm học máy cơ bản, rồi tiến tới các mô hình học sâu. Ở chương trước, ta đã lập trình từng thành phần của một perceptron đa tầng từ đầu và biết được cách tận dụng thư viện Gluon từ MXNet để xây dựng lại mô hình một cách dễ dàng hơn. Để giúp bạn có những bước tiến xa hơn mức mong đợi, chúng tôi tập trung vào việc *sử dụng* các thư viện và không đề cập đến những chi tiết nâng cao hơn về *cách hoạt động của chúng*. Trong chương này, chúng tôi sẽ vén tấm màn bí ẩn và đào sâu vào những yếu tố chính của tính toán học sâu; cụ thể là việc xây dựng mô hình, truy cập và khởi tạo tham số, thiết kế các tầng và khối tùy chỉnh, đọc và ghi mô hình lên ổ cứng và cuối cùng là tận dụng GPU nhằm đạt được tốc độ đáng kể. Những hiểu biết này sẽ giúp bạn từ một *người dùng cuối (end user)* trở thành một *người dùng thành thạo (power user)*, cung cấp cho bạn các công cụ cần thiết để gặt hái lợi ích của một thư viện học sâu trưởng thành, đồng thời giữ được sự linh hoạt để lập trình những mô hình phức tạp hơn, bao gồm cả những mô hình mà bạn tự phát minh! Mặc dù chương này không giới thiệu bất cứ mô hình hay tập dữ liệu mới nào, các chương sau về mô hình nâng cao sẽ phụ thuộc rất nhiều vào những kỹ thuật sắp được nhắc đến.

## 7.1 Tầng và Khối

Khi lần đầu giới thiệu về các mạng nơ-ron, ta tập trung vào các mô hình tuyến tính với một đầu ra duy nhất. Như vậy toàn bộ mô hình chỉ chứa một nơ-ron. Lưu ý rằng một nơ-ron đơn lẻ (i) nhận một vài đầu vào; (ii) tạo một đầu ra (*vô hướng*) tương ứng; và (iii) có một tập các tham số liên quan có thể được cập nhật để tối ưu một hàm mục tiêu nào đó mà ta quan tâm. Sau đó, khi bắt đầu nghĩ về các mạng có nhiều đầu ra, ta tận dụng các phép tính vector để mô tả nguyên một *tầng* nơ-ron. Cũng giống như các nơ-ron riêng lẻ, các tầng (i) nhận một số đầu vào, (ii) tạo các đầu ra tương ứng, và (iii) được mô tả bằng một tập các tham số có thể điều chỉnh được. Trong hồi quy softmax, bản thân *tầng* duy nhất ấy chính là một *mô hình*. Thậm chí đối với các perceptron đa tầng, ta vẫn có thể nghĩ về chúng theo cấu trúc cơ bản này.

Điều thú vị là đối với các perceptron đa tầng, cả *mô hình* và các *tầng cấu thành* đều chia sẻ cấu trúc

này. (Toàn bộ) mô hình nhận các đầu vào thô (các đặc trưng), tạo các đầu ra (các dự đoán) và sở hữu các tham số (được tập hợp từ tất cả các tầng cấu thành). Tương tự, mỗi tầng riêng lẻ cũng nhận các đầu vào (được cung cấp bởi tầng trước đó), tính toán các đầu ra (cũng chính là các đầu vào cho tầng tiếp theo), và có một tập các tham số có thể điều chỉnh thông qua việc cập nhật dựa trên tín hiệu được truyền ngược từ tầng kế tiếp.

Dù bạn có thể nghĩ rằng các nơ-ron, các tầng và các mô hình đã cung cấp đủ sự trừu tượng để bắt tay vào làm việc, hóa ra sẽ là thuận tiện hơn khi ta bàn về các thành phần lớn hơn một tầng riêng lẻ nhưng lại nhỏ hơn toàn bộ mô hình. Ví dụ, kiến trúc ResNet-152, rất phổ biến trong thị giác máy tính, sở hữu hàng trăm tầng. Nó bao gồm các khuôn mẫu *nhóm tầng* được lắp lại nhiều lần. Việc lập trình từng tầng của một mạng như vậy có thể trở nên tẻ nhạt. Mỗi quan tâm này không chỉ là trên lý thuyết — các khuôn mẫu thiết kế như vậy rất phổ biến trong thực tế. Kiến trúc ResNet được đề cập ở trên đã giành chiến thắng trong hai cuộc thi thị giác máy tính ImageNet và COCO năm 2015, trong cả bài toán nhận dạng và bài toán phát hiện (He et al., 2016a) và vẫn là một kiến trúc được tin dùng cho nhiều bài toán thị giác. Các kiến trúc tương tự, trong đó các tầng được sắp xếp thành những khuôn mẫu lắp lại, hiện đã trở nên thông dụng ở nhiều lĩnh vực khác, bao gồm cả xử lý ngôn ngữ tự nhiên và xử lý tiếng nói.

Để lập trình các mạng phức tạp này, ta sẽ giới thiệu khái niệm *khối* trong mạng nơ-ron. Một khối có thể mô tả một tầng duy nhất, một mảng đa tầng hoặc toàn bộ một mô hình! Dưới góc nhìn xây dựng phần mềm, một Block (Khối) là một *lớp*. Bất kỳ một lớp con nào của Block đều phải định nghĩa phương thức forward để chuyển hóa đầu vào thành đầu ra và phải lưu trữ mọi tham số cần thiết. Lưu ý rằng có một vài Block sẽ không yêu cầu chứa bất kỳ tham số nào cả! Ngoài ra, một Block phải sở hữu một phương thức backward cho mục đích tính toán gradient. May mắn thay, nhờ sự trợ giúp đắc lực của gói autograd (được giới thiệu trong Section 4) nên khi định nghĩa Block, ta chỉ cần quan tâm đến các tham số và hàm forward.

Một lợi ích khi làm việc ở mức độ trừu tượng Block đó là ta có thể kết hợp chúng, thường là theo phương pháp đệ quy, để tạo ra các thành phần lớn hơn (xem hình minh họa trong Fig. 7.1.1).

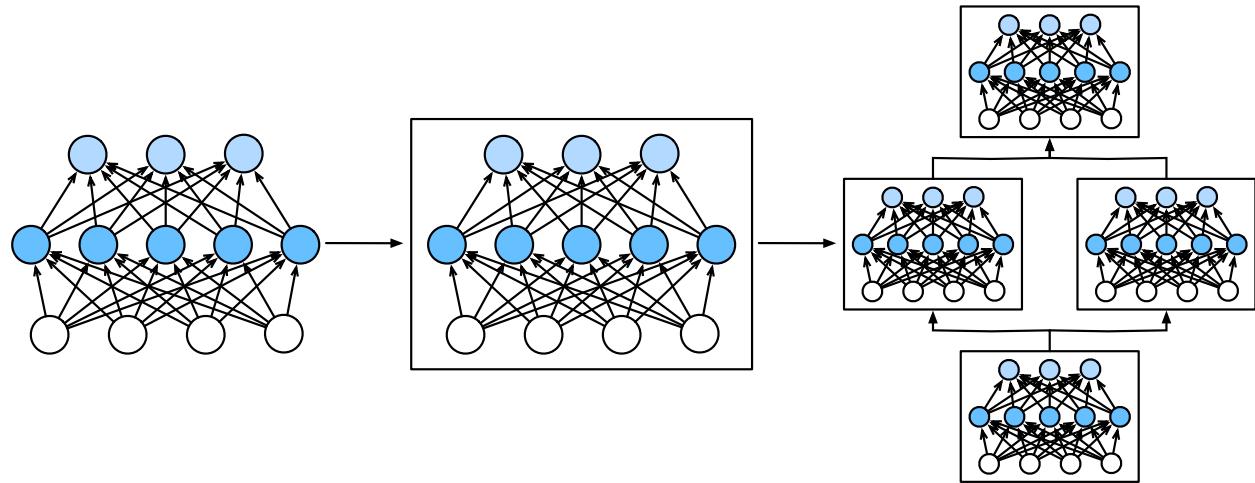


Fig. 7.1.1: Nhiều tầng được kết hợp để tạo thành các khối

Bằng cách định nghĩa các khối với độ phức tạp tùy ý, các mạng nơ-ron phức tạp có thể được lập trình với mã nguồn ngắn gọn một cách đáng ngạc nhiên.

Để bắt đầu, ta sẽ xem lại các khối mà ta đã sử dụng để lập trình perceptron đa tầng (Section 6.3). Đoạn mã nguồn sau tạo ra một mạng gồm một tầng ẩn kết nối đầy đủ với 256 nút sử dụng hàm kích hoạt ReLU, theo sau là một *tầng đầu ra* kết nối đầy đủ với 10 nút (không có hàm kích hoạt).

```

from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

x = np.random.uniform(size=(2, 20))

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)

```

Trong ví dụ này, ta đã xây dựng mô hình bằng cách khởi tạo một đối tượng `nn.Sequential` và gán vào biến `net`. Sau đó, ta gọi phương thức `add` nhiều lần để nối các tầng theo thứ tự mà chúng sẽ được thực thi. Nói một cách ngắn gọn, `nn.Sequential` định nghĩa một loại Block đặc biệt có nhiệm vụ duy trì một danh sách chứa các Block cấu thành được sắp xếp theo thứ tự nhất định. Phương thức `add` chỉ đơn giản hỗ trợ việc thêm liên tiếp từng Block vào trong danh sách đó. Lưu ý rằng mỗi tầng là một thực thể của lớp `Dense`, và bản thân lớp `Dense` lại là một lớp con của `Block`. Hàm `forward` cũng rất đơn giản: nó xâu chuỗi từng Block trong danh sách lại với nhau, chuyển đầu ra của từng khối thành đầu vào cho khối tiếp theo. Lưu ý rằng cho đến giờ, ta đã gọi mô hình thông qua `net(X)` để thu được đầu ra. Thực ra đây chỉ là một cách viết tắt của `net.forward(X)`, một thủ thuật Python khéo léo đạt được thông qua hàm `__call__` của lớp `Block`.

### 7.1.1 Một Khối Tùy chỉnh

Có lẽ cách dễ nhất để hiểu rõ hơn `nn.Block` hoạt động như thế nào là tự lập trình nó. Trước khi tự lập trình một Block tùy chỉnh, hãy cùng tóm tắt ngắn gọn các chức năng cơ bản mà một Block phải cung cấp:

1. Phương thức `forward` nhận đối số là dữ liệu đầu vào.
2. Phương thức `forward` trả về một giá trị đầu ra. Lưu ý rằng đầu ra có thể có kích thước khác với đầu vào. Ví dụ, tầng `Dense` đầu tiên trong mô hình phía trên nhận đầu vào có kích thước tùy ý nhưng trả về đầu ra có kích thước 256.
3. Tính gradient của đầu ra theo đầu vào bằng phương thức `backward`, thường thì việc này được thực hiện tự động.
4. Lưu trữ và cung cấp quyền truy cập tới các tham số cần thiết để tiến hành phương thức tính toán `forward`.
5. Khởi tạo các tham số này khi cần thiết.

Trong đoạn mã dưới đây, chúng ta lập trình từ đầu một Block (Khối) tương đương với một perceptron đa tầng chỉ có một tầng ẩn và 256 nút ẩn, cùng một tầng đầu ra 10 chiều. Lưu ý rằng lớp MLP bên dưới đây kế thừa từ lớp `Block`. Ta sẽ phụ thuộc nhiều vào các phương thức của lớp cha, và chỉ tự viết phương thức `__init__` và `forward`.

```

from mxnet.gluon import nn

class MLP(nn.Block):
    # Declare a layer with model parameters. Here, we declare two fully
    # connected layers

```

(continues on next page)

```

def __init__(self, **kwargs):
    # Call the constructor of the MLP parent class Block to perform the
    # necessary initialization. In this way, other function parameters can
    # also be specified when constructing an instance, such as the model
    # parameter, params, described in the following sections
    super(MLP, self).__init__(**kwargs)
    self.hidden = nn.Dense(256, activation='relu') # Hidden layer
    self.output = nn.Dense(10) # Output layer

    # Define the forward computation of the model, that is, how to return the
    # required model output based on the input x
def forward(self, x):
    return self.output(self.hidden(x))

```

Để bắt đầu, ta sẽ tập trung vào phương thức forward. Lưu ý rằng nó nhận giá trị đầu vào x, tính toán tầng biểu diễn ẩn (self.hidden(x)) và trả về các giá trị logit (self.output( ... )). Ở cách lập trình MLP này, cả hai tầng trên đều là biến thực thể (*instance variables*). Để thấy tại sao điều này có lý, tưởng tượng ta khởi tạo hai MLP, net1 và net2, và huấn luyện chúng với dữ liệu khác nhau. Dĩ nhiên là ta mong đợi chúng đại diện cho hai mô hình học khác nhau.

Ta khởi tạo các tầng của MLP trong phương thức \_\_init\_\_ (hàm khởi tạo) và sau đó gọi các tầng này mỗi khi ta gọi phương thức forward. Hãy chú ý một vài chi tiết quan trọng. Đầu tiên, phương thức \_\_init\_\_ tùy chỉnh của ta gọi phương thức \_\_init\_\_ của lớp cha thông qua super(MLP, self). \_\_init\_\_(\*\*kwargs) để tránh việc viết lại cùng một phần mã nguồn áp dụng cho hầu hết các khối. Chúng ta sau đó khởi tạo hai tầng Dense, gán chúng lần lượt là self.hidden và self.output. Chú ý rằng trừ khi đang phát triển một toán tử mới, chúng ta không cần lo lắng về lan truyền ngược (phương thức backward) hoặc khởi tạo tham số (phương thức initialize). Gluon sẽ tự động khởi tạo các phương thức đó. Hãy cùng thử nghiệm điều này:

```

net = MLP()
net.initialize()
net(x)

```

Một ưu điểm chính của phép trừu tượng hóa Block là tính linh hoạt của nó. Ta có thể kế thừa từ lớp Block để tạo các tầng (chẳng hạn như lớp Dense được cung cấp bởi Gluon), toàn bộ cả mô hình (như MLP ở phía trên) hoặc các thành phần đa dạng với độ phức tạp vừa phải. Ta sẽ tận dụng tính linh hoạt này xuyên suốt ở các chương sau, đặc biệt khi làm việc với các mạng nơ-ron tích chập.

### 7.1.2 Khối Tuần tự

Bây giờ ta có thể có cái nhìn rõ hơn về cách mà lớp Sequential (Tuần tự) hoạt động. Nhắc lại rằng Sequential được thiết kế để xâu chuỗi các Khối lại với nhau. Để tự xây dựng một lớp MySequential đơn giản, ta chỉ cần định nghĩa hai phương thức chính sau: 1. Phương thức add nhằm đẩy từng Block một vào trong danh sách. 2. Phương thức forward nhằm truyền một đầu vào qua chuỗi các Blocks (theo thứ tự mà chúng được nối).

Lớp MySequential dưới đây cung cấp tính năng giống như lớp Sequential mặc định của Gluon:

```

class MySequential(nn.Block):
    def add(self, block):
        # Here, block is an instance of a Block subclass, and we assume it has

```

(continues on next page)

```

# a unique name. We save it in the member variable _children of the
# Block class, and its type is OrderedDict. When the MySequential
# instance calls the initialize function, the system automatically
# initializes all members of _children
self._children[block.name] = block

def forward(self, x):
    # OrderedDict guarantees that members will be traversed in the order
    # they were added
    for block in self._children.values():
        x = block(x)
    return x

```

Phương thức add thêm một Block đơn vào từ điển có thứ tự `_children`. Bạn có thể thắc mắc tại sao mỗi Block của Gluon sở hữu một thuộc tính `_children` và tại sao ta sử dụng nó thay vì tự tạo một danh sách Python. Thật ra, ưu điểm chính của `_children` là trong quá trình khởi tạo trọng số ban đầu của các khối, Gluon sẽ tự động tìm các khối con có trọng số cần được khởi tạo trong từ điển này.

Khi phương thức forward của khối MySequential được gọi, các Block sẽ được thực thi theo thứ tự mà chúng được thêm vào. Bây giờ ta có thể lập trình lại một MLP sử dụng lớp MySequential.

```

net = MySequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize()
net(x)

```

Chú ý rằng việc sử dụng MySequential giống hệt với đoạn mã mà ta đã viết trước đó cho lớp Sequential của Gluon (được mô tả trong Section 6.3).

### 7.1.3 Thực thi Mã trong Phương thức forward

Lớp nn.Sequential giúp việc xây dựng mô hình trở nên dễ hơn, cho phép ta xây dựng các kiến trúc mới mà không cần phải tự định nghĩa một lớp riêng. Tuy nhiên, không phải tất cả mô hình đều có cấu trúc chuỗi xích đơn giản. Khi cần phải linh hoạt hơn, ta vẫn sẽ muốn định nghĩa từng Block theo cách của mình, ví dụ như khi muốn sử dụng luồng điều khiển Python trong lượt truyền xuôi. Hơn nữa, ta cũng có thể muốn thực hiện các phép toán tùy ý thay vì chỉ dựa vào các tầng mạng nơ-ron được định nghĩa từ trước.

Độc giả có thể nhận ra rằng tất cả phép toán trong mạng cho tới giờ đều thao tác trên các giá trị kích hoạt và tham số của mạng. Tuy nhiên, trong một vài trường hợp, ta có thể muốn kết hợp thêm các hằng số. Chúng không phải là kết quả của tầng trước mà cũng không phải là tham số có thể cập nhật được. Trong Gluon, ta gọi chúng là tham số *không đổi* (*constant parameter*). Ví dụ ta muốn một tầng tính hàm  $f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}$ , trong đó  $\mathbf{x}, \mathbf{w}$  là tham số, và  $c$  là một hằng số cho trước được giữ nguyên giá trị trong suốt quá trình tối ưu hóa.

Khai báo các hằng số một cách tường minh (bằng `get_constant`) giúp Gluon tăng tốc độ thực thi. Trong đoạn mã sau, ta lập trình một mô hình mà không hề dễ lắp ráp nếu sử dụng Sequential và các tầng được định nghĩa trước.

```

class FixedHiddenMLP(nn.Block):
    def __init__(self, **kwargs):
        super(FixedHiddenMLP, self).__init__(**kwargs)
        # Random weight parameters created with the get_constant are not
        # iterated during training (i.e., constant parameters)
        self.rand_weight = self.params.get_constant(
            'rand_weight', np.random.uniform(size=(20, 20)))
        self.dense = nn.Dense(20, activation='relu')

    def forward(self, x):
        x = self.dense(x)
        # Use the constant parameters created, as well as the relu
        # and dot functions
        x = npx.relu(np.dot(x, self.rand_weight.data()) + 1)
        # Reuse the fully connected layer. This is equivalent to sharing
        # parameters with two fully connected layers
        x = self.dense(x)
        # Here in Control flow, we need to call asscalar to return the scalar
        # for comparison
        while np.abs(x).sum() > 1:
            x /= 2
        return x.sum()

```

Trong mô hình FixedHiddenMLP, ta lập trình một tầng ẩn có trọng số (**self.rand\_weight**) được khởi tạo ngẫu nhiên và giữ nguyên giá trị về sau. Trọng số này không phải là một tham số mô hình, vì vậy nó không được cập nhật khi sử dụng lan truyền ngược. Sau đó, đầu ra của tầng cố định này được đưa vào tầng Dense.

Lưu ý rằng trước khi trả về giá trị đầu ra, mô hình của ta đã làm điều gì đó bất thường. Ta đã chạy một vòng lặp while, lấy vector đầu ra chia cho 2 cho đến khi nó thỏa mãn điều kiện `np.abs(x).sum() > 1`. Cuối cùng, ta gán giá trị đầu ra bằng tổng các phần tử trong x. Theo sự hiểu biết của chúng tôi, không có mạng nơ-ron tiêu chuẩn nào thực hiện phép toán này. Lưu ý rằng phép toán đặc biệt này có thể không hữu ích gì trong các công việc ngoài thực tế. Mục đích của chúng tôi ở đây là chỉ cho độc giả thấy được cách tích hợp một đoạn mã tùy ý vào luồng tính toán của mạng nơ-ron.

```

net = FixedHiddenMLP()
net.initialize()
net(x)

```

Với Gluon, ta có thể kết hợp nhiều cách khác nhau để lắp ráp các Block lại. Trong ví dụ dưới đây, ta lồng các Block với nhau theo nhiều cách sáng tạo.

```

class NestMLP(nn.Block):
    def __init__(self, **kwargs):
        super(NestMLP, self).__init__(**kwargs)
        self.net = nn.Sequential()
        self.net.add(nn.Dense(64, activation='relu'),
                    nn.Dense(32, activation='relu'))
        self.dense = nn.Dense(16, activation='relu')

    def forward(self, x):
        return self.dense(self.net(x))

```

(continues on next page)

```

chimera = nn.Sequential()
chimera.add(NestMLP(), nn.Dense(20), FixedHiddenMLP())

chimera.initialize()
chimera(x)

```

### 7.1.4 Biên dịch Mã nguồn

Những người đọc có tâm có thể sẽ bắt đầu lo lắng về hiệu năng của một vài đoạn mã trên. Sau cùng thì, chúng ta có rất nhiều thao tác truy cập từ điển, thực thi mã lập trình và rất nhiều thứ “đậm chất Python” khác xuất hiện trong thứ mà lẽ ra nên là một thư viện học sâu hiệu năng cao. Vấn đề của **Khóa Trình thông dịch Toàn cục (Global Interpreter Lock)<sup>126</sup>** trong Python khá phổ biến. Trong bối cảnh học sâu, ta lo sợ rằng GPU cực kỳ nhanh của ta có thể sẽ phải đợi CPU “rùa bò” chạy xong những dòng lệnh Python trước khi nó có thể nhận tác vụ chạy tiếp theo. Cách tốt nhất để tăng tốc Python là tránh không sử dụng nó. Gluon làm việc này bằng cách cho phép việc Hybrid hóa ([Section 14.1](#)). Ở đây, trình thông dịch của Python sẽ thực thi một Khối trong lần chạy đầu tiên.

Môi trường chạy của Gluon sẽ ghi lại những gì đang diễn ra và trong lần chạy tiếp theo, nó sẽ thực hiện các tác vụ gọi trong Python một cách vắn tắt hơn. Điều này có thể giúp tăng tốc độ chạy đáng kể trong một vài trường hợp, tuy nhiên, ta cần quan tâm tới việc luồng điều khiển (như trên) sẽ dẫn đến những nhánh khác nhau với mỗi lần truyền qua mạng. Chúng tôi khuyến khích những độc giả có hứng thú sau khi hoàn tất chương này hãy đọc thêm mục hybrid hóa ([Section 14.1](#)) để tìm hiểu về quá trình biên dịch.

### 7.1.5 Tóm tắt

- Các tầng trong mạng nơ-ron là các Khối.
- Nhiều tầng có thể cấu thành một Khối.
- Nhiều Khối có thể cấu thành một Khối.
- Một Khối có thể chứa các đoạn mã nguồn.
- Các Khối đảm nhiệm nhiều tác vụ bao gồm khởi tạo tham số và lan truyền ngược.
- Việc gắn kết các tầng và khối một cách tuần tự được đảm nhiệm bởi Khối Sequential.

### 7.1.6 Bài tập

1. Vấn đề gì sẽ xảy ra nếu ta bỏ hàm `asscalar` trong lớp `FixedHiddenMLP`?
2. Vấn đề gì sẽ xảy ra nếu `self.net` được định nghĩa trong thực thể `Sequential` ở lớp `NestMLP` được đổi thành `self.net = [nn.Dense(64, activation='relu'), nn.Dense(32, activation='relu')]`?
3. Hãy lập trình một khối nhận đối số là hai khối khác, ví dụ như `net1` và `net2`, và trả về kết quả là phép nối các giá trị đầu ra của cả hai mạng đó khi thực hiện lượt truyền xuôi.

<sup>126</sup> <https://wiki.python.org/moin/GlobalInterpreterLock>

4. Giả sử bạn muốn nối nhiều thực thể của cùng một mạng với nhau. Hãy lập trình một hàm để tạo ra nhiều thực thể của cùng một mạng và dùng chúng để tạo thành một mạng lớn hơn (các hàm này trong thiết kế phần mềm được gọi là Factory Function).

### 7.1.7 Thảo luận

- Tiếng Anh<sup>127</sup>
- Tiếng Việt<sup>128</sup>

### 7.1.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Trần Yến Thy
- Phạm Hồng Vinh
- Lý Phi Long

## 7.2 Quản lý Tham số

Một khi ta đã chọn được kiến trúc mạng và các giá trị siêu tham số, ta sẽ bắt đầu với vòng lặp huấn luyện với mục tiêu là tìm các giá trị tham số để cực tiểu hóa hàm mục tiêu. Sau khi huấn luyện xong, ta sẽ cần các tham số đó để đưa ra dự đoán trong tương lai. Hơn nữa, thi thoảng ta sẽ muốn trích xuất tham số để sử dụng lại trong một hoàn cảnh khác, có thể lưu trữ mô hình để thực thi trong một phần mềm khác hoặc để rút ra hiểu biết khoa học bằng việc phân tích mô hình.

Thông thường, ta có thể bỏ qua những chi tiết chuyên sâu về việc khai báo và xử lý tham số bởi Gluon sẽ đảm nhiệm công việc nặng nhọc này. Tuy nhiên, khi ta bắt đầu tiến xa hơn những kiến trúc chỉ gồm các tầng tiêu chuẩn được xếp chồng lên nhau, đôi khi ta sẽ phải tự đi sâu vào việc khai báo và xử lý tham số. Trong mục này, chúng tôi sẽ đề cập đến những việc sau:

- Truy cập các tham số để gỡ lỗi, chẩn đoán mô hình và biểu diễn trực quan.
- Khởi tạo tham số.
- Chia sẻ tham số giữa các thành phần khác nhau của mô hình.

Chúng ta sẽ bắt đầu từ mạng Perceptron đa tầng với một tầng ẩn.

<sup>127</sup> <https://discuss.mxnet.io/t/2325>

<sup>128</sup> <https://forum.machinelearningcoban.com/c/d21>

```

from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'))
net.add(nn.Dense(10))
net.initialize() # Use the default initialization method

x = np.random.uniform(size=(2, 20))
net(x) # Forward computation

```

### 7.2.1 Truy cập Tham số

Hãy bắt đầu với việc truy cập tham số của những mô hình mà bạn đã biết. Khi một mô hình được định nghĩa bằng lớp Tuần tự (*Sequential*), ta có thể truy cập bất kỳ tầng nào bằng chỉ số, như thể nó là một danh sách. Thuộc tính `params` của mỗi tầng chứa tham số của chúng. Ta có thể quan sát các tham số của mạng `net` định nghĩa ở trên.

```

print(net[0].params)
print(net[1].params)

```

Kết quả của đoạn mã này cho ta một vài thông tin quan trọng. Đầu tiên, mỗi tầng kết nối đầy đủ đều có hai tập tham số, ví dụ như `dense0_weight` và `dense0_bias` tương ứng với trọng số và hệ số điều chỉnh của tầng đó. Chúng đều được lưu trữ ở dạng số thực dấu phẩy động độ chính xác đơn. Lưu ý rằng tên của các tham số cho phép ta xác định tham số của từng tầng *một cách độc nhất*, kể cả khi mạng nơ-ron chứa hàng trăm tầng.

#### Các tham số Mục tiêu

Lưu ý rằng mỗi tham số được biểu diễn bằng một thực thể của lớp `Parameter`. Để làm việc với các tham số, trước hết ta phải truy cập được các giá trị số của chúng. Có một vài cách để làm việc này, một số cách đơn giản hơn trong khi các cách khác lại tổng quát hơn. Để bắt đầu, ta có thể truy cập tham số của một tầng thông qua thuộc tính `bias` hoặc `weight` rồi sau đó truy cập giá trị số của chúng thông qua phương thức `data()`. Đoạn mã sau trích xuất hệ số điều chỉnh của tầng thứ hai trong mạng nơ-ron.

```

print(net[1].bias)
print(net[1].bias.data())

```

Tham số là các đối tượng khá phức tạp bởi chúng chứa dữ liệu, gradient và một vài thông tin khác. Đó là lý do tại sao ta cần yêu cầu dữ liệu một cách tường minh. Lưu ý rằng vector hệ số điều chỉnh chứa các giá trị không vì ta chưa hề cập nhật mô hình kể từ khi nó được khởi tạo. Ta cũng có thể truy cập các tham số theo tên của chúng, chẳng hạn như `dense0_weight` ở dưới. Điều này khả thi vì thực ra mỗi tầng đều chứa một từ điển tham số.

```

print(net[0].params['dense0_weight'])
print(net[0].params['dense0_weight'].data())

```

Chú ý rằng khác với hệ số điều chỉnh, trọng số chứa các giá trị khác không bởi chúng được khởi tạo ngẫu nhiên. Ngoài data, mỗi Parameter còn cung cấp phương thức grad() để truy cập gradient. Gradient sẽ có cùng kích thước với trọng số. Vì ta chưa thực hiện lan truyền ngược với mạng nơ-ron này, các giá trị của gradient đều là 0.

```
net[0].weight.grad()
```

## Tất cả các Tham số cùng lúc

Khi ta cần phải thực hiện các phép toán với tất cả tham số, việc truy cập lần lượt từng tham số sẽ trở nên khá khó chịu. Việc này sẽ càng chậm chạp khi ta làm việc với các khối phức tạp hơn, ví dụ như các khối lồng nhau vì lúc đó ta sẽ phải duyệt toàn bộ cây bằng đệ quy để có thể trích xuất tham số của từng khối con. Để tránh vấn đề này, mỗi khối có thêm một phương thức collect\_params để trả về một từ điển duy nhất chứa tất cả tham số. Ta có thể gọi collect\_params với một tầng duy nhất hoặc với toàn bộ mạng nơ-ron như sau:

```
# parameters only for the first layer
print(net[0].collect_params())
# parameters of the entire network
print(net.collect_params())
```

Từ đó, ta có cách thứ ba để truy cập các tham số của mạng:

```
net.collect_params()['dense1_bias'].data()
```

Xuyên suốt cuốn sách này ta sẽ thấy các khối đặt tên cho khối con theo nhiều cách khác nhau. Khối Sequential chỉ đơn thuần đánh số chúng. Ta có thể tận dụng quy ước định danh này cùng với một tính năng thông minh của collect\_params để lọc ra các tham số được trả về bằng các biểu thức chính quy (*regular expression*).

```
print(net.collect_params('.*weight'))
print(net.collect_params('dense0.*'))
```

## Thu thập Tham số từ các Khối lồng nhau

Hãy cùng xem cách hoạt động của các quy ước định danh tham số khi ta lồng nhiều khối vào nhau. Trước hết ta định nghĩa một hàm tạo khối (có thể gọi là một nhà máy khối) và rồi kết hợp chúng trong các khối lớn hơn.

```
def block1():
    net = nn.Sequential()
    net.add(nn.Dense(32, activation='relu'))
    net.add(nn.Dense(16, activation='relu'))
    return net

def block2():
    net = nn.Sequential()
    for i in range(4):
        net.add(block1())
```

(continues on next page)

```

return net

rgnet = nn.Sequential()
rgnet.add(block2())
rgnet.add(nn.Dense(10))
rgnet.initialize()
rgnet(x)

```

Bây giờ ta đã xong phần thiết kế mạng, hãy cùng xem cách nó được tổ chức. Hãy để ý ở dưới rằng dù hàm `collect_params()` trả về một danh sách các tham số được định danh, việc gọi `collect_params` như một thuộc tính sẽ cho ta biết cấu trúc của mạng.

```

print(rgnet.collect_params)
print(rgnet.collect_params())

```

Bởi vì các tầng được lồng vào nhau theo cơ chế phân cấp, ta cũng có thể truy cập chúng tương tự như cách ta dùng chỉ số để truy cập các danh sách lồng nhau. Chẳng hạn, ta có thể truy cập khối chính đầu tiên, khối con thứ hai bên trong nó và hệ số điều chỉnh của tầng đầu tiên bên trong nữa như sau:

```
rgnet[0][1][0].bias.data()
```

## 7.2.2 Khởi tạo Tham số

Bây giờ khi đã biết cách truy cập tham số, hãy cùng xem xét việc khởi tạo chúng đúng cách. Ta đã thảo luận về sự cần thiết của việc khởi tạo tham số trong [Section 6.8](#). Theo mặc định, MXNet khởi tạo các ma trận trọng số bằng cách lấy mẫu từ phân phối đều  $U[-0, 07, 0, 07]$  và đặt tất cả các hệ số điều chỉnh bằng 0. Tuy nhiên, thường ta sẽ muốn khởi tạo trọng số theo nhiều phương pháp khác. Mô-đun `init` của MXNet cung cấp sẵn nhiều phương thức khởi tạo. Nếu ta muốn một bộ khởi tạo tùy chỉnh, ta sẽ cần làm thêm một chút việc.

### Phương thức Khởi tạo có sẵn

Ta sẽ bắt đầu với việc gọi các bộ khởi tạo có sẵn. Đoạn mã dưới đây khởi tạo tất cả các tham số với các biến ngẫu nhiên Gauss có độ lệch chuẩn bằng 0.01.

```

# force_reinit ensures that variables are freshly initialized
# even if they were already initialized previously
net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
net[0].weight.data()[0]

```

Ta cũng có thể khởi tạo tất cả tham số với một hằng số (ví dụ như 1) bằng cách sử dụng bộ khởi tạo `Constant(1)`.

```

net.initialize(init=init.Constant(1), force_reinit=True)
net[0].weight.data()[0]

```

Ta còn có thể áp dụng các bộ khởi tạo khác nhau cho các khối khác nhau. Ví dụ, trong đoạn mã nguồn bên dưới, ta khởi tạo tầng đầu tiên bằng cách sử dụng bộ khởi tạo Xavier và khởi tạo tầng thứ hai với một hằng số là 42.

```
net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
net[1].initialize(init=init.Constant(42), force_reinit=True)
print(net[1].weight.data()[0, 0])
```

### Phương thức Khởi tạo Tùy chỉnh

Đôi khi, các phương thức khởi tạo mà ta cần không có sẵn trong mô-đun init. Trong trường hợp đó, ta có thể khai báo một lớp con của lớp Initializer. Thông thường, ta chỉ cần lập trình hàm `_init_weight` để nhận một đối số ndarray (data) và gán giá trị khởi tạo mong muốn cho nó. Trong ví dụ bên dưới, ta sẽ khai báo một bộ khởi tạo cho phân phối kì lạ sau:

$$w \sim \begin{cases} U[5, 10] & \text{với xác suất } \frac{1}{4} \\ 0 & \text{với xác suất } \frac{1}{2} \\ U[-10, -5] & \text{với xác suất } \frac{1}{4} \end{cases} \quad (7.2.1)$$

```
class MyInit(init.Initializer):
    def __init_weight(self, name, data):
        print('Init', name, data.shape)
        data[:] = np.random.uniform(-10, 10, data.shape)
        data *= np.abs(data) >= 5

net.initialize(MyInit(), force_reinit=True)
net[0].weight.data()[0]
```

Lưu ý rằng ta luôn có thể trực tiếp đặt giá trị cho tham số bằng cách gọi hàm `data()` để truy cập ndarray của tham số đó. Một lưu ý khác cho người dùng nâng cao: nếu muốn điều chỉnh các tham số trong phạm vi của autograd, bạn cần sử dụng hàm `set_data` để tránh làm rối loạn cơ chế tính vi phân tự động.

```
net[0].weight.data()[:] += 1
net[0].weight.data()[0, 0] = 42
net[0].weight.data()[0]
```

### 7.2.3 Các Tham số bị Trói buộc

Thông thường, ta sẽ muốn chia sẻ các tham số mô hình cho nhiều tầng. Sau này ta sẽ thấy trong quá trình huấn luyện embedding từ, việc sử dụng cùng một bộ tham số để mã hóa và giải mã các từ có thể khá hợp lý. Ta đã thảo luận về một trường hợp như vậy trong Section 7.1. Hãy cùng xem làm thế nào để thực hiện việc này một cách tinh tế hơn. Sau đây ta sẽ tạo một tầng kết nối đầy đủ và sử dụng chính tham số của nó làm tham số cho một tầng khác.

```
net = nn.Sequential()
# We need to give the shared layer a name such that we can reference its
# parameters
```

(continues on next page)

```

shared = nn.Dense(8, activation='relu')
net.add(nn.Dense(8, activation='relu'),
        shared,
        nn.Dense(8, activation='relu', params=shared.params),
        nn.Dense(10))
net.initialize()

x = np.random.uniform(size=(2, 20))
net(x)

# Check whether the parameters are the same
print(net[1].weight.data()[0] == net[2].weight.data()[0])
net[1].weight.data()[0, 0] = 100
# Make sure that they are actually the same object rather than just having the
# same value
print(net[1].weight.data()[0] == net[2].weight.data()[0])

```

Ví dụ này cho thấy các tham số của tầng thứ hai và thứ ba đã bị trói buộc với nhau. Chúng không chỉ có giá trị bằng nhau, chúng còn được biểu diễn bởi cùng một ndarray. Vì vậy, nếu ta thay đổi các tham số của tầng này này thì các tham số của tầng kia cũng sẽ thay đổi theo. Bạn có thể tự hỏi rằng *chuyện gì sẽ xảy ra với gradient khi các tham số bị trói buộc?*. Vì các tham số mô hình chứa gradient nên gradient của tầng ẩn thứ hai và tầng ẩn thứ ba được cộng lại tại `shared.params.grad()` trong quá trình lan truyền ngược.

#### 7.2.4 Tóm tắt

- Ta có vài cách để truy cập, khởi tạo và trói buộc các tham số mô hình.
- Ta có thể sử dụng các phương thức khởi tạo tùy chỉnh.
- Gluon có một cơ chế tinh vi để truy cập các tham số theo phân cấp một cách độc nhất.

#### 7.2.5 Bài tập

1. Sử dụng FancyMLP được định nghĩa trong Section 7.1 và truy cập tham số của các tầng khác nhau.
2. Xem tài liệu MXNet<sup>129</sup> và nghiên cứu các bộ khởi tạo khác nhau.
3. Thử truy cập các tham số mô hình sau khi gọi `net.initialize()` và trước khi gọi `net(x)` và quan sát kích thước của chúng. Điều gì đã thay đổi? Tại sao?
4. Xây dựng và huấn luyện một perceptron đa tầng mà trong đó có một tầng sử dụng tham số được chia sẻ. Trong quá trình huấn luyện, hãy quan sát các tham số mô hình và gradient của từng tầng.
5. Tại sao việc chia sẻ tham số lại là một ý tưởng hay?

<sup>129</sup> <http://beta.mxnet.io/api/gluon-related/mxnet.initializer.html>

## 7.2.6 Thảo luận

- Tiếng Anh<sup>130</sup>
- Tiếng Việt<sup>131</sup>

## 7.2.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Lê Cao Thăng
- Nguyễn Duy Du
- Phạm Hồng Vinh
- Phạm Minh Đức

## 7.3 Khởi tạo trẽ

Cho tới nay, có vẻ như ta chưa phải chịu hậu quả của việc thiết lập mạng cầu thẳ. Cụ thể, ta đã “giả mù” và làm những điều không trực quan sau:

- Ta định nghĩa kiến trúc mạng mà không xét đến chiều đầu vào.
- Ta thêm các tầng mà không xét đến chiều đầu ra của tầng trước đó.
- Ta thậm chí còn “khởi tạo” các tham số mà không có đầy đủ thông tin để xác định số lượng các tham số của mô hình.

Bạn có thể khá bất ngờ khi thấy mã nguồn của ta vẫn chạy. Suy cho cùng, MXNet (hay bất cứ framework nào khác) không thể dự đoán được chiều của đầu vào. Thủ thuật ở đây đó là MXNet đã “khởi tạo trẽ”, tức đợi cho đến khi ta truyền dữ liệu qua mô hình lần đầu để suy ra kích thước của mỗi tầng khi chúng “di chuyển”.

Ở các chương sau, khi làm việc với các mạng nơ-ron tích chập, kỹ thuật này sẽ còn trở nên tiện lợi hơn, bởi chiều của đầu vào (tức độ phân giải của một bức ảnh) sẽ tác động đến chiều của các tầng tiếp theo trong mạng. Do đó, khả năng gán giá trị các tham số mà không cần biết số chiều tại thời điểm viết mã có thể đơn giản hóa việc xác định và sửa đổi mô hình về sau một cách đáng kể. Tiếp theo, chúng ta sẽ đi sâu hơn vào cơ chế của việc khởi tạo.

<sup>130</sup> <https://discuss.mxnet.io/t/2326>

<sup>131</sup> <https://forum.machinelearningcoban.com/c/d21>

### 7.3.1 Khởi tạo Mạng

Để bắt đầu, hãy cùng khởi tạo một MLP.

```
from mxnet import init, np, npx
from mxnet.gluon import nn
npx.set_np()

def getnet():
    net = nn.Sequential()
    net.add(nn.Dense(256, activation='relu'))
    net.add(nn.Dense(10))
    return net

net = getnet()
```

Lúc này, mạng nơ-ron không thể biết được chiều của các trọng số ở tầng đầu vào bởi nó còn chưa biết chiều của đầu vào. Và vì thế MXNet chưa khởi tạo bất kỳ tham số nào cả. Ta có thể xác thực việc này bằng cách truy cập các tham số như dưới đây.

```
print(net.collect_params)
print(net.collect_params())
```

Chú ý rằng mặc dù đối tượng Parameter có tồn tại, chiều đầu vào của mỗi tầng được liệt kê là -1. MXNet sử dụng giá trị đặc biệt -1 để ám chỉ việc chưa biết chiều tham số. Tại thời điểm này, việc thử truy cập `net[0].weight.data()` sẽ gây ra lỗi thực thi báo rằng mạng cần khởi tạo trước khi truy cập tham số. Vậy giờ hãy cùng xem điều gì sẽ xảy ra khi ta thử khởi tạo các tham số với phương thức `initialize`.

```
net.initialize()
net.collect_params()
```

Như ta đã thấy, không có gì thay đổi ở đây cả. Khi chưa biết chiều của đầu vào, việc gọi phương thức khởi tạo không thực sự khởi tạo các tham số. Thay vào đó, việc gọi phương thức trên sẽ chỉ đăng ký với MXNet là chúng ta muốn khởi tạo các tham số và phân phối mà ta muốn dùng để khởi tạo (không bắt buộc). Chỉ khi truyền dữ liệu qua mạng thì MXNet mới khởi tạo các tham số và ta mới thấy được sự khác biệt.

```
x = np.random.uniform(size=(2, 20))
net(x) # Forward computation

net.collect_params()
```

Ngay khi biết được chiều của đầu vào là  $\mathbf{x} \in \mathbb{R}^{20}$ , MXNet có thể xác định kích thước của ma trận trọng số tầng đầu tiên:  $\mathbf{W}_1 \in \mathbb{R}^{256 \times 20}$ . Sau khi biết được kích thước tầng đầu tiên, MXNet tiếp tục tính kích thước tầng thứ hai ( $10 \times 256$ ) và cứ thế đi hết đồ thị tính toán cho đến khi nó biết được kích thước của mọi tầng. Chú ý rằng trong trường hợp này, chỉ tầng đầu tiên cần được khởi tạo trễ, tuy nhiên MXNet vẫn khởi tạo theo thứ tự. Khi mà tất cả kích thước tham số đã được biết, MXNet cuối cùng có thể khởi tạo các tham số.

### 7.3.2 Khởi tạo Trẽ trong Thực tiễn

Giờ ta đã biết nó hoạt động như thế nào về mặt lý thuyết, hãy xem thử khi nào thì việc khởi tạo này thực sự diễn ra. Để làm điều này, chúng ta cần lập trình thử một bộ khởi tạo. Nó sẽ không làm gì ngoài việc gửi một thông điệp gỡ lỗi cho biết khi nào nó được gọi và cùng với các tham số nào.

```
class MyInit(init.Initializer):
    def __init_weight(self, name, data):
        print('Init', name, data.shape)
        # The actual initialization logic is omitted here

net = getnet()
net.initialize(init=MyInit())
```

Lưu ý rằng, mặc dù MyInit sẽ in thông tin về các tham số mô hình khi nó được gọi, hàm khởi tạo initialize ở trên không xuất bất cứ thông tin nào sau khi được thực thi. Do đó, việc khởi tạo tham số không thực sự được thực hiện khi gọi hàm initialize. Kế tiếp, ta định nghĩa đầu vào và thực hiện một lượt phép tính truyền xuôi.

```
x = np.random.uniform(size=(2, 20))
y = net(x)
```

Lúc này, thông tin về các tham số mô hình mới được in ra. Khi thực hiện lượt truyền xuôi dựa trên biến đầu vào x, hệ thống có thể tự động suy ra kích thước các tham số của tất cả các tầng dựa trên kích thước của biến đầu vào này. Một khi hệ thống đã tạo ra các tham số trên, nó sẽ gọi thực thể MyInit để khởi tạo chúng trước khi bắt đầu thực hiện lượt truyền xuôi.

Việc khởi tạo này sẽ chỉ được gọi khi lượt truyền xuôi đầu tiên hoàn thành. Sau thời điểm này, chúng ta sẽ không khởi tạo lại khi dùng lệnh net(x) để thực hiện lượt truyền xuôi, do đó đầu ra của thực thể MyInit sẽ không được sinh ra nữa.

```
y = net(x)
```

Như đã đề cập ở phần mở đầu của mục này, việc khởi tạo trễ cũng có thể gây ra sự khó hiểu. Trước khi lượt truyền xuôi đầu tiên được thực thi, chúng ta không thể thao tác trực tiếp lên các tham số của mô hình. Chẳng hạn, chúng ta sẽ không thể dùng các hàm data và set\_data để nhận và thay đổi các tham số. Do đó, chúng ta thường ép việc khởi tạo diễn ra bằng cách đưa một mẫu dữ liệu qua mạng này.

### 7.3.3 Khởi tạo Cưỡng chế

Khởi tạo trễ không xảy ra nếu hệ thống đã biết kích thước của tất cả các tham số khi gọi hàm initialize. Việc này có thể xảy ra trong hai trường hợp:

- Ta đã truyền dữ liệu vào mạng từ trước và chỉ muốn khởi tạo lại các tham số.
- Ta đã chỉ rõ cả chiều đầu vào và chiều đầu ra của mạng khi định nghĩa nó.

Khởi tạo cưỡng chế hoạt động như minh họa dưới đây.

```
net.initialize(init=MyInit(), force_reinit=True)
```

Trường hợp thứ hai yêu cầu ta chỉ rõ tất cả tham số khi tạo mỗi tầng trong mạng. Ví dụ, với các tầng kết nối dày đặc thì chúng ta cần chỉ rõ `in_units` tại thời điểm tầng đó được khởi tạo.

```
net = nn.Sequential()
net.add(nn.Dense(256, in_units=20, activation='relu'))
net.add(nn.Dense(10, in_units=256))

net.initialize(init=MyInit())
```

### 7.3.4 Tóm tắt

- Khởi tạo trẽ có thể khá tiện lợi, cho phép Gluon suy ra kích thước của tham số một cách tự động và nhờ vậy giúp ta dễ dàng sửa đổi các kiến trúc mạng cũng như loại bỏ những nguồn gây lỗi thông dụng.
- Chúng ta không cần khởi tạo trẽ khi đã định nghĩa các biến một cách tường minh.
- Chúng ta có thể cưỡng chế việc khởi tạo lại các tham số mạng bằng cách gọi khởi tạo với `force_reinit=True`.

### 7.3.5 Bài tập

1. Chuyện gì xảy ra nếu ta chỉ chỉ rõ chiều đầu vào của tầng đầu tiên nhưng không làm vậy với các tầng tiếp theo? Việc khởi tạo có xảy ra ngay lập tức không?
2. Chuyện gì xảy ra nếu ta chỉ định các chiều không khớp nhau?
3. Bạn cần làm gì nếu đầu vào có chiều biến thiên? Gợi ý - hãy tìm hiểu về cách ràng buộc tham số (*parameter tying*).

### 7.3.6 Thảo luận

- Tiếng Anh<sup>132</sup>
- Tiếng Việt<sup>133</sup>

### 7.3.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Lý Phi Long
- Nguyễn Mai Hoàng Long

<sup>132</sup> <https://discuss.mxnet.io/t/2327>

<sup>133</sup> <https://forum.machinelearningcoban.com/c/d21>

- Phạm Minh Đức
- Nguyễn Lê Quang Nhật

## 7.4 Các tầng Tuỳ chỉnh

Một trong những yếu tố dẫn đến thành công của học sâu là sự đa dạng của các tầng. Những tầng này có thể được sắp xếp theo nhiều cách sáng tạo để thiết kế nên những kiến trúc phù hợp với nhiều tác vụ khác nhau. Ví dụ, các nhà nghiên cứu đã phát minh ra các tầng chuyên dụng để xử lý ảnh, chữ viết, lặp trên dữ liệu tuần tự, thực thi quy hoạch động, v.v... Dù sớm hay muộn, bạn cũng sẽ gặp (hoặc sáng tạo) một tầng không có trong Gluon. Đối với những trường hợp như vậy, bạn cần xây dựng một tầng tùy chỉnh. Phần này sẽ hướng dẫn bạn cách thực hiện điều đó.

### 7.4.1 Các tầng không có Tham số

Để bắt đầu, ta tạo một tầng tùy chỉnh (một Khối) không chứa bất kỳ tham số nào. Bước này khá quen thuộc nếu bạn còn nhớ phần giới thiệu về Block của Gluon tại [Section 7.1](#). Lớp CenteredLayer chỉ đơn thuần trừ đi giá trị trung bình từ đầu vào của nó. Để xây dựng nó, chúng ta chỉ cần kế thừa từ lớp Block và lập trình phương thức forward.

```
from mxnet import gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

class CenteredLayer(nn.Block):
    def __init__(self, **kwargs):
        super(CenteredLayer, self).__init__(**kwargs)

    def forward(self, x):
        return x - x.mean()
```

Hãy cùng xác thực rằng tầng này hoạt động như ta mong muốn bằng cách truyền dữ liệu vào nó.

```
layer = CenteredLayer()
layer(np.array([1, 2, 3, 4, 5]))
```

Chúng ta cũng có thể kết hợp tầng này như là một thành phần để xây dựng các mô hình phức tạp hơn.

```
net = nn.Sequential()
net.add(nn.Dense(128), CenteredLayer())
net.initialize()
```

Để kiểm tra thêm, chúng ta có thể truyền dữ liệu ngẫu nhiên qua mạng và chứng thực xem giá trị trung bình đã về 0 hay chưa. Chú ý rằng vì đang làm việc với các số thực dấu phẩy động, chúng ta sẽ thấy một giá trị khác không rất nhỏ.

```
y = net(np.random.uniform(size=(4, 8)))
y.mean()
```

## 7.4.2 Tầng có Tham số

Giờ đây ta đã biết cách định nghĩa các tầng đơn giản, hãy chuyển sang việc định nghĩa các tầng chứa tham số có thể điều chỉnh được trong quá trình huấn luyện. Để tự động hóa các công việc lặp lại, lớp Parameter và từ điển ParameterDict cung cấp một số tính năng quản trị cơ bản. Cụ thể, chúng sẽ quản lý việc truy cập, khởi tạo, chia sẻ, lưu và nạp các tham số mô hình. Bằng cách này, cùng với nhiều lợi ích khác, ta không cần phải viết lại các thủ tục tuần tự hóa (*serialization*) cho mỗi tầng tùy chỉnh mới.

Lớp Block chứa biến params với kiểu dữ liệu ParameterDict. Từ điển này ánh xạ các xâu kí tự biểu thị tên tham số đến các tham số mô hình (thuộc kiểu Parameter). ParameterDict cũng cung cấp hàm get giúp việc tạo tham số mới với tên và chiều cụ thể trở nên dễ dàng.

```
params = gluon.ParameterDict()  
params.get('param2', shape=(2, 3))  
params
```

Giờ đây chúng ta đã có tất cả các thành phần cơ bản cần thiết để tự tạo một phiên bản tùy chỉnh của tầng Dense trong Gluon. Chú ý rằng tầng này yêu cầu hai tham số: một cho trọng số và một cho hệ số điều chỉnh. Trong cách lập trình này, ta sử dụng hàm kích hoạt mặc định là hàm ReLU. Trong hàm \_\_init\_\_, in\_units và units biểu thị lần lượt số lượng đầu vào và đầu ra.

```
class MyDense(nn.Block):  
    # units: the number of outputs in this layer; in_units: the number of  
    # inputs in this layer  
    def __init__(self, units, in_units, **kwargs):  
        super(MyDense, self).__init__(**kwargs)  
        self.weight = self.params.get('weight', shape=(in_units, units))  
        self.bias = self.params.get('bias', shape=(units,))  
  
    def forward(self, x):  
        linear = np.dot(x, self.weight.data()) + self.bias.data()  
        return npx.relu(linear)
```

Vì việc đặt tên cho các tham số cho phép ta truy cập chúng theo tên thông qua tra cứu từ điển sau này. Nhìn chung, bạn sẽ muốn đặt cho các biến những tên đơn giản biểu thị rõ mục đích của chúng. Tiếp theo, ta sẽ khởi tạo lớp MyDense và truy cập các tham số mô hình. Lưu ý rằng tên của Khối được tự động thêm vào trước tên các tham số.

```
dense = MyDense(units=3, in_units=5)  
dense.params
```

Ta có thể trực tiếp thực thi các phép tính truyền xuôi có sử dụng các tầng tùy chỉnh.

```
dense.initialize()  
dense(np.random.uniform(size=(2, 5)))
```

Các tầng tùy chỉnh cũng có thể được dùng để xây dựng mô hình. Chúng có thể được sử dụng như các tầng kết nối dày đặc được lập trình sẵn. Ngoại lệ duy nhất là việc suy luận kích thước sẽ không được thực hiện tự động. Để biết thêm chi tiết về cách thực hiện việc này, vui lòng tham khảo [tài liệu MXNet](#)<sup>134</sup>.

<sup>134</sup> <http://www.mxnet.io>

```
net = nn.Sequential()  
net.add(MyDense(8, in_units=64),  
       MyDense(1, in_units=8))  
net.initialize()  
net(np.random.uniform(size=(2, 64)))
```

### 7.4.3 Tóm tắt

- Ta có thể thiết kế các tầng tùy chỉnh thông qua lớp Block. Điều này cho phép ta định nghĩa một cách linh hoạt các tầng có cách hoạt động khác với các tầng có sẵn trong thư viện.
- Một khi đã được định nghĩa, các tầng tùy chỉnh có thể được gọi trong những bối cảnh và kiến trúc tùy ý.
- Các khối có thể có các tham số cục bộ, được lưu trữ dưới dạng đối tượng ParameterDict trong mỗi thuộc tính params của Block.

### 7.4.4 Bài tập

1. Thiết kế một tầng có khả năng học một phép biến đổi affine của dữ liệu.
2. Thiết kế một tầng nhận đầu vào và tính toán phép giảm tensor, tức trả về  $y_k = \sum_{i,j} W_{ijk}x_i x_j$ .
3. Thiết kế một tầng trả về nửa đầu của các hệ số Fourier của dữ liệu. Gợi ý: hãy tra cứu hàm fft trong MXNet.

### 7.4.5 Thảo luận

- Tiếng Anh<sup>135</sup>
- Tiếng Việt<sup>136</sup>

### 7.4.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Duy Du

<sup>135</sup> <https://discuss.mxnet.io/t/2328>

<sup>136</sup> <https://forum.machinelearningcoban.com/c/d21>

## 7.5 Đọc/Ghi tệp

Đến nay, ta đã thảo luận về cách xử lý dữ liệu và cách xây dựng, huấn luyện, kiểm tra những mô hình học sâu. Tuy nhiên, có thể đến một lúc nào đó ta sẽ cảm thấy hài lòng với những gì thu được và muốn lưu lại kết quả để sau này sử dụng trong những bối cảnh khác nhau (thậm chí có thể đưa ra dự đoán khi triển khai). Ngoài ra, khi vận hành một quá trình huấn luyện dài hơi, cách tốt nhất là lưu kết quả trung gian một cách định kỳ (điểm kiểm tra) nhằm đảm bảo rằng ta sẽ không mất kết quả tính toán sau nhiều ngày nếu chẳng may ta vấp phải dây nguồn của máy chủ. Vì vậy, đã đến lúc chúng ta học cách đọc và lưu trữ đồng thời các vector trọng số riêng lẻ cùng toàn bộ các mô hình. Mục này sẽ giải quyết cả hai vấn đề trên.

### 7.5.1 Đọc và Lưu các ndarray

Đối với ndarray riêng lẻ, ta có thể sử dụng trực tiếp các hàm load và save để đọc và ghi tương ứng. Cả hai hàm đều yêu cầu ta cung cấp tên, và hàm save yêu cầu đầu vào với biến đã được lưu.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

x = np.arange(4)
npx.save('x-file', x)
```

Bây giờ, chúng ta có thể đọc lại dữ liệu từ các tệp được lưu vào bộ nhớ.

```
x2 = npx.load('x-file')
x2
```

MXNet đồng thời cho phép ta lưu một danh sách các ndarray và đọc lại chúng vào bộ nhớ.

```
y = np.zeros(4)
npx.save('x-files', [x, y])
x2, y2 = npx.load('x-files')
(x2, y2)
```

Ta còn có thể ghi và đọc một từ điển ánh xạ từ một chuỗi sang một ndarray. Điều này khá là thuận tiện khi chúng ta muốn đọc hoặc ghi tất cả các trọng số của một mô hình.

```
mydict = {'x': x, 'y': y}
npx.save('mydict', mydict)
mydict2 = npx.load('mydict')
mydict2
```

## 7.5.2 Tham số mô hình Gluon

Khả năng lưu từng vector trọng số đơn lẻ (hoặc các ndarray tensor khác) là hữu ích nhưng sẽ mất nhiều thời gian nếu chúng ta muốn lưu (và sau đó nạp lại) toàn bộ mô hình. Dù sao, chúng ta có thể có hàng trăm nhóm tham số rải rác xuyên suốt mô hình. Vì lý do đó mà Gluon cung cấp sẵn tính năng lưu và nạp toàn bộ các mạng. Một chi tiết quan trọng cần lưu ý là chức năng này chỉ lưu các *tham số* của mô hình, không phải là toàn bộ mô hình. Điều đó có nghĩa là nếu ta có một MLP ba tầng, ta cần chỉ rõ *kiến trúc* này một cách riêng lẻ. Lý do là vì bản thân các mô hình có thể chứa mã nguồn bất kỳ, chúng không được thêm vào tập tin một cách dễ dàng như các tham số (có một cách thực hiện điều này cho các mô hình đã được biên dịch, chi tiết kĩ thuật đọc thêm trong [tài liệu MXNet<sup>137</sup>](#)). Vì vậy, để khôi phục lại một mô hình thì chúng ta cần xây dựng kiến trúc của nó từ mã nguồn rồi nạp các tham số từ ổ cứng vào kiến trúc này. Việc khởi tạo trễ ([Section 7.3](#)) lúc này rất có lợi vì ta chỉ cần định nghĩa một mô hình mà không cần gán giá trị cụ thể cho tham số. Như thường lệ, hãy bắt đầu với một MLP quen thuộc.

```
class MLP(nn.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        self.hidden = nn.Dense(256, activation='relu')
        self.output = nn.Dense(10)

    def forward(self, x):
        return self.output(self.hidden(x))

net = MLP()
net.initialize()
x = np.random.uniform(size=(2, 20))
y = net(x)
```

Tiếp theo, chúng ta lưu các tham số của mô hình vào tệp `mlp.params`. Những khối Gluon hỗ trợ phương thức từ hàm `save_parameter` nhằm ghi tất cả các tham số vào ổ cứng được cung cấp với một chuỗi những tên tệp.

```
net.save_parameters('mlp.params')
```

Để khôi phục mô hình, chúng ta tạo một đối tượng khác dựa trên mô hình MLP gốc. Thay vì khởi tạo ngẫu nhiên những tham số mô hình, ta đọc các tham số được lưu trực tiếp trong tập tin. Và thật thuận tiện, ta đã có thể nạp các tham số vào khối thông qua phương thức từ hàm `load_parameters`.

```
clone = MLP()
clone.load_parameters('mlp.params')
```

Vì cả hai đối tượng của mô hình có cùng bộ tham số, kết quả tính toán với cùng đầu vào `x` sẽ như nhau. Hãy kiểm chứng điều này.

```
yclone = clone(x)
yclone == y
```

<sup>137</sup> <http://www.mxnet.io>

### 7.5.3 Tóm tắt

- Hàm save và load có thể được sử dụng để thực hiện việc xuất nhập tập tin cho các đối tượng ndarray.
- Hàm load\_parameters và save\_parameters cho phép ta lưu toàn bộ tập tham số của một mạng trong Gluon.
- Việc lưu kiến trúc này phải được hoàn thiện trong chương trình thay vì trong các tham số.

### 7.5.4 Bài tập

1. Nếu không cần phải triển khai các mô hình huấn luyện sang một thiết bị khác, theo bạn thì lợi ích thực tế của việc lưu các tham số mô hình là gì?
2. Giả sử chúng ta muốn sử dụng lại chỉ một phần của một mạng nào đó để phối hợp với một mạng của một kiến trúc khác. Trong trường hợp ta muốn sử dụng hai lớp đầu tiên của mạng trước đó vào trong một mạng mới, bạn sẽ làm thế nào để thực hiện được việc này?
3. Làm thế nào để bạn có thể lưu kiến trúc mạng và các tham số? Có những hạn chế nào khi bạn tận dụng kiến trúc này?

### 7.5.5 Thảo luận

- Tiếng Anh<sup>138</sup>
- Tiếng Việt<sup>139</sup>

### 7.5.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Hồng Vinh

<sup>138</sup> <https://discuss.mxnet.io/t/2329>

<sup>139</sup> <https://forum.machinelearningcoban.com/c/d21>

## 7.6 GPU

Trong phần giới thiệu của cuốn sách này, chúng ta đã thảo luận về sự tăng trưởng đột phá của năng lực tính toán trong hai thập niên vừa qua. Một cách ngắn gọn, hiệu năng GPU đã tăng lên gấp 1000 lần trong mỗi thập niên kể từ năm 2000. Điều này mang lại cơ hội to lớn nhưng kèm theo đó là một nhu cầu không hề nhỏ để cung cấp hiệu năng tính toán như vậy.

Thập niên	Tập dữ liệu	Bộ nhớ	Số Phép tính Dấu phẩy động trên Giây
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (Giá nhà tại Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (Nhận diện ký tự quang học)	10 MB	10 MF (Intel 80486)
2000	10 M (các trang web)	100 MB	1 GF (Intel Core)
2010	10 G (quảng cáo)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (mạng xã hội)	100 GB	1 PF (NVIDIA DGX-2)

Trong phần này, ta bắt đầu thảo luận cách khai thác hiệu năng tính toán này cho việc nghiên cứu. Đầu tiên ta sẽ tìm hiểu cách sử dụng một GPU duy nhất, rồi sau này tiến tới nhiều GPU và nhiều máy chủ (cùng với nhiều GPU). Bạn có thể đã nhận ra MXNet ndarray trông gần như giống hệt NumPy, nhưng chúng có một vài điểm khác biệt quan trọng. Một trong những tính năng chính khiến cho MXNet khác với NumPy là MXNet hỗ trợ nhiều loại phần cứng đa dạng.

Trong MXNet, mỗi mảng có một bối cảnh. Cho tới giờ, tất cả các biến và phép toán liên quan đều được giao cho CPU theo mặc định. Các bối cảnh thường có thể là nhiều GPU khác. Mọi thứ còn có thể trở nên rõ ràng hơn khi ta triển khai công việc trên nhiều máy chủ. Bằng cách chỉ định bối cảnh cho các mảng một cách thông minh, ta có thể giảm thiểu thời gian truyền tải dữ liệu giữa các thiết bị. Ví dụ, khi huấn luyện mạng nơ-ron trên máy chủ có GPU, ta thường muốn các tham số mô hình nằm ở trên GPU.

Nói ngắn gọn, với những mạng nơ-ron phức tạp và dữ liệu quy mô lớn, việc chỉ sử dụng CPU để tính toán có thể sẽ không hiệu quả. Trong phần này, ta sẽ thảo luận về cách sử dụng một GPU NVIDIA duy nhất cho việc tính toán. Đầu tiên, hãy chắc chắn rằng bạn đã lắp đặt ít nhất một GPU NVIDIA. Sau đó, hãy tải CUDA<sup>140</sup> và làm theo gợi ý để thiết lập đường dẫn hợp lý. Một khi các bước chuẩn bị đã được hoàn thành, ta có thể dùng lệnh nvidia-smi để xem thông tin card đồ họa.

```
!nvidia-smi
```

Tiếp theo, cần chắc chắn rằng ta đã cài đặt phiên bản GPU của MXNet. Nếu phiên bản CPU của MXNet đã được cài đặt trước, ta cần phải gỡ bỏ nó. Ví dụ, hãy sử dụng lệnh pip uninstall mxnet, sau đó cài đặt phiên bản MXNet tương ứng với phiên bản CUDA. Giả sử như bạn đã cài CUDA 9.0, bạn có thể cài phiên bản MXNet có hỗ trợ CUDA 9.0 bằng lệnh pip install mxnet-cu90. Để chạy các chương trình trong phần này, bạn cần ít nhất hai GPU.

Yêu cầu này có vẻ khá phung phí với hầu hết các bộ máy tính để bàn nhưng lại rất dễ dàng nếu ta dùng các dịch vụ đám mây, chẳng hạn ta có thể thuê một máy chủ AWS EC2 đa GPU. Hầu hết các phần khác trong cuốn sách này *không* yêu cầu đa GPU. Tuy nhiên, việc này chỉ để minh họa cách dữ liệu được truyền giữa các thiết bị khác nhau.

<sup>140</sup> <https://developer.nvidia.com/cuda-downloads>

### 7.6.1 Thiết bị Tính toán

MXNet có thể chỉ định các thiết bị, chẳng hạn như CPU và GPU, cho việc lưu trữ và tính toán. Mặc định, MXNet tạo dữ liệu trong bộ nhớ chính và sau đó sử dụng CPU để tính toán. Trong MXNet, CPU và GPU có thể được chỉ định bởi `cpu()` và `gpu()`. Cần lưu ý rằng `cpu()` (đơn thuần hoặc thêm bất kỳ số nguyên nào trong ngoặc đơn) có nghĩa là sử dụng tất cả các CPU và bộ nhớ vật lý. Điều này có nghĩa các tính toán của MXNet sẽ cố gắng tận dụng tất cả các lõi CPU. Tuy nhiên, `gpu()` đơn thuần chỉ đại diện cho một card đồ họa và bộ nhớ đồ họa tương ứng. Nếu có nhiều GPU, chúng tôi sử dụng `gpu(i)` để thể hiện GPU thứ  $i$  (với  $i$  bắt đầu từ 0). Ngoài ra, `gpu(0)` và `gpu()` tương đương nhau.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

npx.cpu(), npx.gpu(), npx.gpu(1)
```

Ta có thể truy vấn số lượng GPU có sẵn thông qua `num_gpus()`.

```
npx.num_gpus()
```

Bây giờ ta định nghĩa hai hàm chức năng thuận tiện cho việc chạy mã kể cả khi GPU được yêu cầu không tồn tại.

```
# Saved in the d2l package for later use
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    return npx.gpu(i) if npx.num_gpus() >= i + 1 else npx.cpu()

# Saved in the d2l package for later use
def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    ctxes = [npx.gpu(i) for i in range(npx.num_gpus())]
    return ctxes if ctxes else [npx.cpu()]

try_gpu(), try_gpu(3), try_all_gpus()
```

### 7.6.2 ndarray và GPU

Mặc định, các đối tượng `ndarray` được tạo trên CPU. Do đó, ta sẽ thấy định danh `@cpu(0)` mỗi khi ta in một `ndarray`.

```
x = np.array([1, 2, 3])
x.ctx
```

Điều quan trọng cần lưu ý là bất cứ khi nào ta muốn làm các phép toán trên nhiều số hạng, chúng cần phải ở trong cùng một bối cảnh. Chẳng hạn, nếu ta tính tổng hai biến, ta cần đảm bảo rằng cả hai đối số đều nằm trên cùng một thiết bị — nếu không thì MXNet sẽ không biết nơi lưu trữ kết quả hoặc thậm chí cách quyết định nơi thực hiện tính toán.

## Lưu trữ trên GPU

Có một số cách để lưu trữ một ndarray trên GPU. Ví dụ: ta có thể chỉ định một thiết bị lưu trữ với tham số ctx khi tạo một ndarray. Tiếp theo, ta tạo biến ndarray là a trên gpu(0). Lưu ý rằng khi in a ra màn hình, thông tin thiết bị sẽ trở thành @gpu(0). ndarray được tạo trên GPU nào chỉ chiếm bộ nhớ của GPU đó. Ta có thể sử dụng lệnh nvidia-smi để xem việc sử dụng bộ nhớ GPU. Nói chung, ta cần đảm bảo rằng ta không tạo dữ liệu vượt quá giới hạn bộ nhớ GPU.

```
x = np.ones((2, 3), ctx=try_gpu())
x
```

Giả sử bạn có ít nhất hai GPU, đoạn mã sau sẽ tạo ra một mảng ngẫu nhiên trên gpu(1).

```
y = np.random.uniform(size=(2, 3), ctx=try_gpu(1))
y
```

## Sao chép

Nếu ta muốn tính  $x + y$  thì ta cần quyết định nơi thực hiện phép tính này. Chẳng hạn, như trong Fig. 7.6.1, ta có thể chuyển  $x$  sang gpu(1) và thực hiện phép tính ở đó. *Đừng* chỉ thêm  $x + y$  vì điều này sẽ dẫn đến một lỗi. Hệ thống thời gian chạy sẽ không biết phải làm gì và gặp lỗi bởi nó không thể tìm thấy dữ liệu trên cùng một thiết bị.

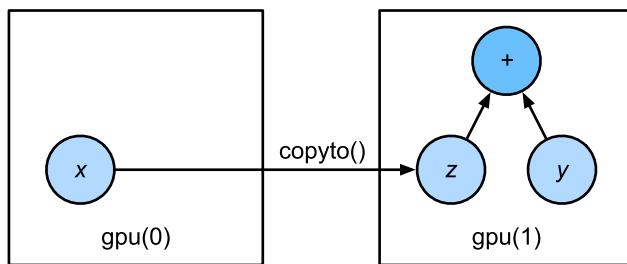


Fig. 7.6.1: Lệnh copyto sao chép các mảng đến thiết bị mục tiêu

Lệnh copyto sao chép dữ liệu sang một thiết bị khác để ta có thể cộng chúng. Vì  $y$  tồn tại trên GPU thứ hai, ta cần di chuyển  $x$  trước khi ta có thể cộng chúng lại.

```
z = x.copyto(try_gpu(1))
print(x)
print(z)
```

Bây giờ dữ liệu đã ở trên cùng một GPU (cả  $z$  và  $y$ ), ta có thể cộng lại. Trong những trường hợp như vậy MXNet lưu kết quả tại cùng thiết bị với các toán hạng. Trong trường hợp này là @gpu(1).

```
y + z
```

Giả sử biến  $z$  hiện đang được lưu trong GPU thứ hai (gpu(1)). Điều gì sẽ xảy ra nếu ta gọi  $z.copyto(gpu(1))$ ? Hàm này sẽ tạo một bản sao của biến và cấp phát vùng nhớ mới cho bản sao, ngay cả khi biến đang có trong thiết bị. Có những lúc mà tuỳ thuộc vào môi trường thực thi lệnh, hai biến có thể đã ở trên cùng một thiết bị. Do đó chúng ta muốn chỉ tạo bản sao khi các biến tồn tại ở các ngữ cảnh khác nhau. Trong các trường hợp đó, ta có thể gọi `as_in_ctx()`. Nếu biến đó

đã tồn tại trong ngữ cảnh thì hàm này không thực hiện lệnh nào. Trên thực tế, trừ trường hợp đặc biệt bạn muốn tạo bản sao, hãy sử dụng `as_in_ctx()`.

```
z = x.as_in_ctx(try_gpu(1))
z
```

Cần lưu ý rằng, nếu ctx của biến nguồn và biến đích là giống nhau, hàm `as_in_ctx` sẽ khiến biến đích có cùng vùng nhớ với biến nguồn.

```
y.as_in_ctx(try_gpu(1)) is y
```

Hàm `copyto` luôn luôn cấp phát vùng nhớ mới cho biến đích.

```
y.copyto(try_gpu(1)) is y
```

### Những lưu ý bên lề

Mọi người sử dụng GPU để thực hiện việc tính toán trong học máy vì họ kỳ vọng chúng sẽ nhanh hơn. Nhưng việc truyền các biến giữa các bối cảnh lại diễn ra chậm. Do đó, chúng tôi mong bạn chắc chắn 100% rằng bạn muốn thực hiện một việc nào đó thật chậm trước khi chúng tôi để bạn thực hiện nó. Nếu MXNet chỉ thực hiện việc sao chép tự động mà không gặp sự cố thì có thể bạn sẽ không nhận ra được mình đã có những đoạn mã chưa tối ưu đến nhường nào.

Thêm vào đó, việc truyền dữ liệu giữa các thiết bị (CPU, GPU và các máy khác) *chậm hơn nhiều* so với việc thực hiện tính toán. Nó cũng làm cho việc song song hóa trở nên khó hơn nhiều, vì chúng ta phải chờ cho dữ liệu được gửi đi (hoặc được nhận về) trước khi chúng ta có thể tiến hành nhiều tác vụ xử lý tính toán hơn. Đây là lý do tại sao các hoạt động sao chép nên được dành sự lưu tâm lớn. Quy tắc nằm lòng là nhiều xử lý tính toán nhỏ thì tệ hơn nhiều so với một xử lý tính toán lớn. Hơn nữa, xử lý nhiều phép tính toán cùng một thời điểm thì tốt hơn nhiều so với nhiều xử lý tính toán đơn lẻ nằm rải rác trong chương trình (trừ khi là bạn hiểu rõ mình đang làm gì). Lý do là ở tình huống này những hoạt động như vậy có thể gây tắc nghẽn nếu một thiết bị phải chờ một thiết bị khác trước khi nó có thể làm điều gì đó khác. Việc này hơi giống việc bạn phải xếp hàng mua cà phê thay vì đặt trước qua điện thoại và biết được khi nào nó đã sẵn sàng để đến lấy.

Sau cùng, khi chúng ta in các `ndarray` hoặc chuyển các `ndarray` sang định dạng Numpy, nếu dữ liệu không có trong bộ nhớ chính, MXNet sẽ sao chép nó tới bộ nhớ chính trước tiên, dẫn tới việc tốn thêm thời gian chờ cho việc truyền dữ liệu. Thậm chí tệ hơn, điều đáng sợ lúc này là nó phụ thuộc vào Bộ Khóa Phiên dịch Toàn cục (*Global Interpreter Lock*) khiến mọi thứ phải chờ Python hoàn tất.

### 7.6.3 Gluon và GPU

Tương tự, mô hình của Gluon có thể chỉ định thiết bị dựa vào tham số `ctx` trong quá trình khởi tạo. Đoạn mã dưới đây khởi tạo các tham số của mô hình trên GPU (sau này chúng ta sẽ thấy nhiều ví dụ về cách chạy các mô hình trên GPU, đơn giản bởi chúng phần nào sẽ cần khả năng tính toán mạnh hơn).

```
net = nn.Sequential()
net.add(nn.Dense(1))
net.initialize(ctx=try_gpu())
```

Khi đầu vào là một ndarray trên GPU, Gluon sẽ tính toán kết quả trên cùng GPU đó.

```
net(x)
```

Hãy kiểm chứng lại rằng các tham số của mô hình được lưu trên cùng GPU.

```
net[0].weight.data().ctx
```

Tóm lại, khi dữ liệu và các tham số ở trên cùng thiết bị, ta có thể huấn luyện mô hình một cách hiệu quả. Ta sẽ xem xét một vài ví dụ như thế trong phần tiếp theo.

#### 7.6.4 Tóm tắt

- MXNet có thể chỉ định các thiết bị thực hiện việc lưu trữ và tính toán như CPU hay GPU. Mặc định, MXNet tạo dữ liệu trên bộ nhớ chính và sử dụng CPU để tính toán.
- MXNet yêu cầu tất cả dữ liệu đầu vào nằm trên cùng thiết bị trước khi thực hiện tính toán, tức cùng một CPU hoặc cùng một GPU.
- Hiệu năng có thể giảm đáng kể nếu di chuyển dữ liệu một cách không cần thận. Một lỗi thường gặp là: việc tính toán mất mát cho các minibatch trên GPU rồi in kết quả ra cửa sổ dòng lệnh (hoặc ghi kết quả vào mảng NumPy) sẽ kích hoạt Bộ Khóa Phiên dịch Toàn cục làm tất cả GPU dừng hoạt động. Sẽ tốt hơn nếu cấp phát bộ nhớ cho việc ghi lại quá trình hoạt động (*logging*) ở GPU và chỉ di chuyển các bản ghi lớn.

#### 7.6.5 Bài tập

1. Thử một tác vụ có khối lượng tính toán lớn, ví dụ như nhân các ma trận kích thước lớn để thấy sự khác nhau về tốc độ giữa CPU và GPU. Và với tác vụ có khối lượng tính toán nhỏ thì sao?
2. Làm thế nào để đọc và ghi các tham số của mô hình trên GPU?
3. Đo thời gian thực hiện 1000 phép nhân ma trận kích thước  $100 \times 100$  và ghi lại giá trị chuẩn  $trMM^T$  của từng kết quả, rồi so sánh với việc lưu tất cả giá trị chuẩn tại một bản ghi ở GPU và chỉ trả về bản ghi đó.
4. Đo thời gian thực hiện hai phép nhân ma trận tại hai GPU cùng lúc so với việc thực hiện chúng lần lượt trên cùng một GPU (gợi ý: bạn sẽ thấy tỉ lệ gần như tuyến tính).

#### 7.6.6 Thảo luận

- [Tiếng Anh](#)<sup>141</sup>
- [Tiếng Việt](#)<sup>142</sup>

<sup>141</sup> <https://discuss.mxnet.io/t/2330>

<sup>142</sup> <https://forum.machinelearningcoban.com/c/d2l>

### **7.6.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Nguyễn Mai Hoàng Long
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Phạm Minh Đức
- Vũ Hữu Tiệp

### **7.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Minh Đức



## 8 | Mạng Nơ-ron Tích chập

Trong những chương đầu tiên, chúng ta đã làm việc trên dữ liệu ảnh với mỗi mẫu là một mảng điểm ảnh 2D. Tùy vào ảnh đen trắng hay ảnh màu mà ta cần xử lý *một* hay *nhiều* giá trị số học tương ứng tại mỗi vị trí điểm ảnh. Cho đến nay, cách ta xử lý dữ liệu với cấu trúc phong phú này vẫn chưa thật sự thỏa đáng. Ta chỉ đang đơn thuần loại bỏ cấu trúc không gian từ mỗi bức ảnh bằng cách chuyển chúng thành các vector và truyền chúng qua một mạng MLP (kết nối đầy đủ). Vì các mạng này là bất biến với thứ tự của các đặc trưng, ta sẽ nhận được cùng một kết quả bất kể việc chúng ta có giữ lại thứ tự cấu trúc không gian của các điểm ảnh hay hoán vị các cột của ma trận đặc trưng trước khi khớp các tham số của mạng MLP. Tốt hơn hết, ta nên tận dụng điều đã biết là các điểm ảnh kề cận thường có tương quan lẫn nhau, để xây dựng những mô hình hiệu quả hơn cho việc học từ dữ liệu ảnh.

Chương này sẽ giới thiệu về các Mạng Nơ-ron Tích chập (*Convolutional Neural Network - CNN*), một họ các mạng nơ-ron ưu việt được thiết kế chính xác cho mục đích trên. Các kiến trúc dựa trên CNN hiện nay xuất hiện trong mọi ngóc ngách của lĩnh vực thị giác máy tính, và đã trở thành kiến trúc chủ đạo mà hiếm ai ngày nay phát triển các ứng dụng thương mại hay tham gia một cuộc thi nào đó liên quan tới nhận dạng ảnh, phát hiện đối tượng, hay phân vùng theo ngữ cảnh mà không xây nền móng dựa trên phương pháp này.

Theo cách hiểu thông dụng, thiết kế của mạng *ConvNets* đã vay mượn rất nhiều ý tưởng từ ngành sinh học, lý thuyết nhóm và lượng rất nhiều những thí nghiệm nhỏ lẻ khác. Bên cạnh hiệu năng cao trên số lượng mẫu cần thiết để đạt được độ chính xác, các mạng nơ-ron tích chập thường có hiệu quả tính toán hơn, bởi đòi hỏi ít tham số hơn và dễ thực thi song song trên nhiều GPU hơn các kiến trúc mạng dày đặc.

Do đó, các mạng CNN sẽ được áp dụng bất cứ khi nào có thể, và chúng đã nhanh chóng trở thành một công cụ quan trọng đáng tin cậy thậm chí với các tác vụ liên quan tới cấu trúc tuần tự một chiều, như là xử lý âm thanh, văn bản, và phân tích dữ liệu chuỗi thời gian (*time series analysis*), mà ở đó các mạng nơ-ron hồi tiếp vốn thường được sử dụng. Với một số điều chỉnh khôn khéo, ta còn có thể dùng mạng CNN cho dữ liệu có cấu trúc đồ thị và hệ thống đề xuất.

Trước hết, chúng ta sẽ đi qua các phép toán cơ bản nhằm tạo nên bộ khung sườn của tất cả các mạng nơ-ron tích chập. Chúng bao gồm các tầng tích chập, các chi tiết cơ bản quan trọng như đệm và sai bước, các tầng gộp dùng để kết hợp thông tin qua các vùng không gian kề nhau, việc sử dụng đa kênh (cũng được gọi là *các bộ lọc*) ở mỗi tầng và một cuộc thảo luận cẩn thận về cấu trúc của các mạng hiện đại. Chúng ta sẽ kết thúc cho chương này với một ví dụ hoàn toàn hoạt động của mạng LeNet, mạng tích chập đầu tiên đã triển khai thành công và tồn tại nhiều năm trước khi có sự trỗi dậy của kỹ thuật học sâu hiện đại. Ở chương kế tiếp, chúng ta sẽ đắm mình vào việc xây dựng hoàn chỉnh một số kiến trúc CNN tương đối gần đây và khá phổ biến. Thiết kế của chúng chứa hầu hết những kỹ thuật mà ngày nay hay được sử dụng.

## 8.1 Từ Tầng Kết nối Dày đặc đến phép Tích chập

Đến nay, các mô hình mà ta đã thảo luận là các lựa chọn phù hợp nếu dữ liệu mà ta đang xử lý có *dạng bảng* với các hàng tương ứng với các mẫu, còn các cột tương ứng với các đặc trưng. Với dữ liệu có dạng như vậy, ta có thể dự đoán rằng khuôn mẫu mà ta đang tìm kiếm có thể yêu cầu việc mô hình hóa sự tương tác giữa các đặc trưng, nhưng ta không giả định trước rằng những đặc trưng nào liên quan tới nhau và mối quan hệ của chúng.

Đôi khi ta thực sự không có bất kỳ kiến thức nào để định hướng việc thiết kế các kiến trúc được sắp xếp khéo léo hơn. Trong những trường hợp này, một perceptron đa tầng thường là giải pháp tốt nhất. Tuy nhiên, một khi ta bắt đầu xử lý dữ liệu tri giác đa chiều, các mạng *không có cấu trúc* này có thể sẽ trở nên quá cồng kềnh.

Hãy quay trở lại với ví dụ phân biệt chó và mèo quen thuộc. Giả sử ta đã thực hiện việc thu thập dữ liệu một cách kỹ lưỡng và thu được một bộ ảnh được gán nhãn chất lượng cao với độ phân giải 1 triệu điểm ảnh. Điều này có nghĩa là đầu vào của mạng sẽ có *1 triệu chiều*. Ngay cả việc giảm mạnh xuống còn *1000 chiều* vẫn sẽ cần tới một *tầng dày đặc* (kết nối dày đặc) có  $10^9$  tham số. Trừ khi ta có một tập dữ liệu cực lớn (có thể là hàng tỷ ảnh?), một số lượng lớn GPU, chuyên môn cao trong việc tối ưu hóa phân tán và sức kiên nhẫn phi thường, việc học các tham số của mạng này có thể là điều bất khả thi.

Độc giả kỹ tính có thể phản đối lập luận này trên cơ sở độ phân giải 1 triệu điểm ảnh có thể là không cần thiết. Tuy nhiên, ngay cả khi chỉ sử dụng 100.000 điểm ảnh, ta đã đánh giá quá thấp số lượng các nút ẩn cần thiết để tìm các biểu diễn ẩn tốt của các ảnh. Việc học một bộ phân loại nhị phân với rất nhiều tham số có thể sẽ cần tới một tập dữ liệu khổng lồ, có lẽ tương đương với số lượng chó và mèo trên hành tinh này. Tuy nhiên, việc cả con người và máy tính đều có thể phân biệt mèo với chó khá tốt dường như mâu thuẫn với các kết luận trên. Đó là bởi vì các ảnh thể hiện cấu trúc phong phú, thường được khai thác bởi con người và các mô hình học máy theo các cách giống nhau.

### 8.1.1 Tính Bất biến

Hãy tưởng tượng rằng ta muốn nhận diện một vật thể trong ảnh. Có vẻ sẽ hợp lý nếu cho rằng bất cứ phương pháp nào ta sử dụng đều không nên quá quan tâm đến vị trí *chính xác* của vật thể trong ảnh. Lý tưởng nhất, ta có thể học một hệ thống có khả năng tận dụng được kiến thức này bằng một cách nào đó. Lợn thường không bay và máy bay thường không bơi. Tuy nhiên, ta vẫn có thể nhận ra một con lợn đang bay nếu nó xuất hiện. Ý tưởng này được thể hiện rõ rệt trong trò chơi trẻ em ‘Đi tìm Waldo’, một ví dụ được miêu tả trong Fig. 8.1.1. Trò chơi này bao gồm một số cảnh hỗn loạn với nhiều hoạt động đan xen và Waldo xuất hiện ở đâu đó trong mỗi cảnh (thường ẩn nấp ở một số vị trí khó ngờ tới). Nhiệm vụ của người chơi là xác định vị trí của anh ta. Mặc dù Waldo có trang phục khá nổi bật, việc này có thể vẫn rất khó khăn do có quá nhiều yếu tố gây nhiễu.



Fig. 8.1.1: Một ảnh trong Walker Books

Quay lại với ảnh, những trực giác mà ta đã thảo luận có thể được cụ thể hóa hơn nữa để thu được một vài nguyên tắc chính trong việc xây dựng mạng nơ-ron cho thị giác máy tính:

1. Ở một khía cạnh nào đó, các hệ thống thị giác nên phản ứng tương tự với cùng một vật thể bất kể vật thể đó xuất hiện ở đâu trong ảnh (tính bất biến tịnh tiến).
2. Ở khía cạnh khác, các hệ thống thị giác nên tập trung vào các khu vực cục bộ và không quan tâm đến bất kỳ thứ gì khác ở xa hơn trong ảnh (tính cục bộ).

Hãy cùng xem cách biểu diễn những điều trên bằng ngôn ngữ toán học.

### 8.1.2 Ràng buộc Perceptron Đa tầng

Trong phần này, ta coi hình ảnh và các tầng ẩn là các mảng hai chiều. Để bắt đầu, hãy tưởng tượng một perceptron đa tầng sẽ như thế nào với đầu vào là ảnh kích thước  $h \times w$  (biểu diễn dưới dạng ma trận trong toán học và mảng hai chiều khi lập trình), và với các biểu diễn ẩn cũng là các ma trận / mảng hai chiều kích thước  $h \times w$ . Đặt  $x[i, j]$  và  $h[i, j]$  lần lượt là điểm ảnh tại vị trí  $(i, j)$  của ảnh và biểu diễn ẩn. Để mỗi nút ẩn trong tổng số  $h \times w$  nút nhận dữ liệu từ tất cả  $h \times w$  đầu vào, ta sẽ chuyển từ việc biểu diễn các tham số bằng ma trận trọng số (như đã thực hiện với perceptron đa tầng trước đây) sang sử dụng các tensor trọng số bốn chiều.

Ta có thể biểu diễn tầng kết nối đầy đủ bằng công thức toán sau:

$$h[i, j] = u[i, j] + \sum_{k, l} W[i, j, k, l] \cdot x[k, l] = u[i, j] + \sum_{a, b} V[i, j, a, b] \cdot x[i + a, j + b]. \quad (8.1.1)$$

Việc chuyển từ  $W$  sang  $V$  hoàn toàn chỉ có mục đích thẩm mĩ (tại thời điểm này) bởi có một sự tương ứng một-một giữa các hệ số trong cả hai tensor. Ta chỉ đơn thuần đặt lại các chỉ số dưới  $(k, l)$  với  $k = i + a$  và  $l = j + b$ . Nói cách khác,  $V[i, j, a, b] = W[i, j, i + a, j + b]$ . Các chỉ số  $a, b$  chạy trên toàn bộ hình ảnh, có thể mang cả giá trị dương và âm. Với bất kỳ vị trí  $(i, j)$  nào ở tầng ẩn, giá trị

biểu diễn ảnh  $h[i, j]$  được tính bằng tổng trọng số của các điểm ảnh nằm xung quanh vị trí  $(i, j)$  của  $x$ , với trọng số là  $V[i, j, a, b]$ .

Bây giờ hãy sử dụng nguyên tắc đầu tiên mà ta đã thiết lập ở trên: *tính bất biến tịnh tiến*. Nguyên tắc này ngụ ý rằng một sự dịch chuyển ở đầu vào  $x$  cũng sẽ tạo ra sự dịch chuyển ở biểu diễn ảnh  $h$ . Điều này chỉ có thể xảy ra nếu  $V$  và  $u$  không phụ thuộc vào  $(i, j)$ , tức  $V[i, j, a, b] = V[a, b]$  và  $u$  là một hằng số. Vì vậy, ta có thể đơn giản hóa định nghĩa của  $h$ .

$$h[i, j] = u + \sum_{a,b} V[a, b] \cdot x[i+a, j+b]. \quad (8.1.2)$$

Đây là một phép tích chập! Ta đang đánh trọng số cho các điểm ảnh  $(i+a, j+b)$  trong vùng lân cận của  $(i, j)$  bằng các hệ số  $V[a, b]$  để thu được giá trị  $h[i, j]$ . Lưu ý rằng  $V[a, b]$  cần ít hệ số hơn hẳn so với  $V[i, j, a, b]$ . Với đầu vào là hình ảnh 1 megapixel (với tối đa 1 triệu hệ số cho mỗi vị trí), lượng tham số của  $V[a, b]$  giảm đi 1 triệu vì không còn phụ thuộc vào vị trí trong ảnh. Ta đã có được tiến triển đáng kể!

Bây giờ hãy sử dụng nguyên tắc thứ hai—*tính cục bộ*. Như trình bày ở trên, giả sử rằng ta không cần thông tin tại các vị trí quá xa  $(i, j)$  để đánh giá những gì đang diễn ra tại  $h[i, j]$ . Điều này có nghĩa là ở các miền giá trị  $|a|, |b| > \Delta$ , ta có thể đặt  $V[a, b] = 0$ . Tương tự, ta có thể đơn giản hóa  $h[i, j]$  như sau

$$h[i, j] = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} V[a, b] \cdot x[i+a, j+b]. \quad (8.1.3)$$

Một cách ngắn gọn, đây chính là biểu diễn toán học của tầng tích chập. Khi vùng cục bộ xung quanh vị trí đang xét (còn được gọi là *vùng tiếp nhận*) nhỏ, sự khác biệt so với mạng kết nối đầy đủ có thể rất lớn. Trước đây ta có thể phải cần hàng tỷ tham số để biểu diễn một tầng duy nhất trong mạng xử lý ảnh, hiện giờ chỉ cần vài trăm. Cái giá phải trả là các đặc trưng sẽ trở nên bất biến tịnh tiến và các tầng chỉ có thể nhận thông tin cục bộ. Toàn bộ quá trình học dựa trên việc áp đặt các thiên kiến quy nạp (*inductive bias*). Khi các thiên kiến đó phù hợp với thực tế, ta sẽ có được các mô hình hoạt động hiệu quả với ít mẫu và khái quát tốt cho dữ liệu chưa gặp. Nhưng tất nhiên, nếu những thiên kiến đó không phù hợp với thực tế, ví dụ như nếu các ảnh không có tính bất biến tịnh tiến, các mô hình có thể sẽ không khái quát tốt.

### 8.1.3 Phép Tích chập

Hãy cùng xem qua lý do tại sao toán tử trên được gọi là *tích chập*. Trong toán học, phép tích chập giữa hai hàm số  $f, g : \mathbb{R}^d \rightarrow R$  được định nghĩa như sau

$$[f \circledast g](x) = \int_{\mathbb{R}^d} f(z)g(x-z)dz. \quad (8.1.4)$$

Trong phép toán này, ta đo lường sự chồng chéo giữa  $f$  và  $g$  khi  $g$  được dịch chuyển một khoảng  $x$  và “bị lật lại”. Đối với các đối tượng rời rạc, phép tích phân trở thành phép lấy tổng. Chẳng hạn, đối với các vector được định nghĩa trên  $\ell_2$ , là tập các vector vô hạn chiều có tổng bình phương hội tụ, với chỉ số chạy trên  $\mathbb{Z}$ , ta có phép tích chập sau:

$$[f \circledast g](i) = \sum_a f(a)g(i-a). \quad (8.1.5)$$

Đối với mảng hai chiều, ta có một tổng tương ứng với các chỉ số  $(i, j)$  cho  $f$  và  $(i - a, j - b)$  cho  $g$ . Tổng này nhìn gần giống với định nghĩa tàng tích chập ở trên, nhưng với một khác biệt lớn. Thay vì  $(i + a, j + b)$ , ta lại sử dụng hiệu. Tuy nhiên, lưu ý rằng sự khác biệt này không phải vấn đề lớn vì ta luôn có thể chuyển về ký hiệu của phép tích chập bằng cách sử dụng  $\tilde{V}[a, b] = V[-a, -b]$  để có  $h = x \circledast \tilde{V}$ . Cũng lưu ý rằng định nghĩa ban đầu thực ra là của phép toán *tương quan chéo*. Ta sẽ quay trở lại phép toán này trong phần tiếp theo.

#### 8.1.4 Xem lại ví dụ về Waldo

Hãy cùng xem việc xây dựng một bộ phát hiện Waldo cài tiến sẽ trông như thế nào. Tàng tích chập chọn các cửa sổ có kích thước cho sẵn và đánh trọng số cường độ dựa theo mặt nạ  $V$ , như được minh họa trong Fig. 8.1.2. Ta hy vọng rằng ở đâu có “tính Waldo” cao nhất, các tầng kích hoạt ẩn cũng sẽ có cao điểm ở đó.



Fig. 8.1.2: Tìm Waldo.

Chỉ có một vấn đề với cách tiếp cận này là cho đến nay ta đã vô tư bỏ qua việc hình ảnh bao gồm 3 kênh màu: đỏ, xanh lá cây và xanh dương. Trong thực tế, hình ảnh không hẳn là các đối tượng hai chiều mà là một tensor bậc ba, ví dụ tensor với kích thước  $1024 \times 1024 \times 3$  điểm ảnh. Chỉ có hai trong số các trục này chứa mối quan hệ về mặt không gian, trong khi trục thứ ba có thể được coi như là một biểu diễn đa chiều *cho từng vị trí điểm ảnh*.

Do đó, ta phải truy cập  $\mathbf{x}$  dưới dạng  $x[i, j, k]$ . Mặt nạ tích chập phải thích ứng cho phù hợp. Thay vì  $V[a, b]$  bây giờ ta có  $V[a, b, c]$ .

Hơn nữa, tương tự như việc đầu vào là các tensor bậc ba, việc xây dựng các biểu diễn ẩn là các tensor bậc ba tương ứng hoá ra cũng là một ý tưởng hay. Nói cách khác, thay vì chỉ có một biểu diễn 1D tương ứng với từng vị trí không gian, ta muốn có một biểu diễn ẩn đa chiều tương ứng với từng vị trí không gian. Ta có thể coi các biểu diễn ẩn như được cấu thành từ các lưới hai chiều xếp chồng lên nhau. Đôi khi chúng được gọi là *kênh* (*channel*) hoặc *ánh xạ đặc trưng* (*feature map*). Theo trực giác, bạn có thể tưởng tượng rằng ở các tầng thấp hơn, một số kênh tập trung vào việc nhận diện cạnh trong khi các kênh khác đảm nhiệm việc nhận diện kết cấu, v.v. Để hỗ trợ đa kênh ở cả đầu vào và kích hoạt ẩn, ta có thể thêm tọa độ thứ tư vào  $V : V[a, b, c, d]$ . Từ mọi điều trên, ta có:

$$h[i, j, k] = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_c V[a, b, c, k] \cdot x[i + a, j + b, c]. \quad (8.1.6)$$

Đây là định nghĩa của một tầng mạng nơ-ron tích chập. Vẫn còn nhiều phép toán mà ta cần phải giải quyết. Chẳng hạn, ta cần tìm ra cách kết hợp tất cả các giá trị kích hoạt thành một đầu ra duy nhất (ví dụ đầu ra cho: có Waldo trong ảnh không). Ta cũng cần quyết định cách tính toán mọi thứ một cách hiệu quả, cách kết hợp các tầng với nhau và liệu có nên sử dụng thật nhiều tầng hép hay chỉ một vài tầng rộng. Tất cả những điều này sẽ được giải quyết trong phần còn lại của chương.

### 8.1.5 Tóm tắt

- Tính bất biến tịnh tiến của hình ảnh ngũ ý rằng tất cả các mảng nhỏ trong một tấm ảnh đều được xử lý theo cùng một cách.
- Tính cục bộ có nghĩa là chỉ một vùng lân cận nhỏ các điểm ảnh sẽ được sử dụng cho việc tính toán.
- Các kênh ở đầu vào và đầu ra cho phép việc phân tích các đặc trưng trở nên ý nghĩa hơn.

### 8.1.6 Bài tập

1. Giả sử rằng kích thước của mặt nạ tích chập có  $\Delta = 0$ . Chứng minh rằng trong trường hợp này, mặt nạ tích chập xây dựng một MLP độc lập cho mỗi một tập kênh.
2. Tại sao tính bất biến tịnh tiến có thể không phải là một ý tưởng tốt? Việc lợn biết bay là có hợp lý không?
3. Điều gì xảy ra ở viền của một tấm ảnh?
4. Hãy suy ra một tầng tích chập tương tự cho âm thanh.
5. Vấn đề gì sẽ xảy ra khi áp dụng các suy luận trên cho văn bản? Gợi ý: cấu trúc của ngôn ngữ là gì?
6. Chứng minh rằng  $f \circledast g = g \circledast f$ .

### 8.1.7 Thảo luận

- Tiếng Anh<sup>143</sup>
- Tiếng Việt<sup>144</sup>

### 8.1.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Phạm Minh Đức

<sup>143</sup> <https://discuss.mxnet.io/t/2348>

<sup>144</sup> <https://forum.machinelearningcoban.com/c/d21>

- Phạm Hồng Vinh
- Nguyễn Văn Cường

## 8.2 Phép Tích chập cho Ảnh

Giờ chúng ta đã hiểu cách các tầng tích chập hoạt động trên lý thuyết, hãy xem chúng hoạt động trong thực tế như thế nào. Dựa vào ý tưởng mạng nơ-ron tích chập là kiến trúc hiệu quả để khám phá cấu trúc của dữ liệu ảnh, chúng tôi vẫn sẽ sử dụng loại dữ liệu này khi lấy ví dụ.

### 8.2.1 Toán tử Tương quan Chéo

Như ta đã biết, tầng *tích chập* là cái tên có phần không chính xác, vì phép toán mà chúng biểu diễn là phép tương quan chéo (*cross correlation*). Trong một tầng tích chập, một mảng đầu vào và một mảng *hạt nhân tương quan* được kết hợp để tạo ra mảng đầu ra bằng phép toán tương quan chéo. Hãy tạm thời bỏ qua chiều kênh và xem phép toán này hoạt động như thế nào với dữ liệu và biểu diễn ẩn hai chiều. Trong Fig. 8.2.1, đầu vào là một mảng hai chiều với chiều dài 3 và chiều rộng 3. Ta ký hiệu kích thước của mảng là  $3 \times 3$  hoặc  $(3, 3)$ . Chiều dài và chiều rộng của hạt nhân đều là 2. Chú ý rằng trong cộng đồng nghiên cứu học sâu, mảng này còn có thể được gọi là *hạt nhân tích chập, bộ lọc* hay đơn thuần là *trọng số* của tầng. Kích thước của cửa sổ hạt nhân là chiều dài và chiều rộng của hạt nhân (ở đây là  $2 \times 2$ ).

Đầu vào	Bộ lọc	Đầu ra																			
<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>3</td><td>4</td><td>5</td></tr> <tr><td>6</td><td>7</td><td>8</td></tr> </table>	0	1	2	3	4	5	6	7	8	$*$	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>1</td></tr> <tr><td>2</td><td>3</td></tr> </table>	0	1	2	3	$=$	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>19</td><td>25</td></tr> <tr><td>37</td><td>43</td></tr> </table>	19	25	37	43
0	1	2																			
3	4	5																			
6	7	8																			
0	1																				
2	3																				
19	25																				
37	43																				

Fig. 8.2.1: Phép tương quan chéo hai chiều. Các phần được tô màu là phần tử đầu tiên của đầu ra cùng với các phần tử của mảng đầu vào và mảng hạt nhân được sử dụng trong phép toán:  $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$ .

Trong phép tương quan chéo hai chiều, ta bắt đầu với cửa sổ tích chập đặt tại vị trí góc trên bên trái của mảng đầu vào và di chuyển cửa sổ này từ trái sang phải và từ trên xuống dưới. Khi cửa sổ tích chập được đẩy tới một vị trí nhất định, mảng con đầu vào nằm trong cửa sổ đó và mảng hạt nhân được nhân theo từng phần tử, rồi sau đó ta lấy tổng các phần tử trong mảng kết quả để có được một giá trị số vô hướng duy nhất. Giá trị này được ghi vào mảng đầu ra tại vị trí tương ứng. Ở đây, mảng đầu ra có chiều dài 2 và chiều rộng 2, với bốn phần tử được tính bằng phép tương quan chéo hai chiều:

$$\begin{aligned}
 0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19, \\
 1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25, \\
 3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37, \\
 4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.
 \end{aligned} \tag{8.2.1}$$

Lưu ý rằng theo mỗi trục, kích thước đầu ra *nhỏ hơn* một chút so với đầu vào. Bởi vì hạt nhân có chiều dài và chiều rộng lớn hơn một, ta chỉ có thể tính độ tương quan chéo cho những vị trí mà ở đó hạt nhân nằm hoàn toàn bên trong ảnh, kích thước đầu ra được tính bằng cách lấy đầu vào

$H \times W$  trừ kích thước của bộ lọc tích chập  $h \times w$  bằng  $(H - h + 1) \times (W - w + 1)$ . Điều này xảy ra vì ta cần đủ không gian để ‘dịch chuyển’ hạt nhân tích chập qua tấm hình (sau này ta sẽ xem làm thế nào để có thể giữ nguyên kích thước bằng cách thêm các số không vào xung quanh biên của hình ảnh sao cho có đủ không gian để dịch chuyển hạt nhân). Kế tiếp, ta lập trình quá trình ở trên trong hàm corr2d. Hàm này nhận mảng đầu vào X với mảng hạt nhân K và trả về mảng đầu ra Y.

```
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = np.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
```

Ta có thể xây dựng mảng đầu vào X và mảng hạt nhân K như hình trên để kiểm tra lại kết quả của cách lập trình phép toán tương quan chéo hai chiều vừa rồi.

```
X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
K = np.array([[0, 1], [2, 3]])
corr2d(X, K)
```

## 8.2.2 Tầng Tích chập

Tầng tích chập thực hiện phép toán tương quan chéo giữa đầu vào và hạt nhân, sau đó cộng thêm một hệ số điều chỉnh để có được đầu ra. Hai tham số của tầng tích chập là hạt nhân và hệ số điều chỉnh. Khi huấn luyện mô hình chứa các tầng tích chập, ta thường khởi tạo hạt nhân ngẫu nhiên, giống như cách ta làm với tầng kết nối đầy đủ.

Bây giờ ta đã sẵn sàng lập trình một tầng tích chập hai chiều dựa vào hàm corr2d ta vừa định nghĩa ở trên. Trong hàm khởi tạo `__init__`, ta khai báo hai tham số của mô hình `weight` và `bias`. Hàm tính lượt truyền xuôi `forward` gọi hàm `corr2d` và cộng thêm hệ số điều chỉnh. Cũng giống cách gọi phép tương quan chéo  $h \times w$ , ta cũng gọi các tầng tích chập là phép tích chập  $h \times w$ .

```
class Conv2D(nn.Block):
    def __init__(self, kernel_size, **kwargs):
        super(Conv2D, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=kernel_size)
        self.bias = self.params.get('bias', shape=(1,))

    def forward(self, x):
        return corr2d(x, self.weight.data()) + self.bias.data()
```

### 8.2.3 Phát hiện Biên của Vật thể trong Ảnh

Hãy quan sát một ứng dụng đơn giản của tầng tích chập: phát hiện đường biên của một vật thể trong một bức ảnh bằng cách xác định vị trí các điểm ảnh thay đổi. Đầu tiên, ta dựng một ‘bức ảnh’ có kích thước là  $6 \times 8$  điểm ảnh. Bốn cột ở giữa có màu đen (giá trị 0) và các cột còn lại có màu trắng (giá trị 1).

```
X = np.ones((6, 8))
X[:, 2:6] = 0
X
```

Sau đó, ta tạo một hạt nhân  $K$  có chiều cao bằng 1 và chiều rộng bằng 2. Khi thực hiện phép tương quan chéo với đầu vào, nếu hai phần tử cạnh nhau theo chiều ngang có giá trị giống nhau thì đầu ra sẽ bằng 0, còn lại đầu ra sẽ khác không.

```
K = np.array([[1, -1]])
```

Ta đã sẵn sàng thực hiện phép tương quan chéo với các đối số  $X$  (đầu vào) và  $K$  (hạt nhân). Bạn có thể thấy rằng các vị trí biên trắng đổi thành đen có giá trị 1, còn các vị trí biên đen đổi thành trắng có giá trị -1. Các vị trí còn lại của đầu ra có giá trị 0.

```
Y = corr2d(X, K)
Y
```

Bây giờ hãy áp dụng hạt nhân này cho chuyển vị của ma trận điểm ảnh. Như kỳ vọng, giá trị tương quan chéo bằng không. Hạt nhân  $K$  chỉ có thể phát hiện biên dọc.

```
corr2d(X.T, K)
```

### 8.2.4 Học một Bộ lọc

Việc thiết kế bộ phát hiện biên bằng sai phân hữu hạn  $[1, -1]$  thì khá gọn gàng nếu ta biết chính xác đây là những gì cần làm. Tuy nhiên, khi xét tới các bộ lọc lớn hơn và các tầng tích chập liên tiếp, việc chỉ định chính xác mỗi bộ lọc cần làm gì một cách thủ công là bất khả thi.

Bây giờ ta hãy xem liệu có thể học một bộ lọc có khả năng tạo ra  $Y$  từ  $X$  chỉ từ các cặp (đầu vào, đầu ra) hay không. Đầu tiên chúng ta xây dựng một tầng tích chập và khởi tạo một mảng ngẫu nhiên làm bộ lọc. Tiếp theo, trong mỗi lần lặp, ta sẽ sử dụng bình phương sai số để so sánh  $Y$  và đầu ra của tầng tích chập, sau đó tính toán gradient để cập nhật trọng số. Để đơn giản, trong tầng tích chập này, ta sẽ bỏ qua hệ số điều chỉnh.

Trước đây ta đã tự xây dựng lớp Conv2D. Tuy nhiên, do ta sử dụng các phép gán một phần tử, Gluon sẽ gặp một số khó khăn khi tính gradient. Thay vào đó, ta sử dụng lớp Conv2D có sẵn của Gluon như sau.

```
# Construct a convolutional layer with 1 output channel
# (channels will be introduced in the following section)
# and a kernel array shape of (1, 2)
conv2d = nn.Conv2D(1, kernel_size=(1, 2))
conv2d.initialize()
```

(continues on next page)

```
# 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)

for i in range(10):
    with autograd.record():
        Y_hat = conv2d(X)
        l = (Y_hat - Y) ** 2
    l.backward()
    # For the sake of simplicity, we ignore the bias here
    conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
    if (i + 1) % 2 == 0:
        print('batch %d, loss %.3f' % (i + 1, l.sum()))
```

Có thể thấy sai số đã giảm xuống còn khá nhỏ sau 10 lần lặp. Bây giờ hãy xem mảng bộ lọc đã học được.

```
conv2d.weight.data().reshape(1, 2)
```

Thật vậy, mảng bộ lọc học được rất gần với mảng bộ lọc K mà ta tự định nghĩa trước đó.

### 8.2.5 Tương quan Chéo và Tích chập

Hãy nhớ lại kiến thức của phần trước về mối liên hệ giữa phép tương quan chéo và tích chập. Trong hình trên, ta dễ dàng nhận thấy điều này. Đơn giản chỉ cần lật bộ lọc từ góc dưới cùng bên trái lên góc trên cùng bên phải. Trong trường hợp này, chỉ số trong phép lấy tổng được đảo ngược, nhưng ta vẫn thu được kết quả tương tự. Để thống nhất với các thuật ngữ tiêu chuẩn trong tài liệu học sâu, ta sẽ tiếp tục đề cập đến phép tương quan chéo như là phép tích chập, mặc dù đúng ra chúng hơi khác nhau một chút.

### 8.2.6 Tóm tắt

- Về cốt lõi, phần tính toán của tầng tích chập hai chiều là phép tương quan chéo hai chiều. Ở dạng đơn giản nhất, phép tương quan chéo thao tác trên dữ liệu đầu vào hai chiều và bộ lọc, sau đó cộng thêm hệ số điều chỉnh.
- Chúng ta có thể thiết kế bộ lọc để phát hiện các biên trong ảnh.
- Chúng ta có thể học các tham số của bộ lọc từ dữ liệu.

### 8.2.7 Bài tập

1. Xây dựng hình ảnh  $X$  với các cạnh chéo.
  - Điều gì xảy ra nếu bạn áp dụng bộ lọc  $K$  lên nó?
  - Điều gì xảy ra nếu bạn chuyển vị  $X$ ?
  - Điều gì xảy ra nếu bạn chuyển vị  $K$ ?
2. Khi thử tự động tìm gradient cho lớp Conv2D mà ta đã tạo, bạn thấy loại thông báo lỗi nào?
3. Làm thế nào để bạn biểu diễn một phép tính tương quan chéo như là một phép nhân ma trận bằng cách thay đổi các mảng đầu vào và mảng bộ lọc?
4. Hãy thiết kế thủ công một số bộ lọc sau.
  - Bộ lọc để tính đạo hàm bậc hai có dạng như thế nào?
  - Bộ lọc của toán tử Laplace là gì?
  - Bộ lọc của phép tích phân là gì?
  - Kích thước tối thiểu của bộ lọc để có được đạo hàm bậc  $d$  là bao nhiêu?

### 8.2.8 Thảo luận

- Tiếng Anh<sup>145</sup>
- Tiếng Việt<sup>146</sup>

### 8.2.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Lý Phi Long
- Phạm Minh Đức
- Trần Yến Thy

<sup>145</sup> <https://discuss.mxnet.io/t/2349>

<sup>146</sup> <https://forum.machinelearningcoban.com/c/d21>

## 8.3 Đệm và Sai Bước

Trong ví dụ trước, đầu vào có cả chiều dài và chiều rộng cùng bằng 3, cửa sổ hạt nhân tích chập có cả chiều dài và chiều rộng cùng bằng 2, nên ta thu được biểu diễn đầu ra có kích thước  $2 \times 2$ . Nói chung, giả sử kích thước của đầu vào là  $n_h \times n_w$  và kích thước của cửa sổ hạt nhân tích chập là  $k_h \times k_w$ , kích thước của đầu ra sẽ là:

$$(n_h - k_h + 1) \times (n_w - k_w + 1). \quad (8.3.1)$$

Do đó, kích thước của đầu ra tầng tích chập được xác định bởi kích thước đầu vào và kích thước cửa sổ hạt nhân tích chập.

Trong vài trường hợp, ta sẽ kết hợp thêm các kỹ thuật khác cũng có ảnh hưởng tới kích thước của đầu ra, như thêm phần đệm và phép tích chập sai bước. Lưu ý rằng vì các hạt nhân thường có chiều rộng và chiều cao lớn hơn 1 nên sau khi áp dụng nhiều phép tích chập liên tiếp, đầu ra thường có kích thước nhỏ hơn đáng kể so với đầu vào. Nếu ta bắt đầu với một ảnh có  $240 \times 240$  điểm ảnh và áp dụng 10 tầng tích chập có kích thước  $5 \times 5$  thì kích thước ảnh này sẽ giảm xuống  $200 \times 200$  điểm ảnh, 30% của ảnh sẽ bị cắt bỏ và mọi thông tin có ích trên viền của ảnh gốc sẽ bị xóa sạch. **Đệm** là công cụ phổ biến nhất để xử lý vấn đề này.

Trong những trường hợp khác, ta có thể muốn giảm đáng kể kích thước ảnh, ví dụ như khi độ phân giải của đầu vào quá cao. **Phép tích chập sai bước (Strided convolution)** là một kỹ thuật phổ biến có thể giúp ích trong trường hợp này.

### 8.3.1 Đệm

Như mô tả ở trên, một vấn đề rắc rối khi áp dụng các tầng tích chập là việc chúng ta có thể mất một số điểm ảnh trên biên của ảnh. Vì chúng ta thường sử dụng các hạt nhân nhỏ, với một phép tích chập ta có thể chỉ mất một ít điểm ảnh, tuy nhiên sự mất mát này có thể tích lũy dần khi ta thực hiện qua nhiều tầng tích chập liên tiếp. Một giải pháp đơn giản cho vấn đề này là chèn thêm các điểm ảnh xung quanh đường biên trên bức ảnh đầu vào, nhờ đó làm tăng kích thước sử dụng của bức ảnh. Thông thường, chúng ta thiết lập các giá trị của các điểm ảnh thêm vào là 0. Trong Fig. 8.3.1, ta đệm một đầu vào  $3 \times 3$ , làm tăng kích thước lên thành  $5 \times 5$ . Đầu ra tương ứng sẽ tăng lên thành một ma trận  $4 \times 4$ .

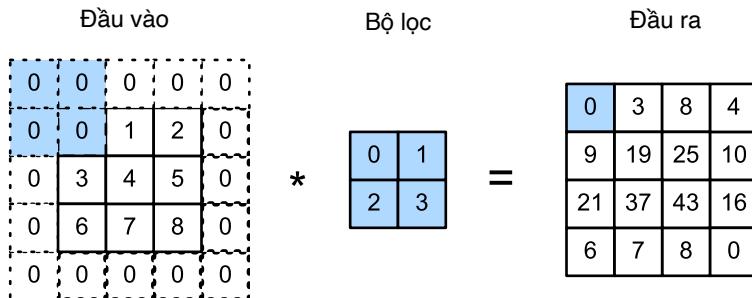


Fig. 8.3.1: Tương quan chéo hai chiều khi thực hiện đệm. Phần tử đệm là các phần tử của mảng đầu vào và hạt nhân được sử dụng để tính phần tử đầu ra thứ nhất:  $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$ .

Nhìn chung nếu chúng ta chèn thêm tổng cộng  $p_h$  hàng đệm (phân nửa ở phía trên và phân nửa ở phía dưới) và  $p_w$  cột đệm (phân nửa bên trái và phân nửa bên phải), kích thước đầu ra sẽ là:

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1). \quad (8.3.2)$$

Điều này có nghĩa là chiều cao và chiều rộng của đầu ra sẽ tăng thêm lần lượt là  $p_h$  và  $p_w$ .

Trong nhiều trường hợp, ta sẽ muốn thiết lập  $p_h = k_h - 1$  và  $p_w = w_k - 1$  để đầu vào và đầu ra có cùng chiều dài và chiều rộng. Điều này sẽ giúp việc dự đoán kích thước đầu ra của mỗi tầng dễ dàng hơn khi ta xây dựng mạng. Giả sử  $k_h$  ở đây chẵn, ta sẽ chèn  $p_h/2$  hàng ở cả phía trên và phía dưới. Nếu  $k_h$  lẻ, ta có thể chèn  $\lceil p_h/2 \rceil$  hàng ở phía trên của đầu vào và  $\lfloor p_h/2 \rfloor$  hàng cho phía dưới. Chúng ta cũng thực hiện chèn cả hai bên của chiều ngang tương tự như vậy.

Các mạng nơ-ron tích chập thường sử dụng các hạt nhân tích chập với chiều dài và chiều rộng là số lẻ, như 1, 3, 5 hay 7. Việc chọn hạt nhân có kích thước lẻ giúp chúng ta bảo toàn được các chiều không gian khi thêm cùng số hàng đệm cho cạnh trên và dưới, và thêm cùng số cột đệm cho cạnh trái và phải.

Hơn nữa, việc sử dụng bộ lọc kích thước lẻ cùng đệm để giữ nguyên số chiều mang lại một lợi ích khác. Với mảng hai chiều X bất kỳ, khi kích thước bộ lọc lẻ và số hàng và số cột đệm bằng nhau, thu được đầu ra có cùng chiều dài và chiều rộng với đầu vào, ta sẽ biết chắc chắn rằng mỗi phần tử đầu ra  $Y[i, j]$  được tính bằng phép tương quan chéo giữa đầu vào và hạt nhân tích chập có tâm nằm tại  $X[i, j]$ .

Trong ví dụ dưới, chúng ta tạo một tầng tích chập hai chiều với chiều dài và chiều rộng 3 và đệm 1 điểm ảnh vào viền các cạnh. Với đầu vào có chiều dài và chiều rộng là 8, ta thấy rằng chiều dài và chiều rộng đầu ra cũng là 8.

```
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# For convenience, we define a 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):
    conv2d.initialize()
    # (1, 1) indicates that the batch size and the number of channels
    # (described in later chapters) are both 1
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    # Exclude the first two dimensions that do not interest us: batch and
    # channel
    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, kernel_size=3, padding=1)
X = np.random.uniform(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Khi chiều dài và chiều rộng của hạt nhân tích chập khác nhau, chúng ta có thể chỉnh chiều dài và chiều rộng khác nhau cho phần đệm để đầu vào và đầu ra có cùng kích thước.

```
# Here, we use a convolution kernel with a height of 5 and a width of 3. The
# padding numbers on both sides of the height and width are 2 and 1,
# respectively
conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
comp_conv2d(conv2d, X).shape
```

### 8.3.2 Sải bước

Khi thực hiện phép tương quan chéo, ta bắt đầu với cửa sổ tích chập tại góc trên bên trái của mảng đầu vào, rồi di chuyển sang phải và xuống dưới qua tất cả các vị trí. Trong các ví dụ trước, ta mặc định di chuyển qua một điểm ảnh mỗi lần. Tuy nhiên, có những lúc để tăng hiệu suất tính toán hoặc vì muốn giảm kích thước của ảnh, ta di chuyển cửa sổ tích chập nhiều hơn một điểm ảnh mỗi lần, bỏ qua các vị trí ở giữa.

Ta gọi số hàng và cột di chuyển qua mỗi lần là *sải bước* (*stride*). Cho đến giờ, chúng ta sử dụng sải bước 1 cho cả chiều dài và chiều rộng. Đôi lúc, chúng ta có thể muốn sử dụng sải bước lớn hơn. Fig. 8.3.2 biểu diễn phép tương quan chéo hai chiều với sải bước 3 theo chiều dọc và 2 theo chiều ngang. Có thể thấy rằng khi tính giá trị phần tử thứ hai của cột đầu tiên, cửa sổ tích chập di chuyển xuống ba hàng. Cửa sổ này di chuyển sang phải hai cột khi tính giá trị phần tử thứ hai của hàng đầu tiên. Khi cửa sổ di chuyển sang phải ba cột ở đầu vào, giá trị đầu ra không tồn tại vì các phần tử đầu vào không lấp đầy cửa sổ (trừ khi ta thêm một cột đệm).

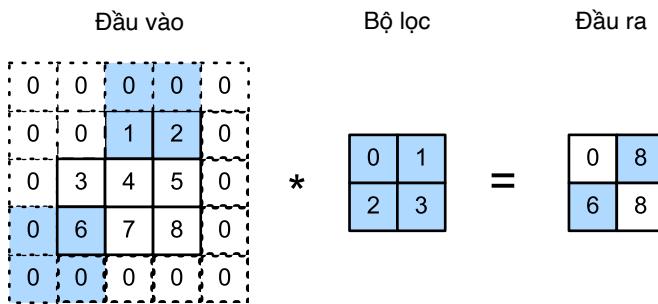


Fig. 8.3.2: Phép tương quan chéo với sải bước 3 theo chiều dài và 2 theo chiều rộng. Phần tô đậm là các phần tử đầu ra, các phần tử đầu vào và bộ lọc được sử dụng để tính các đầu ra này:  $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$ ,  $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$ .

Nhìn chung, khi sải bước theo chiều cao là  $s_h$  và sải bước theo chiều rộng là  $s_w$ , kích thước đầu ra là:

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor. \quad (8.3.3)$$

Nếu đặt  $p_h = k_h - 1$  và  $p_w = k_w - 1$ , kích thước đầu ra sẽ được thu gọn thành  $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$ . Hơn nữa, nếu chiều cao và chiều rộng của đầu vào chia hết cho sải bước theo chiều cao và chiều rộng tương ứng thì kích thước đầu ra sẽ là  $(n_h/s_h) \times (n_w/s_w)$ .

Dưới đây, chúng ta đặt sải bước cho cả chiều cao và chiều rộng là 2, do đó chiều cao và chiều rộng của đầu ra bằng một nửa chiều cao và chiều rộng của đầu vào.

```
conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
comp_conv2d(conv2d, X).shape
```

Tiếp theo, chúng ta sẽ xem xét một ví dụ phức tạp hơn một chút.

```
conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
comp_conv2d(conv2d, X).shape
```

Để đơn giản hóa vấn đề, khi phần đệm theo chiều cao và chiều rộng của đầu vào lần lượt là  $p_h$  và  $p_w$ , chúng ta sẽ ký hiệu phần đệm là  $(p_h, p_w)$ . Ở trường hợp đặc biệt khi  $p_h = p_w = p$ , ta ký hiệu

phần đệm là  $p$ . Khi sai bước trên chiều cao và chiều rộng lần lượt là  $s_h$  và  $s_w$ , chúng ta kí hiệu sai bước là  $(s_h, s_w)$ . Ở trường hợp đặc biệt khi  $s_h = s_w = s$ , ta kí hiệu sai bước là  $s$ . Mặc định, phần đệm là 0 và sai bước là 1. Trên thực tế, ít khi chúng ta sử dụng các giá trị khác nhau cho sai bước hoặc phần đệm, tức ta thường đặt  $p_h = p_w$  và  $s_h = s_w$ .

### 8.3.3 Tóm tắt

- Phần đệm có thể tăng chiều cao vào chiều rộng của đầu ra. Nó thường được sử dụng để đầu ra có cùng kích thước với đầu vào.
- Sai bước có thể giảm độ phân giải của đầu ra, ví dụ giảm chiều cao và chiều rộng của đầu ra xuống  $1/n$  chiều cao và chiều rộng của đầu vào ( $n$  là một số nguyên lớn hơn 1).
- Đệm và sai bước có thể được dùng để điều chỉnh kích thước chiều của dữ liệu một cách hiệu quả.

### 8.3.4 Bài tập

1. Trong ví dụ cuối của phần này, tính kích thước đầu ra bằng công thức và xác nhận lại với kết quả khi chạy mã nguồn.
2. Thử các cách kết hợp đệm và sai bước khác trong các ví dụ ở phần này.
3. Với các tín hiệu âm thanh, sai bước bằng 2 tương ứng với điều gì?
4. Có những lợi ích nào về mặt tính toán khi sử dụng sai bước lớn hơn 1?

### 8.3.5 Thảo luận

- Tiếng Anh<sup>147</sup>
- Tiếng Việt<sup>148</sup>

### 8.3.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Thành Hưng

<sup>147</sup> <https://discuss.mxnet.io/t/2350>

<sup>148</sup> <https://forum.machinelearningcoban.com/c/d21>

## 8.4 Đa kênh Đầu vào và Đầu ra

Mặc dù chúng ta đã mô tả mỗi tấm ảnh được tạo nên bởi nhiều kênh (*channel*) (cụ thể, ảnh màu sử dụng hệ màu RGB tiêu chuẩn với các kênh riêng biệt thể hiện lượng màu đỏ, xanh lá và xanh dương), nhưng cho đến lúc này, ta vẫn đơn giản hóa tất cả các ví dụ tính toán với chỉ một kênh đầu vào và một kênh đầu ra. Điều đó đã cho phép chúng ta coi các đầu vào, các bộ lọc tích chập và các đầu ra như các mảng hai chiều.

Khi chúng ta thêm các kênh vào hỗn hợp ấy, đầu vào cùng với các lớp biểu diễn ẩn của ta trở thành các mảng ba chiều. Chẳng hạn, mỗi ảnh RGB đầu vào có dạng  $3 \times h \times w$ . Ta xem trực này là chiều kênh, có kích thước là 3. Trong phần này, ta sẽ quan sát sâu hơn vào các bộ lọc tích chập với đầu vào và đầu ra đa kênh.

### 8.4.1 Đa kênh Đầu vào

Khi dữ liệu đầu vào có nhiều kênh, ta cần xây dựng một bộ lọc tích chập với cùng số kênh đầu vào như dữ liệu nhập, để nó có thể thực hiện tính tương quan chéo với dữ liệu này. Giả sử số kênh dữ liệu đầu vào là  $c_i$ , ta sẽ cần số kênh đầu vào của bộ lọc tích chập là  $c_i$ . Nếu kích thước cửa sổ của bộ lọc tích chập là  $k_h \times k_w$ , thì khi  $c_i = 1$ , ta có thể xem bộ lọc tích chập này đơn giản là một mảng hai chiều có kích thước  $k_h \times k_w$ .

Tuy nhiên, khi  $c_i > 1$ , chúng ta cần một bộ lọc chứa mảng có kích thước  $k_h \times k_w$  cho mỗi kênh của đầu vào. Gộp  $c_i$  mảng này lại ta được một bộ lọc tích chập kích thước  $c_i \times k_h \times k_w$ . Vì đầu vào và bộ lọc đều có  $c_i$  kênh, ta có thể thực hiện phép tương quan chéo trên từng cặp mảng hai chiều của đầu vào và bộ lọc cho mỗi kênh, rồi cộng kết quả của  $c_i$  kênh lại để tạo ra một mảng hai chiều. Đây là kết quả của phép tương quan chéo hai chiều giữa dữ liệu đầu vào đa kênh và kênh bộ lọc tích chập *đa đầu vào*.

Trong Fig. 8.4.1 minh họa một ví dụ về phép tương quan chéo hai chiều với hai kênh đầu vào. Phần tô đậm là phần tử đầu ra đầu tiên cùng các phần tử của mảng đầu vào và bộ lọc được sử dụng trong phép tính đó:  $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ .

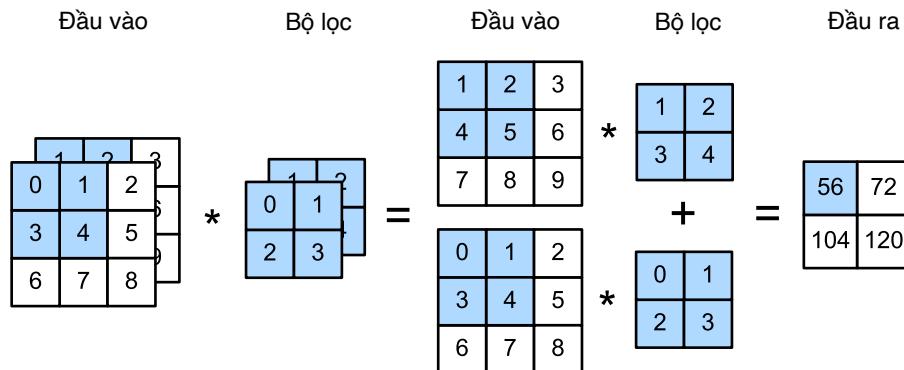


Fig. 8.4.1: Phép tính tương quan chéo với hai kênh đầu vào. Phần tô đậm là phần tử đầu ra đầu tiên cùng các phần tử của mảng đầu vào và bộ lọc được sử dụng trong phép tính đó:  $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$ .

Để thực sự hiểu được những gì đang xảy ra ở đây, chúng ta có thể tự lập trình phép toán tương quan chéo với nhiều kênh đầu vào. Chú ý rằng tất cả những gì chúng ta đang làm là thực hiện một phép tương quan chéo trên mỗi kênh rồi cộng các kết quả lại bằng hàm `add_n`.

```

from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

def corr2d_multi_in(X, K):
    # First, traverse along the 0th dimension (channel dimension) of X and K.
    # Then, add them together by using * to turn the result list into a
    # positional argument of the add_n function
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))

```

Ta có thể tạo mảng đầu vào  $X$  và mảng bộ lọc  $K$  tương ứng với các giá trị trong hình trên để kiểm chứng kết quả đầu ra.

```

X = np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
              [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
K = np.array([[0, 1], [2, 3]], [[1, 2], [3, 4]]]

corr2d_multi_in(X, K)

```

#### 8.4.2 Đa kênh Đầu ra

Cho đến nay, bất kể số lượng kênh đầu vào là bao nhiêu thì ta vẫn luôn kết thúc với chỉ một kênh đầu ra. Tuy nhiên, như đã thảo luận trước đây, hóa ra việc có nhiều kênh ở mỗi tầng là rất cần thiết. Trong các kiến trúc mạng nơ-ron phổ biến nhất, ta thường tăng kích thước chiều kênh khi tiến sâu hơn trong mạng, đồng thời giảm độ phân giải không gian để đánh đổi với *chiều kênh* sâu hơn này. Theo trực giác, ta có thể xem mỗi kênh tương ứng với một tập các đặc trưng khác nhau. Nhưng thực tế phức tạp hơn một chút so với cách diễn giải theo trực giác này vì các biểu diễn không được học độc lập mà được tối ưu hóa để có ích khi kết hợp với nhau. Vì vậy, có thể việc phát hiện biên sẽ được học bởi một vài kênh thay vì chỉ một kênh duy nhất.

Đặt  $c_i$  và  $c_o$  lần lượt là số lượng kênh đầu vào và đầu ra,  $k_h$  và  $k_w$  lần lượt là chiều cao và chiều rộng của bộ lọc. Để có được một đầu ra với nhiều kênh, ta có thể tạo một mảng bộ lọc có kích thước  $c_i \times k_h \times k_w$  cho mỗi kênh đầu ra. Ta nối chúng lại dựa trên chiều kênh đầu ra đã biết, sao cho kích thước của bộ lọc tích chập là  $c_o \times c_i \times k_h \times k_w$ . Trong các phép tính tương quan chéo, kết quả trên mỗi kênh đầu ra được tính từ bộ lọc tích chập tương ứng với kênh đầu ra đó và lấy đầu vào từ tất cả các kênh trong mảng đầu vào.

Ta lập trình một hàm tương quan chéo để tính đầu ra của nhiều kênh như dưới đây.

```

def corr2d_multi_in_out(X, K):
    # Traverse along the 0th dimension of K, and each time, perform
    # cross-correlation operations with input X. All of the results are merged
    # together using the stack function
    return np.stack([corr2d_multi_in(X, k) for k in K])

```

Ta tạo một bộ lọc tích chập với 3 kênh đầu ra bằng cách nối mảng bộ lọc  $K$  với  $K+1$  (cộng một cho mỗi phần tử trong  $K$ ) và  $K+2$ .

```

K = np.stack((K, K + 1, K + 2))
K.shape

```

Dưới đây, ta thực hiện các phép tính tương quan chéo trên mảng đầu vào  $X$  với mảng bộ lọc  $K$ . Đầu ra sẽ gồm có 3 kênh. Kết quả của kênh đầu tiên khớp với kết quả trước đây khi áp dụng bộ lọc đa kênh đầu vào và một kênh đầu ra lên mảng đầu vào  $X$ .

```
corr2d_multi_in_out(X, K)
```

#### 8.4.3 Tầng Tích chập $1 \times 1$

Thoạt nhìn, một phép tích chập  $1 \times 1$ , tức  $k_h = k_w = 1$ , dường như không có nhiều ý nghĩa. Suy cho cùng, một phép tích chập là để tính toán tương quan giữa các điểm ảnh liền kề. Nhưng rõ ràng một phép tích chập  $1 \times 1$  lại không làm như vậy. Mặc dù vậy, chúng là các phép tính phổ biến đôi khi được sử dụng khi thiết kế các mạng sâu phức tạp. Ta sẽ xem kỹ cách hoạt động của chúng.

Do cửa sổ có kích thước tối thiểu nên so với các tầng tích chập lớn hơn, phép tích chập  $1 \times 1$  mất đi khả năng nhận dạng các khuôn mẫu chứa các tương tác giữa các phần tử liền kề theo chiều cao và chiều rộng. Phép tích chập  $1 \times 1$  chỉ xảy ra trên chiều kênh.

Fig. 8.4.2 biểu diễn phép tính tương quan chéo sử dụng bộ lọc tích chập  $1 \times 1$  với 3 kênh đầu vào và 2 kênh đầu ra. Lưu ý rằng đầu vào và đầu ra có cùng chiều cao và chiều rộng. Mỗi phần tử trong đầu ra là một tổ hợp tuyến tính của các phần tử ở *cùng một vị trí* trong ảnh đầu vào. Bạn có thể xem tầng tích chập  $1 \times 1$  như một tầng kết nối đầy đủ được áp dụng lên mỗi vị trí điểm ảnh đơn lẻ để chuyển đổi  $c_i$  giá trị đầu vào thành  $c_o$  giá trị đầu ra tương ứng. Bởi vì đây vẫn là một tầng tích chập nên các trọng số sẽ được chia sẻ giữa các vị trí điểm ảnh. Do đó, tầng tích chập  $1 \times 1$  cần tới  $c_o \times c_i$  trọng số (cộng thêm các hệ số điều chỉnh).

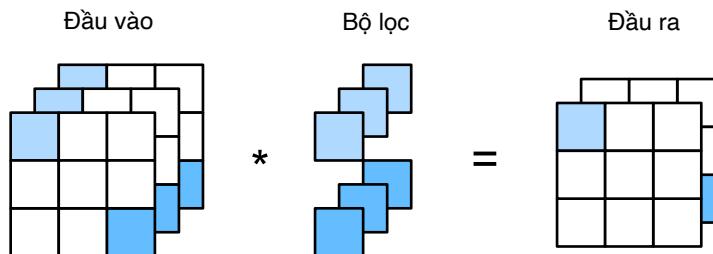


Fig. 8.4.2: Phép tính tương quan chéo sử dụng bộ lọc tích chập  $1 \times 1$  với 3 kênh đầu vào và 2 kênh đầu ra. Các đầu vào và các đầu ra có cùng chiều cao và chiều rộng.

Hãy kiểm tra xem liệu nó có hoạt động trong thực tế: Ta sẽ lập trình một phép tích chập  $1 \times 1$  sử dụng một tầng kết nối đầy đủ. Vấn đề duy nhất là ta cần phải điều chỉnh kích thước dữ liệu trước và sau phép 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)
    Y = np.dot(K, X) # Matrix multiplication in the fully connected layer
    return Y.reshape(c_o, h, w)
```

Khi thực hiện phép tích chập  $1 \times 1$ , hàm bên trên tương đương với hàm tương quan chéo đã được lập trình ở `corr2d_multi_in_out`.

```

X = np.random.uniform(size=(3, 3, 3))
K = np.random.uniform(size=(2, 3, 1, 1))

Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)

np.abs(Y1 - Y2).sum() < 1e-6

```

#### 8.4.4 Tóm tắt

- Ta có thể sử dụng nhiều kênh để mở rộng các tham số mô hình của tầng tích chập.
- Tầng tích chập  $1 \times 1$  khi được áp dụng lên từng điểm ảnh tương đương với tầng kết nối đầy đủ giữa các kênh.
- Tầng tích chập  $1 \times 1$  thường được sử dụng để điều chỉnh số lượng kênh giữa các tầng của mạng và để kiểm soát độ phức tạp của mô hình.

#### 8.4.5 Bài tập

1. Giả sử rằng ta có hai bộ lọc tích chập có kích thước tương ứng là  $k_1$  và  $k_2$  (không có tính phi tuyến ở giữa).
  - Chứng minh rằng kết quả của phép tính có thể được biểu diễn bằng chỉ một phép tích chập.
  - Phép tích chập tương đương này có kích thước là bao nhiêu?
  - Điều ngược lại có đúng không?
2. Giả sử kích thước của đầu vào là  $c_i \times h \times w$  và một bộ lọc tích chập có kích thước  $c_o \times c_i \times k_h \times k_w$ , đồng thời sử dụng đệm ( $p_h, p_w$ ) và sải bước ( $s_h, s_w$ ).
  - Chi phí tính toán (phép nhân và phép cộng) cho lượt truyền xuôi là bao nhiêu?
  - Dung lượng bộ nhớ cho tính toán truyền xuôi là bao nhiêu?
  - Dung lượng bộ nhớ cho tính toán truyền ngược là bao nhiêu?
  - Chi phí tính toán cho lượt lan truyền ngược là bao nhiêu?
3. Số lượng tính toán sẽ tăng lên bao nhiêu lần nếu ta nhân đôi số lượng kênh đầu vào  $c_i$  và số lượng kênh đầu ra  $c_o$ ? Điều gì xảy ra nếu ta gấp đôi phần đệm?
4. Nếu chiều cao và chiều rộng của bộ lọc tích chập là  $k_h = k_w = 1$ , thì độ phức tạp của tính toán truyền xuôi là bao nhiêu?
5. Các biến  $Y1$  và  $Y2$  trong ví dụ cuối cùng của mục này có giống nhau không? Tại sao?
6. Khi cửa sổ tích chập không phải là  $1 \times 1$ , bạn sẽ lập trình các phép tích chập sử dụng phép nhân ma trận như thế nào?

#### 8.4.6 Thảo luận

- Tiếng Anh<sup>149</sup>
- Tiếng Việt<sup>150</sup>

#### 8.4.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Minh Đức
- Nguyễn Duy Du
- Phạm Hồng Vinh

### 8.5 Gộp (Pooling)

Khi xử lý ảnh, ta thường muốn giảm dần độ phân giải không gian của các biểu diễn ẩn, tổng hợp thông tin lại để khi càng đi sâu vào mạng, vùng tiếp nhận (ở đầu vào) ảnh hướng đến mỗi nút ẩn càng lớn.

Nhiệm vụ cuối cùng thường là trả lời một câu hỏi nào đó về toàn bộ tấm ảnh, ví dụ như: *trong ảnh có mèo không?* Vậy nên các nút của tầng cuối cùng thường cần phải chịu ảnh hưởng của toàn bộ đầu vào. Bằng cách dần gộp thông tin lại để tạo ra các ảnh xạ đặc trưng thừa dồn, ta sẽ học được một biểu diễn toàn cục, trong khi vẫn có thể giữ nguyên toàn bộ lợi thế đến từ các tầng tích chập xử lý trung gian.

Hơn nữa, khi phát hiện các đặc trưng cấp thấp như cạnh (được thảo luận tại Section 8.2), ta thường muốn cách biểu diễn này bất biến với phép tịnh tiến trong một chừng mực nào đó. Ví dụ, nếu ta lấy ảnh  $X$  với một ranh giới rõ rệt giữa màu đen và màu trắng và dịch chuyển toàn bộ tấm ảnh sang phải một điểm ảnh, tức  $Z[i, j] = X[i, j+1]$  thì đầu ra cho ảnh mới  $Z$  có thể sẽ khác đi rất nhiều. Đường biên đó và các giá trị kích hoạt sẽ đều dịch chuyển sang một điểm ảnh. Trong thực tế, các vật thể hiếm khi xuất hiện chính xác ở cùng một vị trí. Thậm chí với một chân máy ảnh và một vật thể tĩnh, chuyển động của màn trập vẫn có thể làm rung máy ảnh và dịch chuyển tất cả đi một vài điểm ảnh (các máy ảnh cao cấp được trang bị những tính năng đặc biệt nhằm khắc phục vấn đề này).

Trong mục này, chúng tôi sẽ giới thiệu về các tầng gộp, với hai chức năng là giảm độ nhạy cảm của các tầng tích chập đối với vị trí và giảm kích thước của các biểu diễn.

<sup>149</sup> <https://discuss.mxnet.io/t/2351>

<sup>150</sup> <https://forum.machinelearningcoban.com/c/d21>

### 8.5.1 Gộp cực đại và Gộp trung bình

Giống như các tầng tích chập, các toán tử gộp bao gồm một cửa sổ có kích thước cố định được trượt trên tất cả các vùng đầu vào với giá trị sai bước nhất định, tính toán một giá trị đầu ra duy nhất tại mỗi vị trí mà cửa sổ (đôi lúc được gọi là *cửa sổ gộp*) trượt qua. Tuy nhiên, không giống như phép toán tương quan chéo giữa đầu vào và hạt nhân ở tầng tích chập, tầng gộp không chứa bất kỳ tham số nào (ở đây không có “bộ lọc”). Thay vào đó, các toán tử gộp được định sẵn. Chúng thường tính giá trị cực đại hoặc trung bình của các phần tử trong cửa sổ gộp. Các phép tính này lần lượt được gọi là *gộp cực đại* (*max pooling*) và *gộp trung bình* (*average pooling*).

Trong cả hai trường hợp, giống như với toán tử tương quan chéo, ta có thể xem như cửa sổ gộp bắt đầu từ phía bên trái của mảng đầu vào và trượt qua mảng này từ trái sang phải và từ trên xuống dưới. Ở mỗi vị trí mà cửa sổ gộp dừng, nó sẽ tính giá trị cực đại hoặc giá trị trung bình của mảng con nằm trong cửa sổ (tùy thuộc vào phép gộp được sử dụng).

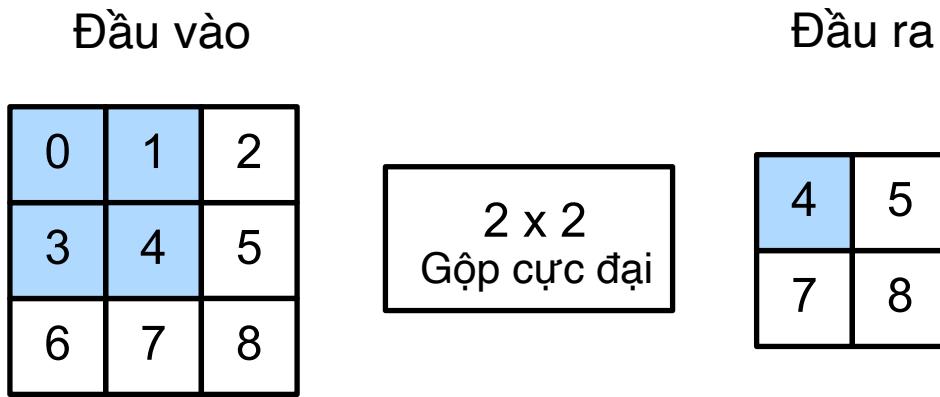


Fig. 8.5.1: Gộp cực đại với cửa sổ có kích thước  $2 \times 2$ . Các phần tử đậm thể hiện phần tử đầu ra đầu tiên và phần tử đầu vào được dùng để tính toán:  $\max(0, 1, 3, 4) = 4$

Mảng đầu ra ở Fig. 8.5.1 phía trên có chiều cao là 2 và chiều rộng là 2. Bốn phần tử của nó được là các giá trị cực đại của hàm max:

$$\begin{aligned} \max(0, 1, 3, 4) &= 4, \\ \max(1, 2, 4, 5) &= 5, \\ \max(3, 4, 6, 7) &= 7, \\ \max(4, 5, 7, 8) &= 8. \end{aligned} \tag{8.5.1}$$

Một tầng gộp với cửa sổ gộp có kích thước  $p \times q$  được gọi là một tầng gộp  $p \times q$ . Phép gộp sẽ được gọi là phép gộp  $p \times q$ .

Hãy cùng quay trở lại với ví dụ nhận diện biển của vật thể được đề cập ở đầu mục. Nay giờ, chúng ta sẽ sử dụng kết quả của tầng tích chập làm giá trị đầu vào cho tầng gộp cực đại  $2 \times 2$ . Đặt giá trị đầu vào của tầng tích chập là  $X$  và kết quả của tầng gộp là  $Y$ . Dù giá trị của  $X[i, j]$  và  $X[i, j+1]$  hay giá trị của  $X[i, j+1]$  và  $X[i, j+2]$  có khác nhau hay không, tất cả giá trị trả về của tầng gộp sẽ là  $Y[i, j] = 1$ . Nói cách khác, khi sử dụng tầng gộp cực đại  $2 \times 2$ , ta vẫn có thể phát hiện ra khuôn mẫu được nhận diện bởi tầng tích chập nếu nó bị chuyển dịch không nhiều hơn một phần tử theo chiều cao và chiều rộng.

Trong đoạn mã bên dưới, ta lập trình lượt truyền xuôi của tầng gộp trong hàm pool2d. Hàm này khá giống với hàm corr2d trong Section 8.2. Tuy nhiên, hàm này không có bộ lọc nên kết quả đầu ra hoặc là giá trị lớn nhất, hoặc là giá trị trung bình tương ứng của mỗi vùng đầu vào.

```

from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = np.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] = np.max(X[i:i + p_h, j:j + p_w])
            elif mode == 'avg':
                Y[i, j] = X[i:i + p_h, j:j + p_w].mean()
    return Y

```

Chúng ta có thể xây dựng mảng đầu vào X ở biểu đồ ở trên để kiểm tra giá trị kết quả của tầng gộp cực đại hai chiều.

```

X = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
pool2d(X, (2, 2))

```

Đồng thời, chúng ta cũng thực hiện thí nghiệm với tầng gộp trung bình.

```
pool2d(X, (2, 2), 'avg')
```

## 8.5.2 Đệm và Sai bước

Cũng giống như các tầng tính chập, các tầng gộp cũng có thể thay đổi kích thước đầu ra. Và cũng như trước, chúng ta có thể thay đổi cách thức hoạt động của tầng gộp để đạt được kích thước đầu ra như mong muốn bằng cách thêm đệm vào đầu vào và điều chỉnh sai bước. Chúng ta có thể minh họa cách sử dụng đệm và sai bước trong các tầng gộp thông qua tầng gộp cực đại hai chiều MaxPool2D được cung cấp trong mô-đun nn của thư viện MXNet Gluon. Đầu tiên, chúng ta tạo ra dữ liệu đầu vào kích thước (1, 1, 4, 4), trong đó hai chiều đầu tiên lần lượt là kích thước batch và số kênh.

```

X = np.arange(16).reshape(1, 1, 4, 4)
X

```

Theo mặc định, sai bước trong lớp MaxPool2D có cùng kích thước với cửa sổ gộp. Dưới đây, chúng ta sử dụng cửa sổ gộp kích thước (3, 3), vì vậy theo mặc định kích thước của sai bước trong tầng gộp này là (3, 3).

```

pool2d = nn.MaxPool2D(3)
# Because there are no model parameters in the pooling layer, we do not need
# to call the parameter initialization function
pool2d(X)

```

Giá trị của sai bước và đệm có thể được gán thủ công.

```

pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)

```

Đĩ nhiên, chúng ta có thể định nghĩa một cửa sổ gộp hình chữ nhật tuỳ ý và chỉ rõ giá trị phần đệm và sai bước tương ứng với chiều cao và chiều rộng của cửa sổ.

```
pool2d = nn.MaxPool2D((2, 3), padding=(1, 2), strides=(2, 3))
pool2d(X)
```

### 8.5.3 Với đầu vào Đa Kênh

Khi xử lý dữ liệu đầu vào đa kênh, tầng gộp sẽ áp dụng lên từng kênh một cách riêng biệt thay vì cộng từng phần tử tương ứng của các kênh lại với nhau như tầng tích chập. Điều này có nghĩa là số lượng kênh đầu ra của tầng gộp sẽ giống số lượng kênh đầu vào. Dưới đây, chúng ta sẽ ghép 2 mảng X và X+1 theo chiều kênh để tạo ra đầu vào 2 kênh.

```
X = np.concatenate((X, X + 1), axis=1)
X
```

Có thể thấy, số kênh của đầu ra vẫn là 2 sau khi gộp.

```
pool2d = nn.MaxPool2D(3, padding=1, strides=2)
pool2d(X)
```

### 8.5.4 Tóm tắt

- Với các phần tử đầu vào nằm trong cửa sổ gộp, tầng gộp cực đại sẽ cho đầu ra là giá trị lớn nhất trong số các phần tử đó và tầng gộp trung bình sẽ cho đầu ra là giá trị trung bình của các phần tử.
- Một trong những chức năng chủ yếu của tầng gộp là giảm thiểu sự ảnh hưởng quá mức của vị trí tới tầng tích chập.
- Chúng ta có thể chỉ rõ giá trị của đệm và sai bước cho tầng gộp.
- Tầng gộp cực đại kết hợp với sai bước lớn hơn 1 có thể dùng để giảm độ phân giải.
- Số lượng kênh đầu ra của tầng gộp sẽ bằng số lượng kênh đầu vào tầng gộp đó.

### 8.5.5 Bài tập

1. Có thể lập trình tầng gộp trung bình như một trường hợp đặc biệt của tầng tích chập không? Nếu được, hãy thực hiện nó.
2. Có thể lập trình tầng gộp cực đại như một trường hợp đặc biệt của tầng tích chập không? Nếu được, hãy thực hiện nó.
3. Hãy tính chi phí tính toán của tầng gộp trong trường hợp, giả sử đầu vào của tầng gộp có kích thước  $c \times h \times w$ , kích thước của cửa sổ gộp  $p_h \times p_w$  với đệm  $(p_h, p_w)$  và sai bước  $(s_h, s_w)$ .
4. Tại sao ta mong đợi tầng gộp cực đại và tầng gộp trung bình có những ảnh hưởng khác nhau?
5. Theo ý kiến của bạn, có cần riêng một tầng gộp cực tiểu không? Có thể thay thế bằng một cơ chế khác không?

- Hãy thử suy nghĩ một cơ chế khác nằm giữa gộp trung bình và gộp cực đại (gợi ý: hãy nhớ lại hàm softmax). Tại sao nó không phổ biến?

### 8.5.6 Thảo luận

- Tiếng Anh<sup>151</sup>
- Tiếng Việt<sup>152</sup>

### 8.5.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Phạm Minh Đức
- Nguyễn Đình Nam
- Lê Khắc Hồng Phúc
- Đinh Đắc
- Phạm Hồng Vinh

## 8.6 Mạng Nơ-ron Tích chập (LeNet)

Bây giờ ta đã sẵn sàng kết hợp tất cả các công cụ lại với nhau để triển khai mạng nơ-ron tích chập hoàn chỉnh đầu tiên. Lần đầu làm việc với dữ liệu ảnh, ta đã áp dụng một perceptron đa tầng (Section 6.2) cho ảnh quần áo trong bộ dữ liệu Fashion-MNIST. Mỗi ảnh trong Fashion-MNIST là một ma trận hai chiều có kích thước  $28 \times 28$ . Để tương thích với đầu vào dạng vector một chiều với độ dài cố định của các perceptron đa tầng, đầu tiên ta trải phẳng từng hình ảnh và thu được các vector có chiều dài 784, trước khi xử lý chúng với một chuỗi các tầng kết nối đầy đủ.

Bây giờ đã có các tầng tích chập, ta có thể giữ nguyên ảnh đầu vào ở dạng không gian hai chiều như ảnh gốc và xử lý chúng với một chuỗi các tầng tích chập liên tiếp. Hơn nữa, vì ta đang sử dụng các tầng tích chập, số lượng tham số cần thiết sẽ giảm đi đáng kể.

Trong phần này, chúng tôi sẽ giới thiệu một trong những mạng nơ-ron tích chập được công bố đầu tiên. Ưu điểm của mạng tích chập được minh họa lần đầu bởi Yann LeCun (lúc đó đang nghiên cứu tại AT&T Bell Labs) với ứng dụng nhận dạng các số viết tay trong ảnh-LeNet<sup>153</sup>. Vào những năm 90, các thí nghiệm của các nhà nghiên cứu với LeNet đã đưa ra bằng chứng thuyết phục đầu tiên về tính khả thi của việc huấn luyện mạng nơ-ron tích chập bằng lan truyền ngược. Mô hình của họ đã đạt được kết quả rất tốt (chỉ có Máy Vector Hỗ trợ – SVM tại thời điểm đó là có thể sánh bằng) và đã được đưa vào sử dụng để nhận diện các chữ số khi xử lý tiền gửi trong máy ATM. Một số máy ATM vẫn chạy các đoạn mã mà Yann và đồng nghiệp Leon Bottou đã viết vào những năm 1990!

<sup>151</sup> <https://discuss.mxnet.io/t/2352>

<sup>152</sup> <https://forum.machinelearningcoban.com/c/d21>

<sup>153</sup> <http://yann.lecun.com/exdb/lenet/>

### 8.6.1 LeNet

Một cách đơn giản, ta có thể xem LeNet gồm hai phần: (i) một khối các tầng tích chập; và (ii) một khối các tầng kết nối đầy đủ. Trước khi đi vào các chi tiết cụ thể, hãy quan sát tổng thể mô hình trong Fig. 8.6.1.

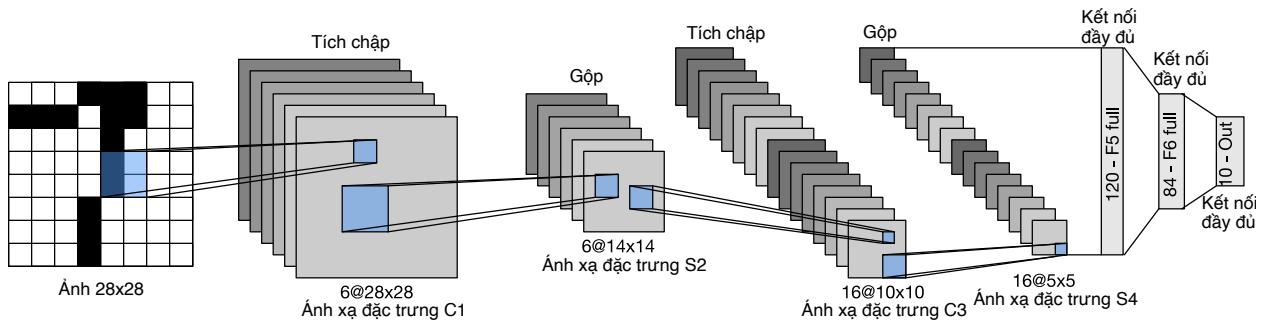


Fig. 8.6.1: Dòng dữ liệu trong LeNet 5. Đầu vào là một chữ số viết tay, đầu ra là một xác suất đối với 10 kết quả khả thi.

Các đơn vị cơ bản trong khối tích chập là một tầng tích chập và một lớp gộp trung bình sau (lưu ý rằng gộp cực đại hoạt động tốt hơn, nhưng nó chưa được phát minh vào những năm 90). Tầng tích chập được sử dụng để nhận dạng các mẫu không gian trong ảnh, chẳng hạn như các đường cạnh và các bộ phận của vật thể, lớp gộp trung bình phía sau được dùng để giảm số chiều. Khối tầng tích chập tạo nên từ việc xếp chồng các khối nhỏ gồm hai đơn vị cơ bản này. Mỗi tầng tích chập sử dụng hạt nhân có kích thước  $5 \times 5$  và xử lý mỗi đầu ra với một hàm kích hoạt sigmoid (nhấn mạnh rằng ReLU hiện được biết là hoạt động đáng tin cậy hơn, nhưng chưa được phát minh vào thời điểm đó). Tầng tích chập đầu tiên có 6 kênh đầu ra và tầng tích chập thứ hai tăng độ sâu kênh hơn nữa lên 16.

Tuy nhiên, cùng với sự gia tăng số lượng kênh này, chiều cao và chiều rộng lại giảm đáng kể. Do đó, việc tăng số lượng kênh đầu ra làm cho kích thước tham số của hai tầng tích chập tương tự nhau. Hai lớp gộp trung bình có kích thước  $2 \times 2$  và sải bước bằng 2 (điều này có nghĩa là chúng không chồng chéo). Nói cách khác, lớp gộp giảm kích thước của các biểu diễn còn *một phần tư* kích thước trước khi gộp.

Đầu ra của khối tích chập có kích thước được xác định bằng (kích thước batch, kênh, chiều cao, chiều rộng). Trước khi chuyển đầu ra của khối tích chập sang khối kết nối đầy đủ, ta phải trải phẳng từng mẫu trong minibatch. Nói cách khác, ta biến đổi đầu vào 4D thành đầu vào 2D tương thích với các tầng kết nối đầy đủ: nhắc lại, chiều thứ nhất là chỉ số các mẫu trong minibatch và chiều thứ hai là biểu diễn vector phẳng của mỗi mẫu. Khối tầng kết nối đầy đủ của LeNet có ba tầng kết nối đầy đủ, với số lượng đầu ra lần lượt là 120, 84 và 10. Bởi vì ta đang thực hiện bài toán phân loại, tầng đầu ra 10 chiều tương ứng với số lượng các lớp đầu ra khả thi (10 chữ số từ 0 đến 9).

Để thực sự hiểu những gì diễn ra bên trong LeNet có thể đòi hỏi một chút nỗ lực, tuy nhiên bạn có thể thấy bên dưới đây việc lập trình Lenet bằng thư viện học sâu hiện đại rất đơn giản. Một lần nữa, ta sẽ dựa vào lớp Sequential.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

(continues on next page)

```

net = nn.Sequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='sigmoid'),
       nn.AvgPool2D(pool_size=2, strides=2),
       nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
       nn.AvgPool2D(pool_size=2, strides=2),
       # Dense will transform the input of the shape (batch size, channel,
       # height, width) into the input of the shape (batch size,
       # channel * height * width) automatically by default
       nn.Dense(120, activation='sigmoid'),
       nn.Dense(84, activation='sigmoid'),
       nn.Dense(10))

```

So với mạng ban đầu, ta đã thay thế kích hoạt Gauss ở tầng cuối cùng bằng một tầng kết nối đầy đủ thông thường mà thường dễ huấn luyện hơn đáng kể. Ngoại trừ điểm đó, mạng này giống với định nghĩa gốc của LeNet5.

Tiếp theo, ta hãy xem một ví dụ dưới đây. Như trong Fig. 8.6.2, ta đưa vào mạng một mẫu đơn kênh kích thước  $28 \times 28$  và thực hiện một lượt truyền xuôi qua các tầng và in kích thước đầu ra ở mỗi tầng để hiểu rõ những gì đang xảy ra bên trong.

```

X = np.random.uniform(size=(1, 1, 28, 28))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)

```

Xin hãy chú ý rằng, chiều cao và chiều rộng của biểu diễn sau mỗi tầng trong toàn bộ khối tích chập sẽ giảm theo chiều sâu của mạng (so với chiều cao và chiều rộng của biểu diễn ở tầng trước). Tầng tích chập đầu tiên sử dụng một hạt nhân với chiều cao và chiều rộng là 5 rồi đệm thêm 2 đơn vị điểm ảnh để giữ nguyên kích thước đầu vào. Trong khi đó, tầng tích chập thứ hai cũng dùng cùng một hạt nhân với kích thước là  $5 \times 5$  mà không có sử dụng giá trị đệm thêm vào, dẫn đến việc chiều cao và chiều rộng giảm đi 4 đơn vị điểm ảnh. Ngoài ra, mỗi tầng gộp sẽ làm giảm đi một nửa chiều cao và chiều rộng của đặc trưng ánh xạ đầu vào. Tuy nhiên, khi chúng ta di theo chiều sâu của mạng, số kênh sẽ tăng lần lượt theo từng tầng. Từ 1 kênh của dữ liệu đầu vào lên tới 6 kênh sau tầng tích chập thứ nhất và 16 kênh sau tầng tích chập thứ hai. Sau đó, giảm số chiều lần lượt qua từng tầng kết nối đầy đủ đến khi trả về một đầu ra có kích thước bằng số lượng lớp của hình ảnh.

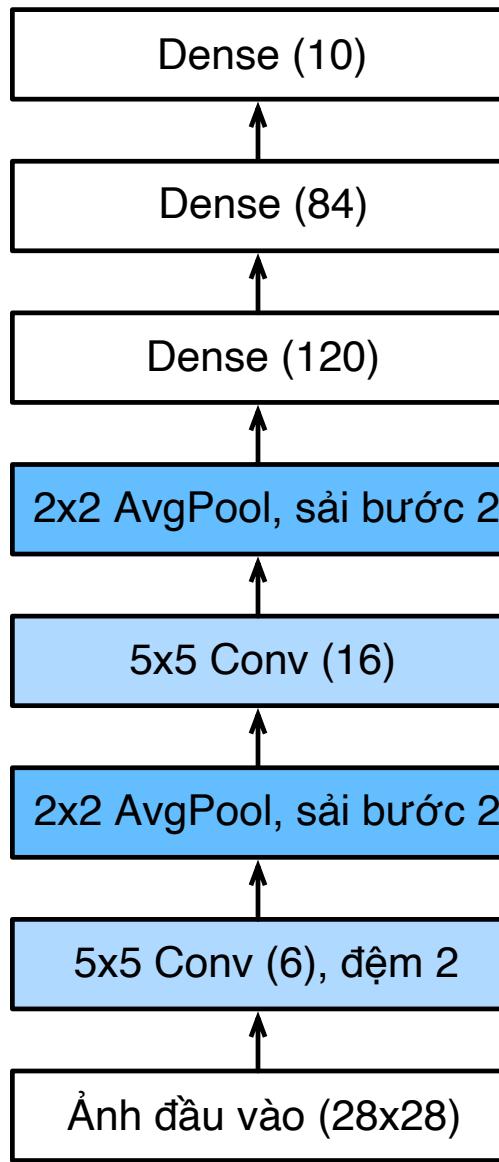


Fig. 8.6.2: Kí hiệu văn tắt cho mô hình LeNet5

### 8.6.2 Thu thập Dữ liệu và Huấn luyện

Sau khi xây dựng xong mô hình, chúng ta sẽ thực hiện một số thử nghiệm để xem chất lượng của mô hình LeNet. Tập dữ liệu Fashion-MNIST sẽ được dùng trong ví dụ này. Việc phân loại tập Fashion-MNIST sẽ khó hơn so với tập MNIST gốc mặc dù chúng đều chứa các ảnh có cùng kích thước  $28 \times 28$ .

```
batch_size = 256
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

Dù mạng tích chập có thể có số lượng tham số không lớn, chúng vẫn tiêu tốn nhiều tài nguyên tính toán hơn so với perceptron sâu đa tầng. Vì vậy, nếu có sẵn GPU, thì đây là thời điểm thích hợp để dùng nó nhằm tăng tốc quá trình huấn luyện.

Để đánh giá mô hình, chúng ta cần điều chỉnh một chút hàm `evaluate_accuracy` đã mô tả ở phần Section 5.6. Vì toàn bộ tập dữ liệu đang nằm trên CPU, ta cần sao chép nó lên GPU trước khi thực

hiện tính toán với mô hình. Việc này được thực hiện thông qua việc gọi hàm `as_in_ctx` được mô tả ở phần Section 7.6.

```
# Saved in the d2l package for later use
def evaluate_accuracy_gpu(net, data_iter, ctx=None):
    if not ctx: # Query the first device the first parameter is on
        ctx = list(net.collect_params().values())[0].list_ctx()[0]
    metric = d2l.Accumulator(2) # num_corrected_examples, num_examples
    for X, y in data_iter:
        X, y = X.as_in_ctx(ctx), y.as_in_ctx(ctx)
        metric.add(d2l.accuracy(net(X), y), y.size)
    return metric[0]/metric[1]
```

Chúng ta cũng cần phải cập nhật hàm huấn luyện để mô hình có thể chạy được trên GPU. Không giống hàm `train_epoch_ch3` được định nghĩa ở phần Section 5.6, giờ chúng ta cần chuyển từng batch dữ liệu tới ngữ cảnh được chỉ định (hy vọng là GPU thay vì CPU) trước khi thực hiện lượt truyền xuôi và lượt truyền ngược.

Hàm huấn luyện `train_ch6` khá giống với hàm huấn luyện `train_ch3` đã được định nghĩa tại Section 5.6. Để đơn giản khi làm việc với mạng nơ-ron có tới hàng chục tầng, hàm `train_ch6` chỉ hỗ trợ các mô hình được xây dựng bằng thư viện Gluon. Để khởi tạo bộ tham số của mô hình trên thiết bị đã được chỉ định bởi `ctx`, ta sẽ sử dụng bộ khởi tạo Xavier. Ta vẫn sử dụng hàm mất mát entropy chéo và thuật toán huấn luyện là phương pháp hạ gradient ngẫu nhiên theo minibatch. Với mỗi epoch tốn khoảng hàng chục giây để chạy, ta sẽ vẽ đường biểu diễn giá trị mất mát huấn luyện với nhiều giá trị chi tiết hơn.

```
# Saved in the d2l package for later use
def train_ch6(net, train_iter, test_iter, num_epochs, lr, ctx=d2l.try_gpu()):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(),
                            'sgd', {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3) # train_loss, train_acc, num_examples
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            # Here is the only difference compared to train_epoch_ch3
            X, y = X.as_in_ctx(ctx), y.as_in_ctx(ctx)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), d2l.accuracy(y_hat, y), X.shape[0])
            timer.stop()
            train_loss, train_acc = metric[0]/metric[2], metric[1]/metric[2]
            if (i+1) % 50 == 0:
                animator.add(epoch + i/len(train_iter),
                            (train_loss, train_acc, None))
            test_acc = evaluate_accuracy_gpu(net, test_iter)
            animator.add(epoch+1, (None, None, test_acc))
    print('loss %.3f, train acc %.3f, test acc %.3f' % (
```

(continues on next page)

```
train_loss, train_acc, test_acc))
print('%.1f examples/sec on %s' % (metric[2]*num_epochs/timer.sum(), ctx))
```

Bây giờ, chúng ta hãy bắt đầu huấn luyện mô hình.

```
lr, num_epochs = 0.9, 10
train_ch6(net, train_iter, test_iter, num_epochs, lr)
```

### 8.6.3 Tóm tắt

- Mạng nơ-ron tích chập (gọi tắt là ConvNet) là một mạng sử dụng các tầng tích chập.
- Trong ConvNet, ta xen kẽ các phép tích chập, các hàm phi tuyến và các phép gộp.
- Độ phân giải được giảm xuống trước khi tạo một đầu ra thông qua một (hoặc nhiều) tầng kết nối dày đặc.
- LeNet là mạng ConvNet đầu tiên được triển khai thành công.

### 8.6.4 Bài tập

1. Điều gì sẽ xảy ra nếu ta thay thế phép gộp trung bình bằng phép gộp cực đại?
2. Thử cải thiện độ chính xác dự đoán dựa trên LeNet bằng cách: \* Điều chỉnh kích thước cửa sổ tích chập. \* Điều chỉnh số lượng kênh đầu ra. \* Điều chỉnh hàm kích hoạt (ReLU?). \* Điều chỉnh số lượng các tầng tích chập. \* Điều chỉnh số lượng các tầng kết nối dày đặc. \* Điều chỉnh tốc độ học và các chi tiết huấn luyện khác (phương thức khởi tạo, số lượng epoch, v.v.)
3. Thử sử dụng mạng đã cải tiến ở phần 3 với tập dữ liệu MNIST ban đầu.
4. Hiển thị các giá trị kích hoạt của tầng thứ nhất và tầng thứ hai của LeNet với các đầu vào khác nhau (ví dụ: áo len, áo khoác).

### 8.6.5 Thảo luận

- [Tiếng Anh](#)<sup>154</sup>
- [Tiếng Việt](#)<sup>155</sup>

<sup>154</sup> <https://discuss.mxnet.io/t/2353>

<sup>155</sup> <https://forum.machinelearningcoban.com/c/d2l>

### **8.6.6 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Văn Cường
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Đinh Đắc
- Phạm Hồng Vinh
- Nguyễn Duy Du

### **8.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc

# 9 | Mạng Nơ-ron Tích chập Hiện đại

Ở chương trước, chúng ta đã nắm rõ các khái niệm cơ bản trong việc xây dựng mạng nơ-ron tích chập, bây giờ hãy cùng tìm hiểu về học sâu hiện đại. Trong chương này, từng phần sẽ trình bày một kiến trúc mạng nơ-ron quan trọng mà tại một thời điểm nào đó (có thể cả trong hiện tại) là mô hình nền tảng được sử dụng trong một lượng lớn các nghiên cứu và dự án. Mỗi kiến trúc mạng này thống trị trong một khoảng thời gian, nhiều mạng đã về nhất hoặc nhì tại ImageNet, một cuộc thi được xem là thước đo sự phát triển của học có giám sát trong thị giác máy tính kể từ năm 2010.

Những mô hình này gồm có: AlexNet, mạng quy mô lớn đầu tiên được triển khai để đánh bại các phương pháp thị giác máy tính truyền thống trong một thử thách quy mô lớn; VGG, mạng tận dụng một số lượng các khối được lặp đi lặp lại; Mạng trong Mạng (*Network in Network - NiN*), một kiến trúc nhằm di chuyển toàn bộ mạng nơ-ron theo từng điểm ảnh ở đầu vào; GoogLeNet sử dụng các phép nối song song giữa các mạng; Mạng phần dư (*residual networks - ResNet*) là kiến trúc mạng được sử dụng phổ biến nhất hiện nay; và cuối cùng là Mạng tích chập kết nối dày đặc (*densely connected network - DenseNet*), dù yêu cầu khối lượng tính toán lớn nhưng thường được sử dụng làm mốc chuẩn trong thời gian gần đây.

## 9.1 Mạng Nơ-ron Tích chập Sâu (AlexNet)

Mặc dù đã trở nên nổi tiếng trong cộng đồng thị giác máy tính và học máy sau khi LeNet được giới thiệu, mạng nơ-ron tích chập chưa lập tức thống trị lĩnh vực này. Dẫu LeNet đã đạt được kết quả tốt trên những tập dữ liệu nhỏ, chất lượng và tính khả thi của việc huấn luyện mạng tích chập trên một tập dữ liệu lớn và sát thực tế hơn vẫn là một câu hỏi. Trên thực tế, hầu hết trong khoảng thời gian từ đầu những năm 1990 cho tới năm 2012 với những thành tựu mang tính bước ngoặt, mạng nơ-ron tích chập thường không sánh bằng những phương pháp học máy khác, như Máy Vector hỗ trợ - SVM.

Với thị giác máy tính, phép so sánh này dường như không công bằng. Nguyên nhân là do giá trị đầu vào của mạng tích chập chỉ bao gồm giá trị điểm ảnh thô hoặc đã xử lý thô (như định tâm ảnh (*centering*)), và kinh nghiệm cho thấy những giá trị thô này không bao giờ nên dùng trực tiếp trong các mô hình truyền thống. Thay vào đó, các hệ thống thị giác máy tính cổ điển dùng những pipeline trích xuất đặc trưng một cách thủ công. Thay vì được *học*, các đặc trưng được *tạo thủ công*. Hầu hết những tiến triển trong ngành đều đến từ các ý tưởng thông minh hơn trong tạo đặc trưng và ít để ý hơn tới thuật toán học.

Mặc dù cũng đã có các thiết bị phần cứng tăng tốc độ thực thi mạng nơ-ron vào đầu những năm 1990, chúng vẫn chưa đủ mạnh để triển khai những mạng nơ-ron nhiều kênh, nhiều tầng với số lượng tham số lớn. Ngoài ra, những tập dữ liệu vẫn còn tương đối nhỏ.Thêm vào đó, những thủ thuật chính để huấn luyện mạng nơ-ron bao gồm khởi tạo tham số dựa trên thực nghiệm, các biến thể tốt hơn của hạ gradient ngẫu nhiên, hàm kích hoạt không ép (*non-squashing activation functions*), và thiếu các kỹ thuật điều chỉnh hiệu quả.

Vì vậy, thay vì huấn luyện các hệ thống *đầu-cuối* (từ điểm ảnh đến phân loại), các pipeline cổ điển sẽ thực hiện các bước sau:

1. Thu thập tập dữ liệu đáng chú ý. Trong những ngày đầu, các tập dữ liệu này đòi hỏi các cảm biến đắt tiền (ảnh có 1 triệu điểm ảnh đã được coi là tối tân nhất vào thời điểm đó).
2. Tiền xử lý tập dữ liệu với các đặc trưng được tạo thủ công dựa trên các kiến thức quang học, hình học, các công cụ phân tích khác và thi thoảng dựa trên các khám phá tình cờ của các nghiên cứu sinh.
3. Đưa dữ liệu qua một bộ trích chọn đặc trưng tiêu chuẩn như SIFT<sup>156</sup>, hoặc SURF<sup>157</sup>, hay bất kỳ pipeline được tinh chỉnh thủ công nào.
4. Dùng các kết quả biểu diễn để huấn luyện một bộ phân loại ưa thích, có thể là một mô hình tuyến tính hoặc phương pháp hạt nhân.

Khi tiếp xúc với những nhà nghiên cứu học máy, bạn sẽ thấy họ tin rằng học máy không những quan trọng mà còn “đẹp” nữa. Bởi lẽ có nhiều lý thuyết tinh vi được đưa ra để chứng minh các tính chất của nhiều bộ phân loại. Và cứ như vậy, lĩnh vực học máy ngày một lớn mạnh, nghiêm ngặt, và hữu dụng hơn bao giờ hết. Tuy nhiên, nếu có dịp thảo luận với một nhà nghiên cứu thị giác máy tính, thì có thể ta lại được nghe một câu chuyện rất khác. Họ sẽ nói rằng sự thật tràn trụi trong nhận dạng ảnh là “đặc trưng mới mang tính quyết định tới chất lượng chứ không phải thuật toán học”. Những nhà nghiên cứu thị giác máy tính thời đó có lý do để tin rằng chỉ cần một tập dữ liệu hơi lớn hơn, sạch hơn hoặc một pipeline trích xuất đặc trưng tốt hơn một chút sẽ có ảnh hưởng lớn hơn bất kỳ thuật toán học nào.

### 9.1.1 Học Biểu diễn Đặc trưng

Nói một cách khác, tại thời điểm đó phần lớn các nhà nghiên cứu tin rằng phần quan trọng nhất của pipeline là sự biểu diễn. Và cho tới năm 2012 việc biểu diễn vẫn được tính toán một cách máy móc. Trong thực tế, thiết kế và xây dựng một tập các hàm đặc trưng mới, cải thiện kết quả, và viết ra phương pháp thực hiện từng là một phần quan trọng của các bài báo nghiên cứu. SIFT<sup>158</sup>, SURF<sup>159</sup>, HOG<sup>160</sup>, Bags of visual words<sup>161</sup> và các bộ trích chọn đặc trưng tương tự đã chiếm ưu thế vượt trội.

Một nhóm các nhà nghiên cứu bao gồm Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, và Juergen Schmidhuber, lại có những kế hoạch khác. Họ tin rằng đặc trưng cũng có thể được học. Hơn nữa, họ cũng cho rằng để có được độ phức tạp hợp lý, các đặc trưng nên được phân thành thứ lớp với nhiều tầng học cùng nhau, mỗi tầng có các tham số có thể được huấn luyện. Trong trường hợp ảnh, các tầng thấp nhất có thể dùng để phát hiện biên, màu sắc và đường nét. Thật vậy, (Krizhevsky et al., 2012) giới thiệu một biến thể mới của mạng nơ-ron tích chập đã đạt được hiệu năng xuất sắc trong cuộc thi ImageNet.

Một điều thú vị là ở các tầng thấp nhất của mạng, mô hình đã học được cách trích xuất đặc trưng giống như các bộ lọc truyền thống. Fig. 9.1.1 được tái tạo từ bài báo khoa học trên mô tả các đặc trưng cấp thấp của hình ảnh.

<sup>156</sup> [https://en.wikipedia.org/wiki/Scale-invariant\\_feature\\_transform](https://en.wikipedia.org/wiki/Scale-invariant_feature_transform)

<sup>157</sup> [https://en.wikipedia.org/wiki/Speeded\\_up\\_robust\\_features](https://en.wikipedia.org/wiki/Speeded_up_robust_features)

<sup>158</sup> [https://en.wikipedia.org/wiki/Scale-invariant\\_feature\\_transform](https://en.wikipedia.org/wiki/Scale-invariant_feature_transform)

<sup>159</sup> [https://en.wikipedia.org/wiki/Speeded\\_up\\_robust\\_features](https://en.wikipedia.org/wiki/Speeded_up_robust_features)

<sup>160</sup> [https://en.wikipedia.org/wiki/Histogram\\_of\\_oriented\\_gradients](https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients)

<sup>161</sup> [https://en.wikipedia.org/wiki/Bag-of-words\\_model\\_in\\_computer\\_vision](https://en.wikipedia.org/wiki/Bag-of-words_model_in_computer_vision)

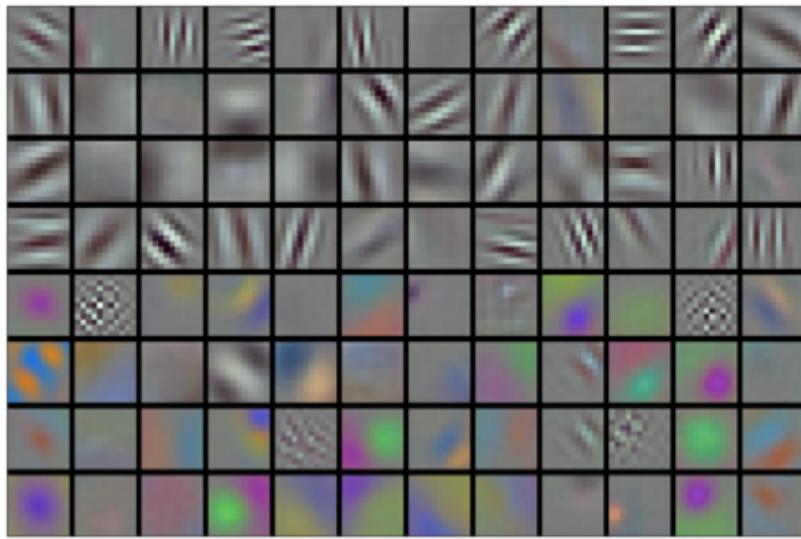


Fig. 9.1.1: Các bộ lọc hình ảnh học được ở tầng đầu tiên của mô hình AlexNet

Các tầng cao hơn của mạng sẽ dựa vào các biểu diễn này để thể hiện các cấu trúc lớn hơn như mắt, mũi, ngọn cỏ, v.v. Thậm chí các tầng cao hơn nữa có thể đại diện cho nguyên một vật thể như con người, máy bay, chó hoặc là đĩa ném. Sau cùng, tầng trạng thái ẩn cuối sẽ học cách biểu diễn cô đọng của toàn bộ hình ảnh để tổng hợp lại nội dung sao cho dữ liệu thuộc các lớp khác nhau có thể được dễ dàng phân biệt.

Mặc dù bước đột phá của các mạng tích chập nhiều tầng xuất hiện vào năm 2012, một nhóm nòng cốt các nhà nghiên cứu đã theo đuổi ý tưởng này, tìm cách học các biểu diễn phân tầng của dữ liệu hình ảnh trong nhiều năm. Có hai yếu tố chính dẫn tới bước đột phá lớn vào năm 2012.

### **Yếu tố bị Thiếu - Dữ liệu**

Mô hình học sâu với nhiều tầng đòi hỏi phải có một lượng dữ liệu lớn để đạt hiệu quả vượt trội so với các phương pháp truyền thống dựa trên tối ưu lồi (ví dụ: phương pháp tuyến tính và phương pháp nhân). Tuy nhiên, do khả năng lưu trữ của máy tính còn hạn chế, các bộ cảm biến khá đắt đỏ, và ngân sách dành cho việc nghiên cứu tương đối bị thắt chặt vào những năm 1990, phần lớn các nghiên cứu đều dựa trên những bộ dữ liệu nhỏ. Có rất nhiều bài báo nghiên cứu khoa học giải quyết các vấn đề dựa trên bộ dữ liệu tổng hợp UCI, nhiều bộ dữ liệu trong số đó chỉ chứa khoảng vài trăm hoặc (một vài) ngàn hình ảnh được chụp trong điều kiện không tự nhiên với độ phân giải thấp.

Năm 2009, tập dữ liệu ImageNet được ban hành, thách thức các nhà nghiên cứu huấn luyện những mô hình với 1 triệu hình ảnh, trong đó có 1.000 ảnh cho mỗi 1.000 lớp đối tượng khác nhau. Các nhà nghiên cứu giới thiệu tập dữ liệu này, dẫn đầu bởi Fei-Fei Li, đã tận dụng công cụ Tìm kiếm Hình ảnh của Google để lọc sơ bộ các tập dữ liệu hình ảnh lớn cho mỗi lớp và sử dụng dịch vụ cộng đồng (*crowdsourcing*) Amazon Mechanical Turk để xác thực nhãn cho từng ảnh. Đây là quy mô lớn chưa từng có từ trước đến nay. Cuộc thi đi liền với tập dữ liệu này được đặt tên là ImageNet Challenge và đã thúc đẩy sự phát triển của nghiên cứu thị giác máy tính và học máy, thách thức các nhà nghiên cứu tìm ra mô hình tốt nhất ở quy mô lớn hơn bao giờ hết trong toàn giới học thuật.

## **Yếu tố bị Thiếu - Phần cứng**

Các mô hình học sâu đòi hỏi rất nhiều chu kỳ tính toán. Quá trình huấn luyện có thể cần hàng trăm epoch, với mỗi vòng lặp yêu cầu đưa dữ liệu qua nhiều tầng nơi các phép toán đại số tuyến tính cồng kềnh được thực thi. Đây là một trong những lý do chính tại sao vào những năm 90 tới đầu những năm 2000, các thuật toán đơn giản dựa trên những mục tiêu tối ưu lồi hiệu quả lại được ưa chuộng hơn.

Bộ xử lý đồ họa (GPU) đóng vai trò thay đổi hoàn toàn cuộc chơi khi làm cho việc học sâu trở nên khả thi. Những vi xử lý này đã được phát triển một thời gian dài để tăng tốc độ xử lý đồ họa dành cho các trò chơi máy tính. Cụ thể, chúng được tối ưu hóa cho các phép nhân ma trận - vector  $4 \times 4$  thông lượng cao, cần thiết cho nhiều tác vụ đồ họa. May mắn thay, phép toán này rất giống với phép toán sử dụng trong các tàng tích chập. Trong khoảng thời gian này, hai công ty NVIDIA và ATI đã bắt đầu tối ưu GPU cho mục đích tính toán tổng quát, thậm chí còn tiếp thị chúng dưới dạng GPU đa dụng (*General Purpose GPUs - GPGPU*).

Để hình dung rõ hơn, hãy cùng xem lại các nhân của bộ vi xử lý CPU hiện đại. Mỗi nhân thì khá mạnh khi chạy ở tần số xung nhịp cao với bộ nhớ đệm lớn (lên đến vài MB ở bộ nhớ đệm L3). Mỗi nhân phù hợp với việc thực hiện hàng loạt các loại chỉ dẫn khác nhau, với các bộ dự báo rẽ nhánh, một pipeline sâu và những tính năng phụ trợ khác cho phép nó có khả năng chạy một lượng lớn các chương trình khác nhau. Tuy nhiên, sức mạnh rõ rệt này cũng có điểm yếu: sản xuất các nhân đa dụng rất đắt đỏ. Chúng đòi hỏi nhiều diện tích cho vi xử lý, cùng cấu trúc hỗ trợ phức tạp (giao diện bộ nhớ, logic bộ nhớ đệm giữa các nhân, kết nối tốc độ cao, v.v.), và chúng tương đối tệ ở bất kỳ tác vụ đơn lẻ nào. Những máy tính xách tay ngày nay chỉ có tối 4 nhân, và thậm chí những máy chủ cao cấp hiếm khi vượt quá 64 nhân, đơn giản bởi vì chúng không hiệu quả về chi phí.

Để so sánh, GPU bao gồm 100-1000 các phần tử xử lý nhỏ (về chi tiết có khác nhau đôi chút giữa NVIDIA, ATI, ARM và các nhà sản xuất khác), thường được gộp thành các nhóm lớn hơn (NVIDIA gọi các nhóm này là luồng (*warp*)). Mặc dù mỗi nhân thì tương đối yếu, đôi khi thậm chí chạy ở tần số xung nhịp dưới 1GHZ, nhưng số lượng của những nhân này giúp cho GPUs có tốc độ nhanh hơn rất nhiều so với CPUs. Chẳng hạn, thế hệ Volta mới nhất của NVIDIA có thể thực hiện tới 120 nghìn phép toán dấu phẩy động (TFlop) với mỗi vi xử lý cho những lệnh chuyên biệt (và lên tới 24 TFlop cho các lệnh đa dụng), trong khi hiệu năng của CPU trong việc thực hiện tính toán số thực dấu phẩy động cho đến nay không vượt quá 1 TFlop. Lý do khá đơn giản: thứ nhất, mức độ tiêu thụ năng lượng có xu hướng tăng theo hàm bậc hai so với tần số xung nhịp. Do đó, với cùng lượng năng lượng để một nhân CPU chạy nhanh gấp 4 lần tốc độ hiện tại (mức tăng thường gấp), chúng ta có thể thay bằng 16 nhân GPU với tốc độ mỗi nhân giảm còn 1/4, cũng sẽ cho kết quả là  $16 \times 1/4 = 4$  lần tốc độ hiện tại. Hơn nữa, các nhân của GPU thì đơn giản hơn nhiều (trên thực tế, trong một khoảng thời gian dài những nhân này thậm chí *không thể* thực thi được các mã lệnh đa dụng), điều này giúp chúng tiết kiệm năng lượng hơn. Cuối cùng, nhiều phép tính trong quá trình học sâu đòi hỏi bộ nhớ băng thông cao. Và một lần nữa, GPUs tỏa sáng khi độ rộng đường bus của nó lớn hơn ít nhất 10 lần so với nhiều loại CPUs.

Quay trở lại năm 2012. Một bước đột phá lớn khi Alex Krizhevsky và Ilya Sutskever đã xây dựng thành công một mạng nơ-ron tích chập sâu có thể chạy trên phần cứng GPU. Họ nhận ra rằng nút thắt cổ chai khi tính toán trong CNN (phép nhân tích chập và ma trận) có thể được xử lý song song trên phần cứng. Sử dụng hai card đồ họa NVIDIA GTX 580s với 3GB bộ nhớ, họ đã xây dựng các phép toán tích chập nhanh. Phần mã nguồn [cuda-convnet<sup>162</sup>](https://code.google.com/archive/p/cuda-convnet/) được xem là ngòi nổ cho sự phát triển vượt bậc của học sâu ngày nay.

<sup>162</sup> <https://code.google.com/archive/p/cuda-convnet/>

### 9.1.2 Mạng AlexNet

Mạng AlexNet được giới thiệu vào năm 2012, được đặt theo tên của Alex Krizhevsky, tác giả thứ nhất của bài báo đột phá trong phân loại ImageNet (Krizhevsky et al., 2012). Mạng AlexNet bao gồm 8 tầng mạng nơ-ron tích chập, đã chiến thắng cuộc thi **ImageNet Large Scale Visual Recognition Challenge** năm 2012 với cách biệt không tưởng. AlexNet lần đầu tiên đã chứng minh được rằng các đặc trưng thu được bởi việc học có thể vượt qua các đặc trưng được thiết kế thủ công, phá vỡ định kiến trước đây trong nghiên cứu thị giác máy tính. Cấu trúc mạng AlexNet và LeNet *rất giống nhau*, như Fig. 9.1.2 đã minh họa. Lưu ý rằng chúng tôi cung cấp một phiên bản AlexNet được sắp xếp hợp lý, loại bỏ một số điểm thiết kế có từ năm 2012 với mục đích làm cho mô hình phù hợp với hai GPU dung lượng nhỏ mà bây giờ đã không còn cần thiết.

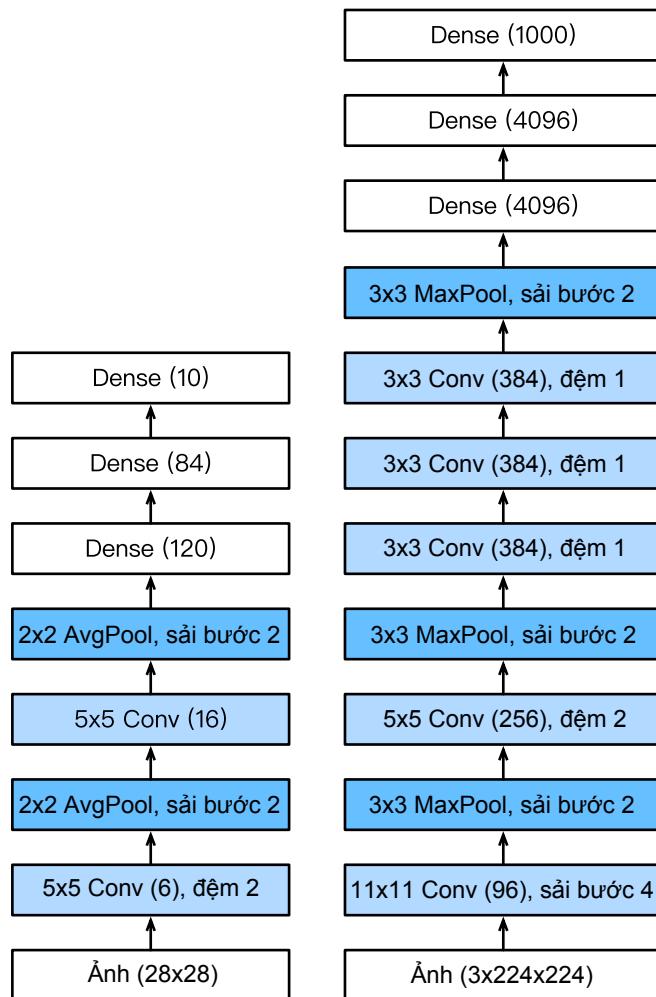


Fig. 9.1.2: LeNet (trái) và AlexNet (phải)

Các triết lý thiết kế của AlexNet và LeNet rất giống nhau, nhưng cũng có những khác biệt đáng kể. Đầu tiên, AlexNet sâu hơn nhiều so với LeNet5. AlexNet có tám tầng gồm: năm tầng tích chập, hai tầng ẩn kết nối đầy đủ, và một tầng đầu ra kết nối đầy đủ. Thứ hai, AlexNet sử dụng ReLU thay vì sigmoid làm hàm kích hoạt. Chúng tôi sẽ đi sâu hơn vào chi tiết ở dưới đây.

## Kiến trúc

Trong tầng thứ nhất của AlexNet, kích thước cửa sổ tích chập là  $11 \times 11$ . Vì hầu hết các ảnh trong ImageNet đều có chiều cao và chiều rộng lớn gấp hơn mười lần so với các ảnh trong MNIST, các vật thể trong dữ liệu ImageNet thường có xu hướng chiếm nhiều điểm ảnh hơn. Do đó, ta cần sử dụng một cửa sổ tích chập lớn hơn để xác định được các vật thể này. Kích thước cửa sổ tích chập trong tầng thứ hai được giảm xuống còn  $5 \times 5$  và sau đó là  $3 \times 3$ . Ngoài ra, theo sau các tầng chập thứ nhất, thứ hai và thứ năm là các tầng gộp cực đại với kích thước cửa sổ là  $3 \times 3$  và sải bước bằng 2. Hơn nữa, số lượng các kênh tích chập trong AlexNet nhiều hơn gấp mười lần so với LeNet.

Sau tầng tích chập cuối cùng là hai tầng kết nối đầy đủ với 4096 đầu ra. Hai tầng này tạo ra tối gần 1 GB các tham số mô hình. Do các GPU thế hệ trước bị giới hạn về bộ nhớ, phiên bản gốc của AlexNet sử dụng thiết kế luồng dữ liệu kép cho hai GPU, trong đó mỗi GPU chỉ phải chịu trách nhiệm lưu trữ và tính toán cho một nửa mô hình. May mắn thay, hiện nay các GPU có bộ nhớ tương đối dồi dào, vì vậy ta hiếm khi cần phải chia nhỏ mô hình trên các GPU (phiên bản mô hình AlexNet của ta khác với bài báo ban đầu ở khía cạnh này).

## Các hàm Kích hoạt

Thứ hai, AlexNet đã thay hàm kích hoạt sigmoid bằng hàm kích hoạt ReLU đơn giản hơn. Một mặt là giảm việc tính toán, bởi ReLU không có phép lũy thừa như trong hàm kích hoạt sigmoid. Mặt khác, hàm kích hoạt ReLU giúp cho việc huấn luyện mô hình trở nên dễ dàng hơn khi sử dụng các phương thức khởi tạo tham số khác nhau. Điều này là do khi đầu ra của hàm kích hoạt sigmoid rất gần với 0 hoặc 1 thì gradient sẽ gần như bằng 0, vì vậy khiến cho 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 bằng 1. Do đó, nếu các tham số mô hình không được khởi tạo đúng cách thì hàm sigmoid có thể có gradient gần bằng 0 trong khoảng dương, dẫn đến việc mô hình không được huấn luyện một cách hiệu quả.

## Kiểm soát năng lực mô hình và Tiền xử lý

AlexNet kiểm soát năng lực của tầng kết nối đầy đủ bằng cách áp dụng dropout (Section 6.6), trong khi LeNet chỉ sử dụng suy giảm trọng số. Để tăng cường dữ liệu thì trong quá trình huấn luyện, AlexNet đã bổ sung rất nhiều kỹ thuật tăng cường hình ảnh chẳng hạn như lật, cắt hay thay đổi màu sắc. Điều này giúp cho mô hình trở nên mạnh mẽ hơn, cùng với đó kích thước dữ liệu lớn hơn giúp làm giảm hiện tượng quá khớp một cách hiệu quả. Ta sẽ thảo luận chi tiết hơn về việc tăng cường dữ liệu trong Section 15.1.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

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
net.add(nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
        nn.MaxPool2D(pool_size=3, strides=2),
        # Make the convolution window smaller, set padding to 2 for consistent
```

(continues on next page)

```

# height and width across the input and output, and increase the
# number of output channels
nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
nn.MaxPool2D(pool_size=3, strides=2),
# 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(384, kernel_size=3, padding=1, activation='relu'),
nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
nn.MaxPool2D(pool_size=3, strides=2),
# 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.Dense(4096, activation="relu"), nn.Dropout(0.5),
nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
# Output layer. Since we are using Fashion-MNIST, the number of
# classes is 10, instead of 1000 as in the paper
nn.Dense(10))

```

Ta sẽ tạo một thực thể dữ liệu đơn kênh với chiều cao và chiều rộng đều bằng 224 để quan sát kích thước đầu ra của mỗi tầng. Kết quả in ra khớp với sơ đồ bên trên.

```

X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)

```

### 9.1.3 Đọc Tập dữ liệu

Mặc dù AlexNet sử dụng ImageNet trong bài báo nêu trên, ở đây ta sẽ sử dụng Fashion-MNIST vì ngay cả với một GPU hiện đại thì việc huấn luyện một mô hình trên ImageNet có thể mất nhiều giờ hoặc nhiều ngày để hội tụ. Một trong những vấn đề khi áp dụng AlexNet trực tiếp trên Fashion-MNIST là các ảnh trong tập dữ liệu này có độ phân giải thấp hơn ( $28 \times 28$  điểm ảnh) so với các ảnh trong ImageNet. Để có thể tiến hành được thử nghiệm, ta sẽ nâng kích thước ảnh lên  $224 \times 224$  (nói chung đây không phải là một giải pháp thông minh, nhưng ta cần làm việc này để có thể sử dụng kiến trúc gốc của AlexNet). Việc thay đổi kích thước có thể được thực hiện thông qua đối số `resize` trong hàm `load_data_fashion_mnist`.

```

batch_size = 128
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)

```

#### 9.1.4 Huấn luyện

Bây giờ, ta có thể bắt đầu quá trình huấn luyện AlexNet. So với LeNet, thay đổi chính ở đây là việc sử dụng tốc độ học nhỏ hơn và quá trình huấn luyện chậm hơn nhiều do tính chất sâu và rộng hơn của mạng, đồng thời do độ phân giải hình ảnh cao hơn và việc tính toán các phép tích chập tốn kém hơn.

```
lr, num_epochs = 0.01, 10  
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr)
```

#### 9.1.5 Tóm tắt

- AlexNet có cấu trúc tương tự như LeNet, nhưng sử dụng nhiều tầng tích chập hơn với không gian tham số lớn hơn để khớp tập dữ liệu ImageNet với kích thước lớn.
- Ngày nay AlexNet đã bị vượt qua bởi các kiến trúc hiệu quả hơn nhiều nhưng nó là một bước quan trọng để đi từ các mạng nông đến các mạng sâu được sử dụng ngày nay.
- Mặc dù có vẻ như chỉ bổ sung thêm một vài dòng trong mã nguồn của AlexNet so với LeNet, nhưng công đồng học thuật đã phải mất nhiều năm để đón nhận sự thay đổi khái niệm này và tận dụng những kết quả thực nghiệm tuyệt vời của nó. Một phần cũng do sự thiếu thốn của các công cụ tính toán hiệu quả.
- Dropout, ReLU và tiền xử lý là những bước quan trọng khác để đạt được kết quả xuất sắc trong các bài toán thị giác máy tính.

#### 9.1.6 Bài tập

1. Thử tăng số lượng epoch và xem kết quả khác như thế nào so với LeNet? Tại sao lại có sự khác nhau như vậy?
2. AlexNet có thể là quá phức tạp đối với tập dữ liệu Fashion-MNIST. Vậy:
  - Hãy thử đơn giản hóa mô hình để làm cho việc huấn luyện trở nên nhanh hơn nhưng đồng thời vẫn đảm bảo độ chính xác không bị giảm đi đáng kể.
  - Hãy thử thiết kế một mô hình tốt hơn có thể hoạt động trực tiếp trên các ảnh có kích thước  $28 \times 28$ .
3. Điều chỉnh kích thước batch và quan sát các thay đổi về độ chính xác và việc tiêu thụ bộ nhớ GPU.
4. Bằng thông bộ nhớ hoạt động như thế nào khi tính toán các kết quả? Phần nào trong thiết kế của AlexNet có ảnh hưởng lớn đến:
  - Việc sử dụng bộ nhớ của mạng này?
  - Sự tính toán của mạng này?
5. Khi áp dụng dropout và ReLU cho LeNet5, độ chính xác của mô hình có được cải thiện hay không? Dữ liệu được tiền xử lý như thế nào?

### 9.1.7 Thảo luận

- Tiếng Anh<sup>163</sup>
- Tiếng Việt<sup>164</sup>

### 9.1.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp
- Nguyễn Cảnh Thưởng
- Phạm Hồng Vinh
- Nguyễn Đình Nam
- Nguyễn Mai Hoàng Long
- Nguyễn Lê Quang Nhật
- Đinh Đắc
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Thành Nhân

## 9.2 Mạng sử dụng Khối (VGG)

Mặc dù AlexNet đã chứng minh rằng các mạng nơ-ron tích chập có thể đạt được kết quả tốt, nó lại không cung cấp một khuôn mẫu chung để định hướng nghiên cứu sau này trong việc thiết kế các mạng mới. Trong các phần tiếp theo, chúng tôi sẽ giới thiệu một số khái niệm dựa trên thực nghiệm được sử dụng rộng rãi khi thiết kế mạng học sâu.

Sự phát triển trong lĩnh vực này có nét tương đồng tiến triển trong ngành thiết kế vi xử lý, chuyển từ việc sắp đặt các bóng bán dẫn, đến các phần tử logic và các khối logic.

Tương tự như vậy, việc thiết kế kiến trúc các mạng nơ-ron đã phát triển theo hướng ngày một trừu tượng hơn. Điển hình là việc các nhà nghiên cứu đã thay đổi suy nghĩ từ quy mô các nơ-ron riêng lẻ sang các tầng, và giờ đây là các khối chứa các tầng lặp lại theo khuôn mẫu.

Ý tưởng sử dụng các khối lần đầu xuất hiện trong mạng VGG, được đặt theo tên của nhóm VGG<sup>165</sup> thuộc Đại học Oxford. Sử dụng bất kỳ các framework học sâu hiện đại nào với vòng lặp và chương trình con để xây dựng các cấu trúc lặp lại này là tương đối dễ dàng.

<sup>163</sup> <https://discuss.mxnet.io/t/2354>

<sup>164</sup> <https://forum.machinelearningcoban.com/c/d21>

<sup>165</sup> <http://www.robots.ox.ac.uk/~vgg/>

### 9.2.1 Khối VGG

Khối cơ bản của mạng tích chập cổ điển là một chuỗi các tầng sau đây: (i) một tầng tích chập (với phần đậm để duy trì độ phân giải), (ii) một tầng phi tuyến như ReLU, (iii) một tầng gộp như tầng gộp cực đại. Một khối VGG gồm một chuỗi các tầng tích chập, tiếp nối bởi một tầng gộp cực đại để giảm chiều không gian. Trong bài báo gốc của VGG (Simonyan & Zisserman, 2014), tác giả sử dụng tích chập với các hạt nhân  $3 \times 3$  và tầng gộp cực đại  $2 \times 2$  với sải bước bằng 2 (giảm một nửa độ phân giải sau mỗi khối). Trong mã nguồn dưới đây, ta định nghĩa một hàm tên `vgg_block` để tạo một khối VGG. Hàm này nhận hai đối số `num_convs` và `num_channels` tương ứng lần lượt với số tầng tích chập và số kênh đầu ra.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def vgg_block(num_convs, num_channels):
    blk = nn.Sequential()
    for _ in range(num_convs):
        blk.add(nn.Conv2D(num_channels, kernel_size=3,
                        padding=1, activation='relu'))
    blk.add(nn.MaxPool2D(pool_size=2, strides=2))
    return blk
```

### 9.2.2 Mạng VGG

Giống như AlexNet và LeNet, mạng VGG có thể được phân chia thành hai phần: phần đầu tiên bao gồm chủ yếu các tầng tích chập và tầng gộp, còn phần thứ hai bao gồm các tầng kết nối đầy đủ. Phần tích chập của mạng gồm các mô-đun `vgg_block` kết nối liên tiếp với nhau. Trong Fig. 9.2.1, biến `conv_arch` bao gồm một danh sách các tuples (một tuple cho mỗi khối), trong đó mỗi tuple chứa hai giá trị: số lượng tầng tích chập và số kênh đầu ra, cũng chính là những đối số cần thiết để gọi hàm `vgg_block`. Mô-đun kết nối đầy đủ giống hệt với mô-đun tương ứng trong AlexNet.

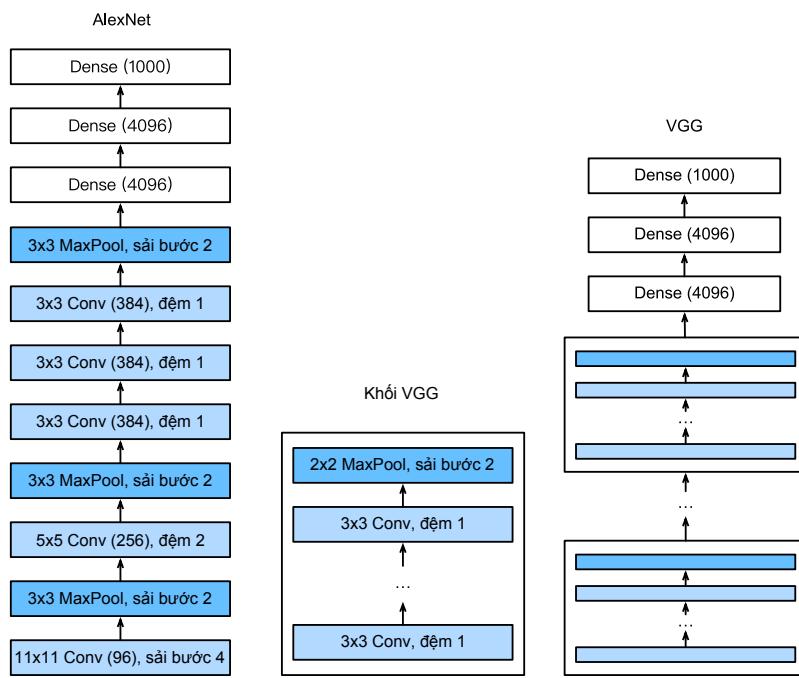


Fig. 9.2.1: Thiết kế mạng từ các khối cơ bản

Mạng VGG gốc có 5 khối tích chập, trong đó hai khối đầu tiên bao gồm một tầng tích chập ở mỗi khối, ba khối còn lại chứa hai tầng tích chập ở mỗi khối. Khối đầu tiên có 64 kênh đầu ra, mỗi khối tiếp theo nhân đôi số kênh đầu ra cho tới khi đạt giá trị 512. Vì mạng này sử dụng 8 tầng tích chập và 3 tầng kết nối đầy đủ nên nó thường được gọi là VGG-11.

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

Đoạn mã nguồn sau đây lập trình mạng VGG-11. Ở đây ta chỉ đơn thuần thực hiện vòng lặp for trên biến conv\_arch.

```
def vgg(conv_arch):
    net = nn.Sequential()
    # The convolutional layer part
    for (num_convs, num_channels) in conv_arch:
        net.add(vgg_block(num_convs, num_channels))
    # The fully connected layer part
    net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
            nn.Dense(10))
    return net

net = vgg(conv_arch)
```

Tiếp theo, chúng ta sẽ tạo một mẫu dữ liệu một kênh với chiều cao và chiều rộng là 224 để quan sát kích thước đầu ra của mỗi tầng.

```
net.initialize()
X = np.random.uniform(size=(1, 1, 224, 224))
for blk in net:
```

(continues on next page)

```
X = blk(X)
print(blk.name, 'output shape:', X.shape)
```

Như bạn thấy, ta đã giảm chiều cao và chiều rộng đi một nửa sau mỗi khối, cuối cùng kích thước của các biểu diễn chỉ còn là 7 trên mỗi chiều trước khi được trải phẳng ra để tiếp tục xử lý trong tầng kết nối đầy đủ.

### 9.2.3 Huấn luyện Mô hình

Vì VGG-11 thực hiện nhiều tính toán hơn AlexNet, ta sẽ xây dựng một mạng với số kênh nhỏ hơn. Như vậy vẫn là quá đủ để huấn luyện trên bộ dữ liệu Fashion-MNIST.

```
ratio = 4
small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
net = vgg(small_conv_arch)
```

Ngoại trừ việc sử dụng tốc độ học lớn hơn một chút, quy trình huấn luyện mô hình này tương tự như của AlexNet trong phần trước.

```
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)
```

### 9.2.4 Tóm tắt

- Mạng VGG-11 được xây dựng bằng cách tái sử dụng các khối tích chập. Các mô hình VGG khác nhau có thể được định nghĩa bằng cách thay đổi số lượng các tầng tích chập và số kênh đầu ra ở mỗi khối.
- Việc sử dụng các khối giúp ta định nghĩa mạng bằng các đoạn mã nguồn ngắn gọn và thiết kế các mạng phức tạp một cách hiệu quả hơn.
- Thử nghiệm nhiều kiến trúc khác nhau, Simonyan và Zisserman đã phát hiện rằng mạng có cửa sổ tích chập hẹp (như  $3 \times 3$ ) và nhiều tầng cho hiệu quả cao hơn mạng có cửa sổ tích chập rộng nhưng ít tầng.

### 9.2.5 Bài tập

- Khi in ra kích thước của các tầng, chúng ta chỉ thấy 8 kết quả chứ không phải 11. Thông tin về 3 tầng còn lại nằm ở đâu?
- So với AlexNet, VGG chậm hơn đáng kể về mặt tính toán và cũng đòi hỏi nhiều bộ nhớ GPU hơn. Hãy phân tích lý do cho hiện tượng này?
- Thử thay đổi chiều cao và chiều rộng của các ảnh trong Fashion-MNIST từ 224 xuống 96. Điều này ảnh hưởng thế nào tới các thử nghiệm?
- Tham khảo Bảng 1 trong ([Simonyan & Zisserman, 2014](#)) để xây dựng các mô hình thông dụng khác, ví dụ như là VGG-16 và VGG-19.

## 9.2.6 Thảo luận

- Tiếng Anh<sup>166</sup>
- Tiếng Việt<sup>167</sup>

## 9.2.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Văn Cường
- Nguyễn Văn Quang
- Nguyễn Cảnh Thuướng

## 9.3 Mạng trong Mạng (*Network in Network - NiN*)

LeNet, AlexNet và VGG đều có chung một khuôn mẫu thiết kế: trích xuất các đặc trưng khai thác cấu trúc *không gian* thông qua một chuỗi các phép tích chập và các tầng gộp, sau đó hậu xử lý các biểu diễn thông qua các tầng kết nối đầy đủ. Những cải tiến so với LeNet của AlexNet và VGG chủ yếu nằm ở việc mở rộng và tăng chiều sâu hai mô-đun này. Một lựa chọn khác là ta có thể sử dụng các tầng kết nối đầy đủ ngay từ giai đoạn trước. Tuy nhiên, việc tùy tiện sử dụng các tầng kết nối dày đặc có thể làm mất đi cấu trúc không gian của biểu diễn. Dùng các khối của Mạng trong Mạng (*Network in Network - NiN*) là một giải pháp thay thế khác. Ý tưởng này được đề xuất trong (Lin et al., 2013) dựa trên một thay đổi rất đơn giản — sử dụng MLP trên các kênh cho từng điểm ảnh riêng biệt.

### 9.3.1 Khối NiN

Hãy nhớ lại rằng đầu vào và đầu ra của các tầng tích chập là các mảng bốn chiều với các trực tương ứng với batch, kênh, chiều cao và chiều rộng. Đầu vào và đầu ra của các tầng kết nối đầy đủ thường là các mảng hai chiều tương ứng với batch và các đặc trưng. Ý tưởng chính của NiN là áp dụng một tầng kết nối đầy đủ tại mỗi vị trí điểm ảnh (theo chiều cao và chiều rộng). Nếu trọng số tại mỗi vị trí không gian được chia sẻ với nhau, ta có thể coi đây là một tầng chập  $1 \times 1$  (như được mô tả trong Section 8.4) hoặc như một tầng kết nối đầy đủ được áp dụng độc lập trên từng vị trí điểm ảnh. Nói theo một cách khác, ta có thể coi từng phần tử trong chiều không gian (chiều cao và chiều rộng) là tương đương với một mẫu và mỗi kênh tương đương với một đặc trưng. Fig. 9.3.1 minh họa sự khác biệt chính về cấu trúc giữa NiN và AlexNet, VGG cũng như các mạng khác.

<sup>166</sup> <https://discuss.mxnet.io/t/2355>

<sup>167</sup> <https://forum.machinelearningcoban.com/c/d21>

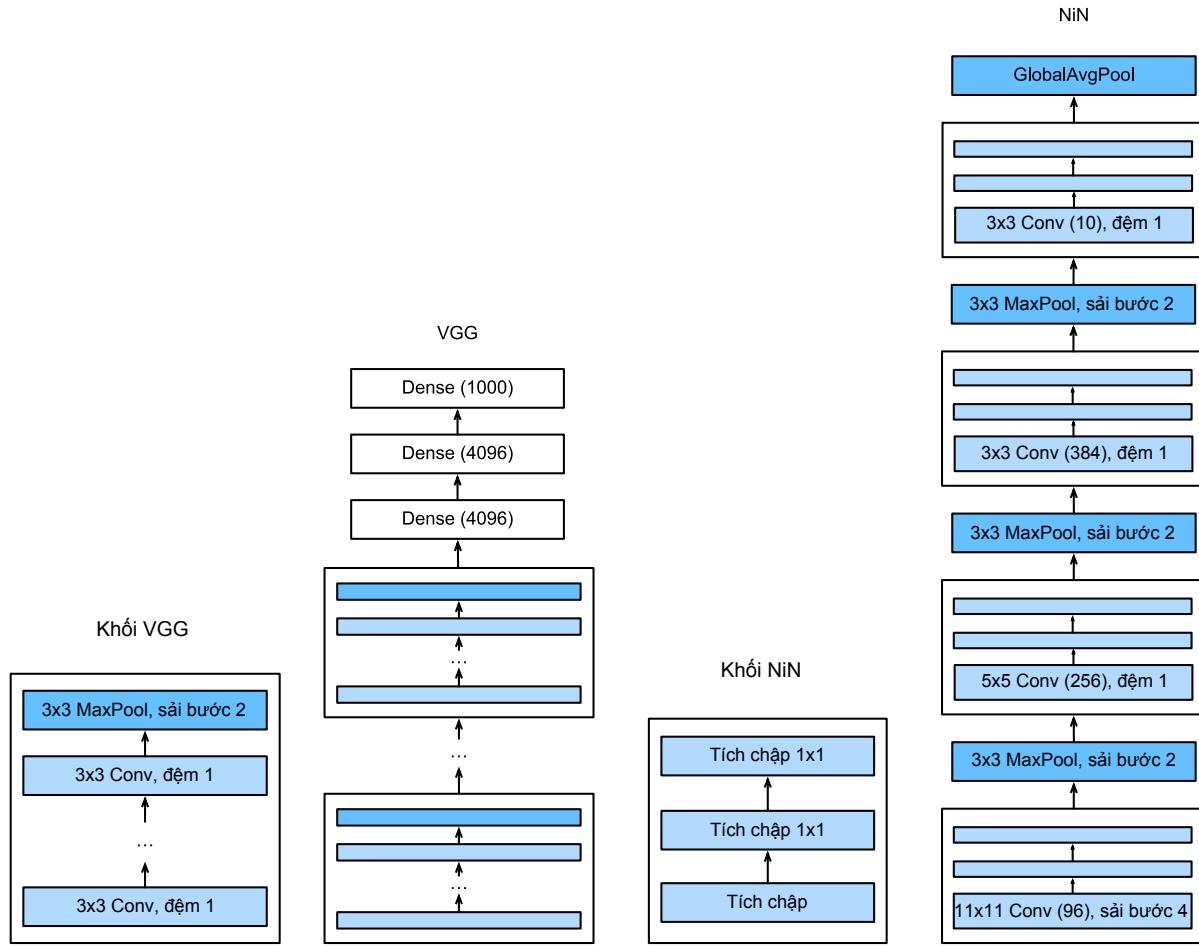


Fig. 9.3.1: Hình bên trái biểu diễn cấu trúc mạng của AlexNet và VGG, và hình bên phải biểu diễn cấu trúc mạng của NiN

Khối NiN bao gồm một tầng tích chập sau bởi hai tầng tích chập  $1 \times 1$  hoạt động như các tầng kết nối đầy đủ trên điểm ảnh và sau đó là hàm kích hoạt ReLU. Kích thước cửa sổ tích chập của tầng thứ nhất thường được định nghĩa bởi người dùng. Kích thước cửa sổ tích chập ở các tầng tiếp theo được cố định bằng  $1 \times 1$ .

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def nin_block(num_channels, kernel_size, strides, padding):
    blk = nn.Sequential()
    blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
                    activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
           nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
    return blk
```

### 9.3.2 Mô hình NiN

Cấu trúc mạng NiN gốc được đề xuất ngay sau và rõ ràng lấy cảm hứng từ mạng Alexnet. NiN sử dụng các tầng tích chập có kích thước cửa sổ  $11 \times 11$ ,  $5 \times 5$ ,  $3 \times 3$ , và số lượng các kênh đầu ra tương ứng giống với AlexNet. Mỗi khối NiN theo sau bởi một tầng gộp cực đại với sải bước 2 và kích thước cửa sổ  $3 \times 3$ .

Một điểm khác biệt đáng chú ý so với AlexNet là NiN tránh hoàn toàn việc sử dụng các kết nối dày đặc. Thay vào đó, mạng này sử dụng các khối NiN với số kênh đầu ra bằng với số lớp nhãn, theo sau bởi một tầng gộp trung bình *toàn cục*, tạo ra một vector logit<sup>168</sup>. Một lợi thế của thiết kế NiN là giảm được các tham số cần thiết của mô hình một cách đáng kể. Tuy nhiên, trong thực tế, cách thiết kế này đôi lúc đòi hỏi tăng thời gian huấn luyện mô hình.

```
net = nn.Sequential()
net.add(nin_block(96, kernel_size=11, strides=4, padding=0),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(256, kernel_size=5, strides=1, padding=2),
        nn.MaxPool2D(pool_size=3, strides=2),
        nin_block(384, kernel_size=3, strides=1, padding=1),
        nn.MaxPool2D(pool_size=3, strides=2),
        nn.Dropout(0.5),
        # There are 10 label classes
        nin_block(10, kernel_size=3, strides=1, padding=1),
        # The global average pooling layer automatically sets the window shape
        # to the height and width of the input
        nn.GlobalAvgPool2D(),
        # Transform the four-dimensional output into two-dimensional output
        # with a shape of (batch size, 10)
        nn.Flatten())
```

Chúng ta tạo một mẫu dữ liệu để kiểm tra kích thước đầu ra của từng khối.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)
```

### 9.3.3 Thu thập Dữ liệu và Huấn luyện

Như thường lệ, ta sẽ sử dụng Fashion-MNIST để huấn luyện mô hình. Quá trình huấn luyện NiN cũng tương tự như AlexNet và VGG, nhưng thường sử dụng tốc độ học lớn hơn.

```
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)
```

<sup>168</sup> <https://en.wikipedia.org/wiki/Logit>

#### 9.3.4 Tóm tắt

- NiN sử dụng các khối được cấu thành từ một tầng tích chập thông thường và nhiều tầng tích chập  $1 \times 1$ . Kỹ thuật này có thể dùng trong các khối tích chập để tăng tính phi tuyến trên điểm ảnh.
- NiN loại bỏ các tầng kết nối dày đủ và thay thế chúng bằng phép gộp trung bình toàn cục (nghĩa là tính trung bình cộng từ tất cả các vị trí) sau khi giảm số lượng kênh xuống bằng với số lượng đầu ra mong muốn (ví dụ: 10 kênh cho Fashion-MNIST).
- Việc bỏ đi các tầng dày đặc giúp làm giảm hiện tượng quá khớp. NiN có số lượng tham số ít hơn đáng kể.
- Thiết kế của NiN đã ảnh hưởng đến thiết kế của nhiều mạng nơ-ron tích chập sau này.

#### 9.3.5 Bài tập

1. Điều chỉnh các siêu tham số để cải thiện độ chính xác phân loại.
2. Tại sao có hai tầng chập  $1 \times 1$  trong khối NiN? Thử loại bỏ một trong số chúng, sau đó quan sát và phân tích các hiện tượng thực nghiệm.
3. Tính toán việc sử dụng tài nguyên của NiN với:
  - Số lượng tham số?
  - Số lượng phép tính?
  - Lượng bộ nhớ cần thiết trong quá trình huấn luyện?
  - Lượng bộ nhớ cần thiết trong quá trình dự đoán?
4. Các vấn đề nào sẽ sinh khi giảm biểu diễn từ  $384 \times 5 \times 5$  xuống  $10 \times 5 \times 5$  trong một bước?

#### 9.3.6 Thảo luận

- Tiếng Anh<sup>169</sup>
- Tiếng Việt<sup>170</sup>

#### 9.3.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Vũ Hữu Tiệp
- Nguyễn Duy Du
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Nguyễn Cảnh Thượng

<sup>169</sup> <https://discuss.mxnet.io/t/2356>

<sup>170</sup> <https://forum.machinelearningcoban.com/c/d21>

- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

## 9.4 Mạng nối song song (GoogLeNet)

Vào năm 2014, bài báo khoa học (Szegedy et al., 2015) đã giành chiến thắng ở cuộc thi ImageNet, bằng việc đề xuất một cấu trúc kết hợp những điểm mạnh của mô hình NiN và mô hình chứa các khối lặp lại. Bài báo này tập trung giải quyết câu hỏi: kích thước nào của bộ lọc tích chập là tốt nhất. Suy cho cùng, các mạng phổ biến trước đây chọn kích thước bộ lọc từ nhỏ như  $1 \times 1$  tới lớn như  $11 \times 11$ . Một góc nhìn sâu sắc trong bài báo này là đôi khi việc kết hợp các bộ lọc có kích thước khác nhau có thể sẽ hiệu quả. Trong phần này, chúng tôi sẽ giới thiệu mô hình GoogLeNet, bằng việc trình bày một phiên bản đơn giản hơn một chút so với phiên bản gốc—bỏ qua một số tính năng đặc biệt trước đây được thêm vào nhằm ổn định quá trình huấn luyện nhưng hiện nay không cần thiết nữa do đã có các thuật toán huấn luyện tốt hơn.

### 9.4.1 Khối Inception

Khối tích chập cơ bản trong mô hình GoogLeNet được gọi là Inception, nhiều khả năng được đặt tên dựa theo câu nói “Chúng ta cần đi sâu hơn” (“We Need To Go Deeper”) trong bộ phim Inception, sau này đã tạo ra một trào lưu lan rộng trên internet.

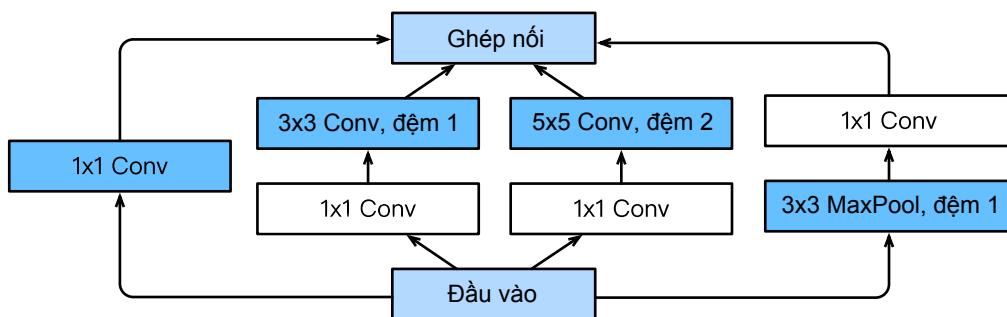


Fig. 9.4.1: Cấu trúc của khối Inception

Như mô tả ở hình trên, khối inception bao gồm bốn nhánh song song với nhau. Ba nhánh đầu sử dụng các tầng tích chập với kích thước cửa sổ trượt lần lượt là  $1 \times 1$ ,  $3 \times 3$ , và  $5 \times 5$  để trích xuất thông tin từ các vùng không gian có kích thước khác nhau. Hai nhánh giữa thực hiện phép tích chập  $1 \times 1$  trên dữ liệu đầu vào để giảm số kênh đầu vào, từ đó giảm độ phức tạp của mô hình. Nhánh thứ tư sử dụng một tầng gộp cực đại kích thước  $3 \times 3$ , theo sau là một tầng tích chập  $1 \times 1$  để thay đổi số lượng kênh. Cả bốn nhánh sử dụng phân đậm phù hợp để đầu vào và đầu ra của khối có cùng chiều cao và chiều rộng. Cuối cùng, các đầu ra của mỗi nhánh sẽ được nối lại theo chiều kênh để tạo thành đầu ra của cả khối. Các tham số thường được tinh chỉnh của khối Inception là số lượng kênh đầu ra mỗi tầng.

```

from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
  
```

(continues on next page)

```

class Inception(nn.Block):
    # c1 - c4 are the number of output channels for each layer in the path
    def __init__(self, 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(c1, kernel_size=1, activation='relu')
        # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
        # convolutional layer
        self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
        self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                             activation='relu')
        # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
        # convolutional layer
        self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
        self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                             activation='relu')
        # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
        # convolutional layer
        self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
        self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

    def forward(self, x):
        p1 = self.p1_1(x)
        p2 = self.p2_2(self.p2_1(x))
        p3 = self.p3_2(self.p3_1(x))
        p4 = self.p4_2(self.p4_1(x))
        # Concatenate the outputs on the channel dimension
        return np.concatenate((p1, p2, p3, p4), axis=1)

```

Để hiểu trực quan tại sao mạng này hoạt động tốt, hãy cùng tìm hiểu sự kết hợp của các bộ lọc. Chúng khám phá hình ảnh trên các vùng có kích thước khác nhau. Tức là những chi tiết ở những mức độ khác nhau sẽ được nhận diện một cách hiệu quả bằng các bộ lọc khác nhau. Đồng thời, chúng ta có thể phân bổ số lượng tham số khác nhau cho những vùng có phạm vi khác nhau (ví dụ: nhiều tham số hơn cho vùng phạm vi nhỏ nhưng không bù qua hoàn toàn vùng phạm vi lớn).

### 9.4.2 Mô hình GoogLeNet

Như trình bày ở Fig. 9.4.2, mô hình GoogLeNet sử dụng tổng cộng 9 khối inception và tầng gộp trung bình toàn cục xếp chồng lên nhau. Phép gộp cực đại giữa các khối inception có tác dụng làm giảm kích thước chiều. Phần đầu tiên của GoogleNet giống AlexNet và LeNet, có các khối xếp chồng lên nhau kế thừa từ thiết kế của VGG và phép gộp trung bình toàn cục giúp tránh phải sử dụng nhiều tầng kết nối đầy đủ liên tiếp ở cuối. Cấu trúc của mô hình được mô tả như dưới đây.

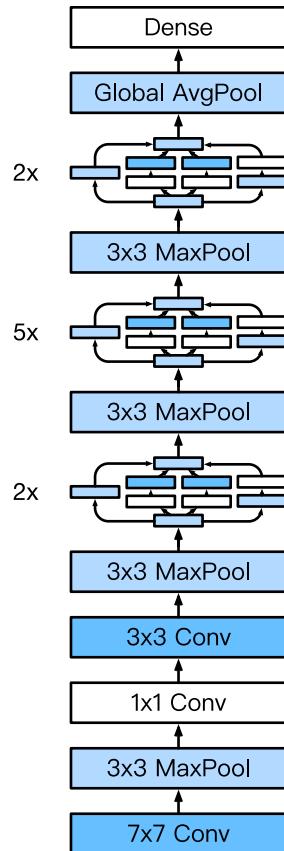


Fig. 9.4.2: Mô hình GoogLeNet đầy đủ

Bây giờ chúng ta có thể lập trình GoogLeNet theo từng phần. Thành phần đầu tiên sử dụng một tầng tích chập đầu ra 64 kênh và cửa sổ trượt kích thước  $7 \times 7$ .

```
b1 = nn.Sequential()
b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Thành phần thứ hai sử dụng hai tầng tích chập: tầng đầu tiên có đầu ra 64 kênh và cửa sổ  $1 \times 1$ , tiếp theo là một tầng có cửa sổ  $3 \times 3$  và số kênh đầu ra gấp ba lần số kênh đầu vào. Phần này giống với nhánh thứ hai trong khối Inception.

```
b2 = nn.Sequential()
b2.add(nn.Conv2D(64, kernel_size=1, activation='relu'),
       nn.Conv2D(192, kernel_size=3, padding=1, activation='relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Thành phần thứ ba kết nối hai khối Inception hoàn chỉnh một cách tuần tự. Số kênh đầu ra của khối Inception đầu tiên là  $64 + 128 + 32 + 32 = 256$ , và tỉ lệ số kênh của bốn nhánh là  $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$ . Nhánh thứ hai và nhánh thứ ba của khối này ở tầng tích chập đầu tiên làm giảm số lượng kênh đầu vào với tỉ lệ lần lượt là  $96/192 = 1/2$  và  $16/192 = 1/12$ , sau đó kết nối với tầng tích chập thứ hai. Số kênh đầu ra của khối Inception thứ hai tăng lên tới  $128 + 192 + 96 + 64 = 480$ , và tỉ lệ số kênh của bốn nhánh là  $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$ . Tầng tích chập đầu tiên của nhánh thứ hai và thứ ba làm giảm số kênh đầu vào với tỉ lệ lần lượt là  $128/256 = 1/2$  và  $32/256 = 1/8$ .

```
b3 = nn.Sequential()
b3.add(Inception(64, (96, 128), (16, 32), 32),
       Inception(128, (128, 192), (32, 96), 64),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Thành phần thứ tư phức tạp hơn. Thành phần này kết nối năm khối Inception có số kênh đầu ra lần lượt là  $192 + 208 + 48 + 64 = 512$ ,  $160 + 224 + 64 + 64 = 512$ ,  $128 + 256 + 64 + 64 = 512$ ,  $112 + 288 + 64 + 64 = 528$ , và  $256 + 320 + 128 + 128 = 832$ . Số kênh được gán cho các nhánh tương tự như trong mô đun thứ ba: nhánh thứ hai với tầng tích chập  $3 \times 3$  sẽ cho đầu ra với số kênh lớn nhất, tiếp theo là nhánh thứ nhất với tầng tích chập  $1 \times 1$ , nhánh thứ ba với tầng tích chập  $5 \times 5$ , cuối cùng là nhánh thứ tư với tầng gộp cực đại  $3 \times 3$ . Đầu tiên, nhánh thứ hai và thứ ba sẽ làm giảm số lượng kênh theo một tỷ lệ nhất định. Tỷ lệ này sẽ hơi khác nhau trong các khối Inception khác nhau.

```
b4 = nn.Sequential()
b4.add(Inception(192, (96, 208), (16, 48), 64),
       Inception(160, (112, 224), (24, 64), 64),
       Inception(128, (128, 256), (24, 64), 64),
       Inception(112, (144, 288), (32, 64), 64),
       Inception(256, (160, 320), (32, 128), 128),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Thành phần thứ năm có hai khối Inception với số kênh đầu ra lần lượt là  $256 + 320 + 128 + 128 = 832$  và  $384 + 384 + 128 + 128 = 1024$ . Số lượng kênh được gán cho mỗi nhánh tương tự như trong mô đun thứ ba và thứ tư, chỉ khác nhau ở giá trị cụ thể. Lưu ý rằng thành phần thứ năm được theo sau bởi tầng đầu ra. Thành phần này sử dụng tầng gộp trung bình toàn cục để giảm chiều cao và chiều rộng của mỗi kênh xuống còn 1, giống như trong mô hình NiN. Cuối cùng, chúng ta biến đổi đầu ra thành một mảng hai chiều, đưa vào một tầng kết nối đầy đủ với số đầu ra bằng số lượng lớp của nhẫn.

```
b5 = nn.Sequential()
b5.add(Inception(256, (160, 320), (32, 128), 128),
       Inception(384, (192, 384), (48, 128), 128),
       nn.GlobalAvgPool2D())

net = nn.Sequential()
net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

Mô hình GoogLeNet khá phức tạp về mặt tính toán, nên không dễ để thay đổi số lượng kênh giống như VGG. Để có thời gian huấn luyện hợp lý trên bộ dữ liệu Fashion-MNIST, chúng ta cần giảm chiều cao và chiều rộng của đầu vào từ 224 xuống 96. Điều này làm đơn giản hóa việc tính toán. Sự thay đổi ở kích thước đầu ra giữa các mô đun khác nhau được minh họa như dưới đây.

```
X = np.random.uniform(size=(1, 1, 96, 96))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:\t', X.shape)
```

### 9.4.3 Thu thập Dữ liệu và Huấn luyện

Vẫn như trước, chúng ta sử dụng tập dữ liệu Fashion-MNIST để huấn luyện mô hình. Chúng ta chuyển đổi độ phân giải hình ảnh thành  $96 \times 96$  điểm ảnh trước khi bắt đầu quá trình huấn luyện.

```
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)
```

### 9.4.4 Tóm tắt

- Khối Inception tương đương với một mạng con với bốn nhánh. Nó trích xuất thông tin một cách song song thông qua các tầng tích chập với kích thước cửa sổ khác nhau và các tầng gộp cực đại.
- Phép tích chập  $1 \times 1$  giảm số kênh ở mức độ điểm ảnh. Phép gộp cực đại giảm độ phân giải.
- Trong GoogLeNet, nhiều khối Inception với thiết kế khéo léo được nối với các tầng khác theo chuỗi. Tỷ lệ số kênh trong khối Inception được xác định dựa vào nhiều kết quả thử nghiệm trên tập dữ liệu ImageNet.
- Mô hình GoogLeNet, cũng như các phiên bản kế tiếp của nó, là một trong những mô hình hiệu quả nhất trên ImageNet, với độ chính xác tương tự trên tập kiểm tra nhưng độ phức tạp tính toán lại thấp hơn.

### 9.4.5 Bài tập

- Có nhiều biến thể của mô hình GoogLeNet. Hãy thử lập trình và chạy chúng. Một số biến thể bao gồm:
  - Thêm vào một tầng chuẩn hoá theo batch (Ioffe & Szegedy, 2015), như đã được mô tả ở phần [Section 9.5](#).
  - Chỉnh sửa khối Inception theo (Szegedy et al., 2016).
  - Sử dụng kỹ thuật “làm mượt nhãn” (*label smoothing*) để điều chỉnh mô hình (Szegedy et al., 2016).
  - Tích hợp nó vào kết nối phần dư (Szegedy et al., 2017), như mô tả trong phần sau [Section 9.6](#).
- Kích thước tối thiểu của hình ảnh với GoogLeNet là bao nhiêu?
- So sánh số lượng tham số mô hình của AlexNet, VGG và NiN với GoogLeNet. NiN và GoogLeNet đã giảm được đáng kể số lượng tham số như thế nào?
- Tại sao chúng ta cần tầng tích chập kích thước lớn ở đầu mạng?

#### 9.4.6 Thảo luận

- Tiếng Anh<sup>171</sup>
- Tiếng Việt<sup>172</sup>

#### 9.4.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đinh Đắc
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Nguyễn Văn Quang
- Phạm Minh Đức

### 9.5 Chuẩn hoá theo batch

Huấn luyện mạng nơ-ron sâu không hề đơn giản, để chúng hội tụ trong khoảng thời gian chấp nhận được là một câu hỏi khá hóc búa. Trong phần này, chúng ta giới thiệu chuẩn hóa theo batch (*Batch Normalization - BN*) ([Ioffe & Szegedy, 2015](#)), một kỹ thuật phổ biến và hiệu quả nhằm tăng tốc độ hội tụ của mạng học sâu một cách ổn định. Cùng với các khối phần dư (*residual block*) được đề cập ở [Section 9.6](#) — BN giúp dễ dàng hơn trong việc huấn luyện mạng học sâu với hơn 100 tầng.

#### 9.5.1 Huấn luyện mạng học sâu

Để thấy mục đích của việc chuẩn hóa theo batch, hãy cùng xem xét lại một vài vấn đề phát sinh trên thực tế khi huấn luyện các mô hình học máy và đặc biệt là mạng nơ-ron.

1. Những lựa chọn tiền xử lý dữ liệu khác nhau thường tạo nên sự khác biệt rất lớn trong kết quả cuối cùng. Hãy nhớ lại việc áp dụng perceptron đa tầng để dự đoán giá nhà ([Section 6.10](#)). Việc đầu tiên khi làm việc với dữ liệu thực tế là chuẩn hóa các đặc trưng đầu vào để chúng có giá trị trung bình bằng *không* và phương sai bằng *một*. Thông thường, việc chuẩn hóa này hoạt động tốt với các bộ tối ưu vì giá trị các tham số tiên nghiệm có cùng một khoảng tỷ lệ.
2. Khi huấn luyện các mạng thường gặp như Perceptron đa tầng hay CNN, các giá trị kích hoạt ở các tầng trung gian có thể nhận các giá trị với mức độ biến thiên lớn- dọc theo các tầng từ đầu vào đến đầu ra, qua các nút ở cùng một tầng, và theo thời gian do việc cập nhật giá trị tham số. Những nhà phát minh kỹ thuật chuẩn hóa theo batch cho rằng sự thay đổi trong phân phối của những giá trị kích hoạt có thể cản trở sự hội tụ của mạng. Để thấy rằng nếu một tầng có các giá trị kích hoạt lớn gấp 100 lần so với các tầng khác, thì cần phải có các điều chỉnh bổ trợ trong tốc độ học.
3. Mạng nhiều tầng có độ phức tạp cao và dễ gặp vấn đề quá khớp. Điều này cũng đồng nghĩa rằng kỹ thuật điều chỉnh càng trở nên quan trọng.

<sup>171</sup> <https://discuss.mxnet.io/t/2357>

<sup>172</sup> <https://forum.machinelearningcoban.com/c/d21>

Chuẩn hoá theo batch được áp dụng cho từng tầng riêng lẻ (hoặc có thể cho tất cả các tầng) và hoạt động như sau: Trong mỗi vòng lặp huấn luyện, tại mỗi tầng, đầu tiên tính giá trị kích hoạt như thường lệ. Sau đó chuẩn hóa những giá trị kích hoạt của mỗi nút bằng việc trừ đi giá trị trung bình và chia cho độ lệch chuẩn. Cả hai đại lượng này được ước tính dựa trên số liệu thống kê của minibatch hiện tại. Chính vì *chuẩn hóa* dựa trên các số liệu thống kê của *batch* nên kỹ thuật này có tên gọi *chuẩn hóa theo batch*.

Lưu ý rằng, khi áp dụng BN với những minibatch có kích thước 1, mô hình sẽ không học được gì. Vì sau khi trừ đi giá trị trung bình, mỗi nút ẩn sẽ nhận giá trị 0! Dễ dàng suy luận ra là BN chỉ hoạt động hiệu quả và ổn định với kích thước minibatch đủ lớn. Cần ghi nhớ rằng, khi áp dụng BN là lựa chọn kích thước minibatch quan trọng hơn so với trường hợp không áp dụng BN.

BN chuyển đổi những giá trị kích hoạt tại tầng  $x$  nhất định theo công thức sau:

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta \quad (9.5.1)$$

Ở đây,  $\hat{\mu}$  là giá trị trung bình và  $\hat{\sigma}$  là độ lệch chuẩn của các mẫu trong minibatch. Sau khi áp dụng BN, những giá trị kích hoạt của minibatch có giá trị trung bình bằng không và phương sai đơn vị. Vì việc lựa chọn phương sai đơn vị (so với một giá trị đặc biệt khác) là tuỳ ý, nên chúng ta thường thêm vào từng cặp tham số tương ứng là hệ số tỷ lệ  $\gamma$  và độ chênh  $\beta$ . Do đó, độ lớn giá trị kích hoạt ở những tầng trung gian không thể phân kỳ trong quá trình huấn luyện vì BN chủ động chuẩn hoá chúng theo giá trị trung bình và phương sai cho trước ( thông qua  $\mu$  và  $\sigma$ ). Qua trực giác và thực nghiệm, dùng BN có thể cho phép chọn tốc độ học nhanh hơn.

Ký hiệu một minibatch là  $\mathcal{B}$ , chúng ta tính  $\hat{\mu}_{\mathcal{B}}$  và  $\hat{\sigma}_{\mathcal{B}}$  theo công thức sau:

$$\hat{\mu}_{\mathcal{B}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ và } \hat{\sigma}_{\mathcal{B}}^2 \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_{\mathcal{B}})^2 + \epsilon \quad (9.5.2)$$

Lưu ý rằng chúng ta thêm hằng số rất nhỏ  $\epsilon > 0$  vào biểu thức tính phương sai để đảm bảo tránh phép chia cho 0 khi chuẩn hoá, ngay cả khi giá trị ước lượng phương sai thực nghiệm bằng không. Các ước lượng  $\hat{\mu}_{\mathcal{B}}$  và  $\hat{\sigma}_{\mathcal{B}}$  giúp đương đầu với vấn đề khi cần mở rộng số tầng của mạng (mạng học sâu hơn) bằng việc sử dụng nhiều khi tính giá trị trung bình và phương sai. Bạn có thể nghĩ rằng nhiều sẽ là vấn đề đáng ngại. Nhưng thực ra, nhiều lại đem đến lợi ích.

Và đây là chủ đề thường xuất hiện trong học sâu. Vì những lý do vẫn chưa được giải thích rõ bằng lý thuyết, nhiều nguồn nhiều khác nhau trong việc tối ưu hoá thường dẫn đến huấn luyện nhanh hơn và giảm quá khớp. Trong khi những nhà lý thuyết học máy truyền thống có thể bị vướng mắc ở việc định rõ điểm này, những thay đổi do nhiều đường như hoạt động giống một dạng điều chuẩn. Trong một số nghiên cứu sơ bộ, (Teye et al., 2018) và (Luo et al., 2018) đã lần lượt chỉ ra các thuộc tính của BN liên quan tới tiên nghiệm Bayesian (*Bayesian prior*) và các lượng phạt (*penalty*). Cụ thể, nghiên cứu này làm sáng tỏ lý do BN hoạt động tốt nhất với các minibatch có kích cỡ vừa phải, trong khoảng 50 - 100.

Cố định một mô hình đã được huấn luyện, bạn có thể nghĩ rằng chúng ta nên sử dụng toàn bộ tập dữ liệu để ước tính giá trị trung bình và phương sai. Và đúng là như vậy. Bởi lẽ khi huấn luyện xong, tại sao ta lại muốn cùng một hình ảnh lại được phân loại khác nhau phụ thuộc vào batch chứa hình ảnh này? Trong quá trình huấn luyện, những tính toán chính xác như vậy không khả thi vì giá trị kích hoạt cho tất cả các điểm dữ liệu thay đổi mỗi khi cập nhật mô hình. Tuy nhiên, một khi mô hình đã được huấn luyện xong, chúng ta có thể tính được giá trị trung bình và phương sai của mỗi tầng dựa trên toàn bộ tập dữ liệu. Thực ra đây là tiêu chuẩn thực hành cho các mô hình sử dụng chuẩn hóa theo batch và do đó các tầng BN của MXNet hoạt động khác nhau giữa

*chế độ huấn luyện* (chuẩn hoá bằng số liệu thống kê của minibatch) và *chế độ dự đoán* (chuẩn hoá bằng số liệu thống kê của toàn bộ tập dữ liệu)

Bây giờ chúng ta đã sẵn sàng để xem chuẩn hoá theo batch hoạt động thế nào trong thực tế.

### 9.5.2 Tầng chuẩn hoá theo batch

Thực hiện việc chuẩn hóa theo batch cho tầng kết nối đầy đủ và tầng tích chập hơi khác nhau một chút. Chúng ta sẽ thảo luận cả hai trường hợp trên. Nhớ rằng một khác biệt lớn của BN so với những tầng khác là vì BN cần số liệu thống kê trên toàn minibatch, chúng ta không thể bỏ qua kích thước batch như đã làm với các tầng khác.

#### Tầng kết nối đầy đủ

Khi áp dụng BN cho tầng kết nối đầy đủ, ta thường chèn BN sau bước biến đổi affine và trước hàm kích hoạt phi tuyến. Kí hiệu đầu vào của tầng là  $\mathbf{x}$ , hàm biến đổi tuyến tính là  $f_\theta(\cdot)$  (với trọng số là  $\theta$ ), hàm kích hoạt là  $\phi(\cdot)$ , và phép tính BN là  $\text{BN}_{\beta,\gamma}$  với tham số  $\beta$  và  $\gamma$ , chúng ta sẽ biểu diễn việc tính toán tầng kết nối đầy đủ  $\mathbf{h}$  khi chèn lớp BN vào như sau:

$$\mathbf{h} = \phi(\text{BN}_{\beta,\gamma}(f_\theta(\mathbf{x}))) \quad (9.5.3)$$

Nhắc lại rằng giá trị trung bình và phương sai sẽ được tính toán trên *chính minibatch  $\mathcal{B}$*  mà sẽ được biến đổi. Cũng cần lưu ý rằng hệ số tỷ lệ  $\gamma$  và độ chệch  $\beta$  là những tham số cần được học cùng với bộ tham số quen thuộc  $\theta$ .

#### Tầng tích chập

Tương tự với tầng tích chập, chúng ta áp dụng BN sau phép tích chập và trước hàm kích hoạt phi tuyến. Khi áp dụng phép tích chập cho đầu ra nhiều kênh, chúng ta cần thực hiện chuẩn hóa theo batch cho *mỗi* đầu ra của những kênh này, và *mỗi* kênh sẽ có riêng cho nó các tham số tỉ lệ và độ chệch, cả hai đều là các số vô hướng. Giả sử các minibatch có kích thước  $m$ , đầu ra cho *mỗi* kênh của phép tích chập có chiều cao  $p$  và chiều rộng  $q$ . Với tầng tích chập, ta sẽ thực hiện *mỗi* phép chuẩn hóa theo batch trên  $m \cdot p \cdot q$  phần tử trên từng kênh đầu ra cùng lúc. Vì thế trên từng kênh, ta sử dụng giá trị trên tất cả các vị trí không gian để tính trung bình  $\hat{\mu}$  và phương sai  $\hat{\sigma}$  và sau đó dùng hai giá trị này để chuẩn hóa các giá trị tại *mỗi* vị trí không gian trên kênh đó.

#### Chuẩn hoá theo Batch trong Quá trình Dự đoán

Như đã đề cập trước đó, BN thường hoạt động khác nhau trong chế độ huấn luyện và chế độ dự đoán. Thứ nhất, nhiều trong  $\mu$  và  $\sigma$  phát sinh từ việc chúng được xấp xỉ trên các minibatch không còn là nhiều được mong muốn một khi ta đã huấn luyện xong mô hình. Thứ hai, trong nhiều trường hợp sẽ là xa xỉ khi tính toán các con số thống kê sau mỗi lần chuẩn hóa theo batch, ví dụ, khi cần áp dụng mô hình để đưa ra một kết quả dự đoán mỗi lần.

Thông thường, sau khi huấn luyện, chúng ta sử dụng toàn bộ tập dữ liệu để tính toán các con số thống kê của các giá trị kích hoạt và sau đó cố định chúng tại thời điểm dự đoán. Do đó, BN hoạt động khác nhau trong quá trình huấn luyện và kiểm tra. Lưu ý rằng dropout cũng có tính chất này.

### 9.5.3 Lập trình từ đầu

Dưới đây, chúng ta lập trình tầng chuẩn hoá theo batch chỉ dùng ndarray.

```
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, init
from mxnet.gluon import nn
npx.set_np()

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether the current mode is training mode or
    # prediction mode
    if not autograd.is_training():
        # If it is the prediction mode, directly use the mean and variance
        # obtained from the incoming moving average
        X_hat = (X - moving_mean) / np.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(axis=0)
            var = ((X - mean) ** 2).mean(axis=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 broadcast operation
            # can be carried out later
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used for the
        # standardization
        X_hat = (X - mean) / np.sqrt(var + eps)
        # Update the mean and variance of the 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, moving_var
```

Giờ ta có thể tạo một tầng BatchNorm đúng cách. Tầng này sẽ duy trì những tham số thích hợp tương ứng với tỉ lệ gamma và độ chêch beta, hai tham số này sẽ được cập nhật trong quá trình huấn luyện. Thêm vào đó, tầng BN sẽ duy trì giá trị trung bình động của trung bình và phương sai để sử dụng về sau khi ở chế độ dự đoán. Tham số num\_features truyền vào BatchNorm là số đầu ra của tầng kết nối đầy đủ hoặc số kênh đầu ra của tầng tích chập. Tham số num\_dims bằng 2 nếu là tầng kết nối đầy đủ và bằng 4 nếu là tầng tích chập.

Tạm để thuật toán sang một bên và tập trung vào khuôn mẫu thiết kế (*design pattern*) của việc lập trình. Thông thường, ta lập trình phần toán trong một hàm riêng biệt, ví dụ như `batch_norm`. Sau đó, ta tích hợp chức năng này vào một tầng tùy chỉnh, với mã nguồn chủ yếu giải quyết các vấn đề phụ trợ như di chuyển dữ liệu đến thiết bị phù hợp, cấp phát và khởi tạo biến, theo dõi các giá trị trung bình động (của trung bình và phương sai trong trường hợp này), v.v. Khuôn mẫu này giúp tách biệt việc tính toán khỏi các đoạn mã rập khuôn. Cũng lưu ý rằng để thuận tiện khi lập trình BN từ đầu, ta không tự động suy ra kích thước đầu vào, do đó ta cần chỉ định số lượng đặc trưng xuyên suốt. Tầng BatchNorm của Gluon sẽ hỗ trợ việc tự động này bằng khởi tạo trễ.

```

class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter involved in gradient
        # finding and iteration are initialized to 0 and 1 respectively
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # All the variables not involved in gradient finding and iteration are
        # initialized to 0 on the CPU
        self.moving_mean = np.zeros(shape)
        self.moving_var = np.zeros(shape)

    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the
        # device where X is located
        if self.moving_mean.ctx != X.ctx:
            self.moving_mean = self.moving_mean.copyto(X.ctx)
            self.moving_var = self.moving_var.copyto(X.ctx)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-12, momentum=0.9)
        return Y

```

#### 9.5.4 Sử dụng LeNet với Chuẩn hóa theo Batch

Để biết cách áp dụng BatchNorm trên thực tế, bên dưới ta áp dụng cho mô hình LeNet truyền thống (Section 8.6). Nhắc lại rằng BN thường được sử dụng sau tầng tích chập và tầng kết nối đầy đủ và trước hàm kích hoạt tương ứng.

```

net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
       BatchNorm(6, num_dims=4),
       nn.Activation('sigmoid'),
       nn.MaxPool2D(pool_size=2, strides=2),
       nn.Conv2D(16, kernel_size=5),
       BatchNorm(16, num_dims=4),
       nn.Activation('sigmoid'),
       nn.MaxPool2D(pool_size=2, strides=2),
       nn.Dense(120),
       BatchNorm(120, num_dims=2),
       nn.Activation('sigmoid'),
       nn.Dense(84),
       BatchNorm(84, num_dims=2),
       nn.Activation('sigmoid'),
       nn.Dense(10))

```

Như thường lệ, ta sẽ huấn luyện trên bộ dữ liệu Fashion-MNIST. Đoạn mã này gần tương tự với đoạn mã khi lần đầu huấn luyện LeNet (Section 8.6). Điểm khác biệt chính là tốc độ học lớn hơn đáng kể.

```
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)
```

Chúng ta hãy xem tham số tỷ lệ gamma và tham số dịch chuyển beta đã học được tại tầng chuẩn hóa theo batch đầu tiên.

```
net[1].gamma.data().reshape(-1,), net[1].beta.data().reshape(-1,)
```

### 9.5.5 Lập trình súc tích

So với lớp BatchNorm tự định nghĩa thì lớp BatchNorm định nghĩa trong nn của Gluon dễ sử dụng hơn. Trong Gluon, ta không cần chỉ rõ num\_features và num\_dims. Thay vào đó, các giá trị này sẽ được tự động suy ra trong quá trình khởi tạo trễ. Ngoại trừ điểm đó, đoạn mã trông giống hệt đoạn mã phía trên.

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
       nn.BatchNorm(),
       nn.Activation('sigmoid'),
       nn.MaxPool2D(pool_size=2, strides=2),
       nn.Conv2D(16, kernel_size=5),
       nn.BatchNorm(),
       nn.Activation('sigmoid'),
       nn.MaxPool2D(pool_size=2, strides=2),
       nn.Dense(120),
       nn.BatchNorm(),
       nn.Activation('sigmoid'),
       nn.Dense(84),
       nn.BatchNorm(),
       nn.Activation('sigmoid'),
       nn.Dense(10))
```

Chúng ta sử dụng cùng các siêu tham số như trước để huấn luyện mô hình. Như thường lệ, biến thể dùng Gluon này chạy nhanh hơn nhiều vì được biên dịch thành C++/CUDA trong khi đoạn mã tùy chỉnh của chúng ta phải qua thông dịch bằng Python.

```
d2l.train_ch6(net, train_iter, test_iter, num_epochs, lr)
```

### 9.5.6 Tranh luận

Theo trực giác, chuẩn hóa theo batch được cho là làm cảnh quan tối ưu (*optimization landscape*) mượt mà hơn. Tuy nhiên, cần cẩn thận phân biệt giữa suy đoán theo trực giác và lời giải thích thực sự cho các hiện tượng quan sát thấy khi huấn luyện các mô hình học sâu. Hãy nhớ lại rằng ngay từ đầu ta thậm chí không rõ tại sao các mạng nơ-ron sâu đơn giản hơn (như Perceptron đa tầng và CNN truyền thống) lại có thể khái quát tốt như vậy. Ngay cả với dropout và điều chỉnh L2, chúng vẫn linh hoạt đến mức khả năng khái quát hóa trên dữ liệu chưa nhìn thấy của chúng không thể giải thích được bằng các điều kiện bảo đảm sự khái quát hóa trong lý thuyết học truyền thống.

Trong bài báo gốc khi đề xuất phương pháp chuẩn hóa theo batch, các tác giả ngoài việc giới thiệu một công cụ mạnh mẽ và hữu ích đã đưa ra lời giải thích lý do BN hoạt động tốt: bằng cách giảm *sự dịch chuyển hiệp biến nội bộ* - *internal covariate shift*. Có thể hiểu ý các tác giả về *sự dịch chuyển hiệp biến nội bộ* giống với cách giải thích ở trên-rằng phân phối của giá trị kích hoạt thay đổi trong quá trình huấn luyện. Tuy nhiên, có hai vấn đề với cách giải thích này: (1) Sự dịch chuyển phân phối này rất khác so với *sự dịch chuyển hiệp biến*, việc đặt tên như vậy có sự nhầm lẫn. (2) Cách giải thích này vẫn chưa đủ cụ thể và chặt chẽ, vẫn để ngỏ câu hỏi: *chính xác thì tại sao kỹ thuật này hoạt động?* Xuyên suốt cuốn sách này, chúng tôi hướng đến việc truyền đạt những kinh nghiệm thực tế để xây dựng các mạng nơ-ron sâu. Tuy nhiên, chúng tôi tin rằng cần phân biệt rõ những kinh nghiệm dựa trên trực giác này với những bằng chứng khoa học rõ ràng. Cuối cùng, khi đã thành thạo tài liệu này và bắt đầu viết các nghiên cứu của riêng mình, bạn cần phân biệt rõ ràng giữa khẳng định và linh cảm.

Nối tiếp thành công của BN, cách giải thích của kỹ thuật này thông qua khái niệm *sự dịch chuyển hiệp biến nội bộ* liên tục xuất hiện trong các tranh luận, các tài liệu kỹ thuật và trên các diễn đàn về cách trình bày nghiên cứu học máy. Trong một bài phát biểu đáng nhớ được đưa ra khi nhận giải thưởng **Test of Time Award** tại hội nghị NeurIPS 2017, Ali Rahimi đã sử dụng *sự dịch chuyển hiệp biến nội bộ* như một tiêu điểm trong một cuộc tranh luận so sánh thực hành học sâu hiện đại với thuật giả kim. Sau đó, cách giải thích này đã được xem xét lại một cách chi tiết trong một bài báo về các xu hướng đáng lo ngại trong học máy ([Lipton & Steinhardt, 2018](#)). Trong các tài liệu kỹ thuật, các tác giả khác ([\(Santurkar et al., 2018\)](#)) đã đề xuất các giải thích thay thế cho sự thành công của BN, dù phần nào đó trái ngược với cách giải thích trong bài báo gốc.

Chúng tôi lưu ý rằng *sự dịch chuyển hiệp biến nội bộ* không đáng bị chỉ trích, có hàng ngàn lập luận mơ hồ được đưa ra mỗi năm trong nhiều tài liệu kỹ thuật về học máy. Việc nó trở thành tâm điểm của những cuộc tranh luận rất có thể là do sự phổ biến của nó trong cộng đồng học máy. Chuẩn hóa theo batch là một phương pháp quan trọng, được áp dụng trong gần như tất cả các bộ phân loại hình ảnh đã được triển khai, mang lại hàng chục ngàn trích dẫn cho bài báo giới thiệu kĩ thuật này.

### 9.5.7 Tóm tắt

- Trong quá trình huấn luyện mô hình, chuẩn hóa theo batch liên tục điều chỉnh đầu ra trung gian của mạng nơ-ron theo giá trị trung bình và độ lệch chuẩn của minibatch, giúp các giá trị này ổn định hơn.
- Chuẩn hóa theo batch có chút khác biệt khi áp dụng cho tầng kết nối đầy đủ và tầng tích chập.
- Giống như tầng dropout, tầng chuẩn hóa theo batch sẽ tính ra kết quả khác nhau trong chế độ huấn luyện và chế độ dự đoán.
- Chuẩn hóa theo batch có nhiều tác dụng phụ có lợi, chủ yếu là về điều kiện. Tuy nhiên, cách giải thích ban đầu về việc giảm *sự dịch chuyển hiệp biến* dường như không hợp lý.

### 9.5.8 Bài tập

1. Trước khi chuẩn hóa theo batch, có thể loại bỏ phép biến đổi affine trong tầng kết nối đầy đủ hoặc tham số độ chêch trong phép tích chập không?
  - Tìm một phép biến đổi tương đương được áp dụng trước tầng kết nối đầy đủ.
  - Sự cải tiến này có hiệu quả không, tại sao?
2. So sánh tốc độ học của LeNet khi có sử dụng và không sử dụng chuẩn hóa theo batch.
  - Vẽ đồ thị biểu diễn sự giảm xuống của lỗi huấn luyện và lỗi kiểm tra.
  - Về miền hội tụ thì sao? Có thể chọn tốc độ học lớn tới đâu?
3. Chúng ta có cần chuẩn hóa theo batch trong tất cả các tầng không? Hãy thử nghiệm điều này.
4. Có thể thay thế Dropout bằng BN không? Sẽ có thay đổi như thế nào?
5. Giữ nguyên các hệ số beta và gamma (thêm tham số grad\_req='null' khi xây dựng mạng để không tính gradient) rồi quan sát và phân tích kết quả.
6. Đọc tài liệu của Gluon về BatchNorm để xem các ứng dụng khác của chuẩn hóa theo batch.
7. Ý tưởng nghiên cứu: nghĩ về các phép biến đổi chuẩn hóa khác có thể áp dụng. Bạn có thể áp dụng biến đổi tích phân xác suất (*probability integral transform*) không? Còn ước lượng ma trận hiệp phương sai hạng tối đa thì sao?

### 9.5.9 Thảo luận

- Tiếng Anh<sup>173</sup>
- Tiếng Việt<sup>174</sup>

### 9.5.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đinh Đắc
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Trần Yến Thy
- Phạm Minh Đức
- Nguyễn Cảnh Thượng
- Phạm Hồng Vinh

<sup>173</sup> <https://discuss.mxnet.io/t/2358>

<sup>174</sup> <https://forum.machinelearningcoban.com/c/d21>

## 9.6 Mạng phần dư (ResNet)

Khi thiết kế các mạng ngày càng sâu, ta cần hiểu việc thêm các tầng sẽ tăng độ phức tạp và khả năng biểu diễn của mạng như thế nào. Quan trọng hơn là khả năng thiết kế các mạng trong đó việc thêm các tầng vào mạng chắc chắn sẽ làm tăng tính biểu diễn thay vì chỉ tạo ra một chút khác biệt. Để làm được điều này, chúng ta cần một chút lý thuyết.

### 9.6.1 Các Lớp Hàm Số

Coi  $\mathcal{F}$  là một lớp các hàm mà một kiến trúc mạng cụ thể (cùng với tốc độ học và các siêu tham số khác) có thể đạt được. Nói cách khác, với mọi hàm số  $f \in \mathcal{F}$ , luôn tồn tại một số tập tham số  $W$  có thể tìm được bằng việc huấn luyện trên một tập dữ liệu phù hợp. Giả sử  $f^*$  là hàm cần tìm. Sẽ rất thuận lợi nếu hàm này thuộc tập  $\mathcal{F}$ , nhưng thường không may mắn như vậy. Thay vào đó, ta sẽ cố gắng tìm các hàm số  $f_{\mathcal{F}}^*$  tốt nhất có thể trong tập  $\mathcal{F}$ .

Ví dụ, có thể thử tìm  $f_{\mathcal{F}}^*$  bằng cách giải bài toán tối ưu sau:

$$f_{\mathcal{F}}^* := \underset{f}{\operatorname{argmin}} L(X, Y, f) \text{ đối tượng thoả mãn } f \in \mathcal{F}. \quad (9.6.1)$$

Khá hợp lý khi giả sử rằng nếu thiết kế một kiến trúc khác  $\mathcal{F}'$  mạnh mẽ hơn thì sẽ đạt được kết quả tốt hơn. Nói cách khác, ta kỳ vọng hàm số  $f_{\mathcal{F}}^*$  sẽ “tốt hơn”  $f_{\mathcal{F}}^*$ . Tuy nhiên, nếu  $\mathcal{F} \not\subseteq \mathcal{F}'$ , thì không khẳng định được  $f_{\mathcal{F}}^*$  “tốt hơn”  $f_{\mathcal{F}}^*$ . Trên thực tế,  $f_{\mathcal{F}}^*$  có thể còn tệ hơn. Và đây là trường hợp thường xuyên xảy ra — việc thêm các tầng không phải lúc nào cũng tăng tính biểu diễn của mạng mà đôi khi còn tạo ra những thay đổi rất khó lường. Fig. 9.6.1 minh họa rõ hơn điều này.

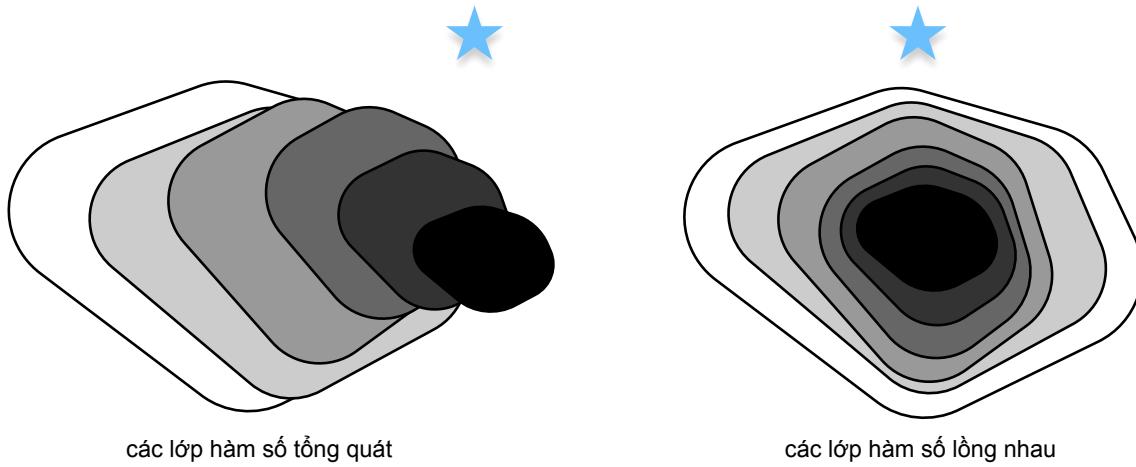


Fig. 9.6.1: Hình trái: Các lớp hàm số tổng quát. Khoảng cách đến hàm cần tìm  $f^*$  (ngôi sao), trên thực tế có thể tăng khi độ phức tạp tăng lên. Hình phải: với các lớp hàm số lồng nhau, điều này không xảy ra.

Chỉ khi các lớp hàm lớn hơn chứa các lớp nhỏ hơn, thì mới đảm bảo rằng việc tăng thêm các tầng sẽ tăng khả năng biểu diễn của mạng. Đây là câu hỏi mà He và các cộng sự đã suy nghĩ khi nghiên cứu các mô hình thị giác sâu năm 2016. Ý tưởng trọng tâm của ResNet là mỗi tầng được thêm vào nên có một thành phần là hàm số đồng nhất. Điều này có nghĩa rằng, nếu ta huấn luyện tầng mới được thêm vào thành một ánh xạ đồng nhất  $f(\mathbf{x}) = \mathbf{x}$ , thì mô hình mới sẽ hiệu quả ít nhất bằng

mô hình ban đầu. Vì tầng được thêm vào có thể khớp dữ liệu huấn luyện tốt hơn, dẫn đến sai số huấn luyện cũng nhỏ hơn. Tốt hơn nữa, hàm số đồng nhất nên là hàm đơn giản nhất trong một tầng thay vì hàm null  $f(\mathbf{x}) = 0$ .

Cách suy nghĩ này khá trừu tượng nhưng lại dẫn đến một lời giải đơn giản đáng ngạc nhiên, một khối phần dư (*residual block*). Với ý tưởng này, (He et al., 2016a) đã chiến thắng cuộc thi Nhận dạng Ảnh ImageNet năm 2015. Thiết kế này có ảnh hưởng sâu sắc tới việc xây dựng các mạng nơ-ron sâu.

### 9.6.2 Khối phần dư

Bây giờ, hãy tập trung vào mạng nơ-ron dưới đây. Ký hiệu đầu vào là  $\mathbf{x}$ . Giả sử ánh xạ lý tưởng muốn học được là  $f(\mathbf{x})$ , và được dùng làm đầu vào của hàm kích hoạt. Phần nằm trong viền nét đứt bên trái phải khớp trực tiếp với ánh xạ  $f(\mathbf{x})$ . Điều này có thể không đơn giản nếu chúng ta không cần khối đó và muốn giữ lại đầu vào  $\mathbf{x}$ . Khi đó, phần nằm trong viền nét đứt bên phải chỉ cần tham số hóa *độ lệch* khỏi giá trị  $\mathbf{x}$ , bởi vì ta đã trả về  $\mathbf{x} + f(\mathbf{x})$ . Trên thực tế, ánh xạ phần dư thường dễ tối ưu hơn, vì chỉ cần đặt  $f(\mathbf{x}) = 0$ . Nửa bên phải Fig. 9.6.2 mô tả khối phần dư cơ bản của ResNet. Về sau, những kiến trúc tương tự đã được đề xuất cho các mô hình chuỗi (*sequence model*), sẽ đề cập ở chương sau.

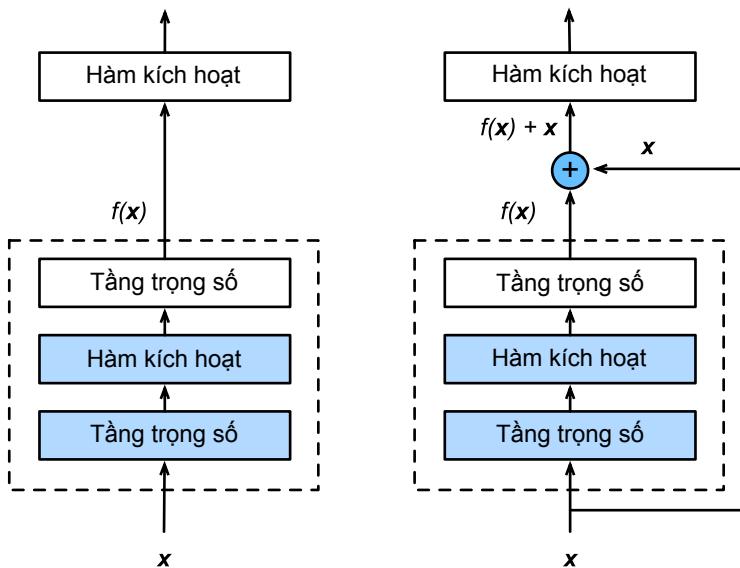


Fig. 9.6.2: Sự khác biệt giữa một khối thông thường (trái) và một khối phần dư (phải). Trong khối phần dư, ta có thể nối tắt các tích chập.

ResNet có thiết kế tầng tích chập  $3 \times 3$  giống VGG. Khối phần dư có hai tầng tích chập  $3 \times 3$  với cùng số kênh đầu ra. Mỗi tầng tích chập được sau bởi một tầng chuẩn hóa theo batch và một hàm kích hoạt ReLU. Ta đưa đầu vào qua khối phần dư rồi cộng với chính nó trước hàm kích hoạt ReLU cuối cùng. Thiết kế này đòi hỏi đầu ra của hai tầng tích chập phải có cùng kích thước với đầu vào, để có thể cộng lại với nhau. Nếu muốn thay đổi số lượng kênh hoặc sải bước trong khối phần dư, cần thêm một tầng tích chập  $1 \times 1$  để thay đổi kích thước đầu vào tương ứng ở nhánh ngoài. Hãy cùng xem đoạn mã bên dưới.

```

from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# Saved in the d2l package for later use
class Residual(nn.Block):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                            strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return npx.relu(Y + X)

```

Đoạn mã này tạo ra hai loại mạng: một loại cộng đầu vào vào đầu ra trước khi áp dụng hàm phi tuyến ReLU (khi use\_1x1conv=True), còn ở loại thứ hai chúng ta thay đổi số kênh và độ phân giải bằng một tầng tích chập  $1 \times 1$  trước khi thực hiện phép cộng. Fig. 9.6.3 minh họa điều này:

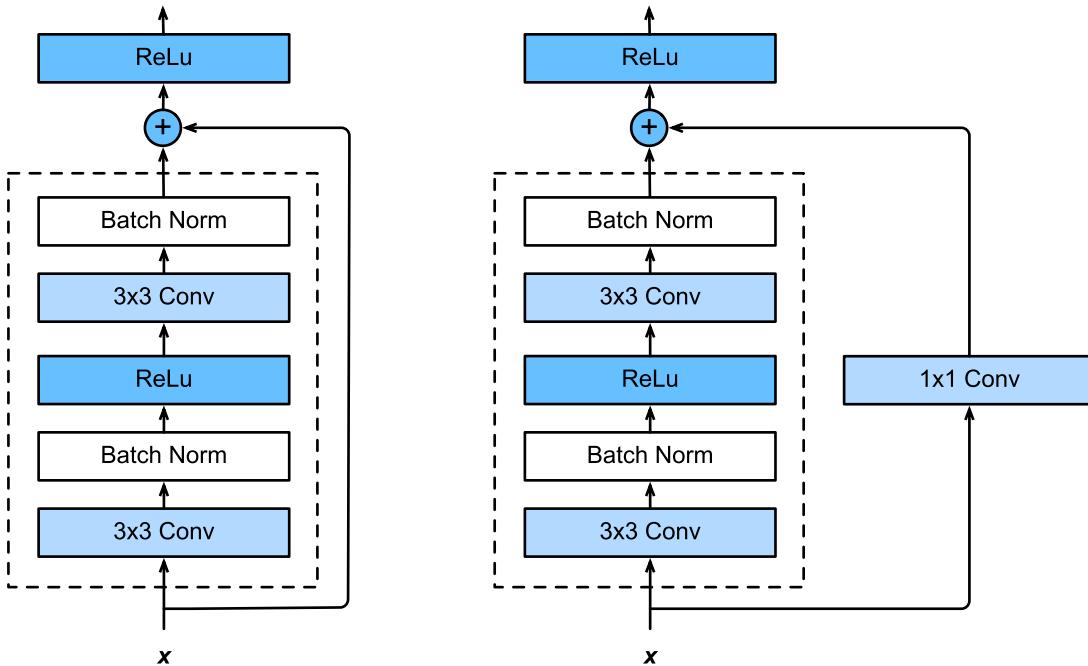


Fig. 9.6.3: Trái: khối ResNet thông thường; Phải: Khối ResNet với tầng tích chập 1x1

Giờ hãy xem xét tình huống khi cả đầu vào và đầu ra có cùng kích thước.

```
blk = Residual(3)
blk.initialize()
X = np.random.uniform(size=(4, 3, 6, 6))
blk(X).shape
```

Chúng ta cũng có thể giảm một nửa kích thước chiều cao và chiều rộng của đầu ra trong khi tăng số kênh.

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk.initialize()
blk(X).shape
```

### 9.6.3 Mô hình ResNet

Hai tầng đầu tiên của ResNet giống hai tầng đầu tiên của GoogLeNet: tầng tích chập  $7 \times 7$  với 64 kênh đầu ra và sải bước 2, theo sau bởi tầng gộp cực đại  $3 \times 3$  với sải bước 2. Sự khác biệt là trong ResNet, mỗi tầng tích chập sau bởi tầng chuẩn hóa theo batch.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
       nn.BatchNorm(), nn.Activation('relu'),
       nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet sử dụng bốn mô-đun được tạo thành từ các khối Inception. ResNet sử dụng bốn mô-đun được tạo thành từ các khối phần dư có cùng số kênh đầu ra. Mô-đun đầu tiên có số kênh bằng số kênh đầu vào. Vì trước đó đã sử dụng tầng gộp cực đại với sải bước 2, nên không cần phải giảm chiều cao và chiều rộng ở mô-đun này. Trong các mô-đun sau, khối phần dư đầu tiên nhân đôi số kênh, đồng thời giảm một nửa chiều cao và chiều rộng.

Bây giờ ta sẽ lập trình mô-đun này. Chú ý rằng mô-đun đầu tiên được xử lý khác một chút.

```
def resnet_block(num_channels, num_residuals, first_block=False):
    blk = nn.Sequential()
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.add(Residual(num_channels))
    return blk
```

Sau đó, chúng ta thêm các khối phần dư vào ResNet. Ở đây, mỗi mô-đun có hai khối phần dư.

```
net.add(resnet_block(64, 2, first_block=True),
        resnet_block(128, 2),
        resnet_block(256, 2),
        resnet_block(512, 2))
```

Cuối cùng, giống như GoogLeNet, ta thêm một tầng gộp trung bình toàn cục và một tầng kết nối đầy đủ.

```
net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

Có 4 tầng tích chập trong mỗi mô-đun (không tính tầng tích chập  $1 \times 1$ ). Công thêm tầng tích chập đầu tiên và tầng kết nối đầy đủ cuối cùng, mô hình có tổng cộng 18 tầng. Do đó, mô hình này thường được gọi là ResNet-18. Có thể thay đổi số kênh và các khối phần dư trong mô-đun để tạo ra các mô hình ResNet khác nhau, ví dụ mô hình 152 tầng của ResNet-152. Mặc dù có kiến trúc lõi tương tự như GoogLeNet, cấu trúc của ResNet đơn giản và dễ sửa đổi hơn. Tất cả các yếu tố này dẫn đến sự phổ cập nhanh chóng và rộng rãi của ResNet. Fig. 9.6.4 là sơ đồ đầy đủ của ResNet-18.

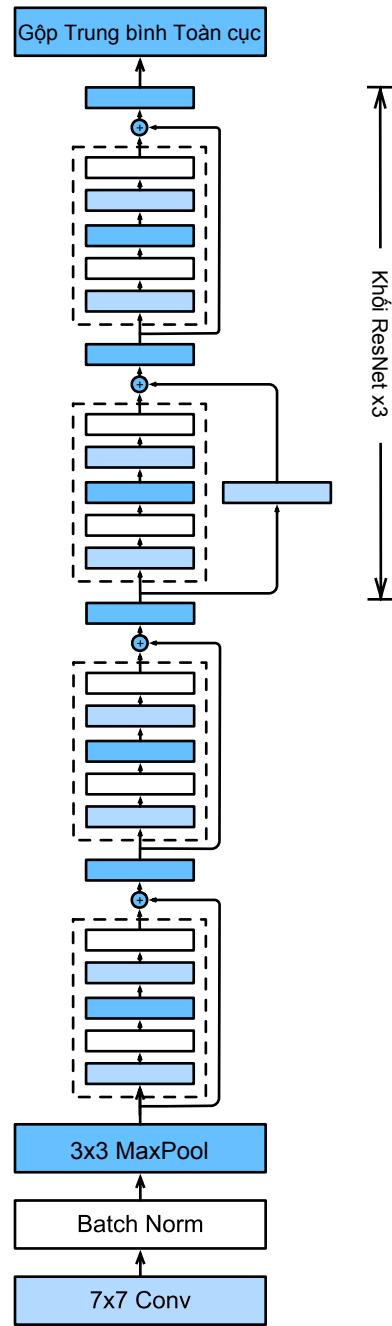


Fig. 9.6.4: ResNet-18

Trước khi huấn luyện, hãy quan sát thay đổi của kích thước đầu vào qua các mô-đun khác nhau

trong ResNet. Như trong tất cả các kiến trúc trước, độ phân giải giảm trong khi số lượng kênh tăng đến khi tầng gộp trung bình toàn cục tổng hợp tất cả các đặc trưng.

```
X = np.random.uniform(size=(1, 1, 224, 224))
net.initialize()
for layer in net:
    X = layer(X)
    print(layer.name, 'output shape:', X.shape)
```

#### 9.6.4 Thu thập dữ liệu và Huấn luyện

Giống như các phần trước, chúng ta huấn luyện ResNet trên bộ dữ liệu Fashion-MNIST. Thay đổi duy nhất là giảm tốc độ học lại do kiến trúc mạng phức tạp hơn.

```
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)
```

#### 9.6.5 Tóm tắt

- Khối phần dư cho phép tham số hóa đến hàm đồng nhất  $f(\mathbf{x}) = \mathbf{x}$ .
- Thêm các khối phần dư làm tăng độ phức tạp của hàm số theo một cách chủ đích.
- Chúng ta có thể huấn luyện hiệu quả mạng nơ-ron sâu nhờ khối phần dư chuyển dữ liệu liên tầng.
- ResNet có ảnh hưởng lớn đến thiết kế sau này của các mạng nơ-ron sâu, cả tích chập và tuần tự.

#### 9.6.6 Bài tập

- Tham khảo Bảng 1 trong (He et al., 2016a) để lập trình các biến thể khác nhau.
- Đối với các mạng sâu hơn, ResNet giới thiệu kiến trúc “thắt cổ chai” để giảm độ phức tạp của mô hình. Hãy thử lập trình kiến trúc đó.
- Trong các phiên bản sau của ResNet, tác giả đã thay đổi kiến trúc “tích chập, chuẩn hóa theo batch, và hàm kích hoạt” thành “chuẩn hóa theo batch, hàm kích hoạt, và tích chập”. Hãy tự lập trình kiến trúc này. Xem hình 1 trong (He et al., 2016b) để biết chi tiết.
- Chứng minh rằng nếu  $\mathbf{x}$  được tạo ra bởi ReLU thì khối ResNet sẽ bao gồm hàm số đồng nhất.
- Tại sao không thể tăng không giới hạn độ phức tạp của các hàm số, ngay cả với các lớp hàm lồng nhau?

### 9.6.7 Thảo luận

- Tiếng Anh<sup>175</sup>
- Tiếng Việt<sup>176</sup>

### 9.6.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Cảnh Thưởng
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Nguyễn Đình Nam
- Phạm Minh Đức
- Phạm Hồng Vinh

## 9.7 Mạng Tích chập Kết nối Dày đặc (DenseNet)

ResNet đã làm thay đổi đáng kể quan điểm về cách tham số hóa các hàm số trong mạng nơ-ron sâu. Ở một mức độ nào đó, DenseNet có thể được coi là phiên bản mở rộng hợp lý của ResNet. Để hiểu cách đi đến kết luận đó, ta cần tìm hiểu một chút lý thuyết. Nhắc lại công thức khai triển Taylor cho hàm một biến vô hướng như sau

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3). \quad (9.7.1)$$

### 9.7.1 Phân tách Hàm số

Điểm mấu chốt là khai triển Taylor phân tách hàm số thành các số hạng có bậc tăng dần. Tương tự, ResNet phân tách các hàm số thành

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x}). \quad (9.7.2)$$

Cụ thể, ResNet tách hàm số  $f$  thành một số hạng tuyến tính đơn giản và một số hạng phi tuyến phức tạp hơn. Nếu ta muốn tách ra thành nhiều hơn hai số hạng thì sao? Một giải pháp đã được đề xuất bởi (Huang et al., 2017) trong kiến trúc DenseNet. Kiến trúc này đạt được hiệu suất kỉ lục trên tập dữ liệu ImageNet.

<sup>175</sup> <https://discuss.mxnet.io/t/2359>

<sup>176</sup> <https://forum.machinelearningcoban.com/c/d21>

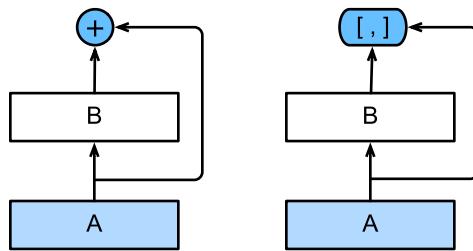


Fig. 9.7.1: Sự khác biệt chính giữa ResNet (bên trái) và DenseNet (bên phải) trong các kết nối xuyên tầng: sử dụng phép cộng và sử dụng phép nối.

Như được thể hiện trong Fig. 9.7.1, điểm khác biệt chính là DenseNet *nối* đầu ra lại với nhau thay vì *cộng* lại như ở ResNet. Kết quả là ta thực hiện một ánh xạ từ  $\mathbf{x}$  đến các giá trị của nó sau khi áp dụng một chuỗi các hàm với độ phức tạp tăng dần.

$$\mathbf{x} \rightarrow [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x}))), \dots]. \quad (9.7.3)$$

Cuối cùng, tất cả các hàm số này sẽ được kết hợp trong một Perceptron đa tầng để giảm số lượng đặc trưng một lần nữa. Lập trình thay đổi này khá đơn giản — thay vì cộng các số hạng với nhau, ta sẽ nối chúng lại. Cái tên DenseNet phát sinh từ việc đồ thị phụ thuộc giữa các biến trở nên khá dày đặc. Tầng cuối cùng của một chuỗi như vậy được kết nối “dày đặc” tới tất cả các tầng trước đó. Thành phần chính của DenseNet là các khối dày đặc và các tầng chuyển tiếp. Các khối dày đặc định nghĩa cách các đầu vào và đầu ra được nối với nhau, trong khi các tầng chuyển tiếp kiểm soát số lượng kênh sao cho nó không quá lớn. Các kết nối dày đặc được biểu diễn trong Fig. 9.7.2.

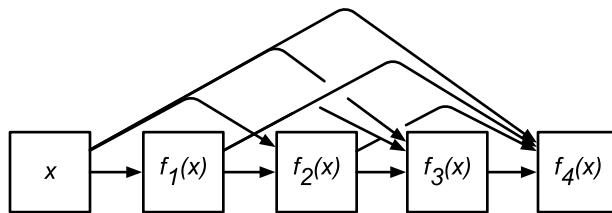


Fig. 9.7.2: Các kết nối dày đặc trong DenseNet

## 9.7.2 Khối Dày Đặc

DenseNet sử dụng kiến trúc “chuẩn hóa theo batch, hàm kích hoạt và phép tích chập” đã qua sửa đổi của ResNet (xem phần bài tập trong Section 9.6). Đầu tiên, ta sẽ lập trình kiến trúc này trong hàm `conv_block`.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

def conv_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(),
           nn.Activation('relu'),
           nn.Conv2D(num_channels, kernel_size=3, padding=1))
    return blk
```

Một khối dày đặc bao gồm nhiều khối conv\_block với cùng số lượng kênh đầu ra. Tuy nhiên, ta sẽ nối đầu vào và đầu ra của từng khối theo chiều kênh khi tính toán lượt truyền xuôi.

```
class DenseBlock(nn.Block):
    def __init__(self, num_convs, num_channels, **kwargs):
        super(DenseBlock, self).__init__(**kwargs)
        self.net = nn.Sequential()
        for _ in range(num_convs):
            self.net.add(conv_block(num_channels))

    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 = np.concatenate((X, Y), axis=1)
        return X
```

Trong ví dụ sau, ta sẽ định nghĩa một khối dày đặc gồm hai khối tích chập với 10 kênh đầu ra. Với một đầu vào gồm 3 kênh, ta sẽ nhận được một đầu ra với  $3 + 2 \times 10 = 23$  kênh. Số lượng kênh của khối tích chập kiểm soát sự gia tăng của số lượng kênh đầu ra so với số lượng kênh đầu vào. Số lượng kênh này còn được gọi là tốc độ tăng trưởng (*growth rate*).

```
blk = DenseBlock(2, 10)
blk.initialize()
X = np.random.uniform(size=(4, 3, 8, 8))
Y = blk(X)
Y.shape
```

### 9.7.3 Tầng Chuyển Tiếp

Mỗi khối dày đặc sẽ làm tăng thêm số lượng kênh. Nhưng việc thêm quá nhiều kênh sẽ tạo nên một mô hình phức tạp quá mức. Do đó, một tầng chuyển tiếp sẽ được sử dụng để kiểm soát độ phức tạp của mô hình. Tầng này dùng một tầng tích chập  $1 \times 1$  để giảm số lượng kênh, theo sau là một tầng gộp trung bình với sải bước bằng 2 để giảm một nửa chiều cao và chiều rộng, từ đó giảm độ phức tạp của mô hình hơn nữa.

```
def transition_block(num_channels):
    blk = nn.Sequential()
    blk.add(nn.BatchNorm(), nn.Activation('relu'),
           nn.Conv2D(num_channels, kernel_size=1),
           nn.AvgPool2D(pool_size=2, strides=2))
    return blk
```

Ta sẽ áp dụng một tầng chuyển tiếp với 10 kênh lên đầu ra của khối dày đặc trong ví dụ trước. Việc này sẽ làm giảm số lượng kênh đầu ra xuống còn 10, đồng thời làm giảm đi một nửa chiều cao và chiều rộng.

```
blk = transition_block(10)
blk.initialize()
blk(Y).shape
```

#### 9.7.4 Mô hình DenseNet

Tiếp theo, ta sẽ xây dựng một mô hình DenseNet. Đầu tiên, DenseNet sử dụng một tầng tích chập và một tầng gộp cực đại như trong ResNet.

```
net = nn.Sequential()
net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
        nn.BatchNorm(), nn.Activation('relu'),
        nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Sau đó, tương tự như cách ResNet sử dụng bốn khối phần dư, DenseNet cũng dùng bốn khối dày đặc. Và cũng giống như ResNet, ta có thể tùy chỉnh số lượng tầng tích chập được sử dụng trong mỗi khối dày đặc. Ở đây, ta sẽ đặt số lượng khối tích chập bằng 4 để giống với kiến trúc ResNet-18 trong phần trước. Ngoài ra, ta đặt số lượng kênh (tức tốc độ tăng trưởng) của các tầng tích chập trong khối dày đặc là 32, vì vậy 128 kênh sẽ được thêm vào trong mỗi khối dày đặc.

Trong ResNet, chiều cao và chiều rộng được giảm sau mỗi khối bằng cách sử dụng một khối phần dư với sải bước bằng 2. Ở đây, ta sẽ sử dụng tầng chuyển tiếp để làm giảm đi một nửa chiều cao, chiều rộng và số kênh.

```
# Num_channels: the current number of channels
num_channels, growth_rate = 64, 32
num_convs_in_dense_blocks = [4, 4, 4, 4]

for i, num_convs in enumerate(num_convs_in_dense_blocks):
    net.add(DenseBlock(num_convs, growth_rate))
    # This is the number of output channels in the previous dense block
    num_channels += num_convs * growth_rate
    # A transition layer that has the number of channels is added between
    # the dense blocks
    if i != len(num_convs_in_dense_blocks) - 1:
        num_channels //= 2
        net.add(transition_block(num_channels))
```

Tương tự như ResNet, một tầng gộp toàn cục và một tầng kết nối dày đủ sẽ được thêm vào cuối mạng để tính toán đầu ra.

```
net.add(nn.BatchNorm(),
        nn.Activation('relu'),
        nn.GlobalAvgPool2D(),
        nn.Dense(10))
```

#### 9.7.5 Thu thập dữ liệu và Huấn luyện

Trong phần này, vì đang sử dụng một mạng sâu hơn nên để đơn giản hóa việc tính toán, ta sẽ giảm chiều cao và chiều rộng của đầu vào từ 224 xuống còn 96.

```
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)
```

### 9.7.6 Tóm tắt

- Về mặt kết nối xuyên tầng, không giống như trong ResNet khi đầu vào và đầu ra được cộng lại với nhau, DenseNet nối các đầu vào và đầu ra theo chiều kenh.
- Các thành phần chính tạo nên DenseNet là các khối dày đặc và các tầng chuyển tiếp.
- Ta cần kiểm soát kích thước của các chiều khi thiết kế mạng bằng cách thêm các tầng chuyển tiếp để làm giảm số lượng kenh.

### 9.7.7 Bài tập

1. Tại sao ta lại sử dụng phép gộp trung bình thay vì gộp cực đại trong tầng chuyển tiếp?
2. Một trong những ưu điểm được đề cập trong bài báo DenseNet là kiến trúc này có số lượng tham số nhỏ hơn so với ResNet. Tại sao lại như vậy?
3. DenseNet thường bị chỉ trích vì nó tiêu tốn nhiều bộ nhớ.
  - Điều này có đúng không? Hãy thử thay đổi kích thước đầu vào thành  $224 \times 224$  để xem mức tiêu thụ bộ nhớ (GPU) thực tế.
  - Hãy tìm các phương án khác để giảm mức tiêu thụ bộ nhớ. Ta cần thay đổi kiến trúc này như thế nào?
4. Lập trình các phiên bản DenseNet khác nhau được trình bày trong Bảng 1 của (Huang et al., 2017).
5. Tại sao ta không cần nối các số hạng nếu ta chỉ quan tâm đến  $\mathbf{x}$  và  $f(\mathbf{x})$  như trong ResNet? Tại sao ta lại cần làm vậy với nhiều hơn hai tầng trong DenseNet?
6. Thiết kế một mạng kết nối đầy đủ tương tự như DenseNet và áp dụng nó vào bài toán Dự đoán Giá Nhà.

### 9.7.8 Thảo luận

- Tiếng Anh<sup>177</sup>
- Tiếng Việt<sup>178</sup>

### 9.7.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Nguyễn Văn Cường
- Nguyễn Cảnh Thướng
- Lê Khắc Hồng Phúc

<sup>177</sup> <https://discuss.mxnet.io/t/2360>

<sup>178</sup> <https://forum.machinelearningcoban.com/c/d21>

- Phạm Minh Đức
- Phạm Hồng Vinh

## 9.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Minh Đức



# 10 | Mạng Nơ-ron Hồi tiếp

Cho đến nay, chúng ta đã gặp hai loại dữ liệu: các vector tổng quát và hình ảnh. Với dữ liệu hình ảnh, ta đã thiết kế các tầng chuyên biệt nhằm tận dụng tính chính quy (*regularity property*) của hình ảnh. Nói cách khác, nếu ta hoán vị các điểm ảnh trong một ảnh, ta sẽ thu được một bức ảnh trông giống như các khuôn mẫu kiểm tra (*test pattern*) hay thấy trong truyền hình analog, và rất khó để suy luận về nội dung của chúng.

Quan trọng hơn là cho đến thời điểm này, chúng ta đã ngầm định rằng dữ liệu được sinh ra từ những phân phối độc lập và giống hệt nhau (*independently and identically distributed - i.i.d.*). Thật không may, điều này lại không đúng với hầu hết các loại dữ liệu. Ví dụ, các từ trong đoạn văn này được viết theo một trình tự nhất định mà nếu bị hoán vị đi một cách ngẫu nhiên thì sẽ rất khó để giải mã ý nghĩa của chúng. Tương tự với các khung hình trong video, tín hiệu âm thanh trong một cuộc hội thoại hoặc hành vi duyệt web, tất cả đều có cấu trúc tuần tự. Do đó, hoàn toàn hợp lý khi ta giả định rằng các mô hình chuyên biệt cho những kiểu dữ liệu này sẽ giúp việc mô tả dữ liệu và giải quyết các bài toán ước lượng được tốt hơn.

Một vấn đề nữa phát sinh khi chúng ta không chỉ nhận một chuỗi làm đầu vào mà còn muốn dự đoán những phần tử tiếp theo của chuỗi. Ví dụ, bài toán có thể là dự đoán phần tử tiếp theo trong dãy 2, 4, 6, 8, 10, ... Tác vụ này khá phổ biến trong phân tích chuỗi thời gian: để dự đoán thị trường chứng khoán, đường cong biểu hiện tình trạng sốt của bệnh nhân, hoặc gia tốc cần thiết cho một chiếc xe đua. Một lần nữa, chúng ta muốn xây dựng các mô hình có thể xử lý ổn thỏa kiểu dữ liệu trên.

Tóm lại, trong khi các mạng nơ-ron tích chập có thể xử lý hiệu quả thông tin trên chiều không gian, thì các mạng nơ-ron hồi tiếp được thiết kế để xử lý thông tin tuần tự tốt hơn. Các mạng này sử dụng các biến trạng thái để lưu trữ thông tin trong quá khứ, sau đó dựa vào chúng và các đầu vào hiện tại để xác định các đầu ra hiện tại.

Ở chương này, đa phần những ví dụ đề cập đến các mạng hồi tiếp đều dựa trên dữ liệu văn bản. Vì vậy, chúng ta sẽ cùng đào sâu tìm hiểu những mô hình ngôn ngữ. Sau khi tìm hiểu về dữ liệu chuỗi, ta sẽ thảo luận các khái niệm cơ bản của mô hình ngôn ngữ để làm bàn đạp cho việc thiết kế các mạng nơ-ron hồi tiếp. Cuối cùng, ta sẽ tiến hành mô tả phương pháp tính toán gradient trong các mạng nơ-ron hồi tiếp để từ đó hiểu rõ hơn các vấn đề có thể gặp phải trong quá trình huấn luyện.

## 10.1 Mô hình chuỗi

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 cẩn thận. Xét cho cùng, bạn muốn xem thêm nhiều bộ phim hay không? Nhưng hóa ra, mọi thứ không hề đơn giản như vậy. Đánh giá của mỗi người về một bộ phim có thể thay đổi đáng kể theo thời gian. Trên thực tế, các nhà tâm lý học thậm chí còn đặt tên cho một số hiệu ứng:

- **Hiệu ứng mò neo<sup>179</sup>**: dựa trên ý kiến của người khác. Ví dụ, xếp hạng của một bộ phim sẽ tăng lên sau khi nó thắng giải Oscar, mặc dù đoàn làm phim này không có bất kỳ tác động nào về mặt quảng bá đến bộ phim. Hiệu ứng này kéo dài trong vòng một vài tháng cho đến khi giải thưởng bị lãng quên. (Wu et al., 2017) chỉ ra rằng hiệu ứng này tăng chỉ số xếp hạng thêm hơn nửa điểm.
- **Hiệu ứng vòng xoáy khoái lạc<sup>180</sup>**: con người nhanh chóng thích nghi để chấp nhận một tình huống tốt hơn (hoặc xấu đi) như một điều bình thường mới. Chẳng hạn, sau khi xem nhiều bộ phim hay, sự kỳ vọng rằng bộ phim tiếp theo sẽ hay tương đương hoặc thậm chí phải hay hơn trở nên khá cao, do đó ngay cả một bộ phim trung bình cũng có thể bị coi là một bộ phim tồi.
- **Tính thời vụ**: rất ít khán giả thích xem một bộ phim về ông già Noel vào tháng 8.
- Trong một số trường hợp, các bộ phim trở nên không được ưa chuộng do những hành động sai trái của các đạo diễn hoặc diễn viên tham gia vào quá trình sản xuất phim.
- Một số phim trở thành “phim cult” vì chúng gần như tê đến mức phát cười. *Plan 9 from Outer Space* và *Troll 2* là hai ví dụ nổi tiếng.

Tóm lại, thứ bậc xếp hạng không hề cố định. Sử dụng các động lực dựa trên thời gian đã giúp (Koren, 2009) đề xuất phim chính xác hơn. Tuy nhiên, vấn đề không chỉ là về phim ảnh.

- Nhiều người dùng có thói quen rất đặc biệt liên quan tới thời gian mở ứng dụng. Chẳng hạn, học sinh sử dụng các ứng dụng mạng xã hội nhiều hơn hẳn sau giờ học. Các ứng dụng giao dịch chứng khoán được sử dụng nhiều khi thị trường mở cửa.
- Việc dự đoán giá cổ phiếu ngày mai khó hơn nhiều so với việc dự đoán giá cổ phiếu bị bỏ lỡ ngày hôm qua, mặc dù cả hai đều là bài toán ước tính một con số. Rốt cuộc, nhìn lại quá khứ dễ hơn nhiều so với dự đoán tương lai. Trong thống kê, bài toán đầu tiên được gọi là *ngoại suy* và bài toán sau được gọi là *nội suy*.
- Âm nhạc, giọng nói, văn bản, phim ảnh, bước đi, v.v ... đều có tính chất tuần tự. Nếu chúng ta hoán vị chúng, chúng sẽ không còn nhiều ý nghĩa. Dòng tiêu đề *chó cắn người* ít gây ngạc nhiên hơn nhiều so với *người cắn chó*, mặc dù các từ giống hệt nhau.
- Các trận động đất có mối tương quan mạnh mẽ, tức sau một trận động đất lớn, rất có thể sẽ có một số dư chấn nhỏ hơn và xác suất xảy ra dư chấn cao hơn nhiều so với trường hợp trận động đất lớn không xảy ra trước đó. Trên thực tế, các trận động đất có mối tương quan về mặt không-thời gian, tức các dư chấn thường xảy ra trong một khoảng thời gian ngắn và ở gần nhau.
- Con người tương tác với nhau một cách tuần tự, điều này có thể được thấy trong các cuộc tranh cãi trên Twitter, các điều nhảy và các cuộc tranh luận.

<sup>179</sup> <https://en.wikipedia.org/wiki/Anchoring>

<sup>180</sup> [https://en.wikipedia.org/wiki/Hedonic\\_treadmill](https://en.wikipedia.org/wiki/Hedonic_treadmill)

### 10.1.1 Các công cụ thống kê

Tóm lại, ta cần các công cụ thống kê và các kiến trúc mạng nơ-ron sâu mới để xử lý dữ liệu chuỗi. Để đơn giản hóa mọi việc, ta sẽ sử dụng giá cổ phiếu được minh họa trong Fig. 10.1.1 để làm ví dụ.

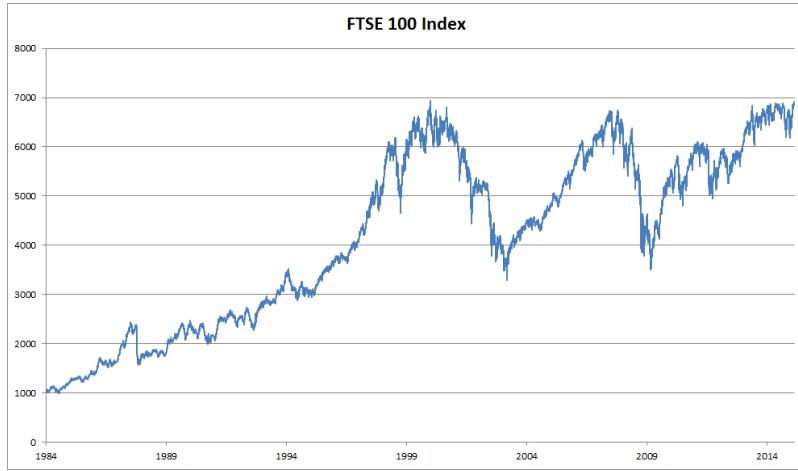


Fig. 10.1.1: Giá cổ phiếu FTSE 100 trong vòng 30 năm

Ta sẽ gọi giá cổ phiếu là  $x_t \geq 0$ , tức tại thời điểm  $t \in \mathbb{N}$  ta thấy giá cổ phiếu bằng  $x_t$ . Để có thể kiếm lời trên thị trường chứng khoán vào ngày  $t$ , một nhà giao dịch sẽ muốn dự đoán  $x_t$  thông qua

$$x_t \sim p(x_t | x_{t-1}, \dots, x_1). \quad (10.1.1)$$

#### Mô hình Tự hồi quy

Để dự đoán giá cổ phiếu, các nhà giao dịch có thể sử dụng một mô hình hồi quy, chẳng hạn như mô hình mà ta đã huấn luyện trong Section 5.3. Chỉ có một vấn đề lớn ở đây, đó là số lượng đầu vào,  $x_{t-1}, \dots, x_1$  thay đổi tùy thuộc vào  $t$ . Cụ thể, số lượng đầu vào sẽ tăng cùng với lượng dữ liệu thu được và ta sẽ cần một phép tính xấp xỉ để làm cho giải pháp này khả thi về mặt tính toán. Phần lớn nội dung tiếp theo trong chương này sẽ xoay quanh việc làm thế nào để ước lượng  $p(x_t | x_{t-1}, \dots, x_1)$  một cách hiệu quả. Nói ngắn gọn, ta có hai chiến lược:

- Giả sử rằng việc sử dụng một chuỗi có thể rất dài  $x_{t-1}, \dots, x_1$  là không thực sự cần thiết. Trong trường hợp này, ta có thể hài lòng với một khoảng thời gian  $\tau$  và chỉ sử dụng các quan sát  $x_{t-1}, \dots, x_{t-\tau}$ . Lợi ích trước mắt là bây giờ số lượng đối số luôn bằng nhau, ít nhất là với  $t > \tau$ . Điều này sẽ cho phép ta huấn luyện một mạng sâu như được đề cập ở bên trên. Các mô hình như vậy được gọi là các mô hình *tự hồi quy* (*autoregressive*), vì chúng tự thực hiện hồi quy trên chính mình.
- Một chiến lược khác, được minh họa trong Fig. 10.1.2, là giữ một giá trị  $h_t$  để tóm tắt các quan sát trong quá khứ, đồng thời cập nhật  $h_t$  bên cạnh việc dự đoán  $\hat{x}_t$ . Kết quả là mô hình sẽ ước tính  $x_t$  với  $\hat{x}_t = p(x_t | x_{t-1}, h_t)$  và cập nhật  $h_t = g(h_{t-1}, x_{t-1})$ . Do  $h_t$  không bao giờ được quan sát nên các mô hình này còn được gọi là các mô hình *tự hồi quy tiềm ẩn* (*latent autoregressive model*). LSTM và GRU là hai ví dụ cho kiểu mô hình này.

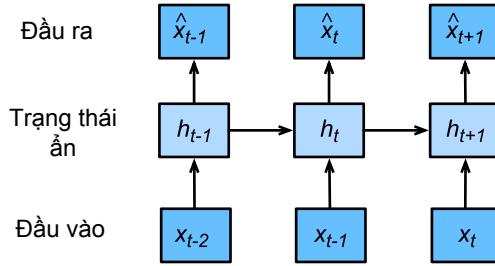


Fig. 10.1.2: Một mô hình tự hồi quy tiềm ẩn.

Cả hai trường hợp đều đặt ra câu hỏi về cách tạo ra dữ liệu huấn luyện. Người ta thường sử dụng các quan sát từ quá khứ cho đến hiện tại để dự đoán các quan sát xảy ra trong tương lai. Rõ ràng chúng ta không thể trông đợi thời gian sẽ đúng yên. Tuy nhiên, một giả định phổ biến là: tuy các giá trị cụ thể của  $x_t$  có thể thay đổi, ít ra động lực của chuỗi thời gian sẽ không đổi. Điều này khá hợp lý, vì nếu động lực thay đổi thì ta sẽ không thể dự đoán được nó bằng cách sử dụng dữ liệu mà ta đang có. Các nhà thống kê gọi các động lực không thay đổi này là *cố định (stationary)*. Dù có làm gì đi chăng nữa, chúng ta vẫn sẽ tìm được ước lượng của toàn bộ chuỗi thời gian thông qua

$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t | x_{t-1}, \dots, x_1). \quad (10.1.2)$$

Lưu ý rằng các xem xét trên vẫn đúng trong trường hợp chúng ta làm việc với các đối tượng rời rạc, chẳng hạn như từ ngữ thay vì số. Sự khác biệt duy nhất trong trường hợp này là chúng ta cần sử dụng một bộ phân loại thay vì một bộ hồi quy để ước lượng  $p(x_t | x_{t-1}, \dots, x_1)$ .

### Mô hình Markov

Nhắc lại phép xấp xỉ trong một mô hình tự hồi quy, chúng ta chỉ sử dụng  $(x_{t-1}, \dots, x_{t-\tau})$  thay vì  $(x_{t-1}, \dots, x_1)$  để ước lượng  $x_t$ . Bất cứ khi nào phép xấp xỉ này là chính xác, chúng ta nói rằng chuỗi thỏa mãn *điều kiện Markov*. Cụ thể, nếu  $\tau = 1$ , chúng ta có mô hình Markov *bậc một* và  $p(x)$  như sau

$$p(x_1, \dots, x_T) = \prod_{t=1}^T p(x_t | x_{t-1}). \quad (10.1.3)$$

Các mô hình như trên rất hữu dụng bất cứ khi nào  $x_t$  chỉ là các giá trị rời rạc, vì trong trường hợp này, quy hoạch động có thể được sử dụng để tính toán chính xác các giá trị theo chuỗi. Ví dụ, chúng ta có thể tính toán  $p(x_{t+1} | x_{t-1})$  một cách hiệu quả bằng cách chỉ sử dụng các quan sát trong một khoảng thời gian ngắn tại quá khứ:

$$p(x_{t+1} | x_{t-1}) = \sum_{x_t} p(x_{t+1} | x_t)p(x_t | x_{t-1}). \quad (10.1.4)$$

Chi tiết về quy hoạch động nằm ngoài phạm vi của phần này, nhưng chúng tôi sẽ giới thiệu nó trong [Section 11.4](#). Các công cụ trên được sử dụng rất phổ biến trong các thuật toán điều khiển và học tăng cường.

## Quan hệ Nhân quả

Về nguyên tắc, không có gì sai khi trải (*unfolding*)  $p(x_1, \dots, x_T)$  theo thứ tự ngược lại. Bằng cách đặt điều kiện như vậy, chúng ta luôn có thể viết chúng như sau

$$p(x_1, \dots, x_T) = \prod_{t=T}^1 p(x_t | x_{t+1}, \dots, x_T). \quad (10.1.5)$$

Trên thực tế, nếu có một mô hình Markov, chúng ta cũng có thể thu được một phân phối xác suất có điều kiện ngược. Tuy nhiên trong nhiều trường hợp vẫn tồn tại một trạng thái tự nhiên cho dữ liệu, cụ thể đó là chiều thuận theo thời gian. Rõ ràng là các sự kiện trong tương lai không thể ảnh hưởng đến quá khứ. Do đó, nếu thay đổi  $x_t$  thì ta có thể ảnh hưởng đến những gì xảy ra tại  $x_{t+1}$  trong tương lai, nhưng lại không thể ảnh hưởng tới quá khứ theo chiều ngược lại. Nếu chúng ta thay đổi  $x_t$ , phân phối trên các sự kiện trong quá khứ sẽ không thay đổi. Do đó, việc giải thích  $p(x_{t+1} | x_t)$  sẽ đơn giản hơn là  $p(x_t | x_{t+1})$ . Ví dụ: (Hoyer et al., 2009) chỉ ra rằng trong một số trường hợp chúng ta có thể tìm  $x_{t+1} = f(x_t) + \epsilon$  khi có thêm nhiều, trong khi điều ngược lại thì không đúng. Đây là một tin tuyệt vời vì chúng ta thường quan tâm tới việc ước lượng theo chiều thuận hơn. Để tìm hiểu thêm về chủ đề này, có thể tìm đọc cuốn sách (Peters et al., 2017a). Chúng ta sẽ chỉ tìm hiểu sơ qua trong phần này.

### 10.1.2 Một ví dụ đơn giản

Sau khi đề cập nhiều về lý thuyết, bây giờ chúng ta hãy thử lập trình minh họa. Đầu tiên, hãy khởi tạo một vài dữ liệu như sau. Để đơn giản, chúng ta tạo chuỗi thời gian bằng cách sử dụng hàm sin cộng thêm một chút nhiễu.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx, gluon, init
from mxnet.gluon import nn
npx.set_np()

T = 1000 # Generate a total of 1000 points
time = np.arange(0, T)
x = np.sin(0.01 * time) + 0.2 * np.random.normal(size=T)
d2l.plot(time, [x])
```

Tiếp theo, chúng ta cần biến chuỗi thời gian này thành các đặc trưng và nhãn có thể được sử dụng để huấn luyện mạng. Dựa trên kích thước embedding  $\tau$ , chúng ta ánh xạ dữ liệu thành các cặp  $y_t = x_t$  và  $\mathbf{z}_t = (x_{t-1}, \dots, x_{t-\tau})$ . Để ý kĩ, có thể thấy rằng ta sẽ mất  $\tau$  điểm dữ liệu đầu tiên, vì chúng ta không có đủ  $\tau$  điểm dữ liệu trong quá khứ để làm đặc trưng cho chúng. Một cách đơn giản để khắc phục điều này, đặc biệt là khi chuỗi thời gian rất dài, là loại bỏ đi số ít các phần tử đó. Một cách khác là đệm giá trị 0 vào chuỗi thời gian. Mã nguồn dưới đây về cơ bản là giống hệt với mã nguồn huấn luyện trong các phần trước. Chúng tôi cố gắng giữ cho kiến trúc đơn giản với vài tầng kết nối đầy đủ, hàm kích hoạt ReLU và hàm mất mát  $\ell_2$ . Do việc mô hình hóa phần lớn là giống với khi ta xây dựng các bộ ước lượng hồi quy viết bằng Gluon trong các phần trước, nên chúng ta sẽ không đi sâu vào chi tiết trong phần này.

```
tau = 4
features = np.zeros((T-tau, tau))
```

(continues on next page)

```

for i in range(tau):
    features[:, i] = x[i: T-tau+i]
labels = x[tau:]

batch_size, n_train = 16, 600
train_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                            batch_size, is_train=True)
test_iter = d2l.load_array((features[:n_train], labels[:n_train]),
                           batch_size, is_train=False)

# Vanilla MLP architecture
def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(10, activation='relu'),
           nn.Dense(1))
    net.initialize(init.Xavier())
    return net

# Least mean squares loss
loss = gluon.loss.L2Loss()

```

Bây giờ chúng ta đã sẵn sàng để huấn luyện.

```

def train_net(net, train_iter, loss, epochs, lr):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                           {'learning_rate': lr})
    for epoch in range(1, epochs + 1):
        for X, y in train_iter:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        print('epoch %d, loss: %f' %
              (epoch, d2l.evaluate_loss(net, train_iter, loss)))

net = get_net()
train_net(net, train_iter, loss, 10, 0.01)

```

### 10.1.3 Dự đoán của Mô hình

Vì cả hai giá trị mất mát trên tập huấn luyện và kiểm tra đều nhỏ, chúng ta kỳ vọng mô hình trên sẽ hoạt động tốt. Hãy cùng xác nhận điều này trên thực tế. Điều đầu tiên cần kiểm tra là mô hình có thể dự đoán những gì sẽ xảy ra trong bước thời gian kế tiếp tốt như thế nào.

```

estimates = net(features)
d2l.plot([time, time[tau:]], [x, estimates],
         legend=['data', 'estimate'])

```

Kết quả khá tốt, đúng như những gì chúng ta mong đợi. Thậm chí sau hơn 600 mẫu quan sát, phép ước lượng vẫn trông khá tin cậy. Chỉ có một chút vấn đề: nếu chúng ta quan sát dữ liệu tới bước thời gian thứ 600, chúng ta không thể hy vọng sẽ nhận được nhãn gốc cho tất cả các dự đoán tương

lai. Thay vào đó, chúng ta cần tiến lên từng bước một:

$$\begin{aligned}x_{601} &= f(x_{600}, \dots, x_{597}), \\x_{602} &= f(x_{601}, \dots, x_{598}), \\x_{603} &= f(x_{602}, \dots, x_{599}).\end{aligned}\tag{10.1.6}$$

Nói cách khác, chúng ta sẽ phải sử dụng những dự đoán của mình để đưa ra dự đoán trong tương lai. Hãy cùng xem cách này có ổn không.

```
predictions = np.zeros(T)
predictions[:n_train] = x[:n_train]
for i in range(n_train, T):
    predictions[i] = net(
        predictions[(i-tau):i].reshape(1, -1)).reshape(1)
d2l.plot([time, time[tau:], time[n_train:]],
         [x, estimates, predictions[n_train:]],
         legend=['data', 'estimate', 'multistep'], figsize=(4.5, 2.5))
```

Ví dụ trên cho thấy, cách này đã thất bại thảm hại. Các giá trị ước lượng rất nhanh chóng suy giảm thành một hằng số chỉ sau một vài bước. Tại sao thuật toán trên hoạt động tệ đến thế? Suy cho cùng, lý do là trên thực tế các sai số dự đoán bị chồng chất qua các bước thời gian. Cụ thể, sau bước thời gian 1 chúng ta có nhận được sai số  $\epsilon_1 = \bar{\epsilon}$ . Tiếp theo, *đầu vào* cho bước thời gian 2 bị nhiễu loạn bởi  $\epsilon_1$ , do đó chúng ta nhận được sai số dự đoán  $\epsilon_2 = \bar{\epsilon} + L\epsilon_1$ . Tương tự như thế cho các bước thời gian tiếp theo. Sai số có thể phân kỳ khá nhanh khỏi các quan sát đúng. Đây là một hiện tượng phổ biến. Ví dụ, dự báo thời tiết trong 24 giờ tới có độ chính xác khá cao nhưng nó giảm đi nhanh chóng với những dự báo xa hơn quãng thời gian đó. Chúng ta sẽ thảo luận về các phương pháp để cải thiện vấn đề trên trong chương này và những chương tiếp theo.

Chúng ta hãy kiểm chứng quan sát trên bằng cách tính toán dự đoán  $k$  bước thời gian trên toàn bộ chuỗi.

```
k = 33 # Look up to k - tau steps ahead

features = np.zeros((k, T-k))
for i in range(tau): # Copy the first tau features from x
    features[i] = x[i:T-k+i]

for i in range(tau, k): # Predict the (i-tau)-th step
    features[i] = net(features[(i-tau):i].T).T

steps = (4, 8, 16, 32)
d2l.plot([time[i:T-k+i] for i in steps], [features[i] for i in steps],
         legend=['step %d' % i for i in steps], figsize=(4.5, 2.5))
```

Điều này minh họa rõ ràng chất lượng của các ước lượng thay đổi như thế nào khi chúng ta cố gắng dự đoán xa hơn trong tương lai. Mặc dù những dự đoán có độ dài là 8 bước vẫn còn khá tốt, bất cứ kết quả dự đoán nào vượt ra ngoài khoảng đó thì khá là vô dụng.

#### 10.1.4 Tóm tắt

- Các mô hình chuỗi thường yêu cầu các công cụ thống kê chuyên biệt để ước lượng. Hai lựa chọn phổ biến đó là các mô hình tự hồi quy và mô hình tự hồi quy biến tiềm ẩn.
- Sai số bị tích lũy và chất lượng của phép ước lượng suy giảm đáng kể khi mô hình dự đoán các bước thời gian xa hơn.
- Khó khăn trong phép nội suy và ngoại suy khá khác biệt. Do đó, nếu bạn có một kiểu dữ liệu chuỗi thời gian, hãy luôn để ý trình tự thời gian của dữ liệu khi huấn luyện, hay nói cách khác, không bao giờ huấn luyện trên dữ liệu thuộc về bước thời gian trong tương lai.
- Đối với các mô hình nhân quả (ví dụ, ở đó thời gian đi về phía trước), ước lượng theo chiều xuôi thường dễ dàng hơn rất nhiều so với chiều ngược lại.

#### 10.1.5 Bài tập

1. Hãy cải thiện mô hình nói trên bằng cách
  - Kết hợp nhiều hơn 4 mẫu quan sát trong quá khứ? Bao nhiêu mẫu quan sát là thực sự cần thiết?
  - Bạn sẽ cần bao nhiêu mẫu nếu dữ liệu không có nhiều? Gợi ý: bạn có thể viết sin và cos dưới dạng phương trình vi phân.
  - Có thể kết hợp các đặc trưng cũ hơn trong khi đảm bảo tổng số đặc trưng là không đổi không? Điều này có cải thiện độ chính xác không? Tại sao?
  - Thay đổi cấu trúc mạng nơ-ron và quan sát tác động của nó.
2. Nếu một nhà đầu tư muốn tìm một mã chứng khoán tốt để mua. Cô ta sẽ nhìn vào lợi nhuận trong quá khứ để quyết định mã nào có khả năng sinh lời. Điều gì có thể khiến chiến lược này trở thành sai lầm?
3. Liệu có thể áp dụng quan hệ nhân quả cho dữ liệu văn bản được không? Nếu có thì ở mức độ nào?
4. Hãy cho một ví dụ khi mô hình tự hồi quy tiềm ẩn có thể cần được dùng để nắm bắt động lực của dữ liệu.

#### 10.1.6 Thảo luận

- [Tiếng Anh<sup>181</sup>](#)
- [Tiếng Việt<sup>182</sup>](#)

<sup>181</sup> <https://discuss.mxnet.io/t/2860>

<sup>182</sup> <https://forum.machinelearningcoban.com/c/d21>

### 10.1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Nguyễn Cảnh Thượng
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

## 10.2 Tiền Xử lý Dữ liệu Văn bản

Dữ liệu văn bản là một ví dụ điển hình của dữ liệu chuỗi. Một bài báo có thể coi là một chuỗi các từ, hoặc một chuỗi các ký tự. Dữ liệu văn bản là một dạng dữ liệu quan trọng bên cạnh dữ liệu hình ảnh được sử dụng trong cuốn sách này, phần này sẽ được dành để giải thích các bước tiền xử lý thường gặp cho loại dữ liệu này. Quá trình tiền xử lý thường bao gồm bốn bước sau:

1. Nạp dữ liệu văn bản ở dạng chuỗi ký tự vào bộ nhớ.
2. Chia chuỗi thành các token trong đó một token có thể là một từ hoặc một ký tự.
3. Xây dựng một bộ từ vựng cho các token để ánh xạ chúng thành các chỉ số (*index*).
4. Ánh xạ tất cả các token trong dữ liệu văn bản thành các chỉ số để dễ dàng đưa vào các mô hình.

### 10.2.1 Đọc Bộ dữ liệu

Để bắt đầu chúng ta nạp dữ liệu văn bản từ cuốn sách *Cỗ máy Thời gian* (Time Machine)<sup>183</sup> của tác giả H. G. Wells. Đây là một kho ngữ liệu khá nhỏ chỉ hơn 30.000 từ, nhưng nó đủ tốt cho mục đích minh họa. Nhiều bộ dữ liệu trên thực tế chứa hàng tỷ từ. Hầm sau đây đọc dữ liệu thành một danh sách các câu, mỗi câu là một chuỗi. Chúng ta bỏ qua dấu câu và chữ viết hoa.

```
import collections
from d2l import mxnet as d2l
import re

# Saved in the d2l package for later use
d2l.DATA_HUB['time_machine'] = (d2l.DATA_URL + 'timemachine.txt',
                                 '090b5e7e70c295757f55df93cb0a180b9691891a')

# Saved in the d2l package for later use
```

(continues on next page)

<sup>183</sup> <http://www.gutenberg.org/ebooks/35>

```

def read_time_machine():
    """Load the time machine book into a list of sentences."""
    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()
# sentences %d' % len(lines)

```

### 10.2.2 Token hoá

Với mỗi câu, chúng ta chia nó thành một danh sách các token. Một token là một điểm dữ liệu mà mô hình sẽ huấn luyện và đưa ra dự đoán từ nó. Hàm dưới đây làm nhiệm vụ tách một câu thành các từ hoặc các ký tự, và trả về một danh sách các chuỗi đã được phân tách.

```

# Saved in the d2l package for later use
def tokenize(lines, token='word'):
    """Split sentences into word or char 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)
tokens[0:2]

```

### 10.2.3 Bộ Từ vựng

Token kiểu chuỗi không phải là kiểu dữ liệu tiện lợi được sử dụng bởi các mô hình, thay vào đó chúng thường nhận dữ liệu đầu vào dưới dạng số. Nay giờ, chúng ta sẽ xây dựng một bộ từ điển, thường được gọi là *bộ từ vựng* (*vocabulary*), để ánh xạ chuỗi token thành các chỉ số bắt đầu từ 0. Để làm điều này, đầu tiên chúng ta lấy các token xuất hiện (*không lặp lại*) trong toàn bộ tài liệu, thường được gọi là kho ngữ liệu (*corpus*), và sau đó gán một giá trị số (*chỉ số*) cho mỗi token dựa trên tần suất xuất hiện của chúng. Các token có tần suất xuất hiện rất ít thường được loại bỏ để giảm độ phức tạp. Một token không xuất hiện trong kho ngữ liệu hay đã bị loại bỏ thường được ánh xạ vào một token vô danh đặc biệt (“*<unk>*”). Chúng ta có thể tùy chọn thêm vào các token dự trữ, ví dụ token “*<pad>*” được sử dụng để đệm từ, token “*<bos>*” để biểu thị vị trí bắt đầu của câu, và token “*<eos>*” để biểu thị vị trí kết thúc của câu.

```

# Saved in the d2l package for later use
class Vocab:
    def __init__(self, tokens, min_freq=0, reserved_tokens=None):
        if reserved_tokens is None:
            reserved_tokens = []
        # Sort according to frequencies
        counter = count_corpus(tokens)

```

(continues on next page)

```

self.token_freqs = sorted(counter.items(), key=lambda x: x[0])
self.token_freqs.sort(key=lambda x: x[1], reverse=True)
self.unk, uniq_tokens = 0, ['<unk>'] + reserved_tokens
uniq_tokens += [token for token, freq in self.token_freqs
                 if freq >= min_freq and token not in uniq_tokens]
self.idx_to_token, self.token_to_idx = [], dict()
for token in uniq_tokens:
    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]

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]

# Saved in the d2l package for later use
def count_corpus(sentences):
    # Flatten a list of token lists into a list of tokens
    tokens = [tk for line in sentences for tk in line]
    return collections.Counter(tokens)

```

Chúng ta xây dựng một bộ từ vựng với tập dữ liệu cỗ máy thời gian nói trên thành một kho ngữ liệu, và in ra phép ánh xạ giữa một vài token với các chỉ số của chúng.

```

vocab = Vocab(tokens)
print(list(vocab.token_to_idx.items())[0:10])

```

Sau đó, chúng ta có thể chuyển đổi từng câu vào một danh sách các chỉ số. Để minh họa một cách chi tiết, chúng ta in hai câu với các chỉ số tương ứng của chúng.

```

for i in range(8, 10):
    print('words:', tokens[i])
    print('indices:', vocab[tokens[i]])

```

#### 10.2.4 Kết hợp Tất cả lại

Chúng ta đóng gói tất cả các hàm trên thành hàm load\_corpus\_time\_machine, trả về corpus, một danh sách các chỉ số của token, và bộ từ vựng vocab của kho ngữ liệu cỗ máy thời gian. Chúng ta đã sửa đổi một vài thứ ở đây là: corpus là một danh sách đơn nhất, không phải một danh sách các danh sách token, vì chúng ta không lưu các thông tin chuỗi trong các mô hình bên dưới. Bên cạnh đó, chúng ta sẽ sử dụng các token ký tự để đơn giản hóa việc huấn luyện mô hình trong các phần sau.

```

# Saved in the d2l package for later use
def load_corpus_time_machine(max_tokens=-1):
    lines = read_time_machine()
    tokens = tokenize(lines, 'char')
    vocab = Vocab(tokens)
    corpus = [vocab[tk] for line in tokens for tk in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]
    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
len(corpus), len(vocab)

```

### 10.2.5 Tóm tắt

Chúng ta đã tiền xử lý các tài liệu văn bản bằng cách token hóa chúng thành các từ hoặc ký tự, và sau đó ánh xạ chúng thành các chỉ số tương ứng.

### 10.2.6 Bài tập

Token hóa là một bước tiền xử lý quan trọng. Mỗi ngôn ngữ có đều có các cách làm khác nhau. Hãy thử tìm thêm 3 phương pháp thường dùng để token hóa các câu.

### 10.2.7 Thảo luận

- Tiếng Anh<sup>184</sup>
- Tiếng Việt<sup>185</sup>

### 10.2.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc

<sup>184</sup> <https://discuss.mxnet.io/t/2363>

<sup>185</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 10.3 Mô hình Ngôn ngữ và Tập dữ liệu

Section 10.2 đã trình bày cách ánh xạ dữ liệu văn bản sang token, những token này có thể được xem như một chuỗi thời gian của các quan sát rời rạc. Giả sử văn bản độ dài  $T$  có dãy token là  $x_1, x_2, \dots, x_T$ , thì  $x_t (1 \leq t \leq T)$  có thể coi là đầu ra (hoặc nhãn) tại bước thời gian  $t$ . Khi đã có chuỗi thời gian trên, mục tiêu của mô hình ngôn ngữ là ước tính xác suất của

$$p(x_1, x_2, \dots, x_T). \quad (10.3.1)$$

Mô hình ngôn ngữ vô cùng hữu dụng. Chẳng hạn, một mô hình lý tưởng có thể tự tạo ra văn bản tự nhiên, chỉ bằng cách chọn một từ  $w_t$  tại thời điểm  $t$  với  $w_t \sim p(w_t | w_{t-1}, \dots, w_1)$ . Khác hoàn toàn với việc chỉ gõ phím ngẫu nhiên như trong định lý con khỉ vô hạn (*infinite monkey theorem*), văn bản được sinh ra từ mô hình này giống ngôn ngữ tự nhiên, giống tiếng Anh chẳng hạn. Hơn nữa, mô hình đủ khả năng tạo ra một đoạn hội thoại có ý nghĩa mà chỉ cần dựa vào đoạn hội thoại trước đó. Trên thực tế, còn rất xa để thiết kế được hệ thống như vậy, vì mô hình sẽ cần hiểu văn bản hơn là chỉ tạo ra nội dung đúng ngữ pháp.

Tuy nhiên, mô hình ngôn ngữ vẫn rất hữu dụng ngay cả khi còn hạn chế. Chẳng hạn, cụm từ “nhận dạng giọng nói” và “nhân gian rộng lỗi” có phát âm khá giống nhau. Điều này có thể gây ra sự mơ hồ trong việc nhận dạng giọng nói, nhưng có thể dễ dàng được giải quyết với một mô hình ngôn ngữ. Mô hình sẽ loại bỏ ngay phương án thứ hai do mang ý nghĩa kì lạ. Tương tự, một thuật toán tóm tắt tài liệu nên phân biệt được rằng câu “chó cắn người” xuất hiện thường xuyên hơn nhiều so với “người cắn chó”, hay như “Cháu muốn ăn bà ngoại” nghe khá kinh dị trong khi “Cháu muốn ăn, bà ngoại” lại là bình thường.

### 10.3.1 Ước tính một Mô hình Ngôn ngữ

Làm thế nào để mô hình hóa một tài liệu hay thậm chí là một chuỗi các từ? Ta có thể sử dụng cách phân tích đã dùng trong mô hình chuỗi ở phần trước. Bắt đầu bằng việc áp dụng quy tắc xác suất cơ bản sau:

$$p(w_1, w_2, \dots, w_T) = p(w_1) \prod_{t=2}^T p(w_t | w_1, \dots, w_{t-1}). \quad (10.3.2)$$

Ví dụ, xác suất của chuỗi văn bản chứa bốn token bao gồm các từ và dấu chấm câu được tính như sau:

$$p(\text{Statistics, is, fun, .}) = p(\text{Statistics})p(\text{is} | \text{Statistics})p(\text{fun} | \text{Statistics, is})p(\cdot | \text{Statistics, is, fun}). \quad (10.3.3)$$

Để tính toán mô hình ngôn ngữ, ta cần tính xác suất các từ và xác suất có điều kiện của một từ khi đã có vài từ trước đó. Đây chính là các tham số của mô hình ngôn ngữ. Ở đây chúng ta giả định rằng, tập dữ liệu huấn luyện là một kho ngữ liệu lớn, chẳng hạn như là tất cả các mục trong Wikipedia của [Dự án Gutenberg](#)<sup>186</sup>, hoặc tất cả văn bản được đăng trên mạng. Xác suất riêng lẻ của từng từ có thể tính bằng tần suất của từ đó trong tập dữ liệu huấn luyện.

Ví dụ,  $p(\text{Statistics})$  có thể được tính là xác suất của bất kỳ câu nào bắt đầu bằng “statistics”. Một cách thiếu chính xác hơn là đếm tất cả số lần xuất hiện của “statistics” và chia số lần đó cho tổng số

<sup>186</sup> [https://en.wikipedia.org/wiki/Project\\_Gutenberg](https://en.wikipedia.org/wiki/Project_Gutenberg)

từ trong kho ngữ liệu văn bản. Cách làm này khá hiệu quả, đặc biệt là với các từ xuất hiện thường xuyên. Tiếp theo, ta tính

$$\hat{p}(\text{is} \mid \text{Statistics}) = \frac{n(\text{Statistics, is})}{n(\text{Statistics})}. \quad (10.3.4)$$

Ở đây  $n(w)$  và  $n(w, w')$  lần lượt là số lần xuất hiện của các từ đơn và cặp từ ghép. Đáng tiếc là việc ước tính xác suất của một cặp từ thường khó khăn hơn, bởi vì sự xuất hiện của cặp từ “Statistics is” hiếm khi xảy ra hơn. Đặc biệt, với các cụm từ ít đi cùng nhau, rất khó tìm đủ số lần xuất hiện để ước tính chính xác. Mọi thứ thậm chí sẽ khó hơn đối với các cụm ba từ trở lên. Sẽ có nhiều cụm ba từ hợp lý mà hầu như không hề xuất hiện trong tập dữ liệu. Trừ khi có giải pháp để đánh trọng số khác không cho các tổ hợp từ đó, nếu không sẽ không thể sử dụng chúng trong một mô hình ngôn ngữ. Nếu kích thước tập dữ liệu nhỏ hoặc nếu các từ rất hiếm, chúng ta thậm chí có thể không tìm thấy nổi một lần xuất hiện của các tổ hợp từ đó.

Một kỹ thuật phổ biến là làm mượt Laplace (*Laplace smoothing*). Chúng ta đã biết kỹ thuật này khi thảo luận về Naive Bayes trong Section 20.9, với giải pháp là cộng thêm một hằng số nhỏ vào tất cả các số đếm như sau

$$\begin{aligned}\hat{p}(w) &= \frac{n(w) + \epsilon_1/m}{n + \epsilon_1}, \\ \hat{p}(w' \mid w) &= \frac{n(w, w') + \epsilon_2 \hat{p}(w')}{n(w) + \epsilon_2}, \\ \hat{p}(w'' \mid w', w) &= \frac{n(w, w', w'') + \epsilon_3 \hat{p}(w', w'')}{n(w, w') + \epsilon_3}.\end{aligned} \quad (10.3.5)$$

Ở đây các hệ số  $\epsilon_i > 0$  xác định mức độ ảnh hưởng của chuỗi ngắn hơn khi ước tính chuỗi dài hơn,  $m$  là tổng số từ trong tập văn bản. Công thức trên là một biến thể khá nguyên thủy của kỹ thuật làm mượt Kneser-Ney và Bayesian phi tham số. Xem (Wood et al., 2011) để biết thêm chi tiết. Thật không may, các mô hình như vậy là bất khả thi vì những lý do sau. Đầu tiên, chúng ta cần lưu trữ tất cả các số đếm. Thứ hai, các mô hình hoàn toàn bỏ qua ý nghĩa của các từ. Chẳng hạn, danh từ “mèo” (“cat”) và tính từ “thuộc về mèo” (“feline”) nên xuất hiện trong các ngữ cảnh có liên quan đến nhau. Rất khó để thêm các ngữ cảnh hỗ trợ vào các mô hình đó, trong khi các mô hình ngôn ngữ dựa trên học sâu hoàn toàn có thể làm được. Cuối cùng, các chuỗi từ dài gần như hoàn toàn mới lạ, do đó một mô hình chỉ đơn giản đếm tần số của các chuỗi từ đã thấy trước đó sẽ hoạt động rất kém.

### 10.3.2 Mô hình Markov và $n$ -grams

Trước khi thảo luận các giải pháp sử dụng học sâu, chúng ta sẽ giải thích một số thuật ngữ và khái niệm. Hãy nhớ lại mô hình Markov đề cập ở phần trước, và áp dụng để mô hình hóa ngôn ngữ. Một phân phối trên các chuỗi thỏa mãn điều kiện Markov bậc nhất nếu  $p(w_{t+1} \mid w_t, \dots, w_1) = p(w_{t+1} \mid w_t)$ . Những bậc cao hơn tương ứng với những chuỗi phụ thuộc dài hơn. Do đó chúng ta có thể áp dụng các phép xấp xỉ để mô hình hóa một chuỗi:

$$\begin{aligned}p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2)p(w_3)p(w_4), \\ p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_2)p(w_4 \mid w_3), \\ p(w_1, w_2, w_3, w_4) &= p(w_1)p(w_2 \mid w_1)p(w_3 \mid w_1, w_2)p(w_4 \mid w_2, w_3).\end{aligned} \quad (10.3.6)$$

Các công thức xác suất liên quan đến một, hai và ba biến được gọi là các mô hình unigram, bigram và trigram. Sau đây, chúng ta sẽ tìm hiểu cách thiết kế các mô hình tốt hơn.

### 10.3.3 Thống kê Ngôn ngữ Tự nhiên

Hãy cùng xem mô hình hoạt động thế nào trên dữ liệu thực tế. Chúng ta sẽ xây dựng bộ từ vựng dựa trên tập dữ liệu “cỗ máy thời gian” tương tự như ở Section 10.2 và in ra 10 từ có tần suất xuất hiện cao nhất.

```
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()

tokens = d2l.tokenize(d2l.read_time_machine())
vocab = d2l.Vocab(tokens)
print(vocab.token_freqs[:10])
```

Có thể thấy những từ xuất hiện nhiều nhất không có gì đáng chú ý. Các từ này được gọi là [từ dừng \(stop words\)](#)<sup>187</sup> và vì thế chúng thường được lọc ra. Dù vậy, những từ này vẫn có nghĩa và ta vẫn sẽ sử dụng chúng. Tuy nhiên, rõ ràng là tần số của từ suy giảm khá nhanh. Từ phổ biến thứ 10 xuất hiện ít hơn, chỉ bằng 1/5 lần so với từ phổ biến nhất. Để hiểu rõ hơn, chúng ta sẽ vẽ đồ thị tần số của 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 ta đang tiến gần tới một đặc điểm cơ bản: tần số của từ suy giảm nhanh chóng theo một cách được xác định rõ. Ngoại trừ bốn từ đầu tiên ('the', 'i', 'and', 'of'), tất cả các từ còn lại đi theo một đường thẳng trên biểu đồ thang log. Theo đó các từ tuân theo [định luật Zipf](#)<sup>188</sup>, tức là tần suất xuất hiện của từ được xác định bởi

$$n(x) \propto (x + c)^{-\alpha} \text{ và do đó } \log n(x) = -\alpha \log(x + c) + \text{const.} \quad (10.3.7)$$

Điều này khiến chúng ta cần suy nghĩ kỹ khi mô hình hóa các từ bằng cách đếm và kỹ thuật làm mượt. Rốt cuộc, chúng ta sẽ ước tính quá cao những từ có tần suất xuất hiện thấp. Vậy còn các tổ hợp từ khác như 2-gram, 3-gram và nhiều hơn thì sao? Hãy xem liệu tần số của bigram có tương tự như unigram hay không.

```
bigram_tokens = [[pair for pair in zip(
    line[:-1], line[1:])] for line in tokens]
bigram_vocab = d2l.Vocab(bigram_tokens)
print(bigram_vocab.token_freqs[:10])
```

Có một điều đáng chú ý ở đây. 9 trong số 10 cặp từ thường xuyên xuất hiện là các từ dừng và chỉ có một là liên quan đến cuốn sách — cặp từ “the time”. Hãy xem tần số của trigram có tương tự hay không.

```
trigram_tokens = [[triple for triple in zip(line[:-2], line[-1:-1], line[2:])]
                  for line in tokens]
trigram_vocab = d2l.Vocab(trigram_tokens)
print(trigram_vocab.token_freqs[:10])
```

<sup>187</sup> [https://en.wikipedia.org/wiki/Stop\\_words](https://en.wikipedia.org/wiki/Stop_words)

<sup>188</sup> [https://en.wikipedia.org/wiki/Zipf%27s\\_law](https://en.wikipedia.org/wiki/Zipf%27s_law)

Cuối cùng, hãy quan sát biểu đồ tần số token của các mô hình: unigram, 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',
         ylabel='frequency', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram'])
```

Có vài điều khá thú vị ở biểu đồ này. Thứ nhất, ngoài unigram, các cụm từ cũng tuân theo định luật Zipf, với số mũ thấp hơn tùy vào chiều dài cụm từ. Thứ hai, số lượng các n-gram độc nhất là không nhiều. Điều này có thể liên quan đến số lượng lớn các cấu trúc trong ngôn ngữ. Thứ ba, rất nhiều n-gram hiếm khi xuất hiện, khiến phép làm mượt Laplace không thích hợp để xây dựng mô hình ngôn ngữ. Thay vào đó, chúng ta sẽ sử dụng các mô hình học sâu.

#### 10.3.4 Chuẩn bị Dữ liệu Huấn luyện

Giả sử cần sử dụng mạng nơ-ron để huấn luyện mô hình ngôn ngữ. Với tính chất tuần tự của dữ liệu chuỗi, làm thế nào để đọc ngẫu nhiên các mini-batch gồm các mẫu và nhãn? Ví dụ đơn giản trong [Section 10.1](#) đã giới thiệu một cách thực hiện. Hãy tổng quát hóa cách làm này một chút.

[Fig. 10.3.1](#), biểu diễn các cách để chia một câu thành các 5-gram, ở đây mỗi token là một ký tự. Ta có thể chọn tùy ý độ dời ở vị trí bắt đầu.

The Time Machine by H. G. Wells

Fig. 10.3.1: Các độ dời khác nhau dẫn đến các chuỗi con khác nhau khi phân tách văn bản.

Chúng ta nên chọn giá trị độ dời nào? Trong thực tế, tất cả các giá trị đó đều tốt như nhau. Nhưng nếu chọn tất cả các giá trị độ dời, dữ liệu sẽ khá dư thừa do trùng lặp lẫn nhau, đặc biệt trong trường hợp các chuỗi rất dài. Việc chỉ chọn một tập ngẫu nhiên các vị trí đầu cũng không tốt vì không đảm bảo sẽ bao quát đồng đều cả mảng. Ví dụ, nếu lấy ngẫu nhiên có hoàn lại  $n$  phần tử từ một tập có  $n$  phần tử, xác suất một phần tử cụ thể không được chọn là  $(1 - 1/n)^n \rightarrow e^{-1}$ . Nghĩa là ta không thể kỳ vọng vào sự bao quát đồng đều, ngay cả khi hoán vị ngẫu nhiên một tập giá trị độ dời. Thay vào đó, có thể sử dụng một cách đơn giản để có được cả tính *bao quát* và tính *ngẫu nhiên*, đó là: chọn một độ dời ngẫu nhiên, sau đó sử dụng tuần tự các giá trị tiếp theo. Điều này được mô tả trong phép lấy mẫu ngẫu nhiên và phép phân tách tuần tự dưới đây.

## Lấy Mẫu Ngẫu nhiên

Đoạn mã sau tạo ngẫu nhiên một minibatch dữ liệu. Ở đây, kích thước batch batch\_size biểu thị số mẫu trong mỗi minibatch, num\_steps biểu thị chiều dài mỗi mẫu (là số bước thời gian trong trường hợp chuỗi thời gian). Trong phép lấy mẫu ngẫu nhiên, mỗi mẫu là một chuỗi tùy ý được lấy ra từ chuỗi gốc. Hai minibatch ngẫu nhiên liên tiếp không nhất thiết phải liền kề nhau trong chuỗi gốc. Mục tiêu của ta là dự đoán phần tử tiếp theo dựa trên các phần tử đã thấy cho đến hiện tại, do đó nhãn của một mẫu chính là mẫu đó dịch chuyển sang phải một phần tử.

```
# Saved in the d2l package for later use
def seq_data_iter_random(corpus, batch_size, num_steps):
    # Offset the iterator over the data for uniform starts
    corpus = corpus[random.randint(0, num_steps):]
    # Subtract 1 extra since we need to account for label
    num_examples = ((len(corpus) - 1) // num_steps)
    example_indices = list(range(0, num_examples * num_steps, num_steps))
    random.shuffle(example_indices)

    def data(pos):
        # This returns a sequence of the length num_steps starting from pos
        return corpus[pos: pos + num_steps]

    # Discard half empty batches
    num_batches = num_examples // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Batch_size indicates the random examples read each time
        batch_indices = example_indices[i:(i+batch_size)]
        X = [data(j) for j in batch_indices]
        Y = [data(j + 1) for j in batch_indices]
        yield np.array(X), np.array(Y)
```

Hãy tạo ra một chuỗi từ 0 đến 29, rồi sinh các minibatch từ chuỗi đó với kích thước batch là 2 và số bước thời gian là 6. Nghĩa là tùy vào độ dời, ta có thể sinh tối đa 4 hoặc 5 cặp  $(x, y)$ . Với kích thước batch bằng 2, ta thu được 2 minibatch.

```
my_seq = list(range(30))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY: ', Y)
```

## Phân tách Tuần tự

Ngoài phép lấy mẫu ngẫu nhiên từ chuỗi gốc, chúng ta cũng có thể làm hai minibatch ngẫu nhiên liên tiếp có vị trí liền kề nhau trong chuỗi gốc.

```
# Saved in the d2l package for later use
def seq_data_iter_consecutive(corpus, batch_size, num_steps):
    # Offset for the iterator over the data for uniform starts
    offset = random.randint(0, num_steps)
    # Slice out data - ignore num_steps and just wrap around
    num_indices = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = np.array(corpus[offset:offset+num_indices])
    Ys = np.array(corpus[offset+1:offset+1+num_indices])
```

(continues on next page)

```
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_batches * num_steps, num_steps):
    X = Xs[:, i:(i+num_steps)]
    Y = Ys[:, i:(i+num_steps)]
    yield X, Y
```

Sử dụng các đối số như ở trên, ta sẽ in đầu vào  $X$  và nhãn  $Y$  cho mỗi minibatch sau khi phân tách tuần tự. Hai minibatch liên tiếp sẽ có vị trí trên chuỗi ban đầu liền kề nhau.

```
for X, Y in seq_data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
    print('X: ', X, '\nY: ', Y)
```

Hãy gộp hai hàm lấy mẫu theo hai cách trên vào một lớp để duyệt dữ liệu trong Gluon ở các phần sau.

```
# Saved in the d2l package for later use
class SeqDataLoader:
    """A 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_consecutive
        self.corpus, self.vocab = d2l.load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)
```

Cuối cùng, ta sẽ viết hàm `load_data_time_machine` trả về cả iterator dữ liệu và bộ từ vựng để sử dụng như các hàm `load_data` khác.

```
# Saved in the d2l package for later use
def load_data_time_machine(batch_size, num_steps, use_random_iter=False,
                           max_tokens=10000):
    data_iter = SeqDataLoader(
        batch_size, num_steps, use_random_iter, max_tokens)
    return data_iter, data_iter.vocab
```

### 10.3.5 Tóm tắt

- Mô hình ngôn ngữ là một kỹ thuật quan trọng trong xử lý ngôn ngữ tự nhiên.
- $n$ -gram là một mô hình khá tốt để xử lý các chuỗi dài bằng cách cắt giảm số phụ thuộc.
- Vấn đề của các chuỗi dài là chúng rất hiếm hoặc thậm chí không bao giờ xuất hiện.
- Định luật Zipf không chỉ mô tả phân phối từ 1-gram mà còn cả các  $n$ -gram khác.
- Có nhiều cấu trúc trong ngôn ngữ nhưng tần suất xuất hiện lại không đủ cao, để xử lý các tổ hợp từ hiếm ta sử dụng làm mượt Laplace.

- Hai giải pháp chủ yếu cho bài toán phân tách chuỗi là lấy mẫu ngẫu nhiên và phân tách tuần tự.
- Nếu tài liệu đủ dài, việc lãng phí một chút và loại bỏ các minibatch rỗng một nửa là điều chấp nhận được.

### 10.3.6 Bài tập

1. Giả sử có 100.000 từ trong tập dữ liệu huấn luyện. Mô hình 4-gram cần phải lưu trữ bao nhiêu tần số của từ đơn và cụm từ liền kề?
2. Hãy xem lại các ước lượng xác suất đã qua làm mượt. Tại sao chúng không chính xác? Gợi ý: chúng ta đang xử lý một chuỗi liền kề chứ không phải riêng lẻ.
3. Bạn sẽ mô hình hóa một cuộc đối thoại như thế nào?
4. Hãy ước tính luỹ thừa của định luật Zipf cho 1-gram, 2-gram, và 3-gram.
5. Hãy thử tìm các cách lấy mẫu minibatch khác.
6. Tại sao việc lấy giá trị độ dời ngẫu nhiên lại là một ý tưởng hay?
  - Liệu việc đó có làm các chuỗi dữ liệu văn bản tuân theo phân phối đều một cách hoàn hảo không?
  - Phải làm gì để có phân phối đều hơn?
7. Những vấn đề gì sẽ nảy sinh khi lấy mẫu minibatch từ một câu hoàn chỉnh? Có lợi ích gì khi lấy mẫu một câu hoàn chỉnh?

### 10.3.7 Thảo luận

- Tiếng Anh<sup>189</sup>
- Tiếng Việt<sup>190</sup>

### 10.3.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Đinh Đắc
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Nguyễn Cảnh Thưởng

<sup>189</sup> <https://discuss.mxnet.io/t/2361>

<sup>190</sup> <https://forum.machinelearningcoban.com/c/d21>

## 10.4 Mạng nơ-ron Hồi tiếp

Section 10.3 đã giới thiệu mô hình  $n$ -gram, trong đó xác suất có điều kiện của từ  $x_t$  tại vị trí  $t$  chỉ phụ thuộc vào  $n - 1$  từ trước đó. Nếu muốn kiểm tra ảnh hưởng có thể có của các từ ở trước vị trí  $t - (n - 1)$  đến từ  $x_t$ , ta cần phải tăng  $n$ . Tuy nhiên, cùng với đó số lượng tham số của mô hình cũng sẽ tăng lên theo hàm mũ, vì ta cần lưu  $|V|^n$  giá trị với một từ điển  $V$  nào đó. Do đó, thay vì mô hình hóa  $p(x_t | x_{t-1}, \dots, x_{t-n+1})$ , sẽ tốt hơn nếu ta sử dụng *mô hình biến tiềm ẩn* (*latent variable model*), trong đó

$$p(x_t | x_{t-1}, \dots, x_1) \approx p(x_t | x_{t-1}, h_t). \quad (10.4.1)$$

$h_t$  được gọi là *biến tiềm ẩn* và nó lưu trữ thông tin của chuỗi. Biến tiềm ẩn còn được gọi là *biến ẩn* (*hidden variable*), *trạng thái ẩn* (*hidden state*) hay *biến trạng thái ẩn* (*hidden state variable*). Trạng thái ẩn tại thời điểm  $t$  có thể được tính dựa trên cả đầu vào  $x_t$  và trạng thái ẩn  $h_{t-1}$  như sau

$$h_t = f(x_t, h_{t-1}). \quad (10.4.2)$$

Với một hàm  $f$  đủ mạnh, mô hình biến tiềm ẩn không phải là một phép xấp xỉ. Sau cùng,  $h_t$  có thể chỉ đơn thuần lưu lại tất cả dữ liệu đã quan sát được cho đến thời điểm hiện tại. Điều này đã được thảo luận tại Section 10.1. Tuy nhiên nó có thể khiến cho việc tính toán và lưu trữ trở nên nặng nề.

Chú ý rằng ta cũng sử dụng  $h$  để kí hiệu số lượng nút ẩn trong một tầng ẩn. Tầng ẩn và trạng thái ẩn là hai khái niệm rất khác nhau. Tầng ẩn, như đã được giải thích, là các tầng không thể nhìn thấy trong quá trình đi từ đầu vào đến đầu ra. Trạng thái ẩn, về mặt kỹ thuật là *đầu vào* của một bước tính toán tại một thời điểm xác định. Chúng chỉ có thể được tính dựa vào dữ liệu tại các vòng lặp trước đó. Về điểm này, trạng thái ẩn giống với các mô hình biến tiềm ẩn trong thống kê như mô hình phân cụm hoặc mô hình chủ đề (*topic model*), trong đó các cụm tác động đến đầu ra nhưng không thể quan sát trực tiếp.

Mạng nơ-ron hồi tiếp là mạng nơ-ron với các trạng thái ẩn. Trước khi tìm hiểu mô hình này, hãy cùng xem lại perceptron đa tầng tại Section 6.1.

### 10.4.1 Mạng Hồi tiếp không có Trạng thái ẩn

Xét một perception đa tầng với một tầng ẩn duy nhất. Giả sử ta có một minibatch  $\mathbf{X} \in \mathbb{R}^{n \times d}$  với  $n$  mẫu và  $d$  đầu vào. Gọi hàm kích hoạt của tầng ẩn là  $\phi$ . Khi đó, đầu ra của tầng ẩn  $\mathbf{H} \in \mathbb{R}^{n \times h}$  được tính như sau

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h). \quad (10.4.3)$$

Trong đó,  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$  là tham số trọng số,  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$  là hệ số điều chỉnh và  $h$  là số nút ẩn của tầng ẩn.

Biến ẩn  $\mathbf{H}$  được sử dụng làm đầu vào của tầng đầu ra. Tầng đầu ra được tính toán bởi

$$\mathbf{O} = \mathbf{HW}_{hq} + \mathbf{b}_q. \quad (10.4.4)$$

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 số và  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  là hệ số điều chỉnh của tầng đầu ra. Nếu đang giải quyết bài toán phân loại, ta có thể sử dụng softmax( $\mathbf{O}$ ) để tính phân phối xác suất của các lớp đầu ra.

Do bài toán này hoàn toàn tương tự với bài toán hồi quy được giải quyết trong [Section 10.1](#), ta sẽ bỏ qua các chi tiết ở đây. Và chỉ cần biết thêm rằng ta có thể chọn các cặp  $(x_t, x_{t-1})$  một cách ngẫu nhiên và ước lượng các tham số  $\mathbf{W}$  và  $\mathbf{b}$  của mạng thông qua phép vi phân tự động và hạ gradient ngẫu nhiên.

#### 10.4.2 Mạng Hồi tiếp có Trạng thái ẩn

Vấn đề sẽ khác đi hoàn toàn nếu ta sử dụng các trạng thái ẩn. Hãy xem xét cấu trúc này một cách chi tiết hơn. Chúng ta thường gọi vòng lặp thứ  $t$  là thời điểm  $t$  trong thuật toán tối ưu, nhưng trong mạng nơ-ron hồi tiếp, thời điểm  $t$  lại tương ứng với các bước trong một vòng lặp. Giả sử trong một vòng lặp ta có  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ ,  $t = 1, \dots, T$ . Và  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$  là biến ẩn tại bước thời gian  $t$  của chuỗi. Khác với perceptron đa tầng, ở đây ta lưu biến ẩn  $\mathbf{H}_{t-1}$  từ bước thời gian trước đó và dùng thêm một tham số trọng số mới  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  để mô tả việc sử dụng biến ẩn của bước thời gian trước đó trong bước thời gian hiện tại. Cụ thể, 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:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \quad (10.4.5)$$

So với [\(10.4.3\)](#), phương trình này có thêm  $\mathbf{H}_{t-1} \mathbf{W}_{hh}$ . Từ mối quan hệ giữa các biến ẩn  $\mathbf{H}_t$  và  $\mathbf{H}_{t-1}$  của các bước thời gian liền kề, ta biết rằng chúng đã lưu lại thông tin lịch sử của chuỗi cho tới bước thời gian hiện tại, giống như trạng thái hay bộ nhớ hiện thời của mạng nơ-ron. Vì vậy, một biến ẩn còn được gọi là một *trạng thái ẩn (hidden state)*. Vì trạng thái ẩn ở bước thời gian hiện tại và trước đó đều có cùng định nghĩa, phương trình trên được tính toán theo phương pháp hồi tiếp. Và đây cũng là lý do dẫn đến cái tên mạng nơ-ron hồi tiếp (*Recurrent Neural Network - RNN*).

Có rất nhiều phương pháp xây dựng RNN. Trong số đó, phổ biến nhất là RNN có trạng thái ẩn như định nghĩa ở phương trình trên. Tại bước thời gian  $t$ , tầng đầu ra trả về kết quả tính toán tương tự như trong perceptron đa tầng:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \quad (10.4.6)$$

Các tham số trong mô hình RNN bao gồm trọng số  $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$ ,  $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$  của tầng ẩn với hệ số điều chỉnh  $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$ , và trọng số  $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$  của tầng đầu ra với hệ số điều chỉnh  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ . Lưu ý rằng RNN luôn sử dụng cùng một bộ tham số mô hình cho dù tính toán ở các bước thời gian khác nhau. Vì thế, việc tăng số bước thời gian không làm tăng lượng tham số mô hình của RNN.

[Fig. 10.4.1](#) minh họa logic tính toán của một RNN tại ba bước thời gian liền kề. Tại bước thời gian  $t$ , sau khi nối đầu vào  $\mathbf{X}_t$  với trạng thái ẩn  $\mathbf{H}_{t-1}$  tại bước thời gian trước đó, ta có thể coi nó như đầu vào của một tầng kết nối đầy đủ với hàm kích hoạt  $\phi$ .

Đầu ra của tầng kết nối đầy đủ chính là trạng thái ẩn ở bước thời gian hiện tại  $\mathbf{H}_t$ . Tham số mô hình ở bước thời gian hiện tại là  $\mathbf{W}_{xh}$  nối với  $\mathbf{W}_{hh}$ , cùng với hệ số điều chỉnh  $\mathbf{b}_h$ . Trạng thái ẩn ở bước thời gian hiện tại  $t$ ,  $\mathbf{H}_t$  được sử dụng để tính trạng thái ẩn  $\mathbf{H}_{t+1}$  tại bước thời gian kế tiếp  $t+1$ . Hơn nữa,  $\mathbf{H}_t$  sẽ trở thành đầu vào cho tầng đầu ra  $\mathbf{O}_t$ , một tầng kết nối đầy đủ, ở bước thời gian hiện tại.

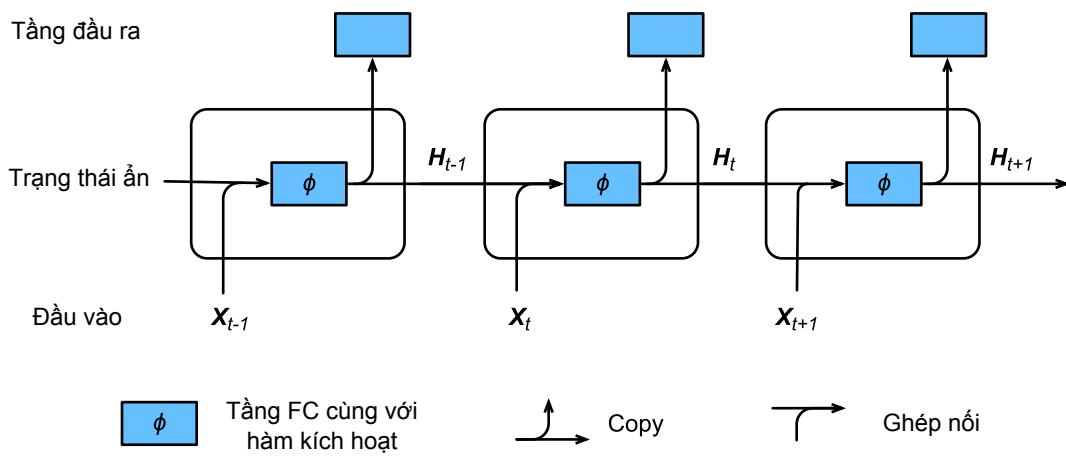


Fig. 10.4.1: Một RNN với một trạng thái ẩn.

### 10.4.3 Các bước trong một Mô hình Ngôn ngữ

Bây giờ hãy cùng xem cách xây dựng mô hình ngôn ngữ bằng RNN. Vì dùng từ thường dễ hiểu hơn dùng chữ, nên các từ sẽ được dùng làm đầu vào trong ví dụ đơn giản này.

Đặt kích thước minibatch là 1, với chuỗi văn bản là phần đầu của tập dữ liệu: “the time machine by H. G. Wells”. Fig. 10.4.2 minh họa cách ước lượng từ tiếp theo dựa trên từ hiện tại và các từ trước đó. Trong quá trình huấn luyện, chúng ta áp dụng softmax cho đầu ra tại mỗi bước thời gian, sau đó sử dụng hàm mất mát entropy chéo để tính toán sai số giữa kết quả và nhãn. Do trạng thái ẩn trong tầng ẩn được tính toán hồi tiếp, đầu ra của bước thời gian thứ 3,  $O_3$ , được xác định bởi chuỗi các từ “the”, “time” và “machine”. Vì từ tiếp theo của chuỗi trong dữ liệu huấn luyện là “by”, giá trị mất mát tại bước thời gian thứ 3 sẽ phụ thuộc vào phân phối xác suất của từ tiếp theo được tạo dựa trên chuỗi đặc trưng “the”, “time”, “machine” và nhãn “by” tại bước thời gian này.

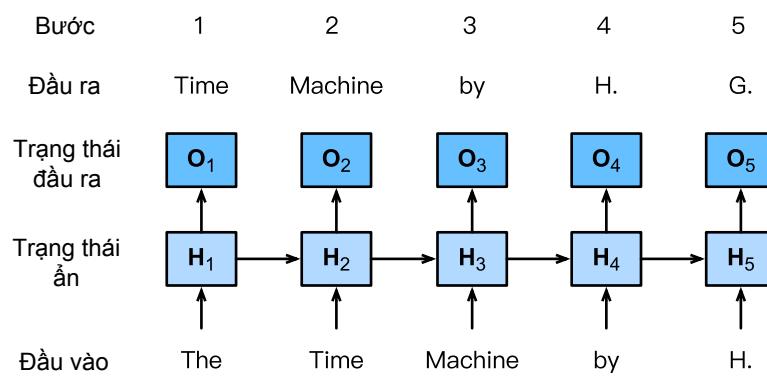


Fig. 10.4.2: Mô hình ngôn ngữ ở mức từ ngữ RNN. Đầu vào và chuỗi nhãn lần lượt là the time machine by H. và time machine by H. G.

Trong thực tế, mỗi từ được biểu diễn bởi một vector  $d$  chiều và kích thước batch thường là  $n > 1$ . Do đó, đầu vào  $X_t$  tại bước thời gian  $t$  sẽ là ma trận  $n \times d$ , giống hệt với những gì chúng ta đã thảo luận trước đây.

#### 10.4.4 Perplexity

Cuối cùng, hãy thảo luận về cách đo lường chất lượng của mô hình chuỗi. Một cách để làm việc này là kiểm tra mức độ gây ngạc nhiên của văn bản. Một mô hình ngôn ngữ tốt có thể dự đoán chính xác các token tiếp theo. Hãy xem xét các cách điền tiếp vào câu “Trời đang mưa” sau, được đề xuất bởi các mô hình ngôn ngữ khác nhau:

1. “Trời đang mưa bên ngoài”
2. “Trời đang mưa cây chuối”
3. “Trời đang mưa piouw;kcj pwepoiut”

Về chất lượng, ví dụ 1 rõ ràng là tốt nhất. Các từ được sắp xếp hợp lý và mạch lạc về mặt logic. Mặc dù nó có thể không phản ánh chính xác hoàn toàn mặt ngữ nghĩa của các từ theo sau (“ở San Francisco” và “vào mùa đông” cũng là các phần mở rộng hoàn toàn hợp lý), mô hình vẫn có thể nắm bắt những từ nghe khá phù hợp. Ví dụ 2 thì tệ hơn đáng kể, mô hình này đã nối dài câu ra theo cách vô nghĩa. Tuy nhiên, ít nhất mô hình đã viết đúng chính tả và học được phần nào sự tương quan giữa các từ. Cuối cùng, ví dụ 3 là một mô hình được huấn luyện kém, không khớp được dữ liệu.

Chúng ta có thể đo lường chất lượng của mô hình bằng cách tính xác suất  $p(w)$ , tức độ hợp lý của một chuỗi  $w$ . Thật không may, đây là một con số khó để hiểu và so sánh. Xét cho cùng, các chuỗi ngắn có khả năng xuất hiện cao hơn các chuỗi dài, do đó việc đánh giá mô hình trên kiệt tác “Chiến tranh và Hòa bình”<sup>191</sup> của Tolstoy chắc chắn sẽ cho kết quả thấp hơn nhiều so với tiểu thuyết “Hoàng tử bé”<sup>192</sup> của Saint-Exupery. Thứ còn thiếu ở đây là một cách tính trung bình qua độ dài chuỗi.

Lý thuyết thông tin rất có ích trong trường hợp này và chúng tôi sẽ giới thiệu thêm về nó trong Section 20.11. Nếu chúng ta muốn nén văn bản, ta có thể yêu cầu ước lượng ký hiệu tiếp theo với bộ ký hiệu hiện tại. Số lượng bit tối thiểu cần thiết là  $-\log_2 p(x_t | x_{t-1}, \dots, x_1)$ . Một mô hình ngôn ngữ tốt sẽ cho phép chúng ta dự đoán từ tiếp theo một cách khá chính xác và do đó số bit cần thiết để nén chuỗi là rất thấp. Vì vậy, ta có thể đo lường mô hình ngôn ngữ bằng số bit trung bình cần sử dụng.

$$\frac{1}{n} \sum_{t=1}^n -\log p(x_t | x_{t-1}, \dots, x_1). \quad (10.4.7)$$

Điều này giúp ta so sánh được chất lượng mô hình trên các tài liệu có độ dài khác nhau. Vì lý do lịch sử, các nhà khoa học xử lý ngôn ngữ tự nhiên thích sử dụng một đại lượng gọi là *perplexity* (*độ rối rắm, hỗn độn*) thay vì *bitrate* (*tốc độ bit*). Nói ngắn gọn, nó là luỹ thừa của biểu thức trên:

$$\text{PPL} := \exp \left( -\frac{1}{n} \sum_{t=1}^n \log p(x_t | x_{t-1}, \dots, x_1) \right). \quad (10.4.8)$$

Giá trị này có thể được hiểu rõ nhất như là trung bình điều hòa của số lựa chọn thực tế mà ta có khi quyết định chọn từ nào là từ tiếp theo. Lưu ý rằng perplexity khái quát hóa một cách tự nhiên ý tưởng của hàm mất mát entropy chéo định nghĩa ở phần hồi quy softmax (Section 5.4). Điều này có nghĩa là khi xét một ký hiệu duy nhất, perplexity chính là lũy thừa của entropy chéo. Hãy cùng xem xét một số trường hợp:

<sup>191</sup> <https://www.gutenberg.org/files/2600/2600-h/2600-h.htm>

<sup>192</sup> [https://en.wikipedia.org/wiki/The\\_Little\\_Prince](https://en.wikipedia.org/wiki/The_Little_Prince)

- Trong trường hợp tốt nhất, mô hình luôn ước tính xác suất của ký hiệu tiếp theo là 1. Khi đó perplexity của mô hình là 1.
- Trong trường hợp xấu nhất, mô hình luôn dự đoán xác suất của nhãn là 0. Khi đó perplexity là vô hạn.
- Tại mức nền, mô hình dự đoán một phân phối đều trên tất cả các token. Trong trường hợp này, perplexity bằng với kích thước của từ điển len(vocab).
- Thực chất, nếu chúng ta lưu trữ chuỗi không nén, đây là cách tốt nhất có thể để mã hóa chúng. Vì vậy, nó cho ta một cận trên mà bất kỳ mô hình nào cũng phải thỏa mãn.

#### 10.4.5 Tóm tắt

- Một mạng sử dụng tính toán hồi tiếp được gọi là mạng nơ-ron hồi tiếp (RNN).
- Trạng thái ẩn của RNN có thể tổng hợp được thông tin lịch sử của chuỗi cho tới bước thời gian hiện tại.
- Số lượng tham số của mô hình RNN không tăng khi số lượng bước thời gian tăng.
- Ta có thể tạo các mô hình ngôn ngữ sử dụng một RNN ở cấp độ ký tự.

#### 10.4.6 Bài tập

1. Nếu sử dụng RNN để dự đoán ký tự tiếp theo trong chuỗi văn bản thì ta sẽ cần đầu ra có bao nhiêu chiều?
2. Thủ thiết kế một ánh xạ trong đó các trạng thái ẩn của RNN là chính xác (không chỉ là xấp xỉ). Gợi ý: nếu có một số lượng từ hữu hạn thì sao?
3. Điều gì xảy ra với gradient nếu ta thực hiện phép lan truyền ngược qua một chuỗi dài?
4. Một số vấn đề liên quan đến mô hình chuỗi đơn giản được mô tả bên trên là gì?

#### 10.4.7 Thảo luận

- [Tiếng Anh](#)<sup>193</sup>
- [Tiếng Việt](#)<sup>194</sup>

#### 10.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Nguyễn Duy Du

<sup>193</sup> <https://discuss.mxnet.io/t/2362>

<sup>194</sup> <https://forum.machinelearningcoban.com/c/d21>

- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Trần Yến Thy
- Phạm Hồng Vinh
- Nguyễn Cảnh Thưởng

## 10.5 Lập trình Mạng nơ-ron Hồi tiếp từ đầu

Trong phần này, ta lập trình từ đầu mô hình ngôn ngữ được giới thiệu trong Section 10. Mô hình này dựa trên mạng nơ-ron hồi tiếp ở cấp độ ký tự (*character-level*) được huấn luyện trên tiểu thuyết *The Time Machine* (*Cỗ máy thời gian*) của H. G. Wells. Cũng như trước, ta bắt đầu với việc đọc tập dữ liệu được đề cập trong Section 10.3.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import autograd, np, npx, gluon
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

### 10.5.1 Biểu diễn One-hot

Lưu ý rằng mỗi token được biểu diễn bằng một chỉ số (*numerical index*) trong `train_iter`. Đưa trực tiếp các chỉ số này vào mạng nơ-ron sẽ gây khó khăn cho việc học. Do đó, mỗi token thường được biểu diễn dưới dạng một vector đặc trưng mang nhiều thông tin hơn. Cách đơn giản nhất là sử dụng *biểu diễn one-hot* (*one-hot encoding*).

Nói ngắn gọn, ta ánh xạ mỗi chỉ số thành một vector đơn vị khác nhau: giả sử số token không trùng lặp trong bộ từ vựng là  $N$  (`len(vocab)`) và chỉ số của chúng nằm trong khoảng từ 0 đến  $N - 1$ . Với token chỉ số  $i$ , ta tạo một vector  $\mathbf{e}_i$  độ dài  $N$  có các phần tử bằng 0, trừ phần tử ở vị trí  $i$  bằng 1. Vector này là vector one-hot của token. Các vector one-hot với các chỉ số 0 và 2 được minh họa phía dưới.

```
npx.one_hot(np.array([0, 2]), len(vocab))
```

Kích thước minibatch mà chúng ta lấy mẫu mỗi lần là (kích thước batch, bước thời gian). Hàm `one_hot` biến đổi một minibatch như vậy thành một tensor 3 chiều với kích thước chiều cuối cùng bằng kích thước bộ từ vựng. Chúng ta thường chuyển vị trí đầu vào để có đầu ra với kích thước (bước thời gian, kích thước batch, kích thước bộ từ vựng), phù hợp hơn để đưa vào mô hình chuỗi.

```
X = np.arange(batch_size * num_steps).reshape(batch_size, num_steps)
npx.one_hot(X.T, len(vocab)).shape
```

### 10.5.2 Khởi tạo Tham số Mô hình

Tiếp theo, ta khởi tạo các tham số cho mô hình RNN. Số nút ẩn num\_hiddens là tham số có thể điều chỉnh.

```
def get_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)
    # Hidden layer parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = np.zeros(num_hiddens, ctx=ctx)
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=ctx)
    # Attach gradients
    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.attach_grad()
    return params
```

### 10.5.3 Mô hình RNN

Đầu tiên, chúng ta khởi tạo trạng thái ẩn bằng hàm init\_rnn\_state. Hàm này trả về tuple gồm một ndarray chứa giá trị 0 và có kích thước là (kích thước batch, số nút ẩn). Trả về tuple giúp ta dễ dàng xử lý các tình huống khi trạng thái ẩn chứa nhiều biến (ví dụ: khi ta cần khởi tạo nhiều tầng được kết hợp trong RNN).

```
def init_rnn_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

Hàm rnn sau định nghĩa cách tính toán trạng thái ẩn và đầu ra tại một bước thời gian. Hàm kích hoạt ở đây là tanh. Như đã đề cập trong Section 6.1, giá trị trung bình của hàm tanh là 0, khi các phần tử được phân bổ đều trên trực số thực.

```
def rnn(inputs, state, params):
    # Inputs shape: (num_steps, batch_size, vocab_size)
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []
    for X in inputs:
        H = np.tanh(np.dot(X, W_xh) + np.dot(H, W_hh) + b_h)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

Sau khi đã định nghĩa tất cả các hàm, ta tạo một lớp để bao các hàm này lại và lưu trữ các tham số.

```
# Saved in the d2l package for later use
class RNNModelScratch:
```

(continues on next page)

```
"""A RNN Model based on scratch implementations."""

def __init__(self, vocab_size, num_hiddens, ctx,
             get_params, init_state, forward):
    self.vocab_size, self.num_hiddens = vocab_size, num_hiddens
    self.params = get_params(vocab_size, num_hiddens, ctx)
    self.init_state, self.forward_fn = init_state, forward

def __call__(self, X, state):
    X = npx.one_hot(X.T, self.vocab_size)
    return self.forward_fn(X, state, self.params)

def begin_state(self, batch_size, ctx):
    return self.init_state(batch_size, self.num_hiddens, ctx)
```

Hãy kiểm tra nhanh chiều của đầu vào và đầu ra, và xem chiều của trạng thái ẩn có thay đổi hay không.

```
num_hiddens, ctx = 512, d2l.try_gpu()
model = RNNModelScratch(len(vocab), num_hiddens, ctx, get_params,
                        init_rnn_state, rnn)
state = model.begin_state(X.shape[0], ctx)
Y, new_state = model(X.as_in_ctx(ctx), state)
Y.shape, len(new_state), new_state[0].shape
```

Có thể thấy kích thước đầu ra là ( $\text{số bước} \times \text{kích thước batch}$ , kích thước bộ từ vựng), trong khi kích thước trạng thái ẩn vẫn giữ nguyên là (kích thước batch, số nút ẩn).

#### 10.5.4 Dự đoán

Trước tiên chúng ta giải thích hàm dự đoán thường xuyên được dùng để kiểm tra trong quá trình huấn luyện. Hàm này dự đoán num\_predicts ký tự tiếp theo dựa trên prefix (một chuỗi chứa một vài ký tự). Ở các ký tự đầu tiên trong chuỗi, ta chỉ cập nhật trạng thái ẩn rồi sau đó mới bắt đầu tạo ra các ký tự mới.

```
# Saved in the d2l package for later use
def predict_ch8(prefix, num_predicts, model, vocab, ctx):
    state = model.begin_state(batch_size=1, ctx=ctx)
    outputs = [vocab[prefix[0]]]

    def get_input():
        return np.array([outputs[-1]], ctx=ctx).reshape(1, 1)
    for y in prefix[1:]: # Warmup state with prefix
        _, state = model(get_input(), state)
        outputs.append(vocab[y])
    for _ in range(num_predicts): # Predict num_predicts steps
        Y, state = model(get_input(), state)
        outputs.append(int(Y.argmax(axis=1).reshape(1)))
    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

Ta chạy thử hàm predict\_ch8 trước. Lúc này đầu ra sẽ là các dự đoán vô nghĩa do mạng chưa được huấn luyện. Ta khởi tạo mạng với chuỗi traveller và cho nó tạo ra thêm 10 ký tự.

```
predict_ch8('time traveller ', 10, model, vocab, ctx)
```

### 10.5.5 Gọt Gradient

Với chuỗi độ dài  $T$ , trong một vòng lặp lan truyền ngược ta tính toán gradient qua  $T$  bước thời gian, dẫn đến một chuỗi các tích của ma trận có độ phức tạp  $\mathcal{O}(T)$ . Như đã đề cập trong [Section 6.8](#), khi  $T$  lớn việc này có thể dẫn đến mất ổn định số học, biểu hiện qua hiện tượng bùng nổ hoặc tiêu biến gradient. Do đó, các mô hình RNN thường cần một chút hỗ trợ để ổn định việc huấn luyện.

Nhớ lại rằng khi giải quyết vấn đề tối ưu, ta thực hiện cập nhật trọng số  $\mathbf{w}$  ngược hướng gradient  $\mathbf{g}_t$  trên một minibatch, theo công thức  $\mathbf{w} - \eta \cdot \mathbf{g}_t$ . Giả sử hàm mục tiêu là hàm liên tục Lipschitz với hằng số  $L$ , tức:

$$|l(\mathbf{w}) - l(\mathbf{w}')| \leq L \|\mathbf{w} - \mathbf{w}'\|. \quad (10.5.1)$$

Trong trường hợp này, có thể nói khi cập nhật vector trọng số theo  $\eta \cdot \mathbf{g}_t$ , sự thay đổi sẽ không lớn hơn  $L\eta \|\mathbf{g}_t\|$ . Điều này vừa có lợi vừa có hại. Có hại ở chỗ tốc độ tối ưu bị giới hạn, có lợi ở chỗ mức độ sai lệch khi tối ưu sai hướng cũng bị hạn chế.

Đôi khi gradient có thể khá lớn và do đó thuật toán tối ưu không hội tụ. Vấn đề này có thể được giải quyết bằng cách giảm tốc độ học  $\eta$  hoặc sử dụng một số thủ thuật liên quan tới đạo hàm bậc cao hơn. Nhưng nếu gradient hiếm khi đạt giá trị lớn, cách giải quyết như vậy không đảm bảo hội tụ hoàn toàn. Một cách khác là gọt gradient (*gradient clipping*) bằng cách chiếu gradient lên mặt cầu bán kính  $\theta$  qua công thức:

$$\mathbf{g} \leftarrow \min \left( 1, \frac{\theta}{\|\mathbf{g}\|} \right) \mathbf{g}. \quad (10.5.2)$$

Như vậy chuẩn của gradient sẽ không vượt quá  $\theta$  và gradient sau khi gọt sẽ cùng hướng gradient  $\mathbf{g}$  ban đầu. Gọt gradient có tác dụng phụ tích cực là hạn chế ảnh hưởng quá lớn của bất kỳ minibatch nào (hoặc bất kỳ mẫu nào) lên các trọng số, làm cho mô hình ổn định hơn. Dù không giải quyết được hoàn toàn vấn đề, đây là một kỹ thuật đơn giản để làm giảm nhẹ vấn đề bùng nổ gradient.

Dưới đây, ta định nghĩa hàm gọt gradient, dùng cho cả mô hình lập trình từ đầu `RNNModelScratch` và mô hình tạo bằng Gluon. Lưu ý rằng ta tính chuẩn của gradient trên tất cả các tham số.

```
# Saved in the d2l package for later use
def grad_clipping(model, theta):
    if isinstance(model, gluon.Block):
        params = [p.data() for p in model.collect_params().values()]
    else:
        params = model.params
    norm = math.sqrt(sum((p.grad ** 2).sum() for p in params))
    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm
```

### 10.5.6 Huấn luyện

Trước tiên, ta định nghĩa hàm huấn luyện trên một epoch dữ liệu. Quá trình huấn luyện ở đây khác với Section 5.6 ở ba điểm:

1. Các phương pháp lấy mẫu khác nhau cho dữ liệu tuần tự (lấy mẫu ngẫu nhiên và phân tách tuần tự) sẽ dẫn đến sự khác biệt trong việc khởi tạo các trạng thái ẩn.
2. Ta gọt gradient trước khi cập nhật tham số mô hình. Việc này đảm bảo rằng mô hình sẽ không phân kỳ ngay cả khi gradient bùng nổ tại một thời điểm nào đó trong quá trình huấn luyện, đồng thời tự động giảm biên độ của bước cập nhật một cách hiệu quả.
3. Ta sử dụng perplexity để đánh giá mô hình. Phương pháp này đảm bảo rằng các chuỗi có độ dài khác nhau có thể so sánh được.

Khi thực hiện lấy mẫu tuần tự, ta chỉ khởi tạo trạng thái ẩn khi bắt đầu mỗi epoch. Vì mẫu thứ  $i^{\text{th}}$  trong minibatch tiếp theo liền kề với mẫu thứ  $i^{\text{th}}$  trong minibatch hiện tại nên ta có thể sử dụng trực tiếp trạng thái ẩn hiện tại cho minibatch tiếp theo, chỉ cần tách gradient để tính riêng cho mỗi minibatch. Còn khi thực hiện lấy mẫu ngẫu nhiên, ta cần tái khởi tạo trạng thái ẩn cho mỗi vòng lặp vì mỗi mẫu được lấy ra ở vị trí ngẫu nhiên. Giống như hàm `train_epoch_ch3` trong Section 5.6, ta sử dụng đối số `updater` để tổng quát hóa cả trường hợp lập trình súc tích với Gluon và lập trình từ đầu.

```
# Saved in the d2l package for later use
def train_epoch_ch8(model, train_iter, loss, updater, ctx, use_random_iter):
    state, timer = None, d2l.Timer()
    metric = d2l.Accumulator(2) # loss_sum, num_examples
    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 = model.begin_state(batch_size=X.shape[0], ctx=ctx)
        else:
            for s in state:
                s.detach()
        y = Y.T.reshape(-1)
        X, y = X.as_in_ctx(ctx), y.as_in_ctx(ctx)
        with autograd.record():
            py, state = model(X, state)
            l = loss(py, y).mean()
        l.backward()
        grad_clipping(model, 1)
        updater(batch_size=1) # Since used mean already
        metric.add(l * y.size, y.size)
    return math.exp(metric[0]/metric[1]), metric[1]/timer.stop()
```

Hàm huấn luyện này hỗ trợ cả mô hình sử dụng Gluon và mô hình lập trình từ đầu.

```
# Saved in the d2l package for later use
def train_ch8(model, train_iter, vocab, lr, num_epochs, ctx,
              use_random_iter=False):
    # Initialize
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='perplexity',
                            legend=['train'], xlim=[1, num_epochs])
```

(continues on next page)

```

if isinstance(model, gluon.Block):
    model.initialize(ctx=ctx, force_reinit=True, init=init.Normal(0.01))
    trainer = gluon.Trainer(model.collect_params(),
                            'sgd', {'learning_rate': lr})

    def updater(batch_size):
        return trainer.step(batch_size)
else:
    def updater(batch_size):
        return d2l.sgd(model.params, lr, batch_size)

def predict(prefix):
    return predict_ch8(prefix, 50, model, vocab, ctx)

# Train and check the progress.
for epoch in range(num_epochs):
    ppl, speed = train_epoch_ch8(
        model, train_iter, loss, updater, ctx, use_random_iter)
    if epoch % 10 == 0:
        print(predict('time traveller'))
        animator.add(epoch+1, [ppl])
    print('Perplexity %.1f, %d tokens/sec on %s' % (ppl, speed, ctx))
    print(predict('time traveller'))
    print(predict('traveller'))

```

Bây giờ ta có thể huấn luyện mô hình. Do chỉ sử dụng 10.000 token trong tập dữ liệu, mô hình này cần nhiều epoch hơn để hội tụ.

```

num_epochs, lr = 500, 1
train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)

```

Cuối cùng, ta kiểm tra kết quả khi lấy mẫu ngẫu nhiên.

```

train_ch8(model, train_iter, vocab, lr, num_epochs, ctx, use_random_iter=True)

```

Mặc dù học được nhiều điều từ việc lập trình từ đầu nhưng cách làm này không thực sự tiện lợi. Trong phần tiếp theo, ta sẽ tìm hiểu cách cải thiện đáng kể mô hình hiện tại, nhanh và dễ lập trình hơn.

### 10.5.7 Tóm tắt

- Mô hình chuỗi cần khởi tạo trạng thái cho quá trình huấn luyện.
- Giữa các mô hình chuỗi, ta cần đảm bảo tách gradient để chắc chắn rằng phép tính vi phân tự động không ảnh hưởng ra ngoài phạm vi mẫu hiện tại.
- Mô hình ngôn ngữ RNN đơn giản bao gồm một bộ mã hóa, một mô hình RNN và một bộ giải mã.
- Gọt gradient có thể hạn chế sự bùng nổ gradient nhưng không thể khắc phục được vấn đề tiêu biến gradient.
- Perplexity đánh giá chất lượng mô hình trên các chuỗi có độ dài khác nhau, được tính bằng trung bình lũy thừa của mất mát entropy chéo.

- Phân tách tuần tự cho kết quả mô hình tốt hơn.

### 10.5.8 Bài tập

1. Chỉ ra rằng mỗi biểu diễn one-hot tương đương với một embedding khác nhau cho từng đối tượng.
2. Điều chỉnh các siêu tham số để cải thiện perplexity.
  - Bạn có thể giảm perplexity xuống bao nhiêu? Hãy thay đổi embedding, số nút ẩn, tốc độ học, vv.
  - Mô hình này sẽ hoạt động tốt đến đâu trên các cuốn sách khác của H. G. Wells, ví dụ như [The War of the Worlds](#)<sup>195</sup>.
3. Thay đổi hàm dự đoán bằng việc lấy mẫu thay vì chọn ký tự tiếp theo là ký tự có khả năng cao nhất.
  - Điều gì sẽ xảy ra?
  - Điều chỉnh để mô hình ưu tiên các đầu ra có khả năng cao hơn, ví dụ, bằng cách lấy mẫu sử dụng  $q(w_t | w_{t-1}, \dots, w_1) \propto p^\alpha(w_t | w_{t-1}, \dots, w_1)$  với  $\alpha > 1$ .
4. Điều gì sẽ xảy ra nếu ta chạy mã nguồn phần này mà không gọt gradient?
5. Thay đổi phép lấy mẫu phân tách tuần tự để các trạng thái ẩn không bị tách khỏi đồ thị tính toán. Thời gian chạy và độ chính xác có thay đổi không?
6. Thay hàm kích hoạt bằng ReLU và thực hiện lại các thử nghiệm.
7. Chứng minh rằng perplexity là nghịch đảo trung bình điều hòa (*harmonic mean*) của xác suất có điều kiện của từ.

### 10.5.9 Thảo luận

- [Tiếng Anh](#)<sup>196</sup>
- [Tiếng Việt](#)<sup>197</sup>

### 10.5.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Trần Yến Thy
- Nguyễn Lê Quang Nhật
- Nguyễn Duy Du
- Phạm Minh Đức

<sup>195</sup> <http://www.gutenberg.org/ebooks/36>

<sup>196</sup> <https://discuss.mxnet.io/t/2364>

<sup>197</sup> <https://forum.machinelearningcoban.com/c/d21>

- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Cảnh Thượng

## 10.6 Lập trình súc tích Mạng nơ-ron Hồi tiếp

Dù Section 10.5 đã mô tả cách lập trình mạng nơ-ron hồi tiếp từ đầu một cách chi tiết, tuy nhiên cách làm này không được nhanh và thuận tiện. Phần này sẽ hướng dẫn cách lập trình cùng một mô hình ngôn ngữ nhưng hiệu quả hơn bằng các hàm của Gluon. Như trước, ta cũng bắt đầu với việc đọc kho dữ liệu “Cỗ máy Thời gian”.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn, rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

### 10.6.1 Định nghĩa Mô hình

Mô-đun `rnn` của Gluon đã lập trình sẵn mạng nơ-ron hồi tiếp (cùng với các mô hình chuỗi khác). Ta xây dựng tầng hồi tiếp `rnn_layer` với một tầng ẩn có 256 nút rồi khởi tạo các trọng số.

```
num_hiddens = 256
rnn_layer = rnn.RNN(num_hiddens)
rnn_layer.initialize()
```

Việc khởi tạo trạng thái cũng khá đơn giản, chỉ cần gọi phương thức `rnn_layer.begin_state(batch_size)`. Phương thức này trả về một trạng thái ban đầu cho mỗi phần tử trong minibatch, có kích thước là (số tầng ẩn, kích thước batch, số nút ẩn). Số tầng ẩn mặc định là 1. Thực ra ta chưa thảo luận việc mạng có nhiều tầng sẽ như thế nào — điều này sẽ được đề cập ở Section 11.3. Tạm thời, có thể nói rằng trong mạng nhiều tầng, đầu ra của một RNN sẽ là đầu vào của RNN tiếp theo.

```
batch_size = 1
state = rnn_layer.begin_state(batch_size=batch_size)
len(state), state[0].shape
```

Với một biến trạng thái và một đầu vào, ta có thể tính đầu ra với trạng thái vừa được cập nhật.

```
num_steps = 1
X = np.random.uniform(size=(num_steps, batch_size, len(vocab)))
Y, state_new = rnn_layer(X, state)
Y.shape, len(state_new), state_new[0].shape
```

Tương tự Section 10.5, ta định nghĩa khối `RNNModel` bằng cách kế thừa lớp `Block` để xây dựng mạng nơ-ron hồi tiếp hoàn chỉnh. Chú ý rằng `rnn_layer` chỉ chứa các tầng hồi tiếp ẩn và ta cần tạo riêng biệt một tầng đầu ra, trong khi ở phần trước tầng đầu ra được tích hợp sẵn trong khối `rnn`.

```

# Saved in the d2l package for later use
class RNNModel(nn.Block):
    def __init__(self, rnn_layer, vocab_size, **kwargs):
        super(RNNModel, self).__init__(**kwargs)
        self.rnn = rnn_layer
        self.vocab_size = vocab_size
        self.dense = nn.Dense(vocab_size)

    def forward(self, inputs, state):
        X = npx.one_hot(inputs.T, self.vocab_size)
        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.dense(Y.reshape(-1, Y.shape[-1]))
        return output, state

    def begin_state(self, *args, **kwargs):
        return self.rnn.begin_state(*args, **kwargs)

```

## 10.6.2 Huấn luyện và Dự đoán

Trước khi huấn luyện, hãy thử dự đoán bằng mô hình có trọng số ngẫu nhiên.

```

ctx = d2l.try_gpu()
model = RNNModel(rnn_layer, len(vocab))
model.initialize(force_reinit=True, ctx=ctx)
d2l.predict_ch8('time traveller', 10, model, vocab, ctx)

```

Khá rõ ràng, mô hình này không tốt. Tiếp theo, ta gọi hàm `train_ch8` với các siêu tham số định nghĩa trong [Section 10.5](#) để huấn luyện mô hình bằng Gluon.

```

num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)

```

So với phần trước, mô hình này đạt được perplexity tương đương, nhưng thời gian huấn luyện tốt hơn do các đoạn mã được tối ưu hơn.

## 10.6.3 Tóm tắt

- Mô-đun `rnn` của Gluon đã lập trình sẵn tầng mạng nơ-ron hồi tiếp.
- Mỗi thực thể của `nn.RNN` trả về đầu ra và trạng thái ẩn sau lượt truyền xuôi. Lượt truyền xuôi này không bao gồm tính toán tại tầng đầu ra.
- Như trước, đồ thị tính toán cần được tách khỏi các bước trước đó để đảm bảo hiệu năng.

#### 10.6.4 Bài tập

1. So sánh với cách lập trình từ đầu ở phần trước.
  - Tại sao lập trình bằng Gluon chạy nhanh hơn?
  - Nếu bạn nhận thấy khác biệt đáng kể nào ngoài tốc độ, hãy thử tìm hiểu tại sao.
2. Bạn có thể làm quá khớp mô hình này không? Hãy thử
  - Tăng số nút ẩn.
  - Tăng số vòng lặp.
  - Thay đổi tham số gọt (*clipping*) thì sao?
3. Hãy lập trình mô hình tự hồi quy ở phần giới thiệu của chương này bằng RNN.
4. Nếu tăng số tầng ẩn của mô hình RNN thì sao? Bạn có thể làm mô hình hoạt động không?
5. Có thể nén văn bản bằng cách sử dụng mô hình này không?
  - Nếu có thì cần bao nhiêu bit?
  - Tại sao không ai sử dụng mô hình này để nén văn bản? Gợi ý: bản thân bộ nén thì sao?

#### 10.6.5 Thảo luận

- Tiếng Anh<sup>198</sup>
- Tiếng Việt<sup>199</sup>

#### 10.6.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc

### 10.7 Lan truyền Ngược qua Thời gian

Cho đến nay chúng ta liên tục nhắc đến những vấn đề như *bùng nổ gradient*, *tiêu biến gradient*, *cắt xén lan truyền ngược* và sự cần thiết của việc *tách đồ thị tính toán*. Ví dụ, trong phần trước chúng ta gọi hàm `s.detach()` trên chuỗi. Vì muốn nhanh chóng xây dựng và quan sát cách một mô hình hoạt động nên những vấn đề này chưa được giải thích một cách đầy đủ. Trong phần này chúng ta sẽ nghiên cứu sâu và chi tiết hơn về lan truyền ngược cho các mô hình chuỗi và giải thích nguyên lý toán học đằng sau. Để hiểu chi tiết hơn về tính ngẫu nhiên và lan truyền ngược, hãy tham khảo bài báo ([Tallec & Ollivier, 2017](#)).

<sup>198</sup> <https://discuss.mxnet.io/t/2365>

<sup>199</sup> <https://forum.machinelearningcoban.com/c/d21>

Chúng ta đã thấy một vài hậu quả của bùng nổ gradient khi lập trình mạng nơ-ron hồi tiếp (Section 10.5). Cụ thể, nếu bạn đã làm xong bài tập ở phần đó, bạn sẽ thấy rằng việc gọt gradient đóng vai trò rất quan trọng để đảm bảo mô hình hội tụ. Để có cái nhìn rõ hơn về vấn đề này, trong phần này chúng ta sẽ xem xét cách tính gradient cho các mô hình chuỗi. Lưu ý rằng, về mặt khái niệm thì không có gì mới ở đây. Sau cùng, chúng ta vẫn chỉ đơn thuần áp dụng các quy tắc dây chuyền để tính gradient. Tuy nhiên, việc ôn lại lan truyền ngược (Section 6.7) vẫn rất hữu ích.

Lượt truyền xuôi trong mạng nơ-ron hồi tiếp tương đối đơn giản. *Lan truyền ngược qua thời gian* thực chất là một ứng dụng cụ thể của lan truyền ngược trong mạng nơ-ron hồi tiếp. Nó đòi hỏi chúng ta mở rộng mạng nơ-ron hồi tiếp theo từng bước thời gian một để thu được sự phụ thuộc giữa các biến mô hình và các tham số. Sau đó, dựa trên quy tắc dây chuyền, chúng ta áp dụng lan truyền ngược để tính toán và lưu các giá trị gradient. Vì chuỗi có thể khá dài nên sự phụ thuộc trong chuỗi cũng có thể rất dài. Ví dụ, đối với một chuỗi gồm 1000 ký tự, ký tự đầu tiên có thể ảnh hưởng đáng kể tới ký tự ở vị trí 1000. Điều này không thực sự khả thi về mặt tính toán (cần quá nhiều thời gian và bộ nhớ) và nó đòi hỏi hơn 1000 phép nhân ma trận-vector trước khi thu được các giá trị gradient khó nắm bắt này. Đây là một quá trình chứa đầy sự bất định về mặt tính toán và thống kê. Trong phần tiếp theo chúng ta sẽ làm sáng tỏ những gì sẽ xảy ra và cách giải quyết vấn đề này trong thực tế.

### 10.7.1 Mạng Hồi tiếp Giản thể

Hãy bắt đầu với một mô hình đơn giản về cách mà mạng RNN hoạt động. Mô hình này bỏ qua các chi tiết cụ thể của trạng thái ẩn và cách trạng thái này được cập nhật. Những chi tiết này không quan trọng đối với việc phân tích dưới đây mà chỉ khiến các ký hiệu trở nên lộn xộn và phức tạp quá mức. Trong mô hình đơn giản này, chúng ta ký hiệu  $h_t$  là trạng thái ẩn,  $x_t$  là đầu vào, và  $o_t$  là đầu ra tại bước thời gian  $t$ . Bên cạnh đó,  $w_h$  và  $w_o$  tương ứng với trọng số của các trạng thái ẩn và tầng đầu ra. Kết quả là, các trạng thái ẩn và kết quả đầu ra tại mỗi bước thời gian có thể được giải thích như sau

$$h_t = f(x_t, h_{t-1}, w_h) \text{ và } o_t = g(h_t, w_o). \quad (10.7.1)$$

Do đó, chúng ta có một chuỗi các giá trị  $\{\dots, (h_{t-1}, x_{t-1}, o_{t-1}), (h_t, x_t, o_t), \dots\}$  phụ thuộc vào nhau thông qua phép tính đệ quy. Lượt truyền xuôi khá đơn giản. Những gì chúng ta cần là lặp qua từng bộ ba  $(x_t, h_t, o_t)$  một. Sau đó, sự khác biệt giữa kết quả đầu ra  $o_t$  và các giá trị mục tiêu mong muốn  $y_t$  được tính bằng một hàm mục tiêu

$$L(x, y, w_h, w_o) = \sum_{t=1}^T l(y_t, o_t). \quad (10.7.2)$$

Đối với lan truyền ngược, mọi thứ lại phức tạp hơn một chút, đặc biệt là khi chúng ta tính gradient theo các tham số  $w_h$  của hàm mục tiêu  $L$ . Cụ thể, theo quy tắc dây chuyền ta có

$$\begin{aligned} \partial_{w_h} L &= \sum_{t=1}^T \partial_{w_h} l(y_t, o_t) \\ &= \sum_{t=1}^T \partial_{o_t} l(y_t, o_t) \partial_{h_t} g(h_t, w_h) [\partial_{w_h} h_t]. \end{aligned} \quad (10.7.3)$$

Ta có thể tính phần đầu tiên và phần thứ hai của đạo hàm một cách dễ dàng. Phần thứ ba  $\partial_{w_h} h_t$  khiến mọi thứ trở nên khó khăn, vì chúng ta cần phải tính toán ảnh hưởng của các tham số tới  $h_t$ .

Để tính được gradient ở trên, giả sử rằng chúng ta có ba chuỗi  $\{a_t\}$ ,  $\{b_t\}$ ,  $\{c_t\}$  thỏa mãn  $a_0 = 0$ ,  $a_1 = b_1$  và  $a_t = b_t + c_t a_{t-1}$  với  $t = 1, 2, \dots$ . Sau đó, với  $t \geq 1$  ta có

$$a_t = b_t + \sum_{i=1}^{t-1} \left( \prod_{j=i+1}^t c_j \right) b_i. \quad (10.7.4)$$

Bây giờ chúng ta áp dụng (10.7.4) với

$$a_t = \partial_{w_h} h_t, \quad (10.7.5)$$

$$b_t = \partial_{w_h} f(x_t, h_{t-1}, w_h), \quad (10.7.6)$$

$$c_t = \partial_{h_{t-1}} f(x_t, h_{t-1}, w_h). \quad (10.7.7)$$

Vì vậy, công thức  $a_t = b_t + c_t a_{t-1}$  trở thành phép đệ quy dưới đây

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w) + \partial_h f(x_t, h_{t-1}, w_h) \partial_{w_h} h_{t-1}. \quad (10.7.8)$$

Sử dụng (10.7.4), phần thứ ba sẽ trở thành

$$\partial_{w_h} h_t = \partial_{w_h} f(x_t, h_{t-1}, w_h) + \sum_{i=1}^{t-1} \left( \prod_{j=i+1}^t \partial_{h_{j-1}} f(x_j, h_{j-1}, w_h) \right) \partial_{w_h} f(x_i, h_{i-1}, w_h). \quad (10.7.9)$$

Dù chúng ta có thể sử dụng quy tắc dây chuyền để tính  $\partial_{w_h} h_t$  một cách đệ quy, dây chuyền này có thể trở nên rất dài khi giá trị  $t$  lớn. Hãy cùng thảo luận về một số chiến lược để giải quyết vấn đề này.

- **Tính toàn bộ tổng.** Cách này rất chậm và gradient có thể bùng nổ vì những thay đổi nhỏ trong các điều kiện ban đầu cũng có khả năng ảnh hưởng đến kết quả rất nhiều. Điều này tương tự như trong hiệu ứng cánh bướm, khi những thay đổi rất nhỏ trong điều kiện ban đầu dẫn đến những thay đổi không cân xứng trong kết quả. Đây thực sự là điều không mong muốn khi xét tới mô hình mà chúng ta muốn ước lượng. Sau cùng, chúng ta đang cố tìm kiếm một bộ ước lượng mạnh mẽ và có khả năng khái quát tốt. Do đó chiến lược này hầu như không bao giờ được sử dụng trong thực tế.
- **Cắt xén tổng sau  $\tau$  bước.** Cho đến giây phút hiện tại, đây là những gì chúng ta đã thảo luận. Điều này dẫn tới một phép *xấp xỉ* của gradient, đơn giản bằng cách kết thúc tổng trên tại  $\partial_{w_h} h_{t-\tau}$ . Do đó lỗi xấp xỉ là  $\partial_h f(x_t, h_{t-1}, w) \partial_{w_h} h_{t-1}$  (nhân với tích của gradient liên quan đến  $\partial_h f$ ). Trong thực tế, chiến lược này hoạt động khá tốt. Phương pháp này thường được gọi là BPTT (*backpropagation through time* – lan truyền ngược qua thời gian) bị cắt xén. Một trong những hệ quả của phương pháp này là mô hình sẽ tập trung chủ yếu vào ảnh hưởng ngắn hạn thay vì dài hạn. Đây thực sự là điều mà chúng ta *mong muốn*, vì nó hướng sự ước lượng tới các mô hình đơn giản và ổn định hơn.
- **Cắt xén Ngẫu nhiên.** Cuối cùng, chúng ta có thể thay thế  $\partial_{w_h} h_t$  bằng một biến ngẫu nhiên có giá trị kỳ vọng đúng nhưng vẫn cắt xén chuỗi.
- Điều này có thể đạt được bằng cách sử dụng một chuỗi các  $\xi_t$  trong đó  $E[\xi_t] = 1$ ,  $P(\xi_t = 0) = 1 - \pi$  và  $P(\xi_t = \pi^{-1}) = \pi$ .
- Chúng ta sẽ sử dụng chúng thay vì gradient:

$$z_t = \partial_w f(x_t, h_{t-1}, w) + \xi_t \partial_h f(x_t, h_{t-1}, w) \partial_w h_{t-1}. \quad (10.7.10)$$

Từ định nghĩa của  $\xi_t$ , ta có  $E[z_t] = \partial_w h_t$ . Bất cứ khi nào  $\xi_t = 0$ , khai triển sẽ kết thúc tại điểm đó. Điều này dẫn đến một tổng trọng số của các chuỗi có chiều dài biến thiên, trong đó chuỗi dài sẽ hiếm hơn nhưng được đánh trọng số cao hơn tương ứng. (Tallec & Ollivier, 2017) đưa ra đề xuất này trong bài báo nghiên cứu của họ. Không may, dù phương pháp này khá hấp dẫn về mặt lý thuyết, nó lại không tốt hơn phương pháp cắt xén đơn giản, nhiều khả năng do các yếu tố sau. Thứ nhất, tác động của một quan sát đến quá khứ sau một vài lượt lan truyền ngược đã là tương đối đủ để nắm bắt các phụ thuộc trên thực tế. Thứ hai, phương sai tăng lên làm phản tác dụng của việc có gradient chính xác hơn. Thứ ba, ta thực sự muốn các mô hình có khoảng tương tác ngắn. Do đó, BPTT có một hiệu ứng điều chỉnh nhỏ mà có thể có ích.

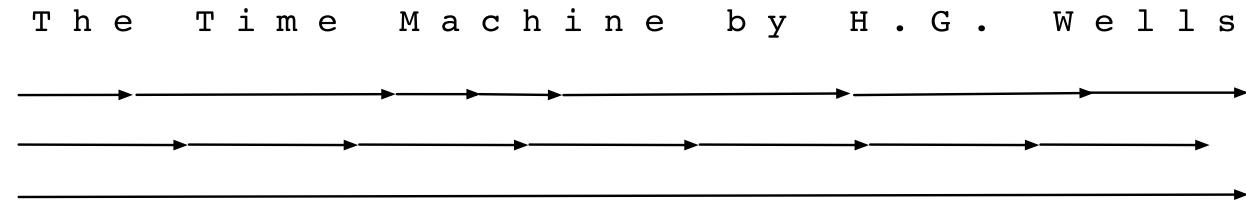


Fig. 10.7.1: Từ trên xuống dưới: BPTT ngẫu nhiên, BPTT bị cắt xén đều và BPTT đầy đủ

Fig. 10.7.1 minh họa ba trường hợp trên khi phân tích một số từ đầu tiên trong *Cỗ máy Thời gian*:

- Dòng đầu tiên biểu diễn sự cắt xén ngẫu nhiên, chia văn bản thành các phần có độ dài biến thiên.
- Dòng thứ hai biểu diễn BPTT bị cắt xén đều, chia văn bản thành các phần có độ dài bằng nhau.
- Dòng thứ ba là BPTT đầy đủ, dẫn đến một biểu thức không khả thi về mặt tính toán.

## 10.7.2 Đồ thị Tính toán

Để minh họa trực quan sự phụ thuộc giữa các biến và tham số mô hình trong suốt quá trình tính toán của mạng nơ-ron hồi tiếp, ta có thể vẽ đồ thị tính toán của mô hình, như trong Fig. 10.7.2. Ví dụ, việc tính toán trạng thái ẩn ở bước thời gian 3,  $\mathbf{h}_3$ , phụ thuộc vào các tham số  $\mathbf{W}_{hx}$  và  $\mathbf{W}_{hh}$  của mô hình, trạng thái ẩn ở bước thời gian trước đó  $\mathbf{h}_2$ , và đầu vào ở bước thời gian hiện tại  $\mathbf{x}_3$ .

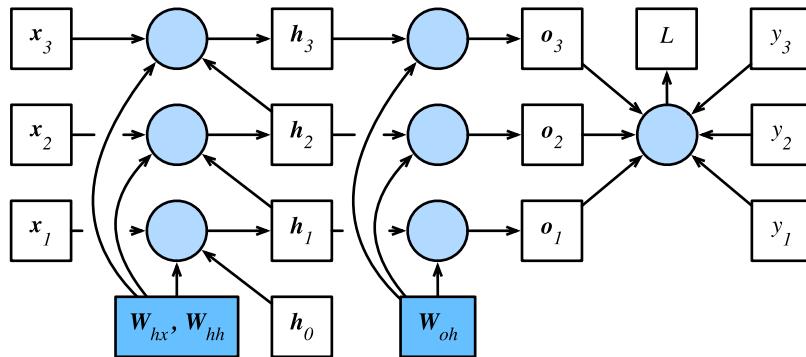


Fig. 10.7.2: Sự phụ thuộc về mặt tính toán của mạng nơ-ron hồi tiếp với ba bước thời gian. Ô vuông tượng trưng cho các biến (không tô đậm) hoặc các tham số (tô đậm), hình tròn tượng trưng cho các phép toán.

### 10.7.3 BPTT chi tiết

Sau khi thảo luận các nguyên lý chung, hãy phân tích BPTT một cách chi tiết. Bằng cách tách  $\mathbf{W}$  thành các tập ma trận trọng số khác nhau  $\mathbf{W}_{hx}$ ,  $\mathbf{W}_{hh}$  và  $\mathbf{W}_{oh}$ ), ta thu được mô hình biến tiệm ẩn tuyến tính đơn giản:

$$\mathbf{h}_t = \mathbf{W}_{hx} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{h}_{t-1} \text{ và } \mathbf{o}_t = \mathbf{W}_{oh} \mathbf{h}_t. \quad (10.7.11)$$

Theo thảo luận ở Section 6.7, ta tính các gradient  $\frac{\partial L}{\partial \mathbf{W}_{hx}}$ ,  $\frac{\partial L}{\partial \mathbf{W}_{hh}}$ ,  $\frac{\partial L}{\partial \mathbf{W}_{oh}}$  cho

$$L(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{t=1}^T l(\mathbf{o}_t, y_t), \quad (10.7.12)$$

với  $l(\cdot)$  là hàm mất mát đã chọn trước. Tính đạo hàm theo  $\mathbf{W}_{oh}$  khá đơn giản, ta có

$$\partial_{\mathbf{W}_{oh}} L = \sum_{t=1}^T \text{prod} (\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{h}_t), \quad (10.7.13)$$

với  $\text{prod}(\cdot)$  là tích của hai hoặc nhiều ma trận.

Sự phụ thuộc vào  $\mathbf{W}_{hx}$  và  $\mathbf{W}_{hh}$  thì khó khăn hơn một chút vì cần sử dụng quy tắc dây chuyền khi tính toán đạo hàm. Ta sẽ bắt đầu với

$$\begin{aligned} \partial_{\mathbf{W}_{hh}} L &= \sum_{t=1}^T \text{prod} (\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hh}} \mathbf{h}_t), \\ \partial_{\mathbf{W}_{hx}} L &= \sum_{t=1}^T \text{prod} (\partial_{\mathbf{o}_t} l(\mathbf{o}_t, y_t), \mathbf{W}_{oh}, \partial_{\mathbf{W}_{hx}} \mathbf{h}_t). \end{aligned} \quad (10.7.14)$$

Sau cùng, các trạng thái ẩn phụ thuộc lẫn nhau và phụ thuộc vào đầu vào quá khứ. Một đại lượng quan trọng là sự ảnh hưởng của các trạng thái ẩn quá khứ tới các trạng thái ẩn tương lai.

$$\partial_{\mathbf{h}_t} \mathbf{h}_{t+1} = \mathbf{W}_{hh}^\top \text{ do đó } \partial_{\mathbf{h}_t} \mathbf{h}_T = \left( \mathbf{W}_{hh}^\top \right)^{T-t}. \quad (10.7.15)$$

Áp dụng quy tắc dây chuyền ta được

$$\begin{aligned} \partial_{\mathbf{W}_{hh}} \mathbf{h}_t &= \sum_{j=1}^t \left( \mathbf{W}_{hh}^\top \right)^{t-j} \mathbf{h}_j \\ \partial_{\mathbf{W}_{hx}} \mathbf{h}_t &= \sum_{j=1}^t \left( \mathbf{W}_{hh}^\top \right)^{t-j} \mathbf{x}_j. \end{aligned} \quad (10.7.16)$$

Ta có thể rút ra nhiều điều từ biểu thức phức tạp này. Đầu tiên, việc lưu lại các kết quả trung gian, tức các luỹ thừa của  $\mathbf{W}_{hh}$  khi tính các số hạng của hàm mất mát  $L$ , là rất hữu ích. Thứ hai, ví dụ tuyến tính này dù đơn giản nhưng đã làm lộ ra một vấn đề chủ chốt của các mô hình chuỗi dài: ta có thể phải làm việc với các luỹ thừa rất lớn của  $\mathbf{W}_{hh}^j$ . Trong đó, khi  $j$  lớn, các trị riêng nhỏ hơn 1 sẽ tiêu biến, còn các trị riêng lớn hơn 1 sẽ phân kì. Các mô hình này không có tính ổn định số học, dẫn đến việc chúng quan trọng hóa quá mức các chi tiết không liên quan trong quá khứ. Một cách giải quyết vấn đề này là cắt xén các số hạng trong tổng ở một mức độ thuận tiện cho việc tính toán. Sau này ở Section 11, ta sẽ thấy cách các mô hình chuỗi phức tạp như LSTM giải quyết vấn đề này tốt hơn. Khi lập trình, ta cắt xén các số hạng bằng cách *tách rời* gradient sau một số lượng bước nhất định.

#### 10.7.4 Tóm tắt

- Lan truyền ngược theo thời gian chỉ là việc áp dụng lan truyền ngược cho các mô hình chuỗi có trạng thái ẩn.
- Việc cắt xén là cần thiết để thuận tiện cho việc tính toán và ổn định các giá trị số.
- Luỹ thừa lớn của ma trận có thể làm các trị riêng tiêu biến hoặc phân kì, biểu hiện dưới hiện tượng tiêu biến hoặc bùng nổ gradient.
- Để tăng hiệu năng tính toán, các giá trị trung gian được lưu lại.

#### 10.7.5 Bài tập

1. Cho ma trận đối xứng  $\mathbf{M} \in \mathbb{R}^{n \times n}$  với các trị riêng  $\lambda_i$ . Không làm mất tính tổng quát, ta giả sử chúng được sắp xếp theo thứ tự tăng dần  $\lambda_i \leq \lambda_{i+1}$ . Chứng minh rằng  $\mathbf{M}^k$  có các trị riêng là  $\lambda_i^k$ .
2. Chứng minh rằng với vector bất kì  $\mathbf{x} \in \mathbb{R}^n$ , xác suất cao là  $\mathbf{M}^k \mathbf{x}$  sẽ xấp xỉ vector trị riêng lớn nhất  $\mathbf{v}_n$  của  $\mathbf{M}$ .
3. Kết quả trên có ý nghĩa như thế nào khi tính gradient của mạng nơ-ron hồi tiếp?
4. Ngoài gọt gradient, có phương pháp nào để xử lý bùng nổ gradient trong mạng nơ-ron hồi tiếp không?

#### 10.7.6 Thảo luận

- Tiếng Anh<sup>200</sup>
- Tiếng Việt<sup>201</sup>

#### 10.7.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Minh Đức
- Phạm Hồng Vinh

<sup>200</sup> <https://discuss.mxnet.io/t/2366>

<sup>201</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 10.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Văn Cường

# 11 | Mạng Nơ-ron Hồi tiếp Hiện đại

Mặc dù đã biết về các kiến thức cơ bản của mạng nơ-ron hồi tiếp, chúng vẫn chưa đủ để ta giải quyết các bài toán học chuỗi hiện nay. Ví dụ như RNN có hiện tượng bất ổn số học khi tính gradient, do đó các mạng nơ-ron hồi tiếp có cổng được sử dụng phổ biến hơn nhiều trong thực tiễn. Chúng ta bắt đầu chương này bằng việc giới thiệu hai cấu trúc mạng phổ biến: nút hồi tiếp có cổng (*gated recurrent unit - GRU*) và bộ nhớ ngắn hạn dài (*long short term memory - LSTM*). Chúng cũng sẽ được áp dụng minh họa trong cùng bài toán mô hình hóa ngôn ngữ đã được giới thiệu ở [Section 10](#).

Hơn nữa, chúng ta sẽ sửa đổi mạng nơ-ron hồi tiếp với một tầng ẩn đơn chiều. Ta cũng sẽ mô tả các kiến trúc mạng sâu và thảo luận về thiết kế hai chiều (*bidirectional*) gồm cả hồi tiếp xuôi và ngược. Chúng thường xuyên được sử dụng trong các mạng nơ-ron hồi tiếp hiện đại.

Trên thực tế, phần lớn các bài toán học chuỗi như nhận dạng giọng nói tự động, chuyển đổi văn bản thành giọng nói và dịch máy, đều có đầu vào và đầu ra là các chuỗi với chiều dài bất kỳ. Cuối cùng, ta sẽ lấy bài toán dịch máy làm ví dụ để giới thiệu kiến trúc mã hóa - giải mã (*encoder-decoder*) dựa trên mạng nơ-ron hồi tiếp cùng các kỹ thuật hiện đại để giải quyết bài toán học từ chuỗi sang chuỗi.

## 11.1 Nút Hồi tiếp có Cổng (GRU)

Trong phần trước, chúng ta đã thảo luận cách tính gradient trong mạng nơ-ron hồi tiếp. Cụ thể ta đã biết rằng tích của một chuỗi dài các ma trận có thể dẫn đến việc gradient tiêu biến hoặc bùng nổ. Hãy điểm qua các tình huống thực tế thể hiện rõ hai bất thường đó:

- Ta có thể gặp tình huống mà những quan sát xuất hiện sớm có ảnh hưởng lớn đến việc dự đoán toàn bộ những quan sát trong tương lai. Xét một ví dụ có chút cường điệu, trong đó quan sát đầu tiên chứa giá trị tổng kiểm (*checksum*) và mục tiêu là kiểm tra xem liệu giá trị tổng kiểm đó có đúng hay không tại cuối chuỗi. Trong trường hợp này, ảnh hưởng của token đầu tiên là tối quan trọng. Do đó ta muốn có cơ chế để lưu trữ những thông tin quan trọng ban đầu trong ô nhớ. Nếu không, ta sẽ phải gán một giá trị gradient cực lớn cho quan sát ban đầu vì nó ảnh hưởng đến toàn bộ các quan sát tiếp theo.
- Một tình huống khác là khi một vài ký hiệu không chứa thông tin phù hợp. Ví dụ, khi phân tích một trang web, ta có thể gặp các mã HTML không giúp ích gì cho việc xác định thông tin được truyền tải. Do đó, ta cũng muốn có cơ chế để bỏ qua những ký hiệu như vậy trong các biểu diễn trạng thái tiềm ẩn.
- Ta cũng có thể gặp những khoảng ngắt giữa các phần trong một chuỗi. Ví dụ như những phần chuyển tiếp giữa các chương của một quyển sách, hay chuyển biến xu hướng giữa thị trường giá lên và thị trường giá xuống trong chứng khoán. Trong trường hợp này, sẽ tốt hơn nếu có một cách để *xóa* hay *đặt lại* các biểu diễn trạng thái ẩn về giá trị ban đầu.

Nhiều phương pháp đã được đề xuất để giải quyết những vấn đề trên. Một trong những phương pháp ra đời sớm nhất là Bộ nhớ ngắn hạn dài (*Long Short Term Memory - LSTM*) (Hochreiter & Schmidhuber, 1997), sẽ được thảo luận ở [Section 11.2](#). Nút Hồi tiếp có Cổng (Gated Recurrent Unit - *GRU*) (Cho et al., 2014) là một biến thể gọn hơn của LSTM, thường có chất lượng tương đương và tính toán nhanh hơn đáng kể. Tham khảo (Chung et al., 2014) để biết thêm chi tiết. Trong chương này, ta sẽ bắt đầu với GRU do nó đơn giản hơn.

### 11.1.1 Kiểm soát Trạng thái Ẩn

Sự khác biệt chính giữa RNN thông thường và GRU là GRU hỗ trợ việc kiểm soát trạng thái ẩn. Điều này có nghĩa là ta có các cơ chế được học để quyết định khi nào nên cập nhật và khi nào nên xóa trạng thái ẩn. Ví dụ, nếu ký tự đầu tiên có mức độ quan trọng cao, mô hình sẽ học để không cập nhật trạng thái ẩn sau lần quan sát đầu tiên. Tương tự, mô hình sẽ học cách bỏ qua những quan sát tạm thời không liên quan, cũng như cách xóa trạng thái ẩn khi cần thiết. Dưới đây ta sẽ thảo luận chi tiết vấn đề này.

#### Cổng Xóa và Cổng Cập nhật

Đầu tiên ta giới thiệu cổng xóa và cổng cập nhật. Ta thiết kế chúng thành các vector có các phần tử trong khoảng  $(0, 1)$  để có thể biểu diễn các tổ hợp lồi. Chẳng hạn, một biến xóa cho phép kiểm soát bao nhiêu phần của trạng thái trước đây được giữ lại. Tương tự, một biến cập nhật cho phép kiểm soát bao nhiêu phần của trạng thái mới sẽ giống trạng thái cũ.

Ta bắt đầu bằng việc thiết kế các cổng tạo ra các biến này. [Fig. 11.1.1](#) minh họa các đầu vào cho cả cổng xóa và cổng cập nhật trong GRU, với đầu vào ở bước thời gian hiện tại  $\mathbf{X}_t$  và trạng thái ẩn ở bước thời gian trước đó  $\mathbf{H}_{t-1}$ . Đầu ra được tạo bởi một tầng kết nối đầy đủ với hàm kích hoạt sigmoid.

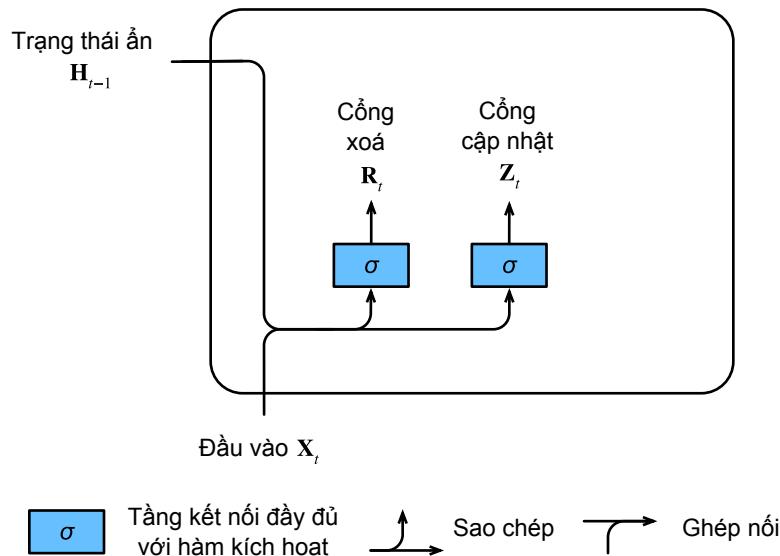


Fig. 11.1.1: Cổng xóa và cổng cập nhật trong GRU.

Tại bước thời gian  $t$ , với đầu vào minibatch là  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (số lượng mẫu:  $n$ , số lượng đầu vào:  $d$ ) và trạng thái ẩn ở bước thời gian gần nhất là  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$  (số lượng trạng thái ẩn:  $h$ ), cổng xóa

$\mathbf{R}_t \in \mathbb{R}^{n \times h}$  và cỗng cập nhật  $\mathbf{Z}_t \in \mathbb{R}^{n \times h}$  được tính như sau:

$$\begin{aligned}\mathbf{R}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r), \\ \mathbf{Z}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z).\end{aligned}\quad (11.1.1)$$

Ở đây,  $\mathbf{W}_{xr}, \mathbf{W}_{xz} \in \mathbb{R}^{d \times h}$  và  $\mathbf{W}_{hr}, \mathbf{W}_{hz} \in \mathbb{R}^{h \times h}$  là các tham số trọng số và  $\mathbf{b}_r, \mathbf{b}_z \in \mathbb{R}^{1 \times h}$  là các hệ số điều chỉnh. Ta sẽ sử dụng hàm sigmoid (như trong Section 6.1) để biến đổi các giá trị đầu vào nằm trong khoảng  $(0, 1)$ .

### Hoạt động của Cổng Xóa

Ta bắt đầu bằng việc tích hợp cổng xóa với một cơ chế cập nhật trạng thái tiềm ẩn thông thường. Trong RNN thông thường, ta cập nhật trạng thái ẩn theo công thức

$$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h). \quad (11.1.2)$$

Điều này về cơ bản giống với những gì đã thảo luận ở phần trước, mặc dù có thêm tính phi tuyến dưới dạng hàm tanh để đảm bảo rằng các giá trị trạng thái ẩn nằm trong khoảng  $(-1, 1)$ . Nếu muốn giảm ảnh hưởng của các trạng thái trước đó, ta có thể nhân  $\mathbf{H}_{t-1}$  với  $\mathbf{R}_t$  theo từng phần tử. Nếu các phần tử trong cổng xóa  $\mathbf{R}_t$  có giá trị gần với 1, kết quả sẽ giống RNN thông thường. Nếu tất cả các phần tử của cổng xóa  $\mathbf{R}_t$  gần với 0, trạng thái ẩn sẽ là đầu ra của một perceptron đa tầng với đầu vào là  $\mathbf{X}_t$ . Bất kỳ trạng thái ẩn nào tồn tại trước đó đều được đặt lại về giá trị mặc định. Tại đây nó được gọi là *trạng thái ẩn tiềm năng*, và chỉ là *tiềm năng* vì ta vẫn cần kết hợp thêm đầu ra của cổng cập nhật.

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h). \quad (11.1.3)$$

Fig. 11.1.2 minh họa luồng tính toán sau khi áp dụng cổng xóa. Ký hiệu  $\odot$  biểu thị phép nhân theo từng phần tử giữa các tensor.

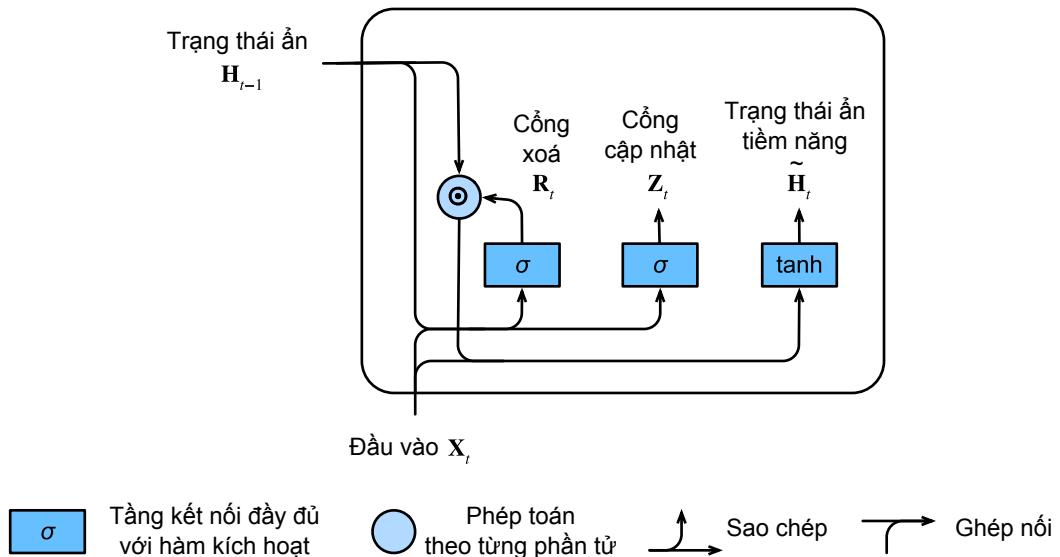


Fig. 11.1.2: Tính toán của trạng thái ẩn tiềm năng trong một GRU. Phép nhân được thực hiện theo phần tử.

## Hoạt động của Cổng Cập nhật

Tiếp theo ta sẽ kết hợp hiệu ứng của cổng cập nhật  $\mathbf{Z}_t$  như trong Fig. 11.1.3. Cổng này xác định mức độ giống nhau giữa trạng thái mới  $\mathbf{H}_t$  và trạng thái cũ  $\mathbf{H}_{t-1}$ , cũng như mức độ trạng thái ẩn tiềm năng  $\tilde{\mathbf{H}}_t$  được sử dụng. Biến cổng (*gating variable*)  $\mathbf{Z}_t$  được sử dụng cho mục đích này, bằng cách áp dụng tổ hợp lòi giữa trạng thái cũ và trạng thái tiềm năng. Ta có phương trình cập nhật cuối cùng cho GRU.

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t. \quad (11.1.4)$$

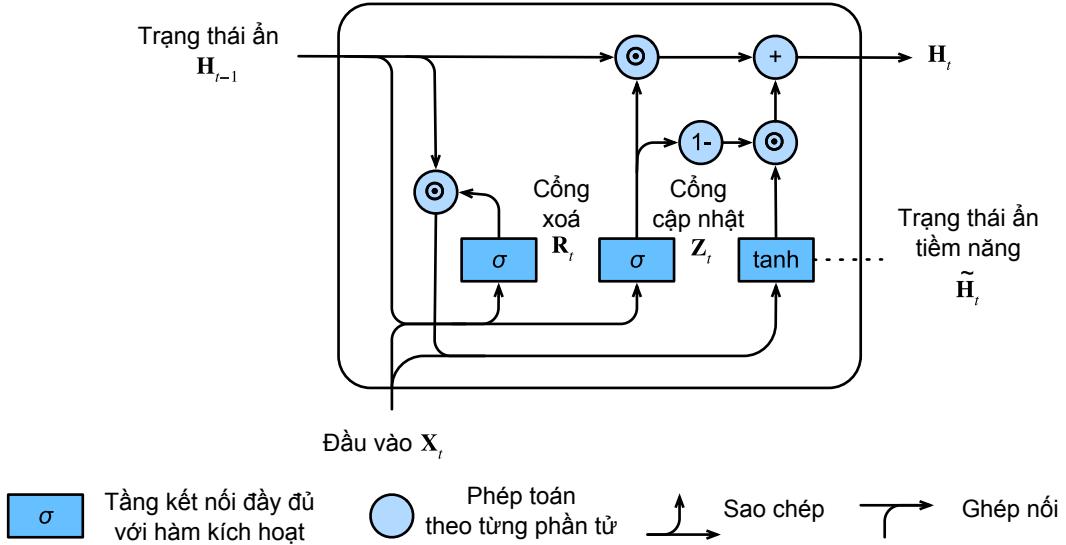


Fig. 11.1.3: Tính toán trạng thái ẩn trong GRU. Như trước đây, phép nhân được thực hiện theo từng phần tử.

Nếu các giá trị trong cổng cập nhật  $\mathbf{Z}_t$  bằng 1, chúng ta chỉ đơn giản giữ lại trạng thái cũ. Trong trường hợp này, thông tin từ  $\mathbf{X}_t$  về cơ bản được bỏ qua, tương đương với việc bỏ qua bước thời gian  $t$  trong chuỗi phụ thuộc. Ngược lại, nếu  $\mathbf{Z}_t$  gần giá trị 0, trạng thái ẩn  $\mathbf{H}_t$  sẽ gần với trạng thái ẩn tiềm năng  $\tilde{\mathbf{H}}_t$ . Những thiết kế trên có thể giúp chúng ta giải quyết vấn đề tiêu biến gradient trong các mạng RNN và nắm bắt tốt hơn sự phụ thuộc xa trong chuỗi thời gian. Tóm lại, các mạng GRU có hai tính chất nổi bật sau:

- Cổng xóa giúp nắm bắt các phụ thuộc ngắn hạn trong chuỗi thời gian.
- Cổng cập nhật giúp nắm bắt các phụ thuộc dài hạn trong chuỗi thời gian.

## 11.1.2 Lập trình từ đầu

Để hiểu rõ hơn, hãy lập trình mô hình GRU từ đầu.

## Đọc Dữ liệu

Chúng ta bắt đầu bằng việc đọc kho dữ liệu *Cỗ máy Thời gian* đã sử dụng trong Section 10.5. Dưới đây là mã nguồn đọc dữ liệu:

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

## Khởi tạo Tham số Mô hình

Bước tiếp theo là khởi tạo các tham số mô hình. Ta khởi tạo các giá trị trọng số theo phân phối Gauss với phương sai 0.01 và thiết lập các hệ số điều chỉnh bằng 0. Siêu tham số num\_hiddens xác định số lượng đơn vị ẩn. Ta khởi tạo tất cả các trọng số và các hệ số điều chỉnh của cổng cập nhật, cổng xóa, và các trạng thái ẩn tiềm năng. Sau đó, gắn gradient cho tất cả các tham số.

```
def get_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=ctx))

    W_xz, W_hz, b_z = three() # Update gate parameter
    W_xr, W_hr, b_r = three() # Reset gate parameter
    W_xh, W_hh, b_h = three() # Candidate hidden state parameter
    # Output layer parameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = np.zeros(num_outputs, ctx=ctx)
    # 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.attach_grad()
    return params
```

## Định nghĩa Mô hình

Bây giờ ta sẽ định nghĩa hàm khởi tạo trạng thái ẩn `init_gru_state`. Cũng giống như hàm `init_rnn_state` trong Section 10.5, hàm này trả về một mảng ndarray chứa các giá trị bằng không với kích thước (kích thước batch, số đơn vị ẩn).

```
def init_gru_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

Giờ ta có thể định nghĩa mô hình GRU. Cấu trúc GRU cũng giống một khối RNN cơ bản nhưng có 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 = npx.sigmoid(np.dot(X, W_xz) + np.dot(H, W_hz) + b_z)
        R = npx.sigmoid(np.dot(X, W_xr) + np.dot(H, W_hr) + b_r)
        H_tilda = np.tanh(np.dot(X, W_xh) + np.dot(R * H, W_hh) + b_h)
        H = Z * H + (1 - Z) * H_tilda
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H,)
```

## Huấn luyện và Dự đoán

Việc huấn luyện và dự đoán cũng được thực hiện tương tự như với RNN. Sau khi huấn luyện một epoch, ta thu được perplexity và câu đầu ra như sau.

```
vocab_size, num_hiddens, ctx = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, ctx, get_params,
                             init_gru_state, gru)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

### 11.1.3 Lập trình Súc tích

Trong Gluon, ta có thể trực tiếp gọi lớp GRU trong mô-đun `rnn`. Mô-đun này đóng gói tất cả các cấu hình đã thực hiện tường minh ở trên. Đoạn mã này nhanh hơn đáng kể do sử dụng các toán tử được biên dịch chứ không phải thuần Python như trên.

```
gru_layer = rnn.GRU(num_hiddens)
model = d2l.RNNModel(gru_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

#### 11.1.4 Tóm tắt

- Các mạng nơ-ron hồi tiếp có cổng nắm bắt các phụ thuộc xa trong chuỗi thời gian tốt hơn.
- Cổng xóa giúp nắm bắt phụ thuộc ngắn hạn trong chuỗi thời gian.
- Cổng cập nhật giúp nắm bắt các phụ thuộc dài hạn trong chuỗi thời gian.
- Trường hợp đặc biệt khi cổng xóa được kích hoạt, GRU trở thành RNN cơ bản. Chúng cũng có thể bỏ qua các thành phần trong chuỗi khi cần.

#### 11.1.5 Bài tập

1. Hãy so sánh thời gian chạy, perplexity và các chuỗi đầu ra của `rnn.RNN` và `rnn.GRU`.
2. Giả sử ta chỉ muốn sử dụng đầu vào tại bước thời gian  $t'$  để dự đoán đầu ra tại bước thời gian  $t > t'$ . Hãy xác định các giá trị tốt nhất cho cổng xóa và cổng cập nhật tại mỗi bước thời gian?
3. Quan sát và phân tích tác động tới thời gian chạy, perplexity và các câu được sinh ra khi điều chỉnh các siêu tham số.
4. Điều gì xảy ra khi GRU được lập trình chỉ có cổng xóa hay chỉ có cổng cập nhật?

#### 11.1.6 Thảo luận

- Tiếng Anh<sup>202</sup>
- Tiếng Việt<sup>203</sup>

#### 11.1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Võ Tấn Phát
- Lê Khắc Hồng Phúc
- Nguyễn Duy Du
- Nguyễn Văn Quang
- Phạm Minh Đức
- Phạm Hồng Vinh
- Nguyễn Cảnh Thưởng

<sup>202</sup> <https://discuss.mxnet.io/t/2367>

<sup>203</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 11.2 Bộ nhớ Ngắn hạn Dài (LSTM)

Thách thức đối với việc lưu trữ những 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 phương pháp tiếp cận sớm nhất để giải quyết vấn đề này là LSTM ([Hochreiter & Schmidhuber, 1997](#)). Nó có nhiều tính chất tương tự Nút Hồi tiếp có Cổng (GRU). Điều thú vị là thiết kế của LSTM chỉ phức tạp hơn GRU một chút nhưng đã xuất hiện trước GRU gần hai thập kỷ.

Có thể cho rằng thiết kế này được lấy cảm hứng từ các cổng logic trong máy tính. Để kiểm soát một ô nhớ chúng ta cần một số các cổng. Một cổng để đọc các thông tin từ ô nhớ đó (trái với việc đọc từ các ô khác). Chúng ta sẽ gọi cổng này là *cổng đầu ra (output gate)*. Một cổng thứ hai để quyết định khi nào cần ghi dữ liệu vào ô nhớ. Chúng ta gọi cổng này là *cổng đầu vào (input gate)*. Cuối cùng, chúng ta cần một cơ chế để thiết lập lại nội dung chứa trong ô nhớ, được chi phối bởi một *cổng quên (forget gate)*. Động lực của thiết kế trên cũng tương tự như trước đây, đó là để đưa ra quyết định khi nào cần nhớ và khi nào nên bỏ qua đầu vào trong trạng thái tiềm ẩn thông qua một cơ chế chuyên dụng. Chúng ta hãy xem thiết kế này hoạt động như thế nào trong thực tế.

### 11.2.1 Các Ô nhớ có Cổng

Ba cổng được giới thiệu trong LSTM đó là: cổng đầu vào, cổng quên và cổng đầu ra. Bên cạnh đó chúng ta sẽ giới thiệu một ô nhớ có kích thước giống với trạng thái ẩn. Nói đúng hơn đây chỉ là phiên bản đặc biệt của trạng thái ẩn, được thiết kế để ghi lại các thông tin bổ sung.

#### Cổng Đầu vào, Cổng Quên và Cổng Đầu ra

Tương tự như với GRU, dữ liệu được đưa vào các cổng LSTM là đầu vào ở bước thời gian hiện tại  $\mathbf{X}_t$  và trạng thái ẩn ở bước thời gian trước đó  $\mathbf{H}_{t-1}$ . Những đầu vào này được xử lý bởi một tầng kết nối đầy đủ và một hàm kích hoạt sigmoid để tính toán các giá trị của các cổng đầu vào, cổng quên và cổng đầu ra. Kết quả là, tất cả các giá trị đầu ra tại ba cổng đều nằm trong khoảng [0, 1]. [Fig. 11.2.1](#) minh họa luồng dữ liệu cho các cổng đầu vào, cổng quên, và cổng đầu ra.

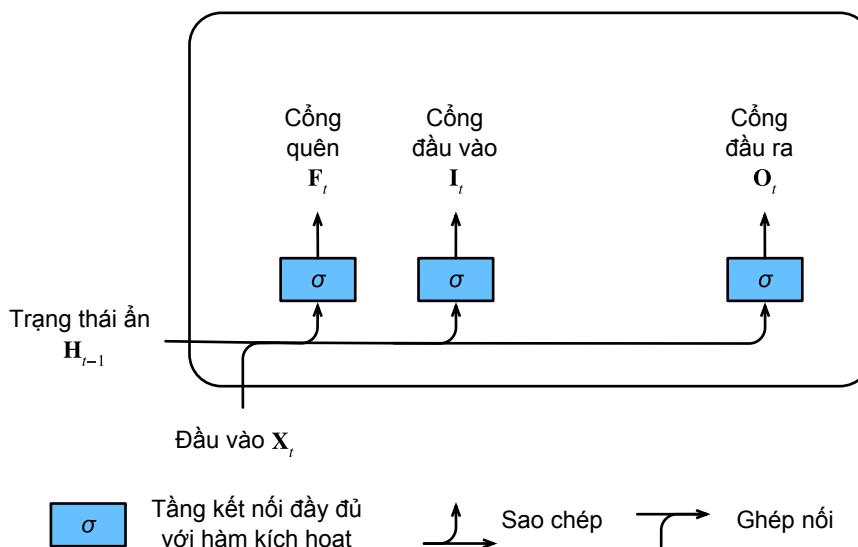


Fig. 11.2.1: Các phép tính tại cổng đầu vào, cổng quên và cổng đầu ra trong một đơn vị LSTM.

Chúng ta giả sử rằng có  $h$  nút ẩn, mỗi minibatch có kích thước  $n$  và kích thước đầu vào là  $d$ . Như vậy, đầu vào là  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  và trạng thái ẩn của bước thời gian trước đó là  $\mathbf{H}_{t-1} \in \mathbb{R}^{n \times h}$ . Tương tự, các cổng 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:

$$\begin{aligned}\mathbf{I}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i), \\ \mathbf{F}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xf} + \mathbf{H}_{t-1} \mathbf{W}_{hf} + \mathbf{b}_f), \\ \mathbf{O}_t &= \sigma(\mathbf{X}_t \mathbf{W}_{xo} + \mathbf{H}_{t-1} \mathbf{W}_{ho} + \mathbf{b}_o),\end{aligned}\tag{11.2.1}$$

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 trọng số và  $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$  là các hệ số điều chỉnh.

### Ô nhớ Tiềm năng

Tiếp theo, chúng ta sẽ thiết kế một ô nhớ. Vì ta vẫn chưa chỉ định tác động của các cổng khác nhau, nên đầu tiên ta sẽ giới thiệu ô nhớ *tiềm năng*  $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$ . Các phép tính toán cũng tương tự như ba cổng mô tả ở trên, ngoài trừ việc ở đây ta sử dụng hàm kích hoạt tanh với miền giá trị nằm trong khoảng  $[-1, 1]$ . Điều này dẫn đến phương trình sau tại bước thời gian  $t$ .

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c).\tag{11.2.2}$$

Ở đây  $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$  và  $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$  là các tham số trọng số và  $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$  là một hệ số điều chỉnh.

Ô nhớ tiềm năng được mô tả ngắn gọn trong Fig. 11.2.2.

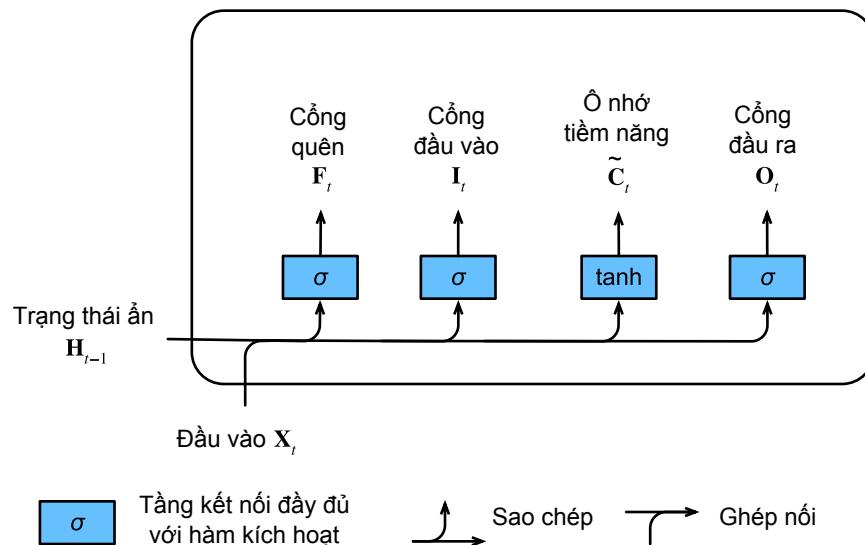


Fig. 11.2.2: Các phép tính toán trong ô nhớ tiềm năng của LSTM.

## Ô nhớ

Trong GRU, chúng ta chỉ có một cơ chế duy nhất để quản lý cả việc nhớ và quên. Trong LSTM, chúng ta có hai tham số,  $\mathbf{I}_t$  điều chỉnh lượng dữ liệu mới được lấy vào thông qua  $\tilde{\mathbf{C}}_t$  và tham số quên  $\mathbf{F}_t$  chỉ định lượng thông tin cũ cần giữ lại trong ô nhớ  $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ . Sử dụng cùng một phép nhân theo từng điểm (*pointwise*) như trước đây, chúng ta đi đến phương trình cập nhật như sau.

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t. \quad (11.2.3)$$

Nếu giá trị ở cổng quên luôn xấp xỉ bằng 1 và cổng đầu vào luôn xấp xỉ bằng 0, thì giá trị ô nhớ trong quá khứ  $\mathbf{C}_{t-1}$  sẽ được lưu lại qua thời gian và truyền tới bước thời gian hiện tại. Thiết kế này được giới thiệu nhằm giảm bớt vấn đề tiêu biến gradient cũng như nắm bắt các phụ thuộc dài hạn trong chuỗi thời gian tốt hơn. Do đó chúng ta có sơ đồ luồng trong Fig. 11.2.3.

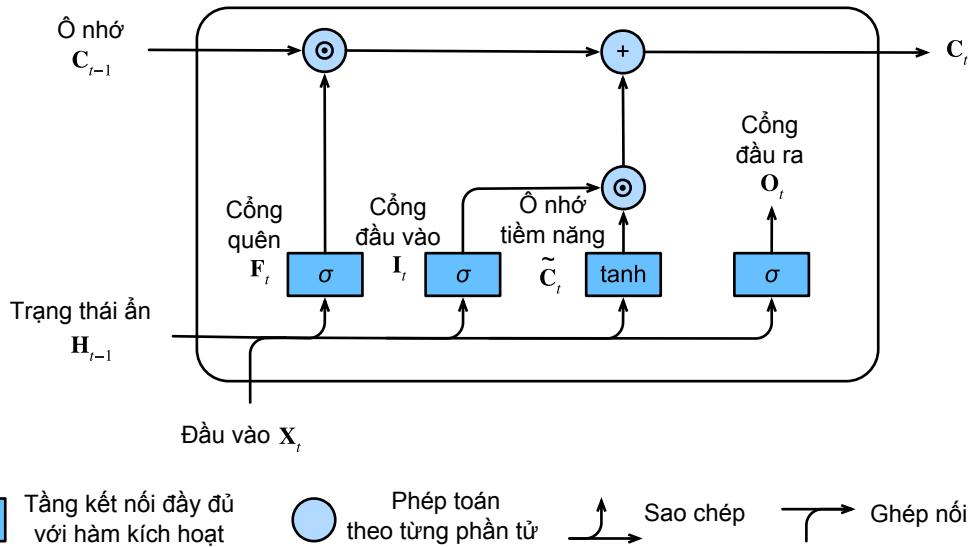


Fig. 11.2.3: Các phép tính toán trong ô nhớ của LSTM. Ở đây, ta sử dụng phép nhân theo từng phần tử.

## Các Trạng thái Ẩn

Cuối cùng, chúng ta cần phải xác định cách tính trạng thái ẩn  $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ . Đây là nơi cống đầu ra được sử dụng. Trong LSTM, đây chỉ đơn giản là một phiên bản có kiểm soát của hàm kích hoạt tanh trong ô nhớ. Điều này đảm bảo rằng các giá trị của  $\mathbf{H}_t$  luôn nằm trong khoảng  $(-1, 1)$ . Bất cứ khi nào giá trị của cống đầu ra là 1, thực chất chúng ta đang đưa toàn bộ thông tin trong ô nhớ tới bộ dự đoán. Ngược lại, khi giá trị của cống đầu ra là 0, chúng ta giữ lại tất cả các thông tin trong ô nhớ và không xử lý gì thêm. Fig. 11.2.4 minh họa các luồng dữ liệu.

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t). \quad (11.2.4)$$

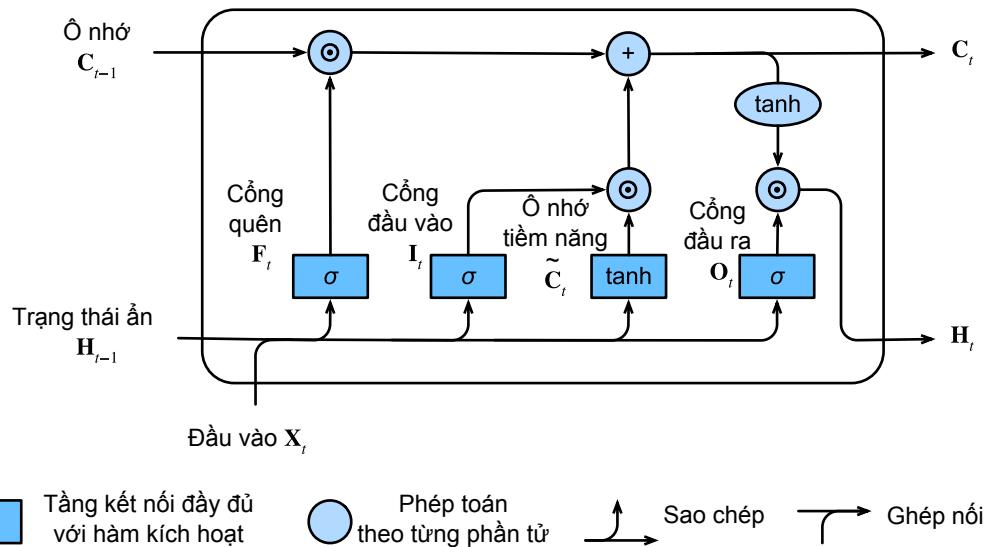


Fig. 11.2.4: Các phép tính của trạng thái ẩn. Phép tính nhân được thực hiện trên từng phần tử.

### 11.2.2 Lập trình Từ đầu

Bây giờ ta sẽ lập trình một LSTM từ đầu. Giống như các thử nghiệm trong các phần trước, đầu tiên ta sẽ nạp tập dữ liệu *The Time Machine*.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

#### Khởi tạo Tham số Mô hình

Tiếp theo ta cần định nghĩa và khởi tạo các tham số mô hình. Cũng giống như trước đây, siêu tham số `num_hiddens` định nghĩa số lượng các nút ẩn. Ta sẽ khởi tạo các trọng số theo phân phối Gauss với độ lệch chuẩn bằng 0.01 và đặt các hệ số điều chỉnh bằng 0.

```
def get_lstm_params(vocab_size, num_hiddens, ctx):
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return np.random.normal(scale=0.01, size=shape, ctx=ctx)

    def three():
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                np.zeros(num_hiddens, ctx=ctx))

    W_xi, W_hi, b_i = three() # Input gate parameters
    W_xf, W_hf, b_f = three() # Forget gate parameters
```

(continues on next page)

```

W_xo, W_ho, b_o = three() # Output gate parameters
W_xc, W_hc, b_c = three() # Candidate cell parameters
# Output layer parameters
W_hq = normal((num_hiddens, num_outputs))
b_q = np.zeros(num_outputs, ctx=ctx)
# 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.attach_grad()
return params

```

## Định nghĩa Mô hình

Trong hàm khởi tạo, trạng thái ẩn của LSTM cần trả về thêm một ô nhớ có giá trị bằng 0 và kích thước bằng (kích thước batch, số lượng các nút ẩn). Do đó ta có hàm khởi tạo trạng thái sau đây.

```

def init_lstm_state(batch_size, num_hiddens, ctx):
    return (np.zeros(shape=(batch_size, num_hiddens), ctx=ctx),
            np.zeros(shape=(batch_size, num_hiddens), ctx=ctx))

```

Mô hình thực sự được định nghĩa giống như những gì ta đã thảo luận trước đây: nó có ba cổng và một ô nhớ phụ. Lưu ý rằng chỉ có trạng thái ẩn được truyền tới tầng đầu ra. Các ô nhớ  $C_t$  không tham gia trực tiếp 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 = npx.sigmoid(np.dot(X, W_xi) + np.dot(H, W_hi) + b_i)
        F = npx.sigmoid(np.dot(X, W_xf) + np.dot(H, W_hf) + b_f)
        O = npx.sigmoid(np.dot(X, W_xo) + np.dot(H, W_ho) + b_o)
        C_tilda = np.tanh(np.dot(X, W_xc) + np.dot(H, W_hc) + b_c)
        C = F * C + I * C_tilda
        H = O * np.tanh(C)
        Y = np.dot(H, W_hq) + b_q
        outputs.append(Y)
    return np.concatenate(outputs, axis=0), (H, C)

```

## Huấn luyện và Dự đoán

Như đã từng làm trong Section 11.1, ta sẽ huấn luyện một LSTM bằng cách gọi hàm `RNNModelScratch` đã được giới thiệu ở Section 10.5.

```

vocab_size, num_hiddens, ctx = len(vocab), 256, d2l.try_gpu()
num_epochs, lr = 500, 1
model = d2l.RNNModelScratch(len(vocab), num_hiddens, ctx, get_lstm_params,

```

(continues on next page)

```
init_lstm_state, lstm)
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

### 11.2.3 Lập trình Súc tích

Trong Gluon, ta có thể gọi trực tiếp lớp LSTM trong mô-đun `rnn`. Lớp này gói gọn tất cả các chi tiết cấu hình mà ta đã lập trình một cách chi tiết ở trên. Mã nguồn sẽ chạy nhanh hơn đáng kể vì nó sử dụng các toán tử đã được biên dịch thay vì các toán tử Python cho nhiều phép tính mà ta đã lập trình trước đây.

```
lstm_layer = rnn.LSTM(num_hiddens)
model = d2l.RNNModel(lstm_layer, len(vocab))
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

Trong nhiều trường hợp, các mô hình LSTM hoạt động tốt hơn một chút so với các mô hình GRU nhưng việc huấn luyện và thực thi các mô hình này khá là tốn kém do chúng có kích thước trạng thái tiềm ẩn lớn hơn. LSTM là nguyên mẫu điển hình của một mô hình tự hồi quy biến tiềm ẩn có cơ chế kiểm soát trạng thái phức tạp. Nhiều biến thể đã được đề xuất qua từng năm, ví dụ như các kiến trúc đa tầng, các kết nối phần dư hay các kiểu điều chỉnh khác nhau. Tuy nhiên, việc huấn luyện LSTM và các mô hình chuỗi khác (như GRU) khá là tốn kém do sự phụ thuộc dài hạn của chuỗi. Sau này ta có thể sử dụng các mô hình khác như *Transformer* để song song hoá việc huấn luyện chuỗi.

### 11.2.4 Tóm tắt

- LSTM có ba loại cổng để kiểm soát luồng thông tin: cổng đầu vào, cổng quên và cổng đầu ra.
- Đầu ra tầng ẩn của LSTM bao gồm các trạng thái ẩn và các ô nhớ. Chỉ các trạng thái ẩn là được truyền tới tầng đầu ra. Các ô nhớ hoàn toàn được sử dụng nội bộ trong tầng.
- LSTM có thể đối phó với vấn đề tiêu biến và bùng nổ gradient.

### 11.2.5 Bài tập

1. Thay đổi các siêu tham số. Quan sát và phân tích tác động đến thời gian chạy, perplexity và đầu ra.
2. Cần thay đổi mô hình như thế nào để sinh ra các từ hoàn chỉnh thay vì các chuỗi ký tự?
3. So sánh chi phí tính toán của GRU, LSTM và RNN thông thường với cùng một chiều ẩn. Đặc biệt chú ý đến chi phí huấn luyện và dự đoán.
4. Dù các ô nhớ tiềm năng đã đảm bảo rằng phạm vi giá trị nằm trong khoảng từ  $-1$  đến  $1$  bằng cách sử dụng hàm tanh, tại sao trạng thái ẩn vẫn phải sử dụng tiếp hàm tanh để đảm bảo rằng phạm vi giá trị đầu ra nằm trong khoảng từ  $-1$  đến  $1$ ?
5. Lập trình một mô hình LSTM để dự đoán chuỗi thời gian thay vì dự đoán chuỗi ký tự.

### 11.2.6 Thảo luận

- Tiếng Anh<sup>204</sup>
- Tiếng Việt<sup>205</sup>

### 11.2.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Nguyễn Duy Du
- Phạm Minh Đức
- Phạm Hồng Vinh

## 11.3 Mạng Nơ-ron Hồi tiếp Sâu

Cho đến nay, chúng ta mới chỉ thảo luận về các mạng nơ-ron hồi tiếp với duy nhất một tầng ẩn đơn hướng. Trong đó, cách các biến tiềm ẩn và các quan sát tương tác với nhau còn khá khái tuỳ ý. Đây không phải là một vấn đề lớn miễn là ta vẫn có đủ độ linh hoạt để mô hình hóa các loại tương tác khác nhau. Tuy nhiên, đây lại là một thách thức với các mạng đơn tầng. Trong trường hợp của perceptron, chúng ta giải quyết vấn đề này bằng cách đưa thêm nhiều tầng vào mạng. Cách này hơi phức tạp một chút với trường hợp của mạng RNN, vì đầu tiên chúng ta cần phải quyết định thêm tính phi tuyến vào mạng ở đâu và như thế nào. Thảo luận dưới đây tập trung chủ yếu vào LSTM, nhưng cũng có thể áp dụng cho các mô hình chuỗi khác.

- Chúng ta có thể bổ sung thêm tính phi tuyến vào các cơ chế cổng. Nghĩa là, thay vì sử dụng một tầng perceptron duy nhất, chúng ta có thể sử dụng nhiều tầng perceptron. Cách này không làm thay đổi cơ chế của mạng LSTM, ngược lại, còn làm cho nó tinh xảo hơn. Điều này chỉ có lợi nếu chúng ta tin rằng cơ chế LSTM biểu diễn một hình thái phổ quát nào đó về cách hoạt động của các mô hình tự hồi quy biến tiềm ẩn.
- Chúng ta có thể chồng nhiều tầng LSTM lên nhau. Cách này tạo ra một cơ chế linh hoạt hơn nhờ vào sự kết hợp giữa các tầng đơn giản. Đặc biệt là, các đặc tính liên quan của dữ liệu có thể được biểu diễn ở các tầng khác nhau. Ví dụ, chúng ta có thể muốn lưu dữ liệu về tình hình thị trường tài chính (thị trường giá lên hay giá xuống) ở tầng cao hơn, trong khi đó chỉ ghi lại động lực thời hạn ngắn hơn ở một tầng thấp hơn.

Ngoài những thứ khá trừu tượng trên, để hiểu được các nhóm mô hình chúng ta đang thảo luận một cách dễ dàng nhất, chúng ta nên xem lại Fig. 11.3.1. Hình trên mô tả một mạng nơ-ron hồi

<sup>204</sup> <https://discuss.mxnet.io/t/2368>

<sup>205</sup> <https://forum.machinelearningcoban.com/c/d21>

tiếp sâu với  $L$  tầng ẩn. Mỗi trạng thái ẩn liên tục được truyền tới bước thời gian kế tiếp ở tầng hiện tại và tới bước thời gian hiện tại ở tầng kế tiếp.

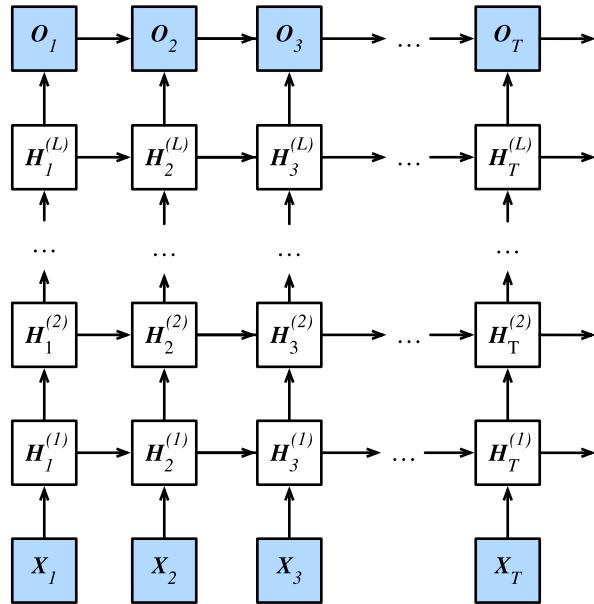


Fig. 11.3.1: Kiến trúc của một mạng nơ-ron hồi tiếp sâu.

### 11.3.1 Các Phụ thuộc Hàm

Tại bước thời gian  $t$ , giả sử rằng chúng ta có một minibatch  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  (số lượng mẫu:  $n$ , số lượng đầu vào:  $d$ ). Trạng thái ẩn của tầng ẩn  $\ell$  ( $\ell = 1, \dots, T$ ) là  $\mathbf{H}_t^{(\ell)} \in \mathbb{R}^{n \times h}$  (số đơn vị ẩn:  $h$ ), biến tầng đầu ra là  $\mathbf{O}_t \in \mathbb{R}^{n \times q}$  (số lượng đầu ra:  $q$ ) và một hàm kích hoạt tầng ẩn  $f_l$  cho tầng  $\ell$ . Chúng ta tính toán trạng thái ẩn của tầng đầu tiên như trước đây, sử dụng đầu vào là  $\mathbf{X}_t$ . Đối với tất cả các tầng tiếp theo, trạng thái ẩn của tầng trước được sử dụng thay cho  $\mathbf{X}_t$ .

$$\begin{aligned}\mathbf{H}_t^{(1)} &= f_1 \left( \mathbf{X}_t, \mathbf{H}_{t-1}^{(1)} \right), \\ \mathbf{H}_t^{(l)} &= f_l \left( \mathbf{H}_t^{(l-1)}, \mathbf{H}_{t-1}^{(l)} \right).\end{aligned}\tag{11.3.1}$$

Cuối cùng, tầng đầu ra chỉ dựa trên trạng thái ẩn của tầng ẩn  $L$ . Chúng ta sử dụng một hàm đầu ra  $g$  để xử lý trạng thái này:

$$\mathbf{O}_t = g \left( \mathbf{H}_t^{(L)} \right).\tag{11.3.2}$$

Giống như perceptron đa tầng, số tầng ẩn  $L$  và số đơn vị ẩn  $h$  được coi là các siêu tham số. Đặc biệt, chúng ta có thể chọn một trong các kiến trúc RNN, GRU, hoặc LSTM thông thường để xây dựng mô hình.

### 11.3.2 Lập trình Súc tích

May mắn thay nhiều chi tiết hỗ trợ cần thiết để lập trình một kiến trúc RNN đa tầng đã có sẵn trong Gluon. Để đơn giản, chúng ta chỉ minh họa việc lập trình bằng cách sử dụng những chức năng được tích hợp sẵn. Mã nguồn dưới đây rất giống đoạn mã mà ta sử dụng cho mạng LSTM trước đây. Trong thực tế, sự khác biệt duy nhất là chúng ta chỉ định số lượng các tầng một cách tường minh thay vì chọn mặc định là một tầng duy nhất. Hãy bắt đầu bằng cách nhập các mô-đun thích hợp và nạp dữ liệu.

```
from d2l import mxnet as d2l
from mxnet import np
from mxnet.gluon import rnn
npx.set_np()

batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
```

Các quyết định liên quan tới kiến trúc mạng (ví dụ như lựa chọn các tham số) rất giống với những phần trước. Chúng ta sử dụng cùng số lượng đầu vào và đầu ra bởi ta có các token không trùng lặp nhau, ở đây là vocab\_size. Số lượng nút ẩn vẫn là 256. Sự khác biệt duy nhất là bây giờ chúng ta sẽ chọn một giá trị lớn hơn 1 cho số tầng num\_layers = 2.

```
vocab_size, num_hiddens, num_layers, ctx = len(vocab), 256, 2, d2l.try_gpu()
lstm_layer = rnn.LSTM(num_hiddens, num_layers)
model = d2l.RNNModel(lstm_layer, len(vocab))
```

### 11.3.3 Huấn luyện

Quá trình huấn luyện tương tự như trước đây. Sự khác biệt duy nhất là bây giờ chúng ta khởi tạo mô hình với hai tầng LSTM. Quá trình huấn luyện sẽ chậm hơn đáng kể do kiến trúc mô hình phức tạp hơn và số lượng epoch lớn.

```
num_epochs, lr = 500, 2
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

### 11.3.4 Tóm tắt

- Trong các mạng nơ-ron hồi tiếp sâu, thông tin trạng thái ẩn được truyền tới bước thời gian kế tiếp ở tầng hiện tại và truyền tới bước thời gian hiện tại ở tầng kế tiếp.
- Có nhiều phiên bản khác nhau của mạng RNN sâu, ví dụ như LSTM, GRU hoặc RNN thông thường. Những mô hình này được lập trình sẵn trong mô-đun `rnn` của Gluon.
- Chúng ta cần phải cẩn thận trong việc khởi tạo mô hình. Nhìn chung, các mạng RNN sâu thường đòi hỏi khá nhiều công sức (ví dụ như việc chọn tốc độ học hay việc gọt gradient) để đảm bảo quá trình học hội tụ một cách hợp lý.

### 11.3.5 Bài tập

1. Hãy lập trình một mạng RNN hai tầng từ đầu sử dụng mã nguồn cho mạng một tầng mà ta đã thảo luận ở Section 10.5.
2. Hãy thay thế khối LSTM bằng khối GRU và so sánh độ chính xác của mô hình.
3. Tăng dữ liệu huấn luyện bằng việc thêm nhiều cuốn sách. Bạn có thể giảm perplexity tới mức nào?
4. Có nên kết hợp nhiều nguồn sách từ các tác giả khác nhau khi mô hình hoá dữ liệu văn bản hay không?. Tại sao việc này lại là một ý tưởng hay? Vấn đề gì có thể xảy ra ở đây?

### 11.3.6 Thảo luận

- Tiếng Anh<sup>206</sup>
- Tiếng Việt<sup>207</sup>

### 11.3.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Phạm Minh Đức

## 11.4 Mạng Nơ-ron Hồi tiếp Hai chiều

Cho đến nay ta giả định mục tiêu là để mô hình hóa bước thời gian kế tiếp dựa trên những thông tin trước đó, điển hình như chuỗi thời gian hay một mô hình ngôn ngữ. Tuy nhiên, đây không phải là trường hợp duy nhất chúng ta có thể gặp. Để minh họa cho vấn đề này, hãy xem xét ba tác vụ điền vào chỗ trống dưới đây:

1. Tôi \_\_\_\_\_
2. Tôi \_\_\_\_\_ đòi lăm.
3. Tôi \_\_\_\_\_ đòi lăm, tôi có thể ăn một nửa con lợn.

Tùy thuộc vào số lượng thông tin có sẵn, chúng ta có thể điền vào chỗ trống với các từ khác nhau như “hạnh phúc”, “không”, và “đang”. Rõ ràng phần kết (nếu có) của câu mang thông tin quan trọng ảnh hưởng lớn đến việc chọn từ. Một mô hình chuỗi sẽ thực hiện các tác vụ liên quan kém

<sup>206</sup> <https://discuss.mxnet.io/t/2369>

<sup>207</sup> <https://forum.machinelearningcoban.com/c/d21>

hiệu quả nếu nó không khai thác tốt được đặc điểm này. Chẳng hạn, để nhận dạng thực thể có tên (ví dụ: phân biệt từ “Bảy” đề cập đến “ông Bảy” hay là số bảy) một cách hiệu quả, ngữ cảnh khoảng dài cũng không kém phần quan trọng. Chúng ta sẽ dành một chút thời gian tìm hiểu các mô hình đồ thị để tìm nguồn cảm hứng giải quyết bài toán trên.

### 11.4.1 Quy hoạch Động

Trong phần này, chúng ta sẽ tìm hiểu bài toán quy hoạch động. Không cần thiết phải hiểu chi tiết về quy hoạch động để hiểu kỹ thuật tương ứng trong học sâu nhưng chúng góp phần giải thích lý do tại sao học sâu được sử dụng và tại sao một vài kiến trúc mạng nhất định lại được lựa chọn.

Nếu muốn giải quyết bài toán bằng mô hình đồ thị thì chúng ta có thể thiết kế một mô hình biến tiệm ẩn như ví dụ sau đây. Giả sử tồn tại biến tiệm ẩn  $h_t$  quyết định giá trị quan sát  $x_t$  qua xác suất  $p(x_t | h_t)$ . Hơn nữa, quá trình chuyển đổi  $h_t \rightarrow h_{t+1}$  được cho bởi xác suất chuyển trạng thái  $p(h_{t+1} | h_t)$ . Mô hình đồ thị khi đó là mô hình Markov ẩn (*Hidden Markov Model - HMM*) như trong Fig. 11.4.1.

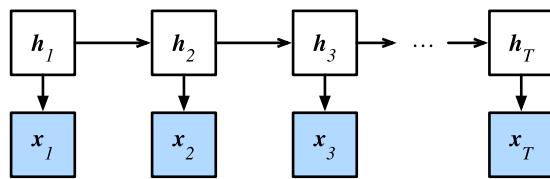


Fig. 11.4.1: Mô hình Markov ẩn.

Như vậy, với chuỗi có  $T$  quan sát, chúng ta có phân phối xác suất kết hợp của các trạng thái ẩn và các quan sát như sau:

$$p(x, h) = p(h_1)p(x_1 | h_1) \prod_{t=2}^T p(h_t | h_{t-1})p(x_t | h_t). \quad (11.4.1)$$

Bây giờ giả sử chúng ta đã có tất cả các quan sát  $x_i$  ngoại trừ một vài quan sát  $x_j$ , mục tiêu là tính xác suất  $p(x_j | x^{-j})$ , trong đó  $x^{-j} = (x_1, x_2, \dots, x_{j-1})$ . Để thực hiện điều này, chúng ta cần tính tổng xác suất trên tất cả các khả năng có thể của  $h = (h_1, \dots, h_T)$ . Trong trường hợp  $h_i$  nhận  $k$  giá trị khác nhau, chúng ta cần tính tổng của  $k^T$  số hạng - đây là một nhiệm vụ bất khả thi. May mắn thay có một phương pháp rất hiệu quả cho bài toán trên, đó là quy hoạch động.

Để hiểu hơn về phương pháp này, hãy xem xét tổng của hai biến ẩn đầu tiên  $h_1$  và  $h_2$ . Ta có:

$$\begin{aligned}
p(x) &= \sum_{h_1, \dots, h_T} p(x_1, \dots, x_T; h_1, \dots, h_T) \\
&= \sum_{h_1, \dots, h_T} p(h_1)p(x_1 | h_1) \prod_{t=2}^T p(h_t | h_{t-1})p(x_t | h_t) \\
&= \sum_{h_2, \dots, h_T} \underbrace{\left[ \sum_{h_1} p(h_1)p(x_1 | h_1)p(h_2 | h_1) \right]}_{=: \pi_2(h_2)} p(x_2 | h_2) \prod_{t=3}^T p(h_t | h_{t-1})p(x_t | h_t) \\
&\quad = \dots \\
&= \sum_{h_3, \dots, h_T} \underbrace{\left[ \sum_{h_2} \pi_2(h_2)p(x_2 | h_2)p(h_3 | h_2) \right]}_{=: \pi_3(h_3)} p(x_3 | h_3) \prod_{t=4}^T p(h_t | h_{t-1})p(x_t | h_t) \\
&= \dots \\
&= \sum_{h_T} \pi_T(h_T)p(x_T | h_T).
\end{aligned} \tag{11.4.2}$$

Cơ bản, chúng ta có công thức *đệ quy xuôi* như sau:

$$\pi_{t+1}(h_{t+1}) = \sum_{h_t} \pi_t(h_t)p(x_t | h_t)p(h_{t+1} | h_t). \tag{11.4.3}$$

Phép đệ quy được khởi tạo với  $\pi_1(h_1) = p(h_1)$ . Nói chung, công thức đệ quy có thể được viết lại là  $\pi_{t+1} = f(\pi_t, x_t)$ , trong đó  $f$  là một hàm được học. Trông rất giống với phương trình cập nhật trong các mô hình biến ẩn mà chúng ta đã thảo luận trong phần RNN. Tương tự, chúng ta có thể tính *đệ quy ngược* như sau:

$$\begin{aligned}
p(x) &= \sum_{h_1, \dots, h_T} p(x_1, \dots, x_T; h_1, \dots, h_T) \\
&= \sum_{h_1, \dots, h_T} \prod_{t=1}^{T-1} p(h_t | h_{t-1})p(x_t | h_t) \cdot p(h_T | h_{T-1})p(x_T | h_T) \\
&= \sum_{h_1, \dots, h_{T-1}} \prod_{t=1}^{T-1} p(h_t | h_{t-1})p(x_t | h_t) \cdot \underbrace{\left[ \sum_{h_T} p(h_T | h_{T-1})p(x_T | h_T) \right]}_{=: \rho_{T-1}(h_{T-1})} \\
&= \sum_{h_1, \dots, h_{T-2}} \prod_{t=1}^{T-2} p(h_t | h_{t-1})p(x_t | h_t) \cdot \underbrace{\left[ \sum_{h_{T-1}} p(h_{T-1} | h_{T-2})p(x_{T-1} | h_{T-1})\rho_{T-1}(h_{T-1}) \right]}_{=: \rho_{T-2}(h_{T-2})} \\
&= \dots, \\
&= \sum_{h_1} p(h_1)p(x_1 | h_1)\rho_1(h_1).
\end{aligned} \tag{11.4.4}$$

Từ đó, chúng ta có thể viết *đệ quy ngược* như sau:

$$\rho_{t-1}(h_{t-1}) = \sum_{h_t} p(h_t | h_{t-1})p(x_t | h_t)\rho_t(h_t), \tag{11.4.5}$$

khi khởi tạo  $\rho_T(h_T) = 1$ . Hai biểu thức đệ quy này cho phép ta tính tổng trên tất cả  $T$  biến trong khoảng  $(h_1, \dots, h_T)$  với thời gian  $\mathcal{O}(kT)$  tăng tuyến tính thay vì luỹ thừa. Đây là một trong những điểm mạnh của kỹ thuật suy luận xác suất với các mô hình đồ thị. Đây là một trường hợp đặc biệt của kỹ thuật được trình bày trong (Aji & McEliece, 2000) bởi Aji và McEliece vào năm 2000.

Kết hợp cả biểu thức xuôi và ngược ta có thể tính được:

$$p(x_j | x_{-j}) \propto \sum_{h_j} \pi_j(h_j) \rho_j(h_j) p(x_j | h_j). \quad (11.4.6)$$

Cần phải chú ý rằng khi suy rộng ra, biểu thức đệ quy ngược có thể được viết dưới dạng  $\rho_{t-1} = g(\rho_t, x_t)$ , trong đó  $g$  là một hàm số được học. Một lần nữa, nó trông giống như một phương trình cập nhật chỉ chạy ngược lại, không giống như những gì chúng ta thấy ở RNN.

Thật vậy, HMM sẽ có lợi từ việc học các dữ liệu trong tương lai (nếu có thể). Các nhà khoa học chuyên về xử lý tín hiệu sẽ tách biệt 2 trường hợp biết trước và không biết trước các kết quả trong tương lai thành nội suy và ngoại suy. Ta có thể tham khảo chương giới thiệu của cuốn (Doucet et al., 2001) về các thuật toán Monte Carlo tuần tự để biết thêm chi tiết.

### 11.4.2 Mô hình Hai chiều

Nếu chúng ta muốn mạng RNN có một cơ chế nhìn trước giống như HMM thì ta cần phải chỉnh sửa một chút thiết kế của các mạng hồi tiếp truyền thống. May mắn là, điều này khá đơn giản về mặt khái niệm. Thay vì chỉ vận hành một RNN chạy từ kí tự đầu đến cuối, ta sẽ khởi tạo một RNN nữa chạy từ kí tự cuối lên đầu. *Mạng nơ-ron hồi tiếp hai chiều (Bidirectional recurrent neural network)* sẽ thêm một tầng ẩn cho phép xử lý dữ liệu theo chiều ngược lại một cách linh hoạt hơn so với RNN truyền thống. Fig. 11.4.2 mô tả cấu trúc của mạng nơ-ron hồi tiếp hai chiều với một tầng ẩn.

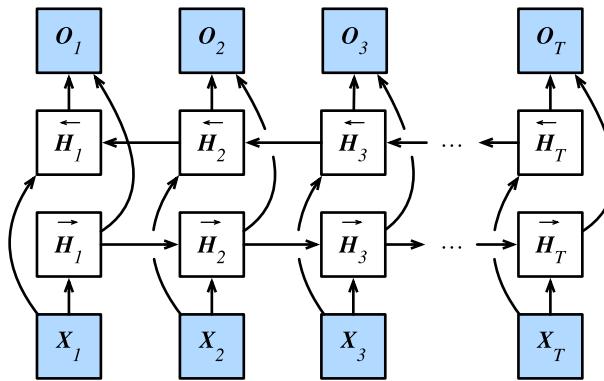


Fig. 11.4.2: Cấu trúc của mạng nơ-ron hồi tiếp hai chiều.

Trên thực tế, điều này không quá khác biệt với phép đệ quy xuôi và ngược mà ta đã đề cập ở phần trước. Điểm khác biệt chính là trước đây các phương trình này có một ý nghĩa thống kê nhất định. Còn bây giờ thì chúng không còn mang một ý nghĩa dễ hiểu nào nhất định, thay vào đó ta sẽ chỉ xét chúng như những hàm tổng quát. Quá trình chuyển đổi này là điển hình cho nhiều nguyên tắc thiết kế các mạng học sâu hiện đại: đầu tiên, sử dụng các dạng quan hệ phụ thuộc hàm của các mô hình thống kê cổ điển, sau đó sử dụng các mô hình này dưới dạng tổng quát.

## Định nghĩa

Các mạng nơ-ron hồi tiếp hai chiều đã được giới thiệu bởi (Schuster & Paliwal, 1997). Ta có thể xem thêm (Graves & Schmidhuber, 2005) về những thảo luận chi tiết của các kiến trúc khác nhau. Còn giờ ta hãy đi vào chi tiết của một mạng như vậy.

Cho một bước thời gian  $t$ , đầu vào minibatch là  $\mathbf{X}_t \in \mathbb{R}^{n \times d}$  ( $n$  là số lượng mẫu,  $d$  là số lượng đầu vào) và hàm kích hoạt của tầng ẩn là  $\phi$ . Trong kiến trúc hai chiều, ta giả định rằng trạng thái ẩn xuôi và ngược của bước thời gian này lần lượt là  $\vec{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$  và  $\overleftarrow{\mathbf{H}}_t \in \mathbb{R}^{n \times h}$ .  $h$  ở đây chỉ số lượng nút ẩn. Chúng ta tính toán việc cập nhật xuôi và ngược của trạng thái ẩn như sau:

$$\begin{aligned}\vec{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \vec{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)}), \\ \overleftarrow{\mathbf{H}}_t &= \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)}).\end{aligned}\quad (11.4.7)$$

Ở đây, các trọng số  $\mathbf{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}$ ,  $\mathbf{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}$ ,  $\mathbf{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}$ , và  $\mathbf{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$  và các độ chêch  $\mathbf{b}_h^{(f)} \in \mathbb{R}^{1 \times h}$  và  $\mathbf{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$  đều là tham số mô hình.

Sau đó, chúng ta nối các trạng thái ẩn xuôi và ngược ( $\vec{\mathbf{H}}_t$ ,  $\overleftarrow{\mathbf{H}}_t$ ) để thu được trạng thái ẩn  $\mathbf{H}_t \in \mathbb{R}^{n \times 2h}$  và truyền nó đến tầng đầu ra. Trong các mạng nơ-ron hồi tiếp hai chiều sâu, thông tin được truyền đi như là *đầu vào* cho tầng hai chiều (*bidirectional layer*) tiếp theo. Cuối cùng, tầng đầu ra sẽ tính toán đầu ra  $\mathbf{O}_t \in \mathbb{R}^{n \times q}$  ( $q$  là số lượng đầu ra) như sau:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q. \quad (11.4.8)$$

Ở đây, trọng số  $\mathbf{W}_{hq} \in \mathbb{R}^{2h \times q}$  và độ chêch  $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$  là các tham số mô hình của tầng đầu ra. Hai chiều ngược và xuôi có thể có số nút ẩn khác nhau.

## Chi phí Tính toán và 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 lượng kết quả đầu ra. Chúng ta sử dụng thông tin từ các quan sát trong tương lai và quá khứ để dự đoán hiện tại (như để làm mượt). Trong trường hợp mô hình ngôn ngữ, đây không hẳn là điều chúng ta muốn. Rốt cuộc, chúng ta không thể biết biểu tượng tiếp sau biểu tượng đang cần dự đoán. Do đó, nếu chúng ta sử dụng RNN hai chiều một cách ngây thơ, chúng ta sẽ không có được độ chính xác đủ tốt: trong quá trình huấn luyện, chúng ta có cả dữ liệu quá khứ và tương lai để ước tính hiện tại. Trong quá trình dự đoán, chúng ta chỉ có dữ liệu trong quá khứ và do đó kết quả dự đoán có độ chính xác kém (điều này được minh họa trong thí nghiệm bên dưới).

Tệ hơn, RNN hai chiều cũng cực kỳ chậm. Những lý do chính cho điều này là vì chúng cần cả lượt truyền xuôi và lượt truyền ngược, và lượt truyền ngược thì phụ thuộc vào kết quả của lượt truyền xuôi. Do đó, gradient sẽ có một chuỗi phụ thuộc rất dài.

Trong thực tế, các tầng hai chiều được sử dụng rất ít và chỉ dành cho một số ít ứng dụng, chẳng hạn như điền từ còn thiếu, chú thích token (ví dụ cho nhận dạng thực thể có tên) hoặc mã hóa nguyên chuỗi tại một bước trong pipeline xử lý chuỗi (ví dụ trong dịch máy). Tóm lại, hãy sử dụng nó một cách cẩn thận!

## Huấn luyện Mạng RNN Hai chiều cho Úng dụng không Phù hợp

Nếu chúng ta bỏ qua tất cả các lời khuyên về việc LSTM hai chiều sử dụng cả dữ liệu trong quá khứ và tương lai, và cứ áp dụng nó cho các mô hình ngôn ngữ, chúng ta sẽ có được các ước lượng với perplexity chấp nhận được. Tuy nhiên, khả năng dự đoán các ký tự trong tương lai của mô hình bị tổn hại nghiêm trọng như minh họa trong ví dụ dưới đây. Mặc dù đạt được mức perplexity hợp lý, nó chỉ sinh ra các chuỗi vô nghĩa ngay cả sau nhiều vòng lặp. Chúng tôi sử dụng đoạn mã dưới đây như một ví dụ cảnh báo về việc sử dụng chúng ở sai bối cảnh.

```
from d2l import mxnet as d2l
from mxnet import np
from mxnet.gluon import rnn
npx.set_np()

# Load data
batch_size, num_steps = 32, 35
train_iter, vocab = d2l.load_data_time_machine(batch_size, num_steps)
# Define the model
vocab_size, num_hiddens, num_layers, ctx = len(vocab), 256, 2, d2l.try_gpu()
lstm_layer = rnn.LSTM(num_hiddens, num_layers, bidirectional=True)
model = d2l.RNNModel(lstm_layer, len(vocab))
# Train the model
num_epochs, lr = 500, 1
d2l.train_ch8(model, train_iter, vocab, lr, num_epochs, ctx)
```

Đầu ra rõ ràng không hề tốt vì những lý do trên. Để thảo luận về việc sử dụng hiệu quả hơn các mô hình hai chiều, vui lòng xem bài toán phân loại cảm xúc trong [Section 17.2](#).

### 11.4.3 Tóm tắt

- Trong các mạng nơ-ron hồi tiếp hai chiều, trạng thái ẩn tại mỗi bước thời gian được xác định đồng thời bởi dữ liệu ở trước và sau bước thời gian đó.
- Các RNN hai chiều có sự tương đồng đáng kinh ngạc với thuật toán xuôi–ngược trong các mô hình đồ thị.
- RNN hai chiều chủ yếu hữu ích cho việc tạo embedding chuỗi và việc ước lượng dữ liệu quan sát được khi biết bối cảnh hai chiều.
- Việc huấn luyện RNN hai chiều rất tốn kém do các chuỗi gradient dài.

### 11.4.4 Bài tập

1. Nếu các hướng khác nhau sử dụng số nút ẩn khác nhau, kích thước của  $\mathbf{H}_t$  sẽ thay đổi như thế nào?
2. Thiết kế một mạng nơ-ron hồi tiếp hai chiều với nhiều tầng ẩn.
3. Lập trình thuật toán phân loại chuỗi bằng cách sử dụng các RNN hai chiều. **Gợi ý:** sử dụng RNN để tạo embedding cho từng từ và sau đó tổng hợp (lấy trung bình) tất cả các embedding đầu ra trước khi đưa chúng vào mô hình MLP để phân loại. Chẳng hạn, nếu chúng ta có  $(\mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3)$ , ta sẽ tính  $\bar{\mathbf{o}} = \frac{1}{3} \sum_i \mathbf{o}_i$  trước rồi sử dụng nó để phân loại cảm xúc.

#### 11.4.5 Thảo luận

- Tiếng Anh<sup>208</sup>
- Tiếng Việt<sup>209</sup>

#### 11.4.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Đinh Phước Lộc
- Võ Tấn Phát
- Nguyễn Thanh Hòa
- Trần Yến Thy
- Phạm Hồng Vinh

### 11.5 Dịch Máy và Tập dữ liệu

Đến nay ta đã thấy cách sử dụng mạng nơ-ron hồi tiếp cho các mô hình ngôn ngữ, mà ở đó ta dự đoán token tiếp theo khi biết tất cả token trước đó. Nay ta sẽ xem xét một ứng dụng khác để dự đoán một chuỗi token thay vì chỉ một token đơn lẻ.

Dịch máy (*Machine translation - MT*) đề cập đến việc dịch tự động một đoạn văn bản từ ngôn ngữ này sang ngôn ngữ khác. Giải quyết bài toán này với các mạng nơ-ron thường được gọi là dịch máy nơ-ron (*neural machine translation - NMT*). So với các mô hình ngôn ngữ (Section 10.3), trong đó kho ngữ liệu chỉ chứa một ngôn ngữ duy nhất, bộ dữ liệu dịch máy có ít nhất hai ngôn ngữ, ngôn ngữ nguồn và ngôn ngữ đích. Ngoài ra, mỗi câu trong ngôn ngữ nguồn được ánh xạ tới bản dịch tương ứng trong ngôn ngữ đích. Do đó, cách tiền xử lý dữ liệu dịch máy sẽ khác so với mô hình ngôn ngữ. Phần này được dành riêng để trình bày cách tiền xử lý và nạp một tập dữ liệu như vậy vào các minibatch.

```
from d2l import mxnet as d2l
from mxnet import np, npx, gluon
import os
npx.set_np()
```

<sup>208</sup> <https://discuss.mxnet.io/t/2370>

<sup>209</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 11.5.1 Đọc và Tiền Xử lý Dữ liệu

Trước tiên ta tải xuống bộ dữ liệu chứa một tập các câu tiếng Anh cùng với các bản dịch tiếng Pháp tương ứng. Có thể thấy mỗi dòng chứa một câu tiếng Anh cùng với bản dịch tiếng Pháp tương ứng, cách nhau bởi một dấu TAB.

```
# Saved in the d2l package for later use
d2l.DATA_HUB['fra-eng'] = (d2l.DATA_URL + 'fra-eng.zip',
                            '94646ad1522d915e7b0f9296181140edcf86a4f5')

# Saved in the d2l package for later use
def read_data_nmt():
    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[0:106])
```

Ta sẽ thực hiện một số bước tiền xử lý trên dữ liệu văn bản thô, bao gồm chuyển đổi tất cả ký tự sang chữ thường, thay thế các ký tự khoảng trắng không ngắt (*non-breaking space*) UTF-8 bằng dấu cách, thêm dấu cách vào giữa các từ và các dấu câu.

```
# Saved in the d2l package for later use
def preprocess_nmt(text):
    def no_space(char, prev_char):
        return char in set(',.!') and prev_char != ' '

    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()
    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[0:95])
```

### 11.5.2 Token hóa

Khác với việc sử dụng ký tự làm token trong Section 10.3, ở đây một token là một từ hoặc dấu câu. Hàm sau đây sẽ token hóa dữ liệu văn bản để trả về source và target là hai danh sách chứa các danh sách token, với source [i] là câu thứ  $i$  trong ngôn ngữ nguồn và target [i] là câu thứ  $i$  trong ngôn ngữ đích. Để việc huấn luyện sau này nhanh hơn, chúng ta chỉ lấy  $\text{num\_examples}$  cặp câu đầu tiên.

```
# Saved in the d2l package for later use
def tokenize_nmt(text, num_examples=None):
    source, target = [], []
    for i, line in enumerate(text.split('\n')):
        if num_examples and i > num_examples:
            break
        parts = line.split('\t')
        if len(parts) == 2:
```

(continues on next page)

```

        source.append(parts[0].split(' '))
        target.append(parts[1].split(' '))
    return source, target

source, target = tokenize_nmt(text)
source[0:3], target[0:3]

```

Dưới đây là biểu đồ tần suất của số lượng token cho mỗi câu. Có thể thấy, trung bình một câu chứa 5 token và hầu hết các câu có ít hơn 10 token.

```

d2l.set_figsize((3.5, 2.5))
d2l.plt.hist([[len(l) for l in source], [len(l) for l in target]],
            label=['source', 'target'])
d2l.plt.legend(loc='upper right');

```

### 11.5.3 Bộ Từ vựng

Vì các token trong ngôn ngữ nguồn có thể khác với các token trong ngôn ngữ đích, ta cần xây dựng một bộ từ vựng cho mỗi ngôn ngữ. Do ta đang sử dụng các từ để làm token chứ không dùng ký tự, kích thước bộ từ vựng sẽ lớn hơn đáng kể. Ở đây ta sẽ ánh xạ mọi token xuất hiện ít hơn 3 lần vào token <unk> như trong [Section 10.2](#). Ngoài ra, ta cần các token đặc biệt khác như token đệm <pad>, hay token bắt đầu câu <bos>.

```

src_vocab = d2l.Vocab(source, min_freq=3,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
len(src_vocab)

```

### 11.5.4 Nạp Dữ liệu

Trong các mô hình ngôn ngữ, mỗi mẫu là một chuỗi có độ dài num\_steps từ kho ngữ liệu, mà có thể là một phân đoạn của một câu hoặc trải dài trên nhiều câu. Trong dịch máy, một mẫu bao gồm một cặp câu nguồn và câu đích. Những câu này có thể có độ dài khác nhau, trong khi đó ta cần các mẫu có cùng độ dài để tạo minibatch.

Một cách giải quyết vấn đề này là nếu một câu dài hơn num\_steps, ta sẽ cắt bớt độ dài của nó, ngược lại nếu một câu ngắn hơn num\_steps, thì ta sẽ đệm thêm token <pad>. Bằng cách này, ta có thể chuyển bất cứ câu nào về một độ dài cố định.

```

# Saved in the d2l package for later use
def truncate_pad(line, num_steps, padding_token):
    if len(line) > num_steps:
        return line[:num_steps] # Trim
    return line + [padding_token] * (num_steps - len(line)) # Pad

truncate_pad(src_vocab[source[0]], 10, src_vocab['<pad>'])

```

Bây giờ ta có thể chuyển đổi danh sách các câu thành mảng chỉ số có kích thước (num\_examples, num\_steps). Ta cũng ghi lại độ dài của mỗi câu khi không có token đệm, được gọi là *độ dài hợp lệ - valid length*. Thông tin này có thể được sử dụng bởi một số mô hình. Ngoài ra, ta sẽ thêm các

token đặc biệt “<bos>” và “<eos>” vào các câu đích để mô hình biết thời điểm bắt đầu và kết thúc dự đoán.

```
# Saved in the d2l package for later use
def build_array(lines, vocab, num_steps, is_source):
    lines = [vocab[l] for l in lines]
    if not is_source:
        lines = [[vocab['<bos>']] + l + [vocab['<eos>']] for l in lines]
    array = np.array([truncate_pad(
        l, num_steps, vocab['<pad>']) for l in lines])
    valid_len = (array != vocab['<pad>']).sum(axis=1)
    return array, valid_len
```

Sau đó, ta có thể xây dựng các minibatch dựa trên các mảng này.

### 11.5.5 Kết hợp Tất cả lại

Cuối cùng, ta định nghĩa hàm load\_data\_nmt để trả về iterator cho dữ liệu cùng với các bộ từ vựng cho ngôn ngữ nguồn và ngôn ngữ đích.

```
# Saved in the d2l package for later use
def load_data_nmt(batch_size, num_steps, num_examples=1000):
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)
    src_vocab = d2l.Vocab(source, min_freq=3,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = d2l.Vocab(target, min_freq=3,
                          reserved_tokens=['<pad>', '<bos>', '<eos>'])
    src_array, src_valid_len = build_array(
        source, src_vocab, num_steps, True)
    tgt_array, tgt_valid_len = build_array(
        target, tgt_vocab, num_steps, False)
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = d2l.load_array(data_arrays, batch_size)
    return src_vocab, tgt_vocab, data_iter
```

Hãy thử đọc batch đầu tiên.

```
src_vocab, tgt_vocab, train_iter = load_data_nmt(batch_size=2, num_steps=8)
for X, X_vlen, Y, Y_vlen in train_iter:
    print('X:', X.astype('int32'))
    print('Valid lengths for X:', X_vlen)
    print('Y:', Y.astype('int32'))
    print('Valid lengths for Y:', Y_vlen)
    break
```

### 11.5.6 Tóm tắt

- Dịch máy (*machine translation* - MT) là việc dịch tự động một đoạn văn bản từ ngôn ngữ này sang ngôn ngữ khác.
- Ta đọc, tiền xử lý và token hóa bộ dữ liệu từ cả ngôn ngữ nguồn và ngôn ngữ đích.

### 11.5.7 Bài tập

Tìm và xử lý một bộ dữ liệu dịch máy.

### 11.5.8 Thảo luận

- Tiếng Anh<sup>210</sup>
- Tiếng Việt<sup>211</sup>

### 11.5.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Nguyễn Văn Quang
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Hồng Vinh

## 11.6 Kiến trúc Mã hoá - Giải mã

*Kiến trúc mã hoá - giải mã* (*encoder-decoder architecture*) là một khuôn mẫu thiết kế mạng nơ-ron. Kiến trúc này có 2 phần: bộ mã hoá và bộ giải mã, có thể thấy trong Fig. 11.6.1. Bộ mã hoá đóng vai trò mã hoá đầu vào thành trạng thái chứa vài tensor. Tiếp đó, trạng thái được truyền vào bộ giải mã để sinh đầu ra. Trong dịch máy, bộ mã hoá biến đổi một câu nguồn, ví dụ như “Hello world.”, thành trạng thái, chẳng hạn là một vector chứa thông tin ngữ nghĩa của câu đó. Sau đó bộ giải mã sử dụng trạng thái này để dịch câu sang ngôn ngữ đích, ví dụ sang tiếng Pháp “Bonjour le monde.”.

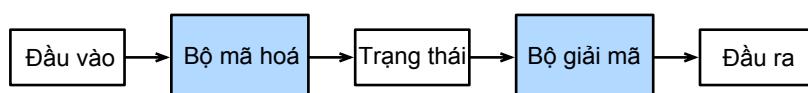


Fig. 11.6.1: Kiến trúc mã hoá - giải mã

<sup>210</sup> <https://discuss.mxnet.io/t/2396>

<sup>211</sup> <https://forum.machinelearningcoban.com/c/d21>

Phần này trình bày một giao diện (*interface*) để lập trình kiến trúc mã hoá - giải mã.

### 11.6.1 Bộ mã hoá

Bộ mã hoá là một mạng nơ-ron thông thường, nhận đầu vào, ví dụ như một câu nguồn, và trả về đầu ra.

```
from mxnet.gluon import nn

# Saved in the d2l package for later use
class Encoder(nn.Block):
    """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
```

### 11.6.2 Bộ giải mã

Bộ giải mã có thêm phương thức `init_state` nhằm phân tích đầu ra của bộ mã hoá với những thông tin bổ sung (nếu có), như độ dài hợp lệ của đầu vào, để đưa ra trạng thái cần thiết. Trong lan truyền xuôi, bộ giải mã nhận hai đầu vào, ví dụ như một câu đích và trạng thái. Nó trả về đầu ra với trạng thái nhiều khả năng đã thay đổi nếu bộ mã hoá chứa các tầng RNN.

```
# Saved in the d2l package for later use
class Decoder(nn.Block):
    """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
```

### 11.6.3 Mô hình

Mô hình mã hoá - giải mã bao gồm một bộ mã hoá và một bộ giải mã. Chúng ta lập trình phương thức truyền xuôi cho quá trình huấn luyện. Phương thức này nhận cả đầu vào bộ mã hoá và đầu vào bộ giải mã cùng các đối số bổ sung không bắt buộc. Mô hình tính đầu ra của bộ mã hoá để khởi tạo trạng thái bộ giải mã, sau đó trả về đầu ra của bộ giải mã.

```
# Saved in the d2l package for later use
class EncoderDecoder(nn.Block):
    """The base class for the encoder-decoder architecture."""
    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
```

(continues on next page)

```

    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)

```

#### 11.6.4 Tóm tắt

- Kiến trúc mã hoá - giải mã là một khuôn mẫu thiết kế mạng nơ-ron chủ yếu được sử dụng trong xử lý ngôn ngữ tự nhiên.
- Bộ mã hoá là một mạng (kết nối đầy đủ - FC, nơ-ron tích chập - CNN, nơ-ron hồi tiếp - RNN, ...) nhận đầu vào và trả về một ánh xạ đặc trưng là một vector hay một tensor.
- Bộ giải mã là một mạng (thường giống kiến trúc mạng của bộ mã hoá) nhận vector đặc trưng từ bộ mã hoá và đưa ra kết quả gần khớp nhất với đầu vào thực tế hoặc đầu ra mong muốn.

#### 11.6.5 Bài tập

1. Ngoài dịch máy, bạn còn biết thêm những ứng dụng nào khác của kiến trúc mã hoá - giải mã không?
2. Bạn có thể thiết kế một kiến trúc mã hoá - giải mã sâu không?

#### 11.6.6 Thảo luận

- Tiếng Anh<sup>212</sup>
- Tiếng Việt<sup>213</sup>

#### 11.6.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Thanh Hòa
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Hồng Vinh

<sup>212</sup> <https://discuss.mxnet.io/t/2393>

<sup>213</sup> <https://forum.machinelearningcoban.com/c/d21>

## 11.7 Chuỗi sang Chuỗi

Mô hình chuỗi sang chuỗi (*Sequence to Sequence – seq2seq*) dựa trên kiến trúc mã hóa - giải mã để sinh ra chuỗi đầu ra từ chuỗi đầu vào như minh họa trong Fig. 11.7.1. Cả bộ mã hóa và bộ giải mã sử dụng mạng nơ-ron hồi tiếp (RNN) để xử lý các chuỗi đầu vào với độ dài khác nhau. Trạng thái ẩn của bộ giải mã được khởi tạo trực tiếp từ trạng thái ẩn của bộ mã hóa, giúp truyền thông tin từ bộ mã hóa tới bộ giải mã.

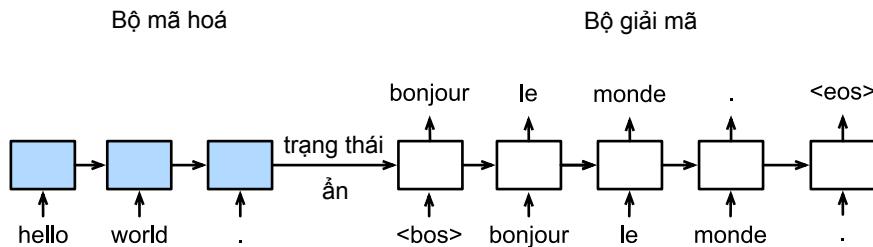


Fig. 11.7.1: Kiến trúc mô hình chuỗi sang chuỗi.

Các tầng trong bộ mã hóa và bộ giải mã được minh họa trong Fig. 11.7.2.

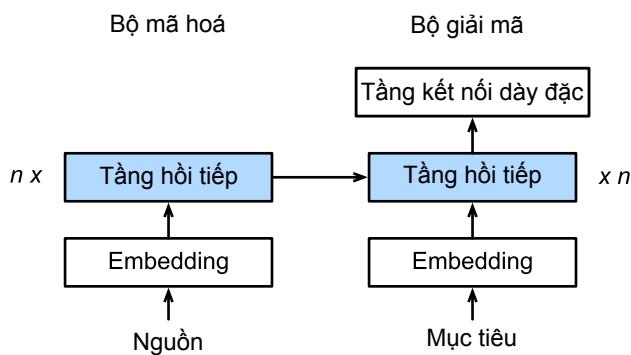


Fig. 11.7.2: Các tầng trong bộ mã hóa và bộ giải mã.

Trong phần này chúng ta sẽ tìm hiểu và lập trình mô hình seq2seq để huấn luyện trên bộ dữ liệu dịch máy.

```
from d2l import mxnet as d2l
from mxnet import np, npx, init, gluon, autograd
from mxnet.gluon import nn, rnn
npx.set_np()
```

### 11.7.1 Bộ Mã hóa

Nhắc lại rằng bộ mã hóa của mô hình seq2seq mã hóa thông tin của các chuỗi đầu vào với độ dài khác nhau thành một vector ngữ cảnh  $\mathbf{c}$ . Ta thường sử dụng các tầng RNN trong bộ mã hóa. Giả sử có một chuỗi đầu vào  $x_1, \dots, x_T$ , trong đó  $x_t$  là từ thứ  $t$ . Tại bước thời gian  $t$ , mô hình RNN sẽ có hai vector đầu vào: vector đặc trưng  $\mathbf{x}_t$  của  $x_t$  và trạng thái ẩn của bước thời gian trước đó  $\mathbf{h}_{t-1}$ . Ta ký hiệu phép chuyển đổi của các trạng thái ẩn trong RNN bằng hàm  $f$ :

$$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}). \quad (11.7.1)$$

Tiếp theo, bộ mã hóa nén bắt thông tin của tất cả các trạng thái ẩn và mã hóa chúng thành vector ngữ cảnh  $\mathbf{c}$  bằng hàm  $q$ :

$$\mathbf{c} = q(\mathbf{h}_1, \dots, \mathbf{h}_T). \quad (11.7.2)$$

Ví dụ, nếu chúng ta chọn  $q$  là  $q(\mathbf{h}_1, \dots, \mathbf{h}_T) = \mathbf{h}_T$ , thì vector ngữ cảnh sẽ là trạng thái ẩn của bước thời gian cuối cùng  $\mathbf{h}_T$ .

Cho đến nay ta mới mô tả bộ mã hóa sử dụng mạng RNN một chiều, ở đó trạng thái ẩn của mỗi bước thời gian chỉ phụ thuộc vào các bước thời gian trước. Ta cũng có thể sử dụng các dạng RNN khác nhau như GRU, LSTM, hay RNN hai chiều để mã hóa chuỗi đầu vào.

Bây giờ hãy lập trình bộ mã hóa của mô hình seq2seq. Ta sử dụng một tầng embedding từ ngữ để lấy vector đặc trưng tương ứng với chỉ số từ trong ngôn ngữ nguồn. Những vector đặc trưng này sẽ được truyền vào một mạng LSTM đa tầng. Batch đầu vào của bộ mã hóa là tensor 2 chiều có kích thước là (kích thước batch, độ dài chuỗi), với số lượng chuỗi bằng kích thước batch. Bộ mã hóa trả về cả đầu ra của LSTM, gồm các trạng thái ẩn của tất cả các bước thời gian, cùng với trạng thái ẩn và ô nhớ ở bước thời gian cuối cùng.

```
# Saved in the d2l package for later use
class Seq2SeqEncoder(d2l.Encoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)

    def forward(self, X, *args):
        X = self.embedding(X) # X shape: (batch_size, seq_len, embed_size)
        # RNN needs first axes to be timestep, i.e., seq_len
        X = X.swapaxes(0, 1)
        state = self.rnn.begin_state(batch_size=X.shape[1], ctx=X.ctx)
        out, state = self.rnn(X, state)
        # out shape: (seq_len, batch_size, num_hiddens)
        # state shape: (num_layers, batch_size, num_hiddens),
        # where "state" contains the hidden state and the memory cell
        return out, state
```

Tiếp theo, chúng ta sẽ tạo một minibatch đầu vào dạng chuỗi với kích thước batch bằng 4 cùng số bước thời gian (độ dài chuỗi) bằng 7. Giả sử nút LSTM có 2 tầng ẩn và 16 nút ẩn. Đầu ra của bộ mã hóa sau khi thực hiện lượt truyền xuôi trên đầu vào có kích thước là (số bước thời gian, kích thước batch, số nút ẩn). Nếu mạng nơ-ron hồi tiếp của bộ mã hóa là nút hồi tiếp có cổng (GRU), danh sách state chỉ chứa một phần tử, đó là trạng thái ẩn với kích thước (số tầng ẩn, kích thước batch, số nút ẩn). Nếu LSTM được sử dụng thì danh sách state sẽ chứa thêm một phần tử khác, đó là ô nhớ với cùng kích thước.

```

encoder = Seq2SeqEncoder(vocab_size=10, embed_size=8, num_hiddens=16,
                         num_layers=2)
encoder.initialize()
X = np.zeros((4, 7))
output, state = encoder(X)
output.shape

```

Trong trường hợp này, vì LSTM đang được sử dụng, danh sách state sẽ chứa cả trạng thái ẩn và ô nhớ với cùng kích thước (số tầng ẩn, kích thước batch, số nút ẩn).

```
len(state), state[0].shape, state[1].shape
```

### 11.7.2 Bộ giải mã

Như đã giới thiệu, vector ngữ cảnh  $\mathbf{c}$  mã hóa thông tin của toàn bộ chuỗi đầu vào  $x_1, \dots, x_T$ . Giả sử đầu ra của tập huấn luyện là  $y_1, \dots, y_{T'}$ . Tại mỗi bước thời gian  $t'$ , xác suất có điều kiện của đầu ra  $y_{t'}$  sẽ phụ thuộc vào đầu ra trước đó  $y_1, \dots, y_{t'-1}$  và vector ngữ cảnh  $\mathbf{c}$ , tức

$$P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}). \quad (11.7.3)$$

Do đó, chúng ta có thể sử dụng một mạng RNN khác trong bộ giải mã. Tại mỗi bước thời gian  $t'$ , bộ giải mã cập nhật trạng thái ẩn của nó thông qua ba đầu vào: vector đặc trưng  $\mathbf{y}_{t'-1}$  của  $y_{t'-1}$ , vector ngữ cảnh  $\mathbf{c}$  và trạng thái ẩn tại bước thời gian trước đó  $\mathbf{s}_{t'-1}$ . Hàm  $g$  dưới đây biểu diễn quá trình biến đổi trạng thái ẩn của mạng RNN trong bộ giải mã:

$$\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}, \mathbf{s}_{t'-1}). \quad (11.7.4)$$

Khi lập trình, ta sử dụng trực tiếp trạng thái ẩn của bộ mã hóa ở bước thời gian cuối cùng để khởi tạo trạng thái ẩn của bộ giải mã. Điều này đòi hỏi bộ mã hóa và bộ giải mã phải có cùng số tầng và số nút ẩn. Các bước tính toán lượt truyền xuôi trong bộ giải mã gần giống trong bộ mã hóa. Điểm khác biệt duy nhất là có thêm một tầng kết nối dày đặc với kích thước bằng kích thước bộ từ vựng được đặt ở sau các tầng LSTM. Tầng này sẽ dự đoán điểm tin cậy cho mỗi từ.

```

# Saved in the d2l package for later use
class Seq2SeqDecoder(d2l.Decoder):
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                 dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)
        self.dense = nn.Dense(vocab_size, flatten=False)

    def init_state(self, enc_outputs, *args):
        return enc_outputs[1]

    def forward(self, X, state):
        X = self.embedding(X).swapaxes(0, 1)
        out, state = self.rnn(X, state)
        # Make the batch to be the first dimension to simplify loss
        # computation
        out = self.dense(out).swapaxes(0, 1)
        return out, state

```

Ta tạo bộ giải mã với cùng các siêu tham số như ở bộ mã hóa. Có thể thấy kích thước đầu ra được thay đổi thành (kích thước batch, độ dài chuỗi, kích thước bộ từ vựng).

```
decoder = Seq2SeqDecoder(vocab_size=10, embed_size=8,
                           num_hiddens=16, num_layers=2)
decoder.initialize()
state = decoder.init_state(encoder(X))
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, state[1].shape
```

### 11.7.3 Hàm Mất mát

Tại mỗi bước thời gian, bộ giải mã tạo ra một vector điểm tin cậy có kích thước bằng bộ từ vựng để dự đoán các từ. Tương tự như trong mô hình hóa ngôn ngữ, ta có thể áp dụng softmax để tính xác suất và sau đó sử dụng hàm mất mát entropy chéo để tính mất mát. Lưu ý rằng ta đã đếm các câu đích để chúng có cùng độ dài, nhưng không cần tính mất mát trên các ký tự đếm này.

Để lập trình hàm mất mát có khả năng lọc ra một số phần tử, ta sẽ sử dụng một toán tử gọi là SequenceMask. Nó có thể gán mặt nạ cho chiều thứ nhất (axis=0) hoặc thứ hai (axis=1). Nếu chiều thứ hai được chọn, với đầu vào là mảng hai chiều X và vector độ dài hợp lệ len, toán tử này sẽ gán  $X[i, \text{len}[i]:] = 0$  với mọi  $i$ .

```
X = np.array([[1, 2, 3], [4, 5, 6]])
npx.sequence_mask(X, np.array([1, 2]), True, axis=1)
```

Áp dụng vào tensor  $n$ -chiều  $X$ , toán tử sẽ gán  $X[i, \text{len}[i]:, :, \dots, :] = 0$ . Ta cũng có thể đặt giá trị mặt nạ khác, ví dụ như  $-1$  dưới đây.

```
X = np.ones((2, 3, 4))
npx.sequence_mask(X, np.array([1, 2]), True, value=-1, axis=1)
```

Bây giờ ta có thể lập trình phiên bản có mặt nạ của hàm mất mát entropy chéo softmax. Lưu ý rằng hàm mất mát trong Gluon cho phép đặt trọng số cho mỗi mẫu, theo mặc định thì giá trị này bằng 1. Để loại bỏ một vài mẫu nhất định, ta có thể đặt trọng số cho chúng bằng 0. Vì vậy, hàm mất mát có mặt nạ sẽ có thêm đối số valid\_len cho toán tử SequenceMask để gán giá trị 0 cho trọng số của các mẫu ta muốn loại bỏ.

```
# Saved in the d2l package for later use
class MaskedSoftmaxCELoss(gluon.loss.SoftmaxCELoss):
    # pred shape: (batch_size, seq_len, vocab_size)
    # label shape: (batch_size, seq_len)
    # valid_len shape: (batch_size, )
    def forward(self, pred, label, valid_len):
        # weights shape: (batch_size, seq_len, 1)
        weights = np.expand_dims(np.ones_like(label), axis=-1)
        weights = npx.sequence_mask(weights, valid_len, True, axis=1)
        return super(MaskedSoftmaxCELoss, self).forward(pred, label, weights)
```

Để kiểm tra sơ bộ, ta tạo ba chuỗi giống hệt nhau, giữ 4 phần tử cho chuỗi thứ nhất, 2 phần tử cho chuỗi thứ hai và không phần tử nào cho chuỗi cuối cùng. Khi đó, giá trị mất mát của chuỗi đầu tiên phải lớn gấp 2 lần so với chuỗi thứ hai, còn giá trị mất mát của chuỗi cuối cùng phải bằng 0.

```
loss = MaskedSoftmaxCELoss()
loss(np.ones((3, 4, 10)), np.ones((3, 4)), np.array([4, 2, 0]))
```

#### 11.7.4 Huấn luyện

Trong quá trình huấn luyện, nếu chuỗi đích có độ dài  $n$ , ta sẽ đưa  $n - 1$  token đầu tiên làm đầu vào bộ giải mã, còn  $n - 1$  token cuối cùng sẽ được sử dụng làm nhãn gốc.

```
# Saved in the d2l package for later use
def train_s2s_ch9(model, data_iter, lr, num_epochs, ctx):
    model.initialize(init.Xavier(), force_reinit=True, ctx=ctx)
    trainer = gluon.Trainer(model.collect_params(),
                            'adam', {'learning_rate': lr})
    loss = MaskedSoftmaxCELoss()
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs], ylim=[0, 0.25])
    for epoch in range(1, num_epochs + 1):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2) # loss_sum, num_tokens
        for batch in data_iter:
            X, X_vlen, Y, Y_vlen = [x.as_in_ctx(ctx) for x in batch]
            Y_input, Y_label, Y_vlen = Y[:, :-1], Y[:, 1:], Y_vlen-1
            with autograd.record():
                Y_hat, _ = model(X, Y_input, X_vlen, Y_vlen)
                l = loss(Y_hat, Y_label, Y_vlen)
            l.backward()
            d2l.grad_clipping(model, 1)
            num_tokens = Y_vlen.sum()
            trainer.step(num_tokens)
            metric.add(l.sum(), num_tokens)
        if epoch % 10 == 0:
            animator.add(epoch, (metric[0]/metric[1],))
    print('loss %.3f, %d tokens/sec on %s' %
          (metric[0]/metric[1], metric[1]/timer.stop(), ctx))
```

Tiếp theo, ta tạo một thực thể của mô hình, đặt các siêu tham số rồi huấn luyện.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, ctx = 0.005, 300, d2l.try_gpu()

src_vocab, tgt_vocab, train_iter = 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)
model = d2l.EncoderDecoder(encoder, decoder)
train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)
```

### 11.7.5 Dự đoán

Ở đây, ta lập trình phương pháp đơn giản nhất có tên gọi *tìm kiếm tham lam (greedy search)*, để tạo chuỗi đầu ra. Như minh họa trong Fig. 11.7.3, trong quá trình dự đoán, ta cũng đưa token bắt đầu câu “`<bos>`” vào bộ giải mã tại bước thời gian 0 giống quá trình huấn luyện. Token đầu vào cho các bước thời gian sau sẽ là token được dự đoán từ bước thời gian trước nó.

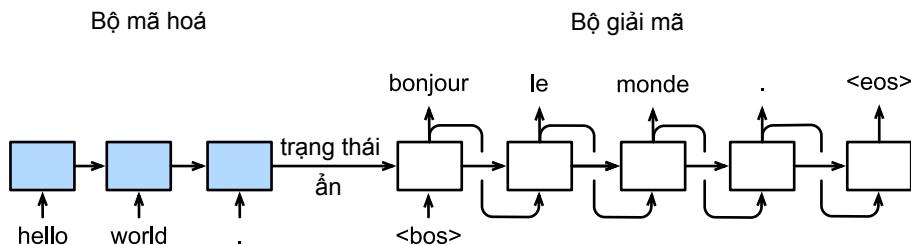


Fig. 11.7.3: Quá trình dự đoán của mô hình chuỗi sang chuỗi sử dụng tìm kiếm tham lam

```
# Saved in the d2l package for later use
def predict_s2s_ch9(model, src_sentence, src_vocab, tgt_vocab, num_steps,
                    ctx):
    src_tokens = src_vocab[src_sentence.lower().split(' ')]
    enc_valid_len = np.array([len(src_tokens)], ctx=ctx)
    src_tokens = d2l.truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    enc_X = np.array(src_tokens, ctx=ctx)
    # Add the batch_size dimension
    enc_outputs = model.encoder(np.expand_dims(enc_X, axis=0),
                                enc_valid_len)
    dec_state = model.decoder.init_state(enc_outputs, enc_valid_len)
    dec_X = np.expand_dims(np.array([tgt_vocab['<bos>']], ctx=ctx), axis=0)
    predict_tokens = []
    for _ in range(num_steps):
        Y, dec_state = model.decoder(dec_X, dec_state)
        # The token with highest score is used as the next timestep input
        dec_X = Y.argmax(axis=2)
        py = dec_X.squeeze(axis=0).astype('int32').item()
        if py == tgt_vocab['<eos>']:
            break
        predict_tokens.append(py)
    return ' '.join(tgt_vocab.to_tokens(predict_tokens))
```

Ta sẽ thử một vài ví dụ:

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))
```

### 11.7.6 Tóm tắt

- Mô hình chuỗi sang chuỗi (*sequence to sequence - seq2seq*) dựa trên kiến trúc mã hóa-giải mã để tạo một chuỗi đầu ra từ chuỗi đầu vào.
- Ta sử dụng nhiều tầng LSTM cho cả bộ mã hóa và bộ giải mã.

### 11.7.7 Bài tập

1. Nêu một vài ứng dụng khác của seq2seq ngoài dịch máy.
2. Nếu chuỗi đầu vào trong các ví dụ trên dài hơn thì sao?
3. Điều gì có thể xảy ra nếu không sử dụng SequenceMask trong hàm mất mát?

### 11.7.8 Thảo luận

- Tiếng Anh<sup>214</sup>
- Tiếng Việt<sup>215</sup>

### 11.7.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Đỗ Trường Giang
- Phạm Minh Đức
- Nguyễn Duy Du
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

## 11.8 Tìm kiếm Chùm

Trong Section 11.7, chúng ta đã thảo luận cách huấn luyện mô hình mã hóa - giải mã với đầu vào và đầu ra có độ dài thay đổi. Phần này giới thiệu cách sử dụng mô hình để dự đoán đầu ra là chuỗi có độ dài thay đổi.

Trong Section 11.5, khi chuẩn bị dữ liệu huấn luyện, ta thường thêm ký hiệu kết thúc câu “<eos>” vào sau mỗi câu. Ta sẽ tiếp tục sử dụng ký hiệu trên trong phần này. Để thuận tiện, giả sử rằng đầu ra của bộ giải mã là một chuỗi văn bản. Gọi kích thước của bộ từ điển đầu ra  $\mathcal{Y}$  (chứa tất cả các từ có thể xuất hiện ở chuỗi đầu ra, bao gồm cả “<eos>”) là  $|\mathcal{Y}|$ , và chiều dài tối đa của chuỗi đầu ra là

<sup>214</sup> <https://discuss.mxnet.io/t/4357>

<sup>215</sup> [https://forum.machinelarningcovan.com/c/d21](https://forum.machinelearningcovan.com/c/d21)

$T'$ . Như vậy có tổng cộng  $\mathcal{O}(|\mathcal{Y}|^{T'})$  chuỗi đầu ra có thể được sinh ra. Tất cả những chuỗi con nằm phía sau “`<eos>`” trong chuỗi đầu ra sẽ bị lược bỏ. Bên cạnh đó, ta ký hiệu  $\mathbf{c}$  là vector ngữ cảnh mã hóa thông tin của tất cả trạng thái ẩn từ đầu vào.

### 11.8.1 Tìm kiếm Tham lam

Đầu tiên, hãy xem xét một phương pháp đơn giản: tìm kiếm tham lam. Tại mỗi bước thời gian  $t'$  của chuỗi đầu ra, ta chọn từ có xác suất có điều kiện cao nhất trong  $|\mathcal{Y}|$  từ làm đầu ra như sau:

$$y_{t'} = \operatorname{argmax}_{y \in \mathcal{Y}} P(y \mid y_1, \dots, y_{t'-1}, \mathbf{c}) \quad (11.8.1)$$

Khi gặp “`<eos>`” hoặc khi chuỗi đầu ra đạt chiều dài tối đa  $T'$ , ta kết thúc việc dự đoán.

Như đã đề cập khi thảo luận về bộ giải mã, xác suất có điều kiện của một chuỗi đầu ra được sinh từ chuỗi đầu vào là  $\prod_{t'=1}^{T'} P(y_{t'} \mid y_1, \dots, y_{t'-1}, \mathbf{c})$ . Chuỗi đầu ra tối ưu là chuỗi có xác suất có điều kiện cao nhất. Vấn đề lớn nhất của tìm kiếm tham lam là không đảm bảo chuỗi tìm được là chuỗi tối ưu.

Xét ví dụ dưới đây. Giả sử ta có bốn từ “A”, “B”, “C”, và “`<eos>`” trong từ điển đầu ra. Bốn giá trị dưới mỗi bước thời gian trong Fig. 11.8.1 là xác suất có điều kiện của “A”, “B”, “C”, và “`<eos>`” tại bước thời gian đó. Tại mỗi bước thời gian, tìm kiếm tham lam chọn từ có xác suất có điều kiện cao nhất. Vì vậy, chuỗi đầu ra “A”, “B”, “C”, và “`<eos>`” được tạo ra như trong Fig. 11.8.1. Xác suất có điều kiện của cả chuỗi đầu ra này là  $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$ .

Bước thời gian	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<code>&lt;eos&gt;</code>	0.1	0.2	0.2	0.6

Fig. 11.8.1: Dưới mỗi bước thời gian là xác suất có điều kiện của “A”, “B”, “C”, và “`<eos>`” tại bước thời gian đó. Tại mỗi bước thời gian, phương pháp tìm kiếm tham lam sẽ chọn từ có xác suất cao nhất.

Bây giờ, hãy xét một ví dụ khác trong Fig. 11.8.2. Khác với Fig. 11.8.1, tại bước thời gian 2 ta chọn “C”, từ có xác suất có điều kiện cao thứ hai. Vì bước thời gian 3 phụ thuộc vào bước thời gian 1 và 2, mà chuỗi con đầu ra tại hai bước thời gian này thay đổi từ “A” và “B” trong Fig. 11.8.1 thành “A” và “C” trong Fig. 11.8.2, nên xác suất có điều kiện của các từ tại bước thời gian 3 cũng thay đổi. Chúng ta chọn “B”, từ có xác suất có điều kiện cao nhất. Bây giờ, chuỗi con đầu ra trước bước thời gian 4 là “A”, “C”, và “B”, khác với “A”, “B”, và “C” trong Fig. 11.8.1. Do đó xác suất có điều kiện của các từ tại bước thời gian 4 cũng thay đổi. Vẫn chọn từ có xác suất cao nhất tại bước thời gian này là “`<eos>`”, ta có xác suất có điều kiện của cả chuỗi đầu ra “A”, “C”, “B”, và “`<eos>`” là  $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$ , cao hơn xác suất của chuỗi được sinh ra dựa trên phương pháp tìm kiếm tham lam. Vì vậy, chuỗi đầu ra “A”, “B”, “C”, và “`<eos>`” có được từ phương pháp tìm kiếm tham lam không phải chuỗi tối ưu.

Bước thời gian	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Fig. 11.8.2: Dưới mỗi bước thời gian là xác suất có điều kiện của “A”, “B”, “C”, và “<eos>” tại bước thời gian đó. Tại bước thời gian 2, từ “C” được chọn có xác suất có điều kiện cao thứ hai.

### 11.8.2 Tìm kiếm Vết cạn

Nếu mục tiêu là tìm được chuỗi tối ưu, ta có thể xem xét giải thuật vết cạn: kiểm tra tất cả những chuỗi đầu ra có thể, trả kết quả là chuỗi có xác suất có điều kiện cao nhất.

Mặc dù chúng ta có thể sử dụng thuật toán tìm kiếm vết cạn để tìm chuỗi tối ưu, nhưng chi phí tính toán của nó  $\mathcal{O}(|\mathcal{Y}|^{T'})$  là quá cao. Ví dụ, khi  $|\mathcal{Y}| = 10000$  và  $T' = 10$ , chúng ta cần kiểm tra  $10000^{10} = 10^{40}$  chuỗi. Điều này gần như là bất khả thi. Chi phí tính toán của tìm kiếm tham lam là  $\mathcal{O}(|\mathcal{Y}| T')$ , ít hơn nhiều so với vết cạn. Ví dụ, khi  $|\mathcal{Y}| = 10000$  và  $T' = 10$ , chúng ta chỉ cần kiểm tra  $10000 \times 10 = 1 \times 10^5$  chuỗi.

### 11.8.3 Tìm kiếm Chùm

*Tìm kiếm chùm* (*beam search*) là một thuật toán cải tiến dựa trên tìm kiếm tham lam. Nó có một siêu tham số  $k$  gọi là *kích thước chùm* (*beam size*). Tại bước thời gian 1, ta chọn  $k$  từ có xác suất có điều kiện cao nhất để bắt đầu  $k$  chuỗi đầu ra ứng viên. Tại các bước thời gian tiếp theo, dựa trên  $k$  chuỗi đầu ra ứng viên từ bước thời gian trước đó, ta tính và chọn  $k$  chuỗi có xác suất có điều kiện cao nhất trong tổng số  $k |\mathcal{Y}|$  khả năng. Đây sẽ là các chuỗi đầu ra ứng viên cho bước thời gian đó. Cuối cùng, ta lọc ra các chuỗi có chứa “<eos>” từ các chuỗi đầu ra ứng viên tại mỗi bước thời gian và loại bỏ tất cả các chuỗi sau ký tự đó để thu được tập các chuỗi đầu ra ứng viên cuối cùng.

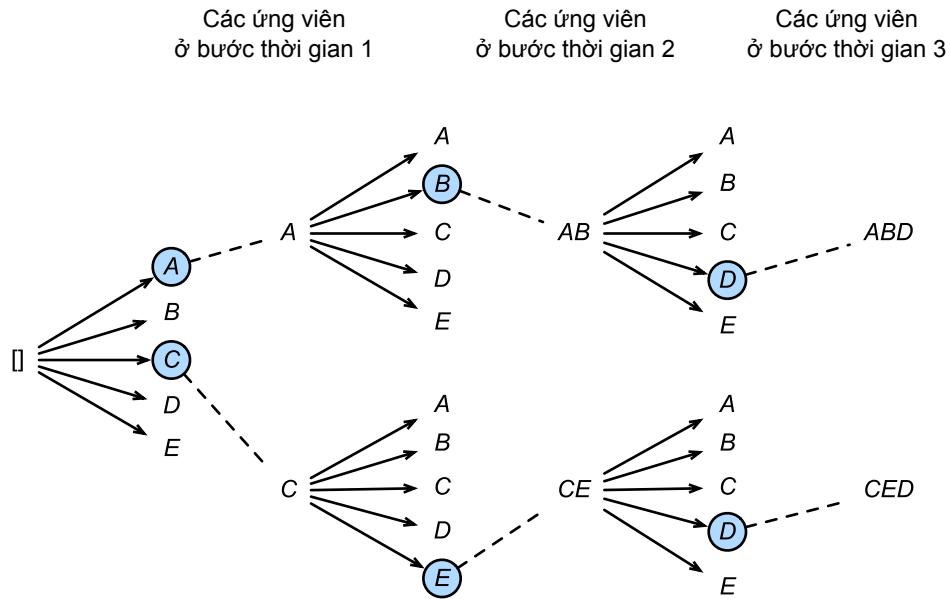


Fig. 11.8.3: Quá trình tìm kiếm chùm. Kích thước chùm bằng 2 và độ dài tối đa của chuỗi đầu ra bằng 3. Các chuỗi đầu ra ứng viên là  $A, C, AB, CE, ABD$ , và  $CED$ .

Fig. 11.8.3 minh họa một ví dụ cho quá trình tìm kiếm chùm. Giả sử bộ từ vựng của chuỗi đầu ra chỉ chứa năm từ:  $\mathcal{Y} = \{A, B, C, D, E\}$  và một trong số chúng là ký hiệu đặc biệt “`<eos>`”. Đặt kích thước chùm bằng 2 và độ dài tối đa của chuỗi đầu ra bằng 3. Tại bước thời gian 1 của chuỗi đầu ra, giả sử các từ có xác suất có điều kiện  $P(y_1 | \mathbf{c})$  cao nhất là  $A$  và  $C$ . Tại bước thời gian 2, với mọi  $y_2 \in \mathcal{Y}$ , ta tính

$$P(A, y_2 | \mathbf{c}) = P(A | \mathbf{c})P(y_2 | A, \mathbf{c}) \quad (11.8.2)$$

và

$$P(C, y_2 | \mathbf{c}) = P(C | \mathbf{c})P(y_2 | C, \mathbf{c}), \quad (11.8.3)$$

và chọn hai giá trị cao nhất trong 10 giá trị này, giả sử đó là

$$P(A, B | \mathbf{c}) \text{ và } P(C, E | \mathbf{c}). \quad (11.8.4)$$

Sau đó, tại bước thời gian 3, với mọi  $y_3 \in \mathcal{Y}$ , ta tính

$$P(A, B, y_3 | \mathbf{c}) = P(A, B | \mathbf{c})P(y_3 | A, B, \mathbf{c}) \quad (11.8.5)$$

và

$$P(C, E, y_3 | \mathbf{c}) = P(C, E | \mathbf{c})P(y_3 | C, E, \mathbf{c}), \quad (11.8.6)$$

và chọn hai giá trị cao nhất trong số 10 giá trị này, giả sử đó là

$$P(A, B, D | \mathbf{c}) \text{ và } P(C, E, D | \mathbf{c}). \quad (11.8.7)$$

Kết quả là, ta thu được 6 chuỗi đầu ra ứng viên: (1)  $A$ ; (2)  $C$ ; (3)  $A, B$ ; (4)  $C, E$ ; (5)  $A, B, D$ ; và (6)  $C, E, D$ . Cuối cùng, ta sẽ có một tập chuỗi đầu ra ứng viên cuối cùng dựa trên 6 chuỗi này.

Trong tập các chuỗi đầu ra ứng viên cuối cùng, ta sẽ lấy chuỗi có điểm số cao nhất làm chuỗi đầu ra. Điểm số cho mỗi chuỗi được tính như sau:

$$\frac{1}{L^\alpha} \log P(y_1, \dots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^L \log P(y_{t'} | y_1, \dots, y_{t'-1}, \mathbf{c}), \quad (11.8.8)$$

Ở đây,  $L$  là độ dài của chuỗi ứng viên cuối cùng và  $\alpha$  thường được đặt bằng 0.75.  $L^\alpha$  trong mẫu số là lượng phạt lên tổng logarit cho các chuỗi dài. Có thể ước tính rằng chi phí tính toán của tìm kiếm chùm là  $\mathcal{O}(k|\mathcal{Y}|T')$ . Nó nằm trong khoảng giữa chi phí tính toán của tìm kiếm tham lam và tìm kiếm vét cạn. Ngoài ra, tìm kiếm tham lam có thể được coi là tìm kiếm chùm với kích thước chùm bằng 1. Tìm kiếm chùm tạo ra sự cân bằng giữa chi phí tính toán và chất lượng tìm kiếm bằng cách sử dụng linh hoạt kích thước chùm  $k$ .

#### 11.8.4 Tóm tắt

- Các phương pháp dự đoán chuỗi có độ dài thay đổi bao gồm tìm kiếm tham lam, tìm kiếm vét cạn và tìm kiếm chùm.
- Tìm kiếm chùm tạo ra sự cân bằng giữa chi phí tính toán và chất lượng tìm kiếm bằng cách sử dụng linh hoạt kích thước chùm.

#### 11.8.5 Bài tập

1. Ta có thể coi tìm kiếm vét cạn là tìm kiếm chùm với kích thước chùm đặc biệt không? Tại sao?
2. Ta đã sử dụng các mô hình ngôn ngữ để tạo các câu trong Section 10.5. Các mô hình này sử dụng phương pháp tìm kiếm đầu ra nào? Hãy cải thiện các phương pháp đó.

#### 11.8.6 Thảo luận

- [Tiếng Anh](#)<sup>216</sup>
- [Tiếng Việt](#)<sup>217</sup>

---

<sup>216</sup> <https://discuss.mxnet.io/t/2394>

<sup>217</sup> <https://forum.machinelearningcoban.com/c/d21>

### **11.8.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Đình Nam
- Nguyễn Duy Du
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

### **11.9 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Phạm Minh Đức
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc



# 12 | Cơ chế Tập trung

Tản mạn một chút về lịch sử khởi nguồn, sự tập trung là một lĩnh vực nghiên cứu rộng lớn và lâu đời trong ngành thần kinh học nhận thức. Trọng tâm ở đây có thể hiểu rằng sự tập trung của ý thức chính là bản chất của sự chú ý, điều này cho phép chúng ta (loài người) ưu tiên năng lực tri giác để giải quyết hiệu quả những sự kiện xoay quanh mình. Kết quả là ta không xử lý toàn bộ những thông tin thu được từ các giác quan. Tại một thời điểm, chúng ta chỉ có thể tiếp nhận một lượng nhỏ thông tin từ môi trường. Trong ngành thần kinh học nhận thức, có tồn tại một vài dạng tập trung khác nhau như cơ chế tập trung có chọn lọc, tập trung ngầm, và tập trung về không gian. Thuyết tập trung mà được lấy làm nguồn cảm hứng trong lĩnh vực học sâu gần đây đó là *thuyết tích hợp đặc trưng* (*feature integration theory*) trong cơ chế tập trung có chọn lọc được phát triển bởi Anne Treisman và Garry Gelade trong ([Treisman & Gelade, 1980](#)) vào năm 1980. Bài báo này phát biểu rằng khi có kích thích thị giác, các đặc trưng sớm được tiếp nhận một cách tự động và đồng thời, trong khi các sự vật sẽ được xác định riêng biệt ở pha tiếp theo trong chu trình xử lý. Lý thuyết này trở thành một trong những mô hình tâm lý học về cơ chế tập trung thị giác của con người có nhiều ảnh hưởng nhất.

Tuy nhiên, ta không đi sâu vào thuyết tập trung trong ngành thần kinh học mà sẽ tìm hiểu cách đưa ý tưởng của cơ chế tập trung vào học sâu. Ở đây, cơ chế tập trung có thể được xem là phép gộp tổng quát theo trọng số trên mỗi giá trị đầu vào. Trong chương này, chúng tôi sẽ giúp bạn hình dung cách biến ý tưởng của cơ chế tập trung thành các mô hình toán học cụ thể có thể hoạt động được.

## 12.1 Cơ chế Tập trung

Trong [Section 11.7](#), chúng ta dùng mạng hồi tiếp để mã hóa thông tin của chuỗi nguồn đầu vào thành trạng thái ẩn và truyền nó tới bộ giải mã để sinh chuỗi đích. Một token trong chuỗi đích có thể chỉ liên quan mật thiết tới một vài token chứ không nhất thiết là toàn bộ token trong chuỗi nguồn. Ví dụ, khi dịch “Hello world.” thành “Bonjour le monde.”, từ “Bonjour” ánh xạ tới từ “Hello” và từ “monde” ánh xạ tới từ “world”. Trong mô hình seq2seq, bộ giải mã có thể ngầm chọn thông tin tương ứng từ trạng thái ẩn được truyền đến từ bộ mã hóa. Tuy nhiên, cơ chế tập trung (*attention mechanism*) thực hiện phép chọn này một cách tường minh.

Cơ chế *tập trung* có thể được coi là phép gộp tổng quát. Nó gộp đầu vào dựa trên các trọng số khác nhau. Thành phần cốt lõi của cơ chế tập trung là tầng tập trung. Đầu vào của tầng tập trung được gọi ngắn gọn là *câu truy vấn* (*query*). Với mỗi câu truy vấn, tầng tập trung trả về đầu ra dựa trên bộ nhớ là tập các cặp khóa-giá trị được mã hóa trong tầng tập trung này. Cụ thể, giả sử bộ nhớ chứa  $n$  cặp vector khóa-giá trị,  $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_n, \mathbf{v}_n)$ , với  $\mathbf{k}_i \in \mathbb{R}^{d_k}$ ,  $\mathbf{v}_i \in \mathbb{R}^{d_v}$ . Với mỗi vector truy vấn  $\mathbf{q} \in \mathbb{R}^{d_q}$ , tầng tập trung trả về đầu ra  $\mathbf{o} \in \mathbb{R}^{d_v}$  có cùng kích thước với vector giá trị.

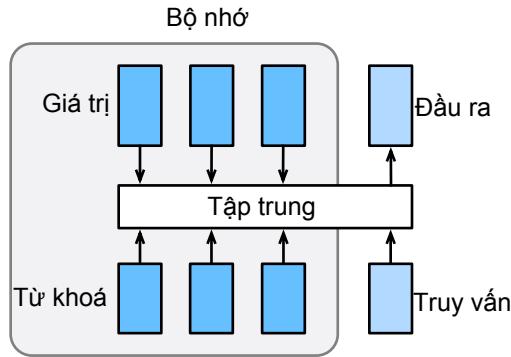


Fig. 12.1.1: Tầng tập trung trả về giá trị dựa trên câu truy vấn đầu vào và bộ nhớ của nó.

Chi tiết về cơ chế tập trung được minh họa trong Fig. 12.1.2. Để tính toán đầu ra của tầng tập trung, chúng ta sử dụng hàm tính điểm  $\alpha$  để đo độ tương đồng giữa câu truy vấn và các khóa. Sau đó, với mỗi khóa  $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_n, \mathbf{v}_n)$ , ta tính điểm  $a_1, \dots, a_n$  như sau:

$$a_i = \alpha(\mathbf{q}, \mathbf{k}_i). \quad (12.1.1)$$

Tiếp theo, chúng ta sử dụng hàm softmax để thu được các trọng số tập trung (*attention weights*), cụ thể:

$$\mathbf{b} = \text{softmax}(\mathbf{a}) \quad \text{trong đó} \quad b_i = \frac{\exp(a_i)}{\sum_j \exp(a_j)}, \mathbf{b} = [b_1, \dots, b_n]^T. \quad (12.1.2)$$

Cuối cùng, đầu ra của tầng là tổng trọng số của các giá trị:

$$\mathbf{o} = \sum_{i=1}^n b_i \mathbf{v}_i. \quad (12.1.3)$$

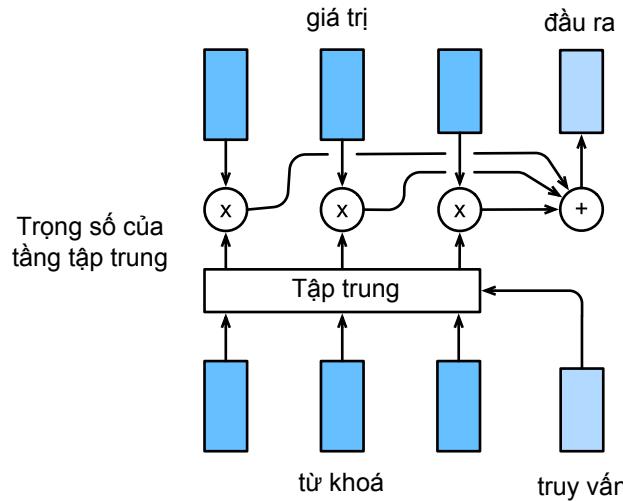


Fig. 12.1.2: Đầu ra của tầng tập trung là tổng trọng số của các giá trị.

Cách lựa chọn hàm tính điểm khác nhau sẽ tạo ra các tầng tập trung khác nhau. Ở dưới đây chúng tôi sẽ trình bày hai tầng tập trung thường hay được sử dụng. Đầu tiên chúng tôi giới thiệu hai toán

từ cần thiết để lập trình hai tầng này: toán tử softmax có mặt nạ masked\_softmax và toán tử tích vô hướng chuyên biệt theo batched\_dot.

```
import math
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()
```

Toán tử softmax có mặt nạ nhận đầu vào là một tensor 3 chiều và cho phép ta lọc ra một số phần tử bằng cách xác định độ dài hợp lệ cho chiều cuối cùng. (Tham khảo Section 11.5 về định nghĩa của độ dài hợp lệ). Do đó, những giá trị nằm ngoài độ dài hợp lệ sẽ được gán bằng 0. Chúng ta lập trình hàm masked\_softmax như sau.

```
# Saved in the d2l package for later use
def masked_softmax(X, valid_len):
    # X: 3-D tensor, valid_len: 1-D or 2-D tensor
    if valid_len is None:
        return npx.softmax(X)
    else:
        shape = X.shape
        if valid_len.ndim == 1:
            valid_len = valid_len.repeat(shape[1], axis=0)
        else:
            valid_len = valid_len.reshape(-1)
        # Fill masked elements with a large negative, whose exp is 0
        X = npx.sequence_mask(X.reshape(-1, shape[-1]), valid_len, True,
                              axis=1, value=-1e6)
    return npx.softmax(X).reshape(shape)
```

Để minh họa cách hàm trên hoạt động, chúng ta hãy khởi tạo hai ma trận đầu vào kích thước là  $2 \times 4$ . Bên cạnh đó, chúng ta sẽ gán độ dài hợp lệ cho mẫu thứ nhất là 2 và mẫu thứ hai là 3. Từ đó, những giá trị đầu ra nằm ngoài độ dài hợp lệ sẽ được gán bằng 0 như dưới đây.

```
masked_softmax(np.random.uniform(size=(2, 2, 4)), np.array([2, 3]))
```

Ngoài ra, toán tử thứ hai batched\_dot nhận hai đầu vào là  $X$  và  $Y$  có kích thước lần lượt là  $(b, n, m)$  và  $(b, m, k)$ , và trả về đầu ra có kích thước là  $(b, n, k)$ . Cụ thể, toán tử này tính  $b$  tích vô hướng với  $i = \{1, \dots, b\}$  như sau:

$$Z[i, :, :] = X[i, :, :]Y[i, :, :]. \quad (12.1.4)$$

```
npx.batch_dot(np.ones((2, 1, 3)), np.ones((2, 3, 2)))
```

### 12.1.1 Tầng Tập trung Tích Vô hướng

Với hai toán tử `masked_softmax` và `batched_dot` ở trên, chúng ta sẽ đi vào chi tiết hai loại tầng tập trung được sử dụng phổ biến. Loại đầu tiên là *tập trung tích vô hướng* (*dot product attention*): nó già định rằng câu truy vấn có cùng kích thước chiều với khóa, cụ thể là  $\mathbf{q}, \mathbf{k}_i \in \mathbb{R}^d$  với mọi  $i$ . Tầng tập trung tích vô hướng sẽ tính điểm bằng cách lấy tích vô hướng giữa câu truy vấn và khóa, sau đó chia cho  $\sqrt{d}$  để giảm thiểu ảnh hưởng không liên quan của số chiều  $d$  lên điểm số. Nói cách khác,

$$\alpha(\mathbf{q}, \mathbf{k}) = \langle \mathbf{q}, \mathbf{k} \rangle / \sqrt{d}. \quad (12.1.5)$$

Mở rộng ra từ các câu truy vấn và khóa một chiều, chúng ta luôn có thể tổng quát hóa chúng lên thành các giá trị truy vấn và khóa đa chiều. Giả định rằng  $\mathbf{Q} \in \mathbb{R}^{m \times d}$  chứa  $m$  câu truy vấn và  $\mathbf{K} \in \mathbb{R}^{n \times d}$  chứa toàn bộ  $n$  khóa. Chúng ta có thể tính toàn bộ  $mn$  điểm số như sau

$$\alpha(\mathbf{Q}, \mathbf{K}) = \mathbf{Q}\mathbf{K}^\top / \sqrt{d}. \quad (12.1.6)$$

Với (12.1.6), chúng ta có thể lập trình tầng tập trung tích vô hướng `DotProductAttention` hỗ trợ một batch các câu truy vấn và các cặp khóa-giá trị. Ngoài ra, chúng ta cũng dùng thêm một tầng `dropout` để điều chuẩn.

```
# Saved in the d2l package for later use
class DotProductAttention(nn.Block):
    def __init__(self, dropout, **kwargs):
        super(DotProductAttention, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)

    # query: (batch_size, #queries, d)
    # key: (batch_size, #kv_pairs, d)
    # value: (batch_size, #kv_pairs, dim_v)
    # valid_len: either (batch_size, ) or (batch_size, xx)
    def forward(self, query, key, value, valid_len=None):
        d = query.shape[-1]
        # Set transpose_b=True to swap the last two dimensions of key
        scores = npx.batch_dot(query, key, transpose_b=True) / math.sqrt(d)
        attention_weights = self.dropout(masked_softmax(scores, valid_len))
        return npx.batch_dot(attention_weights, value)
```

Hãy kiểm tra lớp `DotProductAttention` với một ví dụ nhỏ sau. Đầu tiên ta tạo 2 batch, mỗi batch có 1 câu truy vấn và 10 cặp khóa-giá trị. Thông qua đối số `valid_len`, ta chỉ định rằng ta sẽ kiểm tra 2 cặp khóa-giá trị đầu tiên cho batch đầu tiên và 6 cặp cho batch thứ hai. Do đó, mặc dù cả hai batch đều có cùng câu truy vấn và các cặp khóa-giá trị, chúng ta sẽ thu được các đầu ra khác nhau.

```
atten = DotProductAttention(dropout=0.5)
atten.initialize()
keys = np.ones((2, 10, 2))
values = np.arange(40).reshape(1, 10, 4).repeat(2, axis=0)
atten(np.ones((2, 1, 2)), keys, values, np.array([2, 6]))
```

Như đã thấy ở trên, tập trung tích vô hướng chỉ đơn thuần nhân câu truy vấn và khóa lại với nhau, hi vọng rằng từ đó thu được những điểm tương đồng giữa chúng. Tuy nhiên, câu truy vấn và khóa có thể không có cùng kích thước chiều. Để giải quyết vấn đề này, chúng ta cần nhờ đến cơ chế tập trung perceptron đa tầng.

### 12.1.2 Tập trung Perceptron Đa tầng

Trong cơ chế *tập trung perceptron đa tầng* (*multilayer perceptron attention*), chúng ta chiếu cả câu truy vấn và các khóa lên  $\mathbb{R}^h$  bằng cách tham số trọng số được học. Giả định rằng các trọng số được học là  $\mathbf{W}_k \in \mathbb{R}^{h \times d_k}$ ,  $\mathbf{W}_q \in \mathbb{R}^{h \times d_q}$  và  $\mathbf{v} \in \mathbb{R}^h$ . Hàm tính điểm sẽ được định nghĩa như sau

$$\alpha(\mathbf{k}, \mathbf{q}) = \mathbf{v}^\top \tanh(\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q}). \quad (12.1.7)$$

Một cách trực quan, ta có thể tưởng tượng  $\mathbf{W}_k \mathbf{k} + \mathbf{W}_q \mathbf{q}$  chính là việc nối khóa và giá trị lại với nhau theo chiều đặc trưng và đưa chúng qua perceptron có một tầng ẩn với kích thước là  $h$  và tầng đầu ra với kích thước là 1. Trong tầng ẩn này, hàm kích hoạt là *tanh* và không có hệ số điều chỉnh. Giờ hãy lập trình một tầng tập trung perceptron đa tầng.

```
# Saved in the d2l package for later use
class MLPAttention(nn.Block):
    def __init__(self, units, dropout, **kwargs):
        super(MLPAttention, self).__init__(**kwargs)
        # Use flatten=True to keep query's and key's 3-D shapes
        self.W_k = nn.Dense(units, activation='tanh',
                            use_bias=False, flatten=False)
        self.W_q = nn.Dense(units, activation='tanh',
                            use_bias=False, flatten=False)
        self.v = nn.Dense(1, use_bias=False, flatten=False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, query, key, value, valid_len):
        query, key = self.W_q(query), self.W_k(key)
        # Expand query to (batch_size, #querys, 1, units), and key to
        # (batch_size, 1, #kv_pairs, units). Then plus them with broadcast
        features = np.expand_dims(query, axis=2) + np.expand_dims(key, axis=1)
        scores = np.squeeze(self.v(features), axis=-1)
        attention_weights = self.dropout(masked_softmax(scores, valid_len))
        return npx.batch_dot(attention_weights, value)
```

Để kiểm tra lớp MLPAttention phía trên, chúng ta sẽ sử dụng lại đầu vào ở ví dụ đơn giản trước. Như ta thấy ở dưới, mặc dù MLPAttention chưa thêm một mô hình MLP, chúng ta vẫn thu được đầu ra tương tự DotProductAttention.

```
atten = MLPAttention(units=8, dropout=0.1)
atten.initialize()
atten(np.ones((2, 1, 2)), keys, values, np.array([2, 6]))
```

### 12.1.3 Tóm tắt

- Tầng tập trung lựa chọn một cách tinh minh các thông tin liên quan.
- Ô nhớ của tầng tập trung chứa các cặp khóa-giá trị, do đó đầu ra của nó gần các giá trị có khóa giống với câu truy vấn.
- Hai mô hình tập trung được sử dụng phổ biến là tập trung tích vô hướng và tập trung perceptron đa tầng.

#### 12.1.4 Bài tập

Ưu và khuyết điểm của tầng tập trung tích vô hướng và tập trung perceptron đa tầng là gì?

#### 12.1.5 Thảo luận

- Tiếng Anh<sup>218</sup>
- Tiếng Việt<sup>219</sup>

#### 12.1.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Cảnh Thưởng
- Nguyễn Văn Cường
- Võ Tấn Phát
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Hồng Vinh

## 12.2 Chuỗi sang Chuỗi áp dụng Cơ chế Tập trung

Trong phần này, chúng ta thêm cơ chế tập trung vào mô hình chuỗi sang chuỗi (seq2seq) giới thiệu trong Section 11.7 để gộp các trạng thái theo trọng số tương ứng một cách tường minh. Fig. 12.2.1 mô tả kiến trúc mô hình thực hiện mã hóa và giải mã tại bước thời gian  $t$ . Bộ nhớ của tầng tập trung ở đây bao gồm tất cả thông tin mà bộ mã hóa đã được học—đầu ra của bộ mã hóa tại từng bước thời gian. Trong quá trình giải mã, đầu ra của bộ giải mã tại bước thời gian trước đó  $t - 1$  được sử dụng làm câu truy vấn. Đầu ra của mô hình tập trung có thể được hiểu là thông tin ngữ cảnh của chuỗi, phần ngữ cảnh này được ghép nối với đầu vào của bộ giải mã  $D_t$  và kết quả được đưa vào bộ giải mã.

<sup>218</sup> <https://discuss.mxnet.io/t/4343>

<sup>219</sup> <https://forum.machinelearningcoban.com/c/d21>

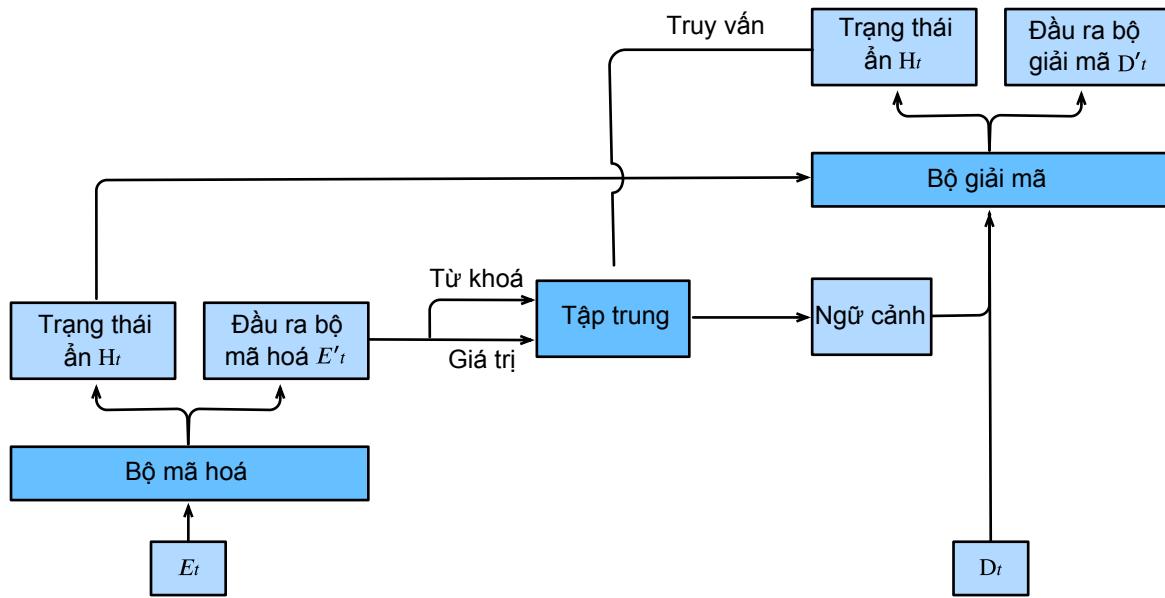


Fig. 12.2.1: Quá trình giải mã tại bước thời gian thứ 2 trong mô hình chuỗi sang chuỗi áp dụng cơ chế tập trung.

Để minh họa kiến trúc tổng thể của mô hình seq2seq áp dụng cơ chế tập trung, cấu trúc tầng của bộ mã hóa và bộ giải mã được mô tả trong Fig. 12.2.2.

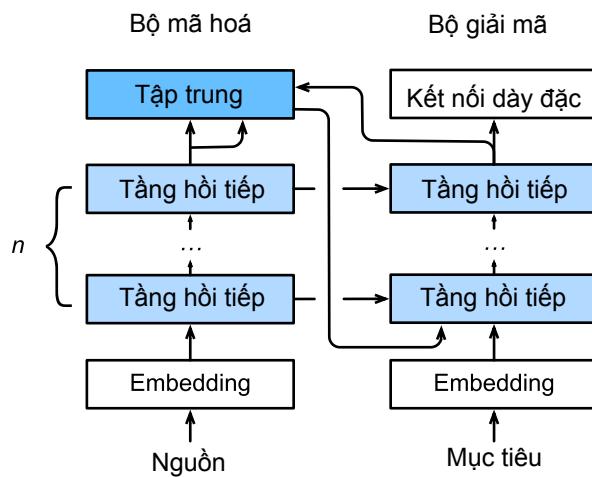


Fig. 12.2.2: Các tầng trong mô hình chuỗi sang chuỗi áp dụng cơ chế tập trung.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import rnn, nn
npx.set_np()
```

### 12.2.1 Bộ Giải mã

Do bộ mã hóa của mô hình seq2seq áp dụng cơ chế tập trung giống với bộ mã hóa của Seq2SeqEncoder trong Section 11.7 nên ở phần này, chúng ta sẽ chỉ tập trung vào bộ giải mã. Ta thêm tầng tập trung MLP (MLPAttention) có cùng kích thước ẩn với tầng LSTM trong bộ giải mã. Sau đó ta khởi tạo trạng thái của bộ giải mã bằng cách truyền vào ba đầu ra thu được từ bộ mã hóa:

- **Đầu ra của bộ mã hóa tại tất cả các bước thời gian:** được sử dụng như bộ nhớ của tầng tập trung có cùng các khóa và giá trị;
- **Trạng thái ẩn của bộ mã hóa tại bước thời gian cuối cùng:** được sử dụng làm trạng thái ẩn ban đầu của bộ giải mã;
- **Độ dài hợp lệ của bộ mã hóa:** để tầng tập trung có thể bỏ qua những token đệm có trong đầu ra của bộ mã hóa.

Ở mỗi bước thời gian trong quá trình giải mã, ta sử dụng trạng thái ẩn của tầng RNN cuối cùng làm câu truy vấn cho tầng tập trung. Đầu ra của mô hình tập trung sau đó được ghép nối với vector embedding đầu vào để đưa vào tầng RNN. Mặc dù trạng thái ẩn của tầng RNN cũng chứa thông tin từ bộ giải mã ở các bước thời gian trước đó nhưng đầu ra của tầng tập trung sẽ lựa chọn các đầu ra của bộ mã hóa một cách tường minh dựa vào enc\_valid\_len nhằm loại bỏ những thông tin không liên quan.

Hãy cùng lập trình bộ giải mã Seq2SeqAttentionDecoder và xem xét sự khác biệt của nó so với bộ giải mã trong mô hình seq2seq ở Section 11.7.2.

```
class Seq2SeqAttentionDecoder(d2l.Decoder):  
    def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,  
                 dropout=0, **kwargs):  
        super(Seq2SeqAttentionDecoder, self).__init__(**kwargs)  
        self.attention_cell = d2l.MLPAttention(num_hiddens, dropout)  
        self.embedding = nn.Embedding(vocab_size, embed_size)  
        self.rnn = rnn.LSTM(num_hiddens, num_layers, dropout=dropout)  
        self.dense = nn.Dense(vocab_size, flatten=False)  
  
    def init_state(self, enc_outputs, enc_valid_len, *args):  
        outputs, hidden_state = enc_outputs  
        # Transpose outputs to (batch_size, seq_len, num_hiddens)  
        return (outputs.swapaxes(0, 1), hidden_state, enc_valid_len)  
  
    def forward(self, X, state):  
        enc_outputs, hidden_state, enc_valid_len = state  
        X = self.embedding(X).swapaxes(0, 1)  
        outputs = []  
        for x in X:  
            # query shape: (batch_size, 1, num_hiddens)  
            query = np.expand_dims(hidden_state[0][-1], axis=1)  
            # context has same shape as query  
            context = self.attention_cell(  
                query, enc_outputs, enc_outputs, enc_valid_len)  
            # Concatenate on the feature dimension  
            x = np.concatenate((context, np.expand_dims(x, axis=1)), axis=-1)  
            # Reshape x to (1, batch_size, embed_size + num_hiddens)  
            out, hidden_state = self.rnn(x.swapaxes(0, 1), hidden_state)  
            outputs.append(out)  
        outputs = self.dense(np.concatenate(outputs, axis=0))
```

(continues on next page)

```
return outputs.swapaxes(0, 1), [enc_outputs, hidden_state,
                                enc_valid_len]
```

Giờ ta có thể chạy thử mô hình seq2seq áp dụng cơ chế tập trung. Để nhất quán khi so sánh với mô hình không áp dụng cơ chế tập trung trong Section 11.7, những siêu tham số vocab\_size, embed\_size, num\_hiddens, và num\_layers sẽ được giữ nguyên. Kết quả, ta thu được đầu ra của bộ giải mã có cùng kích thước nhưng khác về cấu trúc trạng thái.

```
encoder = d2l.Seq2SeqEncoder(vocab_size=10, embed_size=8,
                               num_hiddens=16, num_layers=2)
encoder.initialize()
decoder = Seq2SeqAttentionDecoder(vocab_size=10, embed_size=8,
                                   num_hiddens=16, num_layers=2)
decoder.initialize()
X = np.zeros((4, 7))
state = decoder.init_state(encoder(X), None)
out, state = decoder(X, state)
out.shape, len(state), state[0].shape, len(state[1]), state[1][0].shape
```

## 12.2.2 Huấn luyện

Chúng ta hãy xây dựng một mô hình đơn giản sử dụng cùng một bộ siêu tham số và hàm mất mát để huấn luyện như Section 11.7.4. Từ kết quả, ta thấy tầng tập trung được thêm vào mô hình không tạo ra cải thiện đáng kể nào do các chuỗi trong tập huấn luyện khá ngắn. Bởi chi phí tính toán tốn thêm từ các tầng tập trung trong bộ mã hóa và bộ giải mã, mô hình này hoạt động chậm hơn nhiều so với mô hình seq2seq không áp dụng tập trung.

```
embed_size, num_hiddens, num_layers, dropout = 32, 32, 2, 0.0
batch_size, num_steps = 64, 10
lr, num_epochs, ctx = 0.005, 200, d2l.try_gpu()

src_vocab, tgt_vocab, train_iter = 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)
model = d2l.EncoderDecoder(encoder, decoder)
d2l.train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)
```

Cuối cùng, ta hãy thử dự đoán một vài mẫu dưới đây.

```
for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + d2l.predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))
```

### 12.2.3 Tóm tắt

- Mô hình seq2seq áp dụng cơ chế tập trung thêm một tầng tập trung vào mô hình seq2seq ban đầu.
- Bộ giải mã của mô hình seq2seq áp dụng cơ chế tập trung được truyền vào ba đầu ra từ bộ mã hóa: đầu ra của bộ mã hóa tại tất cả các bước thời gian, trạng thái ẩn của bộ mã hóa tại bước thời gian cuối cùng, độ dài hợp lệ của bộ mã hóa.

### 12.2.4 Bài tập

1. So sánh Seq2SeqAttentionDecoder và Seq2seqDecoder bằng cách sử dụng cùng bộ tham số và kiểm tra giá trị hàm mất mát.
2. Bạn hãy thử suy nghĩ liệu có trường hợp nào mà Seq2SeqAttentionDecoder vượt trội hơn Seq2seqDecoder?

### 12.2.5 Thảo luận

- Tiếng Anh<sup>220</sup>
- Tiếng Việt<sup>221</sup>

### 12.2.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

## 12.3 Kiến trúc Transformer

Trong các chương trước, ta đã đề cập đến các kiến trúc mạng nơ-ron quan trọng như mạng nơ-ron tích chập (CNN) và mạng nơ-ron hồi tiếp (RNN). Ưu nhược điểm của hai kiến trúc mạng này có thể được tóm tắt như sau:

- Các mạng **CNN** có thể dễ dàng được thực hiện song song ở một tầng nhưng không có khả năng nắm bắt các phụ thuộc chuỗi có độ dài biến thiên.
- Các mạng **RNN** có khả năng nắm bắt các thông tin cách xa nhau trong chuỗi có độ dài biến thiên, nhưng không thể thực hiện song song trong một chuỗi.

<sup>220</sup> <https://discuss.mxnet.io/t/4345>

<sup>221</sup> <https://forum.machinelearningcoban.com/c/d21>

Để kết hợp các ưu điểm của CNN và RNN, (Vaswani et al., 2017) đã thiết kế một kiến trúc mới bằng cách sử dụng cơ chế tập trung. Kiến trúc này gọi là *Transformer*, song song hóa bằng cách học chuỗi hồi tiếp với cơ chế tập trung, đồng thời mã hóa vị trí của từng phần tử trong chuỗi. Kết quả là ta có một mô hình tương thích với thời gian huấn luyện ngắn hơn đáng kể.

Tương tự như mô hình seq2seq trong Section 11.7, Transformer cũng dựa trên kiến trúc mã hóa-giải mã. Tuy nhiên, nó thay thế các tầng hồi tiếp trong seq2seq bằng các tầng *tập trung đa đầu* (*multi-head attention*), kết hợp thông tin vị trí thông qua *biểu diễn vị trí* (*positional encoding*) và áp dụng *chuẩn hóa tầng* (*layer normalization*). Fig. 12.3.1 sẽ so sánh cấu trúc của Transformer và seq2seq.

Nhìn chung, hai mô hình này khá giống nhau: các embedding của chuỗi nguồn được đưa vào  $n$  khối lặp lại. Đầu ra của khối mã hóa cuối cùng sau đó được sử dụng làm bộ nhớ tập trung cho bộ giải mã. Tương tự, các embedding của chuỗi đích được đưa vào  $n$  khối lặp lại trong bộ giải mã. Ta thu được đầu ra cuối cùng bằng cách áp dụng một tầng dày đặc có kích thước bằng kích thước bộ từ vựng lên các đầu ra của khối giải mã cuối cùng.

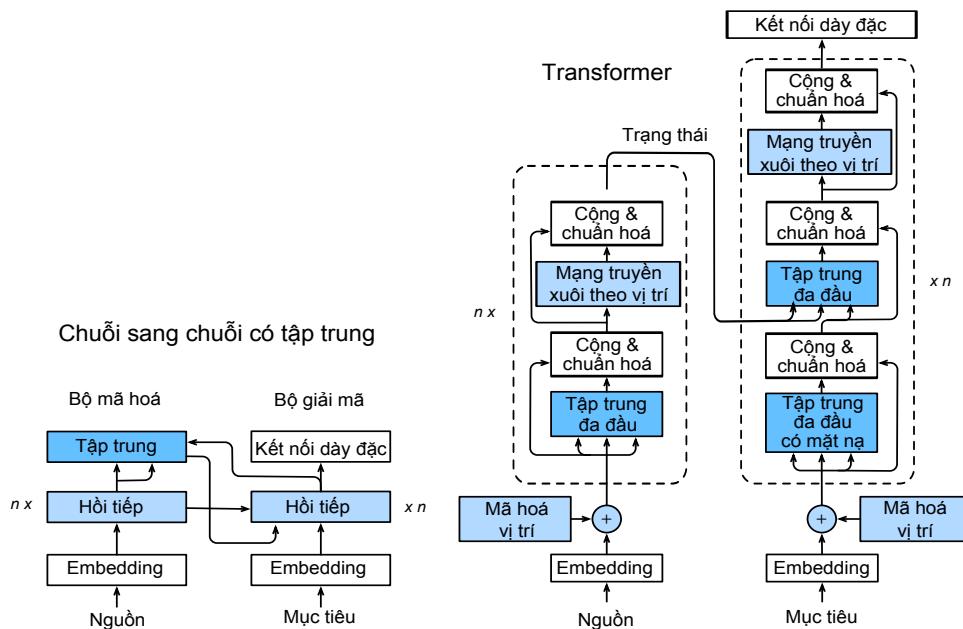


Fig. 12.3.1: Kiến trúc Transformer.

Mặt khác, Transformer khác với mô hình seq2seq sử dụng cơ chế tập trung như sau:

- Khối Transformer:** một tầng hồi tiếp trong seq2seq được thay bằng một *Khối Transformer*. Với bộ mã hóa, khối này chứa một tầng *tập trung đa đầu* và một *mạng truyền xuôi theo vị trí* (*position-wise feed-forward network*) gồm hai tầng dày đặc. Đối với bộ giải mã, khối này có thêm một tầng tập trung đa đầu khác để nhận vào trạng thái bộ mã hóa.
- Cộng và chuẩn hóa:** đầu vào và đầu ra của cả tầng tập trung đa đầu hoặc mạng truyền xuôi theo vị trí được xử lý bởi hai tầng “cộng và chuẩn hóa” bao gồm cấu trúc phân dư và tầng *chuẩn hóa theo tầng* (*layer normalization*).
- Biểu diễn vị trí:** do tầng tự tập trung không phân biệt thứ tự phần tử trong một chuỗi, nên tầng biểu diễn vị trí được sử dụng để thêm thông tin vị trí vào từng phần tử trong chuỗi.

Tiếp theo, chúng ta sẽ tìm hiểu từng thành phần trong Transformer để có thể xây dựng một mô hình dịch máy.

```

from d2l import mxnet as d2l
import math
from mxnet import autograd, np, npx
from mxnet.gluon import nn
npx.set_np()

```

### 12.3.1 Tập trung Đa đầu

Trước khi thảo luận về tầng *tập trung đa đầu*, hãy cùng tìm hiểu qua về kiến trúc *tự tập trung*. Giống như các mô hình tập trung bình thường, mô hình tự tập trung cũng có câu truy vấn, khóa và giá trị nhưng chúng được sao chép từ các phần tử trong chuỗi đầu vào. Như minh họa trong Fig. 12.3.2, tầng tự tập trung trả về một đầu ra tuần tự có cùng độ dài với đầu vào. So với tầng hồi tiếp, các phần tử đầu ra của tầng tự tập trung có thể được tính toán song song, do đó việc xây dựng các đoạn mã tốc độ cao khá dễ dàng.

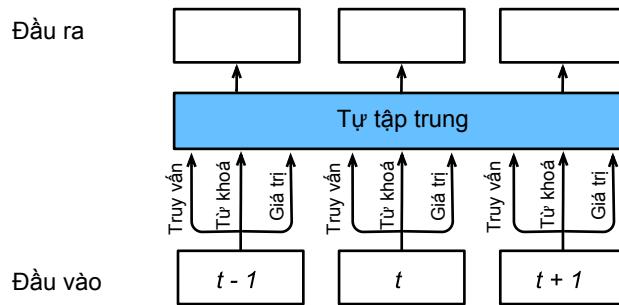


Fig. 12.3.2: Kiến trúc tự tập trung.

Tầng *tập trung đa đầu* bao gồm  $h$  đầu là các tầng tự tập trung song song. Trước khi đưa vào mỗi đầu, ta chiếu các câu truy vấn, khóa và giá trị qua ba tầng dày đặc với kích thước ẩn lần lượt là  $p_q$ ,  $p_k$  và  $p_v$ . Đầu ra của  $h$  đầu này được nối lại và sau đó được xử lý bởi một tầng dày đặc cuối cùng.

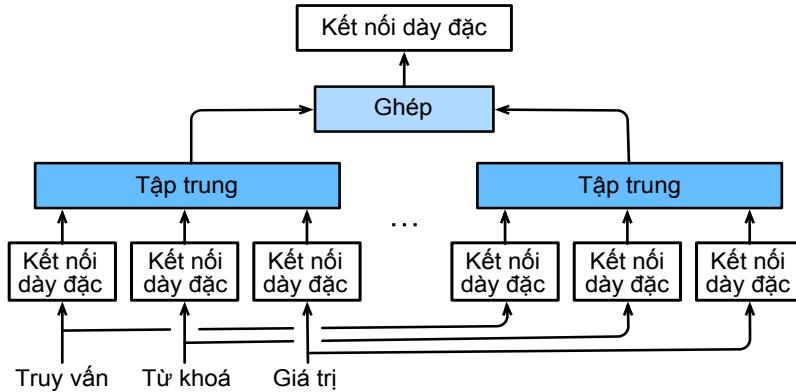


Fig. 12.3.3: Tập trung đa đầu

Giả sử chiều của câu truy vấn, khóa và giá trị lần lượt là  $d_q$ ,  $d_k$  và  $d_v$ . Khi đó, tại mỗi đầu  $i = 1, \dots, h$ , ta có thể học các tham số  $\mathbf{W}_q^{(i)} \in \mathbb{R}^{p_q \times d_q}$ ,  $\mathbf{W}_k^{(i)} \in \mathbb{R}^{p_k \times d_k}$ , và  $\mathbf{W}_v^{(i)} \in \mathbb{R}^{p_v \times d_v}$ . Do đó, đầu ra tại mỗi đầu là

$$\mathbf{o}^{(i)} = \text{attention}(\mathbf{W}_q^{(i)} \mathbf{q}, \mathbf{W}_k^{(i)} \mathbf{k}, \mathbf{W}_v^{(i)} \mathbf{v}), \quad (12.3.1)$$

trong đó attention có thể là bất kỳ tầng tập trung nào, chẳng hạn như DotProductAttention và MLPAttention trong Section 12.1.

Sau đó,  $h$  đầu ra với độ dài  $p_v$  tại mỗi đầu được nối với nhau thành đầu ra có độ dài  $hp_v$ , rồi được đưa vào tầng dày đặc cuối cùng với  $d_o$  nút ẩn. Các trọng số của tầng dày đặc này được ký hiệu là  $\mathbf{W}_o \in \mathbb{R}^{d_o \times hp_v}$ . Do đó, đầu ra cuối cùng của tầng tập trung đa đầu sẽ là

$$\mathbf{o} = \mathbf{W}_o \begin{bmatrix} \mathbf{o}^{(1)} \\ \vdots \\ \mathbf{o}^{(h)} \end{bmatrix}. \quad (12.3.2)$$

Bây giờ chúng ta có thể lập trình tầng tập trung đa đầu. Giả sử tầng tập trung đa đầu có số đầu là  $\text{num\_heads} = h$  và các tầng dày đặc cho câu truy vấn, khóa và giá trị có kích thước ẩn giống nhau  $\text{num\_hiddens} = p_q = p_k = p_v$ . Ngoài ra, do tầng tập trung đa đầu giữ nguyên kích thước chiều đầu vào, kích thước đặc trưng đầu ra cũng là  $d_o = \text{num\_hiddens}$ .

```
# Saved in the d2l package for later use
class MultiHeadAttention(nn.Block):
    def __init__(self, num_hiddens, num_heads, dropout, use_bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_k = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_v = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)
        self.W_o = nn.Dense(num_hiddens, use_bias=use_bias, flatten=False)

    def forward(self, query, key, value, valid_len):
        # For self-attention, query, key, and value shape:
        # (batch_size, seq_len, dim), where seq_len is the length of input
        # sequence. valid_len shape is either (batch_size, ) or
        # (batch_size, seq_len).

        # Project and transpose query, key, and value from
        # (batch_size, seq_len, num_hiddens) to
        # (batch_size * num_heads, seq_len, num_hiddens / num_heads).
        query = transpose_qkv(self.W_q(query), self.num_heads)
        key = transpose_qkv(self.W_k(key), self.num_heads)
        value = transpose_qkv(self.W_v(value), self.num_heads)

        if valid_len is not None:
            # Copy valid_len by num_heads times
            if valid_len.ndim == 1:
                valid_len = np.tile(valid_len, self.num_heads)
            else:
                valid_len = np.tile(valid_len, (self.num_heads, 1))

            # For self-attention, output shape:
            # (batch_size * num_heads, seq_len, num_hiddens / num_heads)
            output = self.attention(query, key, value, valid_len)

            # output_concat shape: (batch_size, seq_len, num_hiddens)
            output_concat = transpose_output(output, self.num_heads)
            return self.W_o(output_concat)
```

Dưới đây là định nghĩa của hai hàm chuyển vị transpose\_qkv và transpose\_output. Hai hàm này là nghịch đảo của nhau.

```
# Saved in the d2l package for later use
def transpose_qkv(X, num_heads):
    # Input X shape: (batch_size, seq_len, num_hiddens).
    # Output X shape:
    # (batch_size, num_heads, num_hiddens / num_heads)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)

    # X shape: (batch_size, num_heads, seq_len, num_hiddens / num_heads)
    X = X.transpose(0, 2, 1, 3)

    # output shape: (batch_size * num_heads, seq_len, num_hiddens / num_heads)
    output = X.reshape(-1, X.shape[2], X.shape[3])
    return output

# Saved in the d2l package for later use
def transpose_output(X, num_heads):
    # A reversed version of transpose_qkv
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.transpose(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

Hãy cùng kiểm tra mô hình MultiHeadAttention qua một ví dụ đơn giản. Tạo một tầng tập trung đa đầu với kích thước ẩn  $d_o = 100$ , đầu ra sẽ có cùng kích thước batch và độ dài chuỗi với đầu vào, nhưng có kích thước chiều cuối cùng bằng num\_hiddens = 100.

```
cell = MultiHeadAttention(90, 9, 0.5)
cell.initialize()
X = np.ones((2, 4, 5))
valid_len = np.array([2, 3])
cell(X, X, X, valid_len).shape
```

### 12.3.2 Mạng truyền Xuôi theo Vị trí

Một thành phần quan trọng khác trong Khối Transformer là *mạng truyền xuôi theo vị trí* (*position-wise feed-forward network*). Nó chấp nhận đầu vào 3 chiều với kích thước là (kích thước batch, độ dài chuỗi, kích thước đặc trưng). Mạng truyền xuôi theo vị trí bao gồm hai tầng dày đặc áp dụng trên chiều cuối cùng của đầu vào. Vì hai tầng dày đặc này cùng được sử dụng cho từng vị trí trong chuỗi, nên ta gọi là mạng truyền xuôi theo vị trí. Cách làm này tương đương với việc áp dụng hai tầng tích chập  $1 \times 1$ .

Lớp PositionWiseFFN dưới đây lập trình mạng truyền xuôi theo vị trí với hai tầng dày đặc có kích thước ẩn lần lượt là ffn\_num\_hiddens và pw\_num\_outputs.

```
# Saved in the d2l package for later use
class PositionWiseFFN(nn.Block):
    def __init__(self, ffn_num_hiddens, pw_num_outputs, **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Dense(ffn_num_hiddens, flatten=False,
                             activation='relu')
```

(continues on next page)

```

    self.dense2 = nn.Dense(pw_num_outputs, flatten=False)

def forward(self, X):
    return self.dense2(self.dense1(X))

```

Tương tự như tầng tập trung đa đầu, mạng truyền xuôi theo vị trí sẽ chỉ thay đổi kích thước chiều cuối cùng của đầu vào — tức kích thước của đặc trưng. Ngoài ra, nếu hai phần tử trong chuỗi đầu vào giống hệt nhau, thì hai đầu ra tương ứng cũng sẽ giống hệt nhau.

```

ffn = PositionWiseFFN(4, 8)
ffn.initialize()
ffn(np.ones((2, 3, 4)))[0]

```

### 12.3.3 Cộng và Chuẩn hóa

Trong kiến trúc Transformer, tầng “cộng và chuẩn hóa” cũng đóng vai trò thiết yếu trong việc kết nối đầu vào và đầu ra của các tầng khác một cách trơn tru. Cụ thể, ta thêm một cấu trúc phần dư và tầng *chuẩn hóa theo tầng* sau tầng tập trung đa đầu và mạng truyền xuôi theo vị trí. *Chuẩn hóa theo tầng* khá giống với chuẩn hóa theo batch trong [Section 9.5](#). Một điểm khác biệt là giá trị trung bình và phương sai của tầng chuẩn hóa này được tính theo chiều cuối cùng, tức  $X.mean(axis=-1)$ , thay vì theo chiều đầu tiên (theo batch)  $X.mean(axis=0)$ . Chuẩn hóa tầng ngăn không cho phạm vi giá trị trong các tầng thay đổi quá nhiều, giúp huấn luyện nhanh hơn và khái quát hóa tốt hơn.

MXNet có cả LayerNorm và BatchNorm được lập trình trong khối nn. Hãy cùng xem sự khác biệt giữa chúng qua ví dụ dưới đây.

```

layer = nn.LayerNorm()
layer.initialize()
batch = nn.BatchNorm()
batch.initialize()
X = np.array([[1, 2], [2, 3]])
# Compute mean and variance from X in the training mode
with autograd.record():
    print('layer norm:', layer(X), '\nbatch norm:', batch(X))

```

Bây giờ hãy cùng lập trình khối AddNorm. AddNorm nhận hai đầu vào  $X$  và  $Y$ . Ta có thể coi  $X$  là đầu vào ban đầu trong mạng phần dư và  $Y$  là đầu ra từ tầng tập trung đa đầu hoặc mạng truyền xuôi theo vị trí. Ngoài ra, ta cũng sẽ áp dụng dropout lên  $Y$  để điều chỉnh.

```

# Saved in the d2l package for later use
class AddNorm(nn.Block):
    def __init__(self, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm()

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)

```

Do có kết nối phần dư,  $X$  và  $Y$  sẽ có kích thước giống nhau.

```

add_norm = AddNorm(0.5)
add_norm.initialize()
add_norm(np.ones((2, 3, 4)), np.ones((2, 3, 4))).shape

```

#### 12.3.4 Biểu diễn Vị trí

Không giống như tầng hồi tiếp, cả tầng tập trung đa đầu và mạng truyền xuôi theo vị trí đều tính toán đầu ra cho từng phần tử trong chuỗi một cách độc lập. Điều này cho phép song song hóa công việc tính toán nhưng lại không mô hình hóa được thông tin tuần tự trong chuỗi đầu vào. Để nắm bắt các thông tin tuần tự một cách hiệu quả, mô hình Transformer sử dụng *biểu diễn vị trí* (*positional encoding*) để duy trì thông tin vị trí của chuỗi đầu vào.

Cụ thể, giả sử  $X \in \mathbb{R}^{l \times d}$  là embedding của mẫu đầu vào, trong đó  $l$  là độ dài chuỗi và  $d$  là kích thước embedding. Tầng biểu diễn vị trí sẽ mã hóa vị trí  $P \in \mathbb{R}^{l \times d}$  của  $X$  và trả về đầu ra  $P + X$ .

Vị trí  $P$  là ma trận 2 chiều, trong đó  $i$  là thứ tự trong câu,  $j$  là vị trí theo chiều embedding. Bằng cách này, mỗi vị trí trong chuỗi ban đầu được biểu diễn bởi hai phương trình dưới đây:

$$P_{i,2j} = \sin(i/10000^{2j/d}), \quad (12.3.3)$$

$$P_{i,2j+1} = \cos(i/10000^{2j/d}), \quad (12.3.4)$$

với  $i = 0, \dots, l - 1$  và  $j = 0, \dots, \lfloor (d - 1)/2 \rfloor$ .

Fig. 12.3.4 minh họa biểu diễn vị trí.

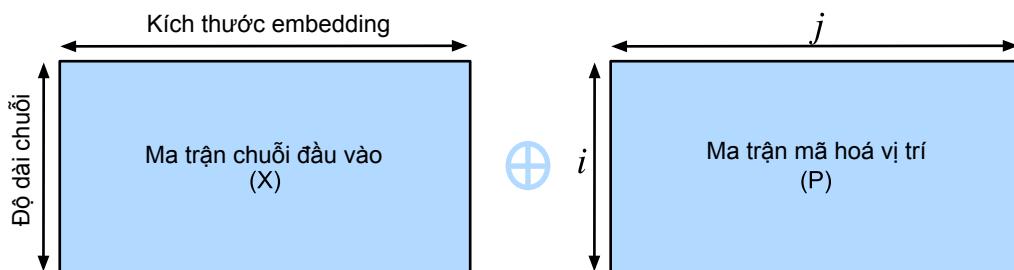


Fig. 12.3.4: Biểu diễn vị trí.

```

# Saved in the d2l package for later use
class PositionalEncoding(nn.Block):
    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 = np.zeros((1, max_len, num_hiddens))
        X = np.arange(0, max_len).reshape(-1, 1) / np.power(
            10000, np.arange(0, num_hiddens, 2) / num_hiddens)
        self.P[:, :, 0::2] = np.sin(X)
        self.P[:, :, 1::2] = np.cos(X)

    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].as_in_ctx(X.ctx)
        return self.dropout(X)

```

Bây giờ chúng ta kiểm tra lớp PositionalEncoding ở trên bằng một mô hình đơn giản cho 4 chiều. Có thể thấy, chiều thứ 4 và chiều thứ 5 có cùng tần số nhưng khác độ dời, còn chiều thứ 6 và 7 có tần số thấp hơn.

```
pe = PositionalEncoding(20, 0)
pe.initialize()
Y = pe(np.zeros((1, 100, 20)))
d2l.plot(np.arange(100), Y[0, :, 4:8].T, figsize=(6, 2.5),
         legend=["dim %d" % p for p in [4, 5, 6, 7]])
```

### 12.3.5 Bộ Mã hóa

Với các thành phần thiết yếu trên, hãy xây dựng bộ mã hóa cho Transformer. Bộ mã hóa này chứa một tầng tập trung đa đầu, một mạng truyền xuôi theo vị trí và hai khối kết nối “cộng và chuẩn hóa”. Trong mã nguồn, có thể thấy cả tầng tập trung và mạng truyền xuôi theo vị trí trong EncoderBlock đều có đầu ra với kích thước là num\_hiddens. Điều này là do kết nối phần dư trong quá trình “cộng và chuẩn hóa”, khi ta cần cộng đầu ra của hai khối này với giá trị đầu vào của chúng.

```
# Saved in the d2l package for later use
class EncoderBlock(nn.Block):
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads, dropout,
                 use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = MultiHeadAttention(num_hiddens, num_heads, dropout,
                                             use_bias)
        self.addnorm1 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(dropout)

    def forward(self, X, valid_len):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_len))
        return self.addnorm2(Y, self.ffn(Y))
```

Nhờ kết nối phần dư, khối mã hóa sẽ không thay đổi kích thước đầu vào. Nói cách khác, giá trị num\_hiddens phải bằng kích thước chiều cuối cùng của đầu vào. Trong ví dụ đơn giản dưới đây, num\_hiddens = 24, ffn\_num\_hiddens = 48, num\_heads = 8 và dropout = 0.5.

```
X = np.ones((2, 100, 24))
encoder_blk = EncoderBlock(24, 48, 8, 0.5)
encoder_blk.initialize()
encoder_blk(X, valid_len).shape
```

Bây giờ hãy lập trình bộ mã hóa của Transformer hoàn chỉnh. Trong bộ mã hóa,  $n$  khối EncoderBlock được xếp chồng lên nhau. Nhờ có kết nối phần dư, tầng embedding và đầu ra khối Transformer đều có kích thước là  $d$ . Cũng lưu ý rằng ta nhân các embedding với  $\sqrt{d}$  để tránh trường hợp giá trị này quá nhỏ.

```
# Saved in the d2l package for later use
class TransformerEncoder(d2l.Encoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
```

(continues on next page)

```

self.num_hiddens = num_hiddens
self.embedding = nn.Embedding(vocab_size, num_hiddens)
self.pos_encoding = PositionalEncoding(num_hiddens, dropout)
self.blks = nn.Sequential()
for _ in range(num_layers):
    self.blks.add(
        EncoderBlock(num_hiddens, ffn_num_hiddens, num_heads, dropout, use_bias))

def forward(self, X, valid_len, *args):
    X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
    for blk in self.blks:
        X = blk(X, valid_len)
    return X

```

Hãy tạo một bộ mã hóa với hai khối mã hóa Transformer với siêu tham số như ví dụ trên. Ngoài ra, ta đặt hai tham số vocab\_size bằng 200 và num\_layers bằng 2.

```

encoder = TransformerEncoder(200, 24, 48, 8, 2, 0.5)
encoder.initialize()
encoder(np.ones((2, 100)), valid_len).shape

```

### 12.3.6 Bộ Giải mã

Khối giải mã của Transformer gần tương tự như khối mã hóa. Tuy nhiên, bên cạnh hai tầng con (tập trung đa đầu và biểu diễn vị trí), khối giải mã còn chứa tầng tập trung đa đầu áp dụng lên đầu ra của bộ mã hóa. Các tầng con này cũng được kết nối bằng các tầng “cộng và chuẩn hóa”, gồm kết nối phần dư và chuẩn hóa theo tầng.

Cụ thể, tại bước thời gian  $t$ , giả sử đầu vào hiện tại là câu truy vấn  $\mathbf{x}_t$ . Như minh họa trong Fig. 12.3.5, các khóa và giá trị của tầng tập trung gồm có câu truy vấn ở bước thời gian hiện tại và tất cả các câu truy vấn ở các bước thời gian trước  $\mathbf{x}_1, \dots, \mathbf{x}_{t-1}$ .

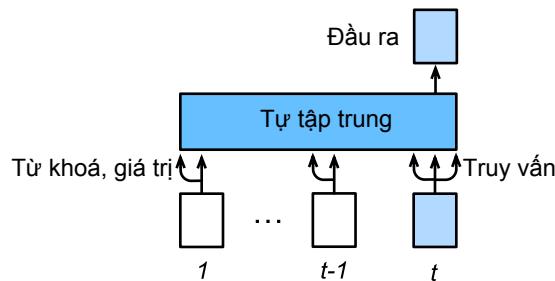


Fig. 12.3.5: Dự đoán ở bước thời gian  $t$  của một tầng tự tập trung.

Trong quá trình huấn luyện, đầu ra của câu truy vấn  $t$  có thể quan sát được tất cả các cặp khóa - giá trị trước đó. Điều này dẫn đến hai cách hoạt động khác nhau của mạng khi huấn luyện và dự đoán. Vì thế, trong lúc dự đoán chúng ta có thể loại bỏ những thông tin không cần thiết bằng cách chỉ định độ dài hợp lệ là  $t$  cho câu truy vấn thứ  $t$ .

```

class DecoderBlock(nn.Block):
    # i means it is the i-th block in the decoder
    def __init__(self, num_hiddens, ffn_num_hiddens, num_heads,
                 dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = MultiHeadAttention(num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(dropout)
        self.attention2 = MultiHeadAttention(num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(dropout)
        self.ffn = PositionWiseFFN(ffn_num_hiddens, num_hiddens)
        self.addnorm3 = AddNorm(dropout)

    def forward(self, X, state):
        enc_outputs, enc_valid_len = state[0], state[1]
        # state[2][i] contains the past queries for this block
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = np.concatenate((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values
        if autograd.is_training():
            batch_size, seq_len, _ = X.shape
            # Shape: (batch_size, seq_len), the values in the j-th column
            # are j+1
            valid_len = np.tile(np.arange(1, seq_len+1, ctx=X.ctx),
                               (batch_size, 1))
        else:
            valid_len = None
        X2 = self.attention1(X, key_values, key_values, valid_len)
        Y = self.addnorm1(X, X2)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_len)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state

```

Tương tự như khối mã hóa của Transformer, num\_hiddens của khối giải mã phải bằng với kích thước chiều cuối cùng của  $X$ .

```

decoder_blk = DecoderBlock(24, 48, 8, 0.5, 0)
decoder_blk.initialize()
X = np.ones((2, 100, 24))
state = [encoder_blk(X, valid_len), valid_len, [None]]
decoder_blk(X, state)[0].shape

```

Bộ giải mã hoàn chỉnh của Transformer được lập trình tương tự như bộ mã hóa, ngoại trừ chi tiết tầng kết nối đầy đủ được thêm vào để tính điểm tin cậy của đầu ra.

Hãy lập trình bộ giải mã đầy đủ TransformerDecoder. Ngoài các siêu tham số thường gấp như vocab\_size và num\_hiddens, bộ giải mã cần thêm kích thước đầu ra của bộ mã hóa enc\_outputs và độ dài hợp lệ env\_valid\_len.

```

class TransformerDecoder(d2l.Decoder):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens,
                 num_heads, num_layers, dropout, **kwargs):

```

(continues on next page)

```

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 = PositionalEncoding(num_hiddens, dropout)
self.blks = nn.Sequential()
for i in range(num_layers):
    self.blks.add(
        DecoderBlock(num_hiddens, ffn_num_hiddens, num_heads,
                    dropout, i))
self.dense = nn.Dense(vocab_size, flatten=False)

def init_state(self, enc_outputs, env_valid_len, *args):
    return [enc_outputs, env_valid_len, [None]*self.num_layers]

def forward(self, X, state):
    X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
    for blk in self.blks:
        X, state = blk(X, state)
    return self.dense(X), state

```

### 12.3.7 Huấn luyện

Cuối cùng, chúng ta có thể xây dựng một mô hình mã hóa - giải mã với kiến trúc Transformer. Tương tự như mô hình seq2seq áp dụng cơ chế tập trung trong Section 12.2, chúng ta sử dụng các siêu tham số sau: hai khối Transformer có kích thước embedding và kích thước đầu ra đều là 32. Bên cạnh đó, chúng ta sử dụng 4 đầu trong tầng tập trung và đặt kích thước ẩn bằng hai lần kích thước đầu ra.

```

num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.0, 64, 10
lr, num_epochs, ctx = 0.005, 100, d2l.try_gpu()
ffn_num_hiddens, num_heads = 64, 4

src_vocab, tgt_vocab, train_iter = d2l.load_data_nmt(batch_size, num_steps)

encoder = TransformerEncoder(
    len(src_vocab), num_hiddens, ffn_num_hiddens, num_heads, num_layers,
    dropout)
decoder = TransformerDecoder(
    len(src_vocab), num_hiddens, ffn_num_hiddens, num_heads, num_layers,
    dropout)
model = d2l.EncoderDecoder(encoder, decoder)
d2l.train_s2s_ch9(model, train_iter, lr, num_epochs, ctx)

```

Dựa trên thời gian huấn luyện và độ chính xác, ta thấy Transformer chạy nhanh hơn trên mỗi epoch và hội tụ nhanh hơn ở giai đoạn đầu so với seq2seq áp dụng cơ chế tập trung.

Chúng ta có thể sử dụng Transformer đã được huấn luyện để dịch một số câu đơn giản dưới đây.

```

for sentence in ['Go .', 'Wow !', "I'm OK .", 'I won !']:
    print(sentence + ' => ' + d2l.predict_s2s_ch9(
        model, sentence, src_vocab, tgt_vocab, num_steps, ctx))

```

### 12.3.8 Tóm tắt

- Mô hình Transformer dựa trên kiến trúc mã hóa - giải mã.
- Tầng tập trung đa đầu gồm có  $h$  tầng tập trung song song.
- Mạng truyền xuôi theo vị trí gồm hai tầng kết nối đầy đủ được áp dụng trên chiều cuối cùng.
- Chuẩn hóa theo tầng được áp dụng trên chiều cuối cùng (chiều đặc trưng), thay vì chiều đầu tiên (kích thước batch) như ở chuẩn hóa theo batch.
- Biểu diễn vị trí là nơi duy nhất đưa thông tin vị trí trong chuỗi vào mô hình Transformer.

### 12.3.9 Bài tập

1. Hãy thử huấn luyện với nhiều epoch hơn và so sánh mất mát giữa seq2seq và Transformer.
2. Biểu diễn vị trí còn có lợi ích gì khác không?
3. So sánh chuẩn hóa theo tầng với chuẩn hóa theo batch. Khi nào nên sử dụng một trong hai tầng chuẩn hóa thay vì tầng còn lại?

### 12.3.10 Thảo luận

- Tiếng Anh<sup>222</sup>
- Tiếng Việt<sup>223</sup>

### 12.3.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Duy Du
- Lê Khắc Hồng Phúc
- Trần Yến Thy
- Phạm Minh Đức
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Nguyễn Cảnh Thưởng

<sup>222</sup> <https://discuss.mxnet.io/t/4344>

<sup>223</sup> <https://forum.machinelearningcoban.com/c/d21>

## 12.4 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

# 13 | Thuật toán Tối ưu

Chúng tôi tin rằng khi đã theo dõi đến chương này của cuốn sách, hẳn là bạn đã kinh qua nhiều dạng thuật toán tối ưu tiên tiến để huấn luyện các mô hình học sâu. Chúng là công cụ cho phép ta liên tục cập nhật các tham số của mô hình và cực tiểu hóa giá trị hàm mất mát khi đánh giá trên tập huấn luyện. Sự thật là có nhiều người hài lòng với việc xem những thuật toán tối ưu như một hộp đen ma thuật (với các câu thần chú như “Adam”, “NAG”, hoặc “SGD”) có tác dụng cực tiểu hóa hàm mục tiêu.

Tuy nhiên, để làm tốt thì ta cần những kiến thức chuyên sâu hơn. Những giải thuật tối ưu đóng vai trò quan trọng trong học sâu. Một mặt, việc huấn luyện một mô hình học sâu phức tạp có thể mất hàng giờ, hàng ngày, thậm chí là hàng tuần. Chất lượng của thuật toán tối ưu ảnh hưởng trực tiếp đến độ hiệu quả của quá trình huấn luyện của mô hình. Mặt khác, việc hiểu rõ nguyên lý của các thuật toán tối ưu khác nhau cùng vai trò của các tham số đi kèm sẽ giúp ta điều chỉnh các siêu tham số một cách có chủ đích nhằm cải thiện hiệu suất của các mô hình học sâu.

Trong chương này, chúng tôi sẽ mô tả sâu hơn các thuật toán tối ưu thông dụng trong học sâu. Hầu hết tất cả các bài toán tối ưu xuất hiện trong học sâu đều là *không lồi* (*nonconvex*). Tuy nhiên, kiến thức từ việc thiết kế và phân tích các thuật toán giải quyết bài toán tối ưu lồi vẫn rất hữu ích. Do vậy, phần này sẽ tập trung vào giới thiệu tối ưu lồi và chứng minh một thuật toán hạ gradient ngẫu nhiên (*stochastic gradient descent*) đơn giản áp dụng cho hàm mục tiêu lồi.

## 13.1 Tối ưu và Học sâu

Trong phần này, ta sẽ thảo luận mối quan hệ giữa tối ưu và học sâu, cũng như những thách thức khi áp dụng các thuật toán tối ưu trong học sâu. Đối với một bài toán học sâu, đầu tiên chúng ta thường định nghĩa hàm mất mát, sau đó sử dụng một thuật toán tối ưu nhằm cực tiểu hóa hàm mất mát đó. Hàm mất mát trong học sâu thường được xem là hàm mục tiêu của bài toán tối ưu. Thông thường, đa số các thuật toán tối ưu thường giải quyết bài toán *cực tiểu hóa*. Tuy nhiên, nếu ta cần cực đại hóa, có một cách khá đơn giản là đổi dấu hàm mục tiêu.

### 13.1.1 Tối ưu và Ước lượng

Mặc dù các phương pháp tối ưu thường được sử dụng để cực tiểu hóa hàm mất mát trong học sâu, nhưng mục đích của tối ưu và học sâu về bản chất là khác nhau. Mối quan tâm của tối ưu chủ yếu là cực tiểu hóa một mục tiêu nào đó, trong khi đối với học sâu là tìm kiếm một mô hình phù hợp với một lượng dữ liệu hữu hạn. Trong Section 6.4, ta đã thảo luận chi tiết về sự khác biệt giữa các mục đích trên. Chẳng hạn như là sự khác biệt giữa lỗi huấn luyện và lỗi khái quát. Do hàm mục tiêu của thuật toán tối ưu thường là hàm mất mát trên tập huấn luyện nên mục đích của tối ưu là giảm thiểu lỗi huấn luyện. Tuy nhiên, mục đích của suy luận thống kê (*statistical inference*) và học

sâu nói riêng là giảm thiểu lỗi khái quát. Để thực hiện điều này, bên cạnh việc giảm thiểu lỗi huấn luyện, ta cần chú ý đến hiện tượng quá khớp. Hãy bắt đầu bằng việc nhập một số thư viện sử dụng trong chương này.

```
%matplotlib inline
from d2l import mxnet as d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

```
<!--
```

```
-->
```

Tiếp theo, ta định nghĩa hai hàm: hàm kỳ vọng  $f$  và hàm thực nghiệm  $g$  để minh họa vấn đề này. Ở đây,  $g$  kém mượt hơn  $f$  vì ta chỉ có một lượng dữ liệu hữu hạn.

```
def f(x): return x * np.cos(np.pi * x)
def g(x): return f(x) + 0.2 * np.cos(5 * np.pi * x)
```

Đồ thị phía dưới mô tả chi tiết hơn về vấn đề trên. Do ta chỉ có một lượng dữ liệu hữu hạn, cực tiểu của lỗi huấn luyện có thể khác so với cực tiểu kỳ vọng của lỗi (lỗi trên tập kiểm tra).

```
def annotate(text, xy, xytext): #@save
    d2l.plt.gca().annotate(text, xy=xy, xytext=xytext,
                           arrowprops=dict(arrowstyle='->'))

x = np.arange(0.5, 1.5, 0.01)
d2l.set_figsize((4.5, 2.5))
d2l.plot(x, [f(x), g(x)], 'x', 'risk')
annotate('empirical risk', (1.0, -1.2), (0.5, -1.1))
annotate('expected risk', (1.1, -1.05), (0.95, -0.5))
```

## 13.2 Các Thách thức của Tối ưu trong Học sâu

Ở chương này, ta sẽ chỉ tập trung vào chất lượng của thuật toán tối ưu trong việc cực tiểu hóa hàm mục tiêu, thay vì lỗi khái quát của mô hình. Trong [Section 5.1](#), ta đã phân biệt giữa nghiệm theo công thức và nghiệm xấp xỉ trong các bài toán tối ưu. Trong học sâu, đa số các hàm mục tiêu khá phức tạp và không tính được nghiệm theo công thức. Thay vào đó, ta phải dùng các thuật toán tối ưu xấp xỉ. Các thuật toán tối ưu dưới đây được liệt vào loại này.

Có rất nhiều thách thức về tối ưu trong học sâu. Các điểm cực tiểu, điểm yên ngựa, tiêu biến gradient là một số vấn đề gây đau đầu hơn cả. Hãy cùng tìm hiểu về các vấn đề này.

### 13.3 Các vùng Cực tiểu

Cho hàm mục tiêu  $f(x)$ , nếu giá trị của  $f(x)$  tại  $x$  nhỏ hơn các giá trị khác của  $f(x)$  tại lân cận của  $x$  thì  $f(x)$  có thể là một *cực tiểu* (*local minimum*). Nếu giá trị của hàm mục tiêu  $f(x)$  tại  $x$  là nhỏ nhất trên toàn tập xác định thì  $f(x)$  là *giá trị nhỏ nhất* (*global minimum*).

Ví dụ, cho hàm

$$f(x) = x \cdot \cos(\pi x) \text{ for } -1.0 \leq x \leq 2.0, \quad (13.3.1)$$

ta có thể tính xấp xỉ cực tiểu và giá trị nhỏ nhất của hàm này.

```
x = np.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))
```

Hàm mục tiêu trong các mô hình học sâu thường có nhiều vùng cực trị. Khi nghiệm xấp xỉ của một bài toán tối ưu đang ở gần giá trị cực tiểu, gradient của hàm mục tiêu tại nghiệm này gần hoặc bằng 0, tuy nhiên nghiệm này có thể chỉ đang cực tiểu hóa hàm mục tiêu một cách cục bộ chứ không phải toàn cục. Chỉ với một mức độ nhiễu nhất định thì mới có thể đẩy tham số ra khỏi vùng cực tiểu. Trên thực tế, nhiễu là một trong những tính chất có lợi của hạ gradient ngẫu nhiên khi sự biến động của gradient qua từng minibatch có thể đẩy các tham số ra khỏi các vùng cực tiểu.

### 13.4 Các điểm Yên ngựa

Ngoài các vùng cực tiểu, các điểm yên ngựa cũng có gradient bằng 0. Một *điểm yên ngựa*<sup>224</sup> là bất cứ điểm nào mà tất cả gradient của một hàm bằng 0, nhưng đó không phải là một điểm cực tiểu hay giá trị nhỏ nhất. Xét hàm  $f(x) = x^3$ , đạo hàm bậc một và bậc hai của hàm này bằng 0 tại điểm  $x = 0$ . Việc tối ưu có thể bị ngưng trệ tại điểm này, cho dù đó không phải là điểm cực tiểu.

```
x = np.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 ngựa* trong không gian nhiều chiều còn khó chịu hơn nhiều, như ví dụ dưới đây. Xét hàm  $f(x, y) = x^2 - y^2$ . Hàm này tồn tại một điểm yên ngựa tại  $(0, 0)$ . Đây là một điểm cực đại theo  $y$  và là điểm cực tiểu theo  $x$ . Tên gọi của tính chất toán học này bắt nguồn từ chính việc đồ thị tại đó có hình dạng giống một cái yên ngựa.

```
x, y = np.meshgrid(np.linspace(-1.0, 1.0, 101), np.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)
```

(continues on next page)

<sup>224</sup> [https://en.wikipedia.org/wiki/Saddle\\_point](https://en.wikipedia.org/wiki/Saddle_point)

```
d2l=plt.yticks(ticks)
ax.set_zticks(ticks)
d2l.plt.xlabel('x')
d2l.plt.ylabel('y');
```

Giả sử đầu vào của một hàm là một vector  $k$  chiều và đầu ra là một số vô hướng; do đó ma trận Hessian của nó có  $k$  trị riêng (xem thêm tại [Section 20.1](#)). Nghiệm của hàm này có thể là một cực tiểu, cực đại, hoặc một điểm yên ngựa tại vị trí mà gradient của hàm bằng 0.

- Điểm cực tiểu là vị trí mà ở đó gradient bằng 0 và các trị riêng của ma trận Hessian đều dương.
- Điểm cực đại là vị trí mà ở đó gradient bằng 0 và các trị riêng của ma trận Hessian đều âm.
- Điểm yên ngựa là vị trí mà ở đó gradient bằng 0 và các trị riêng của ma trận Hessian mang cả giá trị âm lẫn dương.

Đối với bài toán trong không gian nhiều chiều, khả năng có một vài trị riêng âm là khá cao. Do đó các điểm yên ngựa có khả năng xuất hiện cao hơn các cực tiểu. Ta sẽ thảo luận một số ngoại lệ của vấn đề này ở phần tới khi giới thiệu đến tính lồi. Nói ngắn gọn, các hàm lồi là hàm có các trị riêng của ma trận Hessian không bao giờ âm. Nhưng không may, đa số bài toán học sâu đều không thuộc loại này. Dù sao thì tính lồi vẫn là một công cụ tốt để học về các thuật toán tối ưu.

## 13.5 Tiêu biến Gradient

Có lẽ vấn đề khó chịu nhất mà ta phải đối mặt là tiêu biến gradient. Ví dụ, giả sử ta muốn cực tiểu hóa hàm  $f(x) = \tanh(x)$  và ta bắt đầu tại  $x = 4$ . Như ta có thể thấy, gradient của  $f$  gần như là bằng 0. Cụ thể,  $f'(x) = 1 - \tanh^2(x)$  và do đó  $f'(4) = 0.0013$ . Hậu quả là quá trình tối ưu sẽ bị trì trệ khá lâu trước khi có tiến triển. Đây hóa ra lại là lý do tại sao việc huấn luyện các mô hình học sâu khá khó khăn trước khi hàm kích hoạt ReLU xuất hiện.

```
x = np.arange(-2.0, 5.0, 0.01)
d2l.plot(x, [np.tanh(x)], 'x', 'f(x)')
annotate('vanishing gradient', (4, 1), (2, 0.0))
```

Tối ưu trong học sâu mang đầy thử thách. May mắn thay, có khá nhiều thuật toán hoạt động tốt và dễ sử dụng ngay cả đối với người mới bắt đầu. Hơn nữa, việc tìm kiếm giải pháp tốt nhất là không thật cần thiết. Các cực tiểu và ngay cả nghiệm xấp xỉ cũng đã rất hữu dụng rồi.

### 13.5.1 Tóm tắt

- Cực tiểu hóa lồi huấn luyện *không* đảm bảo việc ta sẽ tìm ra tập tham số tốt nhất để cực tiểu hóa lồi khái quát.
- Các bài toán tối ưu thường có nhiều vùng cực tiểu.
- Và do các bài toán thường không có tính lồi, số lượng điểm yên ngựa thậm chí có thể nhiều hơn.
- Tiêu biến gradient có thể khiến cho quá trình tối ưu bị đình trệ. Thường thì việc tái tham số hóa bài toán (*reparameterization*) và khởi tạo tham số cẩn thận cũng sẽ giúp ích.

### 13.5.2 Bài tập

1. Xét một mạng perceptron đa tầng đơn giản với một tầng ẩn  $d$  chiều và một đầu ra duy nhất. Chỉ ra rằng bất kỳ cực tiểu nào cũng có ít nhất  $d!$  nghiệm tương đương khiến mạng vận hành giống nhau.
2. Giả sử ta có một ma trận đối xứng  $\mathbf{M}$  ngẫu nhiên, trong đó mỗi phần tử  $M_{ij} = M_{ji}$  tuân theo phân phối xác suất  $p_{ij}$ . Ngoài ra, giả sử  $p_{ij}(x) = p_{ij}(-x)$ , tức phân phối là đối xứng (xem (Wigner, 1958) để biết thêm chi tiết).
  - Chứng minh rằng phân phối của các trị riêng cũng là đối xứng, tức với mọi vector riêng  $\mathbf{v}$ , trị riêng  $\lambda$  tương ứng thoả mãn  $P(\lambda > 0) = P(\lambda < 0)$ .
  - Tại sao điều trên không có nghĩa là  $P(\lambda > 0) = 0.5$ ?
3. Liệu còn thử thách tối ưu nào trong học sâu không?
4. Giả sử bạn muốn cân bằng một quả bóng (thật) trên một chiếc yên ngựa (thật).
  - Tại sao điều này lại khó khăn?
  - Hãy vận dụng kết quả trên vào các thuật toán tối ưu.

### 13.5.3 Thảo luận

- Tiếng Anh - MXNet<sup>225</sup>
- Tiếng Anh - Pytorch<sup>226</sup>
- Tiếng Anh - Tensorflow<sup>227</sup>
- Tiếng Việt<sup>228</sup>

### 13.5.4 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Nguyễn Văn Quang
- Phạm Minh Đức
- Nguyễn Văn Cường
- Phạm Hồng Vinh

<sup>225</sup> <https://discuss.d2l.ai/t/349>

<sup>226</sup> <https://discuss.d2l.ai/t/487>

<sup>227</sup> <https://discuss.d2l.ai/t/489>

<sup>228</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 13.6 Tính lồi

Tính lồi đóng vai trò then chốt trong việc thiết kế các thuật toán tối ưu. Điều này phần lớn là do tính lồi giúp việc phân tích và kiểm tra thuật toán trở nên dễ dàng hơn. Nói cách khác, nếu thuật toán hoạt động kém ngay cả khi có tính lồi thì ta không nên kì vọng rằng sẽ thu được kết quả tốt trong trường hợp khác. Hơn nữa, mặc dù các bài toán tối ưu hóa trong học sâu đa phần là không lồi, chúng lại thường thể hiện một số tính chất lồi gần các cực tiểu. Điều này dẫn đến các biến thể tối ưu hóa thú vị mới như (Izmailov et al., 2018).

### 13.6.1 Kiến thức Cơ bản

Chúng ta hãy bắt đầu với các kiến thức cơ bản trước.

#### Tập hợp

Tập hợp là nền tảng của tính lồi. Nói một cách đơn giản, một tập hợp  $X$  trong không gian vector là lồi nếu với bất kỳ  $a, b \in X$ , đoạn thẳng nối  $a$  và  $b$  cũng thuộc  $X$ . Theo thuật ngữ toán học, điều này có nghĩa là với mọi  $\lambda \in [0, 1]$ , ta có

$$\lambda \cdot a + (1 - \lambda) \cdot b \in X \text{ với mọi } a, b \in X. \quad (13.6.1)$$

Điều này nghe có vẻ hơi trừu tượng. Hãy xem qua bức ảnh Fig. 13.6.1. Tập hợp đầu tiên là không lồi do tồn tại các đoạn thẳng không nằm trong tập hợp. Hai tập hợp còn lại thì không gặp vấn đề như vậy.

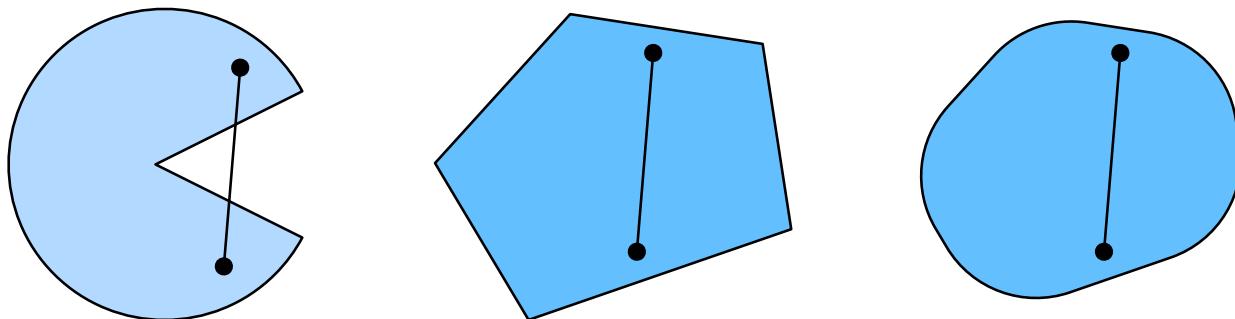


Fig. 13.6.1: Ba hình dạng, hình bên trái là không lồi, hai hình còn lại là lồi

Chỉ một mình định nghĩa thôi thì sẽ không có tác dụng gì trừ khi bạn có thể làm gì đó với chúng. Trong trường hợp này, ta có thể nhìn vào phép hợp và phép giao trong Fig. 13.6.2. Giả sử  $X$  và  $Y$  là các tập hợp lồi, khi đó  $X \cap Y$  cũng sẽ lồi. Để thấy được điều này, hãy xét bất kỳ  $a, b \in X \cap Y$ . Vì  $X$  và  $Y$  lồi, khi đó đoạn thẳng nối  $a$  và  $b$  sẽ nằm trong cả  $X$  và  $Y$ . Do đó, chúng cũng cần phải thuộc  $X \cap Y$ , từ đó chứng minh được định lý đầu tiên của chúng ta.

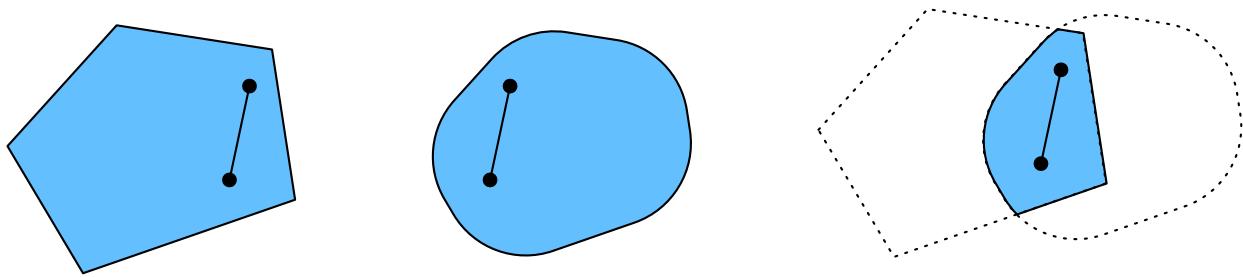


Fig. 13.6.2: Giao của hai tập lồi là một tập lồi

Ta sẽ củng cố kết quả này thêm chút với mệnh đề: giao của các tập lồi  $X_i$  là một tập lồi  $\cap_i X_i$ . Để thấy rằng điều ngược lại là không đúng, hãy xem xét hai tập hợp không giao nhau  $X \cap Y = \emptyset$ . Giờ ta chọn ra  $a \in X$  và  $b \in Y$ . Đoạn thẳng nối  $a$  và  $b$  trong Fig. 13.6.3 chứa một vài phần không thuộc cả  $X$  và  $Y$ , vì chúng ta đã giả định rằng  $X \cap Y = \emptyset$ . Do đó đoạn thẳng này cũng không nằm trong  $X \cup Y$ , từ đó chứng minh rằng hợp của các tập lồi nói chung không nhất thiết phải là tập lồi.

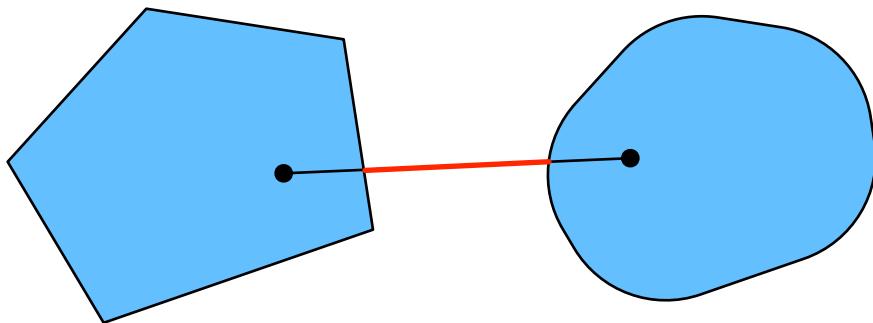


Fig. 13.6.3: Hợp của hai tập lồi không nhất thiết phải là tập lồi

Thông thường, các bài toán trong học sâu đều được định nghĩa trong các miền lồi. Ví dụ  $\mathbb{R}^d$  là tập lồi (xét cho cùng, đoạn thẳng nối hai điểm bất kỳ thuộc  $\mathbb{R}^d$  vẫn thuộc  $\mathbb{R}^d$ ). Trong một vài trường hợp, chúng ta sẽ làm việc với các biến có biên, ví dụ như khối cầu có bán kính  $r$  được định nghĩa bằng  $\{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ và } \|\mathbf{x}\|_2 \leq r\}$ .

## Hàm số

Giờ ta đã biết về tập hợp lồi, ta sẽ làm việc tiếp với các hàm số lồi  $f$ . Cho một tập hợp lồi  $X$ , một hàm số được định nghĩa trên tập đó  $f : X \rightarrow \mathbb{R}$  là hàm lồi nếu với mọi  $x, x' \in X$  và mọi  $\lambda \in [0, 1]$ , ta có

$$\lambda f(x) + (1 - \lambda)f(x') \geq f(\lambda x + (1 - \lambda)x'). \quad (13.6.2)$$

Để minh họa cho điều này, chúng ta sẽ vẽ đồ thị của một vài hàm số và kiểm tra xem hàm số nào thỏa mãn điều kiện trên. Ta sẽ cần phải nhập một vài gói thư viện.

```
%matplotlib inline
from d2l import mxnet as d2l
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()
```

<!--

-->

Hãy định nghĩa một vài hàm số, cả lồi lẫn không lồi.

```
f = lambda x: 0.5 * x**2 # Convex
g = lambda x: np.cos(np.pi * x) # Nonconvex
h = lambda x: np.exp(0.5 * x) # Convex

x, segment = np.arange(-2, 2, 0.01), np.array([-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)
```

Như dự đoán, hàm cô-sin là hàm không lồi, trong khi hàm parabol và hàm số mũ là hàm lồi. Lưu ý rằng để điều kiện trên có ý nghĩa thì  $X$  cần phải là tập hợp lồi. Nếu không, kết quả của  $f(\lambda x + (1 - \lambda)x')$  sẽ không được định nghĩa rõ. Các hàm lồi có một số tính chất mong muốn sau.

### Bất đẳng thức Jensen

Một trong những công cụ hữu dụng nhất là bất đẳng thức Jensen. Nó là sự tổng quát hóa của định nghĩa về tính lồi:

$$\sum_i \alpha_i f(x_i) \geq f\left(\sum_i \alpha_i x_i\right) \text{ và } E_x[f(x)] \geq f(E_x[x]), \quad (13.6.3)$$

với  $\alpha_i$  là các số thực không âm sao cho  $\sum_i \alpha_i = 1$ . Nói cách khác, kỳ vọng của hàm lồi lớn hơn hàm lồi của kỳ vọng. Để chứng minh bất đẳng thức đầu tiên này, chúng ta áp dụng định nghĩa của tính lồi cho từng số hạng của tổng. Kỳ vọng có thể được chứng minh bằng cách tính giới hạn trên các đoạn hữu hạn.

Một trong các ứng dụng phổ biến của bất đẳng thức Jensen liên quan đến log hợp lý của các biến ngẫu nhiên quan sát được một phần. Ta có

$$E_{y \sim P(y)}[-\log P(x | y)] \geq -\log P(x). \quad (13.6.4)$$

Điều này xảy ra vì  $\int P(y)P(x | y)dy = P(x)$ . Nó được sử dụng trong những phương pháp biến phân.  $y$  ở đây thường là một biến ngẫu nhiên không quan sát được,  $P(y)$  là dự đoán tốt nhất về phân phối của nó và  $P(x)$  là phân phối đã được lấy tích phân theo  $y$ . Ví dụ như trong bài toán phân cụm,  $y$  có thể là nhãn cụm và  $P(x | y)$  là mô hình sinh khi áp dụng các nhãn cụm.

#### 13.6.2 Tính chất

Các hàm lồi có một vài tính chất hữu ích dưới đây.

## Không có Cực tiểu Cực bộ

Cụ thể, các hàm lồi không có cực tiểu cục bộ. Hãy giả định điều ngược lại là đúng và chứng minh nó sai. Nếu  $x \in X$  là cực tiểu cục bộ thì sẽ tồn tại một vùng lân cận nào đó của  $x$  mà  $f(x)$  là giá trị nhỏ nhất. Vì  $x$  chỉ là cực tiểu cục bộ nên phải tồn tại một  $x' \in X$  nào khác mà  $f(x') < f(x)$ . Tuy nhiên, theo tính lồi, các giá trị hàm số trên toàn bộ *đường thẳng*  $\lambda x + (1 - \lambda)x'$  phải nhỏ hơn  $f(x')$  với  $\lambda \in [0, 1]$

$$f(x) > \lambda f(x) + (1 - \lambda)f(x') \geq f(\lambda x + (1 - \lambda)x'). \quad (13.6.5)$$

Điều này mâu thuẫn với giả định rằng  $f(x)$  là cực tiểu cục bộ. Ví dụ, hàm  $f(x) = (x + 1)(x - 1)^2$  có cực tiểu cục bộ tại  $x = 1$ . Tuy nhiên nó lại không phải là cực tiểu toàn cục.

```
f = lambda x: (x-1)**2 * (x+1)
d2l.set_figsize()
d2l.plot([x, segment], [f(x), f(segment)], 'x', 'f(x)')
```

Tính chất “các hàm lồi không có cực tiểu cục bộ” rất tiện lợi. Điều này có nghĩa là ta sẽ không bao giờ “mắc kẹt” khi cực tiểu hóa các hàm số. Dù vậy, hãy lưu ý rằng điều này không có nghĩa là hàm số không thể có nhiều hơn một cực tiểu toàn cục, hoặc liệu hàm số có tồn tại cực tiểu toàn cục hay không. Ví dụ, hàm  $f(x) = \max(|x| - 1, 0)$  đạt giá trị nhỏ nhất trên khoảng  $[-1, 1]$ . Ngược lại, hàm  $f(x) = \exp(x)$  không có giá trị nhỏ nhất trên  $\mathbb{R}$ . Với  $x \rightarrow -\infty$  nó sẽ tiệm cận tới 0, tuy nhiên không tồn tại giá trị  $x$  mà tại đó  $f(x) = 0$ .

## Hàm số và Tập hợp Lồi

Các hàm số lồi định nghĩa các tập hợp lồi là các *tập-dưới* (*below-sets*) như sau:

$$S_b := \{x | x \in X \text{ and } f(x) \leq b\}. \quad (13.6.6)$$

Ta hãy chứng minh nó một cách vắn tắt. Hãy nhớ rằng với mọi  $x, x' \in S_b$ , ta cần chứng minh  $\lambda x + (1 - \lambda)x' \in S_b$  với mọi  $\lambda \in [0, 1]$ . Nhưng điều này lại trực tiếp tuân theo định nghĩa về tính lồi vì  $f(\lambda x + (1 - \lambda)x') \leq \lambda f(x) + (1 - \lambda)f(x') \leq b$ .

Hãy nhìn vào đồ thị hàm  $f(x, y) = 0.5x^2 + \cos(2\pi y)$  bên dưới. Nó rõ ràng là không lồi. Các tập mức tương ứng cũng không lồi. Thực tế, chúng thường được cấu thành từ các tập hợp rời rạc.

```
x, y = np.meshgrid(np.linspace(-1, 1, 101), np.linspace(-1, 1, 101))
z = x**2 + 0.5 * np.cos(2 * np.pi * y)

# Plot the 3D surface
d2l.set_figsize((6, 4))
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.contour(x, y, z, offset=-1)
ax.set_zlim(-1, 1.5)

# Adjust labels
for func in [d2l.plt.xticks, d2l.plt.yticks, ax.set_zticks]:
    func([-1, 0, 1])
```

## Đạo hàm và tính lồi

Bất cứ khi nào đạo hàm bậc hai của một hàm số tồn tại, việc kiểm tra tính lồi của hàm số là rất đơn giản. Tất cả những gì cần làm là kiểm tra liệu  $\partial_x^2 f(x) \succeq 0$ , tức là liệu toàn bộ trị riêng của nó đều không âm hay không. Chẳng hạn, hàm  $f(\mathbf{x}) = \frac{1}{2} \|\mathbf{x}\|_2^2$  là lồi vì  $\partial_{\mathbf{x}}^2 f = \mathbf{I}$ , tức là đạo hàm của nó là ma trận đơn vị.

Có thể nhận ra rằng chúng ta chỉ cần chứng minh tính chất này cho các hàm số một chiều. Xét cho cùng, ta luôn có thể định nghĩa một hàm số  $g(z) = f(\mathbf{x} + z \cdot \mathbf{v})$ . Hàm số này có đạo hàm bậc một và bậc hai lần lượt là  $g' = (\partial_{\mathbf{x}} f)^{\top} \mathbf{v}$  và  $g'' = \mathbf{v}^{\top} (\partial_{\mathbf{x}}^2 f) \mathbf{v}$ . Cụ thể,  $g'' \geq 0$  với mọi  $\mathbf{v}$  mỗi khi ma trận Hessian của  $f$  là nửa xác định dương, tức là tất cả các trị riêng của ma trận đều lớn hơn hoặc bằng không. Do đó quay về lại trường hợp vô hướng.

Để thấy tại sao  $f''(x) \geq 0$  đối với các hàm lồi, ta dùng lập luận

$$\frac{1}{2}f(x+\epsilon) + \frac{1}{2}f(x-\epsilon) \geq f\left(\frac{x+\epsilon}{2} + \frac{x-\epsilon}{2}\right) = f(x). \quad (13.6.7)$$

Vì đạo hàm bậc hai được đưa ra bởi giới hạn trên sai phân hữu hạn, nó dẫn tới

$$f''(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x+\epsilon) + f(x-\epsilon) - 2f(x)}{\epsilon^2} \geq 0. \quad (13.6.8)$$

Để chứng minh điều ngược lại, ta dùng lập luận rằng  $f'' \geq 0$  ngụ ý rằng  $f'$  là một hàm tăng đơn điệu. Cho  $a < x < b$  là ba điểm thuộc  $\mathbb{R}$ . Chúng ta sử dụng định lý giá trị trung bình để biểu diễn

$$\begin{aligned} f(x) - f(a) &= (x-a)f'(\alpha) \text{ với } \alpha \in [a, x] \text{ và} \\ f(b) - f(x) &= (b-x)f'(\beta) \text{ với } \beta \in [x, b]. \end{aligned} \quad (13.6.9)$$

Từ tính chất đơn điệu  $f'(\beta) \geq f'(\alpha)$ , ta có

$$\begin{aligned} f(b) - f(a) &= f(b) - f(x) + f(x) - f(a) \\ &= (b-x)f'(\beta) + (x-a)f'(\alpha) \\ &\geq (b-a)f'(\alpha). \end{aligned} \quad (13.6.10)$$

Theo hình học, nó dẫn đến  $f(x)$  nằm dưới đường thẳng nối  $f(a)$  và  $f(b)$ , do đó chứng minh được tính lồi. Ta sẽ bỏ qua việc chứng minh một cách chính quy và thay bằng đồ thị bên dưới.

```
f = lambda x: 0.5 * x**2
x = np.arange(-2, 2, 0.01)
axb, ab = np.array([-1.5, -0.5, 1]), np.array([-1.5, 1])
d2l.set_figsize()
d2l.plot([x, axb, ab], [f(x) for x in [x, axb, ab]], 'x', 'f(x)')
d2l.annotate('a', (-1.5, f(-1.5)), (-1.5, 1.5))
d2l.annotate('b', (1, f(1)), (1, 1.5))
d2l.annotate('x', (-0.5, f(-0.5)), (-1.5, f(-0.5)))
```

### 13.6.3 Ràng buộc

Một trong những tính chất hữu ích của tối ưu hóa lồi là nó cho phép chúng ta xử lý các ràng buộc một cách hiệu quả. Nó cho phép ta giải quyết các bài toán dưới dạng:

$$\begin{aligned} & \underset{\mathbf{x}}{\text{cực tiểu hóa } f(\mathbf{x})} \\ & \text{theo } c_i(\mathbf{x}) \leq 0 \text{ với mọi } i \in \{1, \dots, N\}. \end{aligned} \tag{13.6.11}$$

$f$  ở đây là mục tiêu và các hàm  $c_i$  là các hàm số ràng buộc. Hãy xem nó xử lý thế nào trong trường hợp  $c_1(\mathbf{x}) = \|\mathbf{x}\|_2 - 1$ . Ở trường hợp này, các tham số  $\mathbf{x}$  bị ràng buộc vào khối cầu đơn vị. Nếu ràng buộc thứ hai là  $c_2(\mathbf{x}) = \mathbf{v}^\top \mathbf{x} + b$  thì điều này ứng với mọi  $\mathbf{x}$  nằm trên nửa khoảng. Đáp ứng đồng thời hai ràng buộc này nghĩa là chọn ra một lát cắt của khối cầu làm tập hợp ràng buộc.

### Hàm số Lagrange

Nhìn chung, giải quyết một bài toán tối ưu hóa bị ràng buộc là tương đối khó khăn. Có một cách giải quyết bắt nguồn từ vật lý dựa trên một trực giác khá đơn giản. Hãy tưởng tượng có một quả banh bên trong một chiếc hộp. Quả banh sẽ lăn đến nơi thấp nhất và trọng lực sẽ cân bằng với lực nâng của các cạnh hộp tác động lên quả banh. Tóm lại, gradient của hàm mục tiêu (ở đây là trọng lực) sẽ được bù lại bởi gradient của hàm ràng buộc (cần phải nằm trong chiếc hộp, bị các bức tường “đẩy lại”). Lưu ý rằng bất kỳ ràng buộc nào không kích hoạt (quả banh không đụng đến bức tường) thì sẽ không có bất kỳ lực tác động nào lên quả banh.

Ta hãy bỏ qua phần diễn giải chứng minh của hàm số Lagrange  $L$  (Xem sách của Boyd và Vandenberghe về vấn đề này ([Boyd & Vandenberghe, 2004](#))). Lý luận bên trên có thể được mô tả thông qua bài toán tối ưu hóa điểm yên ngựa:

$$L(\mathbf{x}, \alpha) = f(\mathbf{x}) + \sum_i \alpha_i c_i(\mathbf{x}) \text{ với } \alpha_i \geq 0. \tag{13.6.12}$$

Các biến  $\alpha_i$  ở đây được gọi là *nhân tử Lagrange* (*Lagrange Multipliers*), chúng đảm bảo rằng các ràng buộc sẽ được tuân thủ đúng hoang. Chúng được chọn vừa đủ lớn để đảm bảo rằng  $c_i(\mathbf{x}) \leq 0$  với mọi  $i$ . Ví dụ, với mọi  $\mathbf{x}$  mà  $c_i(\mathbf{x}) < 0$  một cách tự nhiên, chúng ta rốt cuộc sẽ chọn  $\alpha_i = 0$ . Hơn nữa, đây là bài toán tối ưu hóa *điểm yên ngựa*, nơi ta muốn *cực đại hóa*  $L$  theo  $\alpha$  và đồng thời *cực tiểu hóa* nó theo  $\mathbf{x}$ . Có rất nhiều tài liệu giải thích về cách đưa đến hàm  $L(\mathbf{x}, \alpha)$ . Đối với mục đích của chúng ta, sẽ là đủ khi biết rằng điểm yên ngựa của  $L$  là nơi bài toán tối ưu hóa bị ràng buộc ban đầu được giải quyết một cách tối ưu.

### Lượng phạt

Có một cách để thỏa mãn, ít nhất là theo xấp xỉ, các bài toán tối ưu hóa bị ràng buộc là phỏng theo hàm Lagrange  $L$ . Thay vì thỏa mãn  $c_i(\mathbf{x}) \leq 0$ , chúng ta chỉ cần thêm  $\alpha_i c_i(\mathbf{x})$  vào hàm mục tiêu  $f(x)$ . Điều này sẽ đảm bảo rằng các ràng buộc không bị vi phạm quá mức.

Thực tế, chúng ta đã dùng thủ thuật này khá thường xuyên. Hãy xét đến suy giảm trọng số trong [Section 6.5](#). Ở đó chúng ta thêm  $\frac{\lambda}{2} \|\mathbf{w}\|^2$  vào hàm mục tiêu để đảm bảo giá trị  $\mathbf{w}$  không trở nên quá lớn. Dưới góc nhìn tối ưu hóa có ràng buộc, ta có thể thấy nó sẽ đảm bảo  $\|\mathbf{w}\|^2 - r^2 \leq 0$  với giá trị bán kính  $r$  nào đó. Điều chỉnh giá trị của  $\lambda$  cho phép chúng ta thay đổi độ lớn của  $\mathbf{w}$ .

Nhìn chung, thêm các lượng phạt là một cách tốt để đảm bảo việc thỏa mãn ràng buộc xấp xỉ. Trong thực tế, hóa ra phương pháp này ổn định hơn rất nhiều so với trường hợp thỏa mãn chuẩn

xác. Hơn nữa, với các bài toán không lồi, những tính chất khiến phương án tiếp cận chuẩn xác trở nên rất thu hút trong trường hợp lồi (ví dụ như tính tối ưu) không còn đảm bảo nữa.

### Các phép chiếu

Một chiến lược khác để thỏa mãn các ràng buộc là các phép chiếu. Chúng ta cũng đã gặp chúng trước đây, ví dụ như khi bàn về phương pháp gọt gradient ở Section 10.5. Ở phần đó chúng ta đã đảm bảo rằng gradient có độ dài ràng buộc bởi  $c$  thông qua

$$\mathbf{g} \leftarrow \mathbf{g} \cdot \min(1, c/\|\mathbf{g}\|). \quad (13.6.13)$$

Hóa ra đây là một *phép chiếu* của  $g$  lên khối cầu có bán kính  $c$ . Tổng quát hơn, một phép chiếu lên một tập (lồi)  $X$  được định nghĩa là

$$\text{Proj}_X(\mathbf{x}) = \underset{\mathbf{x}' \in X}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{x}'\|_2. \quad (13.6.14)$$

Do đó đây là điểm gần nhất trong  $X$  tới  $\mathbf{x}$ . Điều này nghe có vẻ hơi trừu tượng. Fig. 13.6.4 sẽ giải thích nó một cách rõ ràng hơn. Ở đó ta có hai tập lồi, một hình tròn và một hình thoi. Các điểm nằm bên trong tập (màu vàng) giữ nguyên không đổi. Các điểm nằm bên ngoài tập (màu đen) được ánh xạ tới điểm gần nhất bên trong tập (màu đỏ). Trong khi với các khối cầu  $\ell_2$  hướng của phép chiếu được giữ nguyên không đổi, điều này có thể không đúng trong trường hợp tổng quát, như có thể thấy trong trường hợp của hình thoi.

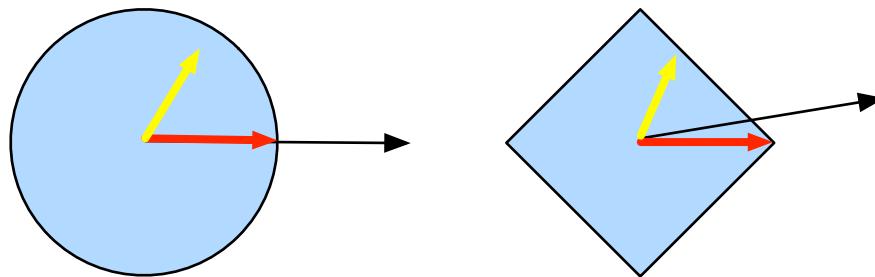


Fig. 13.6.4: Các phép chiếu lồi

Một trong những ứng dụng của các phép chiếu lồi là để tính toán các vector trọng số thưa. Trong trường hợp này chúng ta chiếu  $\mathbf{w}$  lên khối cầu  $\ell_1$  (phiên bản tổng quát của hình thoi ở hình minh họa phía trên).

#### 13.6.4 Tóm tắt

Trong bối cảnh học sâu, mục đích chính của các hàm lồi là để thúc đẩy sự phát triển các thuật toán tối ưu hóa và giúp ta hiểu chúng một cách chi tiết. Phần tiếp theo chúng ta sẽ thấy cách mà hạ gradient và hạ gradient ngẫu nhiên có thể được suy ra từ đó.

- Giao của các tập lồi là tập lồi. Hợp của các tập lồi không bắt buộc phải là tập lồi.
- Kỳ vọng của hàm lồi lớn hơn hàm lồi của kỳ vọng (Bất đẳng thức Jensen).
- Hàm khả vi hai lần là hàm lồi khi và chỉ khi đạo hàm bậc hai của nó chỉ có các trị riêng không âm ở mọi nơi.

- Các ràng buộc lồi có thể được thêm vào hàm Lagrange. Trong thực tế, ta chỉ việc thêm chúng cùng với một mức phạt vào hàm mục tiêu.
- Các phép chiếu ánh xạ đến các điểm trong tập (lồi) nằm gần nhất với điểm gốc.

### 13.6.5 Bài tập

- Giả sử chúng ta muốn xác minh tính lồi của tập hợp bằng cách vẽ mọi đoạn thẳng giữa các điểm bên trong tập hợp và kiểm tra liệu các đoạn thẳng có nằm trong tập hợp đó hay không.
  - Hãy chứng minh rằng ta chỉ cần kiểm tra các điểm ở biên là đủ.
  - Hãy chứng minh rằng ta chỉ cần kiểm tra các đỉnh của tập hợp là đủ.
- Ký hiệu khối cầu có bán kính  $r$  sử dụng chuẩn  $p$  là  $B_p[r] := \{\mathbf{x} | \mathbf{x} \in \mathbb{R}^d \text{ và } \|\mathbf{x}\|_p \leq r\}$ . Hãy chứng minh rằng  $B_p[r]$  là lồi với mọi  $p \geq 1$ .
- Cho các hàm lồi  $f$  và  $g$  sao cho  $\max(f, g)$  cũng là hàm lồi. Hãy chứng minh rằng  $\min(f, g)$  không lồi.
- Hãy chứng minh rằng hàm softmax được chuẩn hóa là hàm lồi. Cụ thể hơn, chứng minh tính lồi của  $f(x) = \log \sum_i \exp(x_i)$ .
- Hãy chứng minh rằng các không gian con tuyến tính là các tập lồi. Ví dụ,  $X = \{\mathbf{x} | \mathbf{Wx} = \mathbf{b}\}$ .
- Hãy chứng minh rằng trong trường hợp của các không gian con tuyến tính với  $\mathbf{b} = 0$ , phép chiếu  $\text{Proj}_X$  có thể được viết dưới dạng  $\mathbf{Mx}$  với một ma trận  $\mathbf{M}$  nào đó.
- Hãy chỉ ra rằng với các hàm số khả vi hai lần  $f$ , ta có thể viết  $f(x + \epsilon) = f(x) + \epsilon f'(x) + \frac{1}{2}\epsilon^2 f''(x + \xi)$  với một giá trị  $\xi \in [0, \epsilon]$  nào đó.
- Cho vector  $\mathbf{w} \in \mathbb{R}^d$  với  $\|\mathbf{w}\|_1 > 1$ , hãy tính phép chiếu lên khối cầu đơn vị  $\ell_1$ .
  - Như một bước trung gian, hãy viết ra mục tiêu có lượng phạt  $\|\mathbf{w} - \mathbf{w}'\|_2^2 + \lambda \|\mathbf{w}'\|_1$  và tính ra đáp án với  $\lambda > 0$ .
  - Bạn có thể tìm ra giá trị ‘chính xác’ của  $\lambda$  mà không phải đoán mò quá nhiều lần không?
- Cho tập lồi  $X$  và hai vector  $\mathbf{x}, \mathbf{y}$ , hãy chứng minh rằng các phép chiếu không bao giờ làm tăng khoảng cách, ví dụ,  $\|\mathbf{x} - \mathbf{y}\| \geq \|\text{Proj}_X(\mathbf{x}) - \text{Proj}_X(\mathbf{y})\|$ .

### 13.6.6 Thảo luận

- Tiếng Anh - MXNet<sup>229</sup>
- Tiếng Anh - Pytorch<sup>230</sup>
- Tiếng Việt<sup>231</sup>

<sup>229</sup> <https://discuss.d2l.ai/t/350>

<sup>230</sup> <https://discuss.d2l.ai/t/488>

<sup>231</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 13.6.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức
- Võ Tấn Phát

## 13.7 Hạ Gradient

Trong phần này chúng tôi sẽ giới thiệu các khái niệm cơ bản trong thuật toán hạ gradient. Nội dung cần thiết sẽ được trình bày ngắn gọn. Đặc giả có thể tham khảo ([Boyd & Vandenberghe, 2004](#)) để có góc nhìn sâu về bài toán tối ưu lồi. Mặc dù tối ưu lồi hiếm khi được áp dụng trực tiếp trong học sâu, kiến thức về thuật toán hạ gradient là chìa khóa để hiểu rõ hơn về thuật toán hạ gradient ngẫu nhiên. Ví dụ, bài toán tối ưu có thể phân kỳ do tốc độ học quá lớn. Hiện tượng này có thể quan sát được trong thuật toán hạ gradient. Tương tự, tiền điều kiện (*preconditioning*) là một kỹ thuật phổ biến trong thuật toán hạ gradient và nó cũng được áp dụng trong các thuật toán tân tiến hơn. Hãy bắt đầu với một trường hợp đặc biệt và đơn giản.

### 13.7.1 Hạ Gradient trong Một Chiều

Hạ gradient trong một chiều là ví dụ tuyệt vời để giải thích tại sao thuật toán hạ gradient có thể giảm giá trị hàm mục tiêu. Hãy xem xét một hàm số thực khả vi liên tục  $f : \mathbb{R} \rightarrow \mathbb{R}$ . Áp dụng khai triển Taylor ([Section 20.3](#)), ta có

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2). \quad (13.7.1)$$

Trong đó xấp xỉ bậc nhất  $f(x + \epsilon)$  được tính bằng giá trị hàm  $f(x)$  và đạo hàm bậc nhất  $f'(x)$  tại  $x$ . Có lý khi giả sử rằng di chuyển theo hướng ngược chiều gradient với  $\epsilon$  nhỏ sẽ làm suy giảm giá trị  $f$ . Để đơn giản hóa vấn đề, ta cố định sai bước cập nhật (tốc độ học)  $\eta > 0$  và chọn  $\epsilon = -\eta f'(x)$ . Thay biểu thức này vào khai triển Taylor ở trên, ta thu được

$$f(x - \eta f'(x)) = f(x) - \eta f'^2(x) + \mathcal{O}(\eta^2 f'^2(x)). \quad (13.7.2)$$

Nếu đạo hàm  $f'(x) \neq 0$  không tiêu biến, quá trình tối ưu sẽ có tiến triển do  $\eta f'^2(x) > 0$ .

Hơn nữa, chúng ta luôn có thể chọn  $\eta$  đủ nhỏ để loại bỏ các hạng tử bậc cao hơn trong phép cập nhật. Do đó, ta có

$$f(x - \eta f'(x)) \lesssim f(x). \quad (13.7.3)$$

Điều này có nghĩa là, nếu chúng ta áp dụng

$$x \leftarrow x - \eta f'(x) \quad (13.7.4)$$

để cập nhật  $x$ , giá trị của hàm  $f(x)$  có thể giảm. Do đó, trong thuật toán hạ gradient, đầu tiên chúng ta chọn giá trị khởi tạo cho  $x$  và hằng số  $\eta > 0$ , từ đó cập nhật giá trị  $x$  liên tục cho tới khi thỏa mãn điều kiện dừng, ví dụ như khi độ lớn của gradient  $|f'(x)|$  đủ nhỏ hoặc số lần cập nhật đạt một ngưỡng nhất định.

Để đơn giản hóa vấn đề, chúng ta chọn hàm mục tiêu  $f(x) = x^2$  để minh họa cách lập trình thuật toán hạ gradient. Ta sử dụng ví dụ đơn giản này để quan sát cách mà  $x$  thay đổi, dù đã biết rằng  $x = 0$  là nghiệm để cực tiểu hóa  $f(x)$ . Như mọi khi, chúng ta bắt đầu bằng cách nhập tất cả các mô-đun cần thiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

<!--

-->

f = lambda x: x**2 # Objective function
gradf = lambda x: 2 * x # Its derivative
```

Tiếp theo, chúng ta sử dụng  $x = 10$  làm giá trị khởi tạo và chọn  $\eta = 0.2$ . Áp dụng thuật toán hạ gradient để cập nhật  $x$  trong 10 vòng lặp, chúng ta có thể thấy cuối cùng giá trị của  $x$  cũng tiệm cận nghiệm tối ưu.

```
def gd(eta):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * gradf(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results
res = gd(0.2)
```

Đồ thị quá trình tối ưu hóa theo  $x$  được vẽ như sau.

```
def show_trace(res):
    n = max(abs(min(res)), abs(max(res)))
    f_line = np.arange(-n, n, 0.01)
    d2l.set figsize()
    d2l.plot([f_line, res], [[f(x) for x in f_line], [f(x) for x in res]],
             'x', 'f(x)', fmts=['-', '-o'])

show_trace(res)
```

## Tốc độ học

Tốc độ học  $\eta$  có thể được thiết lập khi thiết kế thuật toán. Nếu ta sử dụng tốc độ học quá nhỏ thì  $x$  sẽ được cập nhật rất chậm, đòi hỏi số bước cập nhật nhiều hơn để thu được nghiệm tốt hơn. Để minh họa, hãy xem xét quá trình học trong cùng bài toán tối ưu ở phía trên với  $\eta = 0.05$ . Như chúng ta có thể thấy, ngay cả sau 10 bước cập nhật, chúng ta vẫn còn ở rất xa nghiệm tối ưu.

```
show_trace(gd(0.05))
```

Ngược lại, nếu ta sử dụng tốc độ học quá cao, giá trị  $|\eta f'(x)|$  có thể rất lớn trong khai triển Taylor bậc nhất. Cụ thể, hạng tử  $\mathcal{O}(\eta^2 f'^2(x))$  trong :eqref: gd-taylor sẽ có thể có giá trị lớn. Trong trường hợp này, ta không thể đảm bảo rằng việc cập nhật  $x$  sẽ có thể làm suy giảm giá trị của  $f(x)$ . Ví dụ, khi chúng ta thiết lập tốc độ học  $\eta = 1.1$ ,  $x$  sẽ lệch rất xa so với nghiệm tối ưu  $x = 0$  và dần dần phân kỳ.

```
show_trace(gd(1.1))
```

## Cực Tiểu

Để minh họa quá trình học các hàm không lồi, ta xem xét trường hợp  $f(x) = x \cdot \cos cx$ . Hàm này có vô số cực tiểu. Tùy thuộc vào tốc độ học được chọn và điều kiện của bài toán, chúng ta có thể thu được một trong số rất nhiều nghiệm. Ví dụ dưới đây minh họa việc thiết lập tốc độ học quá cao (không thực tế) sẽ dẫn đến điểm cực tiểu không tốt.

```
c = np.array(0.15 * np.pi)
f = lambda x: x * np.cos(c * x)
gradf = lambda x: np.cos(c * x) - c * x * np.sin(c * x)
show_trace(gd(2))
```

### 13.7.2 Hạ Gradient Đa biến

Bây giờ chúng ta đã có trực quan tốt hơn về trường hợp đơn biến, ta hãy xem xét trường hợp trong đó  $\mathbf{x} \in \mathbb{R}^d$ . Cụ thể, hàm mục tiêu  $f : \mathbb{R}^d \rightarrow \mathbb{R}$  ánh xạ các vector tới các giá trị vô hướng. Gradient tương ứng cũng là đa biến, là một vector gồm  $d$  đạo hàm riêng:

$$\nabla f(\mathbf{x}) = \left[ \frac{\partial f(\mathbf{x})}{\partial x_1}, \frac{\partial f(\mathbf{x})}{\partial x_2}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right]^\top. \quad (13.7.5)$$

Mỗi đạo hàm riêng  $\partial f(\mathbf{x})/\partial x_i$  trong gradient biểu diễn tốc độ thay đổi theo  $x_i$  của  $f$  tại  $\mathbf{x}$ . Như trong trường hợp đơn biến giới thiệu ở phần trước, ta sử dụng khai triển Taylor tương ứng cho các hàm đa biến. Cụ thể, ta có

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \mathcal{O}(\|\epsilon\|^2). \quad (13.7.6)$$

Nói cách khác, chiều giảm mạnh nhất được cho bởi gradient âm  $-\nabla f(\mathbf{x})$ , các hạng tử từ bậc hai trở lên trong  $\epsilon$  có thể bỏ qua. Chọn một tốc độ học phù hợp  $\eta > 0$ , ta được thuật toán hạ gradient nguyên bản dưới đây:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \quad (13.7.7)$$

Để xem thuật toán hoạt động như thế nào trong thực tế, ta hãy xây dựng một hàm mục tiêu  $f(\mathbf{x}) = x_1^2 + 2x_2^2$  với đầu vào là vector hai chiều  $\mathbf{x} = [x_1, x_2]^\top$  và đầu ra là một số vô hướng. Gradient được cho bởi  $\nabla f(\mathbf{x}) = [2x_1, 4x_2]^\top$ . Ta sẽ quan sát đường đi của  $\mathbf{x}$  được sinh bởi thuật toán hạ gradient bắt đầu từ vị trí  $[-5, -2]$ . Chúng ta cần thêm hai hàm hỗ trợ. Hàm đầu tiên là hàm cập nhật và được sử dụng 20 lần cho giá trị khởi tạo ban đầu. Hàm thứ hai là hàm vẽ biểu đồ đường đi của  $\mathbf{x}$ .

```
def train_2d(trainer, steps=20):    #@save
    """Optimize a 2-dim objective function with a customized trainer."""
    # s1 and s2 are internal state variables and will
    # be used later in the chapter
    x1, x2, s1, s2 = -5, -2, 0, 0
    results = [(x1, x2)]
    for i in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
        results.append((x1, x2))
    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 = np.meshgrid(np.arange(-5.5, 1.0, 0.1),
                         np.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 sẽ quan sát quỹ đạo của biến tối ưu hóa  $\mathbf{x}$  với tốc độ học  $\eta = 0.1$ . Chúng ta có thể thấy rằng sau 20 bước, giá trị  $\mathbf{x}$  đã đạt cực tiểu tại  $[0, 0]$ . Quá trình khá tốt mặc dù hơi chậm.

```
f = lambda x1, x2: x1 ** 2 + 2 * x2 ** 2    # Objective
gradf = lambda x1, x2: (2 * x1, 4 * x2)    # Gradient

def gd(x1, x2, s1, s2):
    (g1, g2) = gradf(x1, x2)    # Compute gradient
    return (x1 - eta * g1, x2 - eta * g2, 0, 0)    # Update variables

eta = 0.1
show_trace_2d(f, train_2d(gd))
```

### 13.7.3 Những Phương pháp Thích nghi

Như chúng ta có thể thấy ở Section 13.7.1, chọn tốc độ học  $\eta$  “vừa đủ” rất khó. Nếu chọn giá trị quá nhỏ, ta sẽ không có tiến triển. Nếu chọn giá trị quá lớn, nghiệm sẽ dao động và trong trường hợp tệ nhất, thậm chí sẽ phân kỳ. Sẽ ra sao nếu chúng ta có thể chọn  $\eta$  một cách tự động, hoặc giả như loại bỏ được việc chọn kích thước bước? Các phương pháp bậc hai không chỉ dựa vào giá trị và gradient của hàm mục tiêu mà còn dựa vào “độ cong” của hàm, từ đó có thể điều chỉnh tốc độ học. Dù những phương pháp này không thể áp dụng vào học sâu một cách trực tiếp do chi phí tính toán lớn, chúng đem đến những gợi ý hữu ích để thiết kế các thuật toán tối ưu cao cấp hơn, mang nhiều tính chất mong muốn dựa trên các thuật toán dưới đây.

## Phương pháp Newton

Trong khai triển Taylor của  $f$ , ta không cần phải dừng ngay sau số hạng đầu tiên. Trên thực tế, ta có thể viết lại như sau

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2} \epsilon^\top \nabla \nabla^\top f(\mathbf{x}) \epsilon + \mathcal{O}(\|\epsilon\|^3). \quad (13.7.8)$$

Để tránh việc kí hiệu quá nhiều, ta định nghĩa  $H_f := \nabla \nabla^\top f(\mathbf{x})$  là *ma trận Hessian* của  $f$ . Đây là ma trận kích thước  $d \times d$ . Với  $d$  nhỏ và trong các bài toán đơn giản, ta sẽ dễ tính được  $H_f$ . Nhưng với các mạng sâu, kích thước của  $H_f$  có thể cực lớn, do chi phí lưu trữ bậc hai  $\mathcal{O}(d^2)$ . Hơn nữa việc tính toán lan truyền ngược có thể đòi hỏi rất nhiều chi phí tính toán. Tạm thời hãy bỏ qua những lưu ý đó và nhìn vào thuật toán mà ta có được.

Suy cho cùng, cực tiểu của  $f$  sẽ thỏa  $\nabla f(\mathbf{x}) = 0$ . Lấy các đạo hàm của (13.7.8) theo  $\epsilon$  và bỏ qua các số hạng bậc cao ta thu được

$$\nabla f(\mathbf{x}) + H_f \epsilon = 0 \text{ và do đó } \epsilon = -H_f^{-1} \nabla f(\mathbf{x}). \quad (13.7.9)$$

Nghĩa là, ta cần phải nghịch đảo ma trận Hessian  $H_f$  như một phần của bài toán tối ưu hóa.

Với  $f(x) = \frac{1}{2}x^2$  ta có  $\nabla f(x) = x$  và  $H_f = 1$ . Do đó với  $x$  bất kỳ, ta đều thu được  $\epsilon = -x$ . Nói cách khác, một bước đơn lẻ là đã đủ để hội tụ một cách hoàn hảo mà không cần bất kỳ tinh chỉnh nào! Chúng ta khá may mắn ở đây vì khai triển Taylor không cần xấp xỉ. Hãy xem thử điều gì sẽ xảy ra với các bài toán khác.

```
c = np.array(0.5)
f = lambda x: np.cosh(c * x) # Objective
gradf = lambda x: c * np.sinh(c * x) # Derivative
hessf = lambda x: c**2 * np.cosh(c * x) # Hessian

def newton(eta=1):
    x = 10.0
    results = [x]
    for i in range(10):
        x -= eta * gradf(x) / hessf(x)
        results.append(float(x))
    print('epoch 10, x:', x)
    return results
show_trace(newton())
```

Giờ hãy xem điều gì xảy ra với một hàm *không lồi*, ví dụ như  $f(x) = x \cos(cx)$ . Sau tất cả, hãy lưu ý rằng trong phương pháp Newton, chúng ta cuối cùng sẽ phải chia cho ma trận Hessian. Điều này nghĩa là nếu đạo hàm bậc hai là *âm* thì chúng ta phải đi theo hướng *tăng f*. Đó là khiếm khuyết chết người của thuật toán này. Hãy xem điều gì sẽ xảy ra trong thực tế.

```
c = np.array(0.15 * np.pi)
f = lambda x: x * np.cos(c * x)
gradf = lambda x: np.cos(c * x) - c * x * np.sin(c * x)
hessf = lambda x: -2 * c * np.sin(c * x) - x * c**2 * np.cos(c * x)

show_trace(newton())
```

Kết quả trả về là cực kỳ sai. Có một cách khắc phục là “sửa” ma trận Hessian bằng cách lấy giá trị tuyệt đối của nó. Một chiến lược khác là đưa tốc độ học trở lại. Điều này có vẻ sẽ phá hỏng mục

tiêu ban đầu nhưng không hẳn. Có được thông tin bậc hai sẽ cho phép chúng ta thận trọng bất cứ khi nào độ cong trở nên lớn và cho phép thực hiện các bước dài hơn mỗi khi hàm mục tiêu phẳng. Hãy xem nó hoạt động như thế nào với một tốc độ học khá nhỏ,  $\eta = 0.5$  chẳng hạn. Như ta có thể thấy, chúng ta có một thuật toán khá hiệu quả.

```
show_trace(newton(0.5))
```

## Phân tích Hội tụ

Chúng ta sẽ chỉ phân tích tốc độ hội tụ đối với hàm  $f$  lồi và khả vi ba lần, đây là hàm số có đạo hàm bậc hai tại cực tiểu  $x^*$  khác không ( $f''(x^*) > 0$ ).

Đặt  $x_k$  là giá trị của  $x$  tại vòng lặp thứ  $k$  và  $e_k := x_k - x^*$  là khoảng cách đến điểm tối ưu. Theo khai triển Taylor, điều kiện  $f'(x^*) = 0$  được viết lại thành

$$0 = f'(x_k - e_k) = f'(x_k) - e_k f''(x_k) + \frac{1}{2} e_k^2 f'''(\xi_k). \quad (13.7.10)$$

Điều này đúng với một vài  $\xi_k \in [x_k - e_k, x_k]$ . Hãy nhớ rằng chúng ta có công thức cập nhật  $x_{k+1} = x_k - f'(x_k)/f''(x_k)$ . Chia khai triển Taylor ở trên cho  $f''(x_k)$ , ta thu được

$$e_k - f'(x_k)/f''(x_k) = \frac{1}{2} e_k^2 f'''(\xi_k)/f''(x_k). \quad (13.7.11)$$

Thay vào phương trình cập nhật sẽ dẫn đến ràng buộc  $e_{k+1} \leq e_k^2 f'''(\xi_k)/f'(x_k)$ . Do đó, khi nằm trong miền ràng buộc  $f'''(\xi_k)/f''(x_k) \leq c$ , ta sẽ có sai số giảm theo bình phương  $e_{k+1} \leq ce_k^2$ .

Bên cạnh đó, các nhà nghiên cứu tối ưu hóa gọi đây là hội tụ *tuyến tính*, còn điều kiện  $e_{k+1} \leq \alpha e_k$  được gọi là tốc độ hội tụ *không đổi*. Lưu ý rằng phân tích này đi kèm với một số lưu ý: Chúng ta không thực sự biết rằng khi nào mình sẽ tiến tới được vùng hội tụ nhanh. Thay vào đó, ta chỉ biết rằng một khi đến được đó, việc hội tụ sẽ xảy ra rất nhanh chóng.Thêm nữa, điều này yêu cầu  $f$  được xử lý tốt ở các đạo hàm bậc cao. Nó đảm bảo không có bất cứ một tính chất “bất ngờ” nào của  $f$  có thể dẫn đến sự thay đổi giá trị của nó.

## Tiền Điều kiện

Không có gì ngạc nhiên khi việc tính toán và lưu trữ toàn bộ ma trận Hessian là rất tốn kém. Do đó ta cần tìm kiếm một phương pháp thay thế. Một cách để cải thiện vấn đề này là tránh tính toán toàn bộ ma trận Hessian, chỉ tính toán các giá trị thuộc *đường chéo*. Mặc dù cách trên không tốt bằng phương pháp Newton hoàn chỉnh nhưng vẫn tốt hơn nhiều so với không sử dụng nó. Hơn nữa, ước lượng các giá trị đường chéo chính là thứ thúc đẩy sự đổi mới trong các thuật toán tối ưu hóa hạ gradient ngẫu nhiên. Thuật toán cập nhật sẽ có dạng

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \text{diag}(H_f)^{-1} \nabla \mathbf{x}. \quad (13.7.12)$$

Để thấy tại sao điều này có thể là một ý tưởng tốt, ta ví dụ có hai biến số biểu thị chiều cao, một biến với đơn vị mm, biến còn lại với đơn vị km. Với cả hai đơn vị đo, khi quy đổi ra mét, chúng ta đều có sự sai lệch lớn trong việc tham số hóa. Sử dụng tiền điều kiện sẽ loại bỏ vấn đề này. Tiền điều kiện một cách hiệu quả cùng hạ gradient giúp chọn ra các tốc độ học khác nhau cho từng trục tọa độ.

## Hạ gradient cùng Tìm kiếm Đường thẳng

Một trong những vấn đề chính của hạ gradient là chúng ta có thể vượt quá khỏi mục tiêu hoặc không đạt đủ sự tiến bộ. Có một cách khắc phục đơn giản cho vấn đề này là sử dụng tìm kiếm đường thẳng (*line search*) kết hợp với hạ gradient.

Chúng ta sử dụng hướng được cho bởi  $\nabla f(\mathbf{x})$  và sau đó dùng tìm kiếm nhị phân để tìm ra độ dài bước  $\eta$  có thể cực tiểu hóa  $f(\mathbf{x} - \eta \nabla f(\mathbf{x}))$ .

Thuật toán này sẽ hội tụ nhanh chóng (xem phân tích và chứng minh ở (Boyd & Vandenberghe, 2004)). Tuy nhiên, đối với mục đích của học sâu thì nó không thực sự khả thi, lý do là mỗi bước của tìm kiếm đường thẳng sẽ yêu cầu chúng ta ước lượng hàm mục tiêu trên toàn bộ tập dữ liệu. Điều này quá tốn kém để có thể thực hiện.

### 13.7.4 Tổng kết

- Tốc độ học rất quan trọng. Quá lớn sẽ khiến việc tối ưu hóa phân kỳ, quá nhỏ sẽ không thu được sự tiến bộ nào.
- Hạ gradient có thể bị kẹt tại cực tiểu cục bộ.
- Trong bài toán nhiều chiều, tinh chỉnh việc học tốc độ học sẽ phức tạp.
- Tiền điều kiện có thể giúp trong việc tinh chỉnh thang đo.
- Phương pháp Newton nhanh hơn rất nhiều *một khi* hoạt động trên bài toán lồi phù hợp.
- Hãy cẩn trọng trong việc dùng phương pháp Newton cho các bài toán không lồi mà không tinh chỉnh.

### 13.7.5 Bài tập

1. Hãy thử các tốc độ học, hàm mục tiêu khác nhau cho hạ gradient.
2. Khởi tạo tìm kiếm đường thẳng để cực tiểu hóa hàm lồi trong khoảng  $[a, b]$ .
  - Bạn có cần đạo hàm để tìm kiếm nhị phân không, ví dụ, để quyết định xem sẽ chọn  $[a, (a + b)/2]$  hay  $[(a + b)/2, b]$ ?
  - Tốc độ hội tụ của thuật toán nhanh chậm thế nào?
  - Hãy khởi tạo thuật toán và áp dụng nó để cực tiểu hóa  $\log(\exp(x) + \exp(-2 * x - 3))$ .
3. Thiết kế một hàm mục tiêu thuộc  $\mathbb{R}^2$  mà việc hạ gradient rất chậm. Gợi ý: sử dụng trực tọa độ có thang đo khác nhau.
4. Khởi tạo một phiên bản nhỏ gọn của phương pháp Newton sử dụng tiền điều kiện:
  - Dùng ma trận đường chéo Hessian làm tiền điều kiện.
  - Sử dụng các giá trị tuyệt đối của nó thay vì các giá trị có dấu.
  - Áp dụng điều này cho bài toán phía trên.
5. Áp dụng thuật toán phía trên cho các hàm mục tiêu (lồi lẫn không lồi). Điều gì sẽ xảy ra nếu xoay các trực tọa độ một góc 45 độ?

### 13.7.6 Thảo luận

- Tiếng Anh - MXNet<sup>232</sup>
- Tiếng Anh - Pytorch<sup>233</sup>
- Tiếng Việt<sup>234</sup>

### 13.7.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Thanh Hòa
- Võ Tấn Phát

## 13.8 Hạ Gradient Ngẫu nhiên

Trong phần này chúng tôi sẽ giới thiệu các nguyên tắc cơ bản của hạ gradient ngẫu nhiên.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx
npx.set_np()
```

<!--

---

<sup>232</sup> <https://discuss.d2l.ai/t/351>

<sup>233</sup> <https://discuss.d2l.ai/t/491>

<sup>234</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 13.8.1 Cập nhật Gradient Ngẫu nhiên

Trong học sâu, hàm mục tiêu thường là trung bình của các hàm mất mát cho từng mẫu trong tập huấn luyện. Giả sử tập huấn luyện có  $n$  mẫu,  $f_i(\mathbf{x})$  là hàm mất mát của mẫu thứ  $i$ , và vector tham số là  $\mathbf{x}$ . Ta có hàm mục tiêu

$$f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n f_i(\mathbf{x}). \quad (13.8.1)$$

Gradient của hàm mục tiêu tại  $\mathbf{x}$  được tính như sau

$$\nabla f(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}). \quad (13.8.2)$$

Nếu hạ gradient được sử dụng, chi phí tính toán cho mỗi vòng lặp độc lập là  $\mathcal{O}(n)$ , tăng tuyến tính với  $n$ . Do đó, với tập huấn luyện lớn, chi phí của hạ gradient cho mỗi vòng lặp sẽ rất cao.

Hạ gradient ngẫu nhiên (*stochastic gradient descent* - SGD) giúp giảm chi phí tính toán ở mỗi vòng lặp. Ở mỗi vòng lặp, ta lấy ngẫu nhiên một mẫu dữ liệu có chỉ số  $i \in \{1, \dots, n\}$  theo phân phối đều, và chỉ cập nhật  $\mathbf{x}$  bằng gradient  $\nabla f_i(\mathbf{x})$ :

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}). \quad (13.8.3)$$

Ở đây,  $\eta$  là tốc độ học. Ta có thể thấy rằng chi phí tính toán cho mỗi vòng lặp giảm từ  $\mathcal{O}(n)$  của hạ gradient xuống còn hằng số  $\mathcal{O}(1)$ . Nên nhớ rằng gradient ngẫu nhiên  $\nabla f_i(\mathbf{x})$  là một ước lượng không thiên lệch (*unbiased*) của gradient  $\nabla f(\mathbf{x})$ .

$$\mathbb{E}_i \nabla f_i(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \quad (13.8.4)$$

Do đó, trên trung bình, gradient ngẫu nhiên là một ước lượng gradient tốt.

Bây giờ, ta mô phỏng hạ gradient ngẫu nhiên bằng cách thêm nhiễu ngẫu nhiên với trung bình bằng 0 và phương sai bằng 1 vào gradient và so sánh với phương pháp hạ gradient.

```
f = lambda x1, x2: x1 ** 2 + 2 * x2 ** 2 # Objective
gradf = lambda x1, x2: (2 * x1, 4 * x2) # Gradient

def sgd(x1, x2, s1, s2):
    global lr # Learning rate scheduler
    (g1, g2) = gradf(x1, x2)
    # Simulate noisy gradient
    g1 += np.random.normal(0.0, 1, (1,))
    g2 += np.random.normal(0.0, 1, (1,))
    eta_t = eta * lr() # Learning rate at time t
    return (x1 - eta_t * g1, x2 - eta_t * g2, 0, 0) # Update variables

eta = 0.1
lr = (lambda: 1) # Constant learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))
```

Như có thể thấy, quỹ đạo của các biến trong SGD dao động mạnh hơn hạ gradient ở phần trước. Điều này là do bản chất ngẫu nhiên của gradient. Tức là, ngay cả khi tới gần giá trị cực tiểu, ta vẫn gặp phải sự bất định gây ra bởi gradient ngẫu nhiên  $\eta \nabla f_i(\mathbf{x})$ . Thậm chí sau 50 bước thì chất lượng vẫn không tốt lắm. tệ hơn, nó vẫn sẽ không cải thiện với nhiều bước hơn (chúng tôi khuyến khích bạn đọc thử nghiệm với số lượng bước lớn hơn để tự xác nhận điều này). Ta chỉ còn một lựa chọn duy nhất — thay đổi tốc độ học  $\eta$ . Tuy nhiên, nếu chọn giá trị quá nhỏ, ta sẽ không đạt được bất kỳ tiến triển đáng kể nào ở những bước đầu tiên. Mặt khác, nếu chọn giá trị quá lớn, ta sẽ không thu được nghiệm tốt, như đã thấy ở trên. Cách duy nhất để giải quyết hai mục tiêu xung đột này là giảm tốc độ học *một cách linh hoạt* trong quá trình tối ưu.

Đây cũng là lý do cho việc thêm hàm tốc độ học lr vào hàm bước sgd. Trong ví dụ trên, chức năng định thời tốc độ học (*learning rate scheduling*) không được kích hoạt vì ta đặt hàm lr bằng một hằng số, tức lr = (lambda: 1).

### 13.8.2 Tốc độ học Linh hoạt

Thay thế  $\eta$  bằng tốc độ học phụ thuộc thời gian  $\eta(t)$  sẽ khiến việc kiểm soát sự hội tụ của thuật toán tối ưu trở nên phức tạp hơn. Cụ thể, ta cần tìm ra mức độ suy giảm  $\eta$  hợp lý. Nếu giảm quá nhanh, quá trình tối ưu sẽ ngừng quá sớm. Nếu giảm quá chậm, ta sẽ lãng phí rất nhiều thời gian cho việc tối ưu. Có một vài chiến lược cơ bản được sử dụng để điều chỉnh  $\eta$  theo thời gian (ta sẽ thảo luận về các chiến lược cao cấp hơn trong chương sau):

$$\begin{aligned}\eta(t) &= \eta_i \text{ if } t_i \leq t \leq t_{i+1} && \text{hằng số theo khoảng} \\ \eta(t) &= \eta_0 \cdot e^{-\lambda t} && \text{lũy thừa} \\ \eta(t) &= \eta_0 \cdot (\beta t + 1)^{-\alpha} && \text{đa thức}\end{aligned}\tag{13.8.5}$$

Trong trường hợp đầu tiên, ta giảm tốc độ học bất cứ khi nào tiến trình tối ưu bị đình trệ. Đây là một chiến lược phổ biến để huấn luyện các mạng sâu. Ngoài ra, ta có thể làm giảm tốc độ học nhanh hơn bằng suy giảm theo lũy thừa. Thật không may, phương pháp này dẫn đến việc dừng tối ưu quá sớm trước khi thuật toán hội tụ. Một lựa chọn phổ biến khác là suy giảm đa thức với  $\alpha = 0.5$ . Trong trường hợp tối ưu lồi, có các chứng minh cho thấy giá trị này cho kết quả tốt. Hãy cùng xem nó hoạt động như thế nào trong thực tế.

```
def exponential():
    global ctr
    ctr += 1
    return math.exp(-0.1 * ctr)

ctr = 1
lr = exponential # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=1000))
```

Như dự đoán, giá trị phương sai của các tham số giảm đáng kể. Tuy nhiên, suy giảm lũy thừa không hội tụ tới nghiệm tối ưu  $\mathbf{x} = (0, 0)$ . Thậm chí sau 1000 vòng lặp, nghiệm tìm được vẫn cách nghiệm tối ưu rất xa. Trên thực tế, thuật toán này không hội tụ được. Mặt khác, nếu ta sử dụng suy giảm đa thức trong đó tốc độ học suy giảm tỉ lệ nghịch với căn bình phương thời gian, thuật toán hội tụ tốt.

```
def polynomial():
    global ctr
```

(continues on next page)

```

ctr += 1
return (1 + 0.1 * ctr)**(-0.5)

ctr = 1
lr = polynomial # Set up learning rate
d2l.show_trace_2d(f, d2l.train_2d(sgd, steps=50))

```

Vẫn còn có rất nhiều lựa chọn khác để thiết lập tốc độ học. Ví dụ, ta có thể bắt đầu với tốc độ học nhỏ, sau đó tăng nhanh rồi tiếp tục giảm nhưng với tốc độ chậm hơn. Ta cũng có thể thiết lập tốc độ học tăng và giảm luân phiên. Có rất nhiều cách khác nhau để định thời tốc độ học. Nay giờ, chúng ta hãy tập trung vào thiết lập tốc độ học trong điều kiện lồi mà ta có thể phân tích lý thuyết. Với bài toán không lồi tổng quát, rất khó để đảm bảo được mức hội tụ có ý nghĩa, vì nói chung các bài toán tối ưu phi tuyến không lồi đều thuộc dạng NP-hard. Để tìm hiểu thêm, tham khảo các ví dụ trong [tập bài giảng<sup>235</sup>](#) của Tibshirani năm 2015.

### 13.8.3 Phân tích Hội tụ cho Hàm mục tiêu Lồi

Đây là phần đọc thêm để mang lại cái nhìn trực quan hơn về bài toán, giới hạn lại trong một cách chứng minh đơn giản được trình bày trong ([Nesterov & Vial, 2000](#)). Cũng có những cách chứng minh nâng cao hơn, ví dụ như khi hàm mục tiêu được định nghĩa tốt. :cite: Hazan.Rakhlin.Bartlett.2008 chỉ ra rằng với các hàm lồi chặt, cụ thể là các hàm có cận dưới là  $\mathbf{x}^\top \mathbf{Q}\mathbf{x}$ , ta có thể cực tiểu hóa chúng chỉ với một số lượng nhỏ bước lặp trong khi giảm tốc độ học theo  $\eta(t) = \eta_0 / (\beta t + 1)$ . Thật không may, trường hợp này không xảy ra trong học sâu, trên thực tế mức độ giảm tốc độ học chậm hơn rất nhiều.

Xét trường hợp

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}). \quad (13.8.6)$$

Cụ thể, ta giả sử  $\mathbf{x}_t$  được lấy từ phân phối  $P(\mathbf{x})$  và  $l(\mathbf{x}, \mathbf{w})$  là hàm lồi theo biến  $\mathbf{w}$  với mọi  $\mathbf{x}$ . Cuối cùng, ta kí hiệu

$$R(\mathbf{w}) = E_{\mathbf{x} \sim P}[l(\mathbf{x}, \mathbf{w})] \quad (13.8.7)$$

là giá trị mất mát kỳ vọng và  $R^*$  là cực tiểu của hàm mất mát theo  $\mathbf{w}$ . Ta kí hiệu  $\mathbf{w}^*$  là nghiệm tại cực tiểu (*minimizer*) với giả định giá trị này tồn tại trong miền xác định. Trong trường hợp này, chúng ta lần theo khoảng cách giữa tham số hiện tại  $\mathbf{w}_t$  và nghiệm cực tiểu  $\mathbf{w}^*$ , và xem liệu giá trị này có cải thiện theo thời gian không:

$$\begin{aligned} \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 &= \|\mathbf{w}_t - \eta_t \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}) - \mathbf{w}^*\|^2 \\ &= \|\mathbf{w}_t - \mathbf{w}^*\|^2 + \eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 - 2\eta_t \langle \mathbf{w}_t - \mathbf{w}^*, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}) \rangle. \end{aligned} \quad (13.8.8)$$

Gradient  $\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})$  có cận trên là một hằng số Lipschitz  $L$ , do đó ta có

$$\eta_t^2 \|\partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w})\|^2 \leq \eta_t^2 L^2. \quad (13.8.9)$$

<sup>235</sup> <https://www.stat.cmu.edu/~ryantibs/convexopt-F15/lectures/26-nonconvex.pdf>

Điều chúng ta thực sự quan tâm là khoảng cách giữa  $\mathbf{w}_t$  và  $\mathbf{w}^*$  thay đổi như thế nào *theo kỳ vọng*. Thực tế, với chuỗi các bước bất kỳ, khoảng cách này cũng có thể tăng lên, tùy thuộc vào giá trị của  $\mathbf{x}_t$  mà ta gặp phải. Do đó cần xác định cận cho tích vô hướng. Từ tính chất lồi, ta có

$$l(\mathbf{x}_t, \mathbf{w}^*) \geq l(\mathbf{x}_t, \mathbf{w}_t) + \langle \mathbf{w}^* - \mathbf{w}_t, \partial_{\mathbf{w}} l(\mathbf{x}_t, \mathbf{w}_t) \rangle. \quad (13.8.10)$$

Kết hợp hai bất đẳng thức trên, ta tìm được cận cho khoảng cách giữa các tham số tại bước  $t+1$  như sau:

$$\|\mathbf{w}_t - \mathbf{w}^*\|^2 - \|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2 \geq 2\eta_t(l(\mathbf{x}_t, \mathbf{w}_t) - l(\mathbf{x}_t, \mathbf{w}^*)) - \eta_t^2 L^2. \quad (13.8.11)$$

Điều này có nghĩa quá trình học vẫn sẽ cải thiện miễn là hiệu số giữa hàm mất mát hiện tại và giá trị mất mát tối ưu vẫn lớn hơn  $\eta_t L^2$ . Để đảm bảo hàm mất mát hội tụ về 0, tốc độ học  $\eta_t$  cũng cần phải giảm dần.

Tiếp theo chúng ta tính giá trị kỳ vọng cho biểu thức trên

$$E_{\mathbf{w}_t} [\|\mathbf{w}_t - \mathbf{w}^*\|^2] - E_{\mathbf{w}_{t+1} | \mathbf{w}_t} [\|\mathbf{w}_{t+1} - \mathbf{w}^*\|^2] \geq 2\eta_t [E[R[\mathbf{w}_t]] - R^*] - \eta_t^2 L^2. \quad (13.8.12)$$

Ở bước cuối cùng, ta tính tổng các bất đẳng thức trên cho mọi  $t \in \{t, \dots, T\}$ . Rút gọn tổng và bỏ qua các hạng tử thấp hơn, ta có

$$\|\mathbf{w}_0 - \mathbf{w}^*\|^2 \geq 2 \sum_{t=1}^T \eta_t [E[R[\mathbf{w}_t]] - R^*] - L^2 \sum_{t=1}^T \eta_t^2. \quad (13.8.13)$$

Lưu ý rằng ta tận dụng  $\mathbf{w}_0$  cho trước và do đó có thể bỏ qua giá trị kỳ vọng. Cuối cùng, ta định nghĩa

$$\bar{\mathbf{w}} := \frac{\sum_{t=1}^T \eta_t \mathbf{w}_t}{\sum_{t=1}^T \eta_t}. \quad (13.8.14)$$

Từ đó, theo tính chất lồi, ta có

$$\sum_t \eta_t E[R[\mathbf{w}_t]] \geq \sum_t \eta_t \cdot [E[\bar{\mathbf{w}}]]. \quad (13.8.15)$$

Thay vào bất đẳng thức ở trên, ta tìm được cận

$$[E[\bar{\mathbf{w}}]] - R^* \leq \frac{r^2 + L^2 \sum_{t=1}^T \eta_t^2}{2 \sum_{t=1}^T \eta_t}. \quad (13.8.16)$$

Trong đó  $r^2 := \|\mathbf{w}_0 - \mathbf{w}^*\|^2$  là khoảng cách giới hạn giữa giá trị khởi tạo của các tham số và kết quả cuối cùng. Nói tóm lại, tốc độ hội tụ phụ thuộc vào tốc độ thay đổi của hàm mất mát thông qua hằng số Lipschitz  $L$  và khoảng cách  $r$  giữa giá trị ban đầu so với giá trị tối ưu. Chú ý rằng giới hạn ở trên được kí hiệu bởi  $\bar{\mathbf{w}}$  thay vì  $\mathbf{w}_T$  do  $\bar{\mathbf{w}}$  là quỹ đạo tối ưu được làm mượt. Hãy cùng phân tích một số cách lựa chọn  $\eta_t$ .

- Thời điểm xác định.** Với mỗi  $r, L$  và  $T$  xác định ta có thể chọn  $\eta = r/L\sqrt{T}$ . Biểu thức này dẫn tới giới hạn trên  $rL(1 + 1/T)/2\sqrt{T} < rL/\sqrt{T}$ . Điều này nghĩa là hàm sẽ hội tụ đến nghiệm tối ưu với tốc độ  $\mathcal{O}(1/\sqrt{T})$ .

- **Thời điểm chưa xác định.** Khi muốn nghiệm tốt cho *bất kì* thời điểm  $T$  nào, ta có thể chọn  $\eta = \mathcal{O}(1/\sqrt{T})$ . Cách làm trên tốn thêm một thửa số logarit, dẫn tới giới hạn trên có dạng  $\mathcal{O}(\log T/\sqrt{T})$ .

Chú ý rằng đối với những hàm mất mát lồi tuyệt đối  $l(\mathbf{x}, \mathbf{w}') \geq l(\mathbf{x}, \mathbf{w}) + \langle \mathbf{w}' - \mathbf{w}, \partial_{\mathbf{w}} l(\mathbf{x}, \mathbf{w}) \rangle + \frac{\lambda}{2} \|\mathbf{w} - \mathbf{w}'\|^2$  ta có thể thiết kế quy trình tối ưu nhằm tăng tốc độ hội tụ nhanh hơn nữa. Thực tế, sự suy giảm theo cấp số mũ của  $\eta$  dẫn đến giới hạn có dạng  $\mathcal{O}(\log T/T)$ .

#### 13.8.4 Gradient ngẫu nhiên và Mẫu hữu hạn

Tới phần này, ta đi khá nhanh và chưa chặt chẽ khi thảo luận về hạ gradient ngẫu nhiên. Ta ngầm định lấy các đối tượng  $x_i$ , thường là cùng với nhãn  $y_i$  từ phân phối  $p(x, y)$  nào đó và sử dụng chúng để cập nhật các trọng số  $w$  theo cách nào đó. Cụ thể, với kích thước mẫu hữu hạn, ta 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 phép áp dụng SGD.

Tuy nhiên, đó thật ra không phải là cách ta đã làm. Trong các ví dụ đơn giản ở phần này ta chỉ thêm nhiễu vào gradient không ngẫu nhiên, tức giả sử đang có sẵn các cặp giá trị  $(x_i, y_i)$ . hóa ra cách làm đó khá hợp lý (xem phần bài tập để thảo luận chi tiết). Vấn đề là ở tất cả các thảo luận trước, ta không hề làm thế. Thay vào đó ta duyệt qua tất cả các đối tượng đúng một lần. Để hiểu tại sao quá trình này được ưa chuộng, hãy xét trường hợp ngược lại khi ta lấy có hoàn lại  $N$  mẫu từ một phân phối rời rạc. Xác suất phần tử  $i$  được chọn ngẫu nhiên là  $N^{-1}$ . Do đó xác suất chọn  $i$  ít nhất một lần là

$$P(\text{chọn } i) = 1 - P(\text{loại } i) = 1 - (1 - N^{-1})^N \approx 1 - e^{-1} \approx 0.63. \quad (13.8.17)$$

Tương tự, ta có thể chỉ ra rằng xác suất chọn một mẫu đúng một lần là  $\binom{N}{1} N^{-1} (1 - N^{-1})^{N-1} = \frac{N-1}{N} (1 - N^{-1})^N \approx e^{-1} \approx 0.37$ . Điều này gây tăng phương sai và giảm hiệu quả sử dụng dữ liệu so với lấy mẫu không hoàn lại. Do đó trong thực tế, ta thực hiện lấy mẫu không hoàn lại (và đây cũng là lựa chọn mặc định trong quyển sách này). Điều cuối cùng cần chú ý là mỗi lần duyệt lại tập dữ liệu, ta sẽ duyệt theo một thứ tự ngẫu nhiên khác.

#### 13.8.5 Tóm tắt

- Đối với các bài toán lồi, ta có thể chứng minh rằng Hạ Gradient Ngẫu nhiên sẽ hội tụ về nghiệm tối ưu với nhiều tốc độ học khác nhau.
- Trường hợp trên thường không xảy ra trong học sâu. Tuy nhiên việc phân tích các bài toán lồi cho ta kiến thức hữu ích để tiếp cận bài toán tối ưu, đó là giảm dần tốc độ học, dù không quá nhanh.
- Nhiều vấn đề xuất hiện khi tốc độ học quá lớn hoặc quá nhỏ. Trong thực tế, ta chỉ có thể tìm được tốc độ học thích hợp sau nhiều lần thử nghiệm.
- Khi kích thước tập huấn luyện tăng, chi phí tính toán cho mỗi lần lặp của hạ gradient cũng tăng theo, do đó SGD được ưa chuộng hơn trong trường hợp này.
- Trong SGD, không có sự đảm bảo tối ưu đối với các trường hợp không lồi do số cực tiểu cần phải kiểm tra có thể tăng theo cấp số nhân.

### 13.8.6 Bài tập

1. Hãy thử nghiệm với nhiều bộ định thời tốc độ học khác nhau trong SGD và với số vòng lặp khác nhau. Cụ thể, hãy vẽ biểu đồ khoảng cách tối ưu  $(0, 0)$  theo số vòng lặp.
2. Chứng minh rằng với hàm  $f(x_1, x_2) = x_1^2 + 2x_2^2$ , việc thêm nhiễu Gauss (*normal noise*) vào gradient tương đương với việc cực tiểu hóa hàm mất mát  $l(\mathbf{x}, \mathbf{w}) = (x_1 - w_1)^2 + 2(x_2 - w_2)^2$  trong đó  $x$  tuân theo phân phối chuẩn.
  - Suy ra kỳ vọng và phương sai của  $\mathbf{x}$ .
  - Chỉ ra rằng tính chất này có thể áp dụng tổng quát cho hàm mục tiêu  $f(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mu)^\top Q(\mathbf{x} - \mu)$  với  $Q \succeq 0$ .
3. So sánh sự hội tụ của SGD khi lấy mẫu không hoàn lại từ  $\{(x_1, y_1), \dots, (x_m, y_m)\}$  và khi lấy mẫu có hoàn lại.
4. Bạn sẽ thay đổi SGD thế nào nếu như một số gradient (hoặc một số toạ độ liên kết với nó) liên tục lớn hơn tất cả các gradient khác?
5. Giả sử  $f(x) = x^2(1 + \sin x)$ .  $f$  có bao nhiêu cực tiểu? Thay đổi hàm  $f$  sao cho để cực tiểu hóa giá trị hàm này, ta cần xét tất cả các điểm cực tiểu?

### 13.8.7 Thảo luận

- Tiếng Anh - MXNet<sup>236</sup>
- Tiếng Anh - Pytorch<sup>237</sup>
- Tiếng Việt<sup>238</sup>

### 13.8.8 Những người thực hiện

- Đoàn Võ Duy Thành
- Nguyễn Duy Du
- Phạm Minh Đức
- Nguyễn Văn Quang
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

<sup>236</sup> <https://discuss.d2l.ai/t/352>

<sup>237</sup> <https://discuss.d2l.ai/t/497>

<sup>238</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 13.9 Hạ Gradient Ngẫu nhiên theo Minibatch

Đến giờ, ta đã tiếp xúc với hai thái cực trong các phương pháp học dựa theo gradient: tại mỗi lượt Section 13.7 sử dụng toàn bộ tập dữ liệu để tính gradient và cập nhật tham số. Ngược lại, Section 13.8 xử lý từng điểm dữ liệu một để cập nhật các tham số. Mỗi phương pháp đều có mặt hạn chế riêng. Hạ Gradient có *hiệu suất dữ liệu* (*data efficiency*) thấp khi dữ liệu tương đối giống nhau. Hạ Gradient Ngẫu nhiên có *hiệu suất tính toán* (*computational efficiency*) thấp do CPU và GPU không được khai thác hết khả năng vector hóa. Điều này gợi ý rằng có thể có một phương pháp thích hợp ở giữa, và thực tế, ta đã sử dụng phương pháp đó trong các ví dụ đã thảo luận.

### 13.9.1 Vector hóa và Vùng nhớ đệm

Lý do sử dụng minibatch chủ yếu là vì hiệu suất tính toán song song giữa nhiều GPU và giữa nhiều máy chủ. Trong trường hợp này ta cần đưa ít nhất một ảnh vào mỗi GPU. Với 16 máy chủ và 8 GPU mỗi máy, ta có minibatch kích thước 128.

Vấn đề trở nên nhạy cảm hơn đối với GPU đơn hay ngay cả CPU đơn. Những thiết bị này có nhiều loại bộ nhớ, thường có nhiều loại đơn vị thực hiện tính toán và giới hạn băng thông giữa các đơn vị này cũng khác nhau. Ví dụ, một CPU có số lượng ít thanh ghi, bộ nhớ đệm L1, L2 và trong một số trường hợp có cả L3 (phần bộ nhớ được phân phối giữa các lõi của vi xử lý). Các bộ nhớ đệm đang tăng dần về kích thước và độ trễ (và cùng với đó là giảm băng thông). Nói vậy đủ thấy rằng vi xử lý có khả năng thực hiện nhiều tác vụ hơn so với những gì mà giao diện bộ nhớ chính (*main memory interface*) có thể cung cấp.

- Một CPU tốc độ 2GHz với 16 lõi và phép vector hóa AVX-512 có thể xử lý lên lõi  $2 \cdot 10^9 \cdot 16 \cdot 32 = 10^{12}$  byte mỗi giây. Khả năng của GPU dễ dàng vượt qua con số này cả trăm lần. Mặt khác, trong vi xử lý của máy chủ cỡ trung bình, băng thông có lẽ không vượt quá 100 GB/s, tức là chưa bằng một phần mười băng thông yêu cầu để đưa dữ liệu vào bộ xử lý. Vấn đề còn tồi tệ hơn khi ta xét đến việc không phải khả năng truy cập bộ nhớ nào cũng như nhau: đầu tiên, giao diện bộ nhớ thường rộng 64 bit hoặc hơn (ví dụ như trên GPU lên đến 384 bit), do đó việc đọc một byte duy nhất vẫn sẽ phải chịu chi phí giống như truy cập một khoảng bộ nhớ rộng hơn.
- Tổng chi phí cho lần truy cập đầu tiên là khá lớn, trong khi truy cập liên tiếp hao tổn ít (thường được gọi là đọc hàng loạt). Có rất nhiều điều cần lưu ý, ví dụ như lưu trữ đệm khi ta có nhiều điểm truy cập cuối (*sockets*), nhiều chiplet và các cấu trúc khác. Việc thảo luận chi tiết vấn đề trên nằm ngoài phạm vi của phần này. Bạn có thể tham khảo bài viết Wikipedia<sup>239</sup> này để hiểu sâu hơn.

Cách để giảm bớt những ràng buộc trên là sử dụng hệ thống cấp bậc (*hierarchy*) của các vùng nhớ đệm trong CPU, các vùng nhớ này đều nhanh để có thể cung cấp dữ liệu cho vi xử lý. Đây chính là động lực đằng sau việc sử dụng batch trong học sâu. Để đơn giản, xét phép nhân hai ma trận  $\mathbf{A} = \mathbf{BC}$ . Để tính  $\mathbf{A}$  ta có khá nhiều lựa chọn, ví dụ như:

1. Ta có thể tính  $\mathbf{A}_{ij} = \mathbf{B}_{i,:} \mathbf{C}_{:,j}^\top$ , tức là tính từng phần tử bằng tích vô hướng.
2. Ta có thể tính  $\mathbf{A}_{:,j} = \mathbf{B}\mathbf{C}_{:,j}^\top$ , tức là tính theo từng cột. Tương tự, ta có thể tính  $\mathbf{A}$  theo từng hàng  $\mathbf{A}_{i,:}$ .
3. Ta đơn giản có thể tính  $\mathbf{A} = \mathbf{BC}$ .
4. Ta có thể chia  $\mathbf{B}$  và  $\mathbf{C}$  thành nhiều khối ma trận nhỏ hơn và tính  $\mathbf{A}$  theo từng khối một.

<sup>239</sup> [https://en.wikipedia.org/wiki/Cache\\_hierarchy](https://en.wikipedia.org/wiki/Cache_hierarchy)

Nếu sử dụng cách đầu tiên, ta cần sao chép một vector cột và một vector hàng vào CPU cho mỗi lần tính phần tử  $A_{ij}$ . tệ hơn nữa, do các phần tử của ma trận được lưu thành một dãy liên tục dưới bộ nhớ, ta buộc phải truy cập nhiều vùng nhớ rời rạc khi đọc một trong hai vector từ bộ nhớ. Cách thứ hai tốt hơn nhiều. Theo cách này, ta có thể giữ vector cột  $C_{\cdot, j}$  trong vùng nhớ đệm của CPU trong khi ta tiếp tục quét qua  $B$ . Cách này chỉ cần nửa băng thông cần thiết của bộ nhớ, do đó truy cập nhanh hơn. Đương nhiên cách thứ ba là tốt nhất. Đáng tiếc rằng đa số ma trận quá lớn để có thể đưa vào vùng nhớ đệm (dù sao đây cũng chính là điều ta đang thảo luận). Cách thứ tư là một phương pháp thay thế khá tốt: đưa các khối của ma trận vào vùng nhớ đệm và thực hiện phép nhân cục bộ. Các thư viện đã được tối ưu sẽ thực hiện việc này giúp chúng ta. Hãy xem xét hiệu suất của từng phương pháp trong thực tế.

Ngoài hiệu suất tính toán, chi phí tính toán phát sinh đến từ Python và framework học sâu cũng đáng cân nhắc. Mỗi lần ta thực hiện một câu lệnh, bộ thông dịch Python gửi một câu lệnh đến MXNet để chèn câu lệnh đó vào đồ thị tính toán và thực thi nó theo đúng lịch trình. Chi phí đó có thể khá bất lợi. Nói ngắn gọn, nên áp dụng vector hóa (và ma trận) bất cứ khi nào có thể.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

timer = d2l.Timer()
A = np.zeros((256, 256))
B = np.random.normal(0, 1, (256, 256))
C = np.random.normal(0, 1, (256, 256))
```

Phép nhân theo từng phần tử chỉ đơn giản là duyệt qua tất cả các hàng và cột của  $\mathbf{B}$  và  $\mathbf{C}$  theo thứ tự rồi gán kết quả cho  $\mathbf{A}$ .

```
# Compute A = BC one element at a time
timer.start()
for i in range(256):
    for j in range(256):
        A[i, j] = np.dot(B[i, :], C[:, j])
A.wait_to_read()
timer.stop()
```

Một cách nhanh hơn là nhân theo từng cột.

```
# Compute A = BC one column at a time
timer.start()
for j in range(256):
    A[:, j] = np.dot(B, C[:, j])
A.wait_to_read()
timer.stop()
```

Cuối cùng, cách hiệu quả nhất là thực hiện toàn bộ phép nhân trong một khối. Hãy thử xem tốc độ tương ứng của phương pháp này là bao nhiêu.

```
# Compute A = BC in one go
timer.start()
A = np.dot(B, C)
```

(continues on next page)

```

A.wait_to_read()
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}')

```

### 13.9.2 Minibatch

Ở các phần trước ta mặc nhiên đọc dữ liệu theo *minibatch* thay vì từng điểm dữ liệu đơn lẻ để cập nhật các tham số. Ta có thể giải thích ngắn gọn mục đích như sau. Xử lý từng điểm dữ liệu đơn lẻ đòi hỏi phải thực hiện rất nhiều phép nhân ma trận với vector (hay thậm chí vector với vector). Cách này khá tốn kém và đồng thời phải chịu thêm chi phí khá lớn đến từ các framework học sâu bên dưới. Vấn đề này xảy ra ở cả lúc đánh giá một mạng với dữ liệu mới (thường gọi là suy luận - *inference*) và khi tính toán gradient để cập nhật các tham số. Tức là vấn đề xảy ra mỗi khi ta thực hiện  $\mathbf{w} \leftarrow \mathbf{w} - \eta_t \mathbf{g}_t$  trong đó

$$\mathbf{g}_t = \partial_{\mathbf{w}} f(\mathbf{x}_t, \mathbf{w}) \quad (13.9.1)$$

Ta có thể tăng hiệu suất *tính toán* của phép tính này bằng cách áp dụng nó trên mỗi minibatch dữ liệu. Tức là ta thay thế gradient  $\mathbf{g}_t$  trên một điểm dữ liệu đơn lẻ bằng gradient trên một batch nhỏ.

$$\mathbf{g}_t = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}) \quad (13.9.2)$$

Hãy thử xem phương pháp trên tác động thế nào đến các tính chất thống kê của  $\mathbf{g}_t$ : do cả  $\mathbf{x}_t$  và tất cả các phần tử trong minibatch  $\mathcal{B}_t$  được lấy ra từ tập huấn luyện với xác suất như nhau, kỳ vọng của gradient là không đổi. Mặt khác, phương sai giảm một cách đáng kể. Do gradient của minibatch là trung bình của  $b := |\mathcal{B}_t|$  gradient độc lập, độ lệch chuẩn của nó giảm đi theo hệ số  $b^{-\frac{1}{2}}$ . Đây là một điều tốt, cách cập nhật này có độ tin cậy gần bằng việc lấy gradient trên toàn bộ tập dữ liệu.

Tùy ý trên, ta sẽ nhanh chóng cho rằng chọn minibatch  $\mathcal{B}_t$  lớn luôn là tốt nhất. Tiếc rằng đến một mức độ nào đó, độ lệch chuẩn sẽ giảm không đáng kể so với chi phí tính toán tăng tuyến tính. Do đó trong thực tế, ta sẽ chọn kích thước minibatch đủ lớn để hiệu suất tính toán cao trong khi vẫn đủ để đưa vào bộ nhớ của GPU. Để minh họa quá trình lưu trữ này, hãy xem đoạn mã nguồn dưới đây. Trong đó ta vẫn thực hiện phép nhân ma trận với ma trận, tuy nhiên lần này ta tách thành từng minibatch 64 cột.

```

timer.start()
for j in range(0, 256, 64):
    A[:, j:j+64] = np.dot(B, C[:, j:j+64])
timer.stop()
print(f'performance in Gigaflops: block {2 / timer.times[3]:.3f}')

```

Có thể thấy quá trình tính toán trên minibatch về cơ bản có hiệu suất gần bằng thực hiện trên toàn ma trận. Tuy nhiên, cần lưu ý rằng Trong Section 9.5 ta sử dụng một loại điều chuẩn phụ thuộc chặt chẽ vào phương sai của minibatch. Khi tăng kích thước minibatch, phương sai giảm xuống và cùng với đó là lợi ích của việc thêm nhiễu (*noise-injection*) cũng giảm theo do phương pháp chuẩn hóa theo batch. Đọc (Ioffe, 2017) để biết chi tiết cách chuyển đổi giá trị và tính các số hạng phù hợp.

### 13.9.3 Đọc Tập dữ liệu

Hãy xem cách tạo các minibatch từ dữ liệu một cách hiệu quả như thế nào. Trong đoạn mã nguồn dưới ta sử dụng tập dữ liệu được phát triển bởi NASA để kiểm tra tiếng ồn từ các máy bay khác nhau<sup>240</sup> để so sánh các thuật toán tối ưu. Để thuận tiện ta chỉ sử dụng 1,500 ví dụ đầu tiên. Tập dữ liệu được tẩy trắng (whitened) để xử lý, tức là với mỗi tọa độ ta trừ đi giá trị trung bình và chuyển đổi giá trị phuơng sai về 1.

```
#@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 = (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
```

### 13.9.4 Lập trình từ đầu

Hãy nhớ lại cách lập trình SGD theo minibatch trong Section 5.2. Trong phần tiếp theo, chúng tôi sẽ trình bày cách lập trình tổng quát hơn một chút. Để thuận tiện, hàm lập trình SGD và các thuật toán tối ưu khác được giới thiệu sau trong chương này sẽ có danh sách tham số giống nhau. Cụ thể, chúng ta thêm trạng thái đầu vào states và đặt siêu tham số trong từ điển hyperparams. Bên cạnh đó, chúng ta sẽ tính giá trị mất mát trung bình của từng minibatch trong hàm huấn luyện, từ đó không cần phải chia gradient cho kích thước batch trong thuật toán tối ưu nữa.

```
def sgd(params, states, hyperparams):
    for p in params:
        p[:] -= hyperparams['lr'] * p.grad
```

Tiếp theo, chúng ta lập trình một hàm huấn luyện tổng quát, sử dụng được cho cả các thuật toán tối ưu khác được giới thiệu sau trong chương này. Hàm sẽ khởi tạo một mô hình hồi quy tuyến tính và có thể được sử dụng để huấn luyện mô hình với SGD theo minibatch và các thuật toán khác.

```
#@save
def train_ch11(trainer_fn, states, hyperparams, data_iter,
               feature_dim, num_epochs=2):
    # Initialization
    w = np.random.normal(scale=0.01, size=(feature_dim, 1))
    b = np.zeros(1)
    w.attach_grad()
    b.attach_grad()
    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()
```

(continues on next page)

<sup>240</sup> <https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise>

```

for _ in range(num_epochs):
    for X, y in data_iter:
        with autograd.record():
            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 cùng quan sát quá trình tối ưu của thuật toán hạ gradient theo toàn bộ batch. Ta có thể sử dụng toàn bộ batch bằng cách thiết lập kích thước minibatch bằng 1500 (chính là tổng số mẫu). Kết quả là các tham số mô hình chỉ được cập nhật một lần duy nhất trong mỗi epoch. Có thể thấy không có tiến triển đáng kể nào, sau 6 epoch việc tối ưu bị ngừng trệ.

```

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)

```

Khi kích thước batch bằng 1, chúng ta sử dụng thuật toán SGD để tối ưu. Để đơn giản hóa việc lập trình, chúng ta cố định tốc độ học bằng một hằng số (có giá trị nhỏ). Trong SGD, các tham số mô hình được cập nhật bất cứ khi nào một mẫu huấn luyện được xử lý. Trong trường hợp này, sẽ có 1500 lần cập nhật trong mỗi epoch. Có thể thấy, sự suy giảm giá trị của hàm mục tiêu chậm lại sau một epoch. Mặc dù cả hai thuật toán cùng xử lý 1500 mẫu trong một epoch, SGD tốn thời gian hơn hạ gradient trong thí nghiệm trên. Điều này là do SGD cập nhật các tham số thường xuyên hơn và kém hiệu quả khi xử lý đơn lẻ từng mẫu.

```
sgd_res = train_sgd(0.005, 1)
```

Cuối cùng, khi kích thước batch bằng 100, chúng ta sử dụng thuật toán SGD theo minibatch để tối ưu. Thời gian cần thiết cho mỗi epoch ngắn hơn thời gian tương ứng của SGD và hạ gradient theo toàn bộ batch.

```
mini1_res = train_sgd(.4, 100)
```

Giảm kích thước batch bằng 10, thời gian cho mỗi epoch tăng vì thực thi tính toán trên mỗi batch kém hiệu quả hơn.

```
mini2_res = train_sgd(.05, 10)
```

Cuối cùng, chúng ta so sánh tương quan thời gian và giá trị hàm mất mát trong bốn thí nghiệm trên. Có thể thấy, dù hội tụ nhanh hơn GD về số mẫu được xử lý, nhưng SGD tốn nhiều thời gian hơn để đạt được cùng giá trị mất mát như GD vì thuật toán này tính toán gradient trên từng mẫu một. Thuật toán SGD theo minibatch có thể cân bằng giữa tốc độ hội tụ và hiệu quả tính toán. Với

kích thước minibatch bằng 10, thuật toán này hiệu quả hơn SGD; và với kích thước minibatch bằng 100, thời gian chạy của thuật toán này thậm chí nhanh hơn cả GD.

```
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')
```

### 13.9.5 Lập trình Súc tích

Trong Gluon, chúng ta có thể sử dụng lớp Trainer để gọi các thuật toán tối ưu. Cách này được sử dụng để có thể lập trình một hàm huấn luyện tổng quát. Chúng ta sẽ sử dụng hàm này xuyên suốt các phần tiếp theo của chương.

```
#@save
def train_concise_ch11(tr_name, hyperparams, data_iter, num_epochs=2):
    # Initialization
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize(init.Normal(sigma=0.01))
    trainer = gluon.Trainer(net.collect_params(), tr_name, hyperparams)
    loss = gluon.loss.L2Loss()
    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:
            with autograd.record():
                l = loss(net(X), y)
            l.backward()
            trainer.step(X.shape[0])
            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')
```

Lặp lại thí nghiệm với kích thước batch bằng 10 sử dụng Gluon cho kết quả tương tự như trên.

```
data_iter, _ = get_data_ch11(10)
train_concise_ch11('sgd', {'learning_rate': 0.05}, data_iter)
```

### 13.9.6 Tóm tắt

- Vector hóa tính toán sẽ giúp mã nguồn hiệu quả hơn vì nó giảm chi phí phát sinh từ framework học sâu và tận dụng tính cục bộ của bộ nhớ và vùng nhớ đệm trên CPU và GPU tốt hơn.
- Tồn tại sự đánh đổi giữa hiệu quả về mặt thống kê của SGD và hiệu quả tính toán của việc xử lý các batch dữ liệu kích thước lớn cùng một lúc.
- Thuật toán SGD theo minibatch kết hợp cả hai lợi ích trên: hiệu quả tính toán và thống kê.
- Trong thuật toán đó ta xử lý các batch thu được từ hoàn vị ngẫu nhiên của dữ liệu huấn luyện (cụ thể, mỗi mẫu được xử lý chỉ một lần mỗi epoch theo thứ tự ngẫu nhiên).
- Suy giảm tốc độ học trong quá trình huấn luyện được khuyến khích sử dụng.
- Nhìn chung, SGD theo minibatch nhanh hơn SGD và hạ gradient về thời gian hội tụ.

### 13.9.7 Bài tập

1. Sửa đổi kích thước batch và tốc độ học, quan sát tốc độ suy giảm giá trị của hàm mục tiêu và thời gian cho mỗi epoch.
2. Đọc thêm tài liệu MXNet và sử dụng hàm `set_learning_rate` của lớp `Trainer` để giảm tốc độ học của SGD theo minibatch bằng  $1/10$  giá trị trước đó sau mỗi epoch.
3. Hãy so sánh SGD theo minibatch sử dụng một biến thể *lấy mẫu có hoàn lại* từ tập huấn luyện. Điều gì sẽ xảy ra?
4. Một ác thần đã sao chép tập dữ liệu của bạn mà không nói cho bạn biết (cụ thể, mỗi quan sát bị lặp lại hai lần và kích thước tập dữ liệu tăng gấp đôi so với ban đầu). Cách hoạt động của các thuật toán hạ gradient, SGD và SGD theo minibatch sẽ thay đổi như thế nào?

### 13.9.8 Thảo luận

- Tiếng Anh - MXNet<sup>241</sup>
- Tiếng Việt<sup>242</sup>

### 13.9.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Quang
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

<sup>241</sup> <https://discuss.d2l.ai/t/353>

<sup>242</sup> <https://forum.machinelearningcoban.com/c/d2l>

- Nguyễn Văn Cường

## 13.10 Động lượng

Trong Section 13.8 chúng ta đã thảo luận cách hoạt động của hạ gradient ngẫu nhiên, chỉ sử dụng một mẫu gradient có nhiều cho việc tối ưu. Cụ thể, khi có nhiều ta cần cực kỳ cẩn trọng trong việc chọn tốc độ học. Nếu ta giảm tốc độ học quá nhanh, việc hội tụ sẽ ngưng trệ. Nếu tốc độ học giảm chậm, sẽ khó hội tụ tại một kết quả đủ tốt vì nhiều sẽ đẩy điểm hội tụ ra xa điểm tối ưu.

### 13.10.1 Kiến thức Cơ bản

Trong phần này, ta sẽ cùng khám phá những thuật toán tối ưu hiệu quả hơn, đặc biệt là cho một số dạng bài toán tối ưu phổ biến trong thực tế.

#### Giá trị Trung bình Rò rỉ

Trong phần trước, ta đã thảo luận về hạ gradient ngẫu nhiên theo minibatch như một cách để tăng tốc độ tính toán. Đồng thời, kỹ thuật này cũng có một tác dụng phụ tốt là giúp giảm phương sai.

$$\mathbf{g}_{t,t-1} = \partial_{\mathbf{w}} \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} f(\mathbf{x}_i, \mathbf{w}_{t-1}) = \frac{1}{|\mathcal{B}_t|} \sum_{i \in \mathcal{B}_t} \mathbf{h}_{i,t-1}. \quad (13.10.1)$$

Ở đây để đơn giản kí hiệu, ta đặt  $\mathbf{h}_{i,t-1} = \partial_{\mathbf{w}} f(\mathbf{x}_i, \mathbf{w}_{t-1})$  là gradient của mẫu  $i$  với trọng số tại bước thời gian  $t - 1$ . Sẽ rất tốt nếu ta có thể tận dụng hơn nữa lợi ích từ việc giảm phương sai, hơn là chỉ lấy trung bình gradient trên minibatch. Một phương pháp để đạt được điều này đó là thay thế việc tính toán gradient bằng một giá trị “trung bình rò rỉ” (*leaky average*):

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_{t,t-1} \quad (13.10.2)$$

với  $\beta \in (0, 1)$ . Phương pháp này thay thế gradient tức thời bằng một giá trị được lấy trung bình trên các gradient trước đó.  $\mathbf{v}$  được gọi là *động lượng* (*momentum*). Động lượng tích luỹ các gradient trong quá khứ tương tự như cách một quả bóng nặng lăn xuống ngọn đồi sẽ tích hợp hết tất cả các lực tác động lên nó từ lúc bắt đầu. Để hiểu rõ hơn, hãy khai triển đệ quy  $\mathbf{v}_t$  thành

$$\mathbf{v}_t = \beta^2 \mathbf{v}_{t-2} + \beta \mathbf{g}_{t-1,t-2} + \mathbf{g}_{t,t-1} = \dots = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau,t-\tau-1}. \quad (13.10.3)$$

Khi  $\beta$  lớn đồng nghĩa với việc lấy trung bình trong một khoảng rộng, trong khi đó nếu  $\beta$  nhỏ phương pháp này sẽ không khác nhiều so với hạ gradient thông thường. Gradient mới này không còn có hướng đi dốc nhất trong từng trường hợp cụ thể nữa mà thay vào đó đi theo hướng trung bình có trọng số của các gradient trước đó. Điều này giúp chúng ta nhận thêm lợi ích của việc tính trung bình theo batch mà không cần tốn chi phí tính toán gradients trên batch. Chúng ta sẽ xem xét cụ thể hơn quy trình lấy trung bình ở những phần sau.

Các lập luận trên là cơ sở để hình thành các phương pháp *tăng tốc* gradient, chẳng hạn như gradient với động lượng. Một lợi ích phụ là chúng hiệu quả hơn rất nhiều trong các trường hợp bài toán tối ưu có điều kiện xấu (ví dụ: khi một vài hướng có tiến trình tối ưu chậm hơn nhiều so với các hướng khác, giống như ở trong một hẻm núi hẹp). Hơn nữa, cách này cho phép lấy trung bình

các gradient liền kề để đạt được hướng đi xuống ổn định hơn. Thật vậy, việc tăng tốc ngay cả đối với bài toán lồi không nhiễu là một trong những nguyên nhân chính lý giải vì sao động lượng hoạt động và có hiệu quả rất tốt.

Do tính hiệu quả của nó, động lượng là một chủ đề đã được nghiên cứu kỹ trong tối ưu hóa cho học sâu và hơn thế nữa. [Bài báo rất đẹp này](#)<sup>243</sup> của (Goh, 2017) cung cấp phân tích chuyên sâu và minh họa sinh động về phương pháp động lượng. Động lượng được đề xuất bởi (Polyak, 1964). (Nesterov, 2018) có một thảo luận chi tiết về lý thuyết động lượng trong ngữ cảnh tối ưu hóa lồi. Động lượng trong học sâu đã được biết đến từ lâu vì lợi ích mà nó mang lại. Tham khảo (Sutskever et al., 2013) để biết thêm chi tiết.

## Bài toán với Điều kiện Xấu

Để hiểu hơn về các tính chất hình học của phương pháp động lượng, hãy ôn lại thuật toán hạ gradient sử dụng hàm mục tiêu khó chịu hơn. Trong [Section 13.7](#) ta sử dụng hàm mục tiêu dạng elip  $f(\mathbf{x}) = x_1^2 + 2x_2^2$ . Ta sẽ sửa hàm này một chút để kéo dãn theo hướng  $x_1$  như sau:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \quad (13.10.4)$$

Cũng như trước,  $f$  đạt cực tiểu tại điểm  $(0, 0)$ . Hàm này *rất phẳng* theo hướng  $x_1$ . Hãy xem điều gì sẽ xảy ra khi thực hiện hạ gradient tương tự như trước trên hàm mới định nghĩa. Ta đặt tốc độ học bằng 0.4.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

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))
```

Có thể thấy gradient theo hướng  $x_2$  có giá trị *lớn hơn nhiều* và thay đổi nhanh hơn nhiều so với gradient theo hướng ngang  $x_1$ . Vì thế, chúng ta bị mắc kẹt giữa hai lựa chọn không mong muốn: Nếu chọn tốc độ học nhỏ, các nghiệm sẽ không phân kỳ theo hướng  $x_2$ , nhưng tốc độ hội tụ sẽ chậm theo hướng  $x_1$ . Ngược lại, với tốc độ học lớn mô hình sẽ hội tụ nhanh theo hướng  $x_1$  nhưng phân kỳ theo hướng  $x_2$ . Ví dụ dưới đây minh họa kết quả khi tăng nhẹ tốc độ học từ 0.4 lên 0.6. Sự hội tụ theo hướng  $x_1$  được cải thiện nhưng kết quả cuối cùng tệ hơn rất nhiều.

```
eta = 0.6
d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

<sup>243</sup> <https://distill.pub/2017/momentum/>

## Phương pháp Động lượng

Phương pháp động lượng cho phép chúng ta giải quyết vấn đề với hạ gradient mô tả ở trên. Nhìn vào các vết tối ưu trên, có thể thấy sẽ tốt hơn nếu lấy trung bình gradient trong quá khứ. Ở chiều  $x_1$  các gradient là cùng hướng, cách làm này sẽ đơn thuần lấy tổng độ lớn, từ đó tăng khoảng cách di chuyển ở từng bước. Ngược lại, gradient dao động mạnh theo hướng  $x_2$ , do đó kết hợp các gradient sẽ làm giảm kích thước bước do dao động triệt tiêu lẫn nhau. Sử dụng  $\mathbf{v}_t$  thay vì gradient  $\mathbf{g}_t$ , ta có các phương trình cập nhật sau:

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta\mathbf{v}_{t-1} + \mathbf{g}_{t,t-1}, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta_t \mathbf{v}_t.\end{aligned}\tag{13.10.5}$$

Với  $\beta = 0$ , phương pháp này tương đương với thuật toán hạ gradient thông thường. Trước khi đi sâu hơn vào các tính chất toán học, hãy xem thuật toán này hoạt động như thế nào.

```
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))
```

Có thể thấy, ngay cả với tốc độ học như trước, phương pháp động lượng vẫn hội tụ tốt. Giờ hãy xem điều gì xảy ra khi giảm tham số động lượng. Giảm một nửa động lượng  $\beta = 0.25$  dẫn đến một quỹ đạo chưa thật sự hội tụ. Tuy nhiên, kết quả đó vẫn tốt hơn rất nhiều so với khi không sử dụng động lượng (nghiệm phân kỳ).

```
eta, beta = 0.6, 0.25
d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

Ta cũng có thể kết hợp động lượng với SGD và đặc biệt là SGD theo minibatch. Thay đổi duy nhất trong trường hợp đó là các gradient  $\mathbf{g}_{t,t-1}$  được thay bằng  $\mathbf{g}_t$ . Cuối cùng, để thuận tiện ta khởi tạo  $\mathbf{v}_0 = 0$  tại thời điểm  $t = 0$ . Hãy xem phép trung bình rò rỉ thực sự làm gì khi cập nhật.

## Trọng số mẫu hiệu dụng

Hãy nhớ lại rằng  $\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau, t-\tau-1}$ . Tại giới hạn, tổng các số hạng là  $\sum_{\tau=0}^{\infty} \beta^\tau = \frac{1}{1-\beta}$ . Nói cách khác, thay vì kích thước bước  $\eta$  trong GD hoặc SGD, ta thực hiện bước dài hơn  $\frac{\eta}{1-\beta}$ , đồng thời hướng giảm gradient nhiều khả năng cũng tốt hơn. Đây là hai lợi ích trong một. Để minh họa ảnh hưởng của trọng số với các giá trị  $\beta$  khác nhau, hãy xem minh họa dưới đây.

```
d2l.set_figsize()
betas = [0.95, 0.9, 0.6, 0]
for beta in betas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, beta ** x, label=f'beta = {beta:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```

### 13.10.2 Thực nghiệm

Hãy xem động lượng hoạt động như thế nào trong thực tế, khi được sử dụng cùng một bộ tối ưu. Để làm điều này, ta cần các đoạn mã dễ mở rộng hơn.

#### Lập trình từ đầu

So với SGD hoặc SGD theo minibatch, phương pháp động lượng cần duy trì các biến phụ trợ, chính là vận tốc. Nó có kích thước giống gradient (và các biến khác trong bài toán tối ưu). Trong đoạn mã bên dưới, ta gọi các biến vận tốc này là states.

```
def init_momentum_states(feature_dim):
    v_w = np.zeros((feature_dim, 1))
    v_b = np.zeros(1)
    return (v_w, v_b)

def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        v[:] = hyperparams['momentum'] * v + p.grad
        p[:] -= hyperparams['lr'] * v
```

Hãy xem đoạn mã hoạt động như thế nào 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)
```

Khi tăng siêu tham số động lượng momentum lên 0.9, kích thước mẫu hiệu dụng sẽ tăng lên đáng kể thành  $\frac{1}{1-0.9} = 10$ . Ta giảm tốc độ học xuống còn 0.01 để kiểm soát vấn đề.

```
train_momentum(0.01, 0.9)
```

Tiếp tục giảm tốc độ học sẽ giải quyết bất kỳ vấn đề tối ưu không trơn tru nào. Giảm còn 0.005 đem lại các đặc tính hội tụ tốt.

```
train_momentum(0.005, 0.9)
```

#### Lập trình Súc tích

Rất đơn giản nếu sử dụng Gluon vì bộ tối ưu sgd tiêu chuẩn đã tích hợp sẵn phương pháp động lượng. Với cùng các tham số, ta có quỹ đạo rất giống khi lập trình từ đầu.

```
d2l.train_concise_ch11('sgd', {'learning_rate': 0.005, 'momentum': 0.9},
                        data_iter)
```

### 13.10.3 Phân tích Lý thuyết

Cho đến nay, ví dụ hai chiều  $f(x) = 0.1x_1^2 + 2x_2^2$  đường như không thực tế cho lắm. Thực tế, hàm này khá tiêu biểu cho các dạng bài toán có thể gấp phai, ít nhất trong trường hợp cực tiểu hóa các hàm mục tiêu lồi bậc hai.

#### Hàm lồi bậc Hai

Xét hàm số

$$h(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{x}^\top \mathbf{c} + b. \quad (13.10.6)$$

Đây là một hàm bậc hai tổng quát. Với ma trận xác định dương  $\mathbf{Q} \succ 0$ , tức ma trận có trị riêng dương, hàm có nghiệm cực tiểu tại  $\mathbf{x}^* = -\mathbf{Q}^{-1}\mathbf{c}$  với giá trị cực tiểu  $b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ . Do đó ta có thể viết lại  $h$  như sau

$$h(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})^\top \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c}) + b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}. \quad (13.10.7)$$

Gradient được tính bởi  $\partial_{\mathbf{x}} f(\mathbf{x}) = \mathbf{Q}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$ . Nghĩa là bằng khoảng cách giữa  $\mathbf{x}$  và nghiệm cực tiểu nhân với  $\mathbf{Q}$ . Do đó, động lượng cũng là tổ hợp tuyến tính của  $\mathbf{Q}(\mathbf{x}_t - \mathbf{Q}^{-1}\mathbf{c})$ .

Vì  $\mathbf{Q}$  là xác định dương nên nó có thể được phân tích thành hệ riêng thông qua  $\mathbf{Q} = \mathbf{O}^\top \Lambda \mathbf{O}$ , với  $\mathbf{O}$  là ma trận trực giao (xoay vòng) và  $\Lambda$  là ma trận đường chéo của các trị riêng dương. Điều này cho phép ta đổi biến  $\mathbf{x}$  thành  $\mathbf{z} := \mathbf{O}(\mathbf{x} - \mathbf{Q}^{-1}\mathbf{c})$  để có biểu thức đơn giản hơn nhiều:

$$h(\mathbf{z}) = \frac{1}{2} \mathbf{z}^\top \Lambda \mathbf{z} + b'. \quad (13.10.8)$$

Ở đây  $c' = b - \frac{1}{2}\mathbf{c}^\top \mathbf{Q}^{-1}\mathbf{c}$ . Vì  $\mathbf{O}$  chỉ là một ma trận trực giao nên điều này không làm nhiễu các gradient một cách có ý nghĩa. Biểu diễn theo  $\mathbf{z}$ , ta có hạ gradient

$$\mathbf{z}_t = \mathbf{z}_{t-1} - \Lambda \mathbf{z}_{t-1} = (\mathbf{I} - \Lambda) \mathbf{z}_{t-1}. \quad (13.10.9)$$

Một điểm quan trọng trong biểu thức này là hạ gradient *không trộn lẫn* các không gian riêng khác nhau. Nghĩa là, khi được biểu diễn dưới dạng hệ riêng của  $\mathbf{Q}$ , việc tối ưu được thực hiện theo từng trực tọa độ. Điều này cũng đúng với phương pháp động lượng.

$$\begin{aligned} \mathbf{v}_t &= \beta \mathbf{v}_{t-1} + \Lambda \mathbf{z}_{t-1} \\ \mathbf{z}_t &= \mathbf{z}_{t-1} - \eta (\beta \mathbf{v}_{t-1} + \Lambda \mathbf{z}_{t-1}) \\ &= (\mathbf{I} - \eta \Lambda) \mathbf{z}_{t-1} - \eta \beta \mathbf{v}_{t-1}. \end{aligned} \quad (13.10.10)$$

Khi thực hiện điều này, ta đã chứng minh định lý sau: Hạ Gradient có và không có động lượng cho hàm lồi bậc hai có thể được phân tích thành bài toán tối ưu theo hướng các vector riêng của ma trận bậc hai theo từng trực tọa độ.

## Hàm vô hướng

Với kết quả trên hãy xem điều gì xảy ra khi cực tiểu hóa hàm  $f(x) = \frac{\lambda}{2}x^2$ . Ta có hạ gradient

$$x_{t+1} = x_t - \eta\lambda x_t = (1 - \eta\lambda)x_t. \quad (13.10.11)$$

Với  $|1 - \eta\lambda| < 1$ , sau  $t$  bước ta có  $x_t = (1 - \eta\lambda)^t x_0$ , do đó tốc độ hội tụ sẽ theo hàm mũ. Tốc độ hội tụ sẽ tăng khi tăng tốc độ học  $\eta$  cho đến khi  $\eta\lambda = 1$ . Khi  $\eta\lambda > 2$ , bài toán tối ưu sẽ phân kỳ.

```
lambdas = [0.1, 1, 10, 19]
eta = 0.1
d2l.set_figsize((6, 4))
for lam in lambdas:
    t = np.arange(20).asnumpy()
    d2l.plt.plot(t, (1 - eta * lam) ** t, label=f'lambda = {lam:.2f}')
d2l.plt.xlabel('time')
d2l.plt.legend();
```

Để phân tích tính hội tụ khi sử dụng động lượng, ta viết lại các phương trình cập nhật theo hai số vô hướng:  $x$  và động lượng  $v$ . Ta có:

$$\begin{bmatrix} v_{t+1} \\ x_{t+1} \end{bmatrix} = \begin{bmatrix} \beta & \lambda \\ -\eta\beta & (1 - \eta\lambda) \end{bmatrix} \begin{bmatrix} v_t \\ x_t \end{bmatrix} = \mathbf{R}(\beta, \eta, \lambda) \begin{bmatrix} v_t \\ x_t \end{bmatrix}. \quad (13.10.12)$$

Ta kí hiệu  $\mathbf{R}$  là ma trận chi phối hội tụ, kích thước  $2 \times 2$ . Sau  $t$  bước thì giá trị ban đầu  $[v_0, x_0]$  sẽ là  $\mathbf{R}(\beta, \eta, \lambda)^t [v_0, x_0]$ . Do đó, các trị riêng của  $\mathbf{R}$  sẽ quyết định tốc độ hội tụ. Độc giả có thể xem hình ảnh động tại [bài viết của Distill<sup>244</sup>](#) của (Goh, 2017) và đọc thêm (Flammarion & Bach, 2015) để biết phân tích chi tiết. Có thể chỉ ra rằng phương pháp động lượng hội tụ với  $0 < \eta\lambda < 2 + 2\beta$ , có khoảng tham số khả thi lớn hơn khoảng  $0 < \eta\lambda < 2$  của hạ gradient. Điều này cũng gợi ý rằng nhìn chung ta mong muốn  $\beta$  có giá trị lớn. Chi tiết kỹ thuật đòi hỏi nền tảng kiến thức sâu hơn, bạn đọc quan tâm có thể tham khảo các bài báo gốc.

### 13.10.4 Tóm tắt

- Phương pháp động lượng thay thế gradient bằng trung bình rò rỉ của các gradient trong quá khứ, giúp tăng tốc độ hội tụ đáng kể.
- Phương pháp này có thể sử dụng cho cả hạ gradient không nhiễu và hạ gradient ngẫu nhiên (có nhiễu).
- Phương pháp động lượng giúp tránh việc tối ưu bị ngưng trệ, điều nhiều khả năng xảy ra đối với hạ gradient ngẫu nhiên.
- Số lượng gradient hiệu dụng là  $\frac{1}{1-\beta}$ , được tính bằng giới hạn của tổng cấp số nhân.
- Trong trường hợp các bài toán lồi bậc hai, hạ gradient (có và không có động lượng) có thể được phân tích chi tiết một cách tường minh.
- Việc lập trình khá đơn giản nhưng cần lưu trữ thêm một vector trạng thái (động lượng  $v$ ).

<sup>244</sup> <https://distill.pub/2017/momentum/>

### 13.10.5 Bài tập

- Quan sát và phân tích kết quả khi sử dụng các tổ hợp động lượng và tốc độ học khác nhau.
- Hãy thử dùng hạ gradient có động lượng cho bài toán bậc hai có nhiều trị riêng, ví dụ:  $f(x) = \frac{1}{2} \sum_i \lambda_i x_i^2$ , e.g.,  $\lambda_i = 2^{-i}$ . Vẽ đồ thị biểu diễn sự giảm của  $x$  khi khởi tạo  $x_i = 1$ .
- Tính giá trị và nghiệm cực tiểu của  $h(\mathbf{x}) = \frac{1}{2} \mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{x}^\top \mathbf{c} + b$ .
- Điều gì thay đổi khi ta thực hiện SGD và SGD theo minibatch có động lượng? Thử nghiệm với các tham số.

### 13.10.6 Thảo luận

- Tiếng Anh - MXNet<sup>245</sup>
- Tiếng Việt<sup>246</sup>

### 13.10.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Thanh Hòa
- Nguyễn Văn Quang
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Hồng Vinh
- Nguyễn Văn Cường

## 13.11 Adagrad

Để khởi động, hãy cùng xem xét các bài toán với những đặc trưng xuất hiện không thường xuyên.

<sup>245</sup> <https://discuss.d2l.ai/t/354>

<sup>246</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 13.11.1 Đặc trưng Thưa và Tốc độ Học

Hãy tưởng tượng ta đang huấn luyện một mô hình ngôn ngữ. Để đạt độ chính xác cao ta thường muốn giảm dần tốc độ học trong quá trình huấn luyện, thường là với tỉ lệ  $\mathcal{O}(t^{-\frac{1}{2}})$  hoặc chậm hơn. Xét một mô hình huấn luyện dựa trên những đặc trưng thưa, tức là các đặc trưng hiếm khi xuất hiện. Đây là điều thường gặp trong ngôn ngữ tự nhiên, ví dụ từ *preconditioning* hiếm gặp hơn nhiều so với *learning*. Tuy nhiên, đây cũng là vấn đề thường gặp trong nhiều mảng khác như quảng cáo điện toán (*computational advertising*) và lọc cộng tác (*collaborative filtering*). Xét cho cùng, có rất nhiều thứ mà chỉ có một nhóm nhỏ người chú ý đến.

Các tham số liên quan đến các đặc trưng thưa chỉ được cập nhật khi những đặc trưng này xuất hiện. Đối với tốc độ học giảm dần, ta có thể gặp phải trường hợp các tham số của những đặc trưng phổ biến hội tụ khá nhanh đến giá trị tối ưu, trong khi đối với các đặc trưng thưa, ta không có đủ số lượng dữ liệu thích đáng để xác định giá trị tối ưu của chúng. Nói một cách khác, tốc độ học hoặc là giảm quá chậm đối với các đặc trưng phổ biến hoặc là quá nhanh đối với các đặc trưng hiếm.

Một mẹo để khắc phục vấn đề này là đếm số lần ta gặp một đặc trưng nhất định và sử dụng nó để điều chỉnh tốc độ học. Tức là thay vì chọn tốc độ học theo công thức  $\eta = \frac{\eta_0}{\sqrt{t+c}}$  ta có thể sử dụng  $\eta_i = \frac{\eta_0}{\sqrt{s(i,t)+c}}$ . Trong đó  $s(i,t)$  là số giá trị khác không của đặc trưng  $i$  ta quan sát được đến thời điểm  $t$ . Công thức này khá dễ để lập trình và không tốn thêm bao nhiêu công sức. Tuy nhiên, cách này thất bại trong trường hợp khi đặc trưng không hẳn là thưa, chỉ là có gradient nhỏ và hiếm khi đạt giá trị lớn. Xét cho cùng, ta khó có thể phân định rõ ràng khi nào thì một đặc trưng là đã được quan sát hay chưa.

Adagrad được đề xuất trong (Duchi et al., 2011) đã giải quyết vấn đề này bằng cách thay đổi bộ đếm thô  $s(i,t)$  bởi tổng bình phương của tất cả các gradient được quan sát trước đó. Cụ thể, nó sử dụng  $s(i,t+1) = s(i,t) + (\partial_i f(\mathbf{x}))^2$  làm công cụ để điều chỉnh tốc độ học. Việc này đem lại hai lợi ích: trước tiên ta không cần phải quyết định khi nào thì gradient được coi là đủ lớn. Thứ hai, nó tự động thay đổi giá trị tuỳ theo độ lớn của gradient. Các tọa độ thường xuyên có gradient lớn bị giảm đi đáng kể, trong khi các tọa độ khác với gradient nhỏ được xử lý nhẹ nhàng hơn nhiều. Phương pháp này trong thực tế đưa ra một quy trình tối ưu hoạt động rất hiệu quả trong quảng cáo điện toán và các bài toán liên quan. Tuy nhiên, Adagrad vẫn còn ẩn chứa một vài lợi ích khác mà ta sẽ hiểu rõ nhất khi xét đến bối cảnh tiền điều kiện.

### 13.11.2 Tiền điều kiện

Các bài toán tối ưu lồi rất phù hợp để phân tích đặc tính của các thuật toán. Suy cho cùng, với đa số các bài toán không lồi ta khó có thể tìm được các chứng minh lý thuyết vững chắc. Tuy nhiên, *trực giác* và *ý nghĩa hàm chứa* suy ra từ các bài toán tối ưu lồi vẫn có thể được áp dụng. Xét bài toán cực tiểu hóa  $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + b$ .

Như ta đã thấy ở Section 13.10, ta có thể biến đổi bài toán sử dụng phép phân tích trị riêng  $\mathbf{Q} = \mathbf{U}^\top \Lambda \mathbf{U}$  nhằm biến đổi nó về dạng đơn giản hơn mà ta có thể xử lý trên từng tọa độ một:

$$f(\mathbf{x}) = \bar{f}(\bar{\mathbf{x}}) = \frac{1}{2}\bar{\mathbf{x}}^\top \Lambda \bar{\mathbf{x}} + \bar{\mathbf{c}}^\top \bar{\mathbf{x}} + b. \quad (13.11.1)$$

Ở đây ta sử dụng  $\mathbf{x} = \mathbf{U}\bar{\mathbf{x}}$  và theo đó  $\mathbf{c} = \mathbf{U}\bar{\mathbf{c}}$ . Bài toán sau khi được biến đổi có các nghiệm cực tiểu (*minimizer*)  $\bar{\mathbf{x}} = -\Lambda^{-1}\bar{\mathbf{c}}$  và giá trị nhỏ nhất  $-\frac{1}{2}\bar{\mathbf{c}}^\top \Lambda^{-1}\bar{\mathbf{c}} + b$ . Việc tính toán trở nên dễ dàng hơn nhiều do  $\Lambda$  là một ma trận đường chéo chứa các trị riêng của  $\mathbf{Q}$ .

Nếu ta làm nhiều  $\mathbf{c}$  một chút, ta sẽ mong rằng các nghiệm cực tiểu của  $f$  cũng chỉ thay đổi không đáng kể. Đáng tiếc thay, điều đó lại không xảy ra. Mặc dù thay đổi  $\mathbf{c}$  một chút dẫn đến  $\bar{\mathbf{c}}$  cũng thay

đổi một lượng tương ứng, các nghiệm cực tiểu của  $f$  (cũng như  $\bar{f}$ ) lại không như vậy. Mỗi khi các trị riêng  $\Lambda_i$  mang giá trị lớn, ta sẽ thấy  $\bar{x}_i$  và cực tiểu của  $f$  thay đổi khá nhò. Ngược lại, với  $\Lambda_i$  nhỏ, sự thay đổi  $\bar{x}_i$  có thể là đáng kể. Tỉ lệ giữa trị riêng lớn nhất và nhỏ nhất được gọi là hệ số điều kiện (*condition number*) của bài toán tối ưu.

$$\kappa = \frac{\Lambda_1}{\Lambda_d}. \quad (13.11.2)$$

Nếu hệ số điều kiện  $\kappa$  lớn, việc giải bài toán tối ưu một cách chính xác trở nên khá khó khăn. Ta cần đảm bảo việc lựa chọn một khoảng động lớn các giá trị phù hợp. Quá trình phân tích dẫn đến một câu hỏi hiển nhiên dù có phần ngây thơ rằng: chẳng phải ta có thể “sửa chữa” bài toán bằng cách biến đổi không gian sao cho tất cả các trị riêng đều có giá trị bằng 1. Điều này khá đơn giản trên lý thuyết: ta chỉ cần tính các trị riêng và các vector riêng của  $\mathbf{Q}$  nhằm biến đổi bài toán từ  $\mathbf{x}$  sang  $\mathbf{z} := \Lambda^{\frac{1}{2}} \mathbf{U} \mathbf{x}$ . Trong hệ toạ độ mới,  $\mathbf{x}^\top \mathbf{Q} \mathbf{x}$  có thể được đơn giản hóa thành  $\|\mathbf{z}\|^2$ . Nhưng có vẻ hướng giải quyết này không thực tế. Việc tính toán các trị riêng và các vector riêng thường tốn kém hơn *rất nhiều* so với việc tìm lời giải cho bài toán thực tế.

Trong khi việc tính toán chính xác các trị riêng có thể có chi phí cao, việc ước chừng và tính toán xấp xỉ chúng đã là tốt hơn nhiều so với không làm gì cả. Trong thực tế, ta có thể sử dụng các phần tử trên đường chéo của  $\mathbf{Q}$  và tái tỉ lệ chúng một cách tương ứng. Việc này có chi phí tính toán thấp hơn *nhiều* so với tính các trị riêng.

$$\tilde{\mathbf{Q}} = \text{diag}^{-\frac{1}{2}}(\mathbf{Q}) \mathbf{Q} \text{diag}^{-\frac{1}{2}}(\mathbf{Q}). \quad (13.11.3)$$

Trong trường hợp này ta có  $\tilde{\mathbf{Q}}_{ij} = \mathbf{Q}_{ij} / \sqrt{\mathbf{Q}_{ii} \mathbf{Q}_{jj}}$  và cụ thể  $\tilde{\mathbf{Q}}_{ii} = 1$  với mọi  $i$ . Trong đa số các trường hợp, cách làm này sẽ đơn giản hóa đáng kể hệ số điều kiện. Ví dụ đối với các trường hợp ta đã thảo luận ở phần trước, việc này sẽ triệt tiêu hoàn toàn vấn đề đang có do các bài toán đều có cấu trúc hình học với các cạnh song song trực toạ độ (*axis aligned*).

Đáng tiếc rằng ta phải tiếp tục đổi mới với một vấn đề khác: trong học sâu, ta thường không tính được ngay cả đạo hàm bậc hai của hàm mục tiêu. Đối với  $\mathbf{x} \in \mathbb{R}^d$ , đạo hàm bậc hai thậm chí với một minibatch có thể yêu cầu không gian và độ phức tạp lên đến  $\mathcal{O}(d^2)$  để tính toán, do đó khiến cho vấn đề không thể thực hiện được trong thực tế. Sự khéo léo của Adagrad nằm ở việc sử dụng một biến đại diện (*proxy*) để tính toán đường chéo của ma trận Hessian một cách hiệu quả và đơn giản—đó là độ lớn của chính gradient.

Để tìm hiểu tại sao cách này lại có hiệu quả, hãy cùng xét  $\bar{f}(\bar{\mathbf{x}})$ . Ta có:

$$\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}}) = \mathbf{\Lambda} \bar{\mathbf{x}} + \bar{\mathbf{c}} = \mathbf{\Lambda} (\bar{\mathbf{x}} - \bar{\mathbf{x}}_0), \quad (13.11.4)$$

trong đó  $\bar{\mathbf{x}}_0$  là nghiệm cực tiểu của  $\bar{f}$ . Do đó độ lớn của gradient phụ thuộc vào cả  $\mathbf{\Lambda}$  và khoảng cách đến điểm tối ưu. Nếu  $\bar{\mathbf{x}} - \bar{\mathbf{x}}_0$  không đổi thì đây chính là tất cả các giá trị ta cần tính. Suy cho cùng, trong trường hợp này độ lớn của gradient  $\partial_{\bar{\mathbf{x}}} \bar{f}(\bar{\mathbf{x}})$  là đủ. Do AdaGrad là một thuật toán hạ gradient ngẫu nhiên, ta sẽ thấy các gradient có phương sai khác không ngay cả tại điểm tối ưu. Chính vì thế ta có thể yên tâm sử dụng phương sai của các gradient như một biến đại diện để tính cho độ lớn của ma trận Hessian. Việc phân tích chi tiết nằm ngoài phạm vi của phần này (có thể lên đến nhiều trang). Độc giả có thể tham khảo (Duchi et al., 2011) để biết thêm chi tiết.

### 13.11.3 Thuật toán

Hãy cùng công thức hóa phần thảo luận ở trên. Ta sử dụng biến  $\mathbf{s}_t$  để tích luỹ phương sai của các gradient trong quá khứ như sau:

$$\begin{aligned}\mathbf{g}_t &= \partial_{\mathbf{w}} l(y_t, f(\mathbf{x}_t, \mathbf{w})), \\ \mathbf{s}_t &= \mathbf{s}_{t-1} + \mathbf{g}_t^2, \\ \mathbf{w}_t &= \mathbf{w}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \cdot \mathbf{g}_t.\end{aligned}\tag{13.11.5}$$

Ở đây các phép toán được thực hiện theo từng tọa độ. Nghĩa là,  $\mathbf{v}^2$  có các phần tử  $v_i^2$ . Tương tự,  $\frac{1}{\sqrt{v}}$  cũng có các phần tử  $\frac{1}{\sqrt{v_i}}$  và  $\mathbf{u} \cdot \mathbf{v}$  có các phần tử  $u_i v_i$ . Như phần trước  $\eta$  là tốc độ học và  $\epsilon$  là hằng số cộng thêm đảm bảo rằng ta không bị lỗi chia cho 0. Cuối cùng, ta khởi tạo  $\mathbf{s}_0 = \mathbf{0}$ .

Tương tự như trường hợp sử dụng động lượng, ta cần phải theo dõi các biến bổ trợ để mỗi tọa độ có một tốc độ học độc lập. Cách này không làm tăng chi phí của Adagrad so với SGD, lý do đơn giản là bởi chi phí chính yếu thường nằm ở bước tính  $l(y_t, f(\mathbf{x}_t, \mathbf{w}))$  và đạo hàm của nó.

Cần lưu ý, tổng bình phương các gradient trong  $\mathbf{s}_t$  có thể hiểu là về cơ bản  $\mathbf{s}_t$  tăng một cách tuyến tính (có phần chậm hơn so với tuyến tính trong thực tế, do gradient lúc ban đầu bị co lại). Điều này dẫn đến tốc độ học là  $\mathcal{O}(t^{-\frac{1}{2}})$ , mặc dù được điều chỉnh theo từng tọa độ một. Đối với các bài toán lồi, như vậy là hoàn toàn đủ. Tuy nhiên trong học sâu, có lẽ ta sẽ muốn giảm tốc độ học chậm hơn một chút. Việc này dẫn đến một số biến thể của Adagrad mà ta sẽ thảo luận trong các phần tới. Còn bây giờ hãy cùng xét cách thức hoạt động của Adagrad trong một bài toán lồi bậc hai. Ta vẫn giữ nguyên bài toán như cũ:

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2.\tag{13.11.6}$$

Ta sẽ lập trình Adagrad với tốc độ học giữ nguyên như phần trước, tức  $\eta = 0.4$ . Có thể thấy quy đạo của biến độc lập mượt hơn nhiều. Tuy nhiên, do ta tính tổng  $s_t$ , tốc độ học liên tục suy giảm khiến cho các biến độc lập không thay đổi nhiều ở các giai đoạn về sau của vòng lặp.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx
npx.set_np()

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
    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))
```

Nếu tăng tốc độ học lên 2, ta có thể thấy quá trình học tốt hơn đáng kể. Điều này chứng tỏ rằng tốc độ học giảm khá mạnh, ngay cả trong trường hợp không có nhiễu và ta cần phải 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))
```

### 13.11.4 Lập trình từ đầu

Giống như phương pháp động lượng, Adagrad cần duy trì một biến trạng thái có cùng kích thước với các tham số.

```
def init_adagrad_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.zeros(1)
    return (s_w, s_b)

def adagrad(params, states, hyperparams):
    eps = 1e-6
    for p, s in zip(params, states):
        s[:] += np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)
```

Ta sử dụng tốc độ học lớn hơn so với thí nghiệm ở Section 13.9 để huấn luyện 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);
```

### 13.11.5 Lập trình Súc tích

Sử dụng đối tượng Trainer trong thuật toán adagrad, ta có thể gọi thuật toán Adagrad trong Gluon.

```
d2l.train_concise_ch11('adagrad', {'learning_rate': 0.1}, data_iter)
```

### 13.11.6 Tóm tắt

- Adagrad liên tục giảm giá trị của tốc độ học theo từng tọa độ.
- Thuật toán sử dụng độ lớn của gradient như một phương thức để điều chỉnh tiến độ học - các tọa độ với gradient lớn được cân bằng bởi tốc độ học nhỏ.
- Tính đạo hàm bậc hai một cách chính xác thường không khả thi trong các bài toán học sâu do hạn chế về bộ nhớ và khả năng tính toán. Do đó, gradient có thể trở thành một biến đại diện hữu ích.
- Nếu bài toán tối ưu có cấu trúc không được đồng đều, Adagrad có thể làm giảm bớt sự biến dạng đó.
- Adagrad thường khá hiệu quả đối với các đặc trưng thừa, trong đó tốc độ học cần giảm chậm hơn cho các tham số hiếm khi xảy ra.

- Trong các bài toán học sâu, Adagrad đôi khi làm giảm tốc độ học quá mạnh. Ta sẽ thảo luận các chiến lược nhằm giảm bớt vấn đề này trong ngữ cảnh của Section 13.14.

### 13.11.7 Bài tập

- Chứng minh rằng một ma trận trực giao  $\mathbf{U}$  và một vector  $\mathbf{c}$  thoả mãn điều kiện:  $\|\mathbf{c} - \delta\|_2 = \|\mathbf{U}\mathbf{c} - \mathbf{U}\delta\|_2$ . Tại sao biểu thức trên lại biểu thị rằng độ nhiễu loạn không thay đổi khi biến đổi trực giao các biến?
- Thử áp dụng Adagrad đối với  $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$  và đối với hàm mục tiêu được quay 45 độ, tức là  $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ . Adagrad có hoạt động khác đi hay không?
- Chứng minh Định lý Gershgorin<sup>247</sup>, định lý phát biểu rằng với các trị riêng  $\lambda_i$  của ma trận  $\mathbf{M}$ , tồn tại  $j$  thoả mãn  $|\lambda_i - \mathbf{M}_{jj}| \leq \sum_{k \neq j} |\mathbf{M}_{jk}|$ .
- Từ định lý Gershgorin, ta có thể chỉ ra điều gì về các trị riêng của ma trận đường chéo tiền điều kiện (*diagonally preconditioned matrix*)  $\text{diag}^{-\frac{1}{2}}(\mathbf{M})\mathbf{M}\text{diag}^{-\frac{1}{2}}(\mathbf{M})$ ?
- Hãy thử áp dụng Adagrad cho một mạng thực sự sâu như Section 8.6 khi sử dụng Fashion MNIST.
- Bạn sẽ thay đổi Adagrad như thế nào để tốc độ học không suy giảm quá mạnh?

### 13.11.8 Thảo luận

- Tiếng Anh - MXNet<sup>248</sup>
- Tiếng Việt<sup>249</sup>

### 13.11.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Nguyễn Văn Quang
- Phạm Hồng Vinh

<sup>247</sup> [https://en.wikipedia.org/wiki/Gershgorin\\_circle\\_theorem](https://en.wikipedia.org/wiki/Gershgorin_circle_theorem)

<sup>248</sup> <https://discuss.d2l.ai/t/355>

<sup>249</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 13.12 RMSProp

Một trong những vấn đề then chốt trong Section 13.11 là tốc độ học thực tế được giảm theo một thời điểm được định nghĩa sẵn  $\mathcal{O}(t^{-\frac{1}{2}})$ . Nhìn chung, cách này thích hợp với các bài toán lồi nhưng có thể không phải giải pháp lý tưởng cho những bài toán không lồi, chẳng hạn những bài toán gặp phải trong học sâu. Tuy vậy, khả năng thích ứng theo tọa độ của Adagrad là rất tuyệt vời cho một bộ tiền điều kiện (*preconditioner*).

(Tieleman & Hinton, 2012) đề xuất thuật toán RMSProp như một bản vá đơn giản để tách rời tốc độ định thời ra khỏi tốc độ học thay đổi theo tọa độ (*coordinate-adaptive*). Vấn đề ở đây là Adagrad cộng dồn tổng bình phương của gradient  $\mathbf{g}_t$  vào vector trạng thái  $\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2$ . Kết quả là, do không có phép chuẩn hóa,  $\mathbf{s}_t$  vẫn tiếp tục tăng tuyến tính không ngừng trong quá trình hội tụ của thuật toán.

Vấn đề này có thể được giải quyết bằng cách sử dụng  $\mathbf{s}_t/t$ . Với phân phối  $\mathbf{g}_t$  hợp lý, thuật toán sẽ hội tụ. Đáng tiếc là có thể mất rất nhiều thời gian cho đến khi các tính chất tại giới hạn bắt đầu có ảnh hưởng, bởi thuật toán này ghi nhớ toàn bộ quỹ đạo của các giá trị. Một cách khác là sử dụng trung bình ròng rã tương tự như trong phương pháp động lượng, tức là  $\mathbf{s}_t \leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2$  cho các tham số  $\gamma > 0$ . Giữ nguyên tất cả các phần khác và ta có thuật toán RMSProp.

### 13.12.1 Thuật toán

Chúng ta hãy viết các phương trình ra một cách chi tiết.

$$\begin{aligned}\mathbf{s}_t &\leftarrow \gamma \mathbf{s}_{t-1} + (1 - \gamma) \mathbf{g}_t^2, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t.\end{aligned}\tag{13.12.1}$$

Hằng số  $\epsilon > 0$  thường được đặt bằng  $10^{-6}$  để đảm bảo rằng chúng ta sẽ không gặp vấn đề khi chia cho 0 hoặc kích thước bước quá lớn. Với khai triển này, bây giờ chúng ta có thể tự do kiểm soát tốc độ học  $\eta$  độc lập với phép biến đổi tỉ lệ được áp dụng cho từng tọa độ. Về mặt trung bình ròng rã, chúng ta có thể áp dụng các lập luận tương tự như trước trong phương pháp động lượng. Khai triển định nghĩa  $\mathbf{s}_t$  ta có

$$\begin{aligned}\mathbf{s}_t &= (1 - \gamma) \mathbf{g}_t^2 + \gamma \mathbf{s}_{t-1} \\ &= (1 - \gamma) (\mathbf{g}_t^2 + \gamma \mathbf{g}_{t-1}^2 + \gamma^2 \mathbf{g}_{t-2}^2 + \dots).\end{aligned}\tag{13.12.2}$$

Tương tự như Section 13.10, ta có  $1 + \gamma + \gamma^2 + \dots = \frac{1}{1-\gamma}$ . Do đó, tổng trọng số được chuẩn hóa bằng 1 và chu kỳ bán rã của một quan sát là  $\gamma^{-1}$ . Hãy cùng minh họa trực quan các trọng số này trong vòng 40 bước thời gian trước đó với các giá trị  $\gamma$  khác nhau.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import np, npx

npx.set_np()

d2l.set_figsize()
gammas = [0.95, 0.9, 0.8, 0.7]
```

(continues on next page)

```

for gamma in gammas:
    x = np.arange(40).asnumpy()
    d2l.plt.plot(x, (1-gamma) * gamma ** x, label=f'gamma = {gamma:.2f}')
d2l.plt.xlabel('time');

```

### 13.12.2 Lập trình 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 trong Section 13.11, khi chúng ta sử dụng Adagrad với tốc độ học bằng 0.4, các biến di chuyển rất chậm trong các giai đoạn sau của thuật toán do tốc độ học giảm quá nhanh. Do  $\eta$  được kiểm soát riêng biệt, nên 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))

```

Tiếp theo, chúng ta hãy lập trình thuật toán RMSProp để sử dụng trong một mạng nơ-ron sâu. Cách lập trình không quá phức tạp.

```

def init_rmsprop_states(feature_dim):
    s_w = np.zeros((feature_dim, 1))
    s_b = np.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):
        s[:] = gamma * s + (1 - gamma) * np.square(p.grad)
        p[:] -= hyperparams['lr'] * p.grad / np.sqrt(s + eps)

```

Chúng ta khởi tạo tốc độ học ban đầu bằng 0.01 và trọng số  $\gamma$  bằng 0.9. Nghĩa là,  $\mathbf{s}$  là tổng trung bình của  $1/(1 - \gamma) = 10$  quan sát bình phương gradient trong quá khứ.

```

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);

```

### 13.12.3 Lập trình Súc tích

Do RMSProp là thuật toán khá phổ biến, nó cũng được tích hợp sẵn trong thực thể Trainer. Những gì ta cần phải làm là khởi tạo thuật toán có tên là rmsprop, với  $\gamma$  được gán cho tham số gamma1.

```
d2l.train_concise_ch11('rmsprop', {'learning_rate': 0.01, 'gamma1': 0.9},  
    data_iter)
```

### 13.12.4 Tóm tắt

- Thuật toán RMSProp rất giống với Adagrad ở chỗ cả hai đều sử dụng bình phương của gradient để thay đổi tỉ lệ hệ số.
- RMSProp có điểm chung với phương pháp động lượng là chúng đều sử dụng trung bình rò rỉ. Tuy nhiên, RMSProp sử dụng kỹ thuật này để điều chỉnh tiền điều kiện theo hệ số.
- Trong thực tế, tốc độ học cần được định thời bởi người lập trình.
- Hệ số  $\gamma$  xác định độ dài thông tin quá khứ được sử dụng khi điều chỉnh tỉ lệ theo từng tọa độ.

### 13.12.5 Bài tập

1. Điều gì sẽ xảy ra nếu ta đặt  $\gamma = 1$ ? Giải thích tại sao?
2. Biến đổi bài toán tối ưu thành cực tiểu hóa  $f(\mathbf{x}) = 0.1(x_1 + x_2)^2 + 2(x_1 - x_2)^2$ . Sự hội tụ sẽ diễn ra như thế nào?
3. Hãy thử áp dụng RMSProp cho một bài toán học máy cụ thể, chẳng hạn như huấn luyện trên tập Fashion-MNIST. Hãy thí nghiệm với các tốc độ học khác nhau.
4. Bạn có muốn điều chỉnh  $\gamma$  khi việc tối ưu đang tiến triển không? Hãy cho biết độ nhạy của RMSProp với việc điều chỉnh này?

### 13.12.6 Thảo luận

- Tiếng Anh - MXNet<sup>250</sup>
- Tiếng Việt<sup>251</sup>

### 13.12.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức

<sup>250</sup> <https://discuss.d2l.ai/t/356>

<sup>251</sup> <https://forum.machinelearningcoban.com/c/d2l>

- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

## 13.13 Adadelta

Adadelta là một biến thể khác của AdaGrad. Điểm khác biệt chính là Adadelta giảm mức độ mà tốc độ học sẽ thay đổi với các tọa độ. Hơn nữa, Adadelta thường được biết đến là thuật toán không sử dụng tốc độ học vì nó dựa trên chính lượng thay đổi hiện tại để cản chỉnh lượng thay đổi trong tương lai. Thuật toán Adadelta được đề xuất trong (Zeiler, 2012). Nó cũng khá đơn giản nếu bạn đã biết các thuật toán được thảo luận trước đây.

### 13.13.1 Thuật toán

Nói ngắn gọn, Adadelta sử dụng hai biến trạng thái,  $\mathbf{s}_t$  để lưu trữ trung bình rò rỉ mô-men bậc hai của gradient và  $\Delta\mathbf{x}_t$  để lưu trữ trung bình rò rỉ mô-men bậc hai của lượng thay đổi của các tham số trong mô hình. Lưu ý rằng chúng ta sử dụng các ký hiệu và cách đặt tên nguyên bản của chính tác giả để nhất quán với các nghiên cứu khác và các cách lập trình, chứ không có lý do gì đặc biệt để ký hiệu cùng một tham số trong các thuật toán động lượng, Adagrad, RMSProp, và Adadelta bằng các kí hiệu La Mã khác nhau. Tham số suy giảm là  $\rho$ . Chúng ta có được các bước cập nhật rò rỉ như sau:

$$\begin{aligned}\mathbf{s}_t &= \rho\mathbf{s}_{t-1} + (1 - \rho)\mathbf{g}_t^2, \\ \mathbf{g}'_t &= \sqrt{\frac{\Delta\mathbf{x}_{t-1} + \epsilon}{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \\ \mathbf{x}_t &= \mathbf{x}_{t-1} - \mathbf{g}'_t, \\ \Delta\mathbf{x}_t &= \rho\Delta\mathbf{x}_{t-1} + (1 - \rho)\mathbf{x}_t^2.\end{aligned}\tag{13.13.1}$$

Điểm khác biệt so với trước là ta thực hiện các bước cập nhật với gradient  $\mathbf{g}'_t$  được tái tỉ lệ bằng cách lấy căn bậc hai thương của trung bình tốc độ thay đổi bình phương và trung bình mô-men bậc hai của gradient. Việc sử dụng  $\mathbf{g}'_t$  có mục đích đơn thuần là thuận tiện cho việc ký hiệu. Trong thực tế chúng ta có thể lập trình thuật toán này mà không cần phải sử dụng thêm bộ nhớ tạm cho  $\mathbf{g}'_t$ . Như trước đây  $\epsilon$  là tham số đảm bảo kết quả xấp xỉ có ý nghĩa, tức tránh trường hợp kích thước bước bằng 0 hoặc phương sai là vô hạn. Thông thường ta đặt  $\epsilon = 10^{-5}$ .

### 13.13.2 Lập trình

Thuật toán Adadelta cần duy trì hai biến trạng thái ứng với hai biến  $\mathbf{s}_t$  và  $\Delta\mathbf{x}_t$ . Do đó ta lập trình như sau.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

def init_adadelta_states(feature_dim):
```

(continues on next page)

```

s_w, s_b = np.zeros((feature_dim, 1)), np.zeros(1)
delta_w, delta_b = np.zeros((feature_dim, 1)), np.zeros(1)
return ((s_w, delta_w), (s_b, delta_b))

def adadelta(params, states, hyperparams):
    rho, eps = hyperparams['rho'], 1e-5
    for p, (s, delta) in zip(params, states):
        # In-place updates via [:]
        s[:] = rho * s + (1 - rho) * np.square(p.grad)
        g = (np.sqrt(delta + eps) / np.sqrt(s + eps)) * p.grad
        p[:] -= g
        delta[:] = rho * delta + (1 - rho) * g * g

```

Việc chọn  $\rho = 0.9$  ứng với chu kỳ bán rã bằng 10 cho mỗi lần cập nhật tham số, và thường thì nó là lựa chọn khá tốt. Thuật toán sẽ hoạt động như sau.

```

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);

```

Để lập trình súc tích, ta chỉ cần sử dụng thuật toán adadelta từ lớp Trainer. Nhờ vậy mà ta có thể chạy thuật toán chỉ với một dòng lệnh ngắn gọn.

```
d2l.train_concise_ch11('adadelta', {'rho': 0.9}, data_iter)
```

### 13.13.3 Tóm tắt

- Adadelta không sử dụng tham số tốc độ học. Thay vào đó, nó sử dụng tốc độ thay đổi của chính bản thân các tham số để điều chỉnh tốc độ học.
- Adadelta cần sử dụng hai biến trạng thái để lưu trữ các mô-men bậc hai của gradient và của lượng thay đổi trong các tham số.
- Adadelta sử dụng trung bình ròng để lưu ước lượng động của các giá trị thống kê cần thiết.

### 13.13.4 Bài tập

1. Điều gì xảy ra khi giá trị của  $\rho$  thay đổi?
2. Hãy lập trình thuật toán trên mà không cần dùng biến  $\mathbf{g}'_t$ . Giải thích tại sao đây có thể là một ý tưởng tốt?
3. Adadelta có thực sự không cần tốc độ học? Hãy chỉ ra các bài toán tối ưu mà Adadelta không thể giải.
4. Hãy so sánh Adadelta với Adagrad và RMSprop để thảo luận về sự hội tụ của từng thuật toán.

### 13.13.5 Thảo luận

- Tiếng Anh - MXNet<sup>252</sup>
- Tiếng Việt<sup>253</sup>

### 13.13.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Nguyễn Văn Quang
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Cảnh Thưởng

## 13.14 Adam

Từ các thảo luận dẫn trước, chúng ta đã làm quen với một số kỹ thuật để tối ưu hóa hiệu quả. Hãy cùng tóm tắt chi tiết những kỹ thuật này ở đây:

- Chúng ta thấy rằng SGD trong Section 13.8 hiệu quả hơn hạ gradient khi giải các bài toán tối ưu, ví dụ, nó chịu ít ảnh hưởng xấu gây ra bởi dữ liệu dư thừa.
- Chúng ta thấy rằng minibatch SGD trong Section 13.9 mang lại hiệu quả đáng kể nhờ việc vector hóa, tức xử lý nhiều mẫu quan sát hơn trong một minibatch. Đây là chìa khóa để xử lý dữ liệu song song trên nhiều GPU và nhiều máy tính một cách hiệu quả.
- Phương pháp động lượng trong Section 13.10 bổ sung cơ chế gộp các gradient quá khứ, giúp quá trình hội tụ diễn ra nhanh hơn.
- Adagrad trong Section 13.11 sử dụng phép biến đổi tỉ lệ theo từng tọa độ để tạo ra tiền điều kiện hiệu quả về mặt tính toán.
- RMSprop trong Section 13.12 tách rời phép biến đổi tỉ lệ theo từng tọa độ khỏi phép điều chỉnh tốc độ học.

Adam (Kingma & Ba, 2014) kết hợp tất cả các kỹ thuật trên thành một thuật toán học hiệu quả. Như kỳ vọng, đây là một trong những thuật toán tối ưu mạnh mẽ và hiệu quả được sử dụng phổ biến trong học sâu. Tuy nhiên nó cũng có một vài điểm yếu. Cụ thể, (Reddi et al., 2019) đã chỉ ra những trường hợp mà Adam có thể phản ứng kém do việc kiểm soát phương sai kém. Trong một nghiên cứu sau đó, (Zaheer et al., 2018) đã đề xuất Yogi, một bản vá nhanh cho Adam để giải quyết các vấn đề này. Chi tiết về bản vá này sẽ được đề cập sau, còn bây giờ hãy xem xét thuật toán Adam.

<sup>252</sup> <https://discuss.d2l.ai/t/357>

<sup>253</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 13.14.1 Thuật toán

Một trong những thành phần chính của Adam là các trung bình động trọng số mũ (hay còn được gọi là trung bình rò rỉ) để ước lượng cả động lượng và mô-men bậc hai của gradient. Cụ thể, nó sử dụng các biến trạng thái

$$\begin{aligned}\mathbf{v}_t &\leftarrow \beta_1 \mathbf{v}_{t-1} + (1 - \beta_1) \mathbf{g}_t, \\ \mathbf{s}_t &\leftarrow \beta_2 \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2.\end{aligned}\tag{13.14.1}$$

Ở đây  $\beta_1$  và  $\beta_2$  là các tham số trọng số không âm. Các lựa chọn phổ biến cho chúng là  $\beta_1 = 0.9$  và  $\beta_2 = 0.999$ . Điều này có nghĩa là ước lượng phương sai di chuyển chậm hơn nhiều so với số hạng động lượng. Lưu ý rằng nếu ta khởi tạo  $\mathbf{v}_0 = \mathbf{s}_0 = 0$ , thuật toán sẽ có độ chêch ban đầu đáng kể về các giá trị nhỏ hơn. Vấn đề này có thể được giải quyết bằng cách sử dụng  $\sum_{i=0}^t \beta^i = \frac{1-\beta^t}{1-\beta}$  để chuẩn hóa lại các số hạng. Tương tự, các biến trạng thái được chuẩn hóa như sau

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_1^t} \text{ and } \hat{\mathbf{s}}_t = \frac{\mathbf{s}_t}{1 - \beta_2^t}.\tag{13.14.2}$$

Với các ước lượng thích hợp, bây giờ chúng ta có thể viết ra các phương trình cập nhật. Đầu tiên, chúng ta điều chỉnh lại giá trị gradient, tương tự như ở RMSProp để có được

$$\mathbf{g}'_t = \frac{\eta \hat{\mathbf{v}}_t}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}.\tag{13.14.3}$$

Không giống như RMSProp, phương trình cập nhật sử dụng động lượng  $\hat{\mathbf{v}}_t$  thay vì gradient. Hơn nữa, có một sự khác biệt nhỏ ở đây: phép chuyển đổi được thực hiện bằng cách sử dụng  $\frac{1}{\sqrt{\hat{\mathbf{s}}_t} + \epsilon}$  thay vì  $\frac{1}{\sqrt{\mathbf{s}_t} + \epsilon}$ . Trong thực tế, cách đầu tiên hoạt động tốt hơn một chút, dẫn đến sự khác biệt này so với RMSProp. Thông thường, ta chọn  $\epsilon = 10^{-6}$  để cân bằng giữa tính ổn định số học và độ tin cậy.

Bây giờ chúng ta sẽ tổng hợp lại tất cả các điều trên để tính toán bước cập nhật. Có thể bạn sẽ thấy hơi tụt hứng một chút vì thực ra nó khá đơn giản

$$\mathbf{x}_t \leftarrow \mathbf{x}_{t-1} - \mathbf{g}'_t.\tag{13.14.4}$$

Khi xem xét thiết kế của Adam, ta thấy rõ nguồn cảm hứng của thuật toán. Động lượng và khoảng giá trị được thể hiện rõ ràng trong các biến trạng thái. Định nghĩa khá kì lạ của chúng đòi hỏi ta phải giảm độ chêch của các số hạng (có thể được thực hiện bằng cách tinh chỉnh một chút phép khởi tạo và điều kiện cập nhật). Thứ hai, việc kết hợp của cả hai số hạng trên khá đơn giản, dựa trên RMSProp. Cuối cùng, tốc độ học tương minh  $\eta$  cho phép ta kiểm soát độ dài bước cập nhật để giải quyết các vấn đề về hội tụ.

### 13.14.2 Lập trình

Lập trình Adam từ đầu không quá khó khăn. Để thuận tiện, chúng ta lưu trữ biến đếm bước thời gian  $t$  trong từ điển hyperparams. Ngoài điều đó ra, mọi thứ khác khá đơn giản.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()

def init_adam_states(feature_dim):
    v_w, v_b = np.zeros((feature_dim, 1)), np.zeros(1)
    s_w, s_b = np.zeros((feature_dim, 1)), np.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):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = beta2 * s + (1 - beta2) * np.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 / (np.sqrt(s_bias_corr) + eps)
    hyperparams['t'] += 1
```

Chúng ta đã sẵn sàng sử dụng Adam để huấn luyện mô hình. Chúng ta sử dụng tốc độ học  $\eta = 0.01$ .

```
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);
```

Cách lập trình súc tích hơn là gọi trực tiếp adam được cung cấp sẵn trong thư viện tối ưu trainer của Gluon. Do đó ta chỉ cần truyền các tham số cấu hình để lập trình trong Gluon.

```
d2l.train_concise_ch11('adam', {'learning_rate': 0.01}, data_iter)
```

### 13.14.3 Yogi

Một trong những vấn đề của Adam là nó có thể không hội tụ ngay cả trong các điều kiện lồi khi ước lượng mô-men bậc hai trong  $\mathbf{s}_t$  tăng đột biến. (Zaheer et al., 2018) đề xuất phiên bản cải thiện của bước cập nhật (và khởi tạo)  $\mathbf{s}_t$  để giải quyết vấn đề này. Để hiểu rõ hơn, chúng ta hãy viết lại bước cập nhật Adam như sau:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) (\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \quad (13.14.5)$$

Khi  $\mathbf{g}_t^2$  có phuơng sai lớn hay các cập nhật trở nên thưa,  $\mathbf{s}_t$  sẽ có thể nhanh chóng quên mất các giá trị quá khứ. Một cách giải quyết vấn đề trên đó là thay  $\mathbf{g}_t^2 - \mathbf{s}_{t-1}$  bằng  $\mathbf{g}_t^2 \odot \text{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1})$ . Bây giờ, độ lớn của cập nhật không còn phụ thuộc vào giá trị độ lệch. Từ đó ta có bước cập nhật Yogi sau:

$$\mathbf{s}_t \leftarrow \mathbf{s}_{t-1} + (1 - \beta_2) \mathbf{g}_t^2 \odot \text{sgn}(\mathbf{g}_t^2 - \mathbf{s}_{t-1}). \quad (13.14.6)$$

Hơn nữa, các tác giả khuyên nên khởi tạo động lượng trên một batch ban đầu có kích thước lớn hơn thay vì ước lượng ban đầu theo điểm. Chúng ta không đi sâu vào điểm này, vì quá trình hội tụ vẫn diễn ra khá tốt ngay cả khi không áp dụng chúng.

```

def yogi(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-3
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = s + (1 - beta2) * np.sign(
            np.square(p.grad) - s) * np.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 / (np.sqrt(s_bias_corr) + eps)
    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);

```

#### 13.14.4 Tóm tắt

- Adam kết hợp các kỹ thuật của nhiều thuật toán tối ưu thành một quy tắc cập nhật khá mạnh mẽ.
- Dựa trên RMSProp, Adam cũng sử dụng trung bình động trọng số mũ cho gradient ngẫu nhiên theo minibatch.
- Adam sử dụng phép hiệu chỉnh độ chêch (*bias correction*) để điều chỉnh cho trường hợp khởi động chậm khi ước lượng động lượng và mô-men bậc hai.
- Đối với gradient có phương sai đáng kể, chúng ta có thể gặp phải những vấn đề liên quan tới hội tụ. Những vấn đề này có thể được khắc phục bằng cách sử dụng các minibatch có kích thước lớn hơn hoặc bằng cách chuyển sang sử dụng ước lượng được cải tiến cho  $s_t$ . Yogi là một trong những giải pháp như vậy.

#### 13.14.5 Bài tập

1. Hãy điều chỉnh tốc độ học, quan sát và phân tích kết quả thực nghiệm.
2. Bạn có thể viết lại các phương trình cập nhật cho động lượng và mô-men bậc hai mà không cần thực hiện phép hiệu chỉnh độ chêch (*bias correction*) không?
3. Tại sao ta cần phải giảm tốc độ học  $\eta$  khi quá trình hội tụ diễn ra?
4. Hãy xây dựng một trường hợp mà thuật toán Adam phân kỳ nhưng Yogi lại hội tụ?

#### 13.14.6 Thảo luận

- Tiếng Anh - MXNet<sup>254</sup>
- Tiếng Việt<sup>255</sup>

<sup>254</sup> <https://discuss.d2l.ai/t/358>

<sup>255</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 13.14.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

## 13.15 Định thời Tốc độ Học

Đến nay, trong các *thuật toán* tối ưu ta tập trung chủ yếu ở cách cập nhật các vector trọng số thay vì *tốc độ* cập nhật các vector đó. Tuy nhiên, việc điều chỉnh tốc độ học thường cũng quan trọng như thuật toán. Có một vài điều ta cần quan tâm:

- Vấn đề rõ ràng nhất là *độ lớn* của tốc độ học có ảnh hưởng. Nếu tốc độ học quá lớn thì tối ưu phân kỳ, nếu quá nhỏ thì việc huấn luyện mất quá nhiều thời gian hoặc kết quả cuối cùng không đủ tốt. Ta biết rằng hệ số điều kiện (*condition number*) của bài toán rất quan trọng (xem Section 13.10 để biết thêm chi tiết). Theo trực giác, nó là tỷ lệ giữa mức độ thay đổi theo hướng ít nhạy cảm nhất và hướng nhạy cảm nhất.
- Thứ hai, tốc độ suy giảm cũng quan trọng không kém. Nếu duy trì tốc độ học lớn, thuật toán có thể chỉ dao động xung quanh điểm cực tiểu và do đó không đạt được nghiệm tối ưu. Section 13.7 đã thảo luận về vấn đề này và Section 13.8 đã phân tích các đàm bảo hội tụ. Nói ngắn gọn, ta muốn tốc độ hội tụ suy giảm ở mức chậm hơn cả  $\mathcal{O}(t^{-\frac{1}{2}})$ , một mức đã có thể coi là tốt cho các bài toán lồi.
- Một khía cạnh khác cũng quan trọng không kém là *khởi tạo*. Điều này liên quan đến cả cách thức các tham số được khởi tạo (xem lại Section 6.8) và cách chúng thay đổi lúc đầu. Có thể gọi đây là *khởi động* (*warmup*), tức ta bắt đầu tối ưu nhanh như thế nào. Bước tối ưu lớn ban đầu có thể không có lợi, đặc biệt là khi bộ tham số ban đầu là ngẫu nhiên. Các hướng cập nhật ban đầu cũng có thể không quan trọng.
- Cuối cùng, có một số biến thể tối ưu hóa thực hiện điều chỉnh tốc độ học theo chu kỳ. Điều này nằm ngoài phạm vi của chương hiện tại. Độc giả có thể đọc thêm (Izmailov et al., 2018), về làm thế nào để có các giải pháp tốt hơn bằng cách lấy trung bình trên toàn bộ *đường đi* của các tham số.

Vì việc quản lý tốc độ học khá vất vả, hầu hết các framework học sâu đều có các công cụ tự động cho việc này. Trong phần này ta sẽ xem xét ảnh hưởng của các định thời khác nhau lên độ chính xác, cũng như xem cách quản lý hiệu quả tốc độ học thông qua một *bộ định thời tốc độ học*.

### 13.15.1 Ví dụ Đơn giản

Hãy bắt đầu với một ví dụ đơn giản với chi phí tính toán ít nhưng đủ để minh họa một vài điểm cốt lõi. Ở đây ta sử dụng LeNet cải tiến (thay hàm kích hoạt sigmoid bằng relu và thay hàm gộp trung bình bằng hàm gộp cực đại) khi áp dụng trên tập dữ liệu Fashion-MNIST. Hơn nữa, để có hiệu năng tốt, ta hybrid hóa mạng. Vì hầu hết mã nguồn đều tương tự như trong Section 8, ta sẽ không thảo luận chi tiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, lr_scheduler, np, npx
from mxnet.gluon import nn
npx.set_np()

net = nn.HybridSequential()
net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=5, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120, activation='relu'),
        nn.Dense(84, activation='relu'),
        nn.Dense(10))
net.hybridize()
loss = gluon.loss.SoftmaxCrossEntropyLoss()
ctx = 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` that defined in the lenet
# section of chapter convolutional neural networks
def train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx):
    net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
    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):
            X, y = X.as_in_ctx(ctx), y.as_in_ctx(ctx)
            with autograd.record():
                y_hat = net(X)
                l = loss(y_hat, y)
            l.backward()
            trainer.step(X.shape[0])
            metric.add(l.sum(), 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))
    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
```

Ta hãy xem điều gì sẽ xảy ra khi gọi thuật toán với các thiết lập mặc định, như huấn luyện trong 30

epoch với tốc độ học 0.3. Có thể thấy độ chính xác trên tập huấn luyện vẫn tiếp tục tăng trong khi độ chính xác trên tập kiểm tra không tăng thêm khi đạt đến một giá trị nhất định. Khoảng cách giữa hai đường cong cho thấy độ khớp của thuật toán.

```
lr, num_epochs = 0.3, 30
net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

### 13.15.2 Bộ Định thời

Một cách để điều chỉnh tốc độ học là thiết lập giá trị của tốc độ học một cách tường minh ở mỗi bước. Có thể dùng phương thức `set_learning_rate` để làm điều này. Ta có thể giảm tốc độ học sau mỗi epoch (hay thậm chí sau mỗi minibatch) như một cách phản hồi khi quá trình tối ưu đang diễn ra.

```
trainer.set_learning_rate(0.1)
print('Learning rate is now %.2f' % trainer.learning_rate)
```

Tổng quát hơn, ta muốn định nghĩa một bộ định thời. Khi được gọi bằng cách truyền số bước cập nhật, bộ định thời trả về giá trị tương ứng của tốc độ học. Hãy định nghĩa một bộ định thời đơn giản biểu diễn tốc độ học  $\eta = \eta_0(t + 1)^{-\frac{1}{2}}$ .

```
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)
```

Tiếp theo hãy biểu thị sự thay đổi của tốc độ học sử dụng bộ định thời trên với một dải giá trị.

```
scheduler = SquareRootScheduler(lr=1.0)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```

Giờ hãy xem bộ định thời này hoạt động thế nào khi huấn luyện trên Fashion-MNIST. Ta đơn giản đưa bộ định thời vào giải thuật huấn luyện như một đối số bổ sung.

```
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)
```

Kết quả thu được tốt hơn một chút. Hai điểm nổi bật là đồ thị quá trình học mượt hơn và mô hình ít quá khớp hơn. Không may là chưa có lời giải thích đáng cho câu hỏi tại sao những chiến lược như vậy lại dẫn đến việc giảm quá khớp về mặt lý thuyết. Có một số ý kiến cho rằng kích thước bước nhỏ hơn sẽ đưa các tham số tới gần giá trị không hơn và do đó đơn giản hơn. Tuy nhiên, điều này không giải thích hoàn toàn hiện tượng trên bởi chúng ta không hề dùng giải thuật sớm mà chỉ giảm từ từ tốc độ học.

### 13.15.3 Những chính sách

Vì không đủ khả năng xem xét toàn bộ các loại định thời tốc độ học, chúng tôi cố gắng tóm tắt các chiến lược phổ biến dưới đây. Những lựa chọn phổ biến là định thời suy giảm theo đa thức và định thời hằng số theo từng khoảng. Xa hơn nữa, thực nghiệm cho thấy các bộ định thời theo hàm cô-sin làm việc tốt đối với một số bài toán. Sau cùng, với một số bài toán sẽ có lợi khi khởi động (*warmup*) bộ tối ưu trước khi sử dụng tốc độ học lớn.

#### Định thời Thừa số

Một giải pháp thay thế cho suy giảm đa thức đó là sử dụng thừa số nhân  $\alpha \in (0, 1)$ , lúc này  $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$ . Để tránh trường hợp tốc độ học giảm quá thấp, ta thường thêm cận dưới vào phương trình cập nhật  $\eta_{t+1} \leftarrow \max(\eta_{\min}, \eta_t \cdot \alpha)$ .

```
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(np.arange(50), [scheduler(t) for t in range(50)])
```

Cách trên cũng có thể được thực hiện bằng một bộ định thời có sẵn trong MXNet lr\_scheduler. FactorScheduler. Phương pháp này yêu cầu nhiều tham số hơn một chút, ví dụ như thời gian khởi động (*warmup period*), chế độ khởi động (*warmup mode*), số bước cập nhật tối đa, v.v. Trong các phần tiếp theo, ta sẽ sử dụng các bộ định thời tốc độ học được lập trình sẵn, ở đây chỉ giải thích cách thức hoạt động của chúng. Như minh họa, việc tự xây dựng một bộ định thời nếu cần khá đơn giản.

#### Định thời Đa Thừa số

Một chiến lược thường gặp khi huấn luyện các mạng nơ-ron sâu là giữ tốc độ học không đổi trong từng khoảng và giảm tốc độ học một lượng cho trước sau mỗi khoảng. Cụ thể, với một tập thời điểm giảm tốc độ học, ví dụ như với  $s = \{15, 30\}$ , ta giảm  $\eta_{t+1} \leftarrow \eta_t \cdot \alpha$  khi  $t \in s$ . Giả sử tốc độ học giảm một nửa tại mỗi thời điểm trên, ta có thể lập trình như sau.

```
scheduler = lr_scheduler.MultiFactorScheduler(step=[15, 30], factor=0.5,  
                                              base_lr=0.5)  
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])
```

Ý tưởng đằng sau định thời tốc độ học không đổi theo khoảng đó là phương pháp này cho phép quá trình tối ưu tiếp diễn tới khi phân phối của các vector trọng số đạt tới điểm ổn định. Khi và chỉ khi đạt được trạng thái đó, ta mới giảm tốc độ học để nhắm tới điểm cực tiểu chất lượng hơn. Ví dụ dưới đây cho thấy phương pháp này giúp tìm được nghiệm tốt hơn đôi chút.

```

trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)

```

## Định thời Cô-sin

Đây là một phương pháp khá phức tạp dựa trên thực nghiệm được đề xuất bởi (Loshchilov & Hutter, 2016). Phương pháp dựa trên thực nghiệm nói rằng ta có thể không muốn giảm tốc độ học quá nhanh ở giai đoạn đầu. Hơn nữa, ta có thể muốn cải thiện nghiệm thu được ở giai đoạn cuối của quá trình tối ưu bằng cách sử dụng tốc độ học rất nhỏ. Từ đó ta thu được một định thời có dạng giống cô-sin với tốc độ học trong khoảng  $t \in [0, T]$  có công thức như sau.

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2} (1 + \cos(\pi t/T)) \quad (13.15.1)$$

Trong đó  $\eta_0$  là tốc độ học ban đầu,  $\eta_T$  được tốc độ học đích tại thời điểm  $T$ . Hơn nữa, với  $t > T$  ta không tăng giá trị tốc độ học mà đơn giản giữ ở  $\eta_T$ . Trong ví dụ sau, chúng ta thiết lập số bước cập nhật tối đa  $T = 20$ .

```

scheduler = lr_scheduler.CosineScheduler(max_update=20, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])

```

Trong ngữ cảnh thị giác máy tính, cách định thời này *có thể* cải thiện kết quả thu được. Tuy nhiên, lưu ý rằng định thời cô-sin không đảm bảo chắc chắn sẽ cải thiện kết quả (có thể thấy qua ví dụ dưới đây).

```

trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)

```

## Khởi động

Trong một số trường hợp, khởi tạo tham số không đủ để đảm bảo sẽ có kết quả tốt. Đặc biệt đây là vấn đề đối với các cấu trúc mạng tiên tiến, trong đó việc tối ưu hóa có thể không ổn định. Ta có thể giải quyết vấn đề này bằng cách chọn tốc độ học đủ nhỏ để ngăn phân kỳ lúc bắt đầu. Tuy nhiên, tiến trình học sẽ chậm. Ngược lại, tốc độ học lớn ban đầu lại gây ra phân kỳ.

Một giải pháp đơn giản cho vấn đề trên là dùng quá trình khởi động (*warmup*), trong thời gian đó tốc độ học *tăng* tới giá trị lớn nhất, sau đó giảm dần tới khi kết thúc quá trình tối ưu. Để đơn giản, ta có thể khởi động bằng hàm tăng tuyến tính. Kết quả, ta có bộ định thời dưới đây.

```

scheduler = lr_scheduler.CosineScheduler(20, warmup_steps=5, base_lr=0.5,
                                         final_lr=0.01)
d2l.plot(np.arange(num_epochs), [scheduler(t) for t in range(num_epochs)])

```

Có thể thấy rằng mạng hội tụ tốt hơn ban đầu (cụ thể, hãy quan sát 5 epoch đầu tiên).

```

trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'lr_scheduler': scheduler})
train(net, train_iter, test_iter, num_epochs, loss, trainer, ctx)

```

Việc khởi động có thể sử dụng trong bất kỳ bộ định thời nào (không chỉ cô-sin). Để biết chi tiết thảo luận và các thí nghiệm về định thời tốc độ học, có thể đọc thêm ([Gotmare et al., 2018](#)). Đặc biệt, các tác giả thấy rằng quá trình khởi động làm giảm độ phân kỳ của tham số trong các mạng rất sâu. Điều này hợp lý về trực giác, vì ta thấy rằng phân kỳ mạnh là do khởi tạo ngẫu nhiên ở những phần mạng học lâu nhất vào lúc đầu.

#### 13.15.4 Tóm tắt

- Giảm tốc độ học trong huấn luyện có thể cải thiện độ chính xác và giảm tính quá khớp của mô hình.
- Một cách rất hiệu quả trong thực tế đó là giảm tốc độ học theo khoảng bất cứ khi nào quá trình tối ưu không có tiến bộ đáng kể (*plateau*). Về cơ bản, định thời trên đảm bảo quá trình tối ưu sẽ hội tụ đến nghiệm phù hợp và chỉ sau đó mới giảm phương sai vốn có của các tham số bằng cách giảm tốc độ học.
- Định thời cô-sin khá phổ biến trong các bài toán thị giác máy tính. Xem ví dụ [GluonCV<sup>256</sup>](#) để biết thêm chi tiết về định thời này.
- Quá trình khởi động trước khi tối ưu có thể giúp tránh phân kỳ.
- Tối ưu hóa phục vụ nhiều mục đích trong việc học sâu. Bên cạnh việc cực tiểu hóa hàm mục tiêu trên tập huấn luyện, các thuật toán tối ưu và các định thời tốc độ học khác nhau có thể thay đổi tính khái quát hóa và tính quá khớp trên tập kiểm tra (đối với cùng một giá trị lỗi trên tập huấn luyện).

#### 13.15.5 Bài tập

1. Hãy thí nghiệm về cách hoạt động của thuật toán tối ưu với một tốc độ học cố định cho trước. Hãy cho biết mô hình tốt nhất mà bạn có thể đạt được theo cách này?
2. Quá trình hội tụ thay đổi như thế nào nếu bạn thay đổi lũy thừa giảm trong tốc độ học? Để thuận tiện, hãy sử dụng PolyScheduler.
3. Hãy áp dụng định thời cô-sin cho nhiều bài toán thị giác máy tính, ví dụ, huấn luyện trên tập ImageNet. Hãy so sánh chất lượng mô hình khi dùng phương pháp này so với các loại định thời khác.
4. Quá trình khởi động nên kéo dài bao lâu?
5. Bạn có thể liên hệ tối ưu hóa và phép lấy mẫu được không? Hãy bắt đầu bằng cách sử dụng kết quả từ ([Welling & Teh, 2011](#)) về động lực học Langevin của Gradient ngẫu nhiên (*Stochastic Gradient Langevin Dynamics*).

<sup>256</sup> <http://gluon-cv.mxnet.io>

### **13.15.6 Thảo luận**

- Tiếng Anh - MXNet<sup>257</sup>
- Tiếng Việt<sup>258</sup>

### **13.15.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Hoang Van-Tien
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường

### **13.16 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức

---

<sup>257</sup> <https://discuss.d2l.ai/t/359>

<sup>258</sup> <https://forum.machinelearningcoban.com/c/d2l>

# 14 | Hiệu năng Tính toán

Trong học sâu, các tập dữ liệu thường rất lớn và mô hình tính toán rất phức tạp. Vì vậy, ta luôn cần quan tâm tới vấn đề hiệu năng tính toán. Trong chương này, ta sẽ tập trung vào các yếu tố then chốt ảnh hưởng đến hiệu năng tính toán: lập trình mệnh lệnh, lập trình ký hiệu, lập trình bất đồng bộ, tính toán song song tự động và tính toán đa GPU. Qua đó, độc giả có thể cải thiện nhiều hơn về hiệu năng tính toán của mô hình đã được triển khai trong các chương trước, như là giảm thời gian huấn luyện mà không ảnh hưởng tới độ chính xác của mô hình.

## 14.1 Trình biên dịch và Trình thông dịch

Cho đến nay, ta mới chỉ tập trung vào lập trình mệnh lệnh, kiểu lập trình sử dụng các câu lệnh như `print`, `+` hay `if` để thay đổi trạng thái của chương trình. Hãy cùng xét ví dụ đơn giản sau về lập trình mệnh lệnh.

```
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))
```

Python là một ngôn ngữ thông dịch. Khi thực hiện hàm `fancy_func`, nó thực thi các lệnh trong thân hàm một cách *tuần tự*. Như vậy, nó sẽ chạy lệnh `e = add(a, b)` rồi sau đó lưu kết quả vào biến `e`, làm cho trạng thái chương trình thay đổi. Hai câu lệnh tiếp theo `f = add(c, d)` và `g = add(e, f)` sẽ được thực thi tương tự, thực hiện phép cộng và lưu kết quả vào các biến. Fig. 14.1.1 sẽ minh họa luồng dữ liệu.

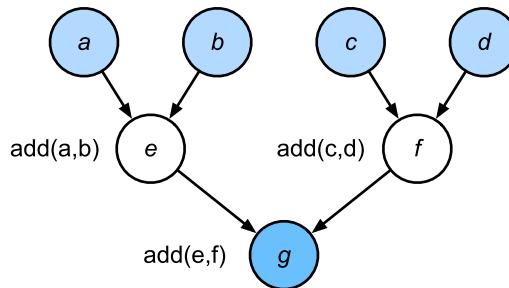


Fig. 14.1.1: Luồng dữ liệu trong lập trình mệnh lệnh.

Mặc dù lập trình mệnh lệnh rất thuận tiện, nhưng nó lại không quá hiệu quả. Ở đây nếu hàm add được gọi nhiều lần trong fancy\_func, Python cũng sẽ thực thi ba lần gọi hàm độc lập. Nếu điều này xảy ra, giả sử trên một GPU (hay thậm chí nhiều GPU), chi phí phát sinh từ trình thông dịch Python có thể sẽ rất lớn. Hơn nữa, nó sẽ cần phải lưu giá trị các biến e và f cho tới khi tất cả các lệnh trong fancy\_func thực thi xong. Điều này là do ta không biết liệu biến e và f có được sử dụng bởi các phần chương trình khác sau hai lệnh e = add(a, b) và f = add(c, d) nữa hay không.

### 14.1.1 Lập trình Ký hiệu

Lập trình ký hiệu là kiểu lập trình mà ở đó các tính toán thường chỉ được thực hiện một khi chương trình đã được định nghĩa đầy đủ. Cơ chế này được sử dụng trong nhiều framework, bao gồm: Theano, Keras và TensorFlow (hai framework sau đã hỗ trợ lập trình mệnh lệnh). Lập trình ký hiệu thường gồm những bước sau:

1. Khai báo các thao tác sẽ được thực thi.
2. Biên dịch các thao tác thành chương trình có thể chạy được.
3. Thực thi bằng cách cung cấp đầu vào và gọi chương trình đã được biên dịch.

Quy trình trên cho phép chúng ta tối ưu hóa chương trình một cách đáng kể. Đầu tiên, ta có thể bỏ qua trình thông dịch Python trong nhiều trường hợp, từ đó loại bỏ được vấn đề nghẽn cổ chai có thể ảnh hưởng nghiêm trọng tới tốc độ tính toán khi sử dụng nhiều GPU tốc độ cao với một luồng Python duy nhất trên CPU. Thứ hai, trình biên dịch có thể tối ưu và viết lại mã nguồn thành print((1 + 2) + (3 + 4)) hoặc thậm chí print(10). Điều này hoàn toàn khả thi bởi trình biên dịch có thể thấy toàn bộ mã nguồn rồi mới dịch sang mã máy. Ví dụ, nó có thể giải phóng bộ nhớ (hoặc không cấp phát) bất cứ khi nào một biến không còn được dùng đến. Hoặc nó có thể chuyển toàn bộ mã nguồn thành một đoạn tương đương. Để hiểu rõ hơn vấn đề, dưới đây ta sẽ thử mô phỏng quá trình lập trình mệnh lệnh (dựa trên Python).

```
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
...

def evoke_():
    return add_() + fancy_func_() + 'print(fancy_func(1, 2, 3, 4))'

prog = evoke_()
print(prog)
y = compile(prog, '', 'exec')
exec(y)
```

Sự khác biệt giữa lập trình mệnh lệnh (thông dịch) và lập trình ký hiệu như sau:

- Lập trình mệnh lệnh dễ hơn. Khi lập trình mệnh lệnh được sử dụng trong Python, mã nguồn trông rất trực quan và dễ viết. Mã nguồn của lập trình mệnh lệnh cũng dễ gỡ lỗi hơn. Điều này là do ta có thể dễ dàng lấy và in ra giá trị của các biến trung gian liên quan, hoặc sử dụng công cụ gỡ lỗi có sẵn của Python.
- Lập trình ký hiệu lại hiệu quả hơn và dễ sử dụng trên nền tảng khác. Nó giúp việc tối ưu mã nguồn trong quá trình biên dịch trở nên dễ dàng hơn, đồng thời cho phép ta chuyển đổi chương trình sang một định dạng khác không phụ thuộc vào Python. Do đó chương trình có thể chạy trong các môi trường khác ngoài Python, từ đó tránh được mọi vấn đề tiềm ẩn về hiệu năng liên quan tới trình thông dịch Python.

### 14.1.2 Lập trình Hybrid

Trong quá khứ, hầu hết các framework đều chọn một trong hai phương án tiếp cận: lập trình mệnh lệnh hoặc lập trình ký hiệu. Ví dụ như Theano, TensorFlow, Keras và CNTK đều xây dựng mô hình dạng ký hiệu. Ngược lại, Chainer và PyTorch tiếp cận theo hướng lập trình mệnh lệnh. Mô hình kiểu mệnh lệnh đã được bổ sung vào TensorFlow 2.0 (through qua chế độ Eager) và Keras trong những bản cập nhật sau này. Khi thiết kế Gluon, các nhà phát triển đã cân nhắc liệu rằng có thể kết hợp ưu điểm của cả hai mô hình lập trình lại với nhau hay không. Điều này đã dẫn đến mô hình hybrid, giúp người dùng phát triển và gỡ lỗi bằng lập trình mệnh lệnh thuận, đồng thời mang lại khả năng chuyển đổi hầu như toàn bộ chương trình sang dạng ký hiệu khi cần triển khai thành sản phẩm với hiệu năng tính toán cao.

Trong thực tiễn, ta sẽ xây dựng mô hình bằng lớp `HybridBlock` hoặc `HybridSequential` và `HybridConcurrent`. Mặc định, chúng được thực thi giống hệt như cách lớp `Block` hoặc `Sequential` và `Concurrent` được thực thi trong kiểu lập trình mệnh lệnh. `HybridSequential` là một lớp con của `HybridBlock` (cũng như `Sequential` là lớp con của `Block`). Khi hàm `hybridize` được gọi, Gluon biên dịch mô hình thành định dạng được dùng trong lập trình ký hiệu. Điều này cho phép ta tối ưu các thành phần nặng về mặt tính toán mà không cần có nhiều thay đổi trong cách lập trình mô hình. Chúng tôi sẽ minh họa lợi ích của việc này ở ví dụ bên dưới, tập trung vào các mô hình `Sequential` và `Block` (mô hình `Concurrent` cũng sẽ hoạt động tương tự).

### 14.1.3 HybridSequential

Cách đơn giản nhất để hiểu cách hoạt động của phép hybrid hóa là xem xét các mạng sâu đa tầng. Thông thường, trình thông dịch Python sẽ thực thi mã nguồn cho tất cả các tầng để sinh một lệnh mà sau đó có thể được truyền tới CPU hoặc GPU. Đối với thiết bị tính toán đơn (và nhanh), quá trình trên không gây ra vấn đề lớn nào cả. Mặt khác, nếu ta sử dụng một máy chủ tiên tiến có 8 GPU, ví dụ như P3dn.24xlarge trên AWS, Python sẽ gặp khó khăn trong việc tận dụng tất cả các GPU cùng lúc. Lúc này trình thông dịch Python đơn luồng sẽ trở thành nút nghẽn cổ chai. Hãy xem làm thế nào để giải quyết vấn đề trên cho phần lớn đoạn mã nguồn bằng cách thay `Sequential` bằng `HybridSequential`. Chúng ta bắt đầu với việc định nghĩa một mạng MLP đơn giản.

```
from d2l import mxnet as d2l
from mxnet import np, npx
from mxnet.gluon import nn
npx.set_np()

# factory for networks
def get_net():
```

(continues on next page)

```

net = nn.HybridSequential()
net.add(nn.Dense(256, activation='relu'),
       nn.Dense(128, activation='relu'),
       nn.Dense(2))
net.initialize()
return net

x = np.random.normal(size=(1, 512))
net = get_net()
net(x)

```

Bằng cách gọi hàm hybridize, ta có thể biên dịch và tối ưu hóa các tính toán trong MLP. Kết quả tính toán của mô hình vẫn không thay đổi.

```

net.hybridize()
net(x)

```

Điều này có vẻ tốt đến mức khó tin: chỉ cần ta chỉ định một khối trở thành HybridSequential, sử dụng đoạn mã y hệt như trước và gọi hàm hybridize. Một khi thực hiện xong những việc trên, mạng sẽ được tối ưu hóa (chúng ta sẽ đánh giá hiệu năng ở phía dưới). Tiếc là cách này không hoạt động với mọi tầng. Nhưng các khối được cung cấp sẵn bởi Gluon mặc định được kế thừa từ lớp HybridBlock và do đó có thể hybrid hóa được. Tầng kế thừa từ lớp Block sẽ không thể tối ưu hóa được.

## Tăng tốc bằng Hybrid hóa

Để minh họa những cải thiện đạt được từ quá trình biên dịch, ta hãy so sánh thời gian cần thiết để đánh giá net(x) trước và sau phép hybrid hóa. Đầu tiên hãy định nghĩa một hàm để đo thời gian trên. Hàm này sẽ hữu ích trong suốt chương này khi chúng ta đo (và cải thiện) hiệu năng.

```

#@save
class Benchmark:
    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 ta có thể gọi mạng hai lần với có hybrid hóa và không hybrid hóa.

```

net = get_net()
with Benchmark('Without hybridization'):
    for i in range(1000): net(x)
    npx.waitall()

net.hybridize()
with Benchmark('With hybridization'):

```

(continues on next page)

```
for i in range(1000): net(x)
npx.waitall()
```

Như quan sát được trong các kết quả trên, sau khi thực thi HybridSequential gọi hàm hybridize, hiệu năng tính toán được cải thiện thông qua việc sử dụng lập trình ký hiệu.

## Chuỗi hóa

Một trong những lợi ích của việc biên dịch các mô hình là ta có thể chuỗi hóa (*serialize*) mô hình và các tham số mô hình để lưu trữ. Điều này cho phép ta lưu trữ mô hình mà không phụ thuộc vào ngôn ngữ front-end. Điều này cũng cho phép ta sử dụng các mô hình đã huấn luyện trên các thiết bị khác và dễ dàng sử dụng các ngôn ngữ lập trình front-end khác. Đồng thời, mã nguồn này thường thực thi nhanh hơn so với khi lập trình mệnh lệnh. Hãy xem xét phương thức export sau.

```
net.export('my_mlp')
!ls -lh my_mlp*
```

Mô hình này được chia ra thành một tập tin (nhị phân) lớn chứa tham số và tập tin JSON mô tả cấu trúc mô hình. Các tập tin có thể được đọc bởi các ngôn ngữ front-end khác được hỗ trợ bởi Python hoặc MXNet, ví dụ như C++, R, Scala, và Perl. Tập tin JSON có dạng như sau

```
!head my_mlp-symbol.json
```

Mỗi thứ trở nên phức tạp hơn một chút khi làm việc với các mô hình gần với mã nguồn. Về cơ bản, việc hybrid hóa cần giải quyết trực tiếp luồng điều khiển và các chi phí tính toán của Python.

Hơn nữa, trong khi thực thi của lớp Block cần sử dụng hàm forward, thực thi của lớp HybridBlock lại sử dụng hàm hybrid\_forward.

Trên đây chúng ta thấy rằng phương thức hybridize có thể giúp mô hình đạt được hiệu năng tính toán và tính cơ động vượt trội hơn. Dù vậy, sự hybrid hóa có thể ảnh hưởng tới tính linh hoạt của mô hình, đặc biệt là trong điều khiển luồng. Ta sẽ minh họa cách thiết kế các mô hình tổng quát hơn cũng như cách trình biên dịch loại bỏ các thành phần thừa trong Python.

```
class HybridNet(nn.HybridBlock):
    def __init__(self, **kwargs):
        super(HybridNet, self).__init__(**kwargs)
        self.hidden = nn.Dense(4)
        self.output = nn.Dense(2)

    def hybrid_forward(self, F, x):
        print('module F: ', F)
        print('value x: ', x)
        x = F.npx.relu(self.hidden(x))
        print('result : ', x)
        return self.output(x)
```

Đoạn mã trên biểu diễn một mạng đơn giản với 4 nút ẩn và 2 đầu ra. Phương thức hybrid\_forward lấy thêm một đối số - mô-đun F. Đối số này là cần thiết để chọn thư viện xử lý phù hợp (ndarray hoặc symbol) tùy vào việc chương trình có được hybrid hóa hay không. Cả hai lớp này thực hiện

các chức năng rất giống nhau và MXNet sẽ tự động xác định đối số đầu vào. Để hiểu chuyện gì đang diễn ra chúng ta sẽ in các đối số đầu vào khi gọi hàm.

```
net = HybridNet()  
net.initialize()  
x = np.random.normal(size=(1, 3))  
net(x)
```

Lặp lại nhiều lần việc tính lượt truyền xuôi sẽ cho ra cùng kết quả (ta bỏ qua chi tiết). Nay xem chuyện gì xảy ra nếu ta kích hoạt phương thức hybridize.

```
net.hybridize()  
net(x)
```

Thay vì ndarray, lúc này ta sử dụng mô-đun symbol cho F. Thêm vào đó, mặc dù đầu vào thuộc kiểu ndarray, dữ liệu truyền qua mạng bây giờ được chuyển thành kiểu symbol như một phần của quá trình biên dịch. Việc gọi lại hàm net dẫn tới một kết quả đáng kinh ngạc:

```
net(x)
```

Điều này khá khác biệt so với những gì ta đã thấy trước đó. Tất cả các lệnh in được định nghĩa trong hybrid\_forward đều bị bỏ qua. Thật vậy, sau khi hybrid hóa, việc thực thi lệnh net(x) không còn liên quan gì tới trình thông dịch của Python nữa. Nghĩa là bất cứ đoạn mã Python nào không cần thiết cho tính toán sẽ bị bỏ qua (chẳng hạn như các lệnh in) để việc thực thi trôi chảy hơn và hiệu năng tốt hơn. Và thay vì gọi Python, MXNet gọi trực tiếp back-end C++.

Cũng nên lưu ý rằng một số hàm không được hỗ trợ trong mô-đun symbol (như asnumpy) và các toán tử thực thi tại chỗ (*in-place*) như  $a += b$  và  $a[:] = a + b$  phải được viết lại là  $a = a + b$ . Tuy nhiên, việc biên dịch mô hình vẫn đáng để thực hiện bất cứ khi nào ta quan tâm đến tốc độ. Lợi ích về tốc độ này có thể tăng từ vài phần trăm tới hơn hai lần, tùy thuộc vào sự phức tạp của mô hình, tốc độ của CPU, tốc độ và số lượng GPU.

#### 14.1.4 Tóm tắt

- Lập trình mệnh lệnh khiến việc thiết kế mô hình mới dễ dàng hơn vì ta có thể viết mã với luồng điều khiển và được sử dụng hệ sinh thái phần mềm của Python.
- Lập trình ký hiệu đòi hỏi chúng ta định nghĩa và biên dịch chương trình trước khi thực thi nó. Lợi ích là hiệu năng được cải thiện.
- MXNet có thể kết hợp những ưu điểm của cả hai phương pháp khi cần thiết.
- Mô hình được xây dựng bởi các lớp HybridSequential và HybridBlock có thể chuyển đổi các chương trình mệnh lệnh thành các chương trình ký hiệu bằng cách gọi phương thức hybridize.

### 14.1.5 Bài tập

1. Hãy thiết kế một mạng bằng cách sử dụng lớp HybridConcurrent, có thể thử với GoogleNet trong :ref: sec\_googlenet.
2. Hãy thêm `x.asnumpy()` vào dòng đầu tiên của hàm `hybrid_forward` trong lớp HybridNet, rồi thực thi mã nguồn và quan sát các lỗi bạn gặp phải. Tại sao các lỗi này xảy ra?
3. Điều gì sẽ xảy ra nếu ta thêm luồng điều khiển, cụ thể là các lệnh Python `if` và `for` trong hàm `hybrid_forward`?
4. Hãy lập trình các mô hình bạn thích trong các chương trước bằng cách sử dụng lớp HybridBlock hoặc HybridSequential.

### 14.1.6 Thảo luận

- Tiếng Anh - MXNet<sup>259</sup>
- Tiếng Việt<sup>260</sup>

### Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Văn Tâm
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Phạm Minh Đức
- Nguyễn Văn Cường

## 14.2 Tính toán Bất đồng bộ

Máy tính ngày nay là các hệ thống có tính song song cao, được cấu thành từ nhiều lõi CPU (mỗi lõi thường có nhiều luồng), nhiều phần tử xử lý trong mỗi GPU và thường có nhiều GPU trong mỗi máy. Nói ngắn gọn, ta có thể xử lý nhiều tác vụ cùng một lúc, thường là trên nhiều thiết bị khác nhau. Tiếc thay, Python không phải là một ngôn ngữ phù hợp để viết mã tính toán song song và bất đồng bộ, nhất là khi không có sự trợ giúp từ bên ngoài. Xét cho cùng, Python là ngôn ngữ đơn luồng, và có lẽ trong tương lai sẽ không có gì thay đổi. Các framework học sâu như MXNet và TensorFlow tận dụng mô hình lập trình bất đồng bộ để cải thiện hiệu năng (PyTorch sử dụng bộ định thời của chính Python nên có tiêu chí đánh đổi hiệu năng khác). Do đó, việc hiểu rõ cách lập trình bất đồng bộ giúp ta phát triển các chương trình hiệu quả hơn bằng cách chủ động giảm thiểu

<sup>259</sup> <https://discuss.d2l.ai/t/360>

<sup>260</sup> <https://forum.machinelearningcoban.com/c/d2l>

yêu cầu tính toán và các quan hệ phụ thuộc tương hỗ. Việc này cho phép ta giảm chi phí tính toán phụ trợ và tăng khả năng tận dụng vi xử lý. Ta bắt đầu bằng việc nhập các thư viện cần thiết.

```
from d2l import mxnet as d2l
import numpy, os, subprocess
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()
```

### 14.2.1 Bắt đồng bộ qua Back-end

Để khởi động, hãy cùng xét một bài toán nhỏ - ta muốn sinh ra một ma trận ngẫu nhiên và nhân nó lên nhiều lần. Hãy thực hiện việc này bằng cả NumPy và NumPy của MXNet để xem xét sự khác nhau.

```
with d2l.Benchmark('numpy'):
    for _ in range(10):
        a = numpy.random.normal(size=(1000, 1000))
        b = numpy.dot(a, a)

with d2l.Benchmark('mxnet.np'):
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
```

NumPy của MXNet nhanh hơn tới cả hàng trăm hàng ngàn lần. Ít nhất là có vẻ là như vậy. Do cả hai thư viện đều được thực hiện trên cùng một bộ xử lý, chắc hẳn phải có gì đó ảnh hưởng đến kết quả. Nếu ta ép MXNet phải hoàn thành tất cả phép tính trước khi trả về kết quả, ta có thể thấy rõ điều gì đã xảy ra ở trên: phần tính toán được thực hiện bởi back-end trong khi front-end đã trả lại quyền điều khiển cho Python.

```
with d2l.Benchmark():
    for _ in range(10):
        a = np.random.normal(size=(1000, 1000))
        b = np.dot(a, a)
        npx.waitall()
```

Nhìn chung, MXNet có front-end cho phép tương tác trực tiếp với người dùng thông qua Python, cũng như back-end được sử dụng bởi hệ thống nhằm thực hiện nhiệm vụ tính toán. Như ở Fig. 14.2.1, người dùng có thể viết chương trình MXNet bằng nhiều ngôn ngữ front-end như Python, R, Scala và C++. Dù sử dụng ngôn ngữ front-end nào, chương trình MXNet chủ yếu thực thi trên back-end lập trình bằng C++. Các thao tác đưa ra bởi ngôn ngữ front-end được truyền vào back-end để thực thi. Back-end tự quản lý các luồng xử lý bằng việc liên tục tập hợp và thực thi các tác vụ trong hàng đợi. Chú ý rằng, back-end cần phải có khả năng theo dõi quan hệ phụ thuộc giữa các bước trong đồ thị tính toán để có thể hoạt động. Nghĩa là ta không thể song song hóa các thao tác phụ thuộc lẫn nhau.

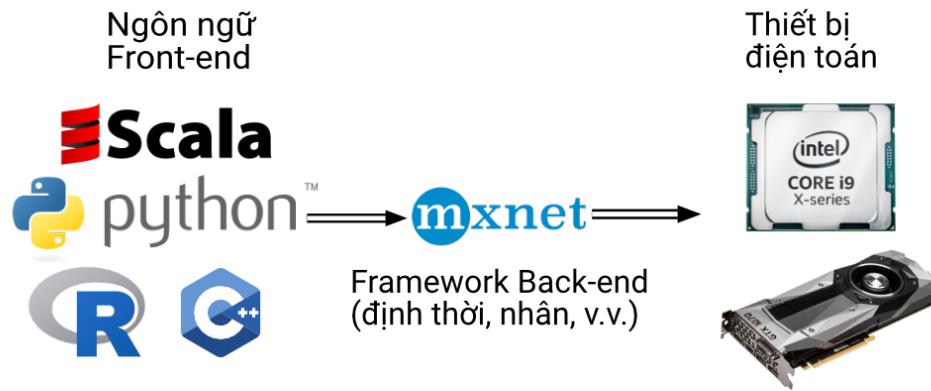


Fig. 14.2.1: Lập trình Front-end.

Hãy xét một ví dụ đơn giản để có thể hiểu rõ hơn đồ thị quan hệ phụ thuộc (*dependency graph*).

```
x = np.ones((1, 2))
y = np.ones((1, 2))
z = x * y + 2
z
```

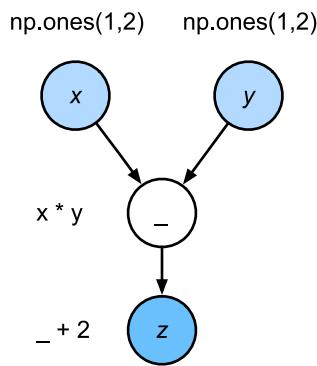


Fig. 14.2.2: Quan hệ phụ thuộc.

Đoạn mã trên cũng được mô tả trong Fig. 14.2.2. Mỗi khi luồng front-end của Python thực thi một trong ba câu lệnh đầu tiên, nó sẽ chỉ đưa tác vụ đó vào hàng chờ của back-end. Khi kết quả của câu lệnh cuối cùng cần được in ra, luồng front-end của Python sẽ chờ luồng xử lý back-end C++ tính toán xong kết quả của biến z. Lợi ích của thiết kế này nằm ở việc luồng front-end Python không cần phải đích thân thực hiện việc tính toán. Do đó, hiệu năng tổng thể của chương trình cũng ít bị ảnh hưởng bởi hiệu năng của Python. Fig. 14.2.3 mô tả cách front-end và back-end tương tác với nhau.

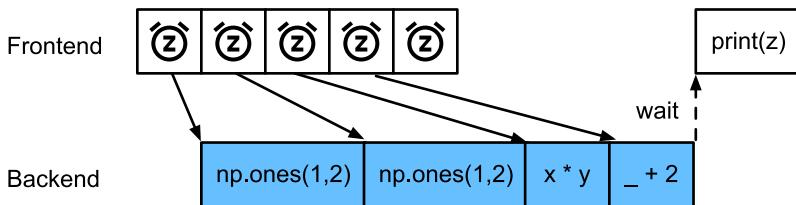


Fig. 14.2.3: Front-end và Back-end

### 14.2.2 Lớp cản và Bộ chặn

Có khá nhiều thao tác buộc Python phải chờ cho đến khi nó hoàn thành:

- Hiển nhiên nhất là lệnh `npx.waitall()` chờ đến khi toàn bộ phép toán đã hoàn thành, bất chấp thời điểm câu lệnh tính toán được đưa ra. Trong thực tế, trừ khi thực sự cần thiết, việc sử dụng thao tác này là một ý tưởng tồi do nó có thể làm giảm hiệu năng.
- Nếu ta chỉ muốn chờ đến khi một biến cụ thể nào đó sẵn sàng, ta có thể gọi `z.wait_to_read()`. Trong trường hợp này MXNet chặn việc trả luồng điều khiển về Python cho đến khi biến `z` đã được tính xong. Các thao tác khác sau đó mới có thể tiếp tục.

Hãy xem cách các lệnh chờ trên hoạt động trong thực tế:

```
with d2l.Benchmark('waitall'):
    b = np.dot(a, a)
    npx.waitall()

with d2l.Benchmark('wait_to_read'):
    b = np.dot(a, a)
    b.wait_to_read()
```

Cả hai thao tác hoàn thành với thời gian xấp xỉ nhau. Ngoài các thao tác chặn (*blocking operation*) tường minh, bạn đọc cũng nên biết về việc chặn *ngầm*. Rõ ràng việc in một biến ra yêu cầu biến đó phải sẵn sàng và do đó nó là một bộ chặn. Cuối cùng, ép kiểu sang NumPy bằng `z.astype()` và ép kiểu sang số vô hướng bằng `z.item()` cũng là bộ chặn, do trong NumPy không có khái niệm bất đồng bộ. Có thể thấy việc ép kiểu cũng cần truy cập giá trị, giống như hàm `print`. Việc thường xuyên sao chép một lượng nhỏ dữ liệu từ phạm vi của MXNet sang NumPy và ngược lại có thể làm giảm đáng kể hiệu năng của một đoạn mã đáng lẽ sẽ có hiệu năng tốt, do mỗi thao tác như vậy buộc đồ thị tính toán phải tính toán bộ các giá trị trung gian để suy ra các số hạng cần thiết *trước khi* thực hiện bất cứ thao tác nào khác.

```
with d2l.Benchmark('numpy conversion'):
    b = np.dot(a, a)
    b.astype()

with d2l.Benchmark('scalar conversion'):
    b = np.dot(a, a)
    b.sum().item()
```

### 14.2.3 Cải thiện Năng lực Tính toán

Trong một hệ thống đa luồng lớn (ngay cả laptop phổ thông cũng có 4 luồng hoặc hơn, và trên các máy trạm đa socket, số luồng có thể vượt quá 256), chi phí phụ trợ từ việc định thời các thao tác có thể trở nên khá lớn. Đó là lý do tại sao hai quá trình tính toán và định thời nên xảy ra song song và bất đồng bộ. Để minh họa cho lợi ích của việc này, hãy so sánh khi liên tục cộng 1 vào một biến theo cách đồng bộ và bất đồng bộ. Ta mô phỏng quá trình thực thi đồng bộ bằng cách chèn một lớp cản `wait_to_read()` giữa mỗi phép cộng.

```
with d2l.Benchmark('synchronous'):
    for _ in range(1000):
        y = x + 1
        y.wait_to_read()

with d2l.Benchmark('asynchronous'):
    for _ in range(1000):
        y = x + 1
        y.wait_to_read()
```

Ta có thể tóm tắt đơn giản sự tương tác giữa luồng front-end Python và luồng back-end C++ như sau:

1. Front-end ra lệnh cho back-end đưa tác vụ tính  $y = x + 1$  vào hàng đợi.
2. Back-end sau đó nhận các tác vụ tính toán từ hàng đợi và thực hiện các phép tính.
3. Back-end trả kết quả tính toán về cho front-end.

Giả sử thời gian thực hiện mỗi giai đoạn trên lần lượt là  $t_1, t_2$  và  $t_3$ . Nếu ta không áp dụng lập trình bất đồng bộ, tổng thời gian để thực hiện 1000 phép tính xấp xỉ bằng  $1000(t_1 + t_2 + t_3)$ . Còn nếu ta áp dụng lập trình bất đồng bộ, tổng thời gian để thực hiện 1000 phép tính có thể giảm xuống còn  $t_1 + 1000t_2 + t_3$  (giả sử  $1000t_2 > 999t_1$ ), do front-end không cần phải chờ back-end trả về kết quả tính toán sau mỗi vòng lặp.

### 14.2.4 Cải thiện Mức chiếm dụng Bộ nhớ

Cùng hình dung với trường hợp ta liên tục thêm các tính toán vào back-end bằng cách thực thi mã Python trên front-end. Ví dụ, trong một khoảng thời gian rất ngắn, front-end liên tục thêm vào một lượng lớn các tác vụ trên minibatch. Xét cho cùng, công việc trên có thể hoàn thành nhanh chóng nếu không có phép tính nào thật sự diễn ra trên Python. Nếu tất cả tác vụ trên cùng được khởi động một cách nhanh chóng thì có thể dẫn đến dung lượng bộ nhớ sử dụng tăng đột ngột. Do dung lượng bộ nhớ có sẵn trên GPU (và ngay cả CPU) là có hạn, điều này có thể gây ra sự tranh chấp tài nguyên hoặc thậm chí làm sập chương trình. Độc giả có lẽ đã nhận ra rằng ở các quy trình huấn luyện trước, ta áp dụng các thao tác đồng bộ như `item` hay ngay cả `asnumpy`.

Chúng tôi khuyến nghị nên sử dụng các thao tác này một cách cẩn thận, ví dụ như với từng mini-batch, ta cần đảm bảo sao cho hiệu năng tính toán và mức chiếm dụng bộ nhớ (*memory footprint*) được cân bằng. Để minh họa, hãy cùng lập trình một vòng lặp huấn luyện đơn giản, đo lượng bộ nhớ tiêu hao và thời gian thực thi, sử dụng hàm sinh dữ liệu và mạng học sâu dưới đây.

```
def data_iter():
    timer = d2l.Timer()
    num_batches, batch_size = 150, 1024
```

(continues on next page)

```

for i in range(num_batches):
    X = np.random.normal(size=(batch_size, 512))
    y = np.ones((batch_size,))
    yield X, y
    if (i + 1) % 50 == 0:
        print(f'batch {i + 1}, time {timer.stop():.4f} sec')

net = nn.Sequential()
net.add(nn.Dense(2048, activation='relu'),
       nn.Dense(512, activation='relu'), nn.Dense(1))
net.initialize()
trainer = gluon.Trainer(net.collect_params(), 'sgd')
loss = gluon.loss.L2Loss()

```

Tiếp theo, ta cần công cụ để đo lường mức chiếm dụng bộ nhớ của đoạn mã trên. Để có thể xây dựng công cụ này, ta sử dụng lệnh ps của hệ điều hành (chỉ hoạt động trên Linux và macOS). Để phân tích chi tiết hoạt động của đoạn mã trên, bạn có thể sử dụng Nsight<sup>261</sup> của Nvidia hoặc vTune<sup>262</sup> của Intel.

```

def get_mem():
    res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
    return int(str(res).split()[15]) / 1e3

```

Trước khi bắt đầu kiểm tra, ta cần khởi tạo các tham số của mạng và xử lý một batch. Nếu không, việc kiểm tra dung lượng bộ nhớ sử dụng thêm sẽ là khá rắc rối. Bạn đọc có thể tham khảo Section 7.3 để hiểu rõ chi tiết việc khởi tạo.

```

for X, y in data_iter():
    break
loss(y, net(X)).wait_to_read()

```

Để đảm bảo bộ đệm tác vụ tại back-end không bị tràn, ta chèn phương thức `wait_to_read` vào back-end cho hàm mất mát ở cuối mỗi vòng lặp. Điều này buộc mỗi lượt truyền xuôi phải hoàn thành trước khi lượt truyền xuôi tiếp theo được bắt đầu. Chú ý rằng có một phương án thay thế khác (có lẽ tinh tế hơn) là theo dõi lượng mất mát ở biến vô hướng và buộc đi qua một lớp chặn (*barrier*) qua việc gọi phương thức `item`.

```

mem = get_mem()
with d2l.Benchmark('time per epoch'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l.backward()
            trainer.step(X.shape[0])
            l.wait_to_read() # Barrier before a new batch
            npx.waitall()
print(f'increased memory: {get_mem() - mem:.f} MB')

```

Như ta có thể thấy, thời gian thực hiện từng minibatch khá khớp so với tổng thời gian chạy của đoạn mã tối ưu. Hơn nữa, lượng bộ nhớ sử dụng tăng không đáng kể. Giờ hãy cùng xem chuyện

<sup>261</sup> [https://developer.nvidia.com/nsight-compute-2019\\_5](https://developer.nvidia.com/nsight-compute-2019_5)

<sup>262</sup> <https://software.intel.com/en-us/vtune>

gì sẽ xảy ra nếu ta bỏ lớp chặn ở cuối mỗi minibatch.

```
mem = get_mem()
with d2l.Benchmark('time per epoch'):
    for X, y in data_iter():
        with autograd.record():
            l = loss(y, net(X))
            l.backward()
            trainer.step(X.shape[0])
    npx.waitall()
print(f'increased memory: {get_mem() - mem:f} MB')
```

Mặc dù thời gian để đưa ra chỉ dẫn cho back-end nhỏ hơn đến hàng chục lần, ta vẫn cần thực hiện các bước tính toán. Hậu quả là một lượng lớn các kết quả trung gian không được đưa ra sử dụng và có thể chất đống trong bộ nhớ. Dù rằng việc này không gây ra bất cứ vấn đề nào trong ví dụ nhỏ trên, nó có thể dẫn đến tình trạng cạn kiệt bộ nhớ nếu không được kiểm tra trong viễn cảnh thực tế.

#### 14.2.5 Tóm tắt

- MXNet tách riêng khối front-end Python khỏi khối back-end thực thi. Điều này cho phép nhanh chóng chèn các câu lệnh một cách bất đồng bộ vào khối back-end và kết hợp tính toán song song.
- Sự bất đồng bộ giúp front-end phản ứng nhanh hơn. Tuy nhiên, cần phải áp dụng cẩn thận để không làm tràn các tác vụ ở trạng thái đợi, gây chiếm dụng bộ nhớ.
- Nên đồng bộ theo từng minibatch một để giữ cho front-end và back-end được đồng bộ tương đối.
- Nên nhớ rằng việc chuyển quản lý bộ nhớ từ MXNet sang Python sẽ buộc back-end phải chờ cho đến khi biến đó sẵn sàng. print, asnumpy và item đều gây ra hiệu ứng trên. Điều này có thể có ích đôi lúc, tuy nhiên lạm dụng chúng có thể làm sụt giảm hiệu năng.
- Nhà sản xuất vi xử lý cung cấp các công cụ phân tích hiệu năng tinh vi, cho phép đánh giá hiệu năng của học sâu một cách chi tiết hơn rất nhiều.

#### 14.2.6 Bài tập

1. Như đã đề cập ở trên, sử dụng tính toán bất đồng bộ có thể giảm tổng thời gian cần thiết để thực hiện 1000 phép tính xuống  $t_1 + 1000t_2 + t_3$ . Tại sao ở đó ta lại phải giả sử  $1000t_2 > 999t_1$ ?
2. Bạn có thể chỉnh sửa vòng lặp huấn luyện như thế nào nếu muốn xử lý 2 batch cùng lúc (đảm bảo batch  $b_t$  hoàn thành trước khi batch  $b_{t+2}$  bắt đầu)?
3. Chuyên gì sẽ xảy ra nếu thực thi mã nguồn đồng thời trên cả CPU và GPU? Liệu có nên tiếp tục đồng bộ sau khi xử lý mỗi minibatch?
4. So sánh sự khác nhau giữa waitall và wait\_to\_read. Gợi ý: thực hiện một số lệnh và đồng bộ theo kết quả trung gian.

#### 14.2.7 Thảo luận

- Tiếng Anh - MXNet<sup>263</sup>
- Tiếng Việt<sup>264</sup>

#### 14.2.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc

### 14.3 Song song hóa Tự động

MXNet tự động xây dựng các đồ thị tính toán (*computational graph*) ở back-end. Sử dụng đồ thị tính toán, hệ thống nhận biết được tất cả thành phần phụ thuộc, từ đó thực hiện song song có chọn lọc các tác vụ không liên quan đến nhau để cải thiện tốc độ. Chẳng hạn, Fig. 14.2.2 trong Section 14.2 khởi tạo hai biến độc lập. Do đó hệ thống có thể chọn để thực hiện chúng song song với nhau.

Thông thường, một toán tử đơn sẽ sử dụng toàn bộ tài nguyên tính toán trên tất cả các CPU hoặc trên một GPU đơn. Chẳng hạn như toán tử dot sẽ sử dụng tất cả các lõi (và các luồng) của toàn bộ CPU trên một máy tính đơn. Điều tương tự cũng xảy ra trên một GPU đơn. Do đó việc song song hóa không thật sự hữu dụng mấy với các máy tính đơn lõi/đơn luồng. Với các thiết bị đa xử lý thì nó lại có giá trị hơn rất nhiều. Trong khi xử lý song song thường liên quan đến các GPU, sử dụng thêm các vi xử lý CPU cục bộ trên máy sẽ tăng hiệu năng tính toán lên chút đỉnh. Tham khảo [Hadjis.Zhang.Motliagkas.ea.2016], một bài báo tập trung về việc huấn luyện mô hình thị giác máy tính kết hợp một GPU và một CPU. Với sự thuận tiện từ một framework cho phép song song hóa một cách tự động, ta có thể thực hiện việc đó chỉ với vài dòng mã lệnh Python. Mở rộng hơn, thảo luận của chúng ta về tính toán song song tự động tập trung vào tính toán song song sử dụng cả CPU và GPU, cũng như tính toán và giao tiếp song song. Chúng ta bắt đầu bằng việc nhập các gói thư viện và mô-đun cần thiết. Lưu ý rằng chúng ta cần ít nhất một GPU để chạy các thử nghiệm trong phần này.

```
from d2l import mxnet as d2l
from mxnet import np, npx
npx.set_np()
```

<sup>263</sup> <https://discuss.d2l.ai/t/361>

<sup>264</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 14.3.1 Tính toán Song song trên CPU và GPU

Ta hãy bắt đầu bằng việc định nghĩa một khối lượng công việc tham khảo để kiểm thử. Hàm `run` dưới đây thực hiện 10 phép nhân ma trận trên thiết bị mà chúng ta lựa chọn bằng cách sử dụng dữ liệu được lưu ở hai biến `x_cpu` và `x_gpu`.

```
def run(x):
    return [x.dot(x) for _ in range(10)]

x_cpu = np.random.uniform(size=(2000, 2000))
x_gpu = np.random.uniform(size=(6000, 6000), ctx=d2l.try_gpu())
```

Bây giờ ta sẽ gọi hàm với dữ liệu. Để chắc chắn rằng bộ nhớ đệm không ảnh hưởng đến kết quả, ta khởi động các thiết bị bằng việc thực hiện một lượt tính cho mỗi biến trước khi bắt đầu đo lường.

```
run(x_cpu) # Warm-up both devices
run(x_gpu)
npx.waitall()

with d2l.Benchmark('CPU time'):
    run(x_cpu)
    npx.waitall()

with d2l.Benchmark('GPU time'):
    run(x_gpu)
    npx.waitall()
```

Nếu ta bỏ `waitall()` giữa hai tác vụ thì hệ thống sẽ tự động song song hóa việc tính toán trên cả hai thiết bị.

```
with d2l.Benchmark('CPU & GPU'):
    run(x_cpu)
    run(x_gpu)
    npx.waitall()
```

Trong trường hợp phía trên, thời gian thi hành toàn bộ các tác vụ ít hơn tổng thời gian thi hành từng tác vụ riêng lẻ, bởi vì MXNet tự động định thời việc tính toán trên cả CPU và GPU mà không đòi hỏi người dùng phải cung cấp các đoạn mã phức tạp.

### 14.3.2 Tính toán và Giao tiếp Song song

Trong nhiều trường hợp ta cần di chuyển dữ liệu giữa các thiết bị như CPU và GPU, hoặc giữa các GPU với nhau. Điều này xảy ra, chẳng hạn như khi ta cần tổng hợp gradient trên các thẻ tăng tốc (*accelerator card*) khi cần thực hiện tối ưu hóa phân tán. Hãy cùng mô phỏng điều này bằng việc tính toán trên GPU và sau đó sao chép kết quả trở lại CPU.

```
def copy_to_cpu(x):
    return [y.copyto(npx.cpu()) for y in x]

with d2l.Benchmark('Run on GPU'):
    y = run(x_gpu)
    npx.waitall()
```

(continues on next page)

```
with d2l.Benchmark('Copy to CPU'):
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

Điều này có phần không hiệu quả. Lưu ý rằng ta có thể bắt đầu sao chép một vài phần đã tính xong của  $y$  đến CPU trong khi các phần còn lại của  $y$  vẫn đang được tính toán. Tình huống này có thể xảy ra khi ta tính gradient (lên truyền ngược) trên một minibatch. Gradient của một vài tham số sẽ được tính xong sớm hơn so với các tham số khác. Do đó sẽ có lợi nếu ta bắt đầu truyền dữ liệu về bằng bus băng thông PCI-Express trong khi GPU vẫn còn đang chạy. Việc bỏ đi `waitall` giữa các phần cho phép ta mô phỏng tình huống này.

```
with d2l.Benchmark('Run on GPU and copy to CPU'):
    y = run(x_gpu)
    y_cpu = copy_to_cpu(y)
    npx.waitall()
```

Thời gian cần cho cả hai thao tác ít hơn hẳn (như mong đợi) so với tổng thời gian thực hiện từng thao tác đơn lẻ. Lưu ý rằng tác vụ này khác với việc tính toán song song bởi nó sử dụng một tài nguyên khác: bus giữa CPU và các GPU. Thực tế, ta có thể vừa tính toán và giao tiếp trên cả hai thiết bị cùng một lúc. Như đã lưu ý phía trên, có một sự phụ thuộc giữa việc tính toán và giao tiếp:  $y[i]$  phải được tính xong trước khi ta có thể sao chép nó qua CPU. May mắn thay, hệ thống có thể sao chép  $y[i-1]$  trong khi tính toán  $y[i]$  để giảm thiểu tổng thời gian chạy.

Để tổng kết phần này, ta xét một ví dụ minh họa đồ thị tính toán và các quan hệ phụ thuộc của nó trong một mạng Perceptron hai tầng đơn giản khi huấn luyện trên một CPU và hai GPU, như miêu tả trong Fig. 14.3.1. Có thể thấy tự mình định thời chương trình tính toán song song từ mô tả trên sẽ khá phức tạp. Do đó, việc sử dụng back-end tính toán dựa trên đồ thị là một lợi thế để tối ưu hóa hiệu năng.

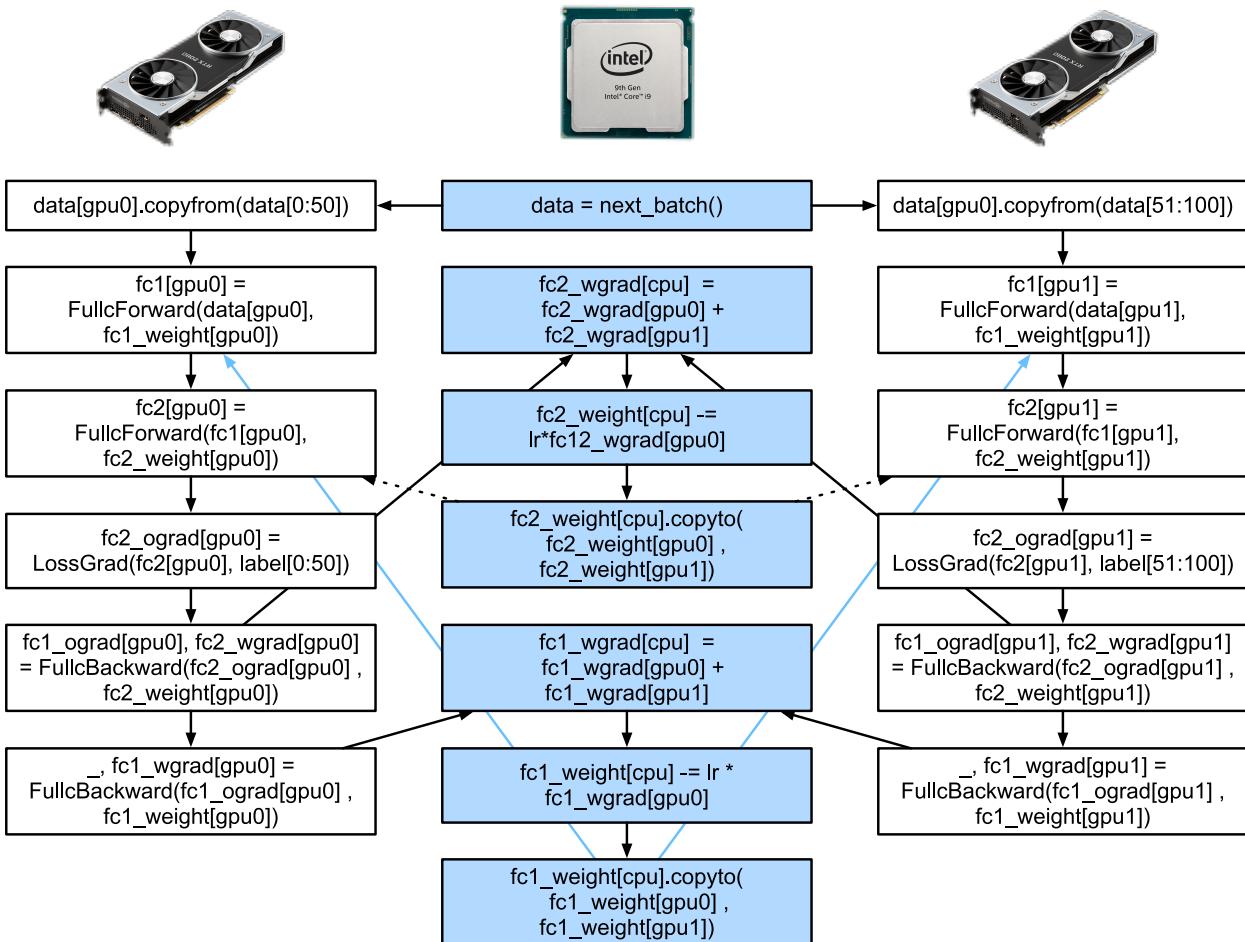


Fig. 14.3.1: Mạng Perceptron hai tầng trên một CPU và hai GPU

### 14.3.3 Tóm tắt

- Các hệ thống hiện đại thường bao gồm nhiều thiết bị, ví dụ như nhiều GPU và CPU. Các thiết bị này có thể được sử dụng song song, một cách bất đồng bộ.
- Các hệ thống hiện đại cũng có nhiều nguồn tài nguyên phục vụ cho giao tiếp, ví dụ như kết nối PCI Express, bộ nhớ (thường là SSD hoặc thông qua mạng), và băng thông mạng. Chúng có thể được sử dụng song song để đạt hiệu năng tối đa.
- Back-end có thể cải thiện hiệu năng thông qua việc tự động tính toán và giao tiếp song song.

### 14.3.4 Bài tập

- Có 10 thao tác được thực hiện trong hàm run đã được định nghĩa trong phần này. Giữa chúng không có bất cứ quan hệ phụ thuộc nào. Thiết kế một thí nghiệm để xem liệu MXNet có tự động thực thi các thao tác này một cách song song.
- Khi khối lượng công việc của một thao tác đủ nhỏ, song song hóa có thể hữu ích ngay cả khi chạy trên CPU hay GPU đơn. Thiết kế một thí nghiệm để kiểm chứng.
- Thiết kế một thí nghiệm sử dụng tính toán song song trên CPU, GPU và giao tiếp giữa cả hai thiết bị.

- Sử dụng một trình gỡ lỗi (*debugger*) như Nsight của NVIDIA để kiểm chứng rằng đoạn mã của bạn hoạt động hiệu quả.
- Thiết kế các tác vụ tính toán chứa nhiều dữ liệu có quan hệ phụ thuộc phức tạp hơn, và thực hiện các thí nghiệm để xem liệu bạn có thể thu lại kết quả chính xác trong khi vẫn cải thiện hiệu năng.

#### 14.3.5 Thảo luận

- Tiếng Anh - MXNet<sup>265</sup>
- Tiếng Việt<sup>266</sup>

#### 14.3.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Trần Yến Thy
- Phạm Minh Đức
- Đỗ Trường Giang

### 14.4 Phần cứng

Để xây dựng các hệ thống có hiệu năng cao, ta cần nắm chắc kiến thức về các thuật toán và mô hình để có thể biểu diễn được những khía cạnh thống kê của bài toán. Đồng thời, ta cũng cần có một chút kiến thức cơ bản về phần cứng thực thi ở bên dưới. Nội dung trong phần này không thể thay thế một khóa học đầy đủ về phần cứng và thiết kế hệ thống, mà sẽ chỉ đóng vai trò như điểm bắt đầu để giúp người đọc hiểu tại sao một số thuật toán lại hiệu quả hơn các thuật toán khác và làm thế nào để đạt được thông lượng cao. Thiết kế tốt có thể dễ dàng tạo ra sự khác biệt rất lớn, giữa việc có thể huấn luyện một mô hình (ví dụ trong khoảng một tuần) và không thể huấn luyện (ví dụ mất 3 tháng để huấn luyện xong, từ đó không kịp tiến độ). Ta sẽ bắt đầu bằng việc quan sát tổng thể một hệ thống máy tính. Tiếp theo, ta sẽ đi sâu hơn và xem xét chi tiết về CPU và GPU. Cuối cùng, ta sẽ tìm hiểu cách các máy tính được kết nối với nhau trong trạm máy chủ hay trên đám mây. Cần lưu ý, phần này sẽ không hướng dẫn cách lựa chọn card GPU. Nếu bạn cần gợi ý, hãy xem Section 21.5. Phần giới thiệu về điện toán đám mây trên AWS có thể tìm thấy tại Section 21.3.

<sup>265</sup> <https://discuss.d2l.ai/t/362>

<sup>266</sup> <https://forum.machinelearningcoban.com/c/d2l>

Bạn đọc có thể tham khảo nhanh thông tin tóm tắt trong Fig. 14.4.1. Nội dung này được trích dẫn từ bài viết của Colin Scott<sup>267</sup> trình bày tổng quan về những tiến bộ trong thập kỉ qua. Số liệu gốc được trích dẫn từ buổi thảo luận của Jeff Dean tại trường Stanford năm 2010<sup>268</sup>. Phần thảo luận dưới đây sẽ giải thích cơ sở cho những con số trên và cách mà chúng dẫn dắt ta trong quá trình thiết kế thuật toán. Nội dung khái quát và ngắn gọn nên nó không thể thay thế một khóa học đầy đủ, nhưng sẽ cung cấp đủ thông tin cho những người làm mô hình thống kê để có thể đưa ra lựa chọn thiết kế phù hợp. Để có cái nhìn tổng quan chuyên sâu về kiến trúc máy tính, bạn đọc có thể tham khảo (Hennessy & Patterson, 2011) hay một khóa học gần đây của Arste Asanovic<sup>269</sup>.

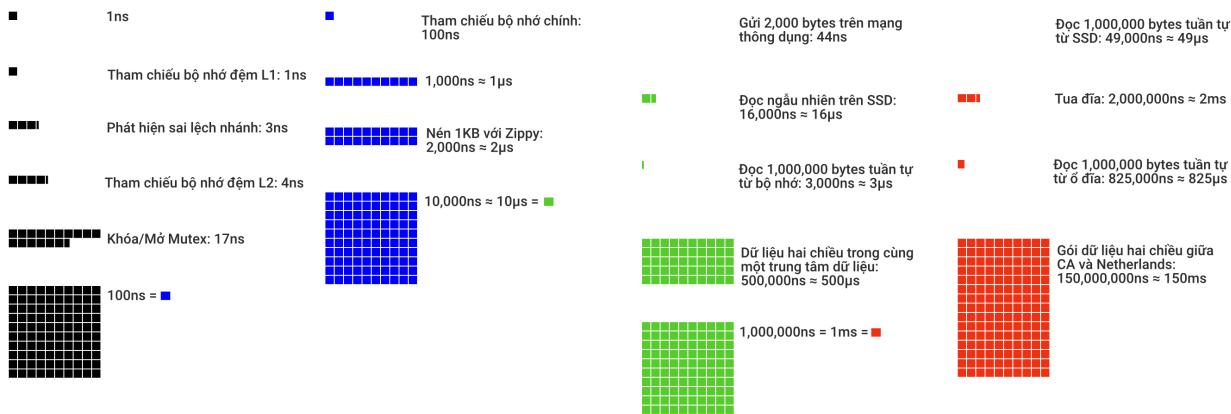


Fig. 14.4.1: Số liệu về độ trễ mà mọi lập trình viên nên biết.

#### 14.4.1 Máy tính

Hầu hết những nhà nghiên cứu học sâu đều được trang bị hệ thống máy tính có bộ nhớ và khả năng tính toán khá lớn với một hay nhiều GPU. Những máy tính này thường có những thành phần chính sau:

- Bộ xử lý, thường được gọi là CPU, có khả năng thực thi các chương trình được nhập bởi người dùng (bên cạnh chức năng chạy hệ điều hành và các tác vụ khác), thường có 8 lõi (*core*) hoặc nhiều hơn.
- Bộ nhớ (RAM) được sử dụng để lưu trữ và truy xuất các kết quả tính toán như vector trọng số, giá trị kích hoạt và dữ liệu huấn luyện.
- Một hay nhiều kết nối Ethernet với tốc độ đường truyền từ 1 Gbit/s tới 100 Gbit/s (các máy chủ tiên tiến còn có các phương thức kết nối cao cấp hơn nữa).
- Cổng giao tiếp bus mở rộng tốc độ cao (PCIe) kết nối hệ thống với một hay nhiều GPU. Các hệ thống máy chủ thường có tới 8 GPU được kết nối với nhau theo cấu trúc liên kết phức tạp. Còn các hệ thống máy tính thông thường thì có 1-2 GPU, phụ thuộc vào túi tiền của người dùng và công suất nguồn điện.
- Bộ lưu trữ tốt, thường là ổ cứng từ (HDD) hay ổ cứng thể rắn (SSD), được kết nối bằng bus PCIe giúp truyền dữ liệu huấn luyện tới hệ thống và sao lưu các checkpoint trung gian một cách hiệu quả.

<sup>267</sup> [https://people.eecs.berkeley.edu/~rcs/research/interactive\\_latency.html](https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html)

<sup>268</sup> <https://static.googleusercontent.com/media/research.google.com/en//people/jeff/Stanford-DL-Nov-2010.pdf>

<sup>269</sup> <http://inst.eecs.berkeley.edu/~cs152/sp19/>

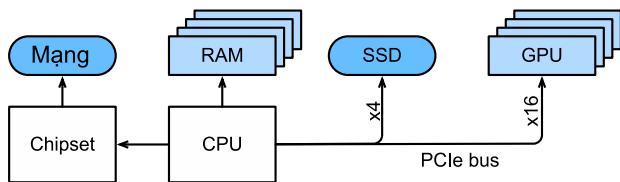


Fig. 14.4.2: Kết nối các thành phần máy tính

Hình Fig. 14.4.2 cho thấy, hầu hết các thành phần (mạng, GPU, ổ lưu trữ) được kết nối tới GPU thông qua đường bus PCI mở rộng. Đường truyền này gồm nhiều làn kết nối trực tiếp tới CPU. Ví dụ, Threadripper 3 của AMD có 64 làn PCIe 4.0, mỗi làn có khả năng truyền dẫn 16 Gbit/s dữ liệu theo cả hai chiều. Bộ nhớ được kết nối trực tiếp tới CPU với tổng băng thông lên đến 100 GB/s.

Khi ta chạy chương trình trên máy tính, ta cần trộn dữ liệu ở các bộ xử lý (CPU hay GPU), thực hiện tính toán và sau đó truyền kết quả tới RAM hay ổ lưu trữ. Do đó, để có hiệu năng tốt, ta cần đảm bảo rằng chương trình chạy mượt mà và hệ thống không có nút nghẽn cổ chai. Ví dụ, nếu ta không thể tải ánh dù nhanh, bộ xử lý sẽ không có dữ liệu để chạy. Tương tự, nếu ta không thể truyền các ma trận tới CPU (hay GPU) dù nhanh, bộ xử lý sẽ thiếu dữ liệu để hoạt động. Cuối cùng, nếu ta muốn đồng bộ nhiều máy tính trong một mạng, kết nối mạng không nên làm chậm việc tính toán. Xen kẽ việc giao tiếp và tính toán giữa các máy tính là một phương án cho vấn đề này. Giờ hãy xem xét các thành phần trên một cách chi tiết hơn.

#### 14.4.2 Bộ nhớ

Về cơ bản, bộ nhớ được sử dụng để lưu trữ dữ liệu khi cần sẵn sàng truy cập. Hiện tại bộ nhớ RAM của CPU thường thuộc loại DDR4<sup>270</sup>, trong đó mỗi mô-đun có băng thông 20-25GB/s và độ rộng bus 64 bit. Thông thường, các cặp mô-đun bộ nhớ cho phép sử dụng đa kênh. CPU có từ 2 đến 4 kênh bộ nhớ, nghĩa là chúng có băng thông bộ nhớ tối đa từ 40 GB/s đến 100 GB/s. Thường thì mỗi kênh có hai dải (bank). Ví dụ, Zen 3 Threadripper của AMD có 8 khe cắm.

Dù những con số trên trông khá ấn tượng, trên thực tế chúng chỉ nói lên một phần nào đó. Khi muốn đọc một phần nào đó từ bộ nhớ, trước tiên ta cần chỉ cho mô-đun bộ nhớ vị trí chứa thông tin, tức cần gửi *địa chỉ* đến RAM. Khi thực hiện xong việc này, ta có thể chọn chỉ đọc một bản ghi 64 bit hoặc một chuỗi dài các bản ghi. Lựa chọn thứ hai được gọi là *đọc nhanh (burst read)*. Nói ngắn gọn, việc gửi một địa chỉ vào bộ nhớ và thiết lập chuyển tiếp sẽ mất khoảng 100ns (thời gian cụ thể phụ thuộc vào hệ số thời gian của từng chip bộ nhớ được sử dụng), mỗi lần chuyển tiếp sau đó chỉ mất 0.2ns. Có thể thấy lần đọc đầu tiên tốn thời gian gấp 500 lần những lần sau! Ta có thể đọc ngẫu nhiên tối đa 10,000,000 lần mỗi giây. Điều này cho thấy rằng ta nên hạn chế tối đa việc truy cập bộ nhớ ngẫu nhiên và thay vào đó nên sử dụng cách đọc (và ghi) nhanh (*burst read*, và *burst write*).

Mọi thứ trở nên phức tạp hơn một chút khi ta tính đến việc có nhiều dải bộ nhớ. Mỗi dải có thể đọc bộ nhớ gần như là độc lập với nhau. Điều này có hai ý sau. Thứ nhất, số lần đọc ngẫu nhiên thực sự cao hơn tới 4 lần, miễn là chúng được trải đều trên bộ nhớ. Điều đó cũng có nghĩa là việc thực hiện các lệnh đọc ngẫu nhiên vẫn không phải là một ý hay vì các lệnh đọc nhanh (*burst read*) cũng nhanh hơn gấp 4 lần. Thứ hai, do việc căn chỉnh bộ nhớ theo biên 64 bit, ta nên căn chỉnh mọi cấu trúc dữ liệu theo cùng biên đó. Trình biên dịch thực hiện việc này một cách tự động<sup>271</sup> khi các cờ thích hợp được đặt. Độc giả có thể tham khảo thêm bài giảng về DRAM ví dụ như Zeshan Chishti<sup>272</sup>.

<sup>270</sup> [https://en.wikipedia.org/wiki/DDR4\\_SDRAM](https://en.wikipedia.org/wiki/DDR4_SDRAM)

<sup>271</sup> [https://en.wikipedia.org/wiki/Data\\_structure\\_allocation](https://en.wikipedia.org/wiki/Data_structure_allocation)

<sup>272</sup> [http://web.cecs.pdx.edu/~zeshan/ece585\\_lec5.pdf](http://web.cecs.pdx.edu/~zeshan/ece585_lec5.pdf)

Bộ nhớ GPU còn yêu cầu băng thông cao hơn nữa vì chúng có nhiều phần tử xử lý hơn CPU. Nhìn chung có hai phương án tiếp cận đối với vấn đề này. Một cách là mở rộng bus bộ nhớ. Chẳng hạn NVIDIA's RTX 2080 Ti dùng bus có kích thước 352 bit. Điều này cho phép truyền đi lượng thông tin lớn hơn cùng lúc. Một cách khác là sử dụng loại bộ nhớ chuyên biệt có hiệu năng cao cho GPU. Các thiết bị hạng phổ thông, điển hình như dòng RTX và Titan của NVIDIA, dùng các chip **GDDR6**<sup>273</sup> với băng thông tổng hợp hơn 500 GB/s. Một loại bộ nhớ chuyên biệt khác là mô-đun HBM (bộ nhớ băng thông rộng). Chúng dùng phương thức giao tiếp rất khác và kết nối trực tiếp với GPU trên một tấm bán dẫn silic chuyên biệt. Điều này dẫn đến giá thành rất cao và chúng chỉ được sử dụng chủ yếu cho các chip máy chủ cao cấp, ví dụ như dòng GPU NVIDIA Volta V100. Không quá ngạc nhiên, kích thước bộ nhớ GPU nhỏ hơn nhiều so với bộ nhớ CPU do giá thành cao của nó. Nhìn chung các đặc tính hiệu năng của bộ nhớ GPU khá giống bộ nhớ CPU, nhưng nhanh hơn nhiều. Ta có thể bỏ qua các chi tiết sâu hơn trong cuốn sách này, do chúng chỉ quan trọng khi cần điều chỉnh các hạt nhân GPU để đạt thông lượng xử lý cao hơn.

#### 14.4.3 Lưu trữ

Chúng ta đã thấy đặc tính then chốt của RAM chính là *băng thông* và *độ trễ*. Điều này cũng đúng đối với các thiết bị lưu trữ, sự khác biệt chỉ có thể là các đặc tính trên lớn hơn nhiều lần.

**Các ổ cứng** đã được sử dụng hơn nửa thế kỷ. Một cách ngắn gọn, chúng chứa một số đĩa quay với những đầu kim có thể di chuyển để đọc/ghi ở bất cứ rãnh nào. Các ổ đĩa cao cấp có thể lưu trữ lên tới 16 TB trên 9 đĩa. Một trong những lợi ích chính của ổ đĩa cứng là chúng tương đối rẻ. Nhược điểm của chúng là độ trễ tương đối cao khi đọc dữ liệu và hay bị hư hỏng nặng dẫn đến không thể đọc dữ liệu, thậm chí là mất dữ liệu.

Để hiểu về nhược điểm thứ hai, hãy xem xét thực tế rằng ổ cứng quay với tốc độ khoảng 7,200 vòng/phút. Nếu tốc độ này cao hơn, các đĩa sẽ vỡ tan do tác dụng của lực ly tâm. Điều này dẫn đến một nhược điểm lớn khi truy cập vào một khu vực cụ thể trên đĩa: chúng ta cần đợi cho đến khi đĩa quay đúng vị trí (chúng ta có thể di chuyển đầu kim nhưng không được tăng tốc các đĩa). Do đó, có thể mất hơn 8ms cho đến khi truy cập được dữ liệu yêu cầu. Vì thế mà ta hay nói ổ cứng có thể hoạt động ở mức xấp xỉ 100 IOP. 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 không kém (ở mức độ 100-200 MB/s). Rốt cuộc, mỗi đầu đọc một rãnh bit, do đó tốc độ bit chỉ tăng theo tỷ lệ căn bậc hai của mật độ thông tin. Kết quả là các ổ cứng đang nhanh chóng biến thành nơi lưu trữ cấp thấp cho các bộ dữ liệu rất lớn.

**Ổ cứng thẻ rắn (SSD)** sử dụng bộ nhớ Flash để liên tục lưu trữ thông tin. Điều này cho phép truy cập nhanh hơn nhiều vào các bản ghi đã được lưu trữ. SSD hiện đại có thể hoạt động ở mức 100,000 đến 500,000 IOP, tức là nhanh hơn gấp 1000 lần so với ổ cứng HDD. Hơn nữa, băng thông của chúng có thể đạt tới 1-3GB/s nghĩa là nhanh hơn 10 lần so với ổ cứng. Những cải tiến này nghe có vẻ tốt đến mức khó tin. Thực vậy, và SSD cũng đi kèm với một số hạn chế do cách mà chúng được thiết kế.

- Các ổ SSD lưu trữ thông tin theo khối (256 KB trở lên). Ta sẽ phải ghi cả khối cùng một lúc, mất thêm thời gian đáng kể. Do đó việc ghi ngẫu nhiên theo bit trên SSD có hiệu suất rất tệ. Tương tự như vậy, việc ghi dữ liệu nói chung mất thời gian đáng kể vì khối phải được đọc, xóa và sau đó viết lại với thông tin mới. Cho đến nay, bộ điều khiển và firmware của SSD đã phát triển các thuật toán để giảm thiểu vấn đề này. Tuy nhiên tốc độ ghi vẫn có thể chậm hơn nhiều, đặc biệt là đối với SSD QLC (ô bốn cấp). Chìa khóa để cải thiện hiệu suất là đưa các thao tác vào một *hàng đợi* để ưu tiên việc đọc trước và chỉ ghi theo các khối lớn nếu có thể.

<sup>273</sup> [https://en.wikipedia.org/wiki/GDDR6\\_SDRAM](https://en.wikipedia.org/wiki/GDDR6_SDRAM)

- Các ô nhớ trong SSD bị hao mòn tương đối nhanh (thường sau vài nghìn lần ghi). Các thuật toán bảo vệ mức hao mòn có thể phân bổ đều sự xuống cấp trên nhiều ô. Dù vậy, vẫn không nên sử dụng SSD cho các tệp hoán đổi (*swap file*) hoặc cho tập hợp lớn các tệp nhật ký (*log file*).
- 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 việc này, được gọi là NVMe (Bộ nhớ không biến động tăng cường - *Non Volatile Memory enhanced*), có thể sử dụng lên tới 4 làn PCIe. Băng thông có thể lên tới 8GB/s trên PCIe 4.0.

**Lưu trữ đám mây** cung cấp nhiều lựa chọn hiệu suất có thể tùy chỉnh. Nghĩa là, việc chỉ định bộ lưu trữ cho các máy ảo là tùy chỉnh, cả về số lượng và tốc độ, do người dùng quyết định. Chúng tôi khuyên người dùng nên tăng số lượng IOP được cung cấp bất cứ khi nào độ trễ quá cao, ví dụ như trong quá trình huấn luyện với dữ liệu gồm nhiều bản ghi nhỏ.

#### 14.4.4 CPU

Bộ xử lý trung tâm (Central Processing Units - CPU) là trung tâm của mọi máy tính (như ở phần trước, chúng tôi đã mô tả tổng quan về những phần cứng quan trọng cho các mô hình học sâu hiệu quả). CPU gồm một số thành tố quan trọng: lõi xử lý (*core*) với khả năng thực thi mã máy, bus kết nối các lõi (cấu trúc kết nối cụ thể có sự khác biệt lớn giữa các mô hình xử lý, đời chip và nhà sản xuất) và bộ nhớ đệm (*cache*) cho phép truy cập với băng thông cao hơn và độ trễ thấp hơn so với việc đọc từ bộ nhớ chính. Cuối cùng, hầu hết CPU hiện đại chứa những đơn vị xử lý vector để hỗ trợ tính toán đại số tuyến tính và tích chập với tốc độ cao vì chúng khá phổ biến trong xử lý phương tiện và học máy.

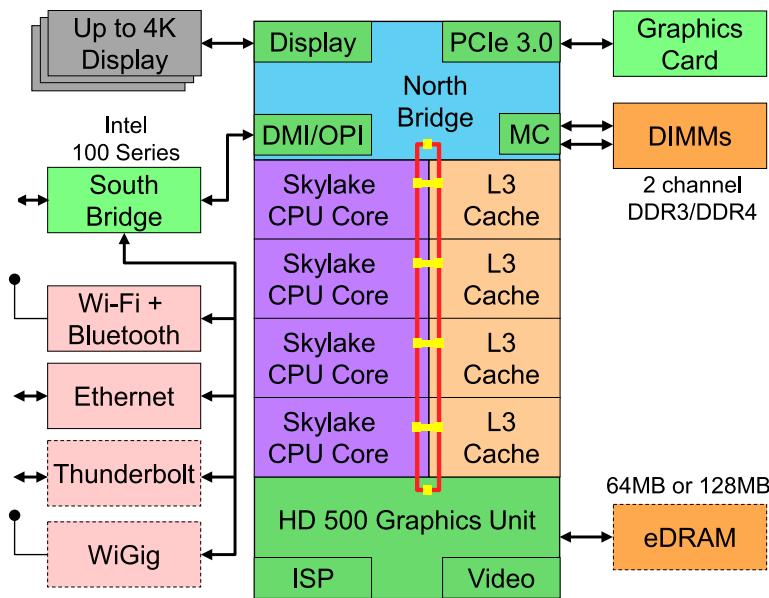


Fig. 14.4.3: CPU lõi tứ của bộ xử lý Intel Skylake

Fig. 14.4.3 minh họa bộ xử lý Intel Skylake với CPU lõi tứ. Nó có một GPU tích hợp, bộ nhớ cache và phương tiện kết nối bốn lõi. Thiết bị ngoại vi (Ethernet, WiFi, Bluetooth, bộ điều khiển SSD, USB, v.v.) là một phần của chipset hoặc được đính kèm trực tiếp (PCIe) với CPU.

#### 14.4.5 Vi kiến trúc (Micro-architecture)

Mỗi nhân xử lý bao gồm các thành phần rất tinh vi. Mặc dù chi tiết khác nhau giữa đời chip và nhà sản xuất, chức năng cơ bản của chúng đã được chuẩn hóa tương đối. Front-end tải các lệnh và dự đoán nhánh nào sẽ được thực hiện (ví dụ: cho luồng điều khiển). Sau đó các lệnh được giải mã từ mã nguồn hợp ngữ (*assembly code*) thành vi lệnh. Mã nguồn hợp ngữ thường chưa phải là mã nguồn cấp thấp nhất mà bộ xử lý thực thi. Thay vào đó, các lệnh phức tạp có thể được giải mã thành một tập hợp các phép tính cấp thấp hơn. Tiếp đó chúng được xử lý bằng một lõi thực thi. Các bộ xử lý đời mới thường có khả năng thực hiện đồng thời nhiều câu lệnh. Ví dụ, lõi ARM Cortex A77 trong Fig. 14.4.4 có thể thực hiện lên đến 8 phép tính cùng một lúc.

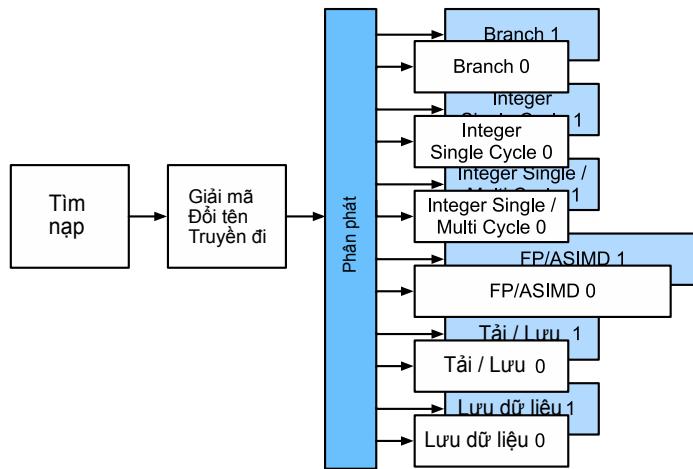


Fig. 14.4.4: Tổng quan về vi kiến trúc ARM Cortex A77

Điều này có nghĩa là các chương trình hiệu quả có thể thực hiện nhiều hơn một lệnh trên một chu kỳ xung nhịp, giả sử rằng chúng có thể được thực hiện một cách độc lập. Không phải tất cả các bộ xử lý đều được tạo ra như nhau. Một số được thiết kế chuyên biệt cho các lệnh về số nguyên, trong khi một số khác được tối ưu hóa cho việc tính toán số thực dấu phẩy động. Để tăng thông lượng, bộ xử lý cũng có thể theo đồng thời nhiều nhánh trong một lệnh rẽ nhánh và sau đó loại bỏ các kết quả của nhánh không được thực hiện. Đây là lý do vì sao đơn vị dự đoán nhánh có vai trò quan trọng (trên front-end), bởi chúng chỉ chọn những nhánh có khả năng cao được rẽ.

#### 14.4.6 Vector hóa (Vectorization)

Học sâu đòi hỏi sức mạnh tính toán cực kỳ lớn. Vì vậy, CPU phù hợp với học máy cần phải thực hiện được nhiều thao tác trong một chu kỳ xung nhịp. Ta có thể đạt được điều này thông qua các đơn vị vector. Trên chip ARM chúng được gọi là NEON, trên x86 thế hệ đơn vị vector mới nhất được gọi là AVX2<sup>274</sup>. Một khía cạnh chung là chúng có thể thực hiện SIMD (đơn lệnh đa dữ liệu - *single instruction multiple data*). Fig. 14.4.5 cho thấy cách cộng 8 số nguyên ngắn trong một chu kỳ xung nhịp trên ARM.

<sup>274</sup> [https://en.wikipedia.org/wiki/Advanced\\_Vector\\_Extensions](https://en.wikipedia.org/wiki/Advanced_Vector_Extensions)

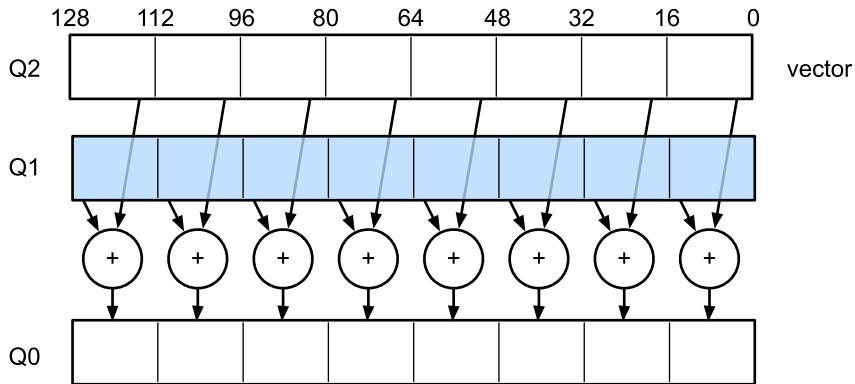


Fig. 14.4.5: Vector hóa NEON 128 bit

Phụ thuộc vào các lựa chọn kiến trúc, các thanh ghi như vậy có thể dài tới 512 bit, cho phép tổ hợp tối đa 64 cặp số. Chẳng hạn, ta có thể nhân hai số và cộng chúng với số thứ ba, cách này còn được biết đến như phép nhân-cộng hợp nhất (*fused multiply-add*). OpenVino<sup>275</sup> của Intel sử dụng thao tác này để đạt được thông lượng đáng kể cho học sâu trên CPU máy chủ. Tuy nhiên, xin lưu ý rằng tốc độ này hoàn toàn không đáng kể so với khả năng của GPU. Ví dụ, RTX 2080 Ti của NVIDIA có 4,352 nhân CUDA, mỗi nhân có khả năng xử lý một phép tính như vậy tại bất cứ thời điểm nào.

#### 14.4.7 Bộ nhớ đệm

Xét tình huống sau: ta có một CPU bình thường với 4 nhân như trong Fig. 14.4.3 trên, hoạt động ở tần số 2 GHz. Thêm nữa, hãy giả sử IPC (*instruction per clock* - số lệnh mỗi xung nhịp) là 1 và mỗi nhân đều đã kích hoạt AVX2 rộng 256 bit. Ngoài ra, giả sử bộ nhớ cần truy cập ít nhất một thanh ghi được sử dụng trong các lệnh AVX2. Điều này có nghĩa CPU xử lý  $4 \times 256$  bit = 1 kbit dữ liệu mỗi chu kỳ xung nhịp. Trừ khi ta có thể truyền  $2 \cdot 10^9 \cdot 128 = 256 \cdot 10^9$  byte đến vi xử lý mỗi giây, các nhân sẽ thiếu dữ liệu để xử lý. Tiếc thay giao diện bộ nhớ của bộ vi xử lý như trên chỉ hỗ trợ tốc độ truyền dữ liệu khoảng 20-40 GB/s, nghĩa là thấp hơn 10 lần. Để khắc phục vấn đề này, ta cần tránh nạp dữ liệu mới từ bộ nhớ ngoài, và tốt hơn hết là lưu trong bộ nhớ cục bộ trên CPU. Đây chính là lúc bộ nhớ đệm trở nên hữu ích (xem bài viết trên Wikipedia<sup>276</sup> này để bắt đầu). Một số tên gọi/khai niệm thường gặp:

- **Thanh ghi** không phải là một bộ phận của bộ nhớ đệm. Chúng hỗ trợ sắp xếp các câu lệnh cho CPU. Nhưng dù sao thanh ghi cũng là một vùng nhớ mà CPU có thể truy cập với tốc độ xung nhịp mà không có độ trễ. Các CPU thường có hàng chục thanh ghi. Việc sử dụng các thanh ghi sao cho hiệu quả hoàn toàn phụ thuộc vào trình biên dịch (hoặc lập trình viên). Ví dụ như trong ngôn ngữ C, ta có thể sử dụng từ khóa register để lưu các biến vào thanh ghi thay vì bộ nhớ.
- Bộ nhớ đệm **L1** là lớp bảo vệ đầu tiên khi nhu cầu băng thông bộ nhớ quá cao. Bộ nhớ đệm L1 rất nhỏ (kích thước điển hình khoảng 32-64 kB) và thường được chia thành bộ nhớ đệm dữ liệu và câu lệnh. Nếu dữ liệu được tìm thấy trong bộ nhớ đệm L1, việc truy cập diễn ra rất nhanh chóng. Nếu không, việc tìm kiếm sẽ tiếp tục theo hệ thống phân cấp bộ nhớ đệm (*cache hierarchy*).
- Bộ nhớ đệm **L2** là điểm dừng tiếp theo. Vùng nhớ này có thể chuyên biệt tùy theo kiến trúc thiết kế và kích thước vi xử lý. Nó có thể chỉ được truy cập từ một lối nhất định hoặc được

<sup>275</sup> <https://01.org/openvinotoolkit>

<sup>276</sup> [https://en.wikipedia.org/wiki/Cache\\_hierarchy](https://en.wikipedia.org/wiki/Cache_hierarchy)

chia sẻ với nhiều lõi khác nhau. Bộ nhớ đệm L2 có kích thước lớn hơn (thường là 256-512 kB mỗi lõi) và chậm hơn L1. Hơn nữa, để truy cập vào dữ liệu trong L2, đầu tiên ta cần kiểm tra để chắc rằng dữ liệu đó không nằm trong L1, việc này làm tăng độ trễ lên một chút.

- Bộ nhớ đệm **L3** được sử dụng chung cho nhiều lõi khác nhau và có thể khá lớn. CPU máy chủ Epyc 3 của AMD có bộ nhớ đệm 256MB cực lớn được phân bổ trên nhiều vi xử lý con (*chiplet*). Thường thì kích thước của L3 nằm trong khoảng 4-8MB.

Việc dự đoán phần tử bộ nhớ nào sẽ cần tiếp theo là một trong những tham số tối ưu chính trong thiết kế vi xử lý. Ví dụ, việc duyệt *xuôi* bộ nhớ được coi là thích hợp do đa số các thuật toán ghi đệm (*caching algorithms*) sẽ cố gắng *đọc về trước* hơn là về sau. Tương tự, việc giữ hành vi truy cập bộ nhớ ở mức cục bộ là một cách tốt để cải thiện hiệu năng. Tăng số lượng bộ nhớ đệm là một con dao hai lưỡi. Một mặt việc này đảm bảo các nhân vi xử lý không bị thiếu dữ liệu. Mặt khác nó tăng kích thước vi xử lý, lấn chiếm phần diện tích mà đáng ra có thể được sử dụng vào việc tăng khả năng xử lý. Xét trường hợp tệ nhất như mô tả trong Fig. 14.4.6. Một địa chỉ bộ nhớ được lưu trữ tại vi xử lý 0 trong khi một luồng của vi xử lý 1 yêu cầu dữ liệu đó. Để có thể lấy dữ liệu, vi xử lý 0 phải dừng công việc đang thực hiện, ghi lại thông tin vào bộ nhớ chính để vi xử lý 1 đọc dữ liệu từ đó. Trong suốt quá trình này, cả hai vi xử lý đều ở trong trạng thái chờ. Một đoạn mã như vậy khả năng cao là sẽ chạy *chậm hơn* trên một hệ đa vi xử lý so với một vi xử lý đơn được lập trình hiệu quả. Đây là một lý do nữa cho việc tại sao thực tế phải giới hạn kích thước bộ nhớ đệm (ngoài việc chiếm diện tích vật lý).

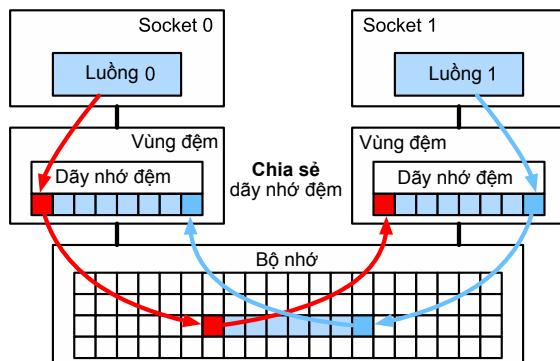


Fig. 14.4.6: Chia sẻ dữ liệu lõi (hình ảnh được sự cho phép của Intel)

#### 14.4.8 GPU và các Thiết bị Tăng tốc khác

Không hề phỏng đại khi nói rằng học sâu có lẽ sẽ không thành công nếu không có GPU. Và cũng nhờ có học sâu mà tài sản của các công ty sản xuất GPU tăng trưởng đáng kể. Sự đồng tiến hóa giữa phần cứng và các thuật toán dẫn tới tình huống mà học sâu trở thành mẫu mô hình thống kê được ưa thích bất kể có hiệu quả hay không. Do đó, ta cần phải hiểu rõ ràng lợi ích mà GPU và các thiết bị tăng tốc khác như TPU ([Jouppi et al., 2017](#)) mang lại.

Ta cần chú ý đến đặc thù thường được sử dụng trong thực tế: thiết bị tăng tốc được tối ưu hoặc cho bước huấn luyện hoặc cho bước suy luận. Đối với bước suy luận, ta chỉ cần tính toán lượt truyền xuôi qua mạng, không cần sử dụng bộ nhớ để lưu dữ liệu trung gian ở bước lan truyền ngược. Hơn nữa, ta có thể không cần đến phép tính quá chính xác (thường thì FP16 hoặc INT8 là đủ). Mặt khác trong quá trình huấn luyện, tất cả kết quả trung gian đều cần phải lưu lại để tính gradient. Hơn nữa, việc tích luỹ gradient yêu cầu độ chính xác cao hơn nhằm tránh lỗi tràn số trên hoặc dưới, do đó bước huấn luyện yêu cầu tối thiểu độ chính xác FP16 (hoặc độ chính xác hỗn hợp khi kết hợp với FP32). Tất cả các yếu tố trên đòi hỏi bộ nhớ nhanh hơn và lớn hơn (HBM2 hoặc GDDR6).

và nhiều khả năng xử lý hơn. Ví dụ, GPU Turing<sup>277</sup> T4 của NVIDIA được tối ưu cho bước suy luận trong khi GPU V100 phù hợp cho quá trình huấn luyện.

Xem lại Fig. 14.4.5. Việc thêm các đơn vị vector vào lõi vi xử lý cho phép ta tăng đáng kể thông lượng xử lý (ở ví dụ trong hình ta có thể thực hiện 16 thao tác cùng lúc). Chuyện gì sẽ xảy ra nếu ta không chỉ tối ưu cho phép tính giữa các vector mà còn tối ưu cho các ma trận? Chiến lược này dẫn tới sự ra đời của Lõi Tensor (chi tiết sẽ được thảo luận sau đây). Thứ hai, nếu tăng số lượng lõi thì sao? Nói tóm lại, hai chiến lược trên tóm tắt việc quyết định thiết kế của GPU. Fig. 14.4.7 mô tả tổng quan một khối xử lý đơn giản, bao gồm 16 đơn vị số nguyên và 16 đơn vị dấu phẩy động. Thêm vào đó, hai Lõi Tensor xử lý một tập nhỏ các thao thác liên quan đến học sâu được thêm vào. Mỗi Hệ vi xử lý Luồng (Streaming Multiprocessor - SM) bao gồm bốn khối như vậy.

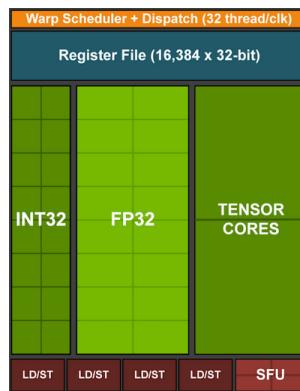


Fig. 14.4.7: Khối Xử lý Turing của NVIDIA

12 hệ vi xử lý luồng sau đó được nhóm vào một cụm xử lý đồ họa tạo nên vi xử lý cao cấp TU102. Số lượng kênh bộ nhớ phong phú và bộ nhớ đệm L2 được bổ sung vào cấu trúc. Thông tin chi tiết được mô tả trong Fig. 14.4.8. Một trong những lý do để thiết kế một thiết bị như vậy là từng khối riêng biệt có thể được thêm vào hoặc bỏ đi tùy theo nhu cầu để có thể tạo thành một vi xử lý nhỏ gọn và giải quyết một số vấn đề phát sinh (các mô-đun lõi có thể không được kích hoạt). May mắn thay, các nhà nghiên cứu học sâu bình thường không cần lập trình cho các thiết bị này do đã có các lớp mã nguồn framework CUDA ở tầng thấp. Cụ thể, có thể có nhiều hơn một chương trình được thực thi đồng thời trên GPU, với điều kiện là còn đủ tài nguyên. Tuy nhiên ta cũng cần để ý đến giới hạn của các thiết bị nhằm tránh việc lựa chọn mô hình quá lớn so với bộ nhớ của thiết bị.

<sup>277</sup> <https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/>



Fig. 14.4.8: Kiến trúc Turing của NVIDIA (hình ảnh được sự cho phép của NVIDIA)

Khía cạnh cuối cùng đáng để bàn luận chi tiết là Lõi Tensor (*TensorCore*). Đây là một ví dụ của xu hướng gần đây là sử dụng thêm nhiều mạch đã được tối ưu để tăng hiệu năng cho học sâu. Ví dụ, TPU có thêm một mảng tâm thu (*systolic array*) (Kung, 1988) để tăng tốc độ nhân ma trận. Thiết kế của TPU chỉ hỗ trợ một số lượng rất ít các phép tính kích thước lớn (thế hệ TPU đầu tiên hỗ trợ một phép tính). Lõi Tensor thì ngược lại, được tối ưu cho các phép tính kích thước nhỏ cho các ma trận kích thước 4x4 đến 16x16, tuỳ vào độ chính xác số học. Fig. 14.4.9 mô tả tổng quan quá trình tối ưu.



Fig. 14.4.9: Lõi Tensor của NVIDIA trong Turing (hình ảnh được sự cho phép của NVIDIA)

Đương nhiên khi tối ưu cho quá trình tính toán, ta buộc phải có một số đánh đổi nhất định. Một trong số đó là GPU không xử lý tốt dữ liệu ngắt quãng hoặc thưa. Trừ một số ngoại lệ đáng chú ý,

ví dụ như [Gunrock](#)<sup>278</sup> (Wang et al., 2016), việc truy cập vector và ma trận thưa không phù hợp với các thao tác đọc theo cụm (*burst read*) với băng thông cao của GPU. Đạt được cả hai mục tiêu là một lĩnh vực đang được đẩy mạnh nghiên cứu. Ví dụ, tham khảo [DGL](#)<sup>279</sup>, một thư viện được điều chỉnh cho phù hợp với học sâu trên đồ thị.

#### 14.4.9 Mạng máy tính và Bus

Mỗi khi một thiết bị đơn không đủ cho quá trình tối ưu, ta cần chuyển dữ liệu đến và đi khỏi nó để đồng bộ hóa quá trình xử lý. Đây chính là lúc mà mạng máy tính và bus trở nên hữu dụng. Ta có một vài tham số thiết kế gồm: băng thông, chi phí, khoảng cách và tính linh hoạt. Tuy ta cũng có Wifi với phạm vi hoạt động tốt, dễ dàng để sử dụng (dù sao cũng là không dây), rẻ nhưng lại có băng thông không quá tốt và độ trễ lớn. Sẽ không có bất cứ nhà nghiên cứu học máy tính táo nào lại nghĩ đến việc sử dụng Wifi để xây dựng một cụm máy chủ. Sau đây, ta sẽ chỉ tập trung vào các cách kết nối phù hợp cho học sâu.

- **PCIe** là một bus riêng chỉ phục vụ cho kết nối điểm – điểm với băng thông trên mỗi làn rất lớn (lên đến 16 GB/s trên PCIe 4.0). Độ trễ thường có giá trị cỡ vài micro giây (5 µs). Kết nối PCIe khá quan trọng. Vì xử lý chỉ có một số lượng làn PCIe nhất định: EPYC 3 của AMD có 128 làn, Xeon của Intel lên đến 48 làn cho mỗi chip; trên CPU dùng cho máy tính để bàn, số lượng này lần lượt là 20 (với Ryzen 9) và 16 (với Core i9). Do GPU thường có 16 luồng nên số lượng GPU có thể kết nối với CPU bị giới hạn tại băng thông tối đa. Xét cho cùng, chúng cần chia sẻ liên kết với các thiết bị ngoại vi khác như bộ nhớ và cổng Ethernet. Giống như việc truy cập RAM, việc truyền lượng lớn dữ liệu thường được ưa chuộng hơn nhằm giảm tổng chi phí theo gói tin.
- **Ethernet** là cách phổ biến nhất để kết nối máy tính với nhau. Dù nó chậm hơn đáng kể so với PCIe, nó rất rẻ và dễ cài đặt, bao phủ khoảng cách lớn hơn nhiều. Băng thông đặc trưng đối với máy chủ cấp thấp là 1 GBit/s. Các thiết bị cao cấp hơn (ví dụ như [máy chủ loại C5](#)<sup>280</sup> trên AWS) cung cấp băng thông từ 10 đến 100 GBit/s. Cũng như các trường hợp trên, việc truyền dữ liệu có tổng chi phí đáng kể. Chú ý rằng ta hầu như không bao giờ sử dụng trực tiếp Ethernet thuần mà sử dụng một giao thức được thực thi ở tầng trên của kết nối vật lý (ví dụ như UDP hay TCP/IP). Việc này làm tăng tổng chi phí. Giống như PCIe, Ethernet được thiết kế để kết nối hai thiết bị, ví dụ như máy tính với một bộ chuyển mạch (*switch*).
- **Bộ chuyển mạch** cho phép ta kết nối nhiều thiết bị theo cách mà bất cứ cặp thiết bị nào cũng có thể (thường là với băng thông tối đa) thực hiện kết nối điểm – điểm cùng lúc. Ví dụ, bộ chuyển mạch Ethernet có thể kết nối 40 máy chủ với băng thông xuyên vùng (*cross-sectional bandwidth*) cao. Chú ý rằng bộ chuyển mạch không phải chỉ có trong mạng máy tính truyền thống. Ngay cả làn PCIe cũng có thể [chuyển mạch](#)<sup>281</sup>. Điều này xảy ra khi kết nối một lượng lớn GPU tới vi xử lý chính, như với trường hợp [máy chủ loại P2](#)<sup>282</sup>.
- **NVLink** là một phương pháp thay thế PCIe khi ta cần kết nối với băng thông rất lớn. NVLink cung cấp tốc độ truyền dữ liệu lên đến 300 Gbit/s mỗi đường dẫn (*link*). GPU máy chủ (Volta V100) có 6 đường dẫn, trong khi GPU thông dụng (RTX 2080 Ti) chỉ có một đường dẫn, hoạt động ở tốc độ thấp 100 Gbit/s. Vì vậy, chúng tôi gợi ý sử dụng [NCCL](#)<sup>283</sup> để có thể đạt được tốc độ truyền dữ liệu cao giữa các GPU.

<sup>278</sup> <https://github.com/gunrock/gunrock>

<sup>279</sup> <http://dgl.ai>

<sup>280</sup> <https://aws.amazon.com/ec2/instance-types/c5/>

<sup>281</sup> <https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches>

<sup>282</sup> <https://aws.amazon.com/ec2/instance-types/p2/>

<sup>283</sup> <https://github.com/NVIDIA/ncc>

#### 14.4.10 Tóm tắt

- Các thiết bị đều có chi phí phụ trợ trên mỗi hành động. Do đó ta nên nhắm tới việc di chuyển ít lần các lượng dữ liệu lớn thay vì di chuyển nhiều lần các lượng dữ liệu nhỏ. Điều này đúng với RAM, SSD, các thiết bị mạng và GPU.
- Vector hóa rất quan trọng để tăng hiệu năng. Hãy đảm bảo bạn hiểu các điểm mạnh đặc thù của thiết bị tăng tốc mình đang có. Ví dụ, một vài CPU Intel Xeon thực hiện cực kỳ hiệu quả phép toán với dữ liệu kiểu INT8, GPU NVIDIA Volta rất phù hợp với các phép toán với ma trận dữ liệu kiểu FP16; còn NVIDIA Turing chạy tốt cho cả các phép toán với dữ liệu kiểu FP16, INT8, INT4.
- Hiện tượng tràn số trên do kiểu dữ liệu không đủ số bit để biểu diễn giá trị có thể là một vấn đề khi huấn luyện (và cả khi suy luận, dù ít nghiêm trọng hơn).
- Việc cùng dữ liệu nhưng có nhiều địa chỉ (*aliasing*) có thể làm giảm đáng kể hiệu năng. Ví dụ, việc sắp xếp dữ liệu trong bộ nhớ (*memory alignment*) trên CPU 64 bit nên được thực hiện theo từng khối 64 bit. Trên GPU, tốt hơn là nên giữ kích thước tích chập đồng bộ, với TensorCores chẳng hạn.
- Sử dụng thuật toán phù hợp với phần cứng (về mức chiếm dụng bộ nhớ, băng thông, v.v.). Thời gian thực thi có thể giảm hàng trăm ngàn lần khi tất cả tham số đều được chứa trong bộ đệm.
- Chúng tôi khuyến khích bạn đọc tính toán trước hiệu năng của một thuật toán mới trước khi kiểm tra bằng thực nghiệm. Sự khác biệt lên tới hàng chục lần hoặc hơn là dấu hiệu cần quan tâm.
- Sử dụng các công cụ phân tích hiệu năng (*profiler*) để tìm điểm nghẽn cổ chai của hệ thống.
- Phần cứng sử dụng cho huấn luyện và suy luận có các cấu hình hiệu quả khác nhau để cân đối giá tiền và hiệu năng.

#### 14.4.11 Độ trễ

Các thông tin trong `table_latency_numbers` và `table_latency_numbers_tesla` được Eliot Eshelman<sup>284</sup> duy trì cập nhật trên GitHub Gist<sup>285</sup>.

: Các độ trễ thường gặp.

Table 14.4.1: label:`table_latency_numbers`

| Hoạt động  | Thời gian | Chú thích      |
|--|-----------|----------------|
| Truy xuất bộ đệm L1  | 1.5 ns    | 4 chu kỳ       |
| Cộng, nhân, cộng kết hợp nhân ( <i>FMA</i> ) số thực dấu phẩy động | 1.5 ns    | 4 chu kỳ       |
| Truy xuất bộ đệm L2  | 5 ns      | 12 ~ 17 chu kỳ |
| Rẽ nhánh sai   | 6 ns      | 15 ~ 20 chu kỳ |
| Truy xuất bộ đệm L3 (không chia sẻ)                                | 16 ns     | 42 chu kỳ      |
| Truy xuất bộ đệm L3 (chia sẻ với nhân khác)                        | 25 ns     | 65 chu kỳ      |
| Khóa/mở đèn báo lập trình ( <i>mutex</i> )                         | 25 ns     |                |
| Truy xuất bộ đệm L3 (được nhân khác thay đổi)                      | 29 ns     | 75 chu kỳ      |

continues

<sup>284</sup> <https://gist.github.com/eshelman>

<sup>285</sup> <https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646>

Table 14.4.1 – continued from previous page

| Hoạt động   | Thời gian | Chú thích                       |
|---|-----------|---------------------------------|
| Truy xuất bộ đệm L3 (tại CPU socket từ xa)            | 40 ns     | 100 ~ 300 chu kỳ (40 ~ 116 ns)  |
| QPI hop đến CPU khác (cho mỗi hop)                    | 40 ns     |                                 |
| Truy xuất 64MB (CPU cục bộ)                           | 46 ns     | TinyMemBench trên Broadwell E5  |
| Truy xuất 64MB (CPU từ xa)                            | 70 ns     | TinyMemBench trên Broadwell E5  |
| Truy xuất 256MB (CPU cục bộ)                          | 75 ns     | TinyMemBench trên Broadwell E5  |
| Ghi ngẫu nhiên vào Intel Optane                       | 94 ns     | UCSD Non-Volatile Systems Lab   |
| Truy xuất 256MB (CPU từ xa)                           | 120 ns    | TinyMemBench trên Broadwell E5  |
| Đọc ngẫu nhiên từ Intel Optane                        | 305 ns    | UCSD Non-Volatile Systems Lab   |
| Truyền 4KB trên sợi HPC 100 Gbps                      | 1 μs      | MVAPICH2 trên Intel Omni-Path   |
| Nén 1KB với Google Snappy                             | 3 μs      |                                 |
| Truyền 4KB trên cáp mạng 10 Gbps                      | 10 μs     |                                 |
| Ghi ngẫu nhiên 4KB vào SSD NVMe                       | 30 μs     | DC P3608 SSD NVMe (QOS 99% kh)  |
| Truyền 1MB từ/đến NVLink GPU                          | 30 μs     | ~33GB/s trên NVIDIA 40GB NVLink |
| Truyền 1MB từ/đến PCI-E GPU                           | 80 μs     | ~12GB/s trên PCIe 3.0 x16 link  |
| Đọc ngẫu nhiên 4KB từ SSD NVMe                        | 120 μs    | DC P3608 SSD NVMe (QOS 99%)     |
| Đọc tuần tự 1MB từ SSD NVMe                           | 208 μs    | ~4.8GB/s DC P3608 SSD NVMe      |
| Ghi ngẫu nhiên 4KB vào SSD SATA                       | 500 μs    | DC S3510 SSD SATA (QOS 99.9%)   |
| Đọc ngẫu nhiên 4KB từ SSD SATA                        | 500 μs    | DC S3510 SSD SATA (QOS 99.9%)   |
| Truyền 2 chiều trong cùng trung tâm dữ liệu           | 500 μs    | Ping một chiều ~250μs           |
| Đọc tuần tự 1MB từ SSD SATA                           | 2 ms      | ~550MB/s DC S3510 SSD SATA      |
| Đọc tuần tự 1MB từ ổ đĩa                              | 5 ms      | ~200MB/s server HDD             |
| Truy cập ngẫu nhiên ổ đĩa (tìm + xoay)                | 10 ms     |                                 |
| Gửi gói dữ liệu từ California -> Hà Lan -> California | 150 ms    |                                 |

: Độ trễ của GPU NVIDIA Tesla.

Table 14.4.2: label:table\_latency\_numbers\_tesla

| Hoạt động                        | Thời gian | Chú thích                                    |
|----------------------------------|-----------|--|
| Truy cập bộ nhớ chung của GPU    | 30 ns     | 30~90 chu kỳ (tính cả xung đột của các bank) |
| Truy cập bộ nhớ toàn cục của GPU | 200 ns    | 200~800 chu kỳ                               |
| Khởi chạy nhân CUDA trên GPU     | 10 μs     | CPU host ra lệnh cho GPU khởi chạy nhân      |
| Truyền 1MB từ/đến GPU NVLink     | 30 μs     | ~33GB/s trên NVIDIA NVLink 40GB              |
| Truyền 1MB từ/đến GPU PCI-E      | 80 μs     | ~12GB/s trên PCI-Express link x16            |

#### 14.4.12 Bài tập

- Viết đoạn mã C để so sánh tốc độ khi truy cập bộ nhớ được sắp xếp theo khối (*aligned memory*) với khi truy cập bộ nhớ không được sắp xếp như vậy (một cách tương đối so với bộ nhớ ngoài). **Gợi ý:** hãy loại bỏ hiệu ứng của bộ nhớ đệm.
- So sánh tốc độ khi truy cập bộ nhớ tuần tự với khi truy cập theo sai bước cho trước.
- Làm thế nào để đo kích thước bộ nhớ đệm trên CPU?
- Bạn sẽ sắp xếp dữ liệu trên nhiều bộ nhớ như thế nào để có băng thông tối đa? Sắp xếp như thế nào nếu bạn có nhiều luồng nhỉ?
- Tốc độ quay của một ổ cứng HDD dùng cho công nghiệp là 10,000 rpm. Thời gian tối thiểu

mà HDD đó cần (trong trường hợp tệ nhất) trước khi có thể đọc dữ liệu là bao nhiêu (có thể giả sử các đầu đọc ổ đĩa di chuyển tức thời)?

6. Giả sử nhà sản xuất HDD tăng sức chứa bộ nhớ từ 1 Tbit mỗi inch vuông lên 5 Tbit mỗi inch vuông. Có thể lưu bao nhiêu dữ liệu trên một đĩa từ của một HDD 2.5"? Có sự khác biệt nào giữa track trong và track ngoài không?
7. Một máy chủ loại P2 trên AWS có 16 GPU K80 Kepler. Sử dụng lệnh `lspci` trên một máy p2.16xlarge và một máy p2.8xlarge để hiểu cách các GPU được kết nối với các CPU. **Gợi ý:** để ý đến chip cầu nối PLX cho chuẩn kết nối PCI.
8. Chuyển từ kiểu dữ liệu 8 bit sang 16 bit cần lượng silicon gấp 4 lần. Tại sao? Tại sao NVIDIA thêm các phép toán cho kiểu dữ liệu INT4 vào GPU Turing?
9. Có 6 đường truyền tốc độ cao giữa các GPU (như GPU Volta V100 chẳng hạn), bạn sẽ kết nối 8 GPU đó như thế nào? Tham khảo cách kết nối cho máy chủ p3.16xlarge trên AWS.
10. Đọc xuôi bộ nhớ nhanh gấp bao nhiêu lần đọc ngược? Sự chênh lệch này có khác nhau giữa các nhà sản xuất máy tính và CPU không? Tại sao? Thí nghiệm với mã nguồn C.
11. Bạn có thể đo kích thước bộ nhớ đệm trên ổ đĩa của mình không? Bộ nhớ đệm trên HDD là gì? SSD có cần bộ nhớ đệm không?
12. Chi phí bộ nhớ phụ trợ khi gửi một gói dữ liệu qua cáp mạng (*Ethernet*) là bao nhiêu. So sánh các giao thức UDP và TCP/IP.
13. Truy cập Bộ nhớ Trực tiếp (*Direct Memory Access*) cho phép các thiết bị khác ngoài CPU ghi (và đọc) trực tiếp vào (từ) bộ nhớ. Tại sao đây là một ý tưởng hay?
14. Nhìn vào thông số hiệu năng của GPU Turing T4. Tại sao hiệu năng chỉ tăng gấp đôi khi chuyển từ phép toán với kiểu dữ liệu FP16 sang INT8 và INT4?
15. Thời gian truyền một gói dữ liệu hai chiều giữa San Francisco và Amsterdam là bao nhiêu? **Gợi ý:** giả sử khoảng cách giữa 2 thành phố là 10,000km.

#### 14.4.13 Thảo luận

- Tiếng Anh<sup>286</sup>
- Tiếng Việt<sup>287</sup>

#### 14.4.14 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long

<sup>286</sup> <https://discuss.d2l.ai/t/363>

<sup>287</sup> <https://forum.machinelearningcoban.com/c/d2l>

- Trần Yến Thy
- Nguyễn Thanh Hòa
- Đỗ Trường Giang
- Phạm Hồng Vinh

## 14.5 Huấn luyện đa GPU

Đến nay ta đã thảo luận về cách huấn luyện mô hình trên CPU và GPU một cách hiệu quả. Trong Section 14.3, ta biết được cách mà các framework học sâu như MXNet (và TensorFlow) thực hiện song song hóa việc tính toán và giao tiếp giữa các thiết bị một cách tự động. Cuối cùng, Section 7.6 đã trình bày cách liệt kê toàn bộ các GPU có trong máy bằng lệnh `nvidia-smi`. Thứ mà ta chưa thảo luận là cách song song hóa quá trình huấn luyện mô hình học sâu. (Ta bỏ qua việc *dự đoán* trên nhiều GPU vì nó ít khi được sử dụng và là một chủ đề nâng cao nằm ngoài phạm vi của cuốn sách này.) Chúng ta mới chỉ ngầm hiểu rằng bằng cách nào đó dữ liệu có thể được chia ra cho nhiều thiết bị khác nhau. Phần này sẽ bổ sung những chi tiết còn thiếu ấy và mô tả cách huấn luyện song song một mạng học sâu từ đầu. Chi tiết về cách tận dụng các tính năng của Gluon sẽ nằm ở Section 14.6. Vì vậy, chúng tôi xin giả định rằng độc giả đã quen với thuật toán SGD theo minibatch được mô tả ở Section 13.9.

### 14.5.1 Chia nhỏ Vấn đề

Hãy bắt đầu bằng một bài toán thị giác máy tính đơn giản cùng một kiến trúc mạng lâu đời chưa vài tầng tích chập, tầng gộp và có thể thêm vài tầng dày đặc ở cuối. Như vậy, mạng này sẽ trông khá tương tự như LeNet (LeCun et al., 1998) hoặc AlexNet (Krizhevsky et al., 2012). Với nhiều GPU (máy chủ để bàn thường có 2, máy chủ g4dn.12xlarge thì có 4, AWS p3.16xlarge có 8, hoặc là 16 trên p2.16xlarge), ta muốn phân chia việc huấn luyện sao cho vừa tăng tốc độ lại vừa tận dụng được các thiết kế đơn giản và tái tạo được. Sau cùng, việc sử dụng nhiều GPU là để tăng cả *bộ nhớ* và *năng lực tính toán*. Nói ngắn gọn, với một minibatch dữ liệu huấn luyện, ta có một vài phương án phân chia khác nhau.

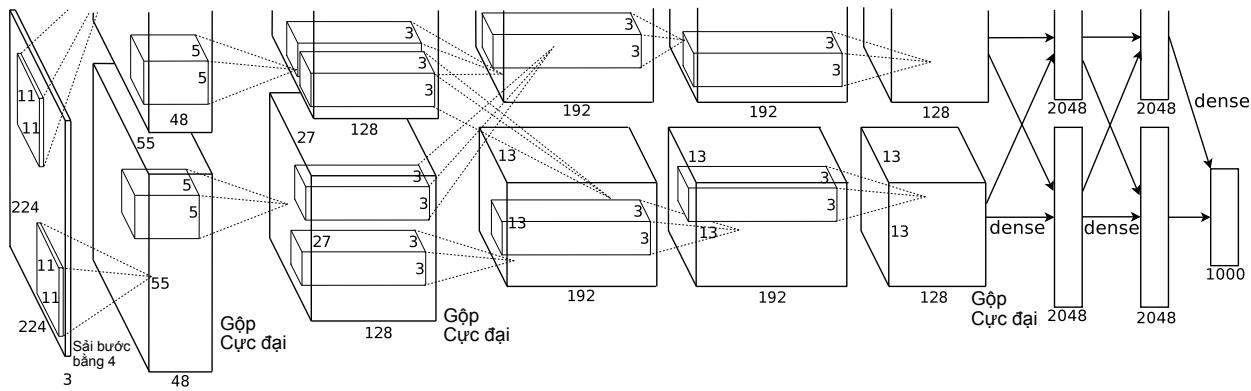


Fig. 14.5.1: Song song hóa mô hình trong thiết kế AlexNet gốc do giới hạn bộ nhớ trên GPU.

- Chúng ta có thể phân chia các tầng mạng trên nhiều GPU. Cụ thể, mỗi GPU sẽ nhận một luồng dữ liệu đưa vào từ một tầng xác định, truyền dữ liệu qua một số tầng kế tiếp nhau rồi gửi dữ liệu tới GPU kế tiếp.

- Điều này cho phép ta xử lý dữ liệu với các mạng lớn hơn, điều nằm ngoài khả năng khi chỉ sử dụng một GPU.
  - Bộ nhớ bị chiếm dụng trên mỗi GPU có thể được kiểm soát dễ dàng (mỗi GPU sẽ chỉ chiếm một phần tổng dung lượng bộ nhớ cấp phát cho cả mạng).
  - Giao tiếp giữa các tầng (cũng như giữa các GPU) đòi hỏi tính đồng bộ chặt chẽ. Điều này có thể sẽ rất khó, đặc biệt nếu khối lượng tính toán không được phân chia hợp lý cho các tầng. Vấn đề sẽ trở nên nghiêm trọng với một số lượng lớn GPU.
  - Giao tiếp giữa các tầng yêu cầu một lượng lớn các thao tác truyền dữ liệu (các hàm kích hoạt, các gradient). Điều này có thể vượt quá mức băng thông các bus của GPU.
  - Các phép tính tuần tự nhưng nặng về mặt tính toán lại không hề dễ phân chia. (Mirhoseini et al., 2017) là nỗ lực tốt nhất để giải quyết vấn đề này. Nó vẫn còn là một vấn đề khó và chưa rõ ràng liệu có thể đạt được khả năng mở rộng tốt (tăng theo tuyến tính) cho các bài toán không quá đơn giản không. Chúng tôi không khuyến khích cách làm này trừ phi có một framework xuất sắc hay một hệ điều hành hỗ trợ cho việc xâu chuỗi nhiều GPU lại với nhau.
- Chúng ta có thể phân chia công việc của các tầng đơn lẻ. Chẳng hạn, thay vì tính toán 64 kênh trên một GPU, ta có thể chia công việc này cho 4 GPU, mỗi GPU sẽ sinh dữ liệu cho 16 kênh. Tương tự, với một tầng kết nối dày đặc ta có thể chia nhỏ số nơ-ron đầu ra. Fig. 14.5.1 mô tả thiết kế kiểu này. Hình này được trích từ (Krizhevsky et al., 2012), khi chiến lược này được sử dụng để làm việc với nhiều GPU có dung lượng bộ nhớ rất nhỏ (2 GB ở thời điểm đó).
- Điều này cho phép việc điều chỉnh kích thước tính toán tốt, với điều kiện là số kênh (hoặc số nơ-ron) không quá nhỏ.
  - Dùng nhiều GPU có thể xử lý các mạng ngày một lớn hơn vì dung lượng bộ nhớ khả dụng cũng tăng tuyến tính.
  - Chúng ta cần một lượng *rất lớn* các phép toán đồng bộ / lớp chặn vì mỗi tầng phụ thuộc vào các kết quả từ tất cả các tầng khác.
  - Lượng dữ liệu cần được truyền thậm chí có thể lớn hơn khi phân phối các tầng giữa các GPU. Chúng tôi không khuyến khích cách tiếp cận này do tính phức tạp và chi phí băng thông của nó.
- Cuối cùng, ta có thể phân chia dữ liệu cho nhiều GPU. Cách này cho phép tất cả GPU thực hiện cùng một công việc, chỉ là với các dữ liệu khác nhau. Các gradient được tổng hợp lại trên các GPU sau mỗi minibatch.
- Đây là phương pháp đơn giản nhất và có thể sử dụng cho bất cứ tình huống nào.
  - Gắn thêm nhiều GPU không cho phép chúng ta huấn luyện mô hình lớn hơn.
  - Chúng ta chỉ cần đồng bộ hóa sau mỗi minibatch. Dù vậy, ta vẫn nên bắt đầu thực hiện trao đổi các gradient đã tính xong kể cả khi việc tính các gradient khác vẫn chưa được hoàn thiện.
  - Số lượng GPU lớn dẫn tới kích thước minibatch rất lớn, do đó giảm hiệu quả huấn luyện.

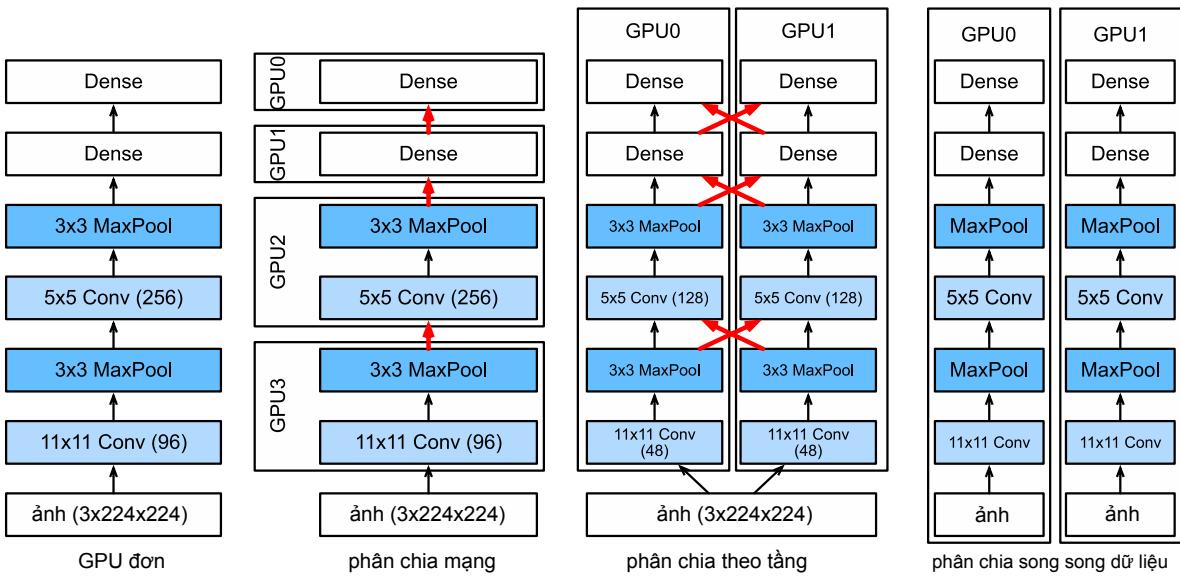


Fig. 14.5.2: Song song hóa trên nhiều GPU. Từ trái sang phải - bài toán ban đầu, phân tách mạng, phân tách tầng, song song hóa dữ liệu

Nhìn chung việc song song hóa dữ liệu là cách thuận tiện nhất, với điều kiện là ta sở hữu các GPU với bộ nhớ đủ lớn. Xem thêm (Li et al., 2014) để biết chi tiết cách phân chia cho việc huấn luyện phân tán. Bộ nhớ GPU từng là một vấn đề trong những ngày đầu của học sâu. Đến thời điểm này thì hầu hết các vấn đề đã được giải quyết trừ một số trường hợp rất ít gặp. Ở phần kế tiếp, chúng ta sẽ tập trung vào việc song song hóa dữ liệu.

### 14.5.2 Song song hóa Dữ liệu

Giả sử ta có một máy tính có  $k$  GPU. Với một mô hình cần được huấn luyện, mỗi GPU duy trì một tập đầy đủ các tham số mô hình độc lập với nhau. Việc huấn luyện diễn ra như sau (xem Fig. 14.5.3 để rõ hơn về việc huấn luyện song song với hai GPU):

- Ở bất cứ vòng huấn luyện nào, với một tập minibatch ngẫu nhiên cho trước, ta chia đều các mẫu từ batch ban đầu này thành  $k$  phần rồi phân bổ cho các GPU.
- Mỗi GPU sẽ tính mất mát và gradient của các tham số mô hình dựa trên tập minibatch con mà nó được cấp và các tham số mô hình nó lưu trữ.
- Các gradient cục bộ từ  $k$  GPU được gom lại để thu được gradient ngẫu nhiên cho minibatch hiện tại.
- Gradient tổng hợp này được phân phối trở lại cho các GPU.
- Mỗi GPU dùng gradient ngẫu nhiên của minibatch này để cập nhật một tập đầy đủ các tham số mô hình mà nó lưu trữ.

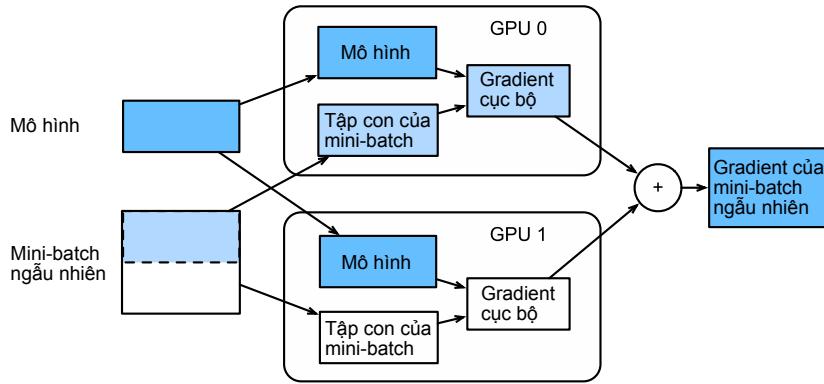


Fig. 14.5.3: Tính toán gradient ngẫu nhiên trên tập minibatch dùng song song hóa dữ liệu với hai GPU.

Fig. 14.5.2 so sánh các cách song song hóa khác nhau trên nhiều GPU. Lưu ý rằng trong thực tế ta cần *tăng* kích thước minibatch lên  $k$  lần khi huấn luyện trên  $k$  GPU để mỗi GPU có cùng khối lượng công việc cần thực hiện như khi ta huấn luyện trên một GPU đơn lẻ. Trên một server có 16 GPU có thể tăng kích thước minibatch một cách đáng kể và ta cũng có thể sẽ phải tăng tốc độ học một cách tương ứng. Chú ý rằng Section 9.5 cũng cần được điều chỉnh lại (ví dụ, ta có thể sử dụng các hệ số chuẩn hóa theo batch riêng cho mỗi GPU). Trong phần tiếp theo ta sẽ dùng Section 8.6 như một mạng thử nghiệm để minh họa việc huấn luyện đa GPU. Như mọi khi, ta bắt đầu bằng cách nạp các gói thư viện và mô-đun liên quan.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
npx.set_np()
```

### 14.5.3 Ví dụ Đơn giản

Ta lập trình từ đầu LeNet trong Section 8.6 để minh họa chi tiết cách trao đổi và đồng bộ tham số.

```
# Initialize model parameters
scale = 0.01
W1 = np.random.normal(scale=scale, size=(20, 1, 3, 3))
b1 = np.zeros(20)
W2 = np.random.normal(scale=scale, size=(50, 20, 5, 5))
b2 = np.zeros(50)
W3 = np.random.normal(scale=scale, size=(800, 128))
b3 = np.zeros(128)
W4 = np.random.normal(scale=scale, size=(128, 10))
b4 = np.zeros(10)
params = [W1, b1, W2, b2, W3, b3, W4, b4]

# Define the model
def lenet(X, params):
    h1_conv = npx.convolution(data=X, weight=params[0], bias=params[1],
                              kernel=(3, 3), num_filter=20)
    h1_activation = npx.relu(h1_conv)
    h1 = npx.pooling(data=h1_activation, pool_type='avg', kernel=(2, 2),
                     stride=(2, 2))
```

(continues on next page)

```

h2_conv = npx.convolution(data=h1, weight=params[2], bias=params[3],
                           kernel=(5, 5), num_filter=50)
h2_activation = npx.relu(h2_conv)
h2 = npx.pooling(data=h2_activation, pool_type='avg', kernel=(2, 2),
                  stride=(2, 2))
h2 = h2.reshape(h2.shape[0], -1)
h3_linear = np.dot(h2, params[4]) + params[5]
h3 = npx.relu(h3_linear)
y_hat = np.dot(h3, params[6]) + params[7]
return y_hat

# Cross-entropy loss function
loss = gluon.loss.SoftmaxCrossEntropyLoss()

```

#### 14.5.4 Đồng bộ Dữ liệu

Để huấn luyện hiệu quả trên nhiều GPU, ta cần hai thao tác cơ bản: thứ nhất là phân phối danh sách tham số đến nhiều GPU và gắn gradient, được định nghĩa trong hàm `get_params` dưới đây. Nếu không có các tham số, ta không thể đánh giá mạng trên GPU. Thứ hai, ta cần tính tổng giá trị các tham số trên nhiều thiết bị, khai báo ở hàm `allreduce`.

```

def get_params(params, device):
    new_params = [p.copyto(device) for p in params]
    for p in new_params:
        p.attach_grad()
    return new_params

```

Hãy thử sao chép các tham số mô hình của LeNet tới `gpu(0)`.

```

new_params = get_params(params, d2l.try_gpu(0))
print('b1 weight:', new_params[1])
print('b1 grad:', new_params[1].grad)

```

Vì chưa thực hiện tính toán nào, gradient ứng với hệ số điều chỉnh vẫn mang giá trị 0. Bây giờ giả sử ta có các vector được phân phối trên nhiều GPU. Hàm `allreduce` dưới đây cộng các vector đó và truyền kết quả về tất cả GPU. Chú ý, để hàm này hoạt động, ta cần sao chép dữ liệu đến GPU đang cộng dồn kết quả.

```

def allreduce(data):
    for i in range(1, len(data)):
        data[0][:] += data[i].copyto(data[0].ctx)
    for i in range(1, len(data)):
        data[0].copyto(data[i])

```

Hãy kiểm tra bằng cách tạo các vector với giá trị khác nhau trên các thiết bị khác nhau và tổng hợp chúng.

```

data = [np.ones((1, 2), ctx=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])

```

### 14.5.5 Phân phối Dữ liệu

Ta cần một hàm hỗ trợ phân phối đều dữ liệu trong minibatch trên nhiều GPU. Ví dụ với 2 GPU, có thể ta sẽ muốn sao chép một nửa dữ liệu tới mỗi GPU. Ta sẽ sử dụng hàm có sẵn trong Gluon để chia và nạp dữ liệu (kiểm thử với ma trận  $4 \times 5$ ).

```
data = np.arange(20).reshape(4, 5)
devices = [npx.gpu(0), npx.gpu(1)]
split = gluon.utils.split_and_load(data, devices)
print('input :', data)
print('load into', devices)
print('output:', split)
```

Để sử dụng về sau, ta định nghĩa hàm `split_batch` để 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 (gluon.utils.split_and_load(X, devices),
            gluon.utils.split_and_load(y, devices))
```

### 14.5.6 Huấn luyện

Giờ chúng ta có thể lập trình việc huấn luyện với một minibatch trên nhiều GPU. Đoạn mã chủ yếu dựa trên phương pháp song song hóa dữ liệu trong chương này. Ta sẽ dùng các hàm phụ trợ `allreduce` và `split_and_load` ở trên để đồng bộ dữ liệu trên nhiều GPU. Lưu ý rằng ta không cần viết bất cứ đoạn mã cụ thể nào để song song hóa. Vì đồ thị tính toán không có phụ thuộc nào xuyên suốt các thiết bị trong một minibatch, chúng được thực thi song song *một cách tự động*.

```
def train_batch(X, y, device_params, devices, lr):
    X_shards, y_shards = split_batch(X, y, devices)
    with autograd.record(): # Loss is calculated separately on each GPU
        losses = [loss(lenet(X_shard, device_W), y_shard)
                  for X_shard, y_shard, device_W in zip(
                      X_shards, y_shards, device_params)]
        for l in losses: # Back Propagation is performed separately on each GPU
            l.backward()
    # Sum all gradients from each GPU and broadcast them to all GPUs
    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 ta có thể định nghĩa hàm huấn luyện. Hàm này có một chút khác biệt so với hàm huấn luyện trong các chương trước: ta cần chỉ định GPU và sao chép các tham số mô hình tới tất cả thiết bị. Mỗi batch được xử lý bằng `train_batch` nhằm tận dụng nhiều GPU. Để thuận tiện (và để mã nguồn ngắn gọn), ta tính độ chính xác trên một GPU (cách này *không hiệu quả* vì các GPU khác không được tận dụng).

```

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, times, acces = 10, [], []
    num_epochs = 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    timer = d2l.Timer()
    for epoch in range(num_epochs):
        timer.start()
        for X, y in train_iter:
            # Perform multi-GPU training for a single minibatch
            train_batch(X, y, device_params, devices, lr)
            npx.waitall()
        timer.stop()
        # Verify 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)}')

```

#### 14.5.7 Thí nghiệm

Hãy xem hàm trên hoạt động như thế nào trên một GPU. Ta sử dụng kích thước batch 256 và tốc độ học 0.2.

```
train(num_gpus=1, batch_size=256, lr=0.2)
```

Giữ nguyên kích thước batch và tốc độ học, tăng số GPU lên 2, ta có thể thấy sự cải thiện về độ chính xác trên tập kiểm tra xấp xỉ bằng thí nghiệm trước. Dưới góc nhìn thuật toán tối ưu, hai thí nghiệm là giống hệt nhau. Không may, ta không đạt được sự tăng tốc đáng kể nào: đơn giản vì mô hình quá nhỏ; hơn nữa tập dữ liệu cũng nhỏ, do đó cách huấn luyện không quá tinh vi của chúng ta trên nhiều GPU sẽ chịu chi phí đáng kể do Python. Về sau ta sẽ gặp các mô hình phức tạp hơn và các cách song song hóa tinh vi hơn. Hiện giờ hãy xem thí nghiệm trên Fashion-MNIST cho kết quả như thế nào.

```
train(num_gpus=2, batch_size=256, lr=0.2)
```

#### 14.5.8 Tóm tắt

- Có nhiều cách để chia việc huấn luyện mạng học sâu cho nhiều GPU. Có thể chia các tầng cho một GPU, dùng nhiều GPU cho một tầng, hoặc nhiều GPU cho dữ liệu. Hai cách đầu yêu cầu điều khiển việc truyền dữ liệu chặt chẽ. Song song hóa dữ liệu là cách đơn giản nhất.
- Không khó để huấn luyện bằng song song hóa dữ liệu. Tuy nhiên, cách này cần tăng kích thước hiệu dụng của minibatch để đạt hiệu quả.
- Dữ liệu được chia cho nhiều GPU, mỗi GPU thực thi các lượt truyền xuôi và ngược, sau đó các gradient được tổng hợp lại và kết quả được truyền về các GPU.
- Minibatch lớn có thể yêu cầu tốc độ học cao hơn một chút.

### 14.5.9 Bài tập

1. Khi huấn luyện trên nhiều GPU, thử thay đổi kích thước minibatch từ  $b$  thành  $k \cdot b$ , tức là nhân thêm số lượng GPU.
2. So sánh độ chính xác với các tốc độ học khác nhau. Tốc độ học thay đổi theo số lượng GPU như thế nào?
3. Lập trình hàm allreduce hiệu quả hơn để tổng hợp các tham số trên các GPU khác nhau (tại sao cách ban đầu không hiệu quả)?
4. Lập trình tính độ chính xác trên tập kiểm tra với nhiều GPU.

### 14.5.10 Thảo luận

- Tiếng Anh - MXNet<sup>288</sup>
- Tiếng Việt<sup>289</sup>

### 14.5.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long
- Phạm Hồng Vinh
- Nguyễn Cảnh Thượng
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật

## 14.6 Cách lập trình Súc tích đa GPU

Lập trình từ đầu việc song song hóa cho từng mô hình mới khá mất công. Hơn nữa, việc tối ưu các công cụ đồng bộ hóa sẽ cho hiệu suất cao. Sau đây chúng tôi sẽ giới thiệu cách thực hiện điều này bằng Gluon. Phần lý thuyết toán và các thuật toán giống trong Section 14.5. Như trước đây, ta bắt đầu bằng cách nhập các mô-đun cần thiết (tất nhiên là ta sẽ cần ít nhất hai GPU để chạy notebook này).

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

<sup>288</sup> <https://discuss.d2l.ai/t/364>

<sup>289</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 14.6.1 Ví dụ Đơn giản

Hãy sử dụng một mạng có ý nghĩa hơn một chút so với LeNet ở phần trước mà vẫn có thể huấn luyện dễ dàng và nhanh chóng. Chúng tôi chọn một biến thể của ResNet-18 (He et al., 2016a). Vì hình ảnh đầu vào rất nhỏ nên ta sửa đổi nó một chút. Cụ thể, điểm khác biệt so với ở Section 9.6 là ở phần đầu, ta sử dụng hạt nhân tích chập có kích thước, sải bước và đệm nhỏ hơn, và cũng loại bỏ đỉ tầng gộp cực đại.

```
#@save
def resnet18(num_classes):
    """A slightly modified ResNet-18 model."""
    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.Sequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(d2l.Residual(
                    num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(d2l.Residual(num_channels))
        return blk

    net = nn.Sequential()
    # This model uses a smaller convolution kernel, stride, and padding and
    # removes the maximum pooling layer
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
            nn.BatchNorm(), nn.Activation('relu'))
    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net
```

### 14.6.2 Khởi tạo Tham số và Công việc phụ trợ

Phương thức `initialize` cho phép ta thiết lập giá trị mặc định ban đầu cho các tham số trên thiết bị được chọn. Với độc giả mới, có thể tham khảo Section 6.8. Một điều rất thuận tiện là nó cũng cho phép ta khởi tạo mạng trên *nhiều* thiết bị cùng một lúc. Hãy thử xem cách nó hoạt động trong thực tế.

```
net = resnet18(10)
# get a list of GPUs
ctx = d2l.try_all_gpus()
# initialize the network on all of them
net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Sử dụng hàm `split_and_load` được giới thiệu trong phần trước, chúng ta có thể phân chia một minibatch dữ liệu và sao chép các phần dữ liệu vào danh sách các thiết bị được cung cấp bởi biến ngữ cảnh. Mạng sẽ *tự động* sử dụng GPU thích hợp để tính giá trị của lượn truyền xuôi. Ta tạo ra 4 mẫu dữ liệu và phân chia chúng trên các GPU như trước đây.

```

x = np.random.uniform(size=(4, 1, 28, 28))
x_shards = gluon.utils.split_and_load(x, ctx)
net(x_shards[0]), net(x_shards[1])

```

Khi dữ liệu được truyền qua mạng, các tham số tương ứng sẽ được khởi tạo *trên thiết bị mà dữ liệu được truyền qua*. Điều này có nghĩa là việc khởi tạo xảy ra theo từng thiết bị. Do ta lựa chọn việc khởi tạo trên GPU 0 và GPU 1, mạng chỉ được khởi tạo trên hai thiết bị này chứ trên CPU thì không. Trong thực tế, các tham số này thậm chí còn không tồn tại trên CPU. Ta có thể kiểm chứng điều này bằng cách in các tham số ra và theo dõi xem liệu có lỗi nào xảy ra hay không.

```

weight = net[0].params.get('weight')

try:
    weight.data()
except RuntimeError:
    print('not initialized on cpu')
weight.data(ctx[0])[0], weight.data(ctx[1])[0]

```

Cuối cùng, hãy cùng thay đổi đoạn mã đánh giá độ chính xác để có thể chạy song song trên nhiều thiết bị. Hàm này được viết lại từ hàm `evaluate_accuracy_gpu` ở [Section 8.6](#). Điểm khác biệt lớn nhất nằm ở việc ta tách một batch ra trước khi truyền vào mạng. Các phần còn lại gần như là giống hệt.

```

#@save
def evaluate_accuracy_gpus(net, data_iter, split_f=d2l.split_batch):
    # Query the list of devices
    ctx = list(net.collect_params().values())[0].list_ctx()
    metric = d2l.Accumulator(2) # num_corrected_examples, num_examples
    for features, labels in data_iter:
        X_shards, y_shards = split_f(features, labels, ctx)
        # Run in parallel
        pred_shards = [net(X_shard) for X_shard in X_shards]
        metric.add(sum(float(d2l.accuracy(pred_shard, y_shard)) for
                      pred_shard, y_shard in zip(
                          pred_shards, y_shards)), labels.size)
    return metric[0] / metric[1]

```

### 14.6.3 Huấn luyện

Như phần trên, đoạn mã huấn luyện cần thực hiện một số hàm cơ bản để quá trình song song hóa đạt hiệu quả:

- Các tham số của mạng cần được khởi tạo trên tất cả các thiết bị.
- Trong suốt quá trình lặp trên tập dữ liệu, các minibatch được chia nhỏ cho tất cả các thiết bị.
- Ta tính toán song song hàm mất mát và gradient của nó trên tất cả các thiết bị.
- Mất mát được tích luỹ (bởi phương thức huấn luyện `trainer`) và các tham số được cập nhật tương ứng.

Cuối cùng ta tính toán (vẫn song song) độ chính xác và báo cáo giá trị cuối cùng của mạng. Quá trình huấn luyện ở đây khá giống với chương trước, trừ việc ta cần chia nhỏ và tổng hợp lại dữ liệu.

```

def train(num_gpus, batch_size, lr):
    train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
    ctx = [d2l.try_gpu(i) for i in range(num_gpus)]
    net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr})
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    timer, num_epochs = d2l.Timer(), 10
    animator = d2l.Animator('epoch', 'test acc', xlim=[1, num_epochs])
    for epoch in range(num_epochs):
        timer.start()
        for features, labels in train_iter:
            X_shards, y_shards = d2l.split_batch(features, labels, ctx)
            with autograd.record():
                losses = [loss(net(X_shard), y_shard) for X_shard, y_shard
                          in zip(X_shards, y_shards)]
                for l in losses:
                    l.backward()
                trainer.step(batch_size)
            npx.waitall()
            timer.stop()
            animator.add(epoch + 1, (evaluate_accuracy_gpus(net, test_iter),))
    print(f'test acc: {animator.Y[0][-1]:.2f}, {timer.avg():.1f} sec/epoch '
          f'on {str(ctx)}')

```

#### 14.6.4 Thử nghiệm

Hãy cùng xem cách hoạt động trong thực tế. Để khởi động, ta huấn luyện mạng này trên một GPU đơn.

```
train(num_gpus=1, batch_size=256, lr=0.1)
```

Tiếp theo, ta sử dụng 2 GPU để huấn luyện. Mô hình ResNet-18 phức tạp hơn đáng kể so với LeNet. Đây chính là cơ hội để song song hóa chứng tỏ lợi thế của nó, vì thời gian dành cho việc tính toán lớn hơn đáng kể so với thời gian đồng bộ hóa các tham số. Điều này giúp cải thiện khả năng mở rộng do tổng chi phí song song hóa không quá đáng kể.

```
train(num_gpus=2, batch_size=512, lr=0.2)
```

#### 14.6.5 Tóm tắt

- Gluon cung cấp các hàm để khởi tạo mô hình trên nhiều thiết bị bằng cách cung cấp một danh sách ngũ cành.
- Dữ liệu được tự động đánh giá trên các thiết bị mà nó được lưu trữ.
- Chú ý việc khởi tạo mạng trên mỗi thiết bị trước khi thử truy cập vào các tham số trên thiết bị đó. Nếu không khả năng cao sẽ có lỗi xảy ra.
- Các thuật toán tối ưu tự động tổng hợp kết quả trên nhiều GPU.

#### 14.6.6 Bài tập

- Phần này ta sử dụng ResNet-18. Hãy thử với số epoch, kích thước batch và tốc độ học khác. Thử sử dụng nhiều GPU hơn để tính toán. Chuyện gì sẽ xảy ra nếu ta chạy mô hình này trên máy chủ p2.16xlarge với 16 GPU?
- Đôi khi mỗi thiết bị khác nhau cung cấp khả năng tính toán khác nhau. Ta có thể sử dụng GPU và CPU cùng lúc. Vậy ta nên phân chia công việc thế nào? Liệu việc phân chia có đáng hay không? Tại sao?
- Chuyện gì sẽ xảy ra nếu ta bỏ hàm `npx.waitall()`? Bạn sẽ thay đổi quá trình huấn luyện thế nào để có thể xử lý song song tối đa 2 bước cùng lúc?

#### 14.6.7 Thảo luận

- Tiếng Anh - MXNet<sup>290</sup>
- Tiếng Việt<sup>291</sup>

#### 14.6.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh

### 14.7 Máy chủ Tham số

Khi ta chuyển từ các GPU đơn sang đa GPU rồi sang nhiều máy chủ đa GPU, có khả năng các GPU được dàn trải qua nhiều khay chứa và bộ chuyển mạch mạng. Điều này khiến các giải thuật huấn luyện phân tán và song song trở nên phức tạp hơn nhiều. Các chi tiết nhỏ cũng trở nên quan trọng vì các phương thức kết nối khác nhau có băng thông rất khác nhau. Chẳng hạn, NVLink có băng thông lên tới 100GB/s qua 6 đường kết nối với cách thiết lập thích hợp, PCIe 3.0 16x làn có băng thông 16GB/s, trong khi ngay cả Ethernet 100GbE tốc độ cao chỉ đạt 10GB/s. Ngoài ra, khó có thể hy vọng rằng một nhà xây dựng mô hình thống kê cũng là một chuyên gia về kết nối mạng và hệ thống.

Ý tưởng cốt lõi của máy chủ tham số được đề xuất từ (Smola & Narayananurthy, 2010) trong ngữ cảnh các mô hình biến ẩn phân tán. Kế tiếp, một bản mô tả về ý nghĩa của tác vụ đẩy và kéo (*push and pull*) được giới thiệu trong (Ahmed et al., 2012) và một bản mô tả về hệ thống này cùng với

<sup>290</sup> <https://discuss.d2l.ai/t/365>

<sup>291</sup> <https://forum.machinelearningcoban.com/c/d2l>

thư viện mã nguồn mở được công bố trong (Li et al., 2014). Trong phần kế tiếp, ta sẽ tìm hiểu các thành phần cần thiết để đạt được hiệu suất cao.

### 14.7.1 Huấn luyện Song song Dữ liệu

Hãy cùng xem xét tổng quan phương pháp huấn luyện song song dữ liệu cho việc huấn luyện phân tán. Ta bắt đầu bằng cách này vì việc lập trình sẽ trở nên đơn giản hơn nhiều so với những cách khác. Vì các GPU ngày nay có khá nhiều bộ nhớ, gần như không có một trường hợp đặc biệt nào (ngoại trừ phương pháp học sâu trên đồ thị) mà một phương pháp song song hóa khác lại thích hợp hơn. Fig. 14.7.1 mô tả biến thể của việc song song hóa dữ liệu mà ta đã lập trình ở phần trước. Khía cạnh then chốt ở dạng này là việc tổng hợp gradient diễn ra trên GPU 0 trước khi các tham số cập nhật được phân phát tới tất cả GPU.

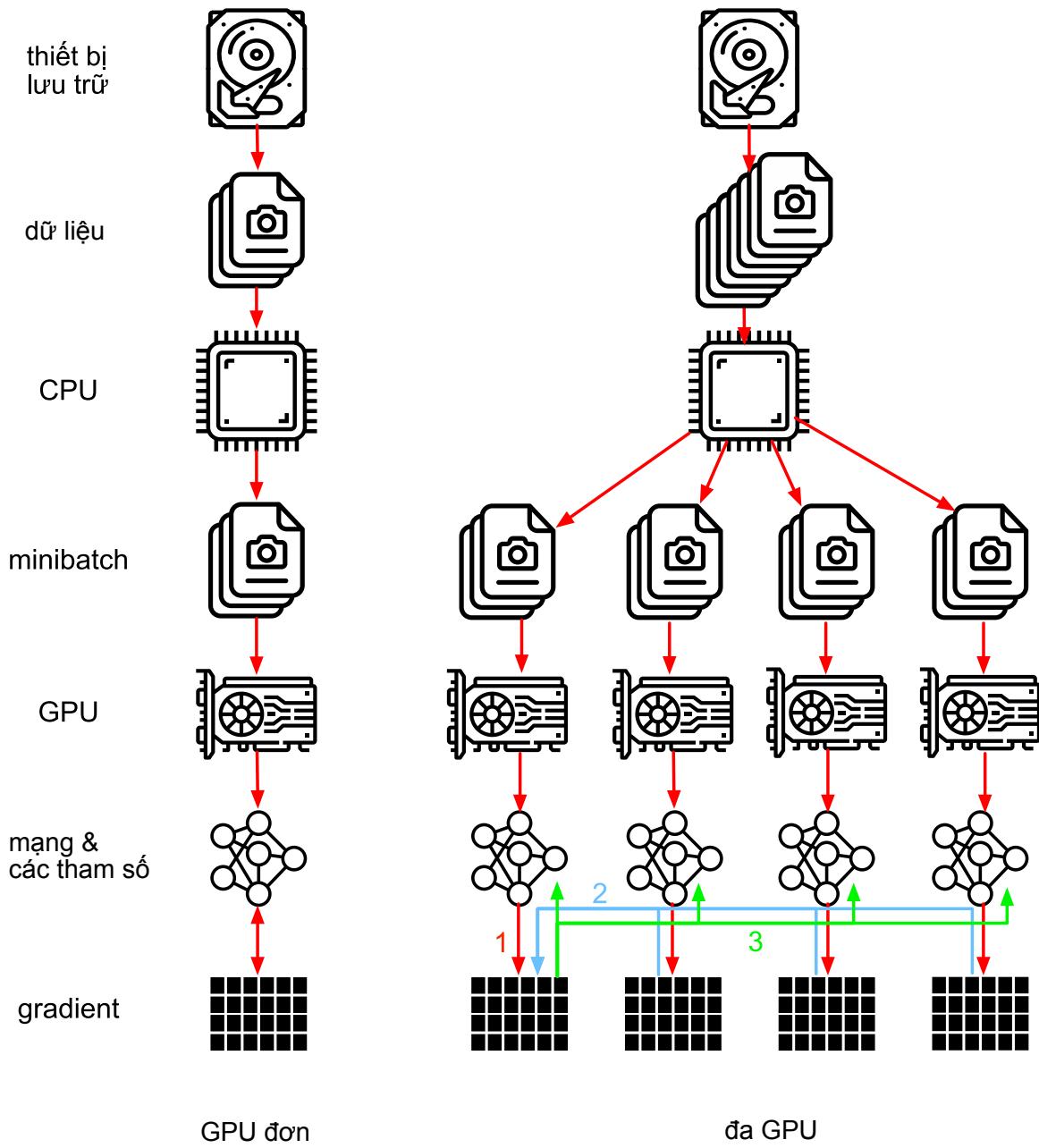


Fig. 14.7.1: Trái: việc huấn luyện trên một GPU; Phải: dạng biến thể của việc huấn luyện trên nhiều GPU. Quá trình diễn ra như sau: (1) ta tính mát mát và gradient, (2) tất cả gradient được tổng hợp trên một GPU, (3) ta cập nhật tham số và các tham số đó được phân phối lại tới tất cả GPU.

Nhìn lại, ta không có lý do gì đặc biệt khi quyết định tổng hợp gradient trên GPU 0. Dù sao thì ta cũng có thể tổng hợp gradient trên CPU. Và ta còn có thể tổng hợp một vài tham số trên một GPU và các tham số còn lại trên một GPU khác. Miễn là thuật toán tối ưu hỗ trợ điều này, ta không có lý do gì để không thể thực hiện. Ví dụ, giả sử ta có bốn vector tham số  $\mathbf{v}_1, \dots, \mathbf{v}_4$  với các gradient tương ứng là  $\mathbf{g}_1, \dots, \mathbf{g}_4$ , ta có thể tổng hợp gradient của mỗi vector tham số trên một GPU.

$$\mathbf{g}_i = \sum_{j \in \text{GPU}} \mathbf{g}_{ij} \quad (14.7.1)$$

Cách lý luận này trông có vẻ rất tùy tiện và vô nghĩa. Sau cùng, phần toán xuyên suốt bên dưới vẫn không thay đổi. Nhưng ở đây chúng ta đang làm việc cùng các thiết bị phần cứng vật lý với các bus có băng thông khác nhau như đã thảo luận ở [Section 14.4](#). Xét một máy chủ GPU 4-chiều được mô tả trong [Fig. 14.7.2](#). Nếu nó được kết nối cực kỳ tốt, nó có thể sở hữu một card mạng với tốc độ 100 GbE. Những con số phổ biến hơn thường nằm trong khoảng 1-10 GbE với băng thông hiệu dụng từ 100MB/s đến 1GB/s. Vì các CPU thường có quá ít làn PCIe để kết nối trực tiếp với toàn bộ GPU (ví dụ, CPU thông dụng của Intel có 24 làn) ta cần một **mạch đa hợp (multiplexer)**<sup>292</sup>. Băng thông tới CPU qua cổng PCIe 16 làn thế hệ 3 là 16GB/s. Đây cũng là tốc độ mà *mỗi* GPU được kết nối với bộ chuyển mạch. Điều này có nghĩa là việc truyền tin trực tiếp giữa các GPU sẽ hiệu quả hơn.

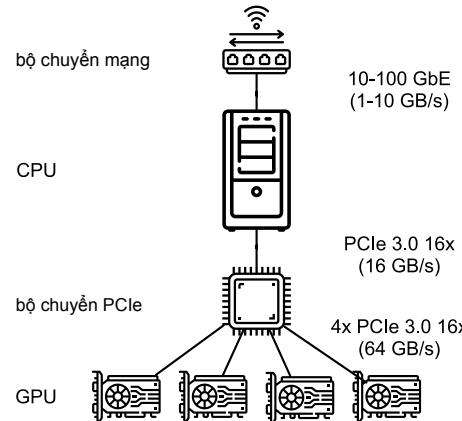


Fig. 14.7.2: Một máy chủ GPU 4-chiều.

Để minh họa cho luận điểm trên, giả sử ta cần 160MB để lưu trữ các gradient. Trong trường hợp này, sẽ tốn 30ms để gửi các giá trị gradient này từ 3 thiết bị GPU đến chiếc GPU còn lại (mỗi đợt truyền tin tốn 10ms =  $160\text{MB} / 16\text{GB/s}$ ). Việc truyền lại các vector trọng số mất thêm 30ms nữa, tổng cộng tốn 60ms. Nếu ta gửi toàn bộ dữ liệu đến CPU sẽ phát sinh thêm 40ms vì *mỗi* GPU cần gửi dữ liệu đến CPU, và tính cả thời gian truyền lại các vector trọng số sẽ tốn 80ms. Cuối cùng, giả định rằng ta có thể chia nhỏ các giá trị gradient thành bốn phần, mỗi phần 40MB. Giờ ta có thể tổng hợp mỗi phần trên một GPU riêng biệt *một cách đồng thời* vì bộ chuyển mạch PCIe cho phép sử dụng toàn bộ băng thông cho mỗi kết nối. Thay vì 30ms như trước, quá trình này chỉ tốn 7.5ms và 15ms cho toàn bộ quá trình đồng bộ. Nói ngắn gọn, tùy thuộc vào cách các tham số được đồng bộ với nhau, quá trình này có thể chiếm từ 15ms đến 80ms. [Fig. 14.7.3](#) minh họa sự khác biệt giữa các chiến lược trao đổi tham số khác nhau.

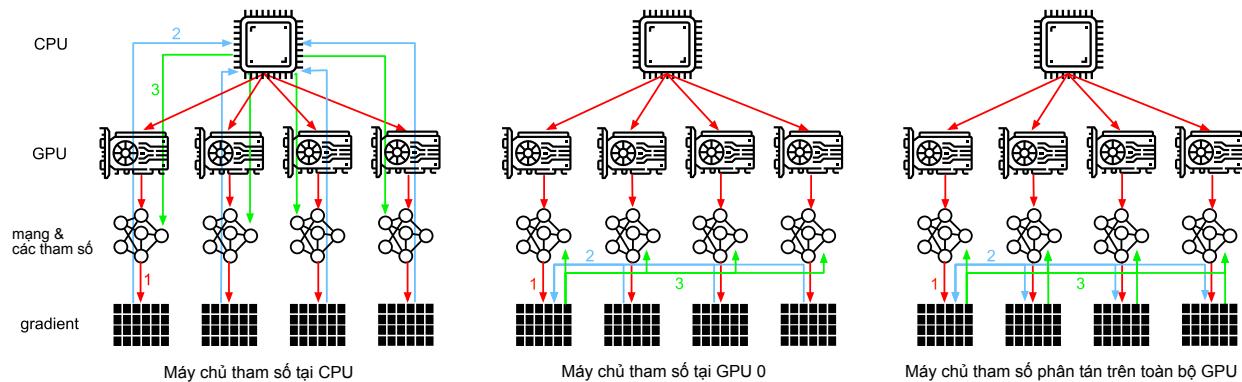


Fig. 14.7.3: Các chiến lược đồng bộ

<sup>292</sup> <https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches>

Lưu ý rằng ta còn một công cụ nữa để sử dụng khi muốn cải thiện hiệu suất: trong một mạng sâu sẽ cần một khoảng thời gian để tính toán toàn bộ gradient từ trên xuống dưới. Ta có thể bắt đầu đồng bộ gradient cho một vài nhóm tham số trong khi chúng ta vẫn đang tính gradient cho những nhóm khác (các chi tiết kỹ thuật để thực hiện việc này khá phức tạp). Bạn đọc hãy tham khảo (Sergeev & DelBalso, 2018) để biết chi tiết cách làm điều này trong Horovod<sup>293</sup>.

### 14.7.2 Đồng bộ dạng Vòng

Khi nói tới đồng bộ hóa trên các phần cứng học sâu tiên tiến, ta thường gặp những cách kết nối mạng rất riêng. Ví dụ, máy P3.16xlarge trên AWS và NVIDIA DGX-2 cùng sử dụng cấu trúc kết nối trong Fig. 14.7.4. Mỗi GPU kết nối với một CPU chủ thông qua kết nối PCIe có tốc độ tối đa là 16 GB/s. Hơn nữa, mỗi GPU có 6 kết nối NVLink với khả năng truyền đến 300 Gbit/s theo cả hai hướng. Điều này có nghĩa là mỗi kết nối sẽ có tốc độ khoảng 18 GB/s theo mỗi hướng. Nói ngắn gọn, băng thông tổng hợp của NVLink lớn hơn đáng kể so với băng thông của PCIe. Câu hỏi đặt ra là làm sao để tận dụng triệt để điều đó.

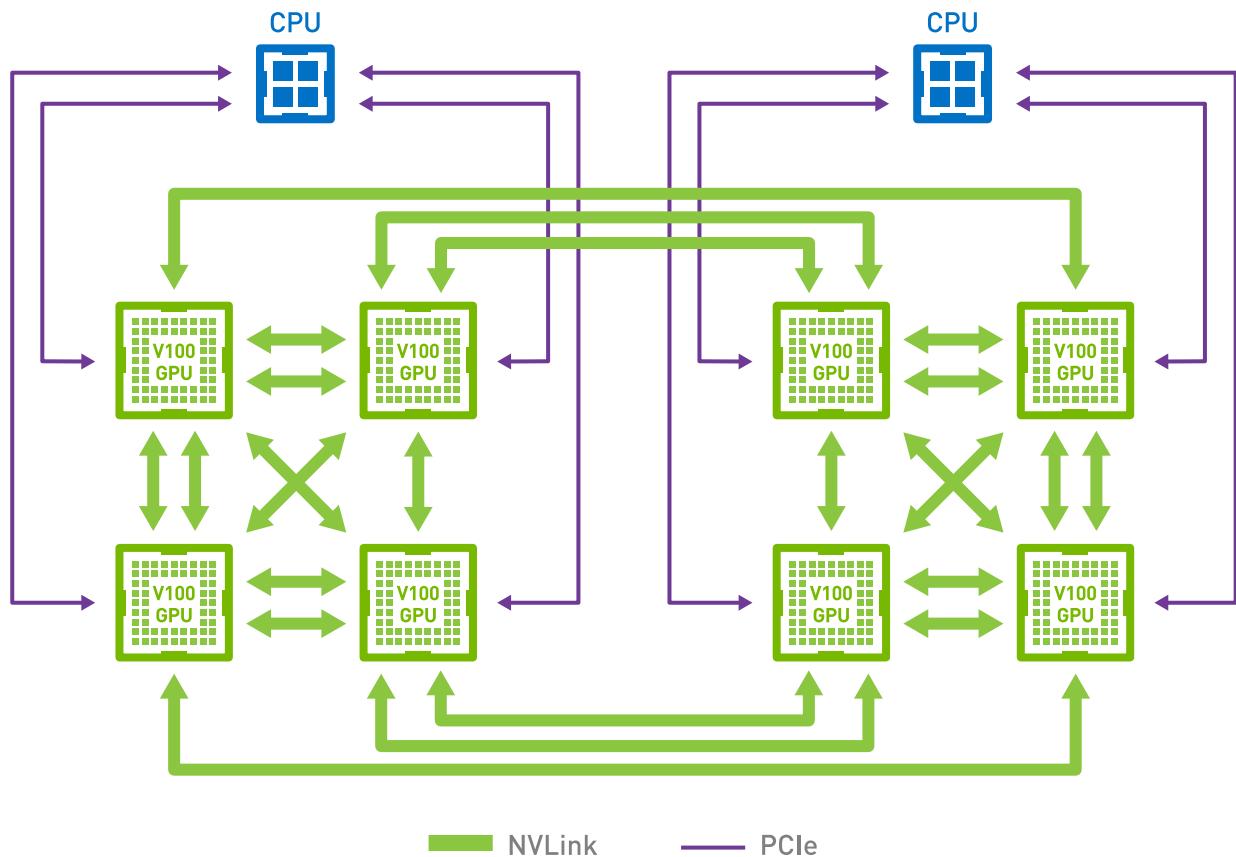


Fig. 14.7.4: Kết nối NVLink trên các máy chủ 8 GPU V100 (hình ảnh được sự đồng ý từ NVIDIA).

Hóa ra theo (Wang et al., 2018), chiến thuật đồng bộ tối ưu là phân tách mạng thành hai kết nối dạng vòng và sử dụng chúng để đồng bộ dữ liệu một cách trực tiếp. Fig. 14.7.5 minh họa việc mạng có thể được phân tách thành một kết nối dạng vòng (1-2-3-4-5-6-7-8-1) với băng thông NVLink gấp đôi và một kết nối dạng vòng khác (1-4-6-3-5-8-2-7-1) với băng thông bình thường. Việc thiết kế một giao thức đồng bộ hóa hiệu quả trong trường hợp này không hề đơn giản.

<sup>293</sup> <https://github.com/horovod/horovod>

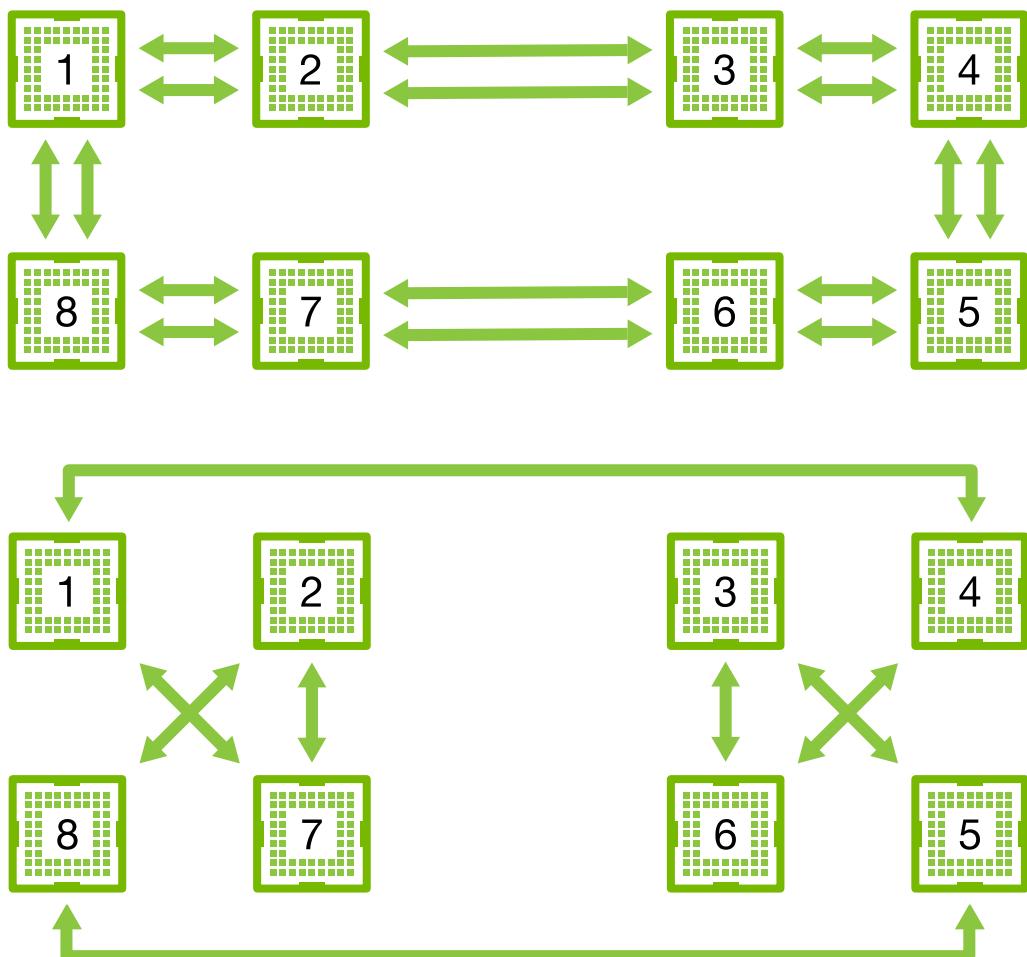


Fig. 14.7.5: Phân tách mạng NVLink thành hai kết nối dạng vòng.

Xét một thí nghiệm tưởng tượng như sau: cho một kết nối dạng vòng có  $n$  đơn vị tính toán (GPU) ta có thể truyền các giá trị gradient từ thiết bị thứ nhất đến thiết bị thứ hai. Ở đó nó sẽ được cộng thêm vào gradient cục bộ và rồi truyền tiếp đến thiết bị thứ ba, và tiếp tục như vậy với các thiết bị sau. Sau  $n - 1$  bước, gradient tổng hợp sẽ nằm ở thiết bị cuối cùng. Điều này có nghĩa là thời gian tổng hợp gradient sẽ tăng tuyến tính theo số lượng thiết bị trong mạng. Nhưng nếu ta làm vậy, thuật toán sẽ hoạt động kém hiệu quả. Dù sao, tại mọi thời điểm chỉ có một thiết bị thực hiện việc truyền tin. Chuyện gì sẽ xảy ra nếu ta chia các giá trị gradient thành  $n$  khúc và bắt đầu đồng bộ khúc thứ  $i$  tại thiết bị  $i$ ? Vì mỗi khúc có kích thước  $1/n$ , tổng thời gian giờ sẽ là  $(n - 1)/n \approx 1$ . Nói cách khác, thời gian tổng hợp gradient *không tăng* khi ta tăng số thiết bị trong mạng. Quả là một kết quả đáng kinh ngạc. Fig. 14.7.6 minh họa chuỗi các bước với số thiết bị  $n = 4$ .

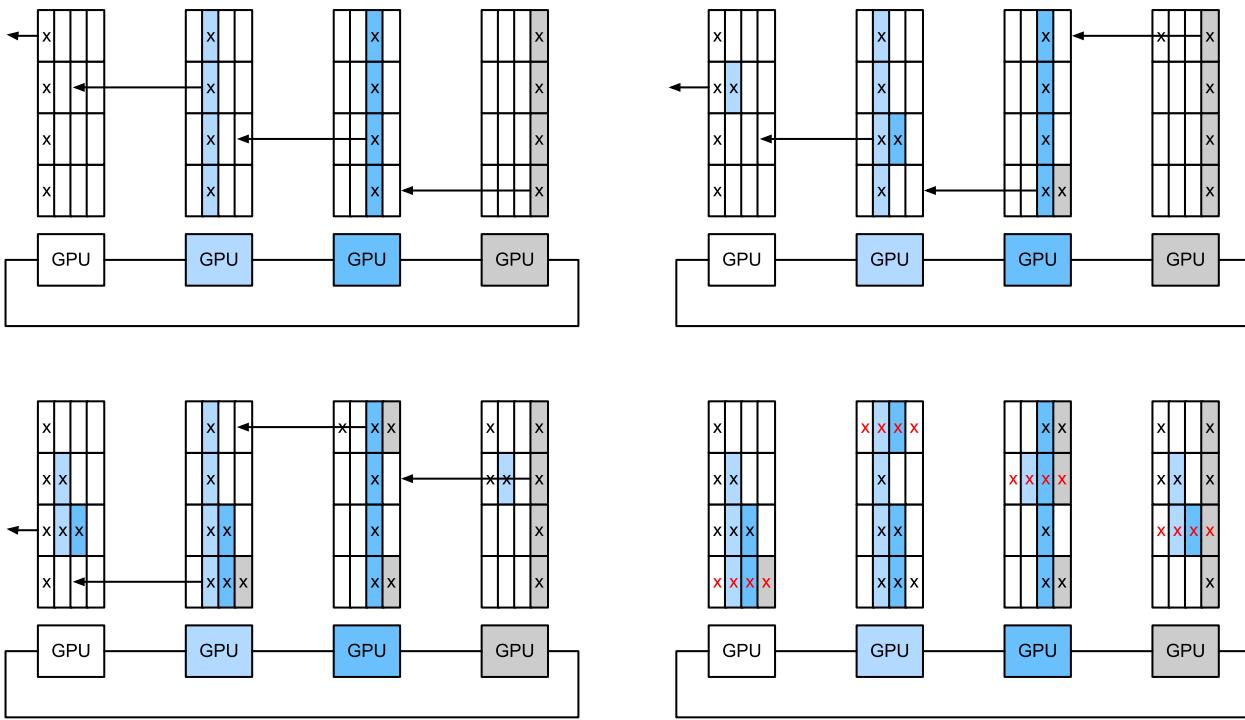


Fig. 14.7.6: Đồng bộ vòng trên 4 nút. Mỗi nút truyền một phần gradient sang nút liền kề bên trái cho đến khi gradient đầy đủ có thể được tìm thấy tại nút liền kề bên phải nó.

Nếu vẫn sử dụng ví dụ đồng bộ 160 MB trên 8 GPU V100, ta có thể đạt xấp xỉ  $2 \cdot 160\text{MB} / (3 \cdot 18\text{GB/s}) \approx 6\text{ms}$ . Kết quả này tốt hơn so với việc sử dụng bus PCIe một chút, mặc dù lúc này ta sử dụng đến 8 GPU. Chú ý rằng trong thực tế những con số này sẽ không được tốt như vậy, do các framework học sâu thường gặp khó khăn trong việc tổng hợp thông tin thành cụm lớn hơn để truyền đi. Hơn nữa, việc định thời là cực kì quan trọng. Lưu ý, mọi người thường hiểu nhầm rằng đồng bộ vòng có bản chất khác hẳn so với các thuật toán đồng bộ khác. Thực ra điểm khác biệt duy nhất nằm ở đường đi đồng bộ có phần tinh vi hơn so với phương pháp cây đơn giản.

### 14.7.3 Huấn luyện trên Nhiều Máy tính

Việc huấn luyện phân tán trên nhiều máy tính tạo nên một thử thách mới: ta cần phải giao tiếp với các máy chủ chỉ được liên kết với nhau qua loại cáp có băng thông tương đối thấp. Trong một số trường hợp tốc độ thậm chí có thể chậm gấp hơn 10 lần. Đồng bộ nhiều thiết bị là công việc khá phức tạp. Suy cho cùng, mỗi máy tính khác nhau chạy đoạn mã huấn luyện với tốc độ khác nhau đôi chút. Do đó ta cần *đồng bộ* chúng nếu muốn sử dụng tối ưu phân tán đồng bộ. Fig. 14.7.7 mô tả quá trình huấn luyện phân tán song song.

1. Một batch dữ liệu (khác nhau) được đọc trên mỗi máy tính, chia đều cho các GPU và truyền đến bộ nhớ của GPU. Ở đó các dự đoán và gradient được tính toán riêng biệt theo từng batch trên các GPU khác nhau.
2. Các gradient trên tất cả các GPU cục bộ được tổng hợp trên một GPU (hoặc các phần khác nhau được tổng hợp trên nhiều GPU khác nhau).
3. Các gradient được truyền đến CPU.
4. CPU truyền các gradient đến máy chủ tham số trung tâm để tổng hợp tất cả các gradient.

5. Các gradient tổng sau đó được sử dụng để cập nhật các vector trọng số. Tiếp đó thì các vector trọng số mới được phân phát cho các CPU.
6. Thông tin cập nhật được truyền tới một (hoặc nhiều) GPU.
7. Các vector trọng số đã được cập nhật sau đó được phân bổ đều cho tất cả các GPU.

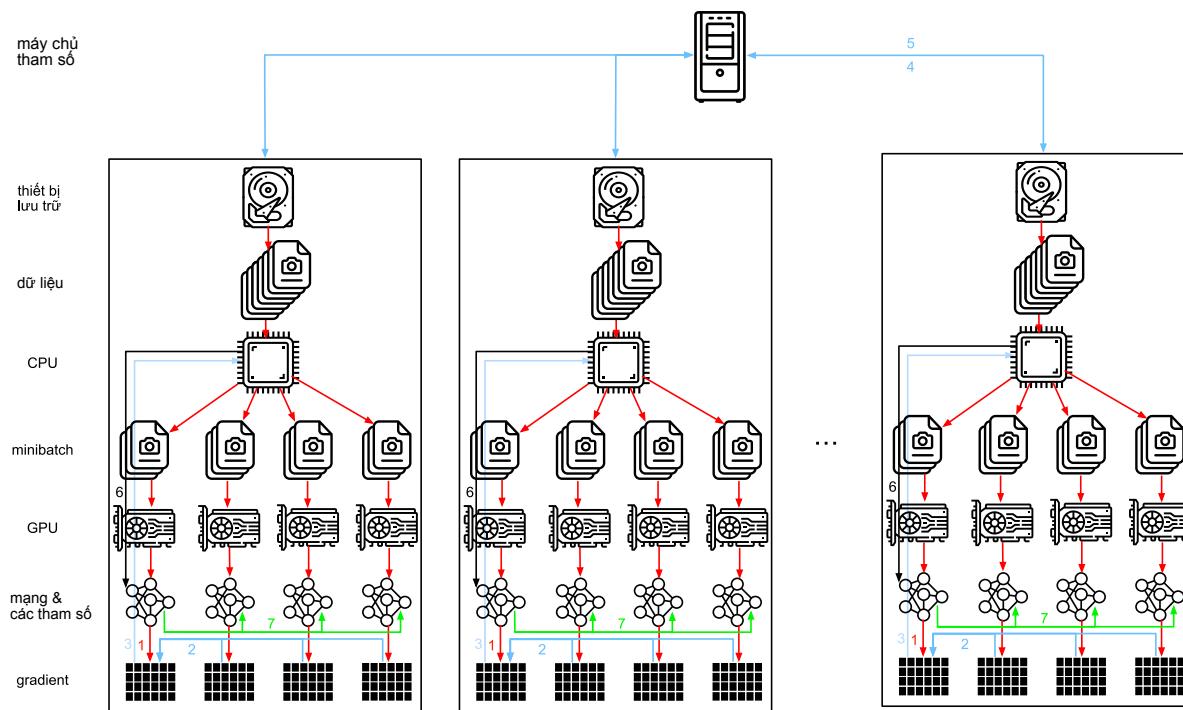


Fig. 14.7.7: Huấn luyện song song phân tán trên nhiều máy tính đa GPU

Các thao tác trên nhìn qua thì có vẻ khá dễ hiểu. Quả thật, chúng có thể được thực hiện một cách hiệu quả *trong* một máy tính. Tuy nhiên khi xét trên nhiều máy tính, ta có thể thấy rằng chính máy chủ tham số trung tâm đã trở thành nút nghẽn cổ chai. Suy cho cùng, băng thông của mỗi máy chủ là có hạn, do đó đối với  $m$  máy thợ, thời gian để truyền toàn bộ gradient đến máy chủ là  $O(m)$ . Ta có thể vượt qua rào cản này bằng cách tăng số lượng máy chủ lên  $n$ . Khi đó mỗi máy chủ chỉ cần lưu trữ  $O(1/n)$  tham số, do đó tổng thời gian cần để cập nhật và tối ưu trở thành  $O(m/n)$ . Tổng thời gian này sẽ tăng lên theo hằng số bất kể số lượng máy thợ ta sử dụng là bao nhiêu. Trong thực tế, các máy tính sẽ vừa là máy chủ và máy thợ. Fig. 14.7.8 minh họa thiết kế này. Đặc giả có thể tham khảo (Li et al., 2014) để biết thêm chi tiết. Đặc biệt, việc đảm bảo các máy tính hoạt động với độ trễ không quá lớn không phải là một chuyện dễ dàng. Chúng tôi sẽ bỏ qua chi tiết về các rào cản và chỉ đề cập ngắn gọn tới việc cập nhật đồng bộ và bất đồng bộ dưới đây.

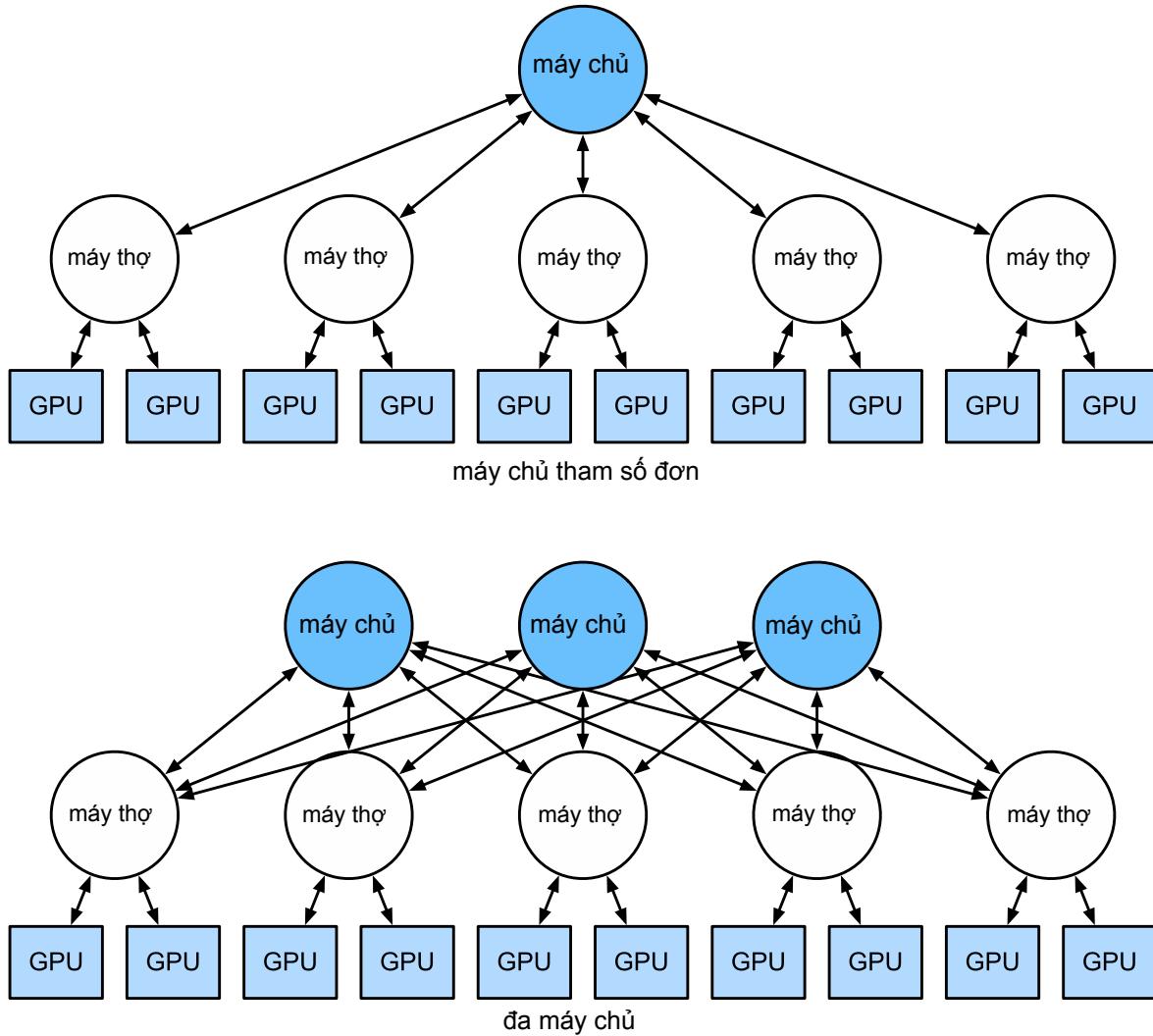


Fig. 14.7.8: Trên - một máy chủ tham số là một nút nghẽn cổ chai do băng thông của nó có hạn. Dưới - nhiều máy chủ tham số lưu trữ từng phần các tham số với băng thông tổng.

#### 14.7.4 Lưu trữ (Khóa, Giá trị)

Lập trình các bước cần thiết trên cho việc huấn luyện phân tán trên nhiều GPU trong thực tế không hề đơn giản. Cụ thể, có khả năng ta sẽ gặp rất nhiều lựa chọn khác nhau. Do đó, rất đáng để sử dụng một phép trừu tượng hóa khá phổ biến là lưu trữ cặp (khóa, giá trị) với cách cập nhật được định nghĩa lại. Trên nhiều máy chủ và nhiều GPU, việc tính toán gradient có thể được định nghĩa như sau

$$\mathbf{g}_i = \sum_{k \in \text{máy chủ}} \sum_{j \in \text{GPU}} \mathbf{g}_{ijk}. \quad (14.7.2)$$

Đặc điểm chính của thao tác này nằm ở việc nó là một *phép rút gọn có tính giao hoán*, tức nó gộp nhiều vector thành một vector và thứ tự áp dụng thao tác này không quan trọng. Vì không cần (phải) kiểm soát chi tiết thời điểm gradient được nhận, thao tác này rất phù hợp với mục đích của

chúng ta. Lưu ý rằng ta có thể thực hiện phép rút gọn theo từng bước. Thêm nữa, chú ý rằng thao tác này độc lập giữa các khối  $i$  gắn liền với các tham số (và các gradient) khác nhau.

Điều này cho phép ta định nghĩa hai thao tác sau: đẩy, để cộng dồn gradient; và kéo, để lấy lại gradient được cộng dồn. Vì ta có nhiều tập gradient (do có nhiều tầng), ta cần gán chỉ số cho gradient bằng khóa  $i$ . Sự giống nhau giữa phương pháp này và việc lưu trữ (khóa, giá trị) như phương pháp được giới thiệu trong Dynamo (DeCandia et al., 2007) không phải là ngẫu nhiên. Chúng thỏa mãn rất nhiều tính chất, cụ thể là khi phân phối các tham số cho nhiều máy chủ.

- **đẩy (khóa, giá trị)** gửi một gradient cụ thể (giá trị) từ máy thợ đến thiết bị lưu trữ chung. Tại đây các tham số được tổng hợp lại, ví dụ bằng cách lấy tổng.
- **kéo (khóa, giá trị)** lấy lại tham số đã được tổng hợp từ thiết bị lưu trữ chung, sau khi đã kết hợp gradient từ tất cả máy thợ.

Bằng cách ẩn đi sự phức tạp của việc đồng bộ sau các thao tác đơn giản là đẩy và kéo, ta có thể tách những mối bận tâm theo hai hướng: của các nhà mô hình thống kê, những người muốn biểu diễn việc tối ưu một cách đơn giản và các kỹ sư hệ thống, những người cần giải quyết sự phức tạp sẵn có trong việc đồng bộ hóa phân tán. Trong phần tiếp theo ta sẽ thử nghiệm việc lưu trữ (khóa, giá trị) trong thực tế.

#### 14.7.5 Tóm tắt

- Việc đồng bộ cần có độ thích ứng cao với hạ tầng mạng cụ thể và kết nối trong máy chủ. Điều này có thể tạo ra khác biệt đáng kể trong thời gian đồng bộ.
- Đồng bộ dạng vòng có thể là phương án tối ưu với các máy chủ P3 và DGX-2, còn với các loại máy chủ khác thì không hẳn.
- Chiến lược đồng bộ phân cấp rất tốt khi thêm nhiều máy chủ tham số để tăng băng thông.
- Giao tiếp bất đồng bộ (khi việc tính toán vẫn đang diễn ra) có thể cải thiện hiệu năng.

#### 14.7.6 Bài tập

1. Bạn có thể cải thiện đồng bộ dạng vòng hơn nữa không? Gợi ý: bạn có thể gửi thông tin theo cả hai chiều.
2. Đồng bộ bất đối xứng hoàn toàn có độ trễ nào không?
3. Nên để khả năng chịu lỗi (*fault tolerance*) như thế nào? Nếu một máy chủ gặp trục trặc thì sao? Đây có phải vấn đề nghiêm trọng không?
4. Lưu checkpoint như thế nào?
5. Bạn có thể tăng tốc việc tổng hợp dạng cây (*tree aggregation*) không?
6. Tìm hiểu các cách rút gọn khác (như dạng bán vòng giao hoán - *commutative semiring*).

#### **14.7.7 Thảo luận**

- Tiếng Anh<sup>294</sup>
- Tiếng Việt<sup>295</sup>

#### **14.7.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Đỗ Trường Giang
- Nguyễn Thành Hòa
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật

#### **14.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Tâm

---

<sup>294</sup> <https://discuss.d2l.ai/t/366>

<sup>295</sup> <https://forum.machinelearningcoban.com/c/d2l>



# 15 | Thị giác Máy tính

Nhiều ứng dụng trong lĩnh vực thị giác máy tính liên quan mật thiết đến cuộc sống hàng ngày của chúng ta ở hiện tại và tương lai; từ chẩn đoán y tế, xe tự hành, camera giám sát đến những bộ lọc thông minh. Trong những năm gần đây, công nghệ học sâu đã nâng cao đáng kể chất lượng của hệ thống thị giác máy tính. Có thể nói rằng những ứng dụng thị giác máy tính tiên tiến nhất gần như không thể tách rời khỏi học sâu.

Ở chương “Mạng Nơ-ron Tích chập”, chúng tôi đã giới thiệu các mô hình học sâu thường được sử dụng trong lĩnh vực thị giác máy tính và đã cùng thực hành một số tác vụ phân loại hình ảnh đơn giản. Trong chương này, chúng tôi sẽ giới thiệu các phương pháp tăng cường hình ảnh (*image augmentation*), phương pháp tinh chỉnh (*fine-tuning*) và áp dụng chúng vào phân loại hình ảnh. Tiếp đến, ta sẽ khám phá các phương pháp phát hiện vật thể khác nhau, cùng tìm hiểu cách sử dụng các mạng tích chập đầy đủ để thực hiện phân vùng ngữ nghĩa trên hình ảnh. Sau đó, chúng tôi giải thích cách sử dụng kỹ thuật truyền tài phong cách (*style transfer*) để tạo nên những hình ảnh trông giống như bìa của cuốn sách này. Cuối cùng, chúng tôi sẽ thực hiện các bài tập thực hành trên hai bộ dữ liệu thị giác máy tính quan trọng để xem lại nội dung của chương này và những chương trước.

## 15.1 Tăng cường Ảnh

Trong Section 9.1 chúng ta có đề cập đến việc các bộ dữ liệu lớn là điều kiện tiên quyết cho sự thành công của các mạng nơ-ron sâu. Kỹ thuật tăng cường ảnh giúp mở rộng kích thước của tập dữ liệu huấn luyện thông qua việc áp dụng một loạt thay đổi ngẫu nhiên trên các mẫu ảnh, từ đó tạo ra các mẫu huấn luyện tuy tương tự nhưng vẫn có sự khác biệt. Cũng có thể giải thích tác dụng của tăng cường ảnh là việc thay đổi ngẫu nhiên các mẫu dùng cho huấn luyện, làm giảm sự phụ thuộc của mô hình vào một số thuộc tính nhất định. Do đó giúp cải thiện năng lực khái quát hóa của mô hình.

Chẳng hạn, ta có thể cắt tập ảnh theo các cách khác nhau, để các đối tượng ta quan tâm xuất hiện ở các vị trí khác nhau, vì vậy giảm sự phụ thuộc của mô hình vào vị trí xuất hiện của đối tượng. Ta cũng có thể điều chỉnh độ sáng, màu sắc, và các yếu tố khác để giảm độ nhạy màu sắc của mô hình. Có thể khẳng định rằng kỹ thuật tăng cường ảnh đóng góp rất lớn cho sự thành công của mạng AlexNet. Tới đây, chúng ta sẽ thảo luận về kỹ thuật mà được sử dụng rộng rãi trong lĩnh vực thị giác máy tính này.

Trước tiên, thực hiện nhập các gói và mô-đun cần thiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
```

(continues on next page)

```
from mxnet.gluon import nn
npx.set_np()
```

### 15.1.1 Phương pháp Tăng cường Ảnh Thông dụng

Trong phần thử nghiệm này, ta sẽ dùng một ảnh có kích thước  $400 \times 500$  làm ví dụ.

```
d2l.set_figsize()
img = image.imread('../img/cat1.jpg')
d2l.plt.imshow(img.asnumpy());
```

Hầu hết các phương pháp tăng cường ảnh có một độ ngẫu nhiên nhất định. Để giúp việc quan sát tính hiệu quả của nó dễ hơn, ta sẽ định nghĩa hàm bổ trợ `apply`. Hàm này thực hiện phương thức tăng cường ảnh `aug` nhiều lần từ ảnh đầu vào `img` và hiển thị tất cả 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 ảnh

Lật hình ảnh sang trái và phải thường không thay đổi thể loại đối tượng. Đây là một trong những phương pháp tăng cường ảnh được sử dụng sớm nhất và rộng rãi nhất. Tiếp theo, chúng ta sử dụng mô-đun `transforms` để tạo thực thể `RandomFlipLeftRight`, ngẫu nhiên lật hình ảnh sang trái hoặc phải với xác suất 50%.

```
apply(img, gluon.data.vision.transforms.RandomFlipLeftRight())
```

Lật lên và xuống không được sử dụng phổ biến như lật trái và phải. Tuy nhiên, ít nhất là đối với hình ảnh ví dụ này, lật lên xuống không gây trở ngại cho việc nhận dạng. Tiếp theo, chúng tôi tạo thực thể `RandomFlipTopBottom` để lật hình ảnh lên và xuống với xác suất 50%.

```
apply(img, gluon.data.vision.transforms.RandomFlipTopBottom())
```

Trong ví dụ chúng ta sử dụng, con mèo nằm ở giữa ảnh, nhưng không phải tất cả các ảnh mèo khác đều sẽ như vậy. Section 8.5 có đề cập rằng tầng gộp có thể làm giảm độ nhạy của tầng tích chập với vị trí mục tiêu. Ngoài ra, chúng ta có thể làm cho các đối tượng xuất hiện ở các vị trí khác nhau trong ảnh theo tỷ lệ khác nhau bằng cách cắt ngẫu nhiên hình ảnh. Điều này cũng có thể làm giảm độ nhạy của mô hình với vị trí mục tiêu.

Trong đoạn mã sau, chúng tôi cắt ngẫu nhiên một vùng có diện tích từ 10% đến 100% diện tích ban đầu và tỷ lệ giữa chiều rộng và chiều cao của vùng được chọn ngẫu nhiên trong khoảng từ 0.5 đến 2. Sau đó, cả chiều rộng và chiều cao của vùng đều được biến đổi tỷ lệ thành 200 pixel. Trừ khi có quy định khác, giá trị ngẫu nhiên liên tục giữa  $a$  và  $b$  thu được bằng cách lấy mẫu đồng nhất trong khoảng  $[a, b]$ .

```
shape_aug = gluon.data.vision.transforms.RandomResizedCrop(  
    (200, 200), scale=(0.1, 1), ratio=(0.5, 2))  
apply(img, shape_aug)
```

## Đổi màu

Một phương pháp tăng cường khác là thay đổi màu sắc. Chúng ta có thể thay đổi bốn khía cạnh màu sắc của hình ảnh: độ sáng, độ tương phản, độ bão hòa và tông màu. Trong ví dụ dưới đây, chúng tôi thay đổi ngẫu nhiên độ sáng của hình ảnh với giá trị trong khoảng từ 50% ( $1 - 0.5$ ) đến 150% ( $1 + 0.5$ ) độ sáng của ảnh gốc.

```
apply(img, gluon.data.vision.transforms.RandomBrightness(0.5))
```

Tương tự vậy, ta có thể ngẫu nhiên thay đổi tông màu của ảnh.

```
apply(img, gluon.data.vision.transforms.RandomHue(0.5))
```

Ta cũng có thể tạo một thực thể RandomColorJitter và thiết lập để ngẫu nhiên thay đổi brightness (độ sáng), contrast (độ tương phản), saturation (độ bão hòa), và hue (tông màu) của ảnh cùng một lúc.

```
color_aug = gluon.data.vision.transforms.RandomColorJitter(  
    brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)  
apply(img, color_aug)
```

## Kết hợp nhiều Phương pháp Tăng cường Ảnh

Trong thực tế, chúng ta sẽ kết hợp nhiều phương pháp tăng cường ảnh. Ta có thể kết hợp các phương pháp trên và áp dụng chúng cho từng hình ảnh bằng cách sử dụng thực thể Compose.

```
augs = gluon.data.vision.transforms.Compose([  
    gluon.data.vision.transforms.RandomFlipLeftRight(), color_aug, shape_aug])  
apply(img, aug)
```

### 15.1.2 Huấn luyện Mô hình dùng Tăng cường Ảnh

Tiếp theo, ta sẽ xem xét làm thế nào để áp dụng tăng cường hình ảnh trong huấn luyện thực tế. Ở đây, ta sử dụng bộ dữ liệu CIFAR-10, thay vì Fashion-MNIST 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 và sự khác biệt về màu sắc và kích thước của các đối tượng trong bộ dữ liệu CIFAR-10 là đáng kể hơn. 32 hình ảnh huấn luyện đầu tiên trong bộ dữ liệu CIFAR-10 được hiển thị bên dưới.

```
d2l.show_images(gluon.data.vision.CIFAR10(  
    train=True)[0:32][0], 4, 8, scale=0.8);
```

Để có được kết quả cuối cùng trong dự đoán, ta thường chỉ áp dụng tăng cường ảnh khi huấn luyện nhưng không sử dụng các biến đổi ngẫu nhiên trong dự đoán. Ở đây, chúng ta chỉ sử dụng phương

pháp lật ngẫu nhiên trái phải đơn giản nhất. Ngoài ra, chúng ta sử dụng một thực thể ToTensor để chuyển đổi minibatch hình ảnh thành định dạng yêu cầu của MXNet, tức là, tensor số thực dấu phẩy động 32-bit có kích thước (kích thước batch, số kênh, chiều cao, chiều rộng) và phạm vi giá trị trong khoảng từ 0 đến 1.

```
train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor()])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor()])
```

Tiếp theo, ta định nghĩa một chức năng phụ trợ để giúp đọc hình ảnh và áp dụng tăng cường ảnh dễ dàng hơn. Hàm transform\_first được cung cấp bởi Gluon giúp thực thi tăng cường ảnh cho phần tử đầu tiên của mỗi mẫu huấn luyện (hình ảnh và nhãn), tức là chỉ áp dụng lên phần ảnh. Để biết thêm chi tiết về DataLoader, hãy tham khảo [Section 5.5](#).

```
def load_cifar10(is_train, augs, batch_size):
    return gluon.data.DataLoader(
        gluon.data.vision.CIFAR10(train=is_train).transform_first(augs),
        batch_size=batch_size, shuffle=is_train,
        num_workers=d2l.get_dataloader_workers())
```

## Sử dụng Mô hình Huấn luyện Đa GPU

Ta huấn luyện mô hình ResNet-18 như mô tả ở [Section 9.6](#) trên tập dữ liệu CIFAR-10. Cùng với đó ta áp dụng các phương pháp được mô tả trong `sec_multi_gpu_concise` và sử dụng mô hình huấn luyện đa GPU.

Tiếp theo, ta định nghĩa hàm huấn luyện để huấn luyện và đánh giá mô hình sử dụng nhiều GPU.

```
#@save
def train_batch_ch13(net, features, labels, loss, trainer, devices,
                      split_f=d2l.split_batch):
    X_shards, y_shards = split_f(features, labels, devices)
    with autograd.record():
        pred_shards = [net(X_shard) for X_shard in X_shards]
        ls = [loss(pred_shard, y_shard) for pred_shard, y_shard
              in zip(pred_shards, y_shards)]
        for l in ls:
            l.backward()
    # The True flag allows parameters with stale gradients, which is useful
    # later (e.g., in fine-tuning BERT)
    trainer.step(labels.shape[0], ignore_stale_grad=True)
    train_loss_sum = sum([float(l.sum()) for l in ls])
    train_acc_sum = sum(d2l.accuracy(pred_shard, y_shard)
                        for pred_shard, y_shard in zip(pred_shards, y_shards))
    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(), split_f=d2l.split_batch):
```

(continues on next page)

```

num_batches, timer = len(train_iter), d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs], ylim=[0, 1],
                        legend=['train loss', 'train acc', 'test acc'])
for epoch in range(num_epochs):
    # Store training_loss, training_accuracy, num_examples, num_features
    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, split_f)
        metric.add(l, acc, labels.shape[0], labels.size)
        timer.stop()
        if (i + 1) % (num_batches // 5) == 0:
            animator.add(epoch + i / num_batches,
                          (metric[0] / metric[2], metric[1] / metric[3],
                           None))
    test_acc = d2l.evaluate_accuracy_gpus(net, test_iter, split_f)
    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)}')

```

Giờ ta có thể định nghĩa hàm `train_with_data_aug` để áp dụng tăng cường ảnh vào huấn luyện mô hình. Hàm này tìm tất cả các GPU có sẵn và sử dụng Adam làm thuật toán tối ưu cho quá trình huấn luyện. Sau đó nó áp dụng tăng cường ảnh vào tập huấn luyện, và cuối cùng gọi đến hàm `train_ch13` được định nghĩa ở trên để huấn luyện và đánh giá mô hình.

```

batch_size, devices, net = 256, d2l.try_all_gpus(), d2l.resnet18(10)
net.initialize(init=init.Xavier(), ctx=devices)

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 = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    train_ch13(net, train_iter, test_iter, loss, trainer, 10, devices)

```

Giờ ta huấn luyện mô hình áp dụng tăng cường ảnh qua phép lật ngẫu nhiên trái và phải.

```
train_with_data_aug(train_augs, test_augs, net)
```

### 15.1.3 Tóm tắt

- Tăng cường ảnh sản sinh ra những ảnh ngẫu nhiên dựa vào dữ liệu có sẵn trong tập huấn luyện để đối phó với hiện tượng quá khớp.
- Để có thể thu được kết quả tin cậy trong quá trình dự đoán, thường thì ta chỉ áp dụng tăng cường ảnh lên mẫu huấn luyện, không áp dụng các biến đổi tăng cường ảnh ngẫu nhiên trong quá trình dự đoán.
- Mô-đun `transforms` của Gluon có các lớp thực hiện tăng cường ảnh.

### 15.1.4 Bài tập

1. Huấn luyện mô hình mà không áp dụng tăng cường ảnh: `train_with_data_aug(no_aug, no_aug)`. So sánh độ chính xác trong huấn luyện và kiểm tra khi áp dụng và không áp dụng tăng cường ảnh. Liệu thí nghiệm so sánh này có thể hỗ trợ cho luận điểm rằng tăng cường ảnh có thể làm giảm hiện tượng quá khớp? Tại sao?
2. Sử dụng thêm các phương thức tăng cường ảnh khác trên tập dữ liệu CIFAR-10 khi huấn luyện mô hình. Theo dõi kết quả.
3. Tham khảo tài liệu của MXNet và cho biết mô-đun `transforms` của Gluon còn cung cấp các phương thức tăng cường ảnh nào khác?

### 15.1.5 Thảo luận

- Tiếng Anh - MXNet<sup>296</sup>
- Tiếng Việt<sup>297</sup>

### 15.1.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Mai Hoàng Long
- Trần Yến Thy
- Lê Khắc Hùng Phúc
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật

<sup>296</sup> <https://discuss.d2l.ai/t/367>

<sup>297</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 15.2 Tinh Chỉnh

Trong các chương trước, chúng ta đã thảo luận cách huấn luyện mô hình trên tập dữ liệu Fashion-MNIST với chỉ 60,000 ảnh. Ta cũng đã nói về ImageNet, tập dữ liệu ảnh quy mô lớn được sử dụng phổ biến trong giới học thuật, với hơn 10 triệu tấm ảnh vật thể thuộc hơn 1000 hạng mục. Tuy nhiên, những tập dữ liệu ta thường gặp chỉ có kích thước đâu đó giữa hai tập này, lớn hơn MNIST nhưng nhỏ hơn ImageNet.

Giả sử ta muốn nhận diện các loại ghế khác nhau trong ảnh rồi gửi đường dẫn mua hàng đến người dùng. Một cách khả dĩ là: đầu tiên ta tìm khoảng một trăm loại ghế phổ biến, chụp một nghìn bức ảnh từ các góc máy khác nhau với mỗi loại, rồi huấn luyện mô hình phân loại trên tập dữ liệu ảnh này. Dù tập dữ liệu này lớn hơn Fashion-MNIST, thì số lượng ảnh vẫn không bằng được một phần mười của ImageNet. Điều này dẫn tới việc các mô hình phức tạp bị quá khớp khi huấn luyện trên tập dữ liệu này, dù chúng hoạt động tốt với Imagenet. Đồng thời, vì lượng dữ liệu khá hạn chế, độ chính xác của mô hình sau khi huấn luyện xong có thể không thỏa mãn được mức yêu cầu trong thực tiễn.

Để giải quyết vấn đề này, một giải pháp dễ thấy là đi thu thập thêm dữ liệu. Tuy nhiên, việc thu thập và gán nhãn dữ liệu có thể tốn rất nhiều tiền và thời gian. Ví dụ, để xây dựng được tập ImageNet, hàng triệu đô la đã được sử dụng từ nguồn tài trợ nghiên cứu. Dù vậy, gần đây chi phí thu thập dữ liệu đã giảm mạnh, nhưng điều này vẫn rất đáng lưu ý.

Một giải pháp khác là áp dụng kỹ thuật học truyền tải (*transfer learning*), mang kiến thức đã học được từ tập dữ liệu gốc áp dụng sang tập dữ liệu mục tiêu. Ví dụ, đa phần ảnh trong ImageNet không chụp ghế, nhưng những mô hình đã được huấn luyện trên ImageNet có khả năng trích xuất các đặc trưng chung của ảnh, rồi từ đó giúp nhận diện ra góc cạnh, bề mặt, hình dáng, và các kết cấu của vật thể. Các đặc trưng tương đồng này cũng sẽ có ích trong bài toán nhận diện ghế.

Trong phần này, chúng tôi giới thiệu một kỹ thuật thông dụng trong việc học truyền tải, đó là tinh chỉnh (*fine tuning*). Như minh họa trong hình Fig. 15.2.1, việc tinh chỉnh được tiến hành theo bốn bước sau đây:

1. Tiền huấn luyện một mô hình mạng nơ-ron, tức là là mô hình gốc, trên tập dữ liệu gốc (chẳng hạn tập dữ liệu ImageNet).
2. Tạo mô hình mạng nơ-ron mới gọi là mô hình mục tiêu. Mô hình này sao chép tất cả các thiết kế cũng như các tham số của mô hình gốc, ngoại trừ tầng đầu ra. Ta giả định rằng các tham số mô hình chứa tri thức đã học từ tập dữ liệu gốc và tri thức này sẽ áp dụng tương tự đối với tập dữ liệu mục tiêu. Ta cũng giả định là tầng đầu ra của mô hình gốc có liên hệ mật thiết với các nhãn của tập dữ liệu gốc và do đó không được sử dụng trong mô hình mục tiêu.
3. Thêm vào một tầng đầu ra cho mô hình mục tiêu mà kích thước của nó là số lớp của dữ liệu mục tiêu, và khởi tạo ngẫu nhiên các tham số mô hình của tầng này.
4. Huấn luyện mô hình mục tiêu trên tập dữ liệu mục tiêu, chẳng hạn như tập dữ liệu ghế. Chúng ta sẽ huấn luyện tầng đầu ra từ đầu, trong khi các tham số của tất cả các tầng còn lại được tinh chỉnh từ các tham số của mô hình gốc.

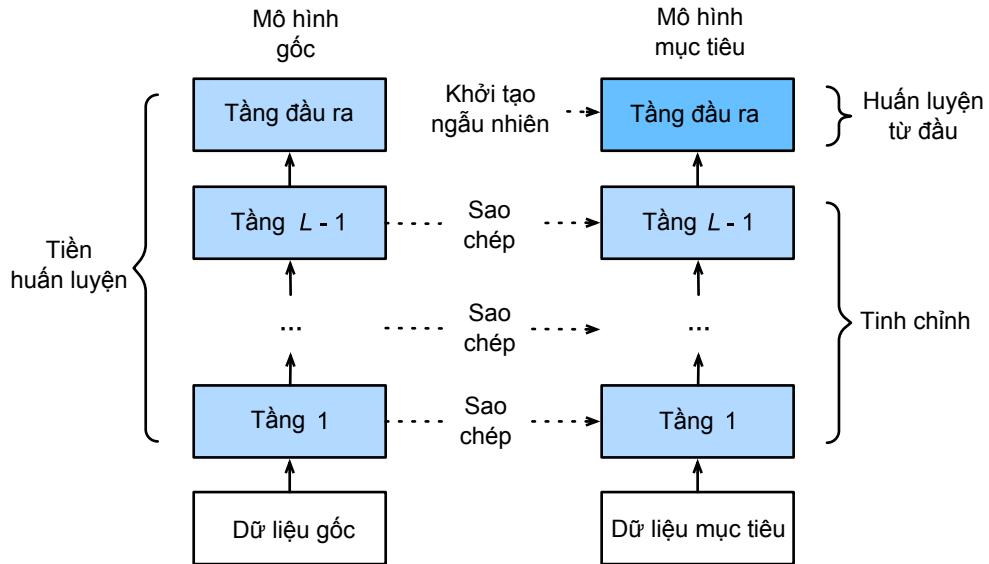


Fig. 15.2.1: Thực hiện tinh chỉnh.

### 15.2.1 Nhận dạng Xúc xích

Tiếp theo, ta sẽ dùng một ví dụ cụ thể để luyện tập đó là: nhận dạng xúc xích. Ta sẽ tinh chỉnh mô hình ResNet đã huấn luyện trên tập dữ liệu ImageNet dựa trên một tập dữ liệu nhỏ. Tập dữ liệu nhỏ này chứa hàng nghìn ảnh, trong đó sẽ có các ảnh chứa hình xúc xích. Ta sẽ sử dụng mô hình thu được từ việc tinh chỉnh để xác định một bức ảnh có chứa món ăn này hay không.

Trước tiên, ta thực hiện nhập các gói và mô-đun cần cho việc thử nghiệm. Gói `model_zoo` trong Gluon cung cấp một mô hình đã được huấn luyện sẵn phổ biến. Nếu bạn muốn lấy thêm các mô hình đã được tiền huấn luyện cho thị giác máy tính, bạn có thể tham khảo [Bộ công cụ GluonCV<sup>298</sup>](#).

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
import os

npx.set_np()
```

### Lấy dữ liệu

Tập dữ liệu xúc xích mà ta sử dụng được lấy từ internet, gồm 1,400 ảnh mẫu dương chứa xúc xích và 1,400 ảnh mẫu âm chứa các loại thức ăn khác. 1,000 ảnh thuộc nhiều lớp khác nhau được sử dụng để huấn luyện và phần còn lại được dùng để kiểm tra.

Đầu tiên ta tải tập dữ liệu nén và thu được 2 tập tin `hotdog/train` và `hotdog/test`. Cả hai đều có hai tập tin phụ `hotdog` và `not-hotdog` chứa các ảnh với phân loại tương ứng.

<sup>298</sup> <https://gluon-cv.mxnet.io>

```

#@save
d2l.DATA_HUB['hotdog'] = (d2l.DATA_URL+'hotdog.zip',
                           'fba480ffa8aa7e0feb511d181409f899b9baa5')

data_dir = d2l.download_extract('hotdog')

```

Ta tạo hai thực thể `ImageFolderDataset` để đọc toàn bộ các tệp ảnh trong tập huấn luyện và tập kiểm tra.

```

train_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'train'))
test_imgs = gluon.data.vision.ImageFolderDataset(
    os.path.join(data_dir, 'test'))

```

Dưới đây là 8 mẫu dương tính đầu tiên và 8 mẫu âm cuối cùng. Bạn có thể thấy những hình ảnh có nhiều kích thước và tỷ lệ khác nhau.

```

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 huấn luyện, chúng ta cắt ảnh với kích thước và tỷ lệ ngẫu nhiên sau đó biến đổi tỷ lệ để có chiều dài và chiều rộng 224 pixel. Khi kiểm tra, ta biến đổi tỷ lệ chiều dài và chiều rộng của ảnh về kích thước 256 pixel, sau đó cắt ở vùng trung tâm để thu được ảnh có chiều dài và rộng là 224 pixel để làm đầu vào cho mô hình.Thêm vào đó, chúng ta chuẩn hóa các giá trị của ba kênh màu RGB (red, green, blue). Tất cả giá trị trên ảnh sẽ được trừ đi giá trị trung bình trên kênh màu, sau đó được chia cho độ lệch chuẩn của chúng để thu được ảnh đã qua xử lý.

```

# We specify the mean and variance of the three RGB channels to normalize the
# image channel
normalize = gluon.data.vision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.RandomResizedCrop(224),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    normalize])

test_augs = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    normalize])

```

## Định nghĩa và Khởi tạo Mô hình

Ta sử dụng ResNet-18 đã được tiền huấn luyện trên tập dữ liệu ImageNet làm mô hình gốc. Ở đây ta chỉ rõ pretrained=True để tự động tải xuống và nạp các tham số mô hình được tiền huấn luyện. Trong lần sử dụng đầu tiên, các tham số mô hình cần được tải xuống từ Internet.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
```

Mô hình gốc được huấn luyện sẵn bao gồm hai biến thành viên: features và output. features bao gồm tất cả các tầng của mô hình ngoại trừ tầng đầu ra, và output chính là tầng đầu ra của mô hình đó. Mục đích chính của việc phân chia này là để tạo điều kiện cho việc tinh chỉnh các tham số của tất cả các tầng của mô hình trừ tầng đầu ra. Biến thành viên output của mô hình gốc là một tầng kết nối đầy đủ. Nó biến đổi đầu ra của tầng gộp trung bình toàn cục thành 1000 lớp đầu ra trên tập dữ liệu ImageNet như dưới đây.

```
pretrained_net.output
```

Sau đó ta xây dựng một mạng nơ-ron mới làm mô hình mục tiêu. Mạng này được định nghĩa giống như mô hình tiền huấn luyện gốc, tuy nhiên số đầu ra cuối cùng bằng với số hạng mục trong tập dữ liệu mục tiêu. Ở đoạn mã phía dưới, các tham số mô hình trong biến thành viên features của mô hình mục tiêu finetune\_net được khởi tạo giống như các tham số của các tầng tương ứng trong mô hình gốc. Các tham số mô hình trong features đã được huấn luyện trên tập dữ liệu ImageNet nên đã tương đối tốt. Vì vậy thường thì ta chỉ cần sử dụng tốc độ học nhỏ để “tinh chỉnh” các tham số trên. Ngược lại, các tham số mô hình trong biến thành viên output được khởi tạo ngẫu nhiên và thường yêu cầu tốc độ học lớn hơn nhiều để học lại từ đầu. Giả sử rằng tốc độ học trong đối tượng Trainer là  $\eta$  thì ta sử dụng tốc độ học là  $10\eta$  để cập nhật tham số mô hình trong biến thành viên output.

```
finetune_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
finetune_net.features = pretrained_net.features
finetune_net.output.initialize(init.Xavier())
# The model parameters in output will be updated using a learning rate ten
# times greater
finetune_net.output.collect_params().setattr('lr_mult', 10)
```

## Tinh chỉnh Mô hình

Đầu tiên, ta định nghĩa hàm huấn luyện tinh chỉnh train\_fine\_tuning để có thể gọi nhiều lần.

```
def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5):
    train_iter = gluon.data.DataLoader(
        train_imgs.transform_first(train_augs), batch_size, shuffle=True)
    test_iter = gluon.data.DataLoader(
        test_imgs.transform_first(test_augs), batch_size)
    devices = d2l.try_all_gpus()
    net.collect_params().reset_ctx(devices)
    net.hybridize()
    loss = gluon.loss.SoftmaxCrossEntropyLoss()
    trainer = gluon.Trainer(net.collect_params(), 'sgd', {
        'learning_rate': learning_rate, 'wd': 0.001})
```

(continues on next page)

```
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs,
                devices)
```

Ta gán giá trị tốc độ học nhỏ cho đối tượng Trainer, ví dụ như 0.01, để tinh chỉnh các tham số mô hình huấn luyện sẵn. Như đề cập ở trên, ta sẽ sử dụng tốc độ học gấp 10 lần để huấn luyện từ đầu các tham số của tầng đầu ra mô hình mục tiêu.

```
train_fine_tuning(finetune_net, 0.01)
```

Để so sánh, ta định nghĩa một mô hình y hệt, tuy nhiên tất cả các tham số mô hình của nó được khởi tạo một cách ngẫu nhiên. Do toàn bộ mô hình cần được huấn luyện từ đầu, ta có thể sử dụng tốc độ học lớn hơn.

```
scratch_net = gluon.model_zoo.vision.resnet18_v2(classes=2)
scratch_net.initialize(init=init.Xavier())
train_fine_tuning(scratch_net, 0.1)
```

Như bạn có thể thấy, với số epoch như nhau, mô hình tinh chỉnh có giá trị precision cao hơn. Lý do là vì các tham số có giá trị khởi tạo ban đầu tốt hơn.

### 15.2.2 Tóm tắt

- Học truyền tải chuyển kiến thức học được từ tập dữ liệu gốc sang tập dữ liệu mục tiêu. Tinh chỉnh là một kỹ thuật phổ biến trong học truyền tải.
- Mô hình mục tiêu tái tạo toàn bộ thiết kế mô hình và các tham số của mô hình gốc, ngoại trừ tầng đầu ra, và tinh chỉnh các tham số này dựa vào tập dữ liệu mục tiêu. Ngược lại, tầng đầu ra của mô hình mục tiêu cần được huấn luyện lại từ đầu.
- Thông thường việc tinh chỉnh các tham số sử dụng tốc độ học nhỏ, trong khi việc huấn luyện lại tầng đầu ra từ đầu có thể sử dụng tốc độ học lớn hơn.

### 15.2.3 Bài tập

1. Liên tục tăng tốc độ học của finetune\_net. Giá trị precision của mô hình thay đổi như thế nào?
2. Tiếp tục điều chỉnh các siêu tham số của finetune\_net và scratch\_net trong thí nghiệm so sánh ở trên. Liệu giá trị precision của chúng vẫn khác nhau hay không?
3. Gán các tham số của finetune\_net.features bằng các tham số của mô hình gốc và không cập nhật chúng suốt quá trình huấn luyện. Điều gì sẽ xảy ra? Bạn có thể sử dụng đoạn mã sau.

```
finetune_net.features.collect_params().setattr('grad_req', 'null')
```

4. Thật ra, lớp “hotdog” có trong tập dữ liệu ImageNet. Các trọng số tương ứng của nó trong tầng đầu ra có thể thu được thông qua việc sử dụng đoạn mã sau. Ta có thể sử dụng các tham số này như thế nào?

```
weight = pretrained_net.output.weight
hotdog_w = np.split(weight.data(), 1000, axis=0)[713]
hotdog_w.shape
```

#### 15.2.4 Thảo luận

- Tiếng Anh - MXNet<sup>299</sup>
- Tiếng Việt<sup>300</sup>

#### 15.2.5 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Mai Sơn Hải
- Phạm Minh Đức
- Phạm Hồng Vinh
- Nguyễn Thanh Hòa
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Nguyễn Mai Hoàng Long

### 15.3 Phát hiện Vật thể và Khoanh vùng Đối tượng (Khung chứa)

Ở phần trước, chúng ta đã giới thiệu nhiều loại mô hình dùng cho phân loại ảnh. Trong tác vụ phân loại ảnh, ta giả định chỉ có duy nhất một đối tượng trong ảnh và ta chỉ tập trung xác định nó thuộc về nhóm nào. Tuy nhiên, ở nhiều tình huống cùng lúc sẽ có nhiều đối tượng trong ảnh mà ta quan tâm. Ta không chỉ muốn phân loại chúng mà còn muốn xác định vị trí cụ thể của chúng ở trong ảnh. Trong lĩnh vực thị giác máy tính, những tác vụ như thế được gọi là phát hiện vật thể (hoặc nhận dạng vật thể).

Phát hiện vật thể được sử dụng rộng rãi trong nhiều lĩnh vực. Chẳng hạn, trong công nghệ xe tự hành, ta cần lên lộ trình bằng cách xác định các vị trí của phương tiện di chuyển, người đi đường, đường xá và các vật cản trong các ảnh được thu về từ video. Robot cần thực hiện kiểu tác vụ này để phát hiện các đối tượng mà chúng quan tâm. Hay các hệ thống an ninh cần phát hiện các mục tiêu bất thường, ví dụ như các đối tượng xâm nhập bất hợp pháp hoặc bom mìn.

<sup>299</sup> <https://discuss.d2l.ai/t/368>

<sup>300</sup> <https://forum.machinelearningcoban.com/c/d2l>

Trong các phần tiếp theo, chúng tôi sẽ giới thiệu nhiều mô hình học sâu dùng để phát hiện vật thể. Trước hết, ta nên bàn qua về khái niệm vị trí vật thể. Đầu tiên, ta hãy nhập các gói và mô-đun cần thiết cho việc thử nghiệm.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import image, npx

npx.set_np()
```

Kế tiếp, ta nạp các ảnh mẫu sẽ sử dụng trong phần này. Ta có thể thấy trong hình là một con chó ở bên trái và một con mèo ở bên phải. Chúng là hai đối tượng chính trong ảnh này.

```
d2l.set_figsize()
img = image.imread('../img/catdog.jpg').asnumpy()
d2l.plt.imshow(img);
```

### 15.3.1 Khung chứa

Để phát hiện vật thể, ta thường sử dụng khung chứa để mô tả vị trí của mục tiêu. Khung chứa là một khung hình chữ nhật có thể được xác định bởi hai tọa độ: tọa độ  $x, y$  góc trên bên trái và tọa độ  $x, y$  góc dưới bên phải của khung hình chữ nhật. Ta có thể định nghĩa các khung chứa của con chó và con mèo trong ảnh dựa vào thông tin tọa độ của ảnh trên. Gốc tọa độ của ảnh trên là góc trên bên trái của ảnh, chiều sang phải và xuống dưới lần lượt là chiều dương của trục  $x$  và trục  $y$ .

```
# bbox is the abbreviation for bounding box
dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 112, 655, 493]
```

Ta có thể vẽ khung chứa ngay trên ảnh để kiểm tra tính chính xác của nó. Trước khi vẽ khung, ta định nghĩa hàm hỗ trợ `bbox_to_rect`. Nó biểu diễn khung chứa theo đúng định dạng khung chứa của `matplotlib`.

```
#@save
def bbox_to_rect(bbox, color):
    """Convert bounding box to matplotlib format."""
    # Convert the bounding box (top-left x, top-left y, bottom-right x,
    # bottom-right y) format to 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 vẽ khung chứa lên ảnh, có thể thấy rằng phần chính của mục tiêu về cơ bản là nằm trong khung chứa.

```
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'));
```

### 15.3.2 Tóm tắt

Để phát hiện vật thể, ta không chỉ cần xác định tất cả đối tượng mong muốn trong ảnh mà còn cả vị trí của chúng. Các vị trí thường được biểu diễn qua các khung chứa hình chữ nhật.

### 15.3.3 Bài tập

Tìm một vài ảnh và thử dán nhãn một khung chứa bao quanh mục tiêu. So sánh sự khác nhau giữa thời gian cần để dán nhãn các khung chứa và dán nhãn các lớp hạng mục.

### 15.3.4 Thảo luận

- Tiếng Anh - MXNet<sup>301</sup>
- Tiếng Việt<sup>302</sup>

### 15.3.5 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật

## 15.4 Khung neo

Các giải thuật phát hiện vật thể thường lấy mẫu ở rất nhiều vùng của ảnh đầu vào, rồi xác định xem các vùng đó có chứa đối tượng cần quan tâm hay không, và điều chỉnh biên của vùng lấy mẫu này để dự đoán khung chứa nhãn gốc của đối tượng một cách chính xác hơn. Các mô hình khác nhau có thể dùng các phương pháp lấy mẫu vùng ảnh khác nhau. Ở đây, chúng tôi sẽ giới thiệu một phương pháp đó là: tạo ra nhiều khung chứa với kích thước và tỷ lệ cạnh khác nhau với tâm trên từng điểm ảnh. Các khung chứa đó được gọi là các khung neo. Chúng ta sẽ thực hành phát hiện vật thể dựa trên các khung neo ở các phần sau.

Trước tiên, hãy nhập các gói và mô-đun cần thiết cho mục này. Tại đây, ta đã chỉnh sửa độ chính xác khi in số thực của Numpy. Vì ta đang gọi hàm in của Numpy khi in các tensor, nên các tensor số thực dấu phẩy động sẽ được in ra dưới dạng súc tích hơn.

<sup>301</sup> <https://discuss.d2l.ai/t/369>

<sup>302</sup> <https://forum.machinelearningcoban.com/c/d2l>

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx

np.set_printoptions(2)
npx.set_np()
```

### 15.4.1 Sinh nhiều Khung neo

Giả sử ảnh đầu vào có chiều cao  $h$  và chiều rộng  $w$ . Ta sinh ra các khung neo với kích thước khác nhau có tâm tại mỗi điểm ảnh. Giả sử kích thước này  $s \in (0, 1]$ , tỷ lệ cạnh là  $r > 0$ , chiều rộng và chiều cao của khung neo lần lượt là  $ws\sqrt{r}$  và  $hs\sqrt{r}$ . Với một vị trí tâm cho trước, ta xác định được khung neo với chiều cao và chiều rộng như trên.

Dưới đây, ta thiết lập một tập kích thước  $s_1, \dots, s_n$  và một tập tỷ lệ khung  $r_1, \dots, r_m$ . Nếu ta dùng tổ hợp tất cả các kích thước và tỷ lệ khung với mỗi điểm ảnh làm một tâm, ảnh đầu vào sẽ có tổng cộng  $whnm$  khung neo. Mặc dù các khung chứa chuẩn đối tượng có thể sẽ nằm trong số đó, nhưng độ phức tạp tính toán này thường quá cao. Do đó, ta thường chỉ chú ý tới tổ hợp chứa  $s_1$  kích thước hoặc  $r_1$  tỷ lệ khung như sau:

$$(s_1, r_1), (s_1, r_2), \dots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \dots, (s_n, r_1). \quad (15.4.1)$$

Ở trên, số khung neo có tâm trên cùng một điểm ảnh là  $n + m - 1$ . Đối với toàn bộ bức ảnh đầu vào, ta sẽ sinh ra tổng cộng  $wh(n + m - 1)$  khung neo.

Phương pháp sinh khung neo ở trên được lập trình sẵn trong hàm `multibox_prior`. Ta chỉ cần thiết lập đầu vào, tập các kích thước và tập các tỉ số cạnh, rồi hàm này sẽ trả về tất cả các khung neo như mong muốn.

```
img = image.imread('../img/catdog.jpg').asnumpy()
h, w = img.shape[0:2]

print(h, w)
X = np.random.uniform(size=(1, 3, h, w)) # Construct input data
Y = npx.multibox_prior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
Y.shape
```

Ta có thể thấy rằng kích thước của khung neo được trả về ở biến `y` là (kích thước batch, số khung neo, 4). Sau khi thay đổi kích thước của `y` thành (chiều cao ảnh, chiều rộng ảnh, số khung neo có tâm trên cùng một điểm ảnh, 4), ta sẽ thu được tất cả các khung neo với tâm ở một vị trí điểm ảnh nhất định. Trong phần ví dụ dưới đây, ta truy xuất khung neo đầu tiên có tâm tại vị trí (250, 250). Nó có bốn phần tử: tọa độ trực  $x, y$  ở góc trên bên trái và tọa độ trực  $x, y$  ở góc dưới bên phải của khung neo. Tọa độ của các trực  $x$  và  $y$  được chia lần lượt cho chiều rộng và độ cao của ảnh, do đó giá trị của chúng sẽ nằm trong khoảng 0 và 1.

```
boxes = Y.reshape(h, w, 5, 4)
boxes[250, 250, 0, :]
```

Để mô tả tất cả các khung neo có tâm trên cùng một điểm của bức ảnh, trước hết ta sẽ định nghĩa hàm `show_bboxes` để vẽ nhóm khung chứa trên ảnh này.

```

#@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.asnumpy(), 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 trục  $x$  và  $y$  trong biến `bboxes` đã được chia lần lượt cho chiều rộng và chiều cao của ảnh. Khi vẽ ảnh, ta cần khôi phục các giá trị tọa độ gốc của các khung neo và xác định biến `bbox_scale`. Lúc này, ta có thể vẽ tất cả các khung neo có tâm tại vị trí (250, 250) của bức ảnh này. Như bạn có thể thấy, khung neo màu xanh dương với kích thước 0.75 và tỉ số cạnh 1 sẽ bao quanh khá tốt chú chó trong hình này.

```

d2l.set_figsize()
bbox_scale = np.array((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'])

```

### 15.4.2 Giao trên Hợp

Chúng ta chỉ mới đề cập rằng khung neo đó bao quanh tốt chú chó trong ảnh. Nếu ta biết khung chứa nhãn gốc của đối tượng, làm thế nào để định lượng được “mức độ tốt” ở đây? Một phương pháp đơn giản là đo điểm tương đồng giữa các khung neo và khung chứa nhãn gốc. Và ta biết rằng hệ số Jaccard có thể đo lường sự tương đồng giữa hai tập dữ liệu. Với hai tập hợp  $\mathcal{A}$  và  $\mathcal{B}$ , chỉ số Jaccard của chúng là kích thước của phần giao trên kích thước của phần hợp:

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}. \quad (15.4.2)$$

Trong thực tế, chúng ta có thể coi vùng điểm ảnh trong khung chứa là một tập hợp các điểm ảnh. Theo cách này, ta có thể đo lường được tính tương đồng của hai khung chứa bằng hệ số Jaccard của các tập điểm ảnh tương ứng. Khi đo sự tương đồng giữa hai khung chứa, hệ số Jaccard thường được gọi là Giao trên Hợp (*Intersection over Union - IoU*), tức tỷ lệ giữa vùng giao nhau và vùng kết hợp của hai khung chứa ảnh, được thể hiện trong Fig. 15.4.1. Miền giá trị của IoU nằm trong khoảng từ 0 đến 1: giá trị 0 có nghĩa là không có điểm ảnh nào giao nhau giữa hai khung chứa, trong khi đó giá trị 1 chỉ ra rằng hai khung chứa ấy hoàn toàn trùng nhau.

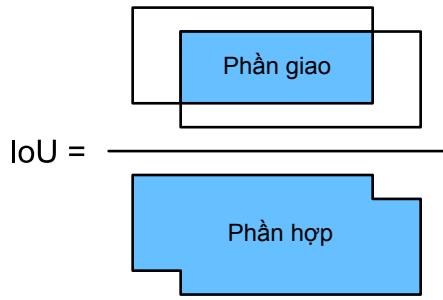


Fig. 15.4.1: IoU là tỷ lệ giữa vùng giao trên vùng hợp của hai khung chứa.

Trong phần còn lại, chúng ta sẽ dùng IoU để đo sự tương đồng giữa các khung neo với khung chứa nhãn gốc và giữa các khung neo với nhau.

### 15.4.3 Gán nhãn Khung neo trong tập Huấn luyện

Trong tập huấn luyện, chúng ta xem mỗi khung neo là một mẫu huấn luyện. Để huấn luyện mô hình phát hiện đối tượng, chúng ta cần đánh dấu hai loại nhãn cho mỗi khung neo: thứ nhất là hạng mục (*category*) của đối tượng trong khung neo, thứ hai là độ dời tương đối (*offset*) của khung chứa nhãn gốc so với khung neo. Trong việc phát hiện đối tượng, trước tiên ta tạo ra nhiều khung neo, dự đoán các hạng mục và độ dời cho từng khung neo, hiệu chỉnh vị trí của chúng dựa theo độ lệch dự kiến để có được những khung chứa và sau cùng là lọc ra các khung chứa mà cần được dự đoán.

Chúng ta biết rằng, trong tập huấn luyện phát hiện đối tượng, mỗi hình ảnh được gán nhãn với vị trí của khung chứa nhãn gốc và hạng mục của đối tượng. Ta gán nhãn cho các khung neo sau khi tạo chủ yếu dựa vào thông tin vị trí và hạng mục của các khung chứa nhãn gốc tương đồng với các khung neo đó. Vậy làm thế nào để gán các khung chứa nhãn gốc cho những khung neo tương đồng với chúng?

Giả sử rằng các khung neo trên ảnh là  $A_1, A_2, \dots, A_{n_a}$  và các khung chứa nhãn gốc là  $B_1, B_2, \dots, B_{n_b}$  và  $n_a \geq n_b$ . Xây dựng ma trận  $\mathbf{X} \in \mathbb{R}^{n_a \times n_b}$ , trong đó mỗi phần tử  $x_{ij}$  trong hàng  $i^{\text{th}}$  và cột  $j^{\text{th}}$  là hệ số IoU của khung neo  $A_i$  so với khung chứa nhãn gốc  $B_j$ . Đầu tiên, ta tìm ra phần tử lớn nhất trong ma trận  $\mathbf{X}$  rồi lưu lại chỉ mục hàng và cột của phần tử đó là  $i_1, j_1$ , rồi gán khung chứa nhãn gốc  $B_{j_1}$  cho khung neo  $A_{i_1}$ . Rõ ràng, khung neo  $A_{i_1}$  và khung chứa nhãn gốc  $B_{j_1}$  có độ tương đồng cao nhất trong số tất cả các cặp “khung neo-khung chứa nhãn gốc”. Tiếp theo, loại bỏ các phần tử trong hàng  $i_1$  và cột  $j_1$  trong ma trận  $\mathbf{X}$ . Tìm phần tử lớn nhất trong các phần tử còn lại trong ma trận  $\mathbf{X}$  rồi cũng lưu lại chỉ mục hàng và cột của phần tử đó là  $i_2, j_2$ . Chúng ta gán khung chứa nhãn gốc  $B_{j_2}$  cho khung neo  $A_{i_2}$  và sau đó loại bỏ mọi phần tử tại hàng  $i_2$  và cột  $j_2$  trong ma trận  $\mathbf{X}$ . Như vậy, tại thời điểm này thì các phần tử trong hai hàng và hai cột của ma trận  $\mathbf{X}$  đã bị loại bỏ.

Ta tiến hành việc này cho đến khi các phần tử ở cột  $n_b$  trong ma trận  $\mathbf{X}$  đều bị loại bỏ. Tại thời điểm này, chúng ta đều đã gán  $n_b$  khung chứa nhãn gốc cho  $n_b$  khung neo. Tiếp đến, chỉ việc duyệt qua  $n_a - n_b$  khung neo còn lại. Với khung neo  $A_i$ , ta cần tìm ra khung chứa nhãn gốc  $B_j$  sao cho khung chứa ấy có hệ số IoU so với  $A_i$  là lớn nhất trên hàng  $i$  của ma trận  $\mathbf{X}$ , và chỉ gán khung chứa nhãn gốc  $B_j$  cho khung neo  $A_i$  khi mà hệ số IoU lớn hơn một ngưỡng cho trước.

Như mô tả ở Fig. 15.4.2 (trái), giả sử giá trị lớn nhất của ma trận  $\mathbf{X}$  là  $x_{23}$ , ta gán khung chứa nhãn gốc  $B_3$  cho khung neo  $A_2$ . Tiếp theo ta loại bỏ tất cả các giá trị ở hàng 2 và cột 3 của ma trận, tìm phần tử lớn nhất  $x_{71}$  của phần ma trận còn lại và gán khung chứa nhãn gốc  $B_1$  cho khung neo  $A_7$ . Sau đó, như trong Fig. 15.4.2 (giữa), ta loại bỏ tất cả các giá trị ở hàng 7 và cột 1 của ma trận, tìm

phần tử lớn nhất  $x_{54}$  của phần ma trận còn lại và gán khung chứa nhãn gốc  $B_4$  cho khung neo  $A_5$ . Cuối cùng, trong Fig. 15.4.2 (phải), ta loại bỏ tất cả các giá trị ở hàng 5 và cột 4 của ma trận, tìm phần tử lớn nhất  $x_{92}$  của phần ma trận còn lại và gán khung chứa nhãn gốc  $B_2$  cho khung neo  $A_9$ . Sau đó ta chỉ cần duyệt các khung neo còn lại  $A_1, A_3, A_4, A_6, A_8$  và dựa vào mức ngưỡng để quyết định có gán khung chứa nhãn gốc cho các khung neo này không.

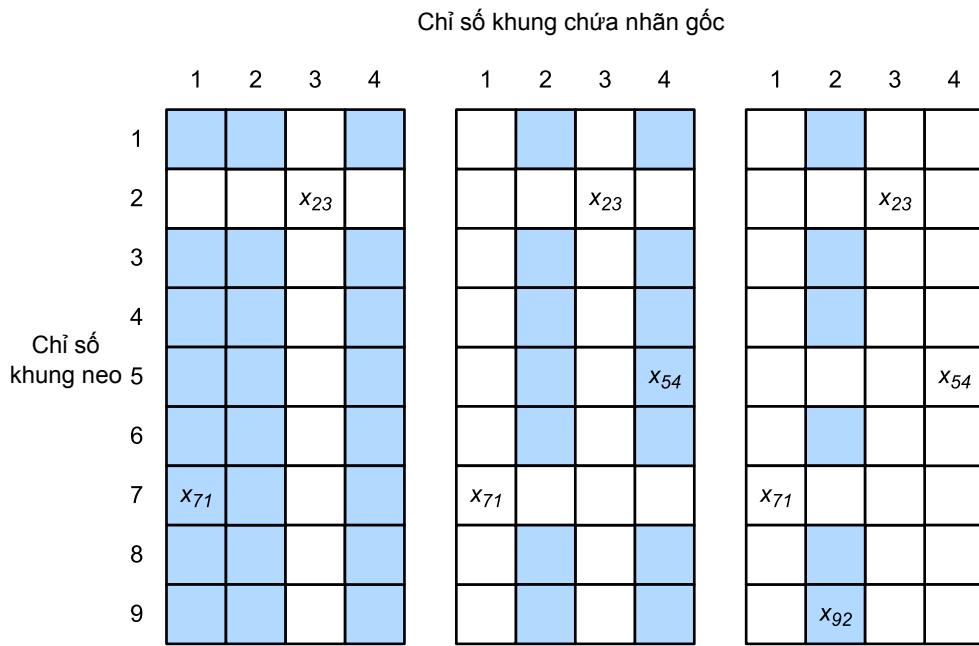


Fig. 15.4.2: Gán khung chứa nhãn gốc cho các khung neo.

Giờ ta có thể gán nhãn hạng mục và độ dời cho các khung neo. Nếu khung neo  $A$  được gán khung chứa nhãn gốc  $B$  thì khung neo  $A$  sẽ có cùng hạng mục với  $B$ . Độ dời của khung neo  $A$  được đặt dựa theo vị trí tương đối của tọa độ tâm của  $B$  và  $A$  cũng như kích thước tương đối của hai khung. Do vị trí và kích thước của các khung trong tập dữ liệu thường khá đa dạng, các vị trí và kích thước tương đối này thường yêu cầu một số phép biến đổi đặc biệt sao cho phân phối của giá trị độ dời trở nên đều và dễ khớp hơn. Giả sử tọa độ tâm của khung neo  $A$  và khung chứa nhãn gốc  $B$  được gán cho nó là  $(x_a, y_a), (x_b, y_b)$ , chiều rộng của  $A$  và  $B$  lần lượt là  $w_a, w_b$ , và chiều cao lần lượt là  $h_a, h_b$ . Đối với trường hợp này, một kỹ thuật phổ biến là gán nhãn độ dời của  $A$  như sau

$$\left( \frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right), \quad (15.4.3)$$

Giá trị mặc định của các hằng số là  $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, v\sigma_w = \sigma_h = 0.2$ . Nếu một khung neo không được gán cho một khung chứa nhãn gốc, ta chỉ cần gán hạng mục của khung neo này là nền. Các khung neo có hạng mục là nền thường được gọi là khung neo âm, và tất cả các khung neo còn lại được gọi là khung neo dương.

Dưới đây chúng tôi sẽ giải thích chi tiết một ví dụ. Ta định nghĩa các khung chứa nhãn gốc cho con mèo và con chó trong ảnh đã đọc, trong đó phần tử đầu tiên là hạng mục (0 là chó, 1 là mèo) và bốn phần tử còn lại là các tọa độ  $x, y$  của góc trên bên trái và tọa độ  $x, y$  của góc dưới bên phải (dài giá trị nằm trong khoảng từ 0 đến 1). Ở đây ta khởi tạo năm khung neo bằng tọa độ của góc trên bên trái và góc dưới bên phải để gán nhãn, được kí hiệu lần lượt là  $A_0, \dots, A_4$  (chỉ số trong chương trình bắt đầu từ 0). Đầu tiên, ta vẽ vị trí của các khung neo này và các khung chứa nhãn gốc vào ảnh.

```

ground_truth = np.array([[0, 0.1, 0.08, 0.52, 0.92],
                       [1, 0.55, 0.2, 0.9, 0.88]])
anchors = np.array([[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']);

```

Ta có thể gán hạng mục và độ dời cho các khung neo này bằng cách sử dụng hàm `multibox_target`. Hàm này đặt hạng mục nền bằng 0 và tăng chỉ số lên 1 với mỗi hạng mục mục tiêu (1 là chó và 2 là mèo). Ta thêm chiều mẫu vào các tensor chứa khung neo và khung chứa nhãn gốc ở ví dụ trên và khởi tạo kết quả dự đoán ngẫu nhiên với kích thước (kích thước batch, số hạng mục tính cả nền, số khung neo) bằng cách sử dụng hàm `expand_dims`.

```

labels = npx.multibox_target(np.expand_dims(anchors, axis=0),
                             np.expand_dims(ground_truth, axis=0),
                             np.zeros((1, 3, 5)))

```

Có ba phần tử trong kết quả trả về, tất cả đều theo định dạng tensor. Phần tử thứ ba là hạng mục được gán nhãn cho khung neo.

```
labels[2]
```

Ta phân tích các hạng mục được gán nhãn này dựa theo vị trí của khung neo và khung chứa nhãn gốc trong ảnh. Đầu tiên, trong tất cả các cặp “khung neo—khung chứa nhãn gốc”, giá trị IoU của khung neo  $A_4$  đối với khung chứa nhãn gốc mèo là lớn nhất, vậy hạng mục của khung neo  $A_4$  được gán là mèo. Nếu ta không xét khung neo  $A_4$  hoặc khung chứa nhãn gốc mèo, trong các cặp “khung neo—khung chứa nhãn gốc” còn lại, cặp với giá trị IoU lớn nhất là khung neo  $A_1$  và khung chứa nhãn gốc chó, vậy hạng mục của khung neo  $A_1$  được gán là chó. Tiếp theo ta xét ba khung neo còn lại chưa được gán nhãn. Hạng mục của khung chứa nhãn gốc có giá trị IoU lớn nhất với khung neo  $A_0$  là chó, tuy nhiên giá trị IoU này lại nhỏ hơn mức ngưỡng (mặc định là 0.5), do đó khung neo này được gán nhãn là nền; hạng mục của khung chứa nhãn gốc có giá trị IoU lớn nhất với khung neo  $A_2$  là mèo và giá trị IoU này lớn hơn mức ngưỡng, do đó khung neo này được gán nhãn là mèo; hạng mục của khung chứa nhãn gốc có giá trị IoU lớn nhất với khung neo  $A_3$  là mèo, tuy nhiên giá trị IoU này lại nhỏ hơn mức ngưỡng, do đó khung neo này được gán nhãn là nền.

Phần tử thứ hai trong giá trị trả về là một biến mặt nạ (*mask variable*), với kích thước (kích thước batch, số khung neo nhân bốn). Các phần tử trong biến mặt nạ tương ứng một - một với bốn giá trị độ dời của mỗi khung neo. Do ta không cần quan tâm đến việc nhận diện nền nên độ dời thuộc lớp âm không ảnh hưởng đến hàm mục tiêu. Qua phép nhân theo từng phần tử, các giá trị 0 trong biến mặt nạ có thể lọc ra các độ dời thuộc lớp âm trước khi tính hàm mục tiêu.

```
labels[1]
```

Phần tử đầu tiên trong giá trị trả về là bốn giá trị độ dời được gán nhãn cho mỗi khung neo, với giá trị độ dời của các khung neo thuộc lớp âm được gán nhãn là 0.

```
labels[0]
```

#### 15.4.4 Khung chứa khi Dự đoán

Trong giai đoạn dự đoán, đầu tiên ta tạo ra nhiều khung neo cho bức ảnh, sau đó dự đoán hạng mục và độ dời của từng khung neo. Tiếp theo, ta thu được những khung chứa dự đoán dựa trên các khung neo và độ dời dự đoán của chúng. Khi tồn tại nhiều khung neo, nhiều khung chứa dự đoán tương tự nhau có thể được tạo ra cho cùng một mục tiêu. Để đơn giản hóa kết quả, ta có thể loại bỏ những khung chứa dự đoán giống nhau. Một phương pháp thường được sử dụng là triệt phi cực đại (*non-maximum suppression - NMS*).

Hãy cùng xem cách NMS hoạt động. Đối với khung chứa dự đoán  $B$ , mô hình sẽ dự đoán xác suất cho từng hạng mục. Giả sử rằng xác suất dự đoán lớn nhất là  $p$ , hạng mục tương ứng với xác suất này sẽ là hạng mục dự đoán của  $B$ . Ta gọi  $p$  là độ tin cậy (*confidence level*) của khung chứa dự đoán  $B$ . Trên cùng một bức ảnh, ta sắp xếp các khung chứa dự đoán không phải là nền theo thứ tự giảm dần độ tin cậy, thu được danh sách  $L$ . Ta chọn ra khung chứa dự đoán có mức tin cậy cao nhất  $B_1$  từ  $L$  để làm chuẩn so sánh và loại bỏ tất cả khung chứa dự đoán “không chuẩn” có hệ số IoU với khung chứa  $B_1$  lớn hơn một ngưỡng nhất định khỏi danh sách  $L$ . Mức ngưỡng này là một siêu tham số được định trước. Tại thời điểm này,  $L$  chỉ còn khung chứa dự đoán có độ tin cậy cao nhất sau khi đã loại bỏ những khung chứa giống nó.

Sau đó, ta chọn tiếp khung chứa dự đoán  $B_2$  có độ tin cậy cao thứ hai trong  $L$  để làm chuẩn so sánh, và loại bỏ tất cả khung chứa dự đoán “không chuẩn” khác có hệ số IoU so với khung chứa  $B_2$  lớn hơn một ngưỡng nhất định khỏi  $L$ . Ta sẽ lặp lại quy trình này cho đến khi tất cả khung chứa dự đoán trong  $L$  đã được sử dụng làm chuẩn so sánh. Lúc này, IoU của bất cứ cặp khung chứa dự đoán nào trong  $L$  đều nhỏ hơn ngưỡng cho trước. Cuối cùng, đầu ra sẽ là mọi khung chứa dự đoán còn lại trong  $L$ .

Tiếp theo, hãy xem xét một ví dụ chi tiết. Trước tiên, ta tạo bốn khung neo. Để đơn giản hóa vấn đề, ta giả định rằng độ dời dự đoán đều bằng 0, nghĩa là các khung chứa dự đoán đều là các khung neo. Cuối cùng, ta định ra một xác suất dự đoán cho từng lớp.

```
anchors = np.array([[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 = np.array([0] * anchors.size)
cls_probs = np.array([[0] * 4, # Predicted probability for background
[0.9, 0.8, 0.7, 0.1], # Predicted probability for dog
[0.1, 0.2, 0.3, 0.9]]) # Predicted probability for cat
```

In các khung chứa dự đoán cùng với độ tin cậy trên 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'])
```

Ta dùng hàm `multibox_detection` để thực hiện triệt phi cực đại và đặt ngưỡng là 0.5. Hàm này tạo thêm chiều mẫu trong tensor đầu vào. Ta có thể thấy kích thước của kết quả trả về là (kích thước batch, số lượng khung neo, 6). 6 phần tử của từng hàng biểu diễn thông tin đầu ra của một khung chứa dự đoán. Phần tử đầu tiên là chỉ số của hạng mục dự đoán, bắt đầu từ 0 (0 là chó, 1 là mèo). Giá trị -1 cho biết đó là nền hoặc khung bị loại bỏ bởi triệt phi cực đại. Phần tử thứ hai chính là độ tin cậy của khung chứa dự đoán. Bốn phần tử còn lại là các tọa độ  $x, y$  của góc trên bên trái và góc dưới bên phải của khung chứa dự đoán (miền giá trị nằm trong khoảng từ 0 đến 1).

```
output = npx.multibox_detection(
    np.expand_dims(cls_probs, axis=0),
```

(continues on next page)

```

np.expand_dims(offset_preds, axis=0),
np.expand_dims(anchors, axis=0),
nms_threshold=0.5)
output

```

Ta loại bỏ các khung chứa dự đoán có giá trị -1 rồi trực quan hóa các kết quả còn được giữ lại sau khi triệt phi cực đại.

```

fig = d2l.plt.imshow(img)
for i in output[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=' , 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [np.array(i[2:]) * bbox_scale], label)

```

Trong thực tế, ta có thể loại bỏ các khung chứa dự đoán có mức độ tin cậy thấp trước khi thực hiện triệt phi cực đại để giảm bớt chi phí tính toán. Ta cũng có thể lọc các đầu ra sau khi triệt phi cực đại, ví dụ, bằng cách chỉ giữ lại những kết quả có độ tin cậy cao để làm đầu ra cuối cùng.

#### 15.4.5 Tóm tắt

- Chúng ta tạo ra nhiều khung neo với nhiều kích thước và tỷ lệ khác nhau, bao quanh từng điểm ảnh.
- IoU, còn được gọi là hệ số Jaccard, đo lường độ tương đồng giữa hai khung chứa. Đó là tỷ lệ của phần giao trên phần hợp của hai khung chứa.
- Trong tập huấn luyện, ta đánh dấu hai loại nhãn cho mỗi khung neo: hạng mục của đối tượng trong khung neo và độ dời của khung chứa chuẩn so với khung neo.
- Khi dự đoán, ta có thể dùng triệt phi cực đại để loại bỏ các khung chứa dự đoán tương tự nhau, từ đó đơn giản hóa kết quả.

#### 15.4.6 Bài tập

1. Hãy thay đổi giá trị size và ratios trong hàm `multibox_prior` và quan sát sự thay đổi của các khung neo được tạo.
2. Tạo hai khung chứa với giá trị IoU là 0.5 và quan sát sự chồng nhau giữa chúng.
3. Xác thực kết quả độ dời `labels[0]` bằng cách đánh dấu các độ dời của khung neo như định nghĩa trong phần này (hàng số là một giá trị mặc định).
4. Thay đổi biến `anchors` trong phần “Gán nhãn Khung neo ở tập Huấn luyện” và “Khung chứa khi Dự đoán”. Kết quả thay đổi như thế nào?

#### 15.4.7 Thảo luận

- Tiếng Anh - MXNet<sup>303</sup>
- Tiếng Việt<sup>304</sup>

#### 15.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Phạm Minh Đức
- Phạm Đăng Khoa
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

### 15.5 Phát hiện Vật thể Đa tỷ lệ

Trong Section 15.4, ta đã tạo ra nhiều khung neo có tâm tại từng điểm ảnh đầu vào. Các khung neo đó được sử dụng để lấy mẫu các vùng khác nhau của ảnh đầu vào. Tuy nhiên, nếu ta sinh khung neo cho mọi điểm trên ảnh thì chẳng mấy chốc sẽ có quá nhiều khung neo phải xử lý. Chẳng hạn, ta giả định rằng ảnh đầu vào có chiều cao và chiều rộng lần lượt là 561 và 728 pixel. Nếu với mỗi điểm ảnh ta sinh ra năm khung neo kích thước khác nhau có cùng tâm ở đó, ta sẽ phải dự đoán và dán nhãn hơn hai triệu khung neo ( $561 \times 728 \times 5$ ).

Việc giảm số lượng khung neo cũng không quá khó. Một cách dễ dàng là lấy mẫu ngẫu nhiên theo phân phối đều trên một lượng nhỏ điểm ảnh từ ảnh đầu vào và tạo ra các khung neo có tâm tại các điểm được chọn.Thêm vào đó, ta có thể tạo ra những khung neo có số lượng và kích thước thay đổi với nhiều tỷ lệ. Lưu ý rằng các vật thể nhỏ hơn nhiều khả năng sẽ được định vị dễ hơn. Ở đây, ta sẽ dùng một ví dụ đơn giản: các vật thể có kích thước  $1 \times 1$ ,  $1 \times 2$ , and  $2 \times 2$  sẽ có thể nằm ở lần lượt 4, 2, và 1 vị trí trên một bức ảnh có kích thước  $2 \times 2$ . Do đó, khi sử dụng những khung neo nhỏ hơn để phát hiện các vật thể nhỏ hơn, ta có thể lấy mẫu nhiều vùng hơn và ngược lại.

Để minh họa cách sinh ra khung neo với nhiều tỷ lệ, trước hết ta hãy đọc một ảnh có kích thước  $561 \times 728$  pixel.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import image, np, npx

npx.set_np()

img = image.imread('../img/catdog.jpg')
```

(continues on next page)

<sup>303</sup> <https://discuss.d2l.ai/t/370>

<sup>304</sup> <https://forum.machinelearningcoban.com/c/d2l>

```
h, w = img.shape[0:2]
h, w
```

Trong [Section 8.2](#), mảng đầu ra 2D của mạng nơ-ron tích chập (CNN) được gọi là một ánh xạ đặc trưng. Ta có thể xác định tâm của các khung neo được lấy mẫu đều trên bất kì ảnh nào bằng cách chỉ định kích thước của ánh xạ đặc trưng này.

Hàm `display_anchors` được định nghĩa như ở dưới. Ta sẽ tạo các khung neo anchors có tâm được đặt theo từng đơn vị (điểm ảnh) trong ánh xạ đặc trưng `fmap`. Do các toạ độ  $x$  và  $y$  trong các khung neo anchors đã được chia cho chiều rộng và chiều cao của ánh xạ đặc trưng `fmap`, ta sử dụng các giá trị trong khoảng từ 0 đến 1 để biểu diễn vị trí tương đối của các khung neo trong ánh xạ đặc trưng. Tâm của các khung neo anchors trùng với tất cả các đơn vị của ánh xạ đặc trưng `fmap`, vị trí tương đối trong không gian của tâm của anchors trên một ảnh bất kỳ bắt buộc phải tuân theo phân phối đều. Cụ thể, khi chiều rộng và chiều cao của một ánh xạ đặc trưng lần lượt được đặt là `fmap_w` và `fmap_h`, hàm này sẽ lấy mẫu các điểm ảnh theo phân phối đều từ `fmap_h` hàng và `fmap_w` cột rồi sử dụng chúng làm tâm để sinh các khung neo với kích thước `s` (ta giả sử rằng độ dài của danh sách `s` là 1) và các tỷ lệ khung ảnh (`ratios`) khác nhau.

```
def display_anchors(fmap_w, fmap_h, s):
    d2l.set_figsize()
    # The values from the first two dimensions will not affect the output
    fmap = np.zeros((1, 10, fmap_w, fmap_h))
    anchors = npx.multibox_prior(fmap, sizes=s, ratios=[1, 2, 0.5])
    bbox_scale = np.array((w, h, w, h))
    d2l.show_bboxes(d2l.plt.imshow(img.asnumpy()).axes,
                    anchors[0] * bbox_scale)
```

Đầu tiên ta sẽ tập trung vào việc phát hiện các vật thể nhỏ. Để dễ dàng phân biệt trong lúc hiển thị, các khung neo có tâm khác nhau ở ví dụ này sẽ không nằm chồng chéo lẫn nhau. Ta giả sử rằng kích thước của các khung neo là 0.15 và chiều cao và chiều rộng của ánh xạ đặc trưng đều bằng 4. Có thể thấy rằng tâm của các khung neo tuân theo phân phối đều trên 4 hàng và 4 cột trong ảnh.

```
display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```

Ta giảm chiều cao và chiều rộng của ánh xạ đặc trưng đi một nửa và sử dụng khung neo lớn hơn để phát hiện vật thể có kích thước lớn hơn. Khi kích thước được đặt bằng 0.4, một số khung neo sẽ nằm chồng chéo nhau.

```
display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```

Cuối cùng, ta sẽ giảm chiều cao và chiều rộng của ánh xạ đặc trưng đi một nửa và tăng kích thước khung neo lên 0.8. Lúc này tâm của khung neo chính là tâm của ảnh.

```
display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```

Do ta sinh các khung neo với kích thước khác nhau trên nhiều tỷ lệ khác nhau, ta sẽ sử dụng chúng để phát hiện các vật thể với kích cỡ đa dạng trên nhiều tỷ lệ khác nhau. Nay giờ chúng tôi sẽ giới thiệu một phương pháp dựa vào mạng nơ-ron tích chập (CNN).

Ở một tỷ lệ nhất định, giả sử rằng ta sinh  $h \times w$  tập hợp khung neo với các tâm khác nhau dựa vào  $c_i$  ánh xạ đặc trưng có kích thước  $h \times w$  và số khung neo của mỗi tập hợp là  $a$ . Ví dụ, đối với tỷ lệ

đầu tiên trong thí nghiệm này, ta sinh 16 tập hợp khung neo với các tâm khác nhau dựa vào 10 (số kênh) ánh xạ đặc trưng có kích thước  $4 \times 4$ , và mỗi tập hợp bao gồm 3 khung neo. Tiếp theo, mỗi khung neo được gán nhãn bằng một danh mục và độ dời dựa vào danh mục được phân loại và vị trí của khung chứa nhãn gốc. Với tỷ lệ hiện tại, mô hình phát hiện vật thể cần phải dự đoán danh mục và độ dời của  $h \times w$  tập hợp khung neo với các tâm khác nhau dựa vào ảnh đầu vào.

Ta giả sử rằng  $c_i$  ánh xạ đặc trưng là đầu ra trung gian của CNN dựa trên ảnh đầu vào. Do mỗi ánh xạ đặc trưng có  $h \times w$  vị trí khác nhau trong không gian, một vị trí sẽ có  $c_i$  đơn vị. Theo định nghĩa của vùng tiếp nhận trong Section 8.2,  $c_i$  đơn vị của ánh xạ đặc trưng nằm ở cùng một vị trí trong không gian sẽ có cùng một vùng tiếp nhận trên ảnh đầu vào. Do đó, chúng biểu diễn thông tin của ảnh đầu vào trên cùng vùng tiếp nhận đó. Vì vậy, ta có thể biến đổi  $c_i$  đơn vị của ánh xạ đặc trưng tại cùng vị trí trong không gian thành danh mục và độ dời cho  $a$  khung neo được sinh ra có tâm tại vị trí đó. Không khó để nhận ra rằng, về bản chất, ta sử dụng thông tin của ảnh đầu vào trong một vùng tiếp nhận nhất định để dự đoán danh mục và độ dời của khung neo gần với vùng đó trên ảnh đầu vào.

Khi các ánh xạ đặc trưng của các tầng khác nhau có các vùng tiếp nhận với kích thước khác nhau trên ảnh đầu vào, chúng được sử dụng để phát hiện vật thể với kích thước khác nhau. Ví dụ, ta có thể thiết kế mạng sao cho mỗi đơn vị trong ánh xạ đặc trưng gần với tầng đầu ra hơn có vùng tiếp nhận rộng hơn, để phát hiện các vật thể với kích thước lớn hơn trong ảnh đầu vào.

Ta sẽ tiến hành lập trình mô hình phát hiện vật thể đa tỷ lệ trong phần kế tiếp.

### 15.5.1 Tóm tắt

- Ta có thể sinh các khung neo với số lượng và kích thước khác nhau trên nhiều tỷ lệ để phát hiện vật thể có kích thước khác nhau trên nhiều tỷ lệ.
- Kích thước của ánh xạ đặc trưng có thể được sử dụng để xác định tâm của các khung neo được lấy mẫu đều trên bất kỳ ảnh nào.
- Ta sử dụng thông tin của ảnh đầu vào từ một vùng tiếp nhận nhất định để dự đoán danh mục và độ dời của các khung neo gần với vùng đó trên ảnh.

### 15.5.2 Bài tập

Cho một ảnh đầu vào, giả sử  $1 \times c_i \times h \times w$  là kích thước của ánh xạ đặc trưng với  $c_i, h, w$  lần lượt là số lượng, chiều cao và chiều dài của ánh xạ đặc trưng. Liệu có phương pháp nào để chuyển đổi biến này thành danh mục và độ dời của một khung neo không? Kích thước của đầu ra là bao nhiêu?

### 15.5.3 Thảo luận

- Tiếng Anh - MXNet<sup>305</sup>
- Tiếng Việt<sup>306</sup>

<sup>305</sup> <https://discuss.d2l.ai/t/371>

<sup>306</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### 15.5.4 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường
- Phạm Minh Đức
- Phạm Hồng Vinh
- Nguyễn Mai Hoàng Long

## 15.6 Tập dữ liệu Phát hiện Đối tượng

Không có một bộ dữ liệu nhỏ nào cho bài toán phát hiện đối tượng như MNIST hay Fashion-MNIST. Để nhanh chóng kiểm định mô hình, ta sẽ tập hợp lại một tập dữ liệu nhỏ. Đầu tiên, ta tạo 1000 bức ảnh Pikachu với nhiều góc độ và kích thước khác nhau bằng mô hình mã nguồn mở Pikachu 3D. Sau đó, ta thu thập một loạt các ảnh nền và đặt ngẫu nhiên ảnh Pikachu lên trên mỗi bức ảnh. Ta dùng [công cụ im2rec<sup>307</sup>](#) do MXNet cung cấp để chuyển đổi hình ảnh gốc sang định dạng RecordIO nhị phân[1]. Định dạng này có khả năng giảm dung lượng lưu trữ và cải thiện hiệu suất đọc dữ liệu. Nếu bạn đọc muốn tìm hiểu thêm về cách đọc ảnh, có thể tham khảo tài liệu cho [bộ công cụ GluonCV<sup>308</sup>](#).

### 15.6.1 Tải xuống tập Dữ liệu

Tập dữ liệu Pikachu ở định dạng RecordIO có thể được tải xuống trực tiếp từ Internet.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()

#@save
d2l.DATA_HUB['pikachu'] = (d2l.DATA_URL + 'pikachu.zip',
                            '68ab1bd42143c5966785eb0d7b2839df8d570190')
```

<sup>307</sup> <https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py>

<sup>308</sup> <https://gluon-cv.mxnet.io/>

## 15.6.2 Đọc dữ liệu

Ta sẽ đọc tập dữ liệu phát hiện đối tượng theo thứ tự ngẫu nhiên bằng thực thể `ImageDetIter`. “`Det`” (viết tắt cho `Detection`), đề cập đến việc phát hiện. Vì định dạng của dữ liệu là `RecordIO`, ta cần có tệp chỉ số '`train.idx`' để đọc minibatch ngẫu nhiên. Ngoài ra, đối với từng ảnh trong tập huấn luyện, ta sẽ cắt xén ngẫu nhiên nhưng vẫn yêu cầu bao phủ ít nhất 95% mỗi đối tượng. Vì việc cắt xén là ngẫu nhiên, yêu cầu này không phải lúc nào cũng được thỏa mãn. Ta cho trước số lần cắt ảnh ngẫu nhiên tối đa là 200 lần. Nếu không có lần nào thỏa mãn yêu cầu, hình ảnh sẽ được giữ nguyên. Để điều ra được đảm bảo, ta sẽ không cắt ngẫu nhiên các hình ảnh trong tập kiểm tra. Ta cũng không cần đọc dữ liệu trong tập kiểm tra theo thứ tự ngẫu nhiên.

```
#@save
def load_data_pikachu(batch_size, edge_size=256):
    """Load the pikachu dataset."""
    data_dir = d2l.download_extract('pikachu')
    train_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'train.rec'),
        path_imgidx=os.path.join(data_dir, 'train.idx'),
        batch_size=batch_size,
        data_shape=(3, edge_size, edge_size), # The shape of the output image
        shuffle=True, # Read the dataset in random order
        rand_crop=1, # The probability of random cropping is 1
        min_object_covered=0.95, max_attempts=200)
    val_iter = image.ImageDetIter(
        path_imgrec=os.path.join(data_dir, 'val.rec'), batch_size=batch_size,
        data_shape=(3, edge_size, edge_size), shuffle=False)
    return train_iter, val_iter
```

Dưới đây, ta đọc một minibatch rồi in ra kích thước ảnh và nhãn. Kích thước ảnh vẫn là (kích thước batch, số kênh, chiều cao, chiều rộng) giống như trong thí nghiệm trước. Kích thước của nhãn là (kích thước batch,  $m$ , 5), trong đó  $m$  là số lượng khung chứa tối đa trên một bức ảnh trong một tập dữ liệu hình ảnh. Mặc dù việc tính toán với minibatch rất hiệu quả, nhưng nó lại yêu cầu mỗi hình ảnh phải cùng một lượng khung chứa để chúng có thể được đặt trong cùng một batch. Vì mỗi hình ảnh có thể có số lượng khung chứa khác nhau, ta có thể thêm các khung chứa ngẫu nhiên để mỗi bức ảnh có  $m$  khung chứa. Do đó, chúng ta có thể đọc được một minibatch mỗi lần. Nhãn của mỗi khung chứa trong bức ảnh được biểu diễn bằng một mảng có độ dài là 5. Phần tử đầu tiên trong mảng là hạng mục của đối tượng xuất hiện trong khung chứa. Khi giá trị là -1, khung chứa ấy chính là khung chứa ngẫu nhiên dùng để lấp đầy. Bốn phần tử còn lại trong mảng đại diện cho toạ độ trực  $x, y$  của góc trên bên trái và góc dưới bên phải của khung chứa (miền giá trị từ 0 đến 1). Tập dữ liệu Pikachu ở đây chỉ có một khung chứa cho mỗi ảnh, vì thế  $m = 1$ .

```
batch_size, edge_size = 32, 256
train_iter, _ = load_data_pikachu(batch_size, edge_size)
batch = train_iter.next()
batch.data[0].shape, batch.label[0].shape
```

### 15.6.3 Minh họa

Ta có mười bức ảnh kèm với các khung chứa trên chúng. Chúng ta có thể thấy rằng góc, kích thước và vị trí của Pikachu khác nhau trong mỗi bức ảnh. Dĩ nhiên, đây là một tập dữ liệu tự tạo đơn giản. Trong thực tế, dữ liệu thường phức tạp hơn nhiều.

```
imgs = (batch.data[0][0:10].transpose(0, 2, 3, 1)) / 255
axes = d2l.show_images(imgs, 2, 5, scale=2)
for ax, label in zip(axes, batch.label[0][0:10]):
    d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=[‘w’])
```

### 15.6.4 Tóm tắt

- Tập dữ liệu Pikachu mà ta tổng hợp có thể được dùng để kiểm tra các mô hình phát hiện đối tượng.
- Việc đọc dữ liệu để phát hiện đối tượng giống như khi phân loại hình ảnh. Tuy nhiên, sau khi đưa ra các khung chứa, kích thước nhãn và việc tăng cường ảnh (ví dụ, cắt xén ngẫu nhiên) sẽ thay đổi.

### 15.6.5 Bài tập

Trong tài liệu MXNet, tham số trong các hàm tạo (*constructors*) của các lớp `image.ImageDetIter` và `image.CreateDetAugmenter` là gì? Cho biết ý nghĩa của chúng?

### 15.6.6 Thảo luận

- Tiếng Anh - MXNet<sup>309</sup>
- Tiếng Việt<sup>310</sup>

### 15.6.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Đăng Khoa
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

<sup>309</sup> <https://discuss.d2l.ai/t/372>

<sup>310</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 15.7 Phát hiện Nhiều khung Một lượt (SSD)

Ở một số phần trước, chúng tôi đã giới thiệu về khung chứa, khung neo, phát hiện vật thể đa tỷ lệ và tập dữ liệu. Giờ ta sẽ sử dụng phần kiến thức nền tảng này để xây dựng một mô hình phát hiện vật thể: phát hiện nhiều khung trong một lần thực hiện (*Single Shot Multibox Detection - SSD*) (Liu et al., 2016). Mô hình này đang được sử dụng rộng rãi nhờ tốc độ và tính đơn giản của nó. Một số khái niệm thiết kế và chi tiết lập trình của mô hình này cũng có thể được áp dụng cho các mô hình phát hiện vật thể khác.

### 15.7.1 Mô hình

Fig. 15.7.1 mô tả thiết kế của một mô hình SSD. Các thành phần chính của mô hình gồm có một khái niệm cơ sở và vài khái niệm đặc trưng đa tỷ lệ được liên kết thành chuỗi. Trong đó khái niệm cơ sở được sử dụng để trích xuất đặc trưng từ ảnh gốc, thường có dạng một mạng nơ-ron tích chập sâu. Bài báo về SSD dùng mạng VGG-16 cụt đặt trước tầng phân loại (Liu et al., 2016), tuy nhiên bây giờ nó thường được thay bằng ResNet. Ta có thể thiết kế mạng cơ sở để đầu ra có chiều cao và chiều rộng lớn hơn. Bằng cách này, ảnh xạ đặc trưng này sẽ sinh ra nhiều khung neo hơn, cho phép ta phát hiện các vật thể nhỏ hơn. Tiếp theo, mỗi khái niệm đặc trưng đa tỷ lệ sẽ giảm chiều cao và chiều rộng của ảnh xạ đặc trưng ở tầng trước (giảm kích thước đi còn một nửa chẳng hạn). Các khái niệm này sau đó sử dụng từng phần tử trong ảnh xạ đặc trưng để mở rộng vùng tiếp nhận trên ảnh đầu vào. Bằng cách này, khái niệm đặc trưng đa tỷ lệ càng gần đỉnh mô hình trong Fig. 15.7.1 thì trả về ảnh xạ đặc trưng càng nhỏ, và số khung neo được sinh ra bởi ảnh xạ đặc trưng đó càng ít. Hơn nữa, khái niệm đặc trưng càng gần đỉnh mô hình thì vùng tiếp nhận của mỗi phần tử trong ảnh xạ đặc trưng càng lớn và càng phù hợp để phát hiện những vật thể lớn. Vì SSD sinh ra các tập khung neo với số lượng và kích thước khác nhau dựa trên khái niệm cơ sở và từng khái niệm đặc trưng đa tỷ lệ, rồi sau đó dự đoán hạng mục và độ dời (tức là dự đoán khung chứa) cho các khung neo để phát hiện các vật thể với kích cỡ khác nhau, có thể nói SSD là một mô hình phát hiện vật thể đa tỷ lệ.

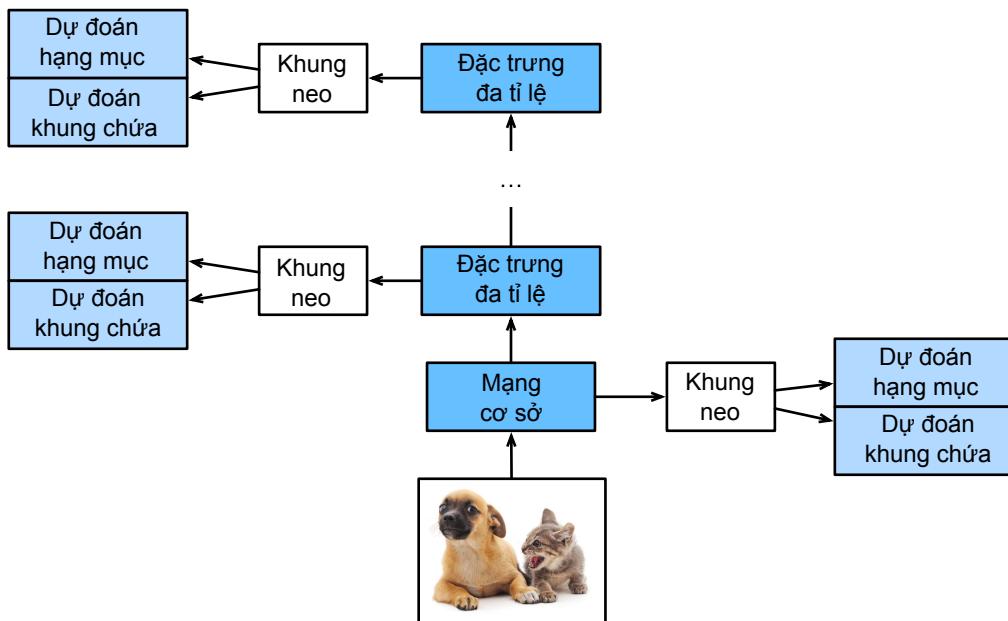


Fig. 15.7.1: SSD được cấu thành bởi một khái niệm cơ sở và nhiều khái niệm đặc trưng đa tỷ lệ được liên kết thành một chuỗi.

Tiếp theo, ta sẽ mô tả chi tiết lập trình cho các mô-đun trong Fig. 15.7.1. Đầu tiên, ta cần phải thảo luận về cách lập trình chức năng dự đoán hạng mục và khung chứa.

### Tầng Dự đoán Hạng mục

Đặt số hạng mục của vật thể là  $q$ . Trong trường hợp này, số hạng mục của khung neo là  $q + 1$ , với 0 kí hiệu khung neo chỉ là nền hậu cảnh. Ở một tỷ lệ nhất định, đặt chiều cao và chiều rộng của ánh xạ đặc trưng lần lượt là  $h$  và  $w$ . Nếu ta sử dụng từng phần tử làm tâm để sinh  $a$  khung neo, ta cần phân loại tổng cộng  $hwa$  khung neo. Nếu ta sử dụng một tầng kết nối đầy đủ (FCN) tại đầu ra thì khả năng cao là số lượng tham số mô hình sẽ quá lớn. Hãy nhớ lại cách ta sử dụng các kênh trong tầng tích chập để đưa ra dự đoán hạng mục trong Section 9.3. SSD sử dụng phương pháp tương tự để giảm độ phức tạp của mô hình.

Cụ thể, tầng dự đoán hạng mục sử dụng một tầng tích chập giữ nguyên chiều cao và chiều rộng của đầu vào. Do đó, tọa độ trong không gian của đầu ra và đầu vào tương quan một-một với nhau dọc theo cả chiều cao và chiều rộng của ánh xạ đặc trưng. Giả sử rằng đầu ra và đầu vào này có cùng tọa độ không gian  $(x, y)$ , các kênh của ánh xạ đặc trưng đầu ra tại tọa độ  $(x, y)$  đại diện cho các dự đoán hạng mục của tất cả các khung neo được sinh ra khi sử dụng tọa độ  $(x, y)$  của ánh xạ đặc trưng đầu vào làm trung tâm. Bởi lẽ đó, có tất cả  $a(q+1)$  kênh đầu ra, với các kênh đầu ra được đánh chỉ số theo  $i(q+1) + j$  ( $0 \leq j \leq q$ ) biểu diễn dự đoán hạng mục thứ  $j$  cho khung neo thứ  $i$ .

Bây giờ, ta định nghĩa một tầng dự đoán hạng mục theo dạng này. Sau khi ta xác định các tham số  $a$  và  $q$ , tầng này sử dụng một tầng tích chập  $3 \times 3$  với đệm bằng 1. Chiều cao và chiều rộng của đầu ra và đầu vào của tầng tích chập này không đổi.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

def cls_predictor(num_anchors, num_classes):
    return nn.Conv2D(num_anchors * (num_classes + 1), kernel_size=3,
                   padding=1)
```

### Tầng Dự đoán Khung chứa

Thiết kế của tầng dự đoán khung chứa cũng tương tự như tầng dự đoán hạng mục. Điểm khác biệt duy nhất đó là ta cần dự đoán 4 giá trị độ dời cho từng khung neo, thay vì  $q + 1$  hạng mục.

```
def bbox_predictor(num_anchors):
    return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

## Ghép nối Dự đoán Đa Tỷ lệ

Như đã đề cập, SSD sử dụng các ảnh xạ đặc trưng trên nhiều tỷ lệ để sinh các khung neo rồi dự đoán hạng mục và độ dời. Vì kích thước và số lượng các khung neo có tâm tại cùng một phần tử là khác nhau đối với ảnh xạ đặc trưng có tỷ lệ khác nhau, các đầu ra dự đoán tại các tỷ lệ khác nhau có thể sẽ có kích thước khác nhau.

Trong ví dụ dưới đây, ta sử dụng cùng một batch dữ liệu để xây dựng ảnh xạ đặc trưng Y1 và Y2 của hai tỷ lệ khác nhau. Trong đó, Y2 có chiều cao và chiều rộng bằng một nửa Y1. Ví dụ khi dự đoán hạng mục, giả sử mỗi phần tử trong ảnh xạ đặc trưng Y1 và Y2 sinh 5 (với Y1) và 3 (với Y2) khung neo tương ứng. Với 10 hạng mục vật thể, số lượng kênh đầu ra của tầng dự đoán hạng mục sẽ là  $5 \times (10 + 1) = 55$  hoặc  $3 \times (10 + 1) = 33$  tương ứng. Định dạng đầu ra dự đoán là (kích thước batch, số lượng kênh, chiều cao, chiều rộng). Ta thấy, ngoại trừ kích thước batch, kích thước của các chiều còn lại là khác nhau. Do đó, ta phải biến đổi chúng về cùng một định dạng và ghép nối dự đoán đa tỷ lệ để dễ tính toán về sau.

```
def forward(x, block):
    block.initialize()
    return block(x)

Y1 = forward(np.zeros((2, 8, 20, 20)), cls_predictor(5, 10))
Y2 = forward(np.zeros((2, 16, 10, 10)), cls_predictor(3, 10))
(Y1.shape, Y2.shape)
```

Chiều kênh chứa dự đoán cho tất cả các khung neo có cùng tâm. Đầu tiên, ta sẽ chuyển chiều kênh thành chiều cuối cùng. Do kích thước batch là giống nhau với mọi tỷ lệ, ta có thể chuyển đổi kết quả dự đoán thành định dạng 2D (kích thước batch, chiều cao  $\times$  chiều rộng  $\times$  số lượng kênh) để việc ghép nối trên chiều thứ nhất dễ dàng hơn.

```
def flatten_pred(pred):
    return npx.batch_flatten(pred.transpose(0, 2, 3, 1))

def concat_preds(preds):
    return np.concatenate([flatten_pred(p) for p in preds], axis=1)
```

Do đó, ta có thể ghép nối kết quả dự đoán cho hai tỷ lệ khác nhau trên cùng một batch dù Y1 và Y2 có kích thước khác nhau.

```
concat_preds([Y1, Y2]).shape
```

## Khối giảm Chiều cao và Chiều rộng

Với bài toán phát hiện vật thể đa tỷ lệ, ta định nghĩa khối down\_sample\_blk sau đây để giảm 50% chiều cao và chiều rộng. Khối này bao gồm 2 tầng tích chập  $3 \times 3$  với đệm bằng 1 và tầng gộp cực đại  $2 \times 2$  cùng sải bước bằng 2 được kết nối tuần tự. Như ta đã biết, tầng tích chập  $3 \times 3$  với đệm bằng 1 sẽ không thay đổi kích thước của ảnh xạ đặc trưng. Tuy nhiên, tầng gộp cực đại tiếp theo sẽ giảm một nửa kích thước của ảnh xạ đặc trưng. Do  $1 \times 2 + (3 - 1) + (3 - 1) = 6$ , mỗi phần tử trong ảnh xạ đặc trưng đầu ra sẽ có vùng tiếp nhận với kích thước  $6 \times 6$  trên ảnh xạ đặc trưng đầu vào. Ta có thể thấy, khối giảm mẫu trên chiều cao và chiều rộng mở rộng vùng tiếp nhận của mỗi phần tử trong ảnh xạ đặc trưng đầu ra.

```

def down_sample_blk(num_channels):
    blk = nn.Sequential()
    for _ in range(2):
        blk.add(nn.Conv2D(num_channels, kernel_size=3, padding=1),
               nn.BatchNorm(in_channels=num_channels),
               nn.Activation('relu'))
    blk.add(nn.MaxPool2D(2))
    return blk

```

Kiểm tra tính toán của lượt truyền xuôi trong khối giảm chiều cao và chiều rộng, ta có thể thấy khối này thay đổi số kênh đầu vào và giảm một nửa chiều cao và chiều rộng.

```
forward(np.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
```

## **Khối Mạng Cơ sở**

Khối mạng cơ sở được sử dụng để trích xuất đặc trưng từ ảnh gốc ban đầu. Để đơn giản hóa phép tính, ta sẽ xây dựng một mạng cơ sở nhỏ, bao gồm các khối giảm chiều cao và chiều rộng được kết nối tuần tự sao cho số lượng kênh tăng gấp đôi sau mỗi bước. Khi ta truyền ảnh đầu vào có kích thước  $256 \times 256$ , khối mạng cơ sở sẽ cho ra ánh xạ đặc trưng có kích thước  $32 \times 32$ .

```

def base_net():
    blk = nn.Sequential()
    for num_filters in [16, 32, 64]:
        blk.add(down_sample_blk(num_filters))
    return blk

```

```
forward(np.zeros((2, 3, 256, 256)), base_net()).shape
```

## **Mô hình hoàn chỉnh**

Mô hình SSD chứa tất cả năm mô-đun. Mỗi mô-đun tạo ra một ánh xạ đặc trưng dùng để sinh các khung neo, dự đoán hạng mục và độ dời của các khung neo đó. Mô-đun đầu tiên là khối mạng cơ sở, các mô-đun từ thứ hai tới thứ tư là các khối giảm chiều cao và chiều rộng, và mô-đun thứ năm là tầng gộp cực đại toàn cục nhằm giảm chiều cao và chiều rộng xuống còn 1. Do đó, mô-đun thứ hai tới thứ năm đều là các khối đặc trưng đa tỷ lệ như mô tả trong Fig. 15.7.1.

```

def get_blk(i):
    if i == 0:
        blk = base_net()
    elif i == 4:
        blk = nn.GlobalMaxPool2D()
    else:
        blk = down_sample_blk(128)
    return blk

```

Bây giờ, ta sẽ định nghĩa lượt truyền xuôi cho từng mô-đun. Khác với các mạng nơ-ron tích chập đã mô tả trước đây, mô-đun này không chỉ trả về ánh xạ đặc trưng  $Y$  xuất ra từ phép tích chập, mà còn sinh ra từ  $Y$  cả các khung neo của tỷ lệ hiện tại cùng với các dự đoán hạng mục và độ dời.

```

def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
    Y = blk(X)
    anchors = npx.multibox_prior(Y, sizes=size, ratios=ratio)
    cls_preds = cls_predictor(Y)
    bbox_preds = bbox_predictor(Y)
    return (Y, anchors, cls_preds, bbox_preds)

```

Như đã đề cập trong Fig. 15.7.1, khối đặc trưng đa tỷ lệ càng gần đĩnh, các vật thể được phát hiện và các khung neo được tạo ra càng lớn. Ở đây, trước hết ta chia khoảng từ 0.2 tới 1.05 thành năm phần bằng nhau để xác định các kích thước của các khung neo nhỏ hơn ở các tỷ lệ: 0.2, 0.37, 0.54, v.v. Kế đến, theo  $\sqrt{0.2 \times 0.37} = 0.272$ ,  $\sqrt{0.37 \times 0.54} = 0.447$ , và các công thức tương tự; ta xác định kích thước của các khung neo lớn hơn ở các tỷ lệ khác nhau.

```

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, ta có thể định nghĩa mô hình hoàn chỉnh, TinySSD.

```

class TinySSD(nn.Block):
    def __init__(self, num_classes, **kwargs):
        super(TinySSD, self).__init__(**kwargs)
        self.num_classes = num_classes
        for i in range(5):
            # The assignment statement is self.blk_i = get_blk(i)
            setattr(self, f'blk_{i}', get_blk(i))
            setattr(self, f'cls_{i}', cls_predictor(num_anchors, num_classes))
            setattr(self, f'bbox_{i}', bbox_predictor(num_anchors))

    def forward(self, X):
        anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
        for i in range(5):
            # 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}'))
        # In the reshape function, 0 indicates that the batch size remains
        # unchanged
        anchors = np.concatenate(anchors, axis=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

```

Bây giờ ta thử tạo một mô hình SSD và sử dụng nó để thực hiện lượt truyền xuôi trên minibatch ảnh X có chiều rộng và chiều cao là 256 pixel. Như đã kiểm nghiệm trước đó, mô-đun đầu tiên xuất ảnh xạ đặc trưng với kích thước  $32 \times 32$ . Bởi vì các mô-đun từ thứ hai tới thứ tư là các khối giảm chiều cao và chiều rộng, còn mô-đun thứ năm là tầng gộp toàn cục, và mỗi phần tử trong ảnh xạ đặc trưng này được dùng làm tâm cho bốn khung neo, tổng cộng  $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$  khung neo được tạo ra cho mỗi ảnh ở năm tỷ lệ đó.

```

net = TinySSD(num_classes=1)
net.initialize()
X = np.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)

```

## 15.7.2 Huấn luyện

Ở bước này chúng tôi sẽ giải thích từng bước cách huấn luyện mô hình SSD để phát hiện vật thể.

### Đọc Dữ liệu và Khởi tạo

Ta đọc tập dữ liệu Pikachu được tạo ở phần trước.

```

batch_size = 32
train_iter, _ = d2l.load_data_pikachu(batch_size)

```

Có 1 hạng mục trong tập dữ liệu Pikachu. Sau khi khai báo mô hình và thiết bị, ta khởi tạo các tham số của mô hình và định nghĩa thuật toán tối ưu.

```

device, net = d2l.try_gpu(), TinySSD(num_classes=1)
net.initialize(init=init.Xavier(), ctx=device)
trainer = gluon.Trainer(net.collect_params(), 'sgd',
                        {'learning_rate': 0.2, 'wd': 5e-4})

```

### Định nghĩa Hàm mất mát và Hàm đánh giá

Phát hiện vật thể có hai loại mất mát. Thứ nhất là mất mát khi phân loại hạng mục của khung neo. Đối với mất mát này, ta hoàn toàn có thể sử dụng lại hàm mất mát entropy chéo trong phân loại ảnh. Loại mất mát thứ hai là mất mát của độ dời khung neo dương. Dự đoán độ dời là một bài toán chuẩn hóa. Tuy nhiên, ở đây ta không sử dụng hàm mất mát bình phương như trước. Thay vào đó, ta sử dụng mất mát chuẩn  $L_1$ , tức là trị tuyệt đối hiệu của giá trị dự đoán và giá trị nhãn gốc. Biến mất nã bbox\_masks loại bỏ các khung neo âm và khung neo đệm khỏi phép tính mất mát. Cuối cùng, ta cộng mất mát hạng mục và mất mát độ dời của khung neo để có hàm mất mát cuối cùng cho mô hình.

```

cls_loss = gluon.loss.SoftmaxCrossEntropyLoss()
bbox_loss = gluon.loss.L1Loss()

def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
    cls = cls_loss(cls_preds, cls_labels)
    bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
    return cls + bbox

```

Ta có thể sử dụng độ chính xác để đánh giá kết quả phân loại. Do ta sử dụng mất mát chuẩn  $L_1$  khi huấn luyện, ta sẽ sử dụng trung bình sai số tuyệt đối (*average absolute error*) để đánh giá kết quả dự đoán khung chứa.

```
def cls_eval(cls_preds, cls_labels):
    # Because the category prediction results are placed in the final
    # dimension, argmax must specify this dimension
    return float((cls_preds.argmax(axis=-1).astype(
        cls_labels.dtype) == cls_labels).sum())

def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
    return float((np.abs((bbox_labels - bbox_preds) * bbox_masks)).sum())
```

## Huấn luyện Mô hình

Trong suốt quá trình huấn luyện, ta phải tạo ra các khung neo đa tỷ lệ (anchors) khi tính toán lượn truyền xuôi rồi dự đoán hạng mục (cls\_preds) và độ dời (bbox\_preds) cho mỗi khung neo. Sau đó, ta gán nhãn hạng mục (cls\_labels) và độ dời (bbox\_labels) cho từng khung neo được tạo ở trên dựa vào thông tin nhãn Y. Cuối cùng, ta tính toán hàm mất mát sử dụng giá trị hạng mục/độ dời nhãn gốc và dự đoán. Để đơn giản hóa mã nguồn, ta sẽ không đánh giá tập huấn luyện ở đây.

```
num_epochs, timer = 20, d2l.Timer()
animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                         legend=['class error', 'bbox mae'])
for epoch in range(num_epochs):
    # accuracy_sum, mae_sum, num_examples, num_labels
    metric = d2l.Accumulator(4)
    train_iter.reset() # Read data from the start.
    for batch in train_iter:
        timer.start()
        X = batch.data[0].as_in_ctx(device)
        Y = batch.label[0].as_in_ctx(device)
        with autograd.record():
            # Generate multiscale anchor boxes and predict the category and
            # offset of each
            anchors, cls_preds, bbox_preds = net(X)
            # Label the category and offset of each anchor box
            bbox_labels, bbox_masks, cls_labels = npx.multibox_target(
                anchors, Y, cls_preds.transpose(0, 2, 1))
            # Calculate the loss function using the predicted and labeled
            # category and offset values
            l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                          bbox_masks)
            l.backward()
            trainer.step(batch_size)
            metric.add(cls_eval(cls_preds, cls_labels), cls_labels.size,
                       bbox_eval(bbox_preds, bbox_labels, bbox_masks),
                       bbox_labels.size)
    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'{train_iter.num_image/timer.stop():.1f} examples/sec on '
      f'{str(device)}')
```

### 15.7.3 Dự đoán

Trong bước dự đoán, ta muốn phát hiện tất cả các vật thể đáng quan tâm trong ảnh. Ở đoạn mã dưới, ta đọc và biến đổi kích thước của ảnh kiểm tra, rồi chuyển thành dạng tensor bốn chiều mà tầng tích chập yêu cầu.

```
img = image.imread('../img/pikachu.jpg')
feature = image.imresize(img, 256, 256).astype('float32')
X = np.expand_dims(feature.transpose(2, 0, 1), axis=0)
```

Ta sử dụng hàm `MultiBoxDetection` để dự đoán các khung chứa dựa theo các khung neo và giá trị độ dời dự đoán của chúng. Sau đó ta sử dụng triệt phi cực đại (*non-maximum suppression*) để loại bỏ các khung chứa giống nhau.

```
def predict(X):
    anchors, cls_preds, bbox_preds = net(X.as_in_ctx(device))
    cls_probs = npx.softmax(cls_preds).transpose(0, 2, 1)
    output = npx.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, ta lấy toàn bộ khung chứa có độ tin cậy tối thiểu là 0.3 và hiển thị chúng làm kết quả cuối cùng.

```
def display(img, output, threshold):
    d2l.set_figsize((5, 5))
    fig = d2l.plt.imshow(img.asnumpy())
    for row in output:
        score = float(row[1])
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * np.array((w, h, w, h), ctx=row.ctx)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output, threshold=0.3)
```

### 15.7.4 Tóm tắt

- SSD là một mô hình phát hiện vật thể đa tỷ lệ. Mô hình này sinh ra các tập khung neo với số lượng và kích thước khác nhau dựa trên khối mạng cơ sở và từng khối đặc trưng đa tỷ lệ, rồi dự đoán hạng mục và độ dời cho các khung neo để phát hiện các vật thể có kích thước khác nhau.
- Trong suốt quá trình huấn luyện, hàm mất mát được tính bằng giá trị dự đoán và nhãn gốc của hạng mục và độ dời.

### 15.7.5 Bài tập

Do nhiều giới hạn, chúng tôi đã bỏ qua một số chi tiết phần lập trình cho mô hình SSD trong thí nghiệm này. Liệu bạn có thể cải thiện mô hình hơn nữa theo các hướng sau?

#### Hàm mất mát

A. Để dự đoán độ dời, thay thế mất mát chuẩn  $L_1$  bằng mất mát điều chuẩn  $L_1$ . Hàm mất mát này sử dụng hàm bình phương xung quanh giá trị không để tăng độ mượt. Đây chính là vùng được điều chuẩn và được xác định bởi siêu tham số  $\sigma$ :

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{nếu } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{mặt khác} \end{cases} \quad (15.7.1)$$

Khi  $\sigma$  lớn, mất mát này tương đương với mất mát chuẩn  $L_1$ . Khi giá trị này nhỏ, hàm mất mát sẽ mượt hơn.

```
sigmas = [10, 1, 0.5]
lines = ['-', '--', '-.']
x = np.arange(-2, 2, 0.1)
d2l.set_figsize()

for l, s in zip(lines, sigmas):
    y = npx.smooth_l1(x, scalar=s)
    d2l.plt.plot(x.asnumpy(), y.asnumpy(), l, label='sigma=%1f' % s)
d2l.plt.legend();
```

Trong thí nghiệm ở phần này, ta sử dụng hàm mất mát entropy chéo để dự đoán hạng mục. Còn giờ, giả sử rằng xác suất dự đoán được đúng hạng mục  $j$  là  $p_j$  và mất mát entropy chéo là  $-\log p_j$ . Ta cũng có thể sử dụng mất mát tiêu điểm (*focal loss*) (Lin et al., 2017). Cho siêu tham số  $\gamma$  and  $\alpha$  dương, mất mát này được định nghĩa như sau:

$$-\alpha(1 - p_j)^\gamma \log p_j. \quad (15.7.2)$$

Như bạn có thể thấy, bằng cách tăng  $\gamma$ , ta thực chất có thể giảm giá trị mất mát đi khi khả năng dự đoán đúng hạng mục là lớn.

```
def focal_loss(gamma, x):
    return -(1 - x) ** gamma * np.log(x)

x = np.arange(0.01, 1, 0.01)
for l, gamma in zip(lines, [0, 1, 5]):
    y = d2l.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                      label='gamma=%1f' % gamma)
d2l.plt.legend();
```

## **Huấn luyện và Dự đoán**

B. Khi một vật thể có kích thước khá lớn so với ảnh, mô hình thường chấp nhận kích thước ảnh đầu vào lớn hơn.

C. Điều này thường sản sinh lượng lớn các khung neo âm khi gán nhãn hạng mục cho khung neo. Ta có thể lấy mẫu các khung neo âm để cân bằng các hạng mục trong dữ liệu tốt hơn. Để thực hiện điều này, ta có thể đặt tham số negative\_mining\_ratio của hàm MultiBoxTarget.

D. Trong hàm mất mát, sử dụng các trọng số khác nhau cho mất mát hạng mục của các khung neo và mất mát độ dời của các khung neo dương.

E. Tham khảo bài báo SSD. Phương pháp nào có thể được sử dụng để đánh giá giá trị precision của các mô hình phát hiện vật thể (Liu et al., 2016)?

### **15.7.6 Thảo luận**

- Tiếng Anh - MXNet<sup>311</sup>
- Tiếng Việt<sup>312</sup>

### **15.7.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức

---

<sup>311</sup> <https://discuss.d2l.ai/t/373>

<sup>312</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 15.8 CNN theo Vùng (R-CNN)

Mạng nơ-ron tích chập theo vùng, hay các vùng với đặc trưng CNN (R-CNN) là một hướng tiếp cận tiên phong ứng dụng mô hình sâu cho bài toán phát hiện vật thể (Girshick et al., 2014). Trong phần này, chúng ta sẽ thảo luận về R-CNN và một loạt các cải tiến sau đó: Fast R-CNN (Girshick, 2015), Faster R-CNN (Ren et al., 2015), và Mask R-CNN (He et al., 2017a).

### 15.8.1 R-CNN

Đầu tiên, các mô hình R-CNN sẽ chọn một số vùng đề xuất từ ảnh (ví dụ, các khung neo cũng là một phương pháp lựa chọn) và sau đó gán nhãn hạng mục và khung chứa (ví dụ, các giá trị độ dời) cho các vùng này. Tiếp đến, các mô hình này sử dụng CNN để thực hiện lượt truyền xuôi nhằm trích xuất đặc trưng từ từng vùng đề xuất. Sau đó, ta sử dụng các đặc trưng của từng vùng được đề xuất để dự đoán hạng mục và khung chứa. Fig. 15.8.1 mô tả một mô hình R-CNN.

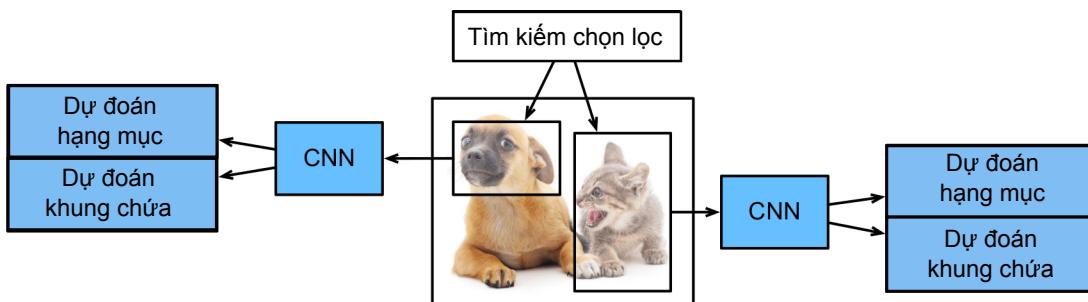


Fig. 15.8.1: Mô hình R-CNN.

Cụ thể, R-CNN có bốn phần chính sau:

1. Tìm kiếm chọn lọc trên ảnh đầu vào để lựa chọn các vùng đề xuất tiềm năng (Uijlings et al., 2013). Các vùng đề xuất thông thường sẽ có nhiều tỷ lệ với hình dạng và kích thước khác nhau. Hạng mục và khung chứa nhãn gốc sẽ được gán cho từng vùng đề xuất.
2. Sử dụng một mạng CNN đã qua tiền huấn luyện, ở dạng rút gọn, đặt trước tầng đầu ra. Mạng này biến đổi từng vùng đề xuất thành các đầu vào có chiều phù hợp với mạng và thực hiện các lượt truyền xuôi để trích xuất đặc trưng từ các vùng đề xuất tương ứng.
3. Các đặc trưng và nhãn hạng mục của từng vùng đề xuất được kết hợp thành một mẫu để huấn luyện các máy vector hỗ trợ cho phép phân loại vật thể. Ở đây, mỗi máy vector hỗ trợ được sử dụng để xác định một mẫu có thuộc về một hạng mục nào đó hay không.
4. Các đặc trưng và khung chứa được gán nhãn của mỗi vùng đề xuất được kết hợp thành một mẫu để huấn luyện mô hình hồi quy tuyến tính, để phục vụ dự đoán khung chứa nhãn gốc.

Mặc dù các mô hình R-CNN sử dụng các mạng CNN đã được tiền huấn luyện để trích xuất các đặc trưng ảnh một cách hiệu quả, điểm hạn chế chính yếu đó là tốc độ chậm. Có thể hình dung, với hàng ngàn vùng đề xuất từ một ảnh, ta cần tới hàng ngàn phép tính truyền xuôi từ mạng CNN để phát hiện vật thể. Khối lượng tính toán nặng nề khiến các mô hình R-CNN không được sử dụng rộng rãi trong các ứng dụng thực tế.

### 15.8.2 Fast R-CNN

Điểm nghẽn cổ chai chính về hiệu năng của R-CNN đó là việc trích xuất đặc trưng cho từng vùng đề xuất một cách độc lập. Do các vùng đề xuất này có độ chồng lặp cao, việc trích xuất đặc trưng một cách độc lập sẽ dẫn đến một số lượng lớn các phép tính lặp lại. Fast R-CNN cải thiện R-CNN bằng cách chỉ thực hiện lượt truyền xuôi qua mạng CNN trên toàn bộ ảnh.

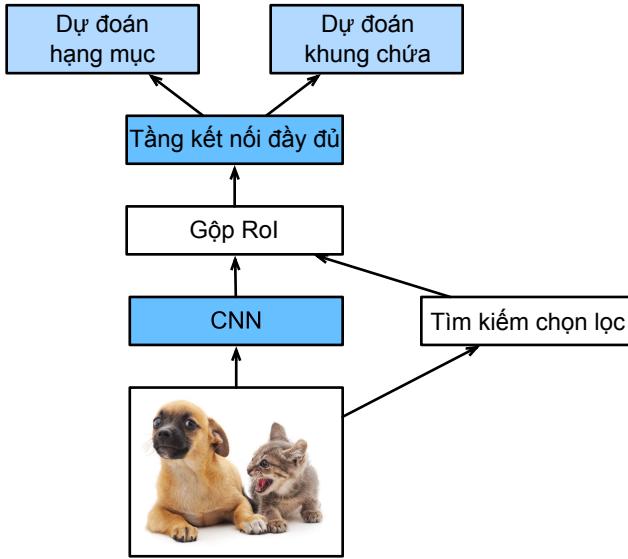


Fig. 15.8.2: Mô hình Fast R-CNN.

Fig. 15.8.2 mô tả mạng Fast R-CNN. Các bước tính toán chính yếu được mô tả như sau:

1. So với mạng R-CNN, mạng Fast R-CNN sử dụng toàn bộ ảnh làm đầu vào cho CNN để trích xuất đặc trưng thay vì từng vùng đề xuất. Hơn nữa, mạng này được huấn luyện như bình thường để cập nhật tham số mô hình. Do đầu vào là toàn bộ ảnh, đầu ra của mạng CNN có kích thước  $1 \times c \times h_1 \times w_1$ .
2. Giả sử thuật toán tìm kiếm chọn lọc chọn ra  $n$  vùng đề xuất, kích thước khác nhau của các vùng này chỉ ra rằng vùng quan tâm (*regions of interests - ROI*) tại đầu ra của CNN có kích thước khác nhau. Các đặc trưng có cùng kích thước phải được trích xuất từ các vùng quan tâm này (giả sử có chiều cao là  $h_2$  và chiều rộng là  $w_2$ ). Mạng Fast R-CNN đề xuất phép gộp ROI (*ROI pooling*), nhận đầu ra từ CNN và các vùng quan tâm làm đầu vào rồi ghép nối các đặc trưng được trích xuất từ mỗi vùng quan tâm làm đầu ra có kích thước  $n \times c \times h_2 \times w_2$ .
3. Tầng kết nối đầy đủ được sử dụng để biến đổi kích thước đầu ra thành  $n \times d$ , trong đó  $d$  được xác định khi thiết kế mô hình.
4. Khi dự đoán hạng mục, kích thước đầu ra của tầng kết nối đầy đủ lại được biến đổi thành  $n \times q$  và áp dụng phép hồi quy softmax ( $q$  là số lượng hạng mục). Khi dự đoán khung chứa, kích thước đầu ra của tầng đầy đủ lại được biến đổi thành  $n \times 4$ . Nghĩa là ta dự đoán hạng mục và khung chứa cho từng vùng đề xuất.

Tầng gộp ROI trong mạng Fast R-CNN có phần khác với các tầng gộp mà ta đã thảo luận trước đó. Trong tầng gộp thông thường, ta thiết lập cửa sổ gộp, giá trị đệm, và sải bước để quyết định kích thước đầu ra. Trong tầng gộp ROI, ta có thể trực tiếp định rõ kích thước đầu ra của từng vùng, ví dụ chiều cao và chiều rộng của từng vùng sẽ là  $h_2, w_2$ . Giả sử chiều cao và chiều rộng của cửa sổ ROI là  $h$  và  $w$ , cửa sổ này được chia thành một lưới có  $h_2 \times w_2$  cửa sổ con. Mỗi cửa sổ con có kích thước xấp xỉ  $(h/h_2) \times (w/w_2)$ . Chiều cao và chiều rộng của cửa sổ con phải luôn là số nguyên và

thành phần lớn nhất được sử dụng là đầu ra cho cửa sổ con đó. Điều này cho phép tầng gộp ROI trích xuất đặc trưng có cùng kích thước từ các vùng quan tâm có kích thước khác nhau.

Trong Fig. 15.8.3, ta chọn một vùng  $3 \times 3$  làm ROI của một đầu vào  $4 \times 4$ . Với ROI này, ta sử dụng một tầng gộp ROI  $2 \times 2$  để thu được một đầu ra đơn  $2 \times 2$ . Khi chia vùng này thành bốn cửa sổ con, chúng lần lượt chứa các phần tử 0, 1, 4 và 5 (5 là lớn nhất); 2 và 6 (6 là lớn nhất); 8 và 9 (9 là lớn nhất); và 10.

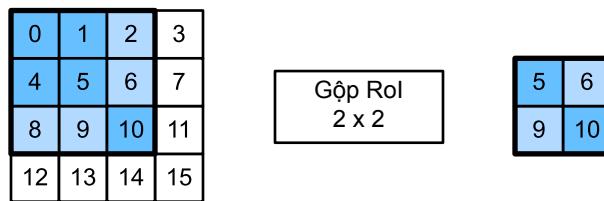


Fig. 15.8.3: Tầng gộp ROI  $2 \times 2$ .

Ta sử dụng hàm ROI Pooling để thực hiện việc tính toán tầng gộp ROI. Giả sử rằng CNN trích xuất đặc trưng  $X$  với chiều rộng và chiều cao là 4 với một kênh duy nhất.

```
from mxnet import np, npx

npx.set_np()

X = np.arange(16).reshape(1, 1, 4, 4)
X
```

Giả sử chiều cao và chiều rộng của ảnh là 40 pixel và tìm kiếm chọn lọc sinh ra hai vùng đề xuất trên ảnh này. Mỗi vùng được biểu diễn bằng 5 phần tử: hạng mục của đối tượng trong vùng đó và các tọa độ  $x, y$  của góc trên bên trái và góc dưới bên phải.

```
rois = np.array([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Vì chiều cao và chiều rộng của  $X$  bằng  $1/10$  chiều cao và chiều rộng của ảnh, các tọa độ của hai vùng được đề xuất sẽ nhân với  $0.1$  thông qua spatial\_scale, tiếp theo các ROI được gán nhãn trên  $X$  lần lượt là  $X[:, :, 0:3, 0:3]$  và  $X[:, :, 1:4, 0:4]$ . Sau cùng, ta chia hai ROI thành một lưới cửa sổ con và trích xuất đặc trưng với chiều cao và chiều rộng là 2.

```
npx.roi_pooling(X, rois, pooled_size=(2, 2), spatial_scale=0.1)
```

### 15.8.3 Faster R-CNN

Để có kết quả phát hiện đối tượng chính xác, Fast R-CNN thường đòi hỏi tạo ra nhiều vùng đề xuất khi tìm kiếm chọn lọc. Faster R-CNN thay thế tìm kiếm chọn lọc bằng mạng đề xuất vùng. Mạng này làm giảm số vùng đề xuất, trong khi vẫn đảm bảo phát hiện chính xác đối tượng.

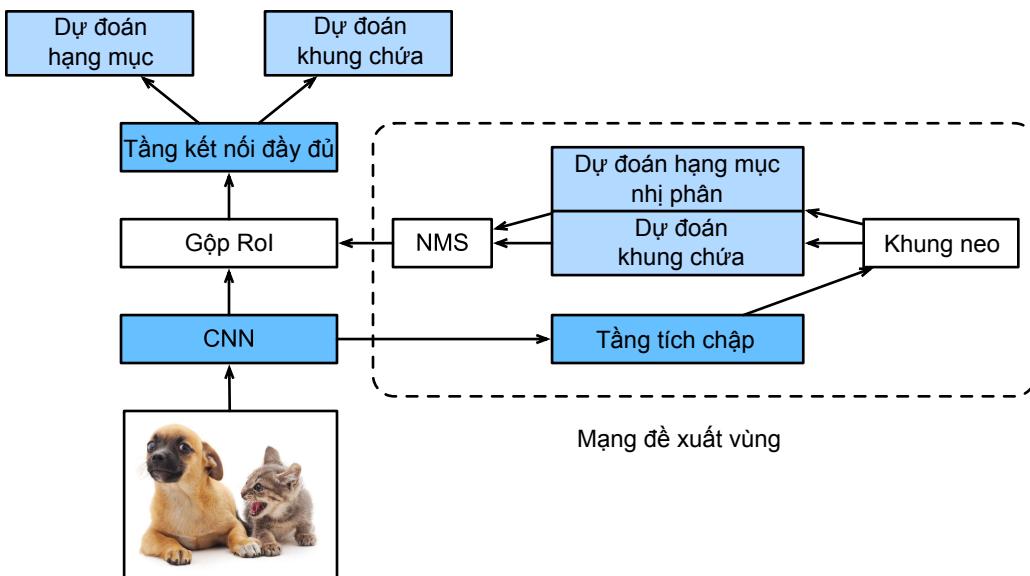


Fig. 15.8.4: Mô hình Faster R-CNN.

Fig. 15.8.4 minh họa mô hình Faster R-CNN. So với Fast R-CNN, Faster R-CNN chỉ thay thế phương pháp sản sinh các vùng đề xuất từ tìm kiếm chọn lọc sang mạng đề xuất vùng. Những phần còn lại trong mô hình không đổi. Quy trình tính toán của mạng đề xuất vùng được mô tả chi tiết dưới đây:

1. Dùng một tầng tích chập  $3 \times 3$  với đệm bằng 1 để biến đổi đầu ra của CNN và đặt số kênh đầu ra bằng  $c$ . Bằng cách này, mỗi phần tử trong ảnh xạ đặc trưng mà CNN trích xuất ra từ bức ảnh là một đặc trưng mới có độ dài bằng  $c$ .
2. Lấy mỗi phần tử trong ảnh xạ đặc trưng làm tâm để tạo ra nhiều khung neo có kích thước và tỷ lệ khác nhau, sau đó gán nhãn cho chúng.
3. Lấy những đặc trưng của các phần tử có độ dài  $c$  ở tâm khung neo để phân loại nhị phân (là vật thể hay là nền) và dự đoán khung chứa tương ứng cho các khung neo.
4. Sau đó, sử dụng triệt phi cực đại (*non-maximum suppression*) để loại bỏ các khung chứa có kết quả giống nhau của hạng mục “vật thể”. Cuối cùng, ta xuất ra các khung chứa dự đoán là các vùng đề xuất rồi đưa vào tầng gộp ROI.

Lưu ý rằng, vì là một phần của mô hình Faster R-CNN, nên mạng đề xuất vùng được huấn luyện cùng với phần còn lại trong mô hình. Ngoài ra, trong đối tượng Faster R-CNN còn chứa các hàm dự đoán hạng mục và khung chứa trong bài toán phát hiện vật thể, cũng như các hàm dự đoán hạng mục nhị phân và khung chứa cho các khung neo trong mạng đề xuất vùng. Sau cùng, mạng đề xuất vùng có thể học được cách sinh ra những vùng đề xuất có chất lượng cao, giảm đi số lượng vùng đề xuất trong khi vẫn giữ được độ chính xác khi phát hiện vật thể.

#### 15.8.4 Mask R-CNN

Nếu dữ liệu huấn luyện được gán nhãn với các vị trí ở cấp độ từng điểm ảnh trong bức hình, thì mô hình Mask R-CNN có thể sử dụng hiệu quả các nhãn chi tiết này để cải thiện độ chính xác của việc phát hiện đối tượng.

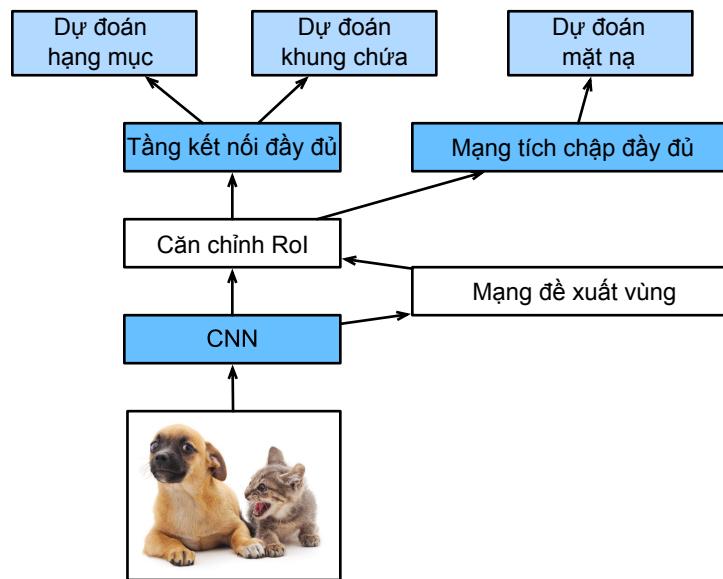


Fig. 15.8.5: Mô hình Mask R-CNN.

Trong Fig. 15.8.5, có thể thấy Mask R-CNN là một sự hiệu chỉnh của Faster R-CNN. Mask R-CNN thay thế tầng tổng hợp ROI bằng tầng căn chỉnh ROI (*ROI alignment layer*). Điều này cho phép sử dụng phép nội suy song tuyến tính (*bilinear interpolation*) để giữ lại thông tin không gian trong ảnh xạ đặc trưng, làm cho Mask R-CNN trở nên phù hợp hơn với các dự đoán cấp điểm ảnh. Lớp căn chỉnh ROI xuất ra các ảnh xạ đặc trưng có cùng kích thước cho mọi ROI. Điều này không những dự đoán các lớp và khung chứa của ROI, mà còn cho phép chúng ta bổ sung một mạng nơ-ron tích chập đầy đủ (*fully convolutional network*) để dự đoán vị trí cấp điểm ảnh của các đối tượng. Chúng tôi sẽ mô tả cách sử dụng mạng nơ-ron tích chập đầy đủ để dự đoán ngữ nghĩa cấp điểm ảnh ở phần sau của chương này.

#### 15.8.5 Tóm tắt

- Mô hình R-CNN chọn ra nhiều vùng đề xuất và sử dụng CNN để thực hiện tính toán truyền xuôi rồi trích xuất đặc trưng từ mỗi vùng đề xuất. Sau đó dùng các đặc trưng này để dự đoán hạng mục và khung chứa của những vùng đề xuất.
- Fast R-CNN cải thiện R-CNN bằng cách chỉ thực hiện tính toán truyền xuôi CNN trên toàn bộ bức ảnh. Mạng này sử dụng một tầng gộp ROI để trích xuất các đặc trưng có cùng kích thước từ các vùng quan tâm có kích thước khác nhau.
- Faster R-CNN thay thế tìm kiếm chọn lọc trong Fast R-CNN bằng mạng đề xuất vùng. Điều này làm giảm số lượng vùng đề xuất tạo ra, nhưng vẫn đảm bảo độ chính xác khi phát hiện đối tượng.
- Mask R-CNN có cấu trúc cơ bản giống Faster R-CNN, nhưng có thêm một mạng nơ-ron tích chập đầy đủ giúp định vị đối tượng ở cấp điểm ảnh và cải thiện hơn nữa độ chính xác của việc phát hiện đối tượng.

### 15.8.6 Bài tập

Tìm hiểu cách thực thi từng mô hình trong bộ công cụ GluonCV<sup>313</sup> liên quan đến phần này.

### 15.8.7 Thảo luận

- Tiếng Anh<sup>314</sup>
- Tiếng Việt<sup>315</sup>

### 15.8.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Nguyễn Mai Hoàng Long
- Phạm Đăng Khoa
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Nguyễn Văn Cường

## 15.9 Phân vùng theo Ngữ nghĩa và Tập dữ liệu

Khi thảo luận ở những phần trước về các vấn đề liên quan tới phát hiện vật thể, chúng ta chỉ sử dụng các khung chứa chữ nhật để gán nhãn và dự đoán các vật thể trong ảnh. Trong phần này, ta sẽ xem xét việc phân vùng theo ngữ nghĩa (*semantic segmentation*), tức là phân chia ảnh thành các vùng với hạng mục ngữ nghĩa khác nhau. Các vùng ngữ nghĩa đó gán nhãn và dự đoán các đối tượng ở mức độ điểm ảnh. Fig. 15.9.1 minh họa một ảnh đã được phân vùng ngữ nghĩa, với các vùng được gán nhãn “chó”, “mèo” và “nền”. Như bạn có thể thấy, so với việc phát hiện vật thể, việc phân vùng theo ngữ nghĩa sẽ gán nhãn các vùng theo đường biên ở mức điểm ảnh, đem lại độ chính xác lớn hơn đáng kể.

<sup>313</sup> <https://github.com/dmlc/gluon-cv/>

<sup>314</sup> <https://discuss.d2l.ai/t/374>

<sup>315</sup> <https://forum.machinelearningcoban.com/c/d2l>

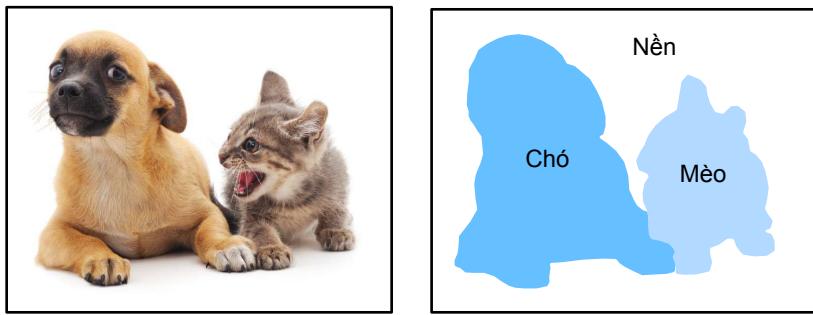


Fig. 15.9.1: Ảnh được phân vùng theo ngữ nghĩa, với các vùng được gán nhãn “chó”, “mèo” và “nền”.

### 15.9.1 Phân vùng Ảnh và Phân vùng Thực thể

Trong lĩnh vực thị giác máy tính, có hai phương pháp quan trọng liên quan tới phân vùng theo ngữ nghĩa, đó là: phân vùng ảnh và phân vùng thực thể. Ta phân biệt các khái niệm này với phân vùng theo ngữ nghĩa như sau:

- Phân vùng ảnh (*Image segmentation*) chia một bức ảnh thành các vùng thành phần. Phương pháp này thường sử dụng độ tương quan giữa các điểm ảnh trên ảnh. Trong suốt quá trình huấn luyện, nhãn cho các điểm ảnh là không cần thiết. Tuy nhiên, trong quá trình dự đoán, phương pháp này có thể không đảm bảo các vùng được phân chia chứa ngữ nghĩa mà ta mong muốn. Nếu ta đưa vào bức ảnh trong Fig. 15.9.1, phân vùng ảnh có thể chia con chó thành hai vùng, một vùng bao phủ phần miệng và cặp mắt nơi màu đen là chủ đạo và vùng thứ hai phủ trên phần còn lại của chú chó nơi màu vàng chiếm ưu thế.
- Phân vùng thực thể (*Instance segmentation*) còn được gọi là phát hiện và phân vùng đồng thời. Phương pháp này cố gắng xác định các vùng ở mức điểm ảnh theo từng đối tượng riêng biệt ngay trong ảnh. Khác với phân vùng theo ngữ nghĩa, phân vùng thực thể không chỉ phân biệt ngữ nghĩa mà còn cả các thực thể khác nhau. Nếu một ảnh có chứa hai chú chó, phân vùng thực thể sẽ phân biệt những điểm ảnh thuộc về từng chú chó.

### 15.9.2 Tập dữ liệu Phân vùng theo Ngữ nghĩa Pascal VOC2012

Trong phân vùng theo ngữ nghĩa, [Pascal VOC2012<sup>316</sup>](#) là một tập dữ liệu quan trọng. Để hiểu rõ hơn về tập dữ liệu này, đầu tiên ta nhập vào gói thư viện hoặc mô-đun cần thiết.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, np, npx
import os

npx.set_np()
```

Trang gốc có thể không ổn định nên ta tải dữ liệu về từ một trang nhân bản. Tập tin nặng khoảng 2 GB nên thời gian tải về có thể sẽ hơi lâu. Sau khi giải nén tập tin, tập dữ liệu được lưu tại đường dẫn `../data/VOCdevkit/VOC2012`.

<sup>316</sup> <http://host.robots.ox.ac.uk/pascal/VOC/voc2012/>

```

#@save
d2l.DATA_HUB['voc2012'] = (d2l.DATA_URL + 'VOCtrainval_11-May-2012.tar',
                            '4e443f8a2eca6b1dac8a6c57641b67dd40621a49')

voc_dir = d2l.download_extract('voc2012', 'VOCdevkit/VOC2012')

```

Đi tới thư mục `./data/VOCdevkit/VOC2012` để quan sát các phần khác nhau của tập dữ liệu. Thư mục `ImageSets/Segmentation` chứa các tệp văn bản định rõ ví dụ để huấn luyện và kiểm tra. Thư mục `JPEGImages` và `SegmentationClass` lần lượt chứa các mẫu ảnh đầu vào và nhãn. Các nhãn này cũng mang định dạng ảnh, với số chiều bằng với ảnh đầu vào tương ứng. Trong các nhãn, các điểm ảnh cùng màu thì thuộc cùng hạng mục ngữ nghĩa. Hàm `read_voc_images` định nghĩa dưới đây đọc và lưu vào bộ nhớ tất cả các ảnh đầu vào và nhãn tương ứng.

```

#@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')
    with open(txt_fname, 'r') as f:
        images = f.read().split()
    features, labels = [], []
    for i, fname in enumerate(images):
        features.append(image.imread(os.path.join(
            voc_dir, 'JPEGImages', f'{fname}.jpg')))
        labels.append(image.imread(os.path.join(
            voc_dir, 'SegmentationClass', f'{fname}.png')))
    return features, labels

train_features, train_labels = read_voc_images(voc_dir, True)

```

Ta vẽ năm ảnh đầu vào đầu tiên và nhãn của chúng. Trong ảnh nhãn, màu trắng biểu diễn viền và màu đen biểu diễn nền. Các màu khác tương ứng với các hạng mục khác nhau.

```

n = 5
imgs = train_features[0:n] + train_labels[0:n]
d2l.show_images(imgs, 2, n);

```

Tiếp theo, ta liệt kê từng giá trị màu RGB của các nhãn và hạng mục của nhãn đó.

```

#@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']

```

Sau khi khai báo hai hằng số trên, ta có thể dễ dàng tìm chỉ số hạng mục cho mỗi điểm ảnh trong

các nhãn.

```
#@save
def build_colormap2label():
    """Build an RGB color to label mapping for segmentation."""
    colormap2label = np.zeros(256 ** 3)
    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):
    """Map an RGB color to a label."""
    colormap = colormap.astype(np.int32)
    idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
           + colormap[:, :, 2])
    return colormap2label[idx]
```

Ví dụ, trong ảnh ví dụ đầu tiên, phần đầu máy bay có chỉ số hạng mục là một và chỉ số của nền là 0.

```
y = voc_label_indices(train_labels[0], build_colormap2label())
y[105:115, 130:140], VOC_CLASSES[1]
```

## Tiền xử lý Dữ liệu

Trong chương trước, ta biến đổi tỷ lệ của ảnh để khớp với kích thước đầu vào của mô hình. Với phương pháp phân vùng theo ngữ nghĩa, ta phải tái ánh xạ hạng mục được dự đoán của điểm ảnh về kích thước gốc của ảnh đầu vào. Sẽ rất khó để thực hiện việc này một cách chính xác, nhất là ở các phân vùng mang ngữ nghĩa khác nhau. Để tránh vấn đề này, ta cắt ảnh để chỉnh kích thước chứ không biến đổi tỷ lệ ảnh. Cụ thể, ta sử dụng phương pháp cắt ngẫu nhiên được sử dụng trong kỹ thuật tăng cường ảnh để cắt cùng một vùng từ cả ảnh đầu vào và nhãn của nó.

```
#@save
def voc_rand_crop(feature, label, height, width):
    """Randomly crop for both feature and label images."""
    feature, rect = image.random_crop(feature, (width, height))
    label = image.fixed_crop(label, *rect)
    return feature, label

imgs = []
for _ in range(n):
    imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```

## Các lớp của Tập dữ liệu cho Phân vùng theo Ngữ nghĩa Tuỳ chỉnh

Ta kế thừa lớp Dataset cung cấp bởi Gluon để tùy chỉnh tập dữ liệu phân vùng theo ngữ nghĩa VOCSegDataset. Bằng việc lập trình hàm `__getitem__`, ta có thể tùy ý truy cập từ tập dữ liệu ảnh đầu vào với chỉ số idx và các chỉ số hạng mục của từng điểm ảnh trong ảnh đó. Do một số ảnh trong tập dữ liệu có thể nhỏ hơn chiều đầu ra mong muốn trong phép cắt ngẫu nhiên, ta cần loại bỏ các ví dụ đó bằng hàm filter. Thêm vào đó, ta định nghĩa hàm `normalize_image` để chuẩn hoá từng kênh RGB của các ảnh đầu vào.

```
#@save
class VOCSegDataset(gluon.data.Dataset):
    """A customized dataset to load VOC dataset"""

    def __init__(self, is_train, crop_size, voc_dir):
        self.rgb_mean = np.array([0.485, 0.456, 0.406])
        self.rgb_std = np.array([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 = build_colormap2label()
        print('read ' + str(len(self.features)) + ' examples')

    def normalize_image(self, img):
        return (img.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

    def filter(self, imgs):
        return [img for img in imgs if (
            img.shape[0] >= self.crop_size[0] and
            img.shape[1] >= self.crop_size[1])]

    def __getitem__(self, idx):
        feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                         *self.crop_size)
        return (feature.transpose(2, 0, 1),
                voc_label_indices(label, self.colormap2label))

    def __len__(self):
        return len(self.features)
```

## Đọc tập Dữ liệu

Sử dụng lớp VOCSegDataset được tuỳ chỉnh ở trên, ta có thể khởi tạo đối tượng tập huấn luyện và tập kiểm tra. Giả sử thao tác cắt ngẫu nhiên tạo ra ảnh có kích thước  $320 \times 480$ . Dưới đây ta có thể thấy số lượng ảnh được giữ lại trong tập huấn luyện và tập kiểm tra.

```
crop_size = (320, 480)
voc_train = VOCSegDataset(True, crop_size, voc_dir)
voc_test = VOCSegDataset(False, crop_size, voc_dir)
```

Ta đặt kích thước batch là 64 và định nghĩa các iterator cho tập huấn luyện và tập kiểm tra. Sau đó ta sẽ in ra kích thước của minibatch đầu tiên. Khác với phân loại ảnh và nhận dạng vật thể, các nhãn ở đây là các mảng ba chiều.

```

batch_size = 64
train_iter = gluon.data.DataLoader(voc_train, batch_size, shuffle=True,
                                   last_batch='discard',
                                   num_workers=d2l.get_dataloader_workers())
for X, Y in train_iter:
    print(X.shape)
    print(Y.shape)
    break

```

### Kết hợp Tất cả lại với nhau

Cuối cùng, ta định nghĩa hàm load\_data\_voc để tải xuống và nạp tập dữ liệu, sau đó trả về các iterator dữ liệu.

```

#@save
def load_data_voc(batch_size, crop_size):
    """Download and load the VOC2012 semantic dataset."""
    voc_dir = d2l.download_extract('voc2012', os.path.join(
        'VOCdevkit', 'VOC2012'))
    num_workers = d2l.get_dataloader_workers()
    train_iter = gluon.data.DataLoader(
        VOCSegDataset(True, crop_size, voc_dir), batch_size,
        shuffle=True, last_batch='discard', num_workers=num_workers)
    test_iter = gluon.data.DataLoader(
        VOCSegDataset(False, crop_size, voc_dir), batch_size,
        last_batch='discard', num_workers=num_workers)
    return train_iter, test_iter

```

### 15.9.3 Tóm tắt

- Phân vùng theo ngữ nghĩa tập trung vào việc phân vùng ảnh thành các vùng với hạng mục ngữ nghĩa khác nhau.
- Trong lĩnh vực này, Pascal VOC2012 là một tập dữ liệu quan trọng.
- Do các ảnh đầu vào và nhãn trong phân vùng ảnh theo ngữ nghĩa có mối tương quan một-một ở cấp độ điểm ảnh, ta cắt các ảnh này một cách ngẫu nhiên theo kích thước cố định thay vì biến đổi tỷ lệ của chúng.

### 15.9.4 Bài tập

Xem lại nội dung được trình bày trong Section 15.1. Phương pháp tăng cường ảnh nào sử dụng trong phân loại ảnh có thể khó sử dụng trong phân vùng ảnh theo ngữ nghĩa?

### 15.9.5 Thảo luận

- Tiếng Anh - MXNet<sup>317</sup>
- Tiếng Việt<sup>318</sup>

### 15.9.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Nguyễn Văn Cường

## 15.10 Tích chập Chuyển vị

Các tầng trong mạng nơ-ron tích chập, bao gồm tầng tích chập (Section 8.2) và tầng gộp (Section 8.5), thường giảm chiều rộng và chiều cao của đầu vào, hoặc giữ nguyên chúng. Tuy nhiên, các ứng dụng như phân vùng theo ngữ nghĩa (Section 15.9) và mạng đối sinh (GAN - Section 19.2), yêu cầu phải dự đoán các giá trị cho mỗi pixel vì thế cần tăng chiều rộng và chiều cao của đầu vào. Tích chập chuyển vị, cũng có tên là tích chập sải bước phân số (*fractionally-strided convolution*) (Dumoulin & Visin, 2016) hay phân tách tích chập (*deconvolution*) (Long et al., 2015), phục vụ cho mục đích này.

```
from mxnet import np, npx, init
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

<sup>317</sup> <https://discuss.d2l.ai/t/375>

<sup>318</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 15.10.1 Tích chập Chuyển vị 2D Cơ bản

Xét một trường hợp cơ bản với số kênh đầu vào và đầu ra là 1, với đệm 0 và sai bước 1. Fig. 15.10.1 mô tả cách tích chập chuyển vị với một hạt nhân  $2 \times 2$  được tính toán trên một ma trận đầu vào kích thước  $2 \times 2$ .

| Đầu vào                                      | Hạt nhân                                     | = | $\begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}$ |  |   | +   | $\begin{matrix} & 0 & 1 \\ & 2 & 3 \end{matrix}$ | + | $\begin{matrix} & & & \\ 0 & 2 & & \\ 4 & 6 & & \end{matrix}$ | + | $\begin{matrix} & & & \\ & 0 & 3 & \\ 6 & 9 & & \end{matrix}$ | = | Đầu ra |
|--|--|---|--|--|---|---|--|---|---|---|---|---|--------|
| $\begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix}$ | $\begin{matrix} 0 & 1 \\ 2 & 3 \end{matrix}$ |   | $\begin{matrix} 0 & 0 \\ 0 & 0 \end{matrix}$ | $\begin{matrix} & 0 & 1 \\ & 2 & 3 \end{matrix}$ | $\begin{matrix} & & & \\ 0 & 2 & & \\ 4 & 6 & & \end{matrix}$ | $\begin{matrix} & & & \\ & 0 & 3 & \\ 6 & 9 & & \end{matrix}$ |  |   |   |   |   |   |        |

Fig. 15.10.1: Tầng tích chập chuyển vị với hạt nhân  $2 \times 2$ .

Ta có thể lập trình phép tính này với ma trận hạt nhân  $K$  và ma trận đầu vào  $X$ .

```
def trans_conv(X, K):
    h, w = K.shape
    Y = np.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
```

Nhớ lại rằng kết quả tích chập là  $Y[i, j] = (X[i: i + h, j: j + w] * K).sum()$  (đọc lại corr2d trong Section 8.2), tức tổng hợp các giá trị đầu vào thông qua hạt nhân. Trong khi tích chập chuyển vị lan truyền từng giá trị đầu vào khắp hạt nhân, tạo thành đầu ra có kích thước lớn hơn.

Kiểm chứng các kết quả trong Fig. 15.10.1.

```
X = np.array([[0, 1], [2, 3]])
K = np.array([[0, 1], [2, 3]])
trans_conv(X, K)
```

Hoặc ta có thể sử dụng nn.Conv2DTranspose để thu được kết quả tương tự. Vì đang sử dụng nn.Conv2D, cả đầu vào và hạt nhân phải là tensor 4 chiều.

```
X, K = X.reshape(1, 1, 2, 2), K.reshape(1, 1, 2, 2)
tconv = nn.Conv2DTranspose(1, kernel_size=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

### 15.10.2 Đệm, Sải bước và Kênh

Khi tính tích chập ta áp dụng đệm lên đầu vào, nhưng với tích chập chuyển vị, chúng được áp dụng vào đầu ra. Ví dụ, với đệm  $1 \times 1$ , đầu tiên ta tính toán đầu ra như bình thường, sau đó bỏ đi dòng và cột đầu tiên / cuối cùng.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, padding=1)
tconv.initialize(init.Constant(K))
tconv(X)
```

Tương tự, sải bước cũng được áp dụng vào các đầu ra.

```
tconv = nn.Conv2DTranspose(1, kernel_size=2, strides=2)
tconv.initialize(init.Constant(K))
tconv(X)
```

Phần mở rộng đa kênh của tích chập chuyển vị cũng giống như tích chập. Khi đầu vào có  $c_i$  kênh, tích chập chuyển vị gán một ma trận hạt nhân có kích thước  $k_h \times k_w$  vào mỗi kênh đầu vào. Nếu số kênh đầu ra là  $c_o$ , thì hạt nhân có kích thước  $c_i \times k_h \times k_w$  cho mỗi kênh đầu ra.

Do đó, nếu ta đưa  $X$  qua một tầng tích chập  $f$  để tính  $Y = f(X)$  và tạo một tầng tích chập chuyển vị  $g$  với cùng một siêu tham số như  $f$  ngoại trừ kênh đầu ra được đặt thành kích thước kênh  $X$ , thì  $g(Y)$  sẽ có cùng kích thước với  $X$ . Ta hãy xác minh phát biểu này.

```
X = np.random.uniform(size=(1, 10, 16, 16))
conv = nn.Conv2D(20, kernel_size=5, padding=2, strides=3)
tconv = nn.Conv2DTranspose(10, kernel_size=5, padding=2, strides=3)
conv.initialize()
tconv.initialize()
tconv(conv(X)).shape == X.shape
```

### 15.10.3 Sự Tương đồng với Chuyển vị Ma trận

Tên của tích chập chuyển vị xuất phát từ phép chuyển vị ma trận. Thực vậy, phép tích chập có thể tính thông qua phép nhân ma trận. Trong ví dụ dưới đây, ta định nghĩa một biến đầu vào  $X$   $3 \times 3$  với một hạt nhân  $K$   $2 \times 2$ , rồi dùng `corr2d` để tính tích chập.

```
X = np.arange(9).reshape(3, 3)
K = np.array([[0, 1], [2, 3]])
Y = d2l.corr2d(X, K)
Y
```

Kế tiếp, ta viết lại hạt nhân chập  $K$  dưới dạng ma trận  $W$ . Kích thước của nó sẽ là  $(4, 9)$ , hàng thứ  $i$  biểu diễn việc sử dụng hạt nhân lén đầu vào để sinh ra phần tử đầu ra thứ  $i$ .

```
def kernel2matrix(K):
    k, W = np.zeros(5), np.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
```

(continues on next page)

```
W = kernel2matrix(K)
W
```

Rồi toán tử chập có thể được thực hiện nhờ phép nhân ma trận với việc chỉnh lại kích thước phù hợp.

```
Y == np.dot(W, X.reshape(-1)).reshape(2, 2)
```

Ta có thể thực hiện phép chập chuyển vị giống như phép nhân ma trận bằng cách sử dụng lại `kernel2matrix`. Để sử dụng lại ma trận  $W$  đã tạo ra, ta xây dựng một đầu vào  $2 \times 2$ , nên ma trận trọng số  $W^\top$  tương ứng sẽ có kích thước  $(9, 4)$ . Ta hãy cùng kiểm tra lại kết quả.

```
X = np.array([[0, 1], [2, 3]])
Y = trans_conv(X, K)
Y == np.dot(W.T, X.reshape(-1)).reshape(3, 3)
```

#### 15.10.4 Tóm tắt

- So với phương pháp tích chập nén đầu vào thông qua hạt nhân, phép tích chập chuyển vị làm tăng số chiều của đầu vào.
- Nếu một tầng tích chập nén chiều rộng và chiều cao của đầu vào lần lượt đi  $n_w$  và  $n_h$  lần, thì một tầng tích chập chuyển vị có cùng kích thước hạt nhân, đệm và sải bước sẽ tăng chiều dài và chiều cao của đầu vào lần lượt lên  $n_w$  và  $n_h$  lần.
- Ta có thể lập trình thao tác tích chập bằng phép nhân ma trận, và phép tích chập chuyển vị tương ứng cũng có thể thực hiện bằng phép nhân ma trận chuyển vị.

#### 15.10.5 Bài tập

Việc sử dụng phép nhân ma trận để lập trình cho thao tác tích chập liệu có thực sự hiệu quả? Tại sao?

#### 15.10.6 Thảo luận

- Tiếng Anh - MXNet<sup>319</sup>
- Tiếng Việt<sup>320</sup>

<sup>319</sup> <https://discuss.d2l.ai/t/376>

<sup>320</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 15.10.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh

## 15.11 Mạng Tích chập Đầy đủ

Ở phần trước, chúng ta đã thảo luận về phân vùng theo ngữ nghĩa bằng cách dự đoán hạng mục trên từng điểm ảnh. Mạng tích chập đầy đủ (*fully convolutional network* - FCN) (Long et al., 2015) sử dụng mạng nơ-ron tích chập để biến đổi các điểm ảnh thành các hạng mục tương ứng của điểm ảnh. Khác với các mạng nơ-ron tích chập được giới thiệu trước đây, mạng FCN biến đổi chiều cao và chiều rộng của ánh xạ đặc trưng ở tầng trung gian về kích thước ảnh đầu vào thông qua các tầng tích chập chuyển vị, sao cho các dự đoán có sự tương xứng một-một với ảnh đầu vào theo không gian (chiều cao và chiều rộng). Với một vị trí trên chiều không gian, đầu ra theo chiều kênh sẽ là hạng mục được dự đoán tương ứng với điểm ảnh tại vị trí đó.

Đầu tiên, ta sẽ nhập gói thư viện và mô-đun cần thiết cho thí nghiệm này, sau đó sẽ đi đến giải thích về tầng tích chập chuyển vị.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

### 15.11.1 Xây dựng Mô hình

Ở đây, chúng tôi sẽ trình bày một thiết kế cơ bản nhất của mô hình tích chập đầy đủ. Như mô tả trong hình Fig. 15.11.1, đầu tiên mạng tích chập đầy đủ sử dụng mạng nơ-ron tích chập để trích xuất đặc trưng ảnh, sau đó biến đổi số lượng kênh thành số lượng hạng mục thông qua tầng tích chập  $1 \times 1$ , và cuối cùng biến đổi chiều cao và chiều rộng của ánh xạ đặc trưng bằng với kích thước của ảnh đầu vào bằng cách sử dụng tầng tích chập chuyển vị Section 15.10. Đầu ra của mạng có cùng chiều cao và chiều rộng với ảnh gốc, đồng thời cũng có sự tương xứng một-một theo vị trí không gian. Kênh đầu ra cuối cùng chứa hạng mục dự đoán của từng điểm ảnh ở vị trí không gian tương ứng.

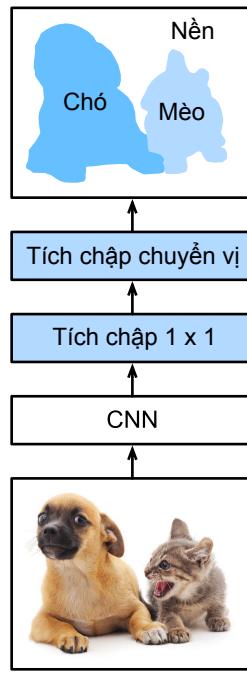


Fig. 15.11.1: Mạng tích chập đầy đủ.

Dưới đây, ta sử dụng mô hình ResNet-18 đã được tiền huấn luyện trên ImageNet để trích xuất đặc trưng và lưu thực thể mô hình là `pretrained_net`. Như ta có thể thấy, hai tầng cuối của mô hình nằm trong biến thành viên `features` là tầng gộp cực đại toàn cục GlobalAvgPool2D và tầng trải phẳng `Flatten`. Mô-đun `output` chứa tầng kết nối đầy đủ được sử dụng cho đầu ra. Các tầng này không bắt buộc phải có trong mạng tích chập đầy đủ.

```
pretrained_net = gluon.model_zoo.vision.resnet18_v2(pretrained=True)
pretrained_net.features[-4:], pretrained_net.output
```

Tiếp theo, ta khởi tạo một thực thể mạng tích chập đầy đủ `net`. Thực thể này sao chép tất cả các tầng nơ-ron ngoại trừ hai tầng cuối cùng của biến thành viên `features` trong `pretrained_net` và các tham số mô hình thu được từ bước tiền huấn luyện.

```
net = nn.HybridSequential()
for layer in pretrained_net.features[:-2]:
    net.add(layer)
```

Với đầu vào có chiều cao và chiều rộng là 320 và 480, bước tính toán lượt truyền xuôi của `net` sẽ giảm chiều cao và chiều rộng của đầu vào thành 1/32 kích thước ban đầu, tức 10 và 15.

```
X = np.random.uniform(size=(1, 3, 320, 480))
net(X).shape
```

Tiếp đến, ta biến đổi số lượng kênh đầu ra bằng với số lượng hạng mục trong tập dữ liệu Pascal VOC2012 (21) thông qua tầng tích chập  $1 \times 1$ . Cuối cùng, ta cần phóng đại chiều cao và chiều rộng của ảnh xạ đặc trưng lên gấp 32 lần để bằng với chiều cao và chiều rộng của ảnh đầu vào. Hãy xem lại phương pháp tính kích thước đầu ra của tầng tích chập được mô tả trong [Section 8.3](#). Vì  $(320 - 64 + 16 \times 2 + 32)/32 = 10$  và  $(480 - 64 + 16 \times 2 + 32)/32 = 15$ , ta sẽ xây dựng một tầng tích chập chuyển vị với sải bước 32, đặt chiều dài và chiều rộng của hạt nhân tích chập bằng 64 và kích thước đệm bằng 16. Không khó để nhận ra rằng nếu sải bước bằng  $s$ , kích thước đệm là  $s/2$  (giả sử

$s/2$  là số nguyên) và hạt nhân tích chập có chiều dài và chiều rộng bằng  $2s$ , thì hạt nhân tích chập chuyển vị sẽ phóng đại chiều cao và chiều rộng của đầu vào lên  $s$  lần.

```
num_classes = 21
net.add(nn.Conv2D(num_classes, kernel_size=1),
        nn.Conv2DTranspose(
            num_classes, kernel_size=64, padding=16, strides=32))
```

### 15.11.2 Khởi tạo Tầng Tích chập Chuyển vị

Chúng ta đã biết tầng tích chập chuyển vị có thể phóng đại ánh xạ đặc trưng. Trong xử lý ảnh, đôi khi ta cần phóng đại ảnh hay còn được biết đến là phép tăng mẫu. Có rất nhiều phương pháp tăng mẫu, và một phương pháp phổ biến là nội suy song tuyến tính (*bilinear interpolation*). Nói một cách đơn giản, để lấy điểm ảnh của ảnh đầu ra tại tọa độ  $(x, y)$ , đầu tiên tọa độ này sẽ được ánh xạ tới tọa độ  $(x', y')$  trong ảnh đầu vào. Điều này có thể được thực hiện dựa trên tỷ lệ kích thước của ba đầu vào tới kích thước của đầu ra. Giá trị được ánh xạ  $x'$  và  $y'$  thường là các số thực. Sau đó, ta tìm bốn điểm ảnh gần tọa độ  $(x', y')$  nhất trên ảnh đầu vào. Cuối cùng, các điểm ảnh tại tọa độ  $(x, y)$  của đầu ra sẽ được tính toán dựa trên bốn điểm ảnh của ảnh đầu vào và khoảng cách tương đối của chúng tới  $(x', y')$ . Phép tăng mẫu bằng nội suy song tuyến tính có thể được lập trình bằng tầng tích chập chuyển vị với hạt nhân tích chập được xây dựng bằng hàm `bilinear_kernel` dưới đây. Do có nhiều giới hạn, chúng tôi sẽ chỉ cung cấp cách lập trình hàm `bilinear_kernel` và sẽ không thảo luận về nguyên tắc của thuật toán.

```
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 = (np.arange(kernel_size).reshape(-1, 1),
          np.arange(kernel_size).reshape(1, -1))
    filt = (1 - np.abs(og[0] - center) / factor) * \
           (1 - np.abs(og[1] - center) / factor)
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size))
    weight[range(in_channels), range(out_channels), :, :] = filt
    return np.array(weight)
```

Bây giờ, ta sẽ thí nghiệm với phép tăng mẫu sử dụng phép nội suy song tuyến tính được thực hiện bằng tầng tích chập chuyển vị. Ta sẽ xây dựng tầng tích chập chuyển vị để phóng đại chiều cao và chiều rộng của đầu vào lên hai lần và khởi tạo hạt nhân tích chập với hàm `bilinear_kernel`.

```
conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

Ta sẽ đọc ảnh X và lưu kết quả của phép tăng mẫu là Y. Để in ảnh, ta cần điều chỉnh vị trí của chiều kênh.

```
img = image.imread('../img/catdog.jpg')
X = np.expand_dims(img.astype('float32')).transpose(2, 0, 1), axis=0) / 255
Y = conv_trans(X)
out_img = Y[0].transpose(1, 2, 0)
```

Như ta có thể thấy, tầng tích chập chuyển vị phóng đại cả chiều cao và chiều rộng của ảnh lên hai lần. Điều đáng nói là bên cạnh sự khác biệt trong tỷ lệ tọa độ, ảnh được phóng đại bởi phép nội suy song tuyến tính và ảnh ban đầu được in Section 15.3 nhìn giống nhau.

```
d2l.set_figsize()  
print('input image shape:', img.shape)  
d2l.plt.imshow(img.asnumpy());  
print('output image shape:', out_img.shape)  
d2l.plt.imshow(out_img.asnumpy());
```

Trong mạng tích chập đầy đủ, ta khởi tạo tầng tích chập chuyển vị để thực hiện phép tăng mẫu nội suy song tuyến tính. Với tầng tích chập  $1 \times 1$ , ta sử dụng phương pháp khởi tạo ngẫu nhiên Xavier.

```
W = bilinear_kernel(num_classes, num_classes, 64)  
net[-1].initialize(init.Constant(W))  
net[-2].initialize(init=init.Xavier())
```

### 15.11.3 Đọc Dữ liệu

Ta đọc dữ liệu bằng phương thức được mô tả ở phần trước. Ở đây, ta định rõ kích thước của ảnh đầu ra sau khi cắt ngẫu nhiên là  $320 \times 480$ , để cả chiều cao và chiều rộng chia hết cho 32.

```
batch_size, crop_size = 32, (320, 480)  
train_iter, test_iter = d2l.load_data_voc(batch_size, crop_size)
```

### 15.11.4 Huấn luyện

Bây giờ ta có thể bắt đầu huấn luyện mô hình. Hàm mất mát và hàm tính độ chính xác được sử dụng ở đây không quá khác biệt so với các hàm được sử dụng trong bài toán phân loại ảnh. Vì ta sử dụng kênh của tầng tích chập chuyển vị để dự đoán hạng mục cho điểm ảnh, tham số `axis=1` (chiều kênh) được định rõ trong `SoftmaxCrossEntropyLoss`.Thêm vào đó, dựa trên hạng mục của từng điểm ảnh có được dự đoán đúng hay không mà mô hình sẽ tính toán độ chính xác.

```
num_epochs, lr, wd, devices = 5, 0.1, 1e-3, d2l.try_all_gpus()  
loss = gluon.loss.SoftmaxCrossEntropyLoss(axis=1)  
net.collect_params().reset_ctx(devices)  
trainer = gluon.Trainer(net.collect_params(), 'sgd',  
                         {'learning_rate': lr, 'wd': wd})  
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

### 15.11.5 Dự đoán

Trong quá trình dự đoán, ta cần chuẩn tắc hóa ảnh đầu vào theo từng kênh và chuyển đổi chúng thành tensor 4 chiều như yêu cầu của mạng nơ-ron tích chập.

```
def predict(img):
    X = test_iter._dataset.normalize_image(img)
    X = np.expand_dims(X.transpose(2, 0, 1), axis=0)
    pred = net(X.as_in_ctx(devices[0])).argmax(axis=1)
    return pred.reshape(pred.shape[1], pred.shape[2])
```

Để biểu diễn trực quan các hạng mục được dự đoán cho từng điểm ảnh, ta ánh xạ các hạng mục dự đoán về nhãn màu trong tập dữ liệu.

```
def label2image(pred):
    colormap = np.array(d2l.VOC_COLORMAP, ctx=devices[0], dtype='uint8')
    X = pred.astype('int32')
    return colormap[X, :]
```

Kích thước và hình dạng của ảnh trong tập kiểm tra không cố định. Vì mô hình sử dụng tầng tích chập chuyển vị với sải bước bằng 32, nên khi chiều cao và chiều rộng của ảnh đầu vào không chia hết cho 32 thì chiều cao và chiều rộng của đầu ra tầng tích chập chuyển vị sẽ chênh lệch so với kích thước của ảnh đầu vào. Để giải quyết vấn đề này, ta có thể cắt nhiều vùng hình chữ nhật trong ảnh với chiều cao và chiều rộng chia hết cho 32, sau đó thực hiện lượt truyền xuôi trên các điểm ảnh của những vùng này. Khi kết hợp các kết quả lại, các vùng này sẽ khôi phục lại toàn bộ ảnh đầu vào. Khi một điểm ảnh nằm trong nhiều vùng khác nhau, trung bình đầu ra của tầng tích chập chuyển vị sau lan truyền xuôi của các vùng khác nhau có thể được sử dụng để làm đầu vào cho phép toán softmax nhằm dự đoán hạng mục.

Để đơn giản, ta chỉ đọc một vài ảnh kiểm tra có kích thước lớn và cắt các vùng với kích thước  $320 \times 480$  từ góc trái trên cùng của ảnh, và chỉ sử dụng vùng này để dự đoán. Với ảnh đầu vào, đầu tiên ta in ra vùng được cắt, sau đó in ra kết quả dự đoán, và cuối cùng in ra hạng mục nhãn gốc.

```
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, 480, 320)
    X = image.fixed_crop(test_images[i], *crop_rect)
    pred = label2image(predict(X))
    imgs += [X, pred, image.fixed_crop(test_labels[i], *crop_rect)]
d2l.show_images(imgs[:3] + imgs[1::3] + imgs[2::3], 3, n, scale=2);
```

### 15.11.6 Tóm tắt

- Đầu tiên, mạng tích chập đầy đủ sử dụng một mạng nơ-ron tích chập để trích xuất đặc trưng ảnh, sau đó biến đổi số lượng kênh thành số lượng các hạng mục bằng tầng tích chập  $1 \times 1$ , và cuối cùng biến đổi chiều cao và chiều rộng của ảnh xạ đặc trưng thành kích thước ban đầu của ảnh bằng cách sử dụng tầng tích chập chuyển vị để cho ra hạng mục của từng điểm ảnh.
- Trong mạng tích chập đầy đủ, ta khởi tạo tầng tích chập chuyển vị cho phép tăng mẫu nội suy song tuyến tính.

### 15.11.7 Bài tập

- Nếu ta sử dụng Xavier để khởi tạo ngẫu nhiên tầng tích chập chuyển vị, kết quả thay đổi ra sao?
- Bạn có thể cải thiện độ chính xác của mô hình bằng cách điều chỉnh các siêu tham số hay không?
- Hãy dự đoán các hạng mục của tất cả các điểm ảnh trong ảnh kiểm tra.
- Trong bài báo về mạng tích chập đầy đủ [1], đầu ra của một số tầng trung gian của mạng nơ-ron tích chập cũng được sử dụng. Hãy thử lập trình lại ý tưởng này.

### 15.11.8 Thảo luận

- Tiếng Anh - MXNet<sup>321</sup>
- Tiếng Việt<sup>322</sup>

### 15.11.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường

<sup>321</sup> <https://discuss.d2l.ai/t/377>

<sup>322</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 15.12 Truyền tải Phong cách Nơ-ron

Nếu có sử dụng qua những ứng dụng mạng xã hội hoặc là một nhiếp ảnh gia không chuyên, chắc hẳn bạn cũng đã quen thuộc với những loại kính lọc (*filter*). Kính lọc có thể biến đổi tông màu của ảnh để làm cho khung cảnh phía sau sắc nét hơn hoặc khuôn mặt của những người trong ảnh trở nên trắng trèo hơn. Tuy nhiên, thường một kính lọc chỉ có thể thay đổi một khía cạnh của bức ảnh. Để có được bức ảnh hoàn hảo, ta thường phải thử nghiệm kết hợp nhiều kính lọc khác nhau. Quá trình này phức tạp ngang với việc tinh chỉnh siêu tham số của mô hình.

Trong phần này, ta sẽ thảo luận cách sử dụng mạng nơ-ron tích chập (CNN) để tự động áp dụng phong cách của ảnh này cho ảnh khác. Thao tác này được gọi là truyền tải phong cách (*style transfer*) (Gatys et al., 2016). Ở đây ta sẽ cần hai ảnh đầu vào, một ảnh nội dung và một ảnh phong cách. Ta sẽ dùng mạng nơ-ron để biến đổi ảnh nội dung sao cho phong cách của nó giống như ảnh phong cách đã cho. Trong Fig. 15.12.1, ảnh nội dung là một bức ảnh phong cảnh được tác giả chụp ở công viên quốc gia Mount Rainier, gần Seattle. Ảnh phong cách là một bức tranh sơn dầu vẽ cây gỗ sồi vào mùa thu. Ảnh kết hợp đầu ra giữ lại được hình dạng tổng thể của các vật trong ảnh nội dung, nhưng được áp dụng phong cách tranh sơn dầu của ảnh phong cách, nhờ đó khiến màu sắc tổng thể trở nên sống động hơn.



Fig. 15.12.1: Ảnh nội dung và ảnh phong cách đầu vào cùng với ảnh kết hợp được tạo ra từ việc truyền tải phong cách.

### 15.12.1 Kỹ thuật

Mô hình truyền tải phong cách dựa trên CNN được biểu diễn trong Fig. 15.12.2. Đầu tiên ta sẽ khởi tạo ảnh kết hợp, có thể bằng cách sử dụng ảnh nội dung. Ảnh kết hợp này là biến (tức tham số mô hình) duy nhất cần được cập nhật trong quá trình truyền tải phong cách. Sau đó, ta sẽ chọn một CNN đã được tiền huấn luyện để thực hiện trích xuất đặc trưng của ảnh. Ta không cần phải cập nhật tham số của mạng CNN này trong quá trình huấn luyện. Mạng CNN sâu sử dụng nhiều tầng nơ-ron liên tiếp để trích xuất đặc trưng của ảnh. Ta có thể chọn đầu ra của một vài tầng nhất định làm đặc trưng nội dung hoặc đặc trưng phong cách. Nếu ta sử dụng cấu trúc trong Fig. 15.12.2, mạng nơ-ron đã tiền huấn luyện sẽ chứa ba tầng tích chập. Đầu ra của tầng thứ hai là đặc trưng nội dung ảnh, trong khi đầu ra của tầng thứ nhất và thứ ba được sử dụng làm đặc trưng phong cách.

Tiếp theo, ta thực hiện lan truyền xuôi (theo hướng của các đường nét liền) để tính hàm mất mát truyền tải phong cách và lan truyền ngược (theo hướng của các đường nét đứt) để liên tục cập nhật ảnh kết hợp. Hàm mất mát được sử dụng trong việc truyền tải phong cách thường có ba phần: 1. Mất mát nội dung giúp ảnh kết hợp có đặc trưng nội dung xấp xỉ với ảnh nội dung. 2. Mất mát phong cách giúp ảnh kết hợp có đặc trưng phong cách xấp xỉ với ảnh phong cách. 3. Mất mát biến thiên toàn phần giúp giảm nhiễu trong ảnh kết hợp. Cuối cùng, sau khi huấn luyện xong, ta sẽ có tham số của mô hình truyền tải phong cách và từ đó thu được ảnh kết hợp cuối.

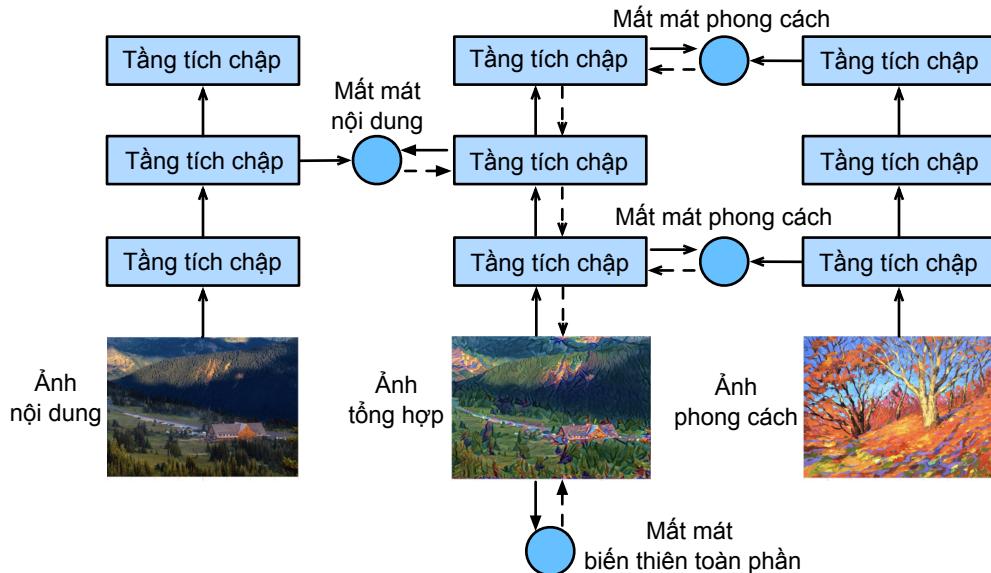


Fig. 15.12.2: Quá trình truyền tải phong cách dựa trên CNN. Các đường nét liền thể hiện hướng của lan truyền xuôi và các đường nét đứt thể hiện hướng của lan truyền ngược.

Tiếp theo, ta sẽ thực hiện một thí nghiệm để hiểu rõ hơn các chi tiết kỹ thuật của truyền tải phong cách.

### 15.12.2 Đọc ảnh Nội dung và Ảnh phong cách

Trước hết, ta đọc ảnh nội dung và ảnh phong cách. Bằng cách in ra các trực tọa độ ảnh, ta có thể thấy rằng chúng có các chiều khác nhau.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, image, init, np, npx
from mxnet.gluon import nn

npx.set_np()

d2l.set_figsize()
content_img = image.imread('../img/rainier.jpg')
d2l.plt.imshow(content_img.asnumpy());
```

```
style_img = image.imread('../img/autumn_oak.jpg')
d2l.plt.imshow(style_img.asnumpy());
```

### 15.12.3 Tiền xử lý và Hậu xử lý

Dưới đây, ta định nghĩa các hàm tiền xử lý và hậu xử lý ảnh. Hàm preprocess chuẩn hóa các kênh RGB của ảnh đầu vào và chuyển kết quả sang định dạng có thể đưa vào mạng CNN. Hàm postprocess khôi phục các giá trị điểm ảnh của ảnh đầu ra về các giá trị gốc trước khi chuẩn hóa. Vì hàm in ảnh đòi hỏi mỗi điểm ảnh có giá trị thực từ 0 tới 1, ta sử dụng hàm clip để thay thế các giá trị nhỏ hơn 0 hoặc lớn hơn 1 lần lượt bằng 0 hoặc 1.

```
rgb_mean = np.array([0.485, 0.456, 0.406])
rgb_std = np.array([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    img = image.imresize(img, *image_shape)
    img = (img.astype('float32') / 255 - rgb_mean) / rgb_std
    return np.expand_dims(img.transpose(2, 0, 1), axis=0)

def postprocess(img):
    img = img[0].as_in_ctx(rgb_std.ctx)
    return (img.transpose(1, 2, 0) * rgb_std + rgb_mean).clip(0, 1)
```

### 15.12.4 Trích xuất Đặc trưng

Ta sử dụng mô hình VGG-19 tiền huấn luyện trên tập dữ liệu ImagNet để trích xuất các đặc trưng của ảnh [1].

```
pretrained_net = gluon.model_zoo.vision.vgg19(pretrained=True)
```

Để trích xuất các đặc trưng nội dung và phong cách, ta có thể chọn đầu ra của một số tầng nhất định trong mạng VGG. Nhìn chung, đầu ra càng gần với tầng đầu vào, việc trích xuất thông tin chi tiết của ảnh càng dễ hơn. Ngược lại khi đầu ra xa hơn thì dễ trích xuất các thông tin toàn cục hơn. Để ngăn ảnh tổng hợp không giữ quá nhiều chi tiết của ảnh nội dung, ta chọn một tầng mạng VGG gần tầng đầu ra để lấy các đặc trưng nội dung của ảnh đó. Tầng này được gọi là tầng nội dung. Ta cũng chọn đầu ra ở các tầng khác nhau từ mạng VGG để phối hợp với các phong cách cục bộ và toàn cục. Các tầng đó được gọi là các tầng phong cách. Như ta đã đề cập trong Section 9.2, mạng VGG có năm khối tích chập. Trong thử nghiệm này, ta chọn tầng cuối của khối tích chập thứ tư làm tầng nội dung và tầng đầu tiên của mỗi khối làm các tầng phong cách. Ta có thể thu thập chỉ số ở các tầng đó thông qua việc in ra thực thể pretrained\_net.

```
style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

Khi trích xuất đặc trưng, ta chỉ cần sử dụng tất cả các tầng VGG bắt đầu từ tầng đầu vào tới tầng nội dung hoặc tầng phong cách gần tầng đầu ra nhất. Dưới đây, ta sẽ xây dựng một mạng net mới, mạng này chỉ giữ lại các tầng ta cần trong mạng VGG. Sau đó ta sử dụng net để trích xuất đặc trưng.

```
net = nn.Sequential()
for i in range(max(content_layers + style_layers) + 1):
    net.add(pretrained_net.features[i])
```

Với đầu vào  $X$ , nếu ta chỉ đơn thuần thực hiện lượt truyền xuôi  $net(X)$ , ta chỉ có thể thu được đầu ra của tầng cuối cùng. Bởi vì ta cũng cần đầu ra của các tầng trung gian, nên ta phải thực hiện phép tính theo từng tầng và giữ lại đầu ra của tầng nội dung và phong cách.

```

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

```

Tiếp theo, ta định nghĩa hai hàm đó là: Hàm `get_contents` để lấy đặc trưng nội dung trích xuất từ ảnh nội dung, và hàm `get_styles` để lấy đặc trưng phong cách trích xuất từ ảnh phong cách. Do trong lúc huấn luyện, ta không cần thay đổi các tham số của mô hình VGG đã được tiền huấn luyện, nên ta có thể trích xuất đặc trưng nội dung từ ảnh nội dung và đặc trưng phong cách từ ảnh phong cách trước khi bắt đầu huấn luyện. Bởi vì ảnh kết hợp là các tham số mô hình sẽ được cập nhật trong quá trình truyền tải phong cách, ta có thể chỉ cần gọi hàm `extract_features` trong lúc huấn luyện để trích xuất đặc trưng nội dung và phong cách của ảnh kết hợp.

```

def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).copyto(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).copyto(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y

```

### 15.12.5 Định nghĩa Hàm Mất mát

Tiếp theo, ta sẽ bàn về hàm mất mát được sử dụng trong truyền tải phong cách. Hàm mất mát gồm có mất mát nội dung, mất mát phong cách, và mất mát biến thiên toàn phần.

#### Mất mát Nội dung

Tương tự như hàm mất mát được sử dụng trong hồi quy tuyến tính, mất mát nội dung sử dụng hàm bình phương sai số để đo sự khác biệt về đặc trưng nội dung giữa ảnh kết hợp và ảnh nội dung. Hai đầu vào của hàm bình phương sai số bao gồm cả hai đầu ra của tầng nội dung thu được từ hàm `extract_features`.

```

def content_loss(Y_hat, Y):
    return np.square(Y_hat - Y).mean()

```

## Mất mát Phong cách

Tương tự như mất mát nội dung, mất mát phong cách sử dụng hàm bình phương sai số để đo sự khác biệt về đặc trưng phong cách giữa ảnh kết hợp và ảnh phong cách. Để biểu diễn đầu ra phong cách của các tầng phong cách, đầu tiên ta sử dụng hàm `extract_features` để tính toán đầu ra tầng phong cách. Giả sử đầu ra có một mẫu,  $c$  kênh, và có chiều cao và chiều rộng là  $h$  và  $w$ , ta có thể chuyển đổi đầu ra thành ma trận  $\mathbf{X}$  có  $c$  hàng và  $h \cdot w$  cột. Bạn có thể xem ma trận  $\mathbf{X}$  là tổ hợp của  $c$  vector  $\mathbf{x}_1, \dots, \mathbf{x}_c$ , có độ dài là  $hw$ . Ở đây, vector  $\mathbf{x}_i$  biểu diễn đặc trưng phong cách của kênh  $i$ . Trong ma trận Gram  $\mathbf{XX}^\top \in \mathbb{R}^{c \times c}$  của các vector trên, phần tử  $x_{ij}$  nằm trên hàng  $i$  cột  $j$  là tích vô hướng của hai vector  $\mathbf{x}_i$  và  $\mathbf{x}_j$ . Phần tử này biểu thị sự tương quan đặc trưng phong cách của hai kênh  $i$  và  $j$ . Ta sử dụng ma trận Gram này để biểu diễn đầu ra phong cách của các tầng phong cách. Độc giả chú ý rằng khi giá trị  $h \cdot w$  lớn, thì thường dẫn đến ma trận Gram cũng có các giá trị lớn. Hơn nữa, chiều cao và chiều rộng của ma trận Gram đều là số kênh  $c$ . Để đảm bảo rằng mất mát phong cách không bị ảnh hưởng bởi các giá trị kích thước, ta định nghĩa hàm `gram` dưới đây thực hiện phép chia ma trận Gram cho số các phần tử của nó, đó là,  $c \cdot h \cdot w$ .

```
def gram(X):
    num_channels, n = X.shape[1], X.size // X.shape[1]
    X = X.reshape(num_channels, n)
    return np.dot(X, X.T) / (num_channels * n)
```

Thông thường, hai ma trận Gram đầu vào của hàm bình phương sai số cho mất mát phong cách được lấy từ ảnh kết hợp và ảnh phong cách của đầu ra tầng phong cách. Ở đây, ta giả sử ma trận Gram của ảnh phong cách, `gram_Y`, đã được tính toán trước.

```
def style_loss(Y_hat, gram_Y):
    return np.square(gram(Y_hat) - gram_Y).mean()
```

## Mất mát Biến thiên Toàn phần

Đôi khi các ảnh tổng hợp mà ta học có nhiều nhiễu tần số cao, cụ thể là các điểm ảnh sáng hoặc tối. Khử nhiễu biến thiên toàn phần (*total variation denoising*) là một phương pháp phổ biến nhằm giảm nhiễu. Giả định  $x_{i,j}$  biểu diễn giá trị điểm ảnh tại tọa độ  $(i, j)$ , ta có mất mát biến thiên toàn phần:

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}| . \quad (15.12.1)$$

Ta sẽ cố gắng làm cho giá trị của các điểm ảnh lân cận càng giống nhau càng tốt.

```
def tv_loss(Y_hat):
    return 0.5 * (np.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  np.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

## Hàm Mất mát

Hàm mất mát truyền tải phong cách được tính bằng tổng có trọng số của mất mát nội dung, mất mát phong cách, và mất mát biến thiên toàn phần. Thông qua việc điều chỉnh các siêu tham số trọng số này, ta có thể cân bằng giữa phần nội dung giữ lại, phong cách truyền tải và mức giảm nhiễu trong ảnh tổng hợp dựa trên tầm ảnh hưởng tương ứng của chúng.

```
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(styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
```

### 15.12.6 Khai báo và Khởi tạo Ảnh Tổng hợp

Trong truyền tải phong cách, ảnh tổng hợp là biến số duy nhất mà ta cần cập nhật. Do đó, ta có thể định nghĩa một mô hình đơn giản, GeneratedImage, và xem ảnh tổng hợp như một tham số mô hình. Trong mô hình này, lượt truyền xuôi chỉ trả về tham số mô hình.

```
class GeneratedImage(nn.Block):
    def __init__(self, img_shape, **kwargs):
        super(GeneratedImage, self).__init__(**kwargs)
        self.weight = self.params.get('weight', shape=img_shape)

    def forward(self):
        return self.weight.data()
```

Tiếp theo, ta định nghĩa hàm get\_inits. Hàm này khai báo một đối tượng mô hình ảnh tổng hợp và khởi tạo đối tượng theo ảnh X. Ma trận Gram cho các tầng phong cách khác nhau của ảnh phong cách, styles\_Y\_gram, được tính trước khi huấn luyện.

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape)
    gen_img.initialize(init.Constant(X), ctx=device, force_reinit=True)
    trainer = gluon.Trainer(gen_img.collect_params(), 'adam',
                           {'learning_rate': lr})
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

### 15.12.7 Huấn luyện

Trong suốt quá trình huấn luyện mô hình, ta liên tục trích xuất các đặc trưng nội dung và đặc trưng phong cách của ảnh tổng hợp và tính toán hàm mất mát. Nhớ lại thảo luận về cách mà các hàm đồng bộ hoá buộc front-end phải chờ kết quả tính toán trong Section 14.2. Vì ta chỉ gọi hàm đồng bộ hoá numpy sau mỗi 10 epoch, quá trình huấn luyện có thể chiếm dụng lượng lớn bộ nhớ. Do đó, ta gọi đến hàm đồng bộ hoá waitall tại tất cả các epoch.

```
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)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[1, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(1, num_epochs+1):
        with autograd.record():
            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(1)
            npx.waitall()
            if epoch % lr_decay_epoch == 0:
                trainer.set_learning_rate(trainer.learning_rate * 0.1)
            if epoch % 10 == 0:
                animator.axes[1].imshow(postprocess(X).asnumpy())
                animator.add(epoch, [float(sum(contents_l)),
                                    float(sum(styles_l)),
                                    float(tv_l)])
    return X
```

Tiếp theo, ta bắt đầu huấn luyện mô hình. Đầu tiên, ta đặt chiều cao và chiều rộng của ảnh nội dung và ảnh phong cách bằng 150 nhân 225 pixel. Ta sử dụng chính ảnh nội dung để khởi tạo cho ảnh tổng hợp.

```
device, image_shape = d2l.try_gpu(), (225, 150)
net.collect_params().reset_ctx(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, 0.01, 500, 200)
```

Như bạn có thể thấy, ảnh tổng hợp giữ lại phong cảnh và vật thể trong ảnh nội dung, trong khi đưa vào màu sắc của ảnh phong cách. Do ảnh này khá nhỏ, các chi tiết có hơi mờ một chút.

Để thu được ảnh tổng hợp rõ ràng hơn, ta sử dụng ảnh có kích cỡ lớn hơn:  $900 \times 600$ , để huấn luyện mô hình. Ta tăng chiều cao và chiều rộng của ảnh vừa sử dụng lên bốn lần và khởi tạo ảnh tổng hợp lớn hơn.

```
image_shape = (900, 600)
_, content_Y = get_contents(image_shape, device)
_, style_Y = get_styles(image_shape, device)
X = preprocess(postprocess(output) * 255, image_shape)
output = train(X, content_Y, style_Y, device, 0.01, 300, 100)
d2l.plt.imsave('../img/neural-style.jpg', postprocess(output).asnumpy())
```

Như bạn có thể thấy, mỗi epoch cần nhiều thời gian hơn do kích thước ảnh lớn hơn. Có thể thấy trong Fig. 15.12.3, ảnh tổng hợp được sinh ra giữ lại nhiều chi tiết hơn nhờ có kích thước lớn hơn. Ảnh tổng hợp không những có các khối màu giống như ảnh phong cách, mà các khối này còn có hoa văn phản phất nét vẽ bút lông.



Fig. 15.12.3: Ảnh tổng hợp kích thước  $900 \times 600$

### 15.12.8 Tóm tắt

- Các hàm mất mát được sử dụng trong truyền tải phong cách nhìn chung bao gồm ba phần:
  1. Mất mát nội dung được sử dụng để biến đổi ảnh tổng hợp gần giống ảnh nội dung dựa trên đặc trưng nội dung.
  2. Mất mát phong cách được sử dụng để biến đổi ảnh tổng hợp gần giống ảnh phong cách dựa trên đặc trưng phong cách.
  3. Mất mát biến thiên toàn phần giúp giảm nhiễu trong ảnh tổng hợp.
- Ta có thể sử dụng CNN đã qua tiền huấn luyện để trích xuất đặc trưng ảnh và cực tiểu hóa hàm mất mát, nhờ đó liên tục cập nhật ảnh tổng hợp.
- Ta sử dụng ma trận Gram để biểu diễn phong cách đầu ra của các tầng phong cách.

### 15.12.9 Bài tập

1. Đầu ra thay đổi thế nào khi bạn chọn tầng nội dung và phong cách khác?
2. Điều chỉnh các siêu tham số trọng số của hàm mất mát. Đầu ra khi đó liệu có giữ lại nhiều nội dung hơn hay có ít nhiễu hơn?
3. Sử dụng ảnh nội dung và ảnh phong cách khác. Bạn hãy thử tạo ra các ảnh tổng hợp khác thú vị hơn.

### 15.12.10 Thảo luận

- Tiếng Anh - MXNet<sup>323</sup>
- Tiếng Việt<sup>324</sup>

### 15.12.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long
- Nguyễn Văn Quang
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh

## 15.13 Phân loại ảnh (CIFAR-10) trên Kaggle

Cho đến lúc này, ta đang sử dụng gói data của Gluon để lấy trực tiếp các tập dữ liệu dưới định dạng tensor. Tuy nhiên, trong thực tế thì các tập dữ liệu ảnh thường ở định dạng tập tin. Trong phần này, ta sẽ sử dụng các tập tin ảnh gốc và từng bước tổ chức, đọc và chuyển đổi các tập tin này sang định dạng tensor.

Chúng ta thử nghiệm trên tập dữ liệu CIFAR-10 trong Section 15.1. Đây là một tập dữ liệu quan trọng trong lĩnh vực thị giác máy tính. Bây giờ, ta sẽ áp dụng kiến thức đã học ở các phần trước để tham gia vào cuộc thi phân loại ảnh CIFAR-10 trên Kaggle. Địa chỉ trang web của cuộc thi tại

<https://www.kaggle.com/c/cifar-10>

<sup>323</sup> <https://discuss.d2l.ai/t/378>

<sup>324</sup> <https://forum.machinelearningcoban.com/c/d2l>

Hình Fig. 15.13.1 cho biết thông tin trên trang web của cuộc thi. Để nộp kết quả, vui lòng đăng ký một tài khoản Kaggle trước.

The screenshot shows the 'CIFAR-10 - Object Recognition in Images' competition page. At the top, there's a grid of small images representing the dataset categories. Below the grid, the title 'CIFAR-10 - Object Recognition in Images' is displayed, followed by the subtitle 'Identify the subject of 60,000 labeled images'. It also shows '231 teams · 4 years ago'. Below this, a navigation bar includes 'Overview' (which is underlined), 'Data', 'Discussion', 'Leaderboard', and 'Rules'. Under the 'Overview' section, there are two tabs: 'Description' and 'Evaluation'. The 'Description' tab contains text about the dataset being an established computer-vision dataset used for object recognition, being a subset of the 80 million tiny images dataset, and consisting of 60,000 32x32 color images containing one of 10 object classes, with 6000 images per class. It was collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton.

Fig. 15.13.1: Thông tin trang web cuộc thi phân loại ảnh CIFAR-10. Tập dữ liệu cho cuộc thi có thể truy cập bằng cách chọn vào thẻ “Data”.

Trước tiên, nạp các gói và mô-đun cần cho cuộc thi này.

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os
import pandas as pd
import shutil
import time

npx.set_np()
```

### 15.13.1 Tải và Tổ chức tập dữ liệu

Dữ liệu cuộc thi được chia thành tập huấn luyện và tập kiểm tra. Tập huấn luyện chứa 50,000 ảnh. Tập kiểm tra chứa 300,000 ảnh, trong đó 10,000 ảnh được sử dụng để tính điểm, 290,000 ảnh còn lại dùng để ngăn ngừa việc gán nhãn thủ công vào tập kiểm tra rồi nộp kết quả đã gán nhãn. Định dạng ảnh trong cả hai tập dữ liệu là PNG, với chiều cao và chiều rộng là 32 pixel với ba kênh màu (RGB). Các ảnh được phân thành 10 hạng mục: máy bay, xe hơi, chim, mèo, nai, chó, ếch, ngựa, thuyền và xe tải. Góc trên bên trái của Fig. 15.13.1 hiển thị một số ảnh máy bay, xe hơi và chim trong tập dữ liệu.

### 15.13.2 Tải tập Dữ liệu

Sau khi đăng nhập vào Kaggle, ta có thể chọn thẻ “Data” trên trang của cuộc thi phân loại ảnh CIFAR-10 như trong Fig. 15.13.1 và tải tập dữ liệu này bằng cách nhấp chuột vào nút “Download All”. Sau khi giải nén tập tin đã tải về vào thư mục .. /data, và giải nén train.7z và test.7z trong tập tin này, bạn sẽ tìm thấy toàn bộ tập dữ liệu ở đường dẫn thư mục 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

Các thư mục train và test chứa các ảnh cho việc huấn luyện và kiểm tra tương ứng, tập tin trainLabels.csv chứa các nhãn dùng cho ảnh huấn luyện và tập tin sample\_submission.csv là một tệp nộp ví dụ.

Để việc bắt đầu đơn giản hơn, chúng tôi cung cấp một mẫu thu nhỏ của tập dữ liệu này: chứa 1000 ảnh huấn luyện đầu tiên và 5 ảnh kiểm tra ngẫu nhiên. Để sử dụng toàn bộ tập dữ liệu của cuộc thi Kaggle, bạn cần đặt biến demo 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/'
```

### 15.13.3 Tổ chức tập Dữ liệu

Ta cần tổ chức tập dữ liệu để thuận tiện cho việc huấn luyện và kiểm tra. Hãy bắt đầu bằng cách đọc các nhãn từ tập tin csv. Hàm sau đây trả về một từ điển thực hiện ánh xạ tên tập tin (không bao gồm phần mở rộng) sang nhãn của nó.

```
#@save
def read_csv_labels(fname):
    """Read fname to return a name 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())))
```

Kế tiếp, ta định nghĩa hàm `reorg_train_valid` để phân đoạn tập kiểm định từ tập huấn luyện gốc. Tham số `valid_ratio` trong hàm này là tỉ số của số mẫu trong tập kiểm định đối với số mẫu trong tập huấn luyện gốc. Cụ thể, gọi  $n$  là số ảnh của lớp có ít mẫu nhất, và  $r$  là tỉ số thì ta sẽ dùng  $\max([nr], 1)$  ảnh trong mỗi lớp làm tập kiểm định. Ta hãy chọn `valid_ratio=0.1` làm ví dụ. Vì tập ảnh huấn luyện gốc có 50,000 ảnh, do đó ta sẽ có 45,000 ảnh dùng để huấn luyện và lưu ở thư mục “`train_valid_test/train`” khi tinh chỉnh các siêu tham số, trong khi 5,000 ảnh còn lại sử dụng làm tập kiểm định sẽ được lưu ở thư mục “`train_valid_test/valid`”. Sau khi tổ chức dữ liệu, ảnh của một lớp sẽ được đặt ở cùng thư mục để đọc chúng sau này.

```
#@save
def copyfile(filename, target_dir):
    """Copy a file into a target directory."""
    d2l.mkdir_if_not_exist(target_dir)
    shutil.copy(filename, target_dir)

#@save
def reorg_train_valid(data_dir, labels, valid_ratio):
    # The number of examples of the class with the least 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)
        # Copy to train_valid_test/train_valid with a subfolder per class
        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:
            # Copy to train_valid_test/valid
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'valid', label))
            label_count[label] = label_count.get(label, 0) + 1
        else:
            # Copy to train_valid_test/train
            copyfile(fname, os.path.join(data_dir, 'train_valid_test',
                                         'train', label))
    return n_valid_per_label
```

Hàm `reorg_test` dưới đây được dùng để tổ chức tập kiểm tra để thuận tiện cho việc đọc tệp trong quá trình dự đoán.

```
#@save
def reorg_test(data_dir):
    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'))
```

Sau cùng, ta sử dụng một hàm để gọi các hàm `read_csv_labels`, `reorg_train_valid`, và `reorg_test` đã được định nghĩa trước đó.

```
def reorg_cifar10_data(data_dir, valid_ratio):
```

(continues on next page)

```
labels = read_csv_labels(os.path.join(data_dir, 'trainLabels.csv'))
reorg_train_valid(data_dir, labels, valid_ratio)
reorg_test(data_dir)
```

Chúng ta chỉ thiết lập kích thước batch là 4 đối với tập dữ liệu chạy thử. Trong suốt quá trình huấn luyện và kiểm thử thật sự, nên sử dụng tập huấn luyện đầy đủ của cuộc thi Kaggle và batch\_size nên được thiết lập một giá trị số nguyên lớn hơn như là 128. Ta sử dụng 10% mẫu huấn luyện làm tập kiểm định để tinh chỉnh các siêu tham số.

```
batch_size = 4 if demo else 128
valid_ratio = 0.1
reorg_cifar10_data(data_dir, valid_ratio)
```

## Tăng cường ảnh

Để tránh hiện tượng quá khớp, ta sẽ áp dụng tăng cường ảnh. Ví dụ, ta có thể lật ngẫu nhiên các ảnh bằng cách thêm transforms.RandomFlipLeftRight(). Ta cũng có thể thực hiện chuẩn hóa trên ba kênh màu RGB của ảnh bằng cách sử dụng transforms.Normalize(). Dưới đây, chúng tôi liệt kê một số thao tác tăng cường ảnh để bạn có thể lựa chọn sử dụng hoặc chỉnh sửa tùy theo nhu cầu.

```
transform_train = gluon.data.vision.transforms.Compose([
    # Magnify the image to a square of 40 pixels in both height and width
    gluon.data.vision.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 shrink it to a square of 32 pixels in both height and
    # width
    gluon.data.vision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                                   ratio=(1.0, 1.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    gluon.data.vision.transforms.ToTensor(),
    # Normalize each channel of the image
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                          [0.2023, 0.1994, 0.2010])])
```

Để đảm bảo tính chắc chắn của đầu ra trong quá trình kiểm tra, ta chỉ thực hiện chuẩn hóa trên ảnh.

```
transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                         [0.2023, 0.1994, 0.2010])])
```

## Đọc tập Dữ liệu

Tiếp theo, ta tạo đối tượng `ImageFolderDataset` để đọc tập dữ liệu đã được tổ chức ở trên bao gồm các tệp ảnh gốc, trong đó mỗi ví dụ gồm có ảnh và nhãn.

```
train_ds, valid_ds, train_valid_ds, test_ds = [
    gluon.data.vision.ImageFolderDataset(
        os.path.join(data_dir, 'train_valid_test', folder))
    for folder in ['train', 'valid', 'train_valid', 'test']]
```

Trong `DataLoader` ta chỉ rõ thao tác tăng cường ảnh đã xác định ở trên. Trong suốt quá trình huấn luyện, ta chỉ sử dụng tập kiểm định để đánh giá mô hình, do đó cần đảm bảo tính chắc chắn của đầu ra. Trong quá trình dự đoán, ta sẽ huấn luyện mô hình trên tập huấn luyện và tập kiểm định gộp lại để tận dụng tất cả dữ liệu có gán nhãn.

```
train_iter, train_valid_iter = [gluon.data.DataLoader(
    dataset.transform_first(transform_train), batch_size, shuffle=True,
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]

valid_iter = gluon.data.DataLoader(
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='discard')

test_iter = gluon.data.DataLoader(
    test_ds.transform_first(transform_test), batch_size, shuffle=False,
    last_batch='keep')
```

## Định nghĩa Mô hình

Ở phần này, ta xây dựng các khối phần dư dựa trên lớp `HybridBlock`, khối này có đôi chút khác biệt so với cách lập trình trong [Section 9.6](#) nhằm cải thiện hiệu suất thực thi.

```
class Residual(nn.HybridBlock):
    def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
        super(Residual, self).__init__(**kwargs)
        self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                            strides=strides)
        self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
        if use_1x1conv:
            self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                strides=strides)
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def hybrid_forward(self, F, X):
        Y = F.npx.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return F.npx.relu(Y + X)
```

Tiếp theo, ta định nghĩa mô hình ResNet-18.

```

def resnet18(num_classes):
    net = nn.HybridSequential()
    net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
           nn.BatchNorm(), nn.Activation('relu'))

    def resnet_block(num_channels, num_residuals, first_block=False):
        blk = nn.HybridSequential()
        for i in range(num_residuals):
            if i == 0 and not first_block:
                blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
            else:
                blk.add(Residual(num_channels))
        return blk

    net.add(resnet_block(64, 2, first_block=True),
            resnet_block(128, 2),
            resnet_block(256, 2),
            resnet_block(512, 2))
    net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
    return net

```

Thử thách phân loại ảnh CIFAR-10 bao gồm 10 hạng mục. Ta sẽ thực hiện khởi tạo ngẫu nhiên Xavier trên mô hình trước khi bắt đầu huấn luyện.

```

def get_net(devices):
    num_classes = 10
    net = resnet18(num_classes)
    net.initialize(ctx=devices, init=init.Xavier())
    return net

loss = gluon.loss.SoftmaxCrossEntropyLoss()

```

## Định nghĩa Hàm Huấn luyện

Ta tiến hành lựa chọn mô hình và điều chỉnh các siêu tham số tùy theo kết quả của mô hình trên tập kiểm định. Tiếp theo, ta định nghĩa hàm huấn luyện mô hình train. Ta ghi lại thời gian huấn luyện mỗi epoch nhằm giúp ta có thể so sánh thời gian mà các mô hình khác nhau yêu cầu.

```

def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    trainer = gluon.Trainer(net.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'train acc', 'valid acc'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(3)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            l, acc = d2l.train_batch_ch13(
                net, features, labels.astype('float32'), loss, trainer,

```

(continues on next page)

```

        devices, d2l.split_batch)
metric.add(1, acc, labels.shape[0])
timer.stop()
if (i + 1) % (num_batches // 5) == 0:
    animator.add(epoch + i / num_batches,
                  (metric[0] / metric[2], metric[1] / metric[2],
                   None))
if valid_iter is not None:
    valid_acc = d2l.evaluate_accuracy_gpus(net, valid_iter, d2l.split_batch)
    animator.add(epoch + 1, (None, None, valid_acc))
if valid_iter is not None:
    print(f'loss {metric[0] / metric[2]:.3f}, '
          f'train acc {metric[1] / metric[2]:.3f}, '
          f'valid acc {valid_acc:.3f}')
else:
    print(f'loss {metric[0] / metric[2]:.3f}, '
          f'train acc {metric[1] / metric[2]:.3f}')
print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
      f'on {str(devices)})')

```

## Huấn luyện và Kiểm định Mô hình

Bây giờ ta có thể huấn luyện và kiểm định mô hình. Các siêu tham số sau có thể được điều chỉnh: num\_epochs, lr\_period và lr\_decay. Ta có thể tăng số epoch. Để đơn giản, ở đây ta chỉ huấn luyện 5 epoch. Do lr\_period và lr\_decay được đặt lần lượt bằng 50 và 0.1, tốc độ học của thuật toán tối ưu sẽ giảm đi 10 lần sau mỗi 50 epoch.

```

devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.1, 5e-4
lr_period, lr_decay, net = 50, 0.1, get_net(devices)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

```

## Phân loại Tập Kiểm tra và Nộp Kết quả trên Kaggle

Sau khi thu được thiết kế mô hình và các siêu tham số vừa ý, ta sử dụng toàn bộ tập huấn luyện (bao gồm tập kiểm định) để huấn luyện lại mô hình và tiến hành phân loại tập kiểm tra.

```

net, preds = get_net(devices), []
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

for X, _ in test_iter:
    y_hat = net(X.as_in_ctx(devices[0]))
    preds.extend(y_hat.argmax(axis=1).astype(int).asnumpy())
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.synsets[x])
df.to_csv('submission.csv', index=False)

```

Sau khi chạy đoạn mã trên, ta sẽ thu được tệp “submission.csv”. Tệp này có định dạng phù hợp với yêu cầu của cuộc thi trên Kaggle. Cách thức nộp kết quả giống với cách thức trong [Section 6.10](#).

## Tóm tắt

- Ta có thể tạo một đối tượng `ImageFolderDataset` để đọc tập dữ liệu gồm có các tệp ảnh gốc.
- Ta có thể sử dụng mạng nơ-ron tích chập, tăng cường ảnh, và lập trình hybrid để tham gia vào cuộc thi phân loại ảnh.

## Bài tập

1. Sử dụng tập dữ liệu CIFAR-10 đầy đủ cho cuộc thi trên Kaggle. Thay đổi `batch_size` và `num_epochs` lần lượt bằng 128 và 100. Quan sát độ chính xác và xem bạn có thể đạt thứ hạng bao nhiêu trong cuộc thi này.
2. Bạn có thể đạt độ chính xác bằng bao nhiêu nếu không sử dụng tăng cường ảnh?
3. Quét mã QR để truy cập các thảo luận liên quan và trao đổi ý tưởng về các phương pháp được sử dụng và kết quả thu được với mọi người. Bạn có khám phá ra kỹ thuật nào khác tốt hơn không?

## Thảo luận

- Tiếng Anh - MXNet<sup>325</sup>
- Tiếng Việt<sup>326</sup>

## Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Nguyễn Mai Hoàng Long
- Phạm Hồng Vinh
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Nguyễn Văn Cường

<sup>325</sup> <https://discuss.d2l.ai/t/379>

<sup>326</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 15.14 Nhận diện Giống Chó (ImageNet Dogs) trên Kaggle

Trong phần này, ta sẽ giải quyết thử thách nhận diện giống chó trong một cuộc thi trên Kaggle. Cuộc thi có địa chỉ tại

<https://www.kaggle.com/c/dog-breed-identification>

Trong cuộc thi này, ta cần nhận diện 120 giống chó khác nhau. Tập dữ liệu trong cuộc thi này thực chất là một tập con của tập dữ liệu ImageNet nổi tiếng. Khác với ảnh trong tập dữ liệu CIFAR-10 được sử dụng trong phần trước, các ảnh trong tập dữ liệu ImageNet có chiều dài và chiều rộng lớn hơn, đồng thời kích thước của chúng không nhất quán.

Fig. 15.14.1 mô tả thông tin trên trang web của cuộc thi. Để có thể nộp kết quả, trước tiên vui lòng đăng ký tài khoản trên Kaggle.

A screenshot of the Kaggle Dog Breed Identification competition page. At the top, there's a large image of a dog looking up. Below it, the title "Dog Breed Identification" and subtitle "Determine the breed of a dog in an image". It shows "Kaggle · 1,286 teams · 4 months ago". The navigation bar includes "Overview" (which is underlined), "Data", "Kernels", "Discussion", "Leaderboard", and "Rules". The "Overview" section has tabs for "Description" and "Evaluation". The "Description" tab contains text about the competition and a grid of 10 small dog images. The "Evaluation" tab contains text about the dataset and a grid of 10 small dog images.

Fig. 15.14.1: Trang web cuộc thi nhận diện giống chó. Tập dữ liệu cho cuộc thi này có thể được truy cập bằng cách nhấn vào thẻ “Data”.

Đầu tiên, ta nhập vào các gói thư viện hoặc các mô-đun cần cho cuộc thi.

```
import collections
from d2l import mxnet as d2l
import math
from mxnet import autograd, gluon, init, npx
from mxnet.gluon import nn
import os
import time

npx.set_np()
```

### 15.14.1 Tải xuống và Tổ chức Tập dữ liệu

Dữ liệu cuộc thi được chia thành tập huấn luyện và tập kiểm tra. Tập huấn luyện bao gồm 10,222 ảnh và tập kiểm tra bao gồm 10,357 ảnh. Tất cả các ảnh trong hai tập đều có định dạng JPEG. Các ảnh này gồm có ba kênh (màu) RGB và có chiều cao và chiều rộng khác nhau. Có tất cả 120 giống chó trong tập huấn luyện, gồm có Chó tha mồi (*Labrador*), Chó săn vịt (*Poodle*), Chó Dachshund, Samoyed, Huskie, Chihuahua, và Chó sục Yorkshire (*Yorkshire Terriers*).

#### Tải tập dữ liệu

Sau khi đăng nhập vào Kaggle, ta có thể chọn thẻ “Data” trong trang web cuộc thi nhận diện giống chó như mô tả trong Fig. 15.14.1 và tải tập dữ liệu về bằng cách nhấn vào nút “Download All”. Sau khi giải nén tệp đã tải về trong thư mục `../data`, bạn có thể tìm thấy toàn bộ tập dữ liệu theo các đường dẫn sau:

- `../data/dog-breed-identification/labels.csv`
- `../data/dog-breed-identification/sample_submission.csv`
- `../data/dog-breed-identification/train`
- `../data/dog-breed-identification/test`

Có thể bạn đã nhận ra rằng cấu trúc trên khá giống với cấu trúc thư mục của cuộc thi CIFAR-10 trong Section 15.13, trong đó thư mục `train/` và `test/` lần lượt chứa ảnh chó để huấn luyện và kiểm tra, và `labels.csv` chứa nhãn cho các ảnh huấn luyện.

Tương tự, để đơn giản, chúng tôi cung cấp một tập mẫu nhỏ của tập dữ liệu kể trên, “`train_valid_test_tiny.zip`”. Nếu bạn sử dụng tập dữ liệu đầy đủ cho cuộc thi Kaggle, bạn cần thay đổi biến `demo` phía dưới thành `False`.

```
#@save
d2l.DATA_HUB['dog_tiny'] = (d2l.DATA_URL + 'kaggle_dog_tiny.zip',
                            '75d1ec6b9b2616d2760f211f72a83f73f3b83763')

# 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')
```

#### Tổ chức Tập dữ liệu

Ta có thể tổ chức tập dữ liệu tương tự như cách ta đã làm trong Section 15.13, tức là tách riêng một tập kiểm định từ tập huấn luyện, sau đó đưa các ảnh vào từng thư mục con theo nhãn của chúng.

Hàm `reorg_dog_data` dưới đây được sử dụng để đọc nhãn của dữ liệu huấn luyện, tách riêng tập kiểm định và tổ chức tập huấn luyện.

```

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 = 4 if demo else 128
valid_ratio = 0.1
reorg_dog_data(data_dir, valid_ratio)

```

### 15.14.2 Tăng cường Ảnh

Trong phần này, kích thước ảnh lớn hơn phần trước. Dưới đây là một số kỹ thuật tăng cường ảnh có thể sẽ hữu dụng.

```

transform_train = gluon.data.vision.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 image with a height and width of 224
    # pixels each
    gluon.data.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                                   ratio=(3.0/4.0, 4.0/3.0)),
    gluon.data.vision.transforms.RandomFlipLeftRight(),
    # Randomly change the brightness, contrast, and saturation
    gluon.data.vision.transforms.RandomColorJitter(brightness=0.4,
                                                   contrast=0.4,
                                                   saturation=0.4),
    # Add random noise
    gluon.data.vision.transforms.RandomLighting(0.1),
    gluon.data.vision.transforms.ToTensor(),
    # Standardize each channel of the image
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                          [0.229, 0.224, 0.225]))]

```

Trong quá trình kiểm tra, ta chỉ sử dụng một số bước tiền xử lý ảnh nhất định.

```

transform_test = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(256),
    # Crop a square of 224 by 224 from the center of the image
    gluon.data.vision.transforms.CenterCrop(224),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                          [0.229, 0.224, 0.225]))]

```

### 15.14.3 Đọc Dữ liệu

Như trong phần trước, ta có thể tạo thực thể ImageFolderDataset để đọc dữ liệu chứa các tệp ảnh gốc.

```
train_ds, valid_ds, train_valid_ds, test_ds = [  
    gluon.data.vision.ImageFolderDataset(  
        os.path.join(data_dir, 'train_valid_test', folder))  
    for folder in ('train', 'valid', 'train_valid', 'test')]
```

Ở đây, ta tạo các thực thể DataLoader giống như trong Section 15.13.

```
train_iter, train_valid_iter = [gluon.data.DataLoader(  
    dataset.transform_first(transform_train), batch_size, shuffle=True,  
    last_batch='discard') for dataset in (train_ds, train_valid_ds)]  
  
valid_iter = gluon.data.DataLoader(  
    valid_ds.transform_first(transform_test), batch_size, shuffle=False,  
    last_batch='discard')  
  
test_iter = gluon.data.DataLoader(  
    test_ds.transform_first(transform_test), batch_size, shuffle=False,  
    last_batch='keep')
```

### 15.14.4 Định nghĩa Mô hình

Dữ liệu cho cuộc thi này là một phần của tập dữ liệu ImageNet. Do đó, ta có thể sử dụng cách tiếp cận được thảo luận trong Section 15.2 để lựa chọn mô hình đã được tiền huấn luyện trên toàn bộ dữ liệu ImageNet và sử dụng nó để trích xuất đặc trưng ảnh làm đầu vào cho một mạng tùy biến cỡ nhỏ. Gluon cung cấp một số mô hình đã được tiền huấn luyện. Ở đây, ta sử dụng mô hình ResNet-34 đã được tiền huấn luyện. Do dữ liệu của cuộc thi là tập con của tập dữ liệu tiền huấn luyện, ta đơn thuần sử dụng lại đầu vào của tầng đầu ra mô hình đã được tiền huấn luyện làm đặc trưng được trích xuất. Sau đó, ta có thể thay thế tầng đầu ra gốc bằng một mạng đầu ra tùy biến cỡ nhỏ để huấn luyện bao gồm hai tầng kết nối đầy đủ. Khác với thí nghiệm trong Section 15.2, ở đây ta không huấn luyện lại mô hình trích xuất đặc trưng đã được tiền huấn luyện. Điều này giúp giảm thời gian huấn luyện và bộ nhớ cần thiết để lưu trữ gradient của tham số mô hình.

Độc giả cần lưu ý, trong quá trình tăng cường ảnh, ta sử dụng giá trị trung bình và độ lệch chuẩn của ba kênh RGB lấy từ toàn bộ dữ liệu ImageNet để chuẩn hóa. Điều này giúp dữ liệu nhất quán với việc chuẩn hóa của mô hình tiền huấn luyện.

```
def get_net(devices):  
    finetune_net = gluon.model_zoo.vision.resnet34_v2(pretrained=True)  
    # Define a new output network  
    finetune_net.output_new = nn.HybridSequential(prefix='')  
    finetune_net.output_new.add(nn.Dense(256, activation='relu'))  
    # There are 120 output categories  
    finetune_net.output_new.add(nn.Dense(120))  
    # Initialize the output network  
    finetune_net.output_new.initialize(init.Xavier(), ctx=devices)  
    # Distribute the model parameters to the CPUs or GPUs used for computation  
    finetune_net.collect_params().reset_ctx(devices)  
    return finetune_net
```

Khi tính toán mất mát, đầu tiên ta sử dụng biến thành viên `features` để lấy đầu vào của tầng đầu ra trong mô hình được tiền huấn luyện làm đặc trưng trích xuất. Sau đó, ta sử dụng đặc trưng này làm đầu vào cho mạng đầu ra tùy biến cỡ nhỏ và tính toán đầu ra.

```
loss = gluon.loss.SoftmaxCrossEntropyLoss()

def evaluate_loss(data_iter, net, devices):
    l_sum, n = 0.0, 0
    for features, labels in data_iter:
        X_shards, y_shards = d2l.split_batch(features, labels, devices)
        output_features = [net.features(X_shard) for X_shard in X_shards]
        outputs = [net.output_new(feature) for feature in output_features]
        ls = [loss(output, y_shard).sum() for output, y_shard
              in zip(outputs, y_shards)]
        l_sum += sum([float(l.sum()) for l in ls])
        n += labels.size
    return l_sum / n
```

### 15.14.5 Định nghĩa Hàm Huấn luyện

Ta sẽ lựa chọn mô hình và điều chỉnh siêu tham số dựa trên chất lượng mô hình trên tập kiểm định. Hàm huấn luyện mô hình train chỉ huấn luyện mạng đầu ra tùy biến cỡ nhỏ.

```
def train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
          lr_decay):
    # Only train the small custom output network
    trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
    num_batches, timer = len(train_iter), d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[0, num_epochs],
                            legend=['train loss', 'valid loss'])
    for epoch in range(num_epochs):
        metric = d2l.Accumulator(2)
        if epoch > 0 and epoch % lr_period == 0:
            trainer.set_learning_rate(trainer.learning_rate * lr_decay)
        for i, (features, labels) in enumerate(train_iter):
            timer.start()
            X_shards, y_shards = d2l.split_batch(features, labels, devices)
            output_features = [net.features(X_shard) for X_shard in X_shards]
            with autograd.record():
                outputs = [net.output_new(feature) for feature in output_features]
                ls = [loss(output, y_shard).sum() for output, y_shard
                      in zip(outputs, y_shards)]
                for l in ls:
                    l.backward()
            trainer.step(batch_size)
            metric.add(sum([float(l.sum()) for l in ls]), labels.shape[0])
            timer.stop()
            if (i + 1) % (num_batches // 5) == 0:
                animator.add(epoch + i / num_batches,
                             (metric[0] / metric[1], None))
        if valid_iter is not None:
            valid_loss = evaluate_loss(valid_iter, net, devices)
            animator.add(epoch + 1, (None, valid_loss))
```

(continues on next page)

```

if valid_iter is not None:
    print(f'train loss {metric[0] / metric[1]:.3f}, '
          f'valid loss {valid_loss:.3f}')
else:
    print(f'train loss {metric[0] / metric[1]:.3f}')
print(f'{metric[1] * num_epochs / timer.sum():.1f} examples/sec '
      f'on {str(devices)})'

```

### 15.14.6 Huấn luyện và Kiểm định Mô hình

Bây giờ, ta có thể huấn luyện và kiểm định mô hình. Các siêu tham số dưới đây có thể được điều chỉnh: num\_epochs, lr\_period và lr\_decay. Ví dụ, ta có thể tăng số lượng epoch. Do lr\_period và lr\_decay được thiết lập bằng 10 và 0.1, tốc độ học của thuật toán tối ưu sẽ được nhân với 0.1 sau mỗi 10 epoch.

```

devices, num_epochs, lr, wd = d2l.try_all_gpus(), 5, 0.01, 1e-4
lr_period, lr_decay, net = 10, 0.1, get_net(devices)
net.hybridize()
train(net, train_iter, valid_iter, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

```

### 15.14.7 Dự đoán trên tập Kiểm tra và Nộp Kết quả lên Kaggle

Sau khi thu được một thiết kế mô hình và các siêu tham số vừa ý, ta sử dụng tất cả dữ liệu huấn luyện (bao gồm dữ liệu kiểm định) để huấn luyện lại mô hình, sau đó thực hiện dự đoán trên tập kiểm tra. Chú ý rằng các dự đoán được lấy từ mạng đầu ra mà ta đã huấn luyện.

```

net = get_net(devices)
net.hybridize()
train(net, train_valid_iter, None, num_epochs, lr, wd, devices, lr_period,
      lr_decay)

preds = []
for data, label in test_iter:
    output_features = net.features(data.as_in_ctx(devices[0]))
    output = npx.softmax(net.output_new(output_features))
    preds.extend(output.asnumpy())
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.synsets) + '\n')
    for i, output in zip(ids, preds):
        f.write(i.split('.')[0] + ',' + ','.join(
            [str(num) for num in output]) + '\n')

```

Chạy đoạn mã trên sẽ sinh tệp “submission.csv”. Định dạng của tệp này nhất quán với yêu cầu của cuộc thi Kaggle. Cách thức nộp kết quả tương tự như trong Section 6.10.

#### 15.14.8 Tóm tắt

- Ta có thể sử dụng mô hình đã được tiền huấn luyện trên tập dữ liệu ImageNet để trích xuất đặc trưng và chỉ huấn luyện trên mạng đầu ra tùy biến cỡ nhỏ.
- Điều này cho phép ta có thể thực hiện dự đoán trên tập con của tập dữ liệu ImageNet với chi phí bộ nhớ và tính toán thấp hơn.

#### 15.14.9 Bài tập

1. Khi sử dụng toàn bộ dữ liệu Kaggle, bạn sẽ thu được kết quả như thế nào khi tăng batch\_size (kích thước batch) và num\_epochs (số lượng epoch)?
2. Bạn có đạt được kết quả tốt hơn nếu sử dụng mô hình đã được tiền huấn luyện sâu hơn không?
3. Quét mã QR để tham gia thảo luận và trao đổi ý tưởng về các phương pháp đã được sử dụng và kết quả thu được từ cộng đồng Kaggle. Bạn có thể nghĩ ra một ý tưởng hay kỹ thuật tốt hơn không?

#### 15.14.10 Thảo luận

- Tiếng Anh - MXNet<sup>327</sup>
- Tiếng Việt<sup>328</sup>

#### 15.14.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Nguyễn Văn Cường

### 15.15 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

---

<sup>327</sup> <https://discuss.d2l.ai/t/380>

<sup>328</sup> <https://forum.machinelearningcoban.com/c/d2l>

# 16 | Xử lý Ngôn ngữ Tự nhiên: Tiền Huấn luyện

Con người luôn có nhu cầu được giao tiếp. Chính từ nhu cầu cơ bản này mà một lượng lớn dữ liệu văn bản được tạo ra mỗi ngày. Với lượng dữ liệu văn bản đa dạng từ mạng xã hội, ứng dụng trò chuyện, email, đánh giá sản phẩm, tài liệu nghiên cứu và sách báo, việc giúp máy tính hiểu được những dữ liệu này trở nên quan trọng, nhằm đưa ra cách thức hỗ trợ hoặc quyết định dựa trên ngôn ngữ của con người.

Xử lý ngôn ngữ tự nhiên nghiên cứu sự tương tác bằng ngôn ngữ tự nhiên giữa máy tính và con người. Trong thực tế, việc 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) rất phổ biến, chẳng hạn như các mô hình ngôn ngữ trong Section 10.3 hay các mô hình dịch máy trong Section 11.5.

Để hiểu dữ liệu văn bản, ta có thể bắt đầu với cách biểu diễn loại dữ liệu này, chẳng hạn xem mỗi từ hay từ con như một token riêng lẻ. Trong chương này, biểu diễn của mỗi token có thể được tiền huấn luyện trên một kho ngữ liệu lớn, sử dụng các mô hình word2vec, GloVe, hay embedding cho từ con. Sau khi tiền huấn luyện, biểu diễn của mỗi token có thể là một vector. Tuy nhiên, biểu diễn này vẫn không đổi dù ngữ cảnh xung quanh bất kể là gì. Ví dụ, biểu diễn vector của từ “bank” là giống nhau trong câu “go to the bank to deposit some money” (ra ngân hàng để gửi tiền) và “go to the bank to sit down” (ra bờ hồ ngồi hóng mát). Do đó, nhiều mô hình tiền huấn luyện gần đây điều chỉnh biểu diễn của cùng một token với các ngữ cảnh khác nhau. Trong số đó có BERT, một mô hình sâu hơn rất nhiều dựa trên bộ mã hóa Transformer. Trong chương này, ta sẽ tập trung vào cách tiền huấn luyện các biểu diễn như vậy cho văn bản, như được mô tả trong Fig. 16.1.

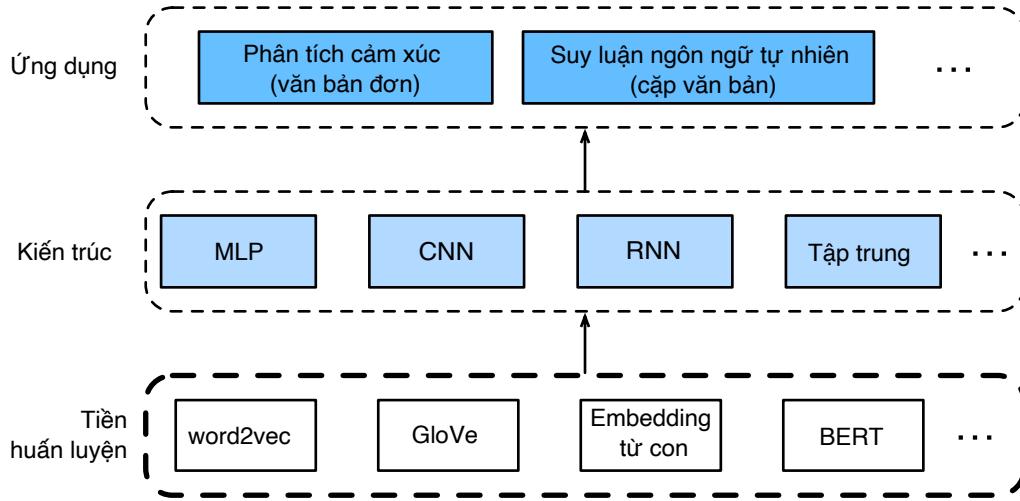


Fig. 16.1: Các biểu diễn văn bản được tiền huấn luyện có thể được truyền vào các kiến trúc học sâu khác nhau cho các ứng dụng xử lý ngôn ngữ tự nhiên xuôi dòng khác nhau. Chương này tập trung vào cách tiền huấn luyện biểu diễn văn bản ngược dòng (*upstream*).

Như mô tả trong Fig. 16.1, các biểu diễn văn bản được tiền huấn luyện có thể được truyền vào những kiến trúc học sâu cho các ứng dụng xử lý ngôn ngữ tự nhiên xuôi dòng khác nhau. Chúng tôi sẽ trình bày các phần này trong Section 17.

## 16.1 Embedding Từ (word2vec)

Ngôn ngữ tự nhiên là một hệ thống phức tạp mà con người sử dụng để diễn đạt ngữ nghĩa. Trong hệ thống này, từ là đơn vị cơ bản của ngữ nghĩa. Như tên gọi của nó, một vector từ (*word vector*) là một vector được sử dụng để biểu diễn một từ. Vector từ cũng có thể được xem là vector đặc trưng của một từ. Kỹ thuật ánh xạ từ ngữ sang vector số thực còn được gọi là kỹ thuật embedding từ (*word embedding*). Trong vài năm gần đây, embedding từ dần trở thành kiến thức cơ bản trong xử lý ngôn ngữ tự nhiên.

### 16.1.1 Tại sao không Sử dụng Vector One-hot?

Chúng ta đã sử dụng vector one-hot để đại diện cho từ (thực chất là ký tự) trong Section 10.5. Nhớ lại rằng khi giả sử số lượng các từ riêng biệt trong từ điển (tức kích thước từ điển) là  $N$ , mỗi từ có thể tương ứng một-một với các số nguyên liên tiếp từ 0 đến  $N - 1$ , được gọi là chỉ số của từ. Giả sử chỉ số của một từ là  $i$ . Để thu được biểu diễn vector one-hot của từ đó, ta tạo một vector có  $N$  phần tử có giá trị là 0 và đặt phần tử thứ  $i$  bằng 1. Theo đó, mỗi từ được biểu diễn dưới dạng vector có độ dài  $N$  có thể được trực tiếp đưa vào mạng nơ-ron.

Mặc dù rất dễ xây dựng các vector one-hot, nhưng chúng thường không phải là lựa chọn tốt. Một trong những lý do chính là các vector one-hot không thể biểu diễn một cách chính xác độ tương tự giữa các từ khác nhau, chẳng hạn như độ tương tự cô-sin mà ta thường sử dụng. Độ tương tự cô-sin của hai vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$  là giá trị cô-sin của góc giữa chúng:

$$\frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} \in [-1, 1]. \quad (16.1.1)$$

Do độ tương tự cô-sin giữa các vector one-hot của bất kỳ hai từ khác nhau nào đều bằng 0, nên rất khó sử dụng vector one-hot để biểu diễn độ tương tự giữa các từ khác nhau.

Word2vec<sup>329</sup> là một công cụ được phát minh để giải quyết vấn đề trên. Nó biểu diễn mỗi từ bằng một vector có độ dài cố định và sử dụng những vector này để biểu thị tốt hơn độ tương tự và và các quan hệ loại suy (*analogy relationship*) giữa các từ. Công cụ Word2vec gồm hai mô hình: skip-gam (Mikolov et al., 2013b) và túi từ liên tục (*continuous bag of words – CBOW*) (Mikolov et al., 2013a). Tiếp theo, ta sẽ xem xét hai mô hình này và phương pháp huấn luyện chúng.

### 16.1.2 Mô hình Skip-Gram

Mô hình skip-gam giả định rằng một từ có thể được sử dụng để sinh ra các từ xung quanh nó trong một chuỗi văn bản. Ví dụ, giả sử chuỗi văn bản là “the”, “man”, “loves”, “his” và “son”. Ta sử dụng “loves” làm từ đích trung tâm và đặt kích thước cửa sổ ngữ cảnh bằng 2. Như mô tả trong Fig. 16.1.1, với từ đích trung tâm “loves”, mô hình skip-gram quan tâm đến xác suất có điều kiện sinh ra các từ ngữ cảnh (“the”, “man”, “his” và “son”) nằm trong khoảng cách không quá 2 từ:

$$P(\text{"the"}, \text{"man"}, \text{"his"}, \text{"son"} | \text{"loves"}). \quad (16.1.2)$$

Ta giả định rằng, với từ đích trung tâm cho trước, các từ ngữ cảnh được sinh ra độc lập với nhau. Trong trường hợp này, công thức trên có thể được viết lại thành

$$P(\text{"the"} | \text{"loves"}) \cdot P(\text{"man"} | \text{"loves"}) \cdot P(\text{"his"} | \text{"loves"}) \cdot P(\text{"son"} | \text{"loves"}). \quad (16.1.3)$$

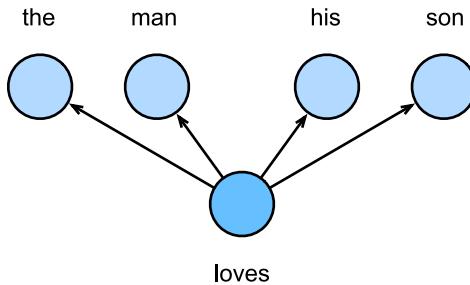


Fig. 16.1.1: Mô hình skip-gram quan tâm đến xác suất có điều kiện sinh ra các từ ngữ cảnh với một từ đích trung tâm cho trước.

Trong mô hình skip-gam, mỗi từ được biểu diễn bằng hai vector  $d$ -chiều để tính xác suất có điều kiện. Giả sử chỉ số của một từ trong từ điển là  $i$ , vector của từ được biểu diễn là  $\mathbf{v}_i \in \mathbb{R}^d$  khi từ này là từ đích trung tâm và là  $\mathbf{u}_i \in \mathbb{R}^d$  khi từ này là một từ ngữ cảnh. Gọi  $c$  và  $o$  lần lượt là chỉ số của từ đích trung tâm  $w_c$  và từ ngữ cảnh  $w_o$  trong từ điển. Có thể thu được xác suất có điều kiện sinh ra từ ngữ cảnh cho một từ đích trung tâm cho trước bằng phép toán softmax trên tích vô hướng của vector:

$$P(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}, \quad (16.1.4)$$

trong đó, tập chỉ số trong bộ từ vựng là  $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$ . Giả sử trong một chuỗi văn bản có độ dài  $T$ , từ tại bước thời gian  $t$  được ký hiệu là  $w^{(t)}$ . Giả sử rằng các từ ngữ cảnh được sinh độc lập

<sup>329</sup> <https://code.google.com/archive/p/word2vec/>

với từ trung tâm cho trước. Khi kích thước cửa sổ ngữ cảnh là  $m$ , hàm hợp lý (*likelihood*) của mô hình skip-gam là xác suất kết hợp sinh ra tất cả các từ ngữ cảnh với bất kỳ từ trung tâm cho trước nào

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} | w^{(t)}), \quad (16.1.5)$$

Ở đây, bất kỳ bước thời gian nào nhỏ hơn 1 hoặc lớn hơn  $T$  đều có thể được bỏ qua.

### Huấn luyện Mô hình Skip-Gram

Các tham số trong mô hình skip-gram là vector từ đích trung tâm và vector từ ngữ cảnh cho từng từ riêng lẻ. Trong quá trình huấn luyện, chúng ta sẽ học các tham số mô hình bằng cách cực đại hóa hàm hợp lý, còn gọi là ước lượng hợp lý cực đại. Việc này tương tự với việc giảm thiểu hàm mất mát sau đây:

$$-\sum_{t=1}^T \sum_{-m \leq j \leq m, j \neq 0} \log P(w^{(t+j)} | w^{(t)}). \quad (16.1.6)$$

Nếu ta dùng SGD, thì trong mỗi vòng lặp, ta chọn ra một chuỗi con nhỏ hơn bằng việc lấy mẫu ngẫu nhiên để tính toán mất mát cho chuỗi con đó, rồi sau đó tính gradient để cập nhật các tham số mô hình. Điểm then chốt của việc tính toán gradient là tính gradient của logarit xác suất có điều kiện cho vector từ trung tâm và vector từ ngữ cảnh. Đầu tiên, theo định nghĩa ta có

$$\log P(w_o | w_c) = \mathbf{u}_o^\top \mathbf{v}_c - \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \quad (16.1.7)$$

Qua phép tính đạo hàm, ta nhận được giá trị gradient  $\mathbf{v}_c$  từ công thức trên.

$$\begin{aligned} \frac{\partial \log P(w_o | w_c)}{\partial \mathbf{v}_c} &= \mathbf{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\mathbf{u}_j^\top \mathbf{v}_c) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} \left( \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)} \right) \mathbf{u}_j \\ &= \mathbf{u}_o - \sum_{j \in \mathcal{V}} P(w_j | w_c) \mathbf{u}_j. \end{aligned} \quad (16.1.8)$$

Phép tính cho ra xác suất có điều kiện cho mọi từ có trong từ điển với từ đích trung tâm  $w_c$  cho trước. Sau đó, ta lại sử dụng phương pháp đó để tìm gradient cho các vector từ khác.

Sau khi huấn luyện xong, với từ bất kỳ có chỉ số là  $i$  trong từ điển, ta sẽ nhận được tập hai vector từ  $\mathbf{v}_i$  và  $\mathbf{u}_i$ . Trong các ứng dụng xử lý ngôn ngữ tự nhiên, vector từ đích trung tâm trong mô hình skip-gram thường được sử dụng để làm vector biểu diễn một từ.

### 16.1.3 Mô hình Túi từ Liên tục (CBOW)

Mô hình túi từ liên tục (*Continuous bag of words* - CBOW) tương tự như mô hình skip-gram. Khác biệt lớn nhất là mô hình CBOW giả định rằng từ đích trung tâm được tạo ra dựa trên các từ ngữ cảnh phía trước và sau nó trong một chuỗi văn bản. Với cùng một chuỗi văn bản gồm các từ “the”, “man”, “loves”, “his” và “son”, trong đó “love” là từ đích trung tâm, với kích thước cửa sổ ngữ cảnh bằng 2, mô hình CBOW quan tâm đến xác suất có điều kiện để sinh ra từ đích “love” dựa trên các từ ngữ cảnh “the”, “man”, “his” và “son” (minh họa ở Fig. 16.1.2) như sau:

$$P(\text{"loves"} \mid \text{"the"}, \text{"man"}, \text{"his"}, \text{"son"}). \quad (16.1.9)$$

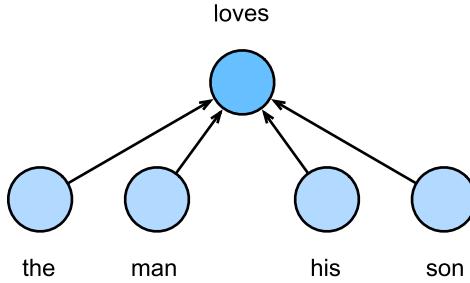


Fig. 16.1.2: Mô hình CBOW quan tâm đến xác suất có điều kiện tạo ra từ đích trung tâm dựa trên các từ ngữ cảnh cho trước.

Vì có quá nhiều từ ngữ cảnh trong mô hình CBOW, ta sẽ lấy trung bình các vector từ của chúng và sau đó sử dụng phương pháp tương tự như trong mô hình skip-gram để tính xác suất có điều kiện. Giả sử  $\mathbf{v}_i \in \mathbb{R}^d$  và  $\mathbf{u}_i \in \mathbb{R}^d$  là vector từ ngữ cảnh và vector từ đích trung tâm của từ có chỉ số  $i$  trong từ điển (lưu ý rằng các ký hiệu này ngược với các ký hiệu trong mô hình skip-gram). Gọi  $c$  là chỉ số của từ đích trung tâm  $w_c$ , và  $o_1, \dots, o_{2m}$  là chỉ số các từ ngữ cảnh  $w_{o_1}, \dots, w_{o_{2m}}$  trong từ điển. Do đó, xác suất có điều kiện sinh ra từ đích trung tâm dựa vào các từ ngữ cảnh cho trước là

$$P(w_c \mid w_{o_1}, \dots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\mathbf{u}_c^\top(\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\mathbf{u}_i^\top(\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})\right)}. \quad (16.1.10)$$

Để rút gọn, ký hiệu  $\mathcal{W}_o = \{w_{o_1}, \dots, w_{o_{2m}}\}$ , và  $\bar{\mathbf{v}}_o = (\mathbf{v}_{o_1} + \dots + \mathbf{v}_{o_{2m}})/(2m)$ . Phương trình trên được đơn giản hóa thành

$$P(w_c \mid \mathcal{W}_o) = \frac{\exp(\mathbf{u}_c^\top \bar{\mathbf{v}}_o)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)}. \quad (16.1.11)$$

Cho một chuỗi văn bản có độ dài  $T$ , ta giả định rằng từ xuất hiện tại bước thời gian  $t$  là  $w^{(t)}$ , và kích thước của cửa sổ ngữ cảnh là  $m$ . Hàm hợp lý của mô hình CBOW là xác suất sinh ra bất kỳ từ đích trung tâm nào dựa vào những từ ngữ cảnh.

$$\prod_{t=1}^T P(w^{(t)} \mid w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}). \quad (16.1.12)$$

## Huấn luyện Mô hình CBOW

Quá trình huấn luyện mô hình CBOW khá giống với quá trình huấn luyện mô hình skip-gram. Uớc lượng hợp lý cực đại của mô hình CBOW tương đương với việc cực tiểu hóa hàm mất mát:

$$-\sum_{t=1}^T \log P(w^{(t)} | w^{(t-m)}, \dots, w^{(t-1)}, w^{(t+1)}, \dots, w^{(t+m)}). \quad (16.1.13)$$

Lưu ý rằng

$$\log P(w_c | \mathcal{W}_o) = \mathbf{u}_c^\top \bar{\mathbf{v}}_o - \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o) \right). \quad (16.1.14)$$

Thông qua phép đạo hàm, ta có thể tính log của xác suất có điều kiện của gradient của bất kỳ vector từ ngữ cảnh nào  $\mathbf{v}_{o_i}$  ( $i = 1, \dots, 2m$ ) trong công thức trên.

$$\frac{\partial \log P(w_c | \mathcal{W}_o)}{\partial \mathbf{v}_{o_i}} = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\mathbf{u}_j^\top \bar{\mathbf{v}}_o) \mathbf{u}_j}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \bar{\mathbf{v}}_o)} \right) = \frac{1}{2m} \left( \mathbf{u}_c - \sum_{j \in \mathcal{V}} P(w_j | \mathcal{W}_o) \mathbf{u}_j \right). \quad (16.1.15)$$

Sau đó, ta sử dụng cùng phương pháp đó để tính gradient cho các vector của từ khác. Không giống như mô hình skip-gam, trong mô hình CBOW ta thường sử dụng vector từ ngữ cảnh làm vector biểu diễn một từ.

### 16.1.4 Tóm tắt

- Vector từ là một vector được sử dụng để biểu diễn một từ. Kỹ thuật ánh xạ các từ sang vector số thực còn được gọi là kỹ thuật embedding từ.
- Word2vec bao gồm cả mô hình túi từ liên tục (CBOW) và mô hình skip-gam. Mô hình skip-gam giả định rằng các từ ngữ cảnh được sinh ra dựa trên từ đích trung tâm. Mô hình CBOW giả định rằng từ đích trung tâm được sinh ra dựa trên các từ ngữ cảnh.

### 16.1.5 Bài tập

1. Độ phức tạp tính toán của mỗi gradient là bao nhiêu? Nếu từ điển chứa một lượng lớn các từ, điều này sẽ gây ra vấn đề gì?
2. Có một số cụm từ cố định trong tiếng Anh bao gồm nhiều từ, chẳng hạn như “new york”. Bạn sẽ huấn luyện các vector từ của chúng như thế nào? Gợi ý: Xem phần 4 trong bài báo Word2vec[2].
3. Sử dụng mô hình skip-gam làm ví dụ để tìm hiểu về thiết kế của mô hình word2vec. Mỗi quan hệ giữa tích vô hướng của hai vector từ và độ tương tự cô-sin trong mô hình skip-gam là gì? Đối với một cặp từ có ngữ nghĩa gần nhau, tại sao hai vector từ này lại thường có độ tương tự cô-sin cao?

### 16.1.6 Thảo luận

- Tiếng Anh: MXNet<sup>330</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>331</sup>

### 16.1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Phạm Đăng Khoa
- Lê Khắc Hồng Phúc

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 29/08/2020)

## 16.2 Huấn luyện Gần đúng

Hãy nhớ lại nội dung của phần trước. Đặc điểm cốt lõi của mô hình skip-gram là việc sử dụng các toán tử softmax để tính xác suất có điều kiện sinh ra từ ngữ cảnh  $w_o$  dựa trên từ đích trung tâm cho trước  $w_c$ .

$$P(w_o | w_c) = \frac{\exp(\mathbf{u}_o^\top \mathbf{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c)}. \quad (16.2.1)$$

Mất mát logarit tương ứng với xác suất có điều kiện trên được tính như sau

$$-\log P(w_o | w_c) = -\mathbf{u}_o^\top \mathbf{v}_c + \log \left( \sum_{i \in \mathcal{V}} \exp(\mathbf{u}_i^\top \mathbf{v}_c) \right). \quad (16.2.2)$$

Do toán tử softmax xem xét từ ngữ cảnh có thể là bất kỳ từ nào trong từ điển  $\mathcal{V}$ , nên mất mát được đề cập ở trên thật ra bao gồm phép lấy tổng qua tất cả phần tử trong từ điển. Ở phần trước, ta đã biết rằng cả hai mô hình skip-gram và CBOW đều tính xác suất có điều kiện thông qua toán tử softmax, do đó việc tính toán gradient cho mỗi bước bao gồm phép lấy tổng qua toàn bộ các phần tử trong từ điển. Đối với các từ điển lớn hơn với hàng trăm nghìn hoặc thậm chí hàng triệu từ, chi phí tính toán cho mỗi gradient có thể rất cao. Để giảm độ phức tạp tính toán này, chúng tôi sẽ giới thiệu hai phương pháp huấn luyện gần đúng trong phần này, đó là lấy mẫu âm (*negative sampling*) và toán tử softmax phân cấp (*hierarchical softmax*). Do không có sự khác biệt lớn giữa mô hình skip-gram và mô hình CBOW, trong phần này ta chỉ sử dụng mô hình skip-gram làm ví dụ để giới thiệu hai phương pháp huấn luyện trên.

<sup>330</sup> <https://discuss.d2l.ai/t/381>

<sup>331</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 16.2.1 Lấy mẫu Âm

Phương pháp lấy mẫu âm sửa đổi hàm mục tiêu ban đầu. Cho một cửa sổ ngữ cảnh với từ đích trung tâm  $w_c$ , ta xem việc từ ngữ cảnh  $w_o$  xuất hiện trong cửa sổ ngữ cảnh là một sự kiện và tính xác suất của sự kiện này theo

$$P(D = 1 \mid w_c, w_o) = \sigma(\mathbf{u}_o^\top \mathbf{v}_c), \quad (16.2.3)$$

Ở đây, hàm  $\sigma$  có cùng định nghĩa với hàm kích hoạt sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}. \quad (16.2.4)$$

Đầu tiên, ta sẽ xem xét việc huấn luyện vector từ (*word vector*) bằng cách cực đại hóa xác suất kết hợp của tất cả các sự kiện trong chuỗi văn bản. Cho một chuỗi văn bản có độ dài  $T$ , ta giả sử rằng từ tại bước thời gian  $t$  là  $w^{(t)}$  và kích thước cửa sổ ngữ cảnh là  $m$ . Nay giờ, ta sẽ xem xét việc cực đại hóa xác suất kết hợp

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(D = 1 \mid w^{(t)}, w^{(t+j)}). \quad (16.2.5)$$

Tuy nhiên, các sự kiện trong mô hình chỉ xem xét các mẫu dương. Trong trường hợp này, chỉ khi tất cả các vector từ bằng nhau và giá trị của chúng tiến tới vô cùng, xác suất kết hợp trên mới có thể đạt giá trị cực đại bằng 1. Rõ ràng, các vector từ như vậy là vô nghĩa. Phương pháp lấy mẫu âm khiến hàm mục tiêu có ý nghĩa hơn bằng cách lấy thêm các mẫu âm. Giả sử sự kiện  $P$  xảy ra khi từ ngữ cảnh  $w_o$  xuất hiện trong cửa sổ ngữ cảnh của từ đích trung tâm  $w_c$ , và ta lấy mẫu  $K$  từ không xuất hiện trong cửa sổ ngữ cảnh, đóng vai trò là các từ nhiễu, theo phân phối  $P(w)$ . Ta giả sử sự kiện từ nhiễu  $w_k (k = 1, \dots, K)$  không xuất hiện trong cửa sổ ngữ cảnh của từ đích trung tâm  $w_c$  là  $N_k$ . Giả sử các sự kiện  $P$  và  $N_1, \dots, N_K$  cho cả mẫu dương lẫn và mẫu âm là độc lập với nhau. Bằng cách xem xét phương pháp lấy mẫu âm, ta có thể viết lại xác suất kết hợp chỉ xem xét các mẫu dương ở trên như sau

$$\prod_{t=1}^T \prod_{-m \leq j \leq m, j \neq 0} P(w^{(t+j)} \mid w^{(t)}), \quad (16.2.6)$$

Ở đây, xác suất có điều kiện được tính gần đúng bằng

$$P(w^{(t+j)} \mid w^{(t)}) = P(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1, w_k \sim P(w)}^K P(D = 0 \mid w^{(t)}, w_k). \quad (16.2.7)$$

Đặt chỉ số của từ  $w^{(t)}$  trong chuỗi văn bản tại bước thời gian  $t$  là  $i_t$  và chỉ số của từ nhiễu  $w_k$  trong từ điển là  $h_k$ . Mất mát logarit cho xác suất có điều kiện ở trên là

$$\begin{aligned} -\log P(w^{(t+j)} \mid w^{(t)}) &= -\log P(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1, w_k \sim P(w)}^K \log P(D = 0 \mid w^{(t)}, w_k) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log (1 - \sigma(\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t})) \\ &= -\log \sigma(\mathbf{u}_{i_{t+j}}^\top \mathbf{v}_{i_t}) - \sum_{k=1, w_k \sim P(w)}^K \log \sigma(-\mathbf{u}_{h_k}^\top \mathbf{v}_{i_t}). \end{aligned} \quad (16.2.8)$$

Ở đây, tính toán gradient trong mỗi bước huấn luyện không còn liên quan đến kích thước từ điển, mà có quan hệ tuyến tính với  $K$ . Khi  $K$  có giá trị nhỏ hơn, thì phương pháp lấy mẫu âm có chi phí tính toán cho mỗi bước thấp hơn.

### 16.2.2 Softmax Phân cấp

Softmax phân cấp (*Hierarchical softmax*) là một phương pháp huấn luyện gần đúng khác. Phương pháp này sử dụng cấu trúc dữ liệu cây nhị phân như minh họa trong Fig. 16.2.1, với các nút lá của cây biểu diễn tất cả các từ trong từ điển  $\mathcal{V}$ .

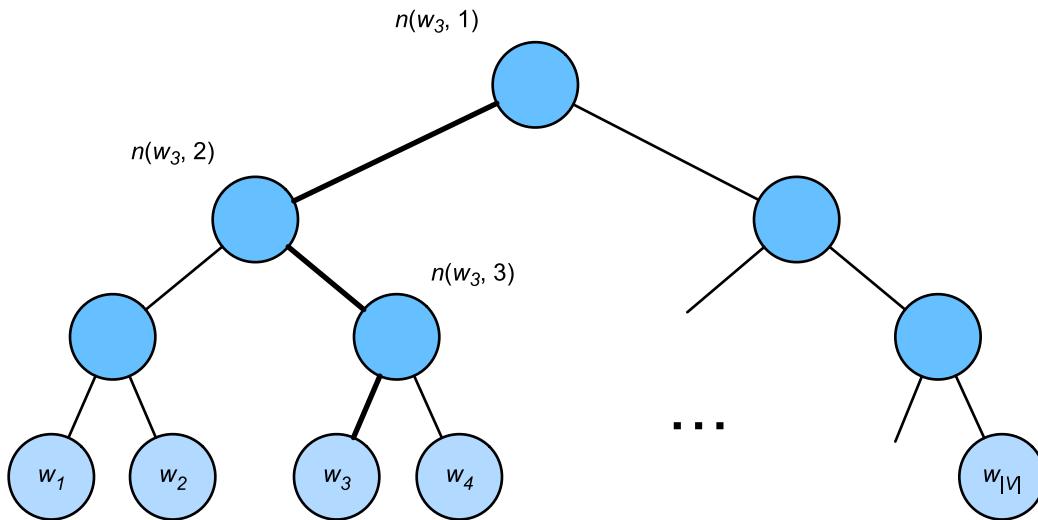


Fig. 16.2.1: Softmax Phân cấp. Mỗi nút lá của cây biểu diễn một từ trong từ điển.

Ta giả định  $L(w)$  là số nút trên đường đi (gồm cả gốc lẫn các nút lá) từ gốc của cây nhị phân đến nút lá của từ  $w$ . Gọi  $n(w, j)$  là nút thứ  $j$  trên đường đi này, với vector ngữ cảnh của từ là  $\mathbf{u}_{n(w,j)}$ . Ta sử dụng ví dụ trong Fig. 16.2.1, theo đó  $L(w_3) = 4$ . Softmax phân cấp tính xấp xỉ xác suất có điều kiện trong mô hình skip-gram như sau

$$P(w_o | w_c) = \prod_{j=1}^{L(w_o)-1} \sigma \left( \llbracket n(w_o, j+1) = \text{leftChild}(n(w_o, j)) \rrbracket \cdot \mathbf{u}_{n(w_o, j)}^\top \mathbf{v}_c \right), \quad (16.2.9)$$

Trong đó hàm  $\sigma$  có định nghĩa giống với hàm kích hoạt sigmoid, và  $\text{leftChild}(n)$  là nút con bên trái của nút  $n$ . Nếu  $x$  đúng thì  $\llbracket x \rrbracket = 1$ ; ngược lại  $\llbracket x \rrbracket = -1$ . Giờ ta sẽ tính xác suất có điều kiện của việc sinh ra từ  $w_3$  dựa theo từ  $w_c$  được cho trong Fig. 16.2.1. Ta cần tìm tích vô hướng của vector từ  $\mathbf{v}_c$  (cho từ  $w_c$ ) với mỗi vector nút mà không phải là nút lá trên đường đi từ nút gốc đến  $w_3$ . Do trong cây nhị phân, đường đi từ nút gốc đến nút lá  $w_3$  là trái, phải, rồi lại trái (đường đi được in đậm trong Fig. 16.2.1) nên ta có

$$P(w_3 | w_c) = \sigma(\mathbf{u}_{n(w_3, 1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3, 2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3, 3)}^\top \mathbf{v}_c). \quad (16.2.10)$$

Do  $\sigma(x) + \sigma(-x) = 1$  nên điều kiện mà tổng xác suất có điều kiện của bất kì từ nào trong từ điển  $\mathcal{V}$  được sinh ra dựa trên từ đích trung tâm cho trước  $w_c$  phải bằng 1 cũng được thỏa mãn:

$$\sum_{w \in \mathcal{V}} P(w | w_c) = 1. \quad (16.2.11)$$

Hơn nữa, do độ lớn của  $L(w_o) - 1$  là  $\mathcal{O}(\log_2 |\mathcal{V}|)$  nên khi kích thước từ điển  $\mathcal{V}$  lớn, chi phí tính toán phụ trợ tại mỗi bước trong softmax phân cấp được giảm đáng kể so với khi không áp dụng huấn luyện gần đúng.

### 16.2.3 Tóm tắt

- Lấy mẫu âm xây dựng hàm mất mát bằng cách xét các sự kiện độc lập bao gồm cả mẫu âm lẫn mẫu dương. Chi phí tính toán gradient tại mỗi bước trong quá trình huấn luyện có mối quan hệ tuyến tính với số từ nhiều mà ta lấy mẫu.
- Softmax phân cấp sử dụng một cây nhị phân và xây dựng hàm mất mát dựa trên đường đi từ nút gốc đến nút lá. Chi phí phụ trợ khi tính toán gradient tại mỗi bước trong quá trình huấn luyện có mối quan hệ theo hàm logarit với kích thước từ điển.

### 16.2.4 Bài tập

1. Trước khi đọc phần tiếp theo, hãy nghĩ xem ta nên lấy mẫu các từ nào như thế nào trong kĩ thuật lấy mẫu âm.
2. Điều gì giúp cho công thức cuối cùng trong phần này là đúng?
3. Ta có thể áp dụng lấy mẫu âm và softmax phân cấp như thế nào trong mô hình skip-gram?

### 16.2.5 Thảo luận

- Tiếng Anh: Main Forum<sup>332</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>333</sup>

### 16.2.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Văn Cường
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức
- Lê Khắc Hồng Phúc

*Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 29/08/2020)*

<sup>332</sup> <https://discuss.d2l.ai/t/382>

<sup>333</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 16.3 Tập dữ liệu để Tiền Huấn luyện Embedding Từ

Trong phần này, chúng tôi sẽ giới thiệu cách tiền xử lý một tập dữ liệu với phương pháp lấy mẫu âm Section 16.2 và tạo các minibatch để huấn luyện word2vec. Tập dữ liệu mà ta sẽ sử dụng là Penn Tree Bank (PTB)<sup>334</sup>, một kho ngữ liệu nhỏ nhưng được sử dụng phổ biến. Tập dữ liệu này được thu thập từ các bài báo của Wall Street Journal và bao gồm các tập huấn luyện, tập kiểm định và tập kiểm tra.

Đầu tiên, ta nhập các gói và mô-đun cần thiết cho thí nghiệm.

```
from d2l import mxnet as d2l
import math
from mxnet import gluon, np
import os
import random
```

### 16.3.1 Đọc và Tiền xử lý Dữ liệu

Tập dữ liệu này đã được tiền xử lý trước. Mỗi dòng của tập dữ liệu được xem là một câu. Tất cả các từ trong một câu được phân cách bằng dấu cách. Trong bài toán embedding từ, mỗi từ là một token.

```
#@save
d2l.DATA_HUB['ptb'] = (d2l.DATA_URL + 'ptb.zip',
                        '319d85e578af0cdc590547f26231e4e31cdf1e42')

#@save
def read_ptb():
    data_dir = d2l.download_extract('ptb')
    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)}'
```

Tiếp theo ta sẽ xây dựng bộ từ vựng, trong đó các từ xuất hiện dưới 10 lần sẽ được xem như token “<unk>”. Lưu ý rằng tập dữ liệu PTB đã được tiền xử lý cũng chứa các token “<unk>” đại diện cho các từ hiếm gặp.

```
vocab = d2l.Vocab(sentences, min_freq=10)
f'vocab size: {len(vocab)}'
```

<sup>334</sup> <https://catalog.ldc.upenn.edu/LDC99T42>

### 16.3.2 Lấy mẫu con

Trong dữ liệu văn bản, thường có một số từ xuất hiện với tần suất cao, chẳng hạn như các từ “the”, “a” và “in” trong tiếng Anh. Nói chung, trong cửa sổ ngữ cảnh, sẽ tốt hơn nếu ta huấn luyện mô hình embedding từ khi một từ bình thường (chẳng hạn như “chip”) và một từ có tần suất thấp hơn (chẳng hạn như “microprocessor”) xuất hiện cùng lúc, hơn là khi một từ bình thường xuất hiện với một từ có tần suất cao hơn (chẳng hạn như “the”). Do đó, khi huấn luyện mô hình embedding từ, ta có thể thực hiện lấy mẫu con [2] trên các từ. Cụ thể, mỗi từ  $w_i$  được gán chỉ số trong tập dữ liệu sẽ bị loại bỏ với một xác suất nhất định. Xác suất loại bỏ được tính như sau:

$$P(w_i) = \max \left( 1 - \sqrt{\frac{t}{f(w_i)}}, 0 \right), \quad (16.3.1)$$

Ở đây,  $f(w_i)$  là tỷ lệ giữa số lần xuất hiện từ  $w_i$  với tổng số từ trong tập dữ liệu, và hằng số  $t$  là một siêu tham số (có giá trị bằng  $10^{-4}$  trong thí nghiệm này). Như ta có thể thấy, từ  $w_i$  chỉ có thể được loại bỏ trong lúc lấy mẫu con khi  $f(w_i) > t$ . Tần suất của từ càng cao, xác suất loại bỏ càng lớn.

```
#@save
def subsampling(sentences, vocab):
    # Map low frequency words into <unk>
    sentences = [[vocab.idx_to_token[vocab[tk]] for tk in line]
                  for line in sentences]
    # Count the frequency for each word
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # Return True if to keep this token during subsampling
    def keep(token):
        return(random.uniform(0, 1) <
               math.sqrt(1e-4 / counter[token] * num_tokens))

    # Now do the subsampling
    return [[tk for tk in line if keep(tk)] for line in sentences]

subsampled = subsampling(sentences, vocab)
```

So sánh độ dài chuỗi trước và sau khi lấy mẫu, ta có thể thấy việc lấy mẫu con làm giảm đáng kể độ dài chuỗi.

```
d2l.set_figsize()
d2l.plt.hist([[len(line) for line in sentences],
              [len(line) for line in subsampled]])
d2l.plt.xlabel('# tokens per sentence')
d2l.plt.ylabel('count')
d2l.plt.legend(['origin', 'subsampled']);
```

Với các token riêng lẻ, tỉ lệ lấy mẫu của các từ có tần suất cao như từ “the” nhỏ hơn 1/20.

```
def compare_counts(token):
    return (f'# of "{token}": '
           f'before={sum([line.count(token) for line in sentences])}, '
           f'after={sum([line.count(token) for line in subsampled])}')

compare_counts('the')
```

Nhưng các từ có tần số thấp như từ “join” hoàn toàn được giữ nguyên.

```
compare_counts('join')
```

Cuối cùng, ta ánh xạ từng token tới một chỉ số tương ứng để xây dựng kho ngữ liệu.

```
corpus = [vocab[line] for line in subsampled]
corpus[0:3]
```

### 16.3.3 Nạp Dữ liệu

Tiếp theo, ta đọc kho ngữ liệu với các chỉ số token thành các batch dữ liệu cho quá trình huấn luyện.

#### Trích xuất từ Đích Trung tâm và Từ Ngữ cảnh

Ta sử dụng các từ với khoảng cách tới từ đích trung tâm không quá độ dài cửa sổ ngữ cảnh để làm từ ngữ cảnh cho từ đích trung tâm đó. Hàm sau đây trích xuất tất cả từ đích trung tâm và các từ ngữ cảnh của chúng. Ta chọn kích thước cửa sổ ngữ cảnh là một số nguyên từ 1 tới max\_window\_size (kích thước cửa sổ tối đa), được lấy ngẫu nhiên theo phân phối đều.

```
#@save
def get_centers_and_contexts(corpus, max_window_size):
    centers, contexts = [], []
    for line in corpus:
        # Each sentence needs at least 2 words to form a "central target word"
        # - context word" pair
        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 central target word from the context words
            indices.remove(i)
            contexts.append([line[idx] for idx in indices])
    return centers, contexts
```

Kế tiếp, ta tạo một tập dữ liệu nhân tạo chứa hai câu có lần lượt 7 và 3 từ. Hãy giả sử cửa sổ ngữ cảnh cực đại là 2 và in tất cả các từ đích 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)
```

Ta thiết lập cửa sổ ngữ cảnh cực đại là 5. Đoạn mã sau trích xuất tất cả các từ đích 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: {len(all_centers)}}'
```

## Lấy mẫu Âm

Ta thực hiện lấy mẫu âm để huấn luyện gần đúng. Với mỗi cặp từ đích trung tâm và ngữ cảnh, ta lấy mẫu ngẫu nhiên  $K$  từ nhiều ( $K = 5$  trong thử nghiệm này). Theo đề xuất trong bài báo Word2vec, xác suất lấy mẫu từ nhiều  $P(w)$  là tỷ lệ giữa tần suất xuất hiện của từ  $w$  và tổng tần suất xuất hiện của tất cả các từ, lấy mủ 0.75 [2].

Trước hết ta sẽ định nghĩa một lớp để lấy ra một ứng cử viên dựa theo các trọng số lấy mẫu. Lớp này sẽ lưu lại 10000 số ngẫu nhiên một lần thay vì gọi `random.choices` liên tục.

```
#@save
class RandomGenerator:
    """Draw a random int in [0, n] according to n sampling weights."""
    def __init__(self, sampling_weights):
        self.population = list(range(len(sampling_weights)))
        self.sampling_weights = sampling_weights
        self.candidates = []
        self.i = 0

    def draw(self):
        if self.i == len(self.candidates):
            self.candidates = random.choices(
                self.population, self.sampling_weights, k=10000)
            self.i = 0
        self.i += 1
        return self.candidates[self.i-1]

generator = RandomGenerator([2, 3, 4])
[generator.draw() for _ in range(10)]
```

```
#@save
def get_negatives(all_contexts, corpus, K):
    counter = d2l.count_corpus(corpus)
    sampling_weights = [counter[i]**0.75 for i in range(len(counter))]
    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, corpus, 5)
```

## Đọc Dữ liệu thành Batch

Chúng ta trích xuất tất cả các từ đích trung tâm all\_centers, cũng như các từ ngữ cảnh all\_contexts và những từ nhiễu của mỗi từ đích trung tâm trong tập dữ liệu, rồi đọc chúng thành các minibatch ngẫu nhiên.

Trong một minibatch dữ liệu, mẫu thứ  $i$  bao gồm một từ đích trung tâm cùng  $n_i$  từ ngữ cảnh và  $m_i$  từ nhiễu tương ứng với từ đích trung tâm đó. Do kích thước cửa sổ ngữ cảnh của mỗi mẫu có thể khác nhau, nên tổng số từ ngữ cảnh và từ nhiễu,  $n_i + m_i$ , cũng sẽ khác nhau. Khi tạo một minibatch, chúng ta nối (*concatenate*) các từ ngữ cảnh và các từ nhiễu của mỗi mẫu, và đệm thêm các giá trị 0 để độ dài của các đoạn nối bằng nhau, tức bằng  $\max_i n_i + m_i$  (`max_len`). Nhằm tránh ảnh hưởng của phần đệm lên việc tính toán hàm mất mát, chúng ta tạo một biến mặt nạ `masks`, mỗi phần tử trong đó tương ứng với một phần tử trong phần nối giữa từ ngữ cảnh và từ nhiễu, `contexts_negatives`. Khi một phần tử trong biến `contexts_negatives` là đệm, thì phần tử trong biến mặt nạ `masks` ở vị trí đó sẽ là 0, còn lại là bằng 1. Để phân biệt giữa các mẫu dương và âm, chúng ta cũng cần phân biệt các từ ngữ cảnh với các từ nhiễu trong biến `contexts_negatives`. Dựa trên cấu tạo của biến mặt nạ, chúng ta chỉ cần tạo một biến nhãn `labels` có cùng kích thước với biến `contexts_negatives` và đặt giá trị các phần tử tương ứng với các từ ngữ cảnh (mẫu dương) bằng 1 và phần còn lại bằng 0.

Tiếp đó, chúng ta lập trình chức năng đọc minibatch `batchify`, với đầu vào minibatch data là một danh sách có độ dài là kích thước batch, mỗi phần tử trong đó chứa các từ đích trung tâm `center`, các từ ngữ cảnh `context` và các từ nhiễu `negative`. Dữ liệu trong minibatch được trả về bởi hàm này đều tuân theo định dạng chúng ta cần, bao gồm biến mặt nạ.

```
#@save
def batchify(data):
    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)]
        masks += [[1] * cur_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len - len(context))]

    return (np.array(centers).reshape(-1, 1), np.array(contexts_negatives),
            np.array(masks), np.array(labels))
```

Tạo hai ví dụ mẫu đơn giản:

```
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)
```

Chúng ta dùng hàm `batchify` vừa được định nghĩa để chỉ định phương thức đọc minibatch trong thực thể `DataLoader`.

#### 16.3.4 Kết hợp mọi thứ cùng nhau

Cuối cùng, chúng ta định nghĩa hàm load\_data\_ptb để đọc tập dữ liệu PTB và trả về iterator dữ liệu.

```
#@save
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    num_workers = d2l.get_dataloader_workers()
    sentences = read_ptb()
    vocab = d2l.Vocab(sentences, min_freq=10)
    subsampled = subsampling(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, corpus, num_noise_words)
    dataset = gluon.data.ArrayDataset(
        all_centers, all_contexts, all_negatives)
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle=True,
                                      batchify_fn=batchify,
                                      num_workers=num_workers)
    return data_iter, vocab
```

Ta hãy cùng in ra minibatch đầu tiên trong iterator 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
```

#### 16.3.5 Tóm tắt

- Việc lấy mẫu con cố gắng giảm thiểu tác động của các từ có tần suất cao đến việc huấn luyện mô hình embedding từ.
- Ta có thể đếm để tạo ra các minibatch với các mẫu có cùng độ dài và sử dụng các biến mặt nạ để phân biệt phần tử đếm, vì thế chỉ có những phần tử không phải đếm mới được dùng để tính toán hàm mất mát.

#### 16.3.6 Bài tập

Chúng ta sử dụng hàm batchify để chỉ định phương thức đọc minibatch trong thực thể DataLoader và in ra kích thước của từng biến trong lần đọc batch đầu tiên. Những kích thước này được tính toán như thế nào?

### 16.3.7 Thảo luận

- Tiếng Anh - MXNet<sup>335</sup>
- Tiếng Việt<sup>336</sup>

### 16.3.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Phạm Đăng Khoa
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Hồng Vinh

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

## 16.4 Tiền huấn luyện word2vec

Trong phần này, ta sẽ huấn luyện một mô hình skip-gram đã được định nghĩa ở Section 16.1.

Đầu tiên, ta nhập các gói thư viện và mô-đun cần thiết cho thí nghiệm, và nạp tập dữ liệu PTB.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
npx.set_np()

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)
```

<sup>335</sup> <https://discuss.d2l.ai/t/383>

<sup>336</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### 16.4.1 Mô hình Skip-Gram

Ta sẽ lập trình mô hình skip-gram bằng cách sử dụng các tầng embedding và phép nhân minibatch. Các phương pháp này cũng thường được sử dụng để lập trình các ứng dụng xử lý ngôn ngữ tự nhiên khác.

##### Tầng Embedding

Để thu được các embedding từ, ta sử dụng tầng embedding, có thể được tạo bằng một thực thể `nn.Embedding` trong Gluon. Trọng số của tầng embedding là một ma trận có số hàng là kích thước từ điển (`input_dim`) và số cột là chiều của mỗi vector từ (`output_dim`). Ta đặt kích thước từ điển bằng 20 và chiều vector từ là 4.

```
embed = nn.Embedding(input_dim=20, output_dim=4)
embed.initialize()
embed.weight
```

Đầu vào của tầng embedding là chỉ số của từ. Khi ta nhập vào chỉ số  $i$  của một từ, tầng embedding sẽ trả về vector từ tương ứng là hàng thứ  $i$  của ma trận trọng số. Dưới đây ta nhập vào tầng embedding một chỉ số có kích thước  $(2, 3)$ . Vì số chiều vector từ là 4, ta thu được vector từ kích thước  $(2, 3, 4)$ .

```
x = np.array([[1, 2, 3], [4, 5, 6]])
embed(x)
```

##### Phép nhân Minibatch

Ta có thể nhân các ma trận trong hai minibatch bằng toán tử nhân minibatch `batch_dot`. Giả sử batch đầu tiên chứa  $n$  ma trận  $\mathbf{X}_1, \dots, \mathbf{X}_n$  có kích thước là  $a \times b$ , và batch thứ hai chứa  $n$  ma trận  $\mathbf{Y}_1, \dots, \mathbf{Y}_n$  có kích thước là  $b \times c$ . Đầu ra của toán tử nhân ma trận trên hai batch đầu vào là  $n$  ma trận  $\mathbf{X}_1\mathbf{Y}_1, \dots, \mathbf{X}_n\mathbf{Y}_n$  có kích thước là  $a \times c$ .

Do đó, với hai tensor có kích thước là  $(n, a, b)$  và  $(n, b, c)$ , kích thước đầu ra của toán tử nhân minibatch là  $(n, a, c)$ .

```
X = np.ones((2, 1, 4))
Y = np.ones((2, 4, 6))
npx.batch_dot(X, Y).shape
```

##### Tính toán Truyền xuôi của Mô hình Skip-Gram

Ở lượt truyền xuôi, đầu vào của mô hình skip-gram chứa chỉ số center của từ đích trung tâm và chỉ số `contexts_and_negatives` được nối lại từ chỉ số của từ ngữ cảnh và từ nhiễu. Trong đó, biến `center` có kích thước là (kích thước batch, 1), và biến `contexts_and_negatives` có kích thước là (kích thước batch, `max_len`). Đầu tiên hai biến này được biến đổi từ chỉ số từ thành vector từ bởi tầng embedding từ, sau đó đầu ra có kích thước là (kích thước batch, 1, `max_len`) thu được bằng phép nhân minibatch. Mỗi phần tử của đầu ra là tích vô hướng của vector từ đích trung tâm và vector từ ngữ cảnh hoặc vector từ nhiễu.

```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = npx.batch_dot(v, u.swapaxes(1, 2))
    return pred
```

Hãy xác nhận kích thước đầu ra là (kích thước batch, 1, max\_len).

```
skip_gram(np.ones((2, 1)), np.ones((2, 4)), embed, embed).shape
```

### 16.4.2 Huấn luyện

Trước khi huấn luyện mô hình embedding từ, ta cần định nghĩa hàm mất mát của mô hình.

#### Hàm Mất mát Entropy chéo Nhị phân

Theo định nghĩa hàm mất mát trong phương pháp lấy mẫu âm, ta có thể sử dụng trực tiếp hàm mất mát entropy chéo nhị phân của Gluon SigmoidBinaryCrossEntropyLoss.

```
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
```

Lưu ý là ta có thể sử dụng biến mặt nạ để chỉ định một phần giá trị dự đoán và nhãn được dùng khi tính hàm mất mát trong minibatch: khi mặt nạ bằng 1, giá trị dự đoán và nhãn của vị trí tương ứng sẽ được dùng trong phép tính hàm mất mát; khi mặt nạ bằng 0, giá trị dự đoán và nhãn của vị trí tương ứng sẽ không được dùng trong phép tính hàm mất mát. Như đã đề cập, các biến mặt nạ có thể được sử dụng nhằm tránh ảnh hưởng của vùng đệm lêp trên phép tính hàm mất mát.

Với hai mẫu giống nhau, mặt nạ khác nhau sẽ dẫn đến giá trị mất mát cũng khác nhau.

```
pred = np.array([[.5]*4]*2)
label = np.array([[1, 0, 1, 0]]*2)
mask = np.array([[1, 1, 1, 1], [1, 1, 0, 0]])
loss(pred, label, mask)
```

Ta có thể chuẩn hóa mất mát trong từng mẫu do các mẫu có độ dài khác nhau.

```
loss(pred, label, mask) / mask.sum(axis=1) * mask.shape[1]
```

#### Khởi tạo Tham số Mô hình

Ta khai báo tầng embedding lần lượt của từ trung tâm và từ ngữ cảnh, và đặt siêu tham số số chiều của vector từ embed\_size bằng 100.

```
embed_size = 100
net = nn.Sequential()
net.add(nn.Embedding(input_dim=len(vocab), output_dim=embed_size),
        nn.Embedding(input_dim=len(vocab), output_dim=embed_size))
```

## Huấn luyện

Hàm huấn luyện được định nghĩa như dưới đây. Do có phần đệm nên phép tính mất mát có một chút khác biệt so với các hàm huấn luyện trước.

```
def train(net, data_iter, lr, num_epochs, device=d2l.try_gpu()):
    net.initialize(ctx=device, force_reinit=True)
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                            {'learning_rate': lr})
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[0, num_epochs])
    for epoch in range(num_epochs):
        timer = d2l.Timer()
        metric = d2l.Accumulator(2) # Sum of losses, no. of tokens
        for i, batch in enumerate(data_iter):
            center, context_negative, mask, label = [
                data.as_in_ctx(device) for data in batch]
            with autograd.record():
                pred = skip_gram(center, context_negative, net[0], net[1])
                l = (loss(pred.reshape(label.shape), label, mask)
                     / mask.sum(axis=1) * mask.shape[1])
            l.backward()
            trainer.step(batch_size)
            metric.add(l.sum(), l.size)
            if (i+1) % 50 == 0:
                animator.add(epoch+(i+1)/len(data_iter),
                             (metric[0]/metric[1],))
        print(f'loss {metric[0] / metric[1]:.3f}, '
              f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')
```

Giờ ta có thể huấn luyện một mô hình skip-gram sử dụng phương pháp lấy mẫu âm.

```
lr, num_epochs = 0.01, 5
train(net, data_iter, lr, num_epochs)
```

### 16.4.3 Áp dụng Mô hình Embedding Từ

Sau khi huấn luyện mô hình embedding từ, ta có thể biểu diễn sự tương tự về nghĩa giữa các từ dựa trên độ tương tự cô-sin giữa hai vector từ. Có thể thấy, khi sử dụng mô hình embedding từ đã được huấn luyện, các từ có nghĩa gần nhất với từ “chip” hầu hết là những từ có liên quan đến chip xử lý.

```
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 = np.dot(W, x) / np.sqrt(np.sum(W * W, axis=1) * np.sum(x * x) + 1e-9)
    topk = npx.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype('int32')
    for i in topk[1:]: # Remove the input words
        print(f'cosine sim={float(cos[i]):.3f}: {vocab.idx_to_token[i]}')

get_similar_tokens('chip', 3, net[0])
```

#### 16.4.4 Tóm tắt

Ta có thể tiền huấn luyện một mô hình skip-gram thông qua phương pháp lấy mẫu âm.

#### 16.4.5 Bài tập

- Đặt `sparse_grad=True` khi tạo một đối tượng `nn.Embedding`. Việc này có tăng tốc quá trình huấn luyện không? Hãy tra tài liệu của MXNet để tìm hiểu ý nghĩa của tham số này.
- Hãy tìm từ đồng nghĩa cho các từ khác.
- Điều chỉnh các siêu tham số, quan sát và phân tích kết quả thí nghiệm.
- Khi tập dữ liệu lớn, ta thường lấy mẫu các từ ngữ cảnh và các từ nhiễu cho từ đích trung tâm trong minibatch hiện tại chỉ khi cập nhật tham số mô hình. Nói cách khác, cùng một từ đích trung tâm có thể có các từ ngữ cảnh và từ nhiễu khác nhau với mỗi epoch khác nhau. Cách huấn luyện này có lợi ích gì? Hãy thử lập trình phương pháp huấn luyện này.

#### 16.4.6 Thảo luận

- Tiếng Anh: [MXNet<sup>337</sup>](#)
- Tiếng Việt: [Diễn đàn Machine Learning Cơ Bản<sup>338</sup>](#)

#### 16.4.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Đỗ Trường Giang
- Phạm Minh Đức
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

*Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 21/07/2020)*

<sup>337</sup> <https://discuss.d2l.ai/t/384>

<sup>338</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 16.5 Embedding từ với Vector Toàn cục (GloVe)

Trước tiên, ta sẽ xem lại mô hình skip-gram trong word2vec. Xác suất có điều kiện  $P(w_j | w_i)$  được biểu diễn trong mô hình skip-gram bằng hàm kích hoạt softmax sẽ được gọi là  $q_{ij}$  như sau:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)}, \quad (16.5.1)$$

Ở đây  $\mathbf{v}_i$  và  $\mathbf{u}_i$  là các biểu diễn vector từ  $w_i$  với chỉ số  $i$ , lần lượt khi nó là từ trung tâm và từ ngữ cảnh, và  $\mathcal{V} = \{0, 1, \dots, |\mathcal{V}| - 1\}$  là tập chứa các chỉ số của bộ từ vựng.

Từ  $w_i$  có thể xuất hiện trong tập dữ liệu nhiều lần. Ta gom tất cả các từ ngữ cảnh mỗi khi  $w_i$  là từ trung tâm và giữ các lần trùng lặp, rồi ký hiệu đó là tập bội  $C_i$ . Số lượng của một phần tử trong tập bội được gọi là bội số của phần tử đó. Chẳng hạn, giả sử rằng từ  $w_i$  xuất hiện hai lần trong tập dữ liệu: khi hai từ  $w_i$  đó là từ trung tâm trong chuỗi văn bản, hai cửa sổ ngữ cảnh tương ứng chứa các chỉ số từ ngữ cảnh 2, 1, 5, 2 và 2, 3, 2, 1. Khi đó, ta sẽ có tập bội  $C_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$ , trong đó bội số của phần tử 1 là 2, bội số của phần tử 2 là 4, và bội số của phần tử 3 và 5 đều là 1. Ta ký hiệu bội số của phần tử  $j$  trong tập bội  $C_i$  là  $x_{ij}$ : nó là số lần từ  $w_j$  xuất hiện trong cửa sổ ngữ cảnh khi từ trung tâm là  $w_i$  trong toàn bộ tập dữ liệu. Kết quả là hàm mất mát của mô hình skip-gram có thể được biểu diễn theo một cách khác:

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}. \quad (16.5.2)$$

Ta tính tổng số lượng tất cả các từ ngữ cảnh đối với từ trung tâm  $w_i$  để có  $x_i$ , rồi thu được xác suất có điều kiện để sinh ra từ ngữ cảnh  $w_j$  dựa trên từ trung tâm  $w_i$  là  $p_{ij}$  bằng  $x_{ij}/x_i$ . Ta có thể viết lại hàm mất mát của mô hình skip-gram như sau

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}. \quad (16.5.3)$$

Trong công thức trên,  $\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$  tính toán phân phối xác suất có điều kiện  $p_{ij}$  của việc sinh từ ngữ cảnh dựa trên từ đích trung tâm  $w_i$  và entropy chéo với phân phối xác suất có điều kiện  $q_{ij}$  được dự đoán bởi mô hình. Hàm mất mát được đánh trọng số bằng cách sử dụng tổng số từ ngữ cảnh cho từ đích trung tâm  $w_i$ . Việc cực tiểu hóa hàm mất mát theo công thức trên cho phép phân phối xác suất có điều kiện được dự đoán một cách gần nhất có thể tới phân phối xác suất có điều kiện thật sự.

Tuy nhiên, mặc dù là hàm mất mát phổ biến nhất, đôi khi hàm mất mát entropy chéo lại không phải là một lựa chọn phù hợp. Một mặt, như ta đã đề cập trong Section 16.2, chi phí để mô hình đưa ra dự đoán  $q_{ij}$  trở thành phân phối xác suất hợp lệ gồm phép lấy tổng qua toàn bộ các từ trong từ điển ở mẫu số của nó. Điều này có thể dễ dàng khiến tổng chi phí tính toán trở nên quá lớn. Mặt khác, thường sẽ có rất nhiều từ hiếm gặp trong từ điển, và chúng ít khi xuất hiện trong tập dữ liệu. Trong hàm mất mát entropy chéo, dự đoán cuối cùng cho phân phối xác suất có điều kiện trên một lượng lớn các từ hiếm gặp rất có thể sẽ không được chính xác.

### 16.5.1 Mô hình GloVe

Để giải quyết vấn đề trên, GloVe (Pennington et al., 2014), một mô hình embedding từ xuất hiện sau word2vec đã áp dụng mất mát bình phương và đề xuất ba thay đổi trong mô hình skip-gram dựa theo mất mát này.

- Ở đây, ta sử dụng các biến phân phối phi xác suất  $p'_{ij} = x_{ij}$  và  $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$  rồi tính log của chúng. Do đó, ta có mất mát bình phương  $(\log p'_{ij} - \log q'_{ij})^2 = (\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij})^2$ .
- Ta thêm hai tham số mô hình cho mỗi từ  $w_i$ : hệ số điều chỉnh  $b_i$  (cho các từ trung tâm) và  $c_i$  (cho các từ ngữ cảnh).
- Thay thế trọng số của mỗi giá trị mất mát bằng hàm  $h(x_{ij})$ . Hàm trọng số  $h(x)$  là hàm đơn điệu tăng trong khoảng  $[0, 1]$ .

Do đó, mục tiêu của GloVe là cực tiểu hóa hàm mất mát.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) (\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij})^2. \quad (16.5.4)$$

Ở đây, chúng tôi có một đề xuất đối với việc lựa chọn hàm trọng số  $h(x)$ : khi  $x < c$  (ví dụ  $c = 100$ ) thì  $h(x) = (x/c)^\alpha$  (ví dụ  $\alpha = 0.75$ ), nếu không thì  $h(x) = 1$ . Do  $h(0) = 0$ , ta có thể đơn thuần bỏ qua mất mát bình phương tại  $x_{ij} = 0$ . Khi sử dụng minibatch SGD trong quá trình huấn luyện, ta tiến hành lấy mẫu ngẫu nhiên để được một minibatch  $x_{ij}$  khác không tại mỗi bước thời gian và tính toán gradient để cập nhật các tham số mô hình. Các giá trị  $x_{ij}$  khác không trên được tính trước trên toàn bộ tập dữ liệu và là thống kê toàn cục của tập dữ liệu. Do đó, tên gọi GloVe được lấy từ “Global Vectors (Vector Toàn cục)”.

Chú ý rằng nếu từ  $w_i$  xuất hiện trong cửa sổ ngữ cảnh của từ  $w_j$  thì từ  $w_j$  cũng sẽ xuất hiện trong cửa sổ ngữ cảnh của từ  $w_i$ . Do đó,  $x_{ij} = x_{ji}$ . Không như word2vec, GloVe khớp  $\log x_{ij}$  đối xứng thay vì xác suất có điều kiện  $p_{ij}$  bất đối xứng. Do đó, vector từ đích trung tâm và vector từ ngữ cảnh của bất kỳ từ nào đều tương đương nhau trong GloVe. Tuy vậy, hai tập vector từ được học bởi cùng một mô hình về cuối có thể sẽ khác nhau do giá trị khởi tạo khác nhau. Sau khi học tất cả các vector từ, GloVe sẽ sử dụng tổng của vector từ đích trung tâm và vector từ ngữ cảnh để làm vector từ cuối cùng cho từ đó.

### 16.5.2 Lý giải GloVe bằng Tỷ số Xác suất Có điều kiện

Ta cũng có thể cố gắng lý giải embedding từ GloVe theo một cách nhìn khác. Ta sẽ tiếp tục sử dụng các ký hiệu như ở trên,  $P(w_j | w_i)$  biểu diễn xác suất có điều kiện sinh từ ngữ cảnh  $w_j$  với từ trung tâm  $w_i$  trong tập dữ liệu, và xác suất này được ghi lại bằng  $p_{ij}$ . Xét ví dụ thực tế từ một kho dữ liệu lớn, ở đây ta có hai tập các xác suất có điều kiện với “ice” và “steam” là các từ trung tâm và tỷ số giữa chúng:

| $w_k =$                       | solid    | gas      | water  | fashion  |
|-------------------------------|----------|----------|--------|----------|
| $p_1 = P(w_k   \text{ice})$   | 0.00019  | 0.000066 | 0.003  | 0.000017 |
| $p_2 = P(w_k   \text{steam})$ | 0.000022 | 0.00078  | 0.0022 | 0.000018 |
| $p_1/p_2$                     | 8.9      | 0.085    | 1.36   | 0.96     |

Ta có thể quan sát thấy các hiện tượng như sau:

Với từ  $w_k$  liên quan tới từ “ice (đá)” nhưng không liên quan đến từ “steam (hơi nước)”, như là  $w_k = \text{solid}$  (rắn), ta kỳ vọng là tỷ số xác suất có điều kiện sẽ lớn hơn, như trường hợp này là 8.9 ở hàng cuối cùng của bảng trên. Với từ  $w_k$  liên quan tới từ “steam (hơi nước)” mà không có liên quan nào với từ “ice (đá)”, như là  $w_k = \text{gas}$  (khí), ta kỳ vọng là tỷ số xác suất có điều kiện sẽ nhỏ hơn, như trường hợp này là 0.085 ở hàng cuối cùng của bảng trên. Với từ  $w_k$  liên quan tới cả hai từ “steam (hơi nước)” và từ “ice (đá)”, như là  $w_k = \text{water}$  (nước), ta kỳ vọng là tỷ số xác suất có điều kiện sẽ gần với 1, như trường hợp này là 1.36 ở hàng cuối cùng của bảng trên. Với từ  $w_k$  không liên quan tới cả hai từ “steam (hơi)” và từ “ice (đá)”, như là  $w_k = \text{fashion}$  (thời trang), ta kỳ vọng là tỷ số xác suất có điều kiện sẽ gần với 1, như trường hợp này là 0.96 ở hàng cuối cùng của bảng trên.

Có thể thấy rằng tỷ số xác suất có điều kiện thể hiện mối quan hệ giữa các từ khác nhau trực quan hơn. Ta có thể tạo một hàm vector của từ để khớp tỷ số xác suất có điều kiện một cách hiệu quả hơn. Như đã biết, để thu được bất cứ tỷ số nào loại này đòi hỏi phải có ba từ  $w_i, w_j$ , và  $w_k$ . Tỷ số xác suất có điều kiện với  $w_i$  làm từ trung tâm là  $p_{ij}/p_{ik}$ . Ta có thể tìm một hàm dùng các vector từ để khớp với tỷ số xác suất có điều kiện này.

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) \approx \frac{p_{ij}}{p_{ik}}. \quad (16.5.5)$$

Thiết kế khả dĩ của hàm  $f$  ở đây không phải duy nhất. Ta chỉ cần quan tâm một lựa chọn hợp lý hơn. Do tỷ số xác suất có điều kiện là một số vô hướng, ta có thể giới hạn  $f$  vào một hàm vô hướng:  $f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = f((\mathbf{u}_j - \mathbf{u}_k)^\top \mathbf{v}_i)$ . Sau khi hoán đổi chỉ số  $j$  và  $k$ , ta có thể thấy rằng hàm  $f$  thỏa mãn điều kiện  $f(x)f(-x) = 1$ , do đó một lựa chọn có thể là  $f(x) = \exp(x)$ . Ta có:

$$f(\mathbf{u}_j, \mathbf{u}_k, \mathbf{v}_i) = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\exp(\mathbf{u}_k^\top \mathbf{v}_i)} \approx \frac{p_{ij}}{p_{ik}}. \quad (16.5.6)$$

Một xác suất thỏa mãn về phải biểu thức xấp xỉ là  $\exp(\mathbf{u}_j^\top \mathbf{v}_i) \approx \alpha p_{ij}$ , ở đây  $\alpha$  là một hằng số. Xét  $p_{ij} = x_{ij}/x_i$ , sau khi lấy logarit ta được  $\mathbf{u}_j^\top \mathbf{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$ . Ta sử dụng thêm hệ số điều chỉnh để khớp  $-\log \alpha + \log x_i$ , cụ thể là hệ số điều chỉnh từ trung tâm  $b_i$  và hệ số điều chỉnh từ ngữ cảnh  $c_j$ :

$$\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j \approx \log(x_{ij}). \quad (16.5.7)$$

Bằng cách lấy sai số bình phương và đặt trọng số vào vế trái và vế phải của biểu thức trên, ta tính được hàm mất mát của GloVe.

### 16.5.3 Tóm tắt

- Trong một số trường hợp, hàm mất mát entropy chéo có sự hạn chế. GloVe sử dụng mất mát bình phương và vector từ để khớp các thống kê toàn cục được tính trước dựa trên toàn bộ dữ liệu.
- Vector từ đích trung tâm và vector từ ngữ cảnh của bất kì từ nào là như nhau trong GloVe.

#### 16.5.4 Bài tập

1. Nếu một từ xuất hiện trong cửa sổ ngữ cảnh của từ khác, làm thế nào để sử dụng khoảng cách giữa hai từ này trong chuỗi văn bản để thiết kế lại phương pháp tính toán xác suất có điều kiện  $p_{ij}$ ? Gợi ý: Tham khảo phần 4.2 trong bài báo GloVe (Pennington et al., 2014).
2. Với một từ bất kỳ, liệu hệ số điều chỉnh của từ đích trung tâm và từ ngữ cảnh là như nhau trong GloVe không? Tại sao?

#### 16.5.5 Thảo luận

- Tiếng Anh: Main Forum<sup>339</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>340</sup>

#### 16.5.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long
- Nguyễn Văn Quang
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 29/08/2020)

### 16.6 Embedding từ con

Các từ tiếng Anh thường có những cấu trúc nội tại và phương thức cấu thành. Chẳng hạn, ta có thể suy ra mối quan hệ giữa các từ “dog”, “dogs” và “dogcatcher” thông qua cách viết của chúng. Tất cả các từ đó có cùng từ gốc là “dog” nhưng có hậu tố khác nhau làm thay đổi nghĩa của từ. Hơn nữa, sự liên kết này có thể được mở rộng ra đối với các từ khác. Chẳng hạn, mối quan hệ giữa từ “dog” và “dogs” đơn giản giống như mối quan hệ giữa từ “cat” và “cats”. Mối quan hệ giữa từ “boy” và “boyfriend” đơn giản giống mối quan hệ giữa từ “girl” và “girlfriend”. Đặc tính này không phải là duy nhất trong tiếng Anh. Trong tiếng Pháp và Tây Ban Nha, rất nhiều động từ có thể có hơn 40 dạng khác nhau tùy thuộc vào ngữ cảnh. Trong tiếng Phần Lan, một danh từ có thể có hơn 15 dạng. Thật vậy, hình thái học (*morphology*) là một nhánh quan trọng của ngôn ngữ học chuyên nghiên cứu về cấu trúc và hình thái của các từ.

<sup>339</sup> <https://discuss.d2l.ai/t/385>

<sup>340</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 16.6.1 fastText

Trong word2vec, ta không trực tiếp sử dụng thông tin hình thái học. Trong cả mô hình skip-gram và túi từ (*bag-of-word*) liên tục, ta sử dụng các vector khác nhau để biểu diễn các từ ở các dạng khác nhau. Chẳng hạn, “dog” và “dogs” được biểu diễn bởi hai vector khác nhau, trong khi mối quan hệ giữa hai vector đó không biểu thị trực tiếp trong mô hình. Từ quan điểm này, fastText (Bojanowski et al., 2017) đề xuất phương thức embedding từ con (*subword embedding*), thông qua việc thực hiện đưa thông tin hình thái học vào trong mô hình skip-gram trong word2vec.

Trong fastText, mỗi từ trung tâm được biểu diễn như một tập hợp của các từ con. Dưới đây ta sử dụng từ “where” làm ví dụ để hiểu cách các từ tố được tạo thành. Trước hết, ta thêm một số ký tự đặc biệt “<” và “>” vào phần bắt đầu và kết thúc của từ để phân biệt các từ con được dùng làm tiền tố và hậu tố. Rồi ta sẽ xem từ này như một chuỗi các ký tự để trích xuất *n-grams*. Chẳng hạn, khi  $n = 3$ , ta có thể nhận tất cả từ tố với chiều dài là 3:

$$<\text{wh}>, \text{"whe"}, \text{"her"}, \text{"ere"}, \text{"re"}>, \quad (16.6.1)$$

và từ con đặc biệt “<where>”.

Trong fastText, với một từ  $w$ , ta ghi tập hợp của tất cả các từ con của nó với chiều dài từ 3 đến 6 và các từ con đặc biệt là  $\mathcal{G}_w$ . Do đó, từ điển này là tập hợp các từ con của tất cả các từ. Giả sử vector của từ con  $g$  trong từ điển này là  $\mathbf{z}_g$ . Thì vector từ trung tâm  $\mathbf{u}_w$  cho từ  $w$  trong mô hình skip-gram có thể biểu diễn là

$$\mathbf{u}_w = \sum_{g \in \mathcal{G}_w} \mathbf{z}_g. \quad (16.6.2)$$

Phần còn lại của tiến trình xử lý trong fastText đồng nhất với mô hình skip-gram, vì vậy ta không mô tả lại ở đây. Như chúng ta có thể thấy, so sánh với mô hình skip-gram, từ điển của fastText lớn hơn dẫn tới nhiều tham số mô hình hơn. Hơn nữa, vector của một từ đòi hỏi tính tổng của tất cả vector từ con dẫn tới độ phức tạp tính toán cao hơn. Tuy nhiên, ta có thể thu được các vector tốt hơn cho nhiều từ phức hợp ít thông dụng, thậm chí cho cả các từ không hiện diện trong từ điển này nhờ tham chiếu tới các từ khác có cấu trúc tương tự.

### 16.6.2 Mã hoá cặp byte

Trong fastText, tất cả các từ con được trích xuất phải nằm trong khoảng độ dài cho trước, ví dụ như từ 3 đến 6, do đó kích thước bộ từ vựng không thể được xác định trước. Để cho phép các từ con có độ dài biến thiên trong bộ từ vựng có kích thước cố định, chúng ta có thể áp dụng thuật toán nén gọi là *mã hoá cặp byte* (Byte Pair Encoding -BPE) để trích xuất các từ con (Sennrich et al., 2015).

Mã hóa cặp byte thực hiện phân tích thống kê tập dữ liệu huấn luyện để tìm các ký hiệu chung trong một từ, chẳng hạn như các ký tự liên tiếp có độ dài tùy ý. Bắt đầu từ các ký hiệu có độ dài bằng 1, mã hóa cặp byte lặp đi lặp lại việc gộp các cặp ký hiệu liên tiếp thường gặp nhất để tạo ra các ký hiệu mới dài hơn. Lưu ý rằng để tăng hiệu năng, các cặp vượt qua ranh giới từ sẽ không được xét. Cuối cùng, chúng ta có thể sử dụng các ký hiệu đó như từ con để phân đoạn các từ. Mã hóa cặp byte và các biến thể của nó đã được sử dụng để biểu diễn đầu vào trong các mô hình tiền huấn luyện cho xử lý ngôn ngữ tự nhiên phổ biến như GPT-2 (Radford et al., 2019) và RoBERTa (Liu et al., 2019). Tiếp theo, chúng tôi sẽ minh họa cách hoạt động của mã hóa cặp byte.

Đầu tiên, ta khởi tạo bộ từ vựng của các ký hiệu dưới dạng tất cả các ký tự viết thường trong tiếng Anh và hai ký hiệu đặc biệt: ký hiệu kết thúc của từ ‘\_’, và ký hiệu không xác định ‘[UNK]’.

```

import collections

symbols = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
           'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
           '_', '[UNK]']

```

Vì không xét các cặp ký hiệu vượt qua ranh giới của các từ, chúng ta chỉ cần một từ điển raw\_token\_freqs ánh xạ các từ tới tần suất của chúng (số lần xuất hiện) trong một tập dữ liệu. Lưu ý rằng ký hiệu đặc biệt '\_' được thêm vào mỗi từ để có thể dễ dàng khôi phục chuỗi từ (ví dụ: "a taller man") từ chuỗi ký hiệu đầu ra (ví dụ: "a\_tall\_er\_man"). Vì chúng ta bắt đầu quá trình gộp một từ vựng chỉ gồm các ký tự đơn và các ký hiệu đặc biệt, khoảng trắng được chèn giữa mọi cặp ký tự liên tiếp trong mỗi từ (các khóa của từ điển token\_freqs). Nói cách khác, khoảng trắng là ký tự phân cách (*delimiter*) giữa các ký hiệu trong một từ.

```

raw_token_freqs = {'fast_': 4, 'faster_': 3, 'tall_': 5, 'taller_': 4}
token_freqs = {}
for token, freq in raw_token_freqs.items():
    token_freqs[' '.join(list(token))] = raw_token_freqs[token]
token_freqs

```

Chúng ta định nghĩa hàm get\_max\_freq\_pair trả về cặp ký hiệu liên tiếp thường gặp nhất trong một từ, với từ là các khóa của từ điển đầu vào token\_freqs.

```

def get_max_freq_pair(token_freqs):
    pairs = collections.defaultdict(int)
    for token, freq in token_freqs.items():
        symbols = token.split()
        for i in range(len(symbols) - 1):
            # Key of `pairs` is a tuple of two consecutive symbols
            pairs[symbols[i], symbols[i + 1]] += freq
    return max(pairs, key=pairs.get) # Key of `pairs` with the max value

```

Là một thuật toán tham lam dựa trên tần suất của các ký hiệu liên tiếp nhau, mã hoá cặp byte sẽ dùng hàm merge\_symbols để gộp cặp ký hiệu thường gặp nhất để tạo ra những ký hiệu mới.

```

def merge_symbols(max_freq_pair, token_freqs, symbols):
    symbols.append(' '.join(max_freq_pair))
    new_token_freqs = dict()
    for token, freq in token_freqs.items():
        new_token = token.replace(' '.join(max_freq_pair),
                                  ' '.join(max_freq_pair))
        new_token_freqs[new_token] = token_freqs[token]
    return new_token_freqs

```

Bây giờ ta thực hiện vòng lặp giải thuật biểu diễn cặp byte với các khóa của từ điển token\_freqs. Ở vòng lặp đầu tiên, cặp biểu tượng liền kề có tần suất cao nhất là 't' và 'a', do đó biểu diễn cặp byte ghép chúng lại để tạo ra một biểu tượng mới là 'ta'. Ở vòng lặp thứ hai, biểu diễn cặp byte tiếp tục ghép 2 biểu tượng 'ta' và 'l' tạo ra một biểu tượng mới khác là 'tal'.

```

num_merges = 10
for i in range(num_merges):
    max_freq_pair = get_max_freq_pair(token_freqs)

```

(continues on next page)

```
token_freqs = merge_symbols(max_freq_pair, token_freqs, symbols)
print(f'merge #{i + 1}:', max_freq_pair)
```

Sau 10 vòng lặp biểu diễn cặp byte, ta có thể thấy là danh sách symbols lúc này chứa hơn 10 biểu tượng đã được lần lượt ghép từ các biểu tượng khác.

```
print(symbols)
```

Với cùng tập dữ liệu đặc tả trong các khóa của từ điển raw\_token\_freqs, mỗi từ trong tập dữ liệu này bây giờ được phân đoạn bởi các từ con là “fast\_”, “fast”, “er\_”, “tall\_”, và “tall” theo giải thuật biểu diễn cặp byte. Chẳng hạn, từ “faster\_” và từ “taller\_” được phân đoạn lần lượt là “fast er\_” và “tall er\_”.

```
print(list(token_freqs.keys()))
```

Chú ý là kết quả của biểu diễn cặp byte tùy thuộc vào tập dữ liệu đang được sử dụng. Ta cũng có thể dùng các từ con đã học từ một tập dữ liệu để phân đoạn các từ của một tập dữ liệu khác. Với cách tiếp cận tham lam, hàm segment\_BPE sau đây cố gắng tách các từ thành các từ con dài nhất có thể từ đối số đầu vào symbols.

```
def segment_BPE(tokens, symbols):
    outputs = []
    for token in tokens:
        start, end = 0, len(token)
        cur_output = []
        # Segment token with the longest possible subwords from symbols
        while start < len(token) and start < end:
            if token[start: end] in symbols:
                cur_output.append(token[start: end])
                start = end
                end = len(token)
            else:
                end -= 1
            if start < len(token):
                cur_output.append('[UNK]')
        outputs.append(' '.join(cur_output))
    return outputs
```

Trong phần tiếp theo, ta sử dụng các từ con trong danh sách symbols đã được học từ tập dữ liệu ở trên để phân đoạn các tokens biểu diễn tập dữ liệu khác.

```
tokens = ['tallest_', 'fatter_']
print(segment_BPE(tokens, symbols))
```

### 16.6.3 Tóm tắt

- FastText đề xuất phương pháp embedding cho từ con. Dựa trên mô hình skip-gram trong word2vec, phương pháp này biểu diễn vector từ trung tâm thành tổng các vector từ con của từ đó.
- Embedding cho từ con sử dụng nguyên tắc trong hình thái học, thường giúp cải thiện chất lượng biểu diễn của các từ ít gặp.
- Mã hoá cặp byte thực hiện phân tích thống kê trên tập dữ liệu huấn luyện để phát hiện các ký hiệu chung trong một từ. Là một giải thuật tham lam, mã hoá cặp byte lần lượt gộp các cặp ký hiệu liên tiếp thường gặp nhất lại với nhau.

### 16.6.4 Bài tập

1. Khi có quá nhiều từ con (ví dụ, 6 từ trong tiếng Anh có thể tạo ra  $3 \times 10^8$  các tổ hợp khác nhau), vấn đề gì sẽ xảy ra? Bạn có thể giải quyết vấn đề trên không? Gợi ý: Tham khảo đoạn cuối phần 3.2 của bài báo fastText [1].
2. Làm sao để thiết kế một mô hình embedding cho từ con dựa trên mô hình túi từ liên tục CBOW?
3. Để thu được bộ từ vựng có kích thước  $m$ , bao nhiêu phép gộp cần được thực hiện khi bộ từ vựng ký hiệu ban đầu có kích thước là  $n$ ?
4. Ta có thể mở rộng ý tưởng của thuật toán mã hoá cặp byte để trích xuất các cụm từ bằng cách nào?

### 16.6.5 Thảo luận

- Tiếng Anh: MXNet<sup>341</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>342</sup>

### 16.6.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Mai Hoàng Long
- Phạm Đăng Khoa
- Nguyễn Văn Cường

<sup>341</sup> <https://discuss.d2l.ai/t/386>

<sup>342</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 16.7 Tìm kiếm từ Đồng nghĩa và Loại suy

Trong Section 16.4 ta đã huấn luyện mô hình embedding từ word2vec trên tập dữ liệu cỡ nhỏ và tìm kiếm các từ đồng nghĩa sử dụng độ tương tự cosin giữa các vector từ. Trong thực tế, các vector từ được tiền huấn luyện trên kho ngữ liệu cỡ lớn thường được áp dụng cho các bài toán xử lý ngôn ngữ tự nhiên cụ thể. Phần này sẽ trình bày cách sử dụng các vector từ đã tiền huấn luyện để tìm các từ đồng nghĩa và các loại suy (*analogy*). Ta sẽ tiếp tục áp dụng các vector từ được tiền huấn luyện trong các phần sau.

```
from d2l import mxnet as d2l
from mxnet import np, npx
import os

npx.set_np()
```

### 16.7.1 Sử dụng các Vector Từ đã được Tiền Huấn luyện

Dưới đây là các embedding GloVe đã được tiền huấn luyện với kích thước chiều là 50, 100, và 300, có thể được tải từ [trang web GloVe<sup>343</sup>](#). Các embedding cho fastText được tiền huấn luyện trên nhiều ngôn ngữ. Ở đây, ta quan tâm tới phiên bản cho tiếng Anh (“wiki.en” có chiều là 300) có thể được tải từ [trang web fastText<sup>344</sup>](#).

```
#@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')
```

Ta định nghĩa lớp TokenEmbedding để nạp các embedding GloVe và fastText ở trên.

```
#@save
class TokenEmbedding:
    """Token Embedding."""
    def __init__(self, embedding_name):
        self.idx_to_token, self.idx_to_vec = self._load_embedding(
            embedding_name)
```

(continues on next page)

<sup>343</sup> <https://nlp.stanford.edu/projects/glove/>

<sup>344</sup> <https://fasttext.cc/>

```

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):
    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, np.array(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[np.array(indices)]
    return vecs

def __len__(self):
    return len(self.idx_to_token)

```

Tiếp theo, ta sử dụng embedding GloVe có chiều là 50 được tiền huấn luyện trên tập con của Wikipedia. Embedding tương ứng của từ sẽ được tự động tải về khi tạo một thực thể TokenEmbedding lần đầu.

```
glove_6b50d = TokenEmbedding('glove.6b.50d')
```

Ta có thể in ra kích thước từ điển. Từ điển chứa 400,000 từ và một token đặc biệt cho các từ không biết.

```
len(glove_6b50d)
```

Ta có thể lấy chỉ số của một từ trong từ điển, hoặc ngược lại tra từ tương ứng với chỉ số cho trước.

```
glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]
```

## 16.7.2 Áp dụng các Vector Từ đã được Tiền huấn luyện

Dưới đây, ta minh họa việc áp dụng các vector từ đã được tiền huấn luyện sử dụng Glove làm ví dụ.

### Tìm các từ đồng nghĩa

Tại đây, ta lập trình lại thuật toán tìm các từ đồng nghĩa bằng độ tương tự cosin giữa hai vector trong [Section 16.1](#).

Để sử dụng lại logic tìm kiếm  $k$  láng giềng gần nhất ( $k$ -nearest neighbors) khi tìm kiếm các từ loại suy, ta đóng gói phần này một cách tách biệt trong hàm knn .

```
def knn(W, x, k):
    # The added 1e-9 is for numerical stability
    cos = np.dot(W, x.reshape(-1,)) / (
        np.sqrt(np.sum(W * W, axis=1) + 1e-9) * np.sqrt((x * x).sum()))
    topk = npx.topk(cos, k=k, ret_type='indices')
    return topk, [cos[int(i)] for i in topk]
```

Kế tiếp, ta tìm kiếm các từ đồng nghĩa nhờ tiền huấn luyện thực thể vector từ 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:]): # Remove input words
        print(f'cosine sim={float(c):.3f}: {embed.idx_to_token[int(i)]}' )
```

Từ điển vector từ được tiền huấn luyện glove\_6b50d đã tạo chứa 400,000 từ và một token các từ không biết. Loại trừ những từ đầu vào và những từ không biết, ta tìm kiếm ba từ có nghĩa gần với từ “chip”.

```
get_similar_tokens('chip', 3, glove_6b50d)
```

Kế tiếp, ta tìm các từ gần nghĩa với “baby” và “beautiful”.

```
get_similar_tokens('baby', 3, glove_6b50d)
```

```
get_similar_tokens('beautiful', 3, glove_6b50d)
```

### Tìm kiếm các Loại suy

Bên cạnh việc tìm kiếm các từ đồng nghĩa, ta cũng có thể sử dụng các vector từ đã tiền huấn luyện để tìm kiếm các loại suy giữa các từ. Ví dụ, “man”:“woman”::“son”:“daughter” là một loại suy, “man (nam)” với “woman (nữ)” giống như “son (con trai)” với “daughter (con gái)”. Bài toán tìm kiếm loại suy có thể được định nghĩa như sau: với bốn từ trong quan hệ loại suy  $a : b :: c : d$ , cho trước ba từ  $a$ ,  $b$  và  $c$ , ta muốn tìm từ  $d$ . Giả sử, vector từ cho từ  $w$  là  $\text{vec}(w)$ . Để giải quyết bài toán loại suy, ta cần tìm vector từ gần nhất với vector là 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]]
```

(continues on next page)

```
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
```

Kiểm tra quan hệ loại suy “nam giới - nữ giới”.

```
get_analogy('man', 'woman', 'son', glove_6b50d)
```

Loại suy “thủ đô-quốc gia”: từ “beijing” với từ “china” tương tự như từ “tokyo” với từ nào? Đáp án là “japan”.

```
get_analogy('beijing', 'china', 'tokyo', glove_6b50d)
```

Loại suy “tính từ - tính từ so sánh nhất”: từ “bad” với từ “worst” tương tự như từ “big” với từ nào? Đáp án là “biggest”.

```
get_analogy('bad', 'worst', 'big', glove_6b50d)
```

Loại suy “động từ thì hiện tại - động từ thì quá khứ”: từ “do” với từ “did” tương tự như từ “go” với từ nào? Đáp án là “went”.

```
get_analogy('do', 'did', 'go', glove_6b50d)
```

### 16.7.3 Tóm tắt

- Các vector từ được tiền huấn luyện trên kho ngữ liệu cỡ lớn thường được áp dụng cho các tác vụ xử lý ngôn ngữ tự nhiên.
- Ta có thể sử dụng các vector từ được tiền huấn luyện để tìm kiếm các từ đồng nghĩa và các loại suy.

### 16.7.4 Bài tập

1. Hãy kiểm tra kết quả với fastText bằng cách sử dụng TokenEmbedding('wiki.en').
2. Nếu từ điển quá lớn, ta có thể tăng tốc tìm kiếm các từ đồng nghĩa và các loại suy bằng cách nào?

### 16.7.5 Thảo luận

- Tiếng Anh: MXNet<sup>345</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>346</sup>

<sup>345</sup> <https://discuss.d2l.ai/t/387>

<sup>346</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 16.7.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 01/07/2020)

## 16.8 Biểu diễn Mã hóa hai chiều từ Transformer (BERT)

Chúng tôi đã giới thiệu một vài mô hình embedding từ cho bài toán hiểu ngôn ngữ tự nhiên. Sau khi tiền huấn luyện, đầu ra của các mô hình này có thể xem là một ma trận trong đó mỗi hàng là một vector biểu diễn một từ trong bộ từ vựng được định nghĩa trước. Trong thực tế, tất cả các mô hình embedding từ này đều *độc lập ngữ cảnh* (*context-independent*). Hãy bắt đầu bằng việc minh họa tính chất này.

### 16.8.1 Từ Độc lập Ngữ cảnh đến Nhạy Ngữ cảnh

Hãy nhớ lại các thí nghiệm trong [Section 16.4](#) và [Section 16.7](#). Cả word2vec và GloVe đều gán cùng một vector được tiền huấn luyện cho cùng một từ bất kể ngữ cảnh (nếu có) của nó như thế nào. Về mặt hình thức, biểu diễn độc lập ngữ cảnh của một token bất kỳ  $x$  là một hàm  $f(x)$  chỉ nhận  $x$  làm đầu vào. Do hiện tượng đa nghĩa cũng như sự phức tạp ngữ nghĩa xuất hiện khá phổ biến trong ngôn ngữ tự nhiên, biểu diễn độc lập ngữ cảnh có những hạn chế rõ ràng. Ví dụ, từ “crane” trong ngữ cảnh “a crane is flying (một con sếu đang bay)” và ngữ cảnh “a crane driver came (tài xế xe cẩu đã tới)” có nghĩa hoàn toàn khác nhau; do đó, cùng một từ nên được gán các biểu diễn khác nhau tùy ngữ cảnh.

Điều này thúc đẩy sự phát triển của các biểu diễn từ *nhạy ngữ cảnh* (*context-sensitive*), trong đó biểu diễn của từ phụ thuộc vào ngữ cảnh của từ đó. Do đó, biểu diễn nhạy ngữ cảnh của một token bất kỳ  $x$  là hàm  $f(x, c(x))$  phụ thuộc vào cả từ  $x$  lẫn ngữ cảnh của từ  $c(x)$ . Các biểu diễn nhạy ngữ cảnh phổ biến bao gồm TagLM (Bộ Tag chuỗi được tăng cường với mô hình ngôn ngữ (*language-model-augmented sequence tagger*)) ([Peters et al., 2017b](#)), CoVe (vector ngữ cảnh (*Context Vectors*)) ([McCann et al., 2017](#)), và ELMo (embedding từ các mô hình ngôn ngữ (*Embeddings from Language Models*)) ([Peters et al., 2018](#)).

Ví dụ, bằng cách lấy toàn bộ chuỗi làm đầu vào, ELMo gán một biểu diễn cho mỗi từ trong chuỗi đầu vào. Cụ thể, ELMo kết hợp tất cả các biểu diễn tầng trung gian từ LSTM hai chiều đã được tiền huấn luyện làm biểu diễn đầu ra. Sau đó, biểu diễn ELMo sẽ được đưa vào một mô hình học có giám sát cho các tác vụ xuôi dòng như một đặc trưng bổ sung, chẳng hạn bằng cách nối biểu diễn ELMo và biểu diễn gốc (ví dụ như GloVe) của token trong mô hình hiện tại. Một mặt, tất cả các trọng số trong mô hình LSTM hai chiều được tiền huấn luyện đều bị đóng băng sau khi các biểu diễn ELMo được thêm vào. Mặt khác, mô hình học có giám sát được tùy biến cụ thể cho một tác vụ nhất định. Tại thời điểm được công bố, thêm ELMo vào các mô hình tiên tiến nhất giúp

cải thiện chất lượng các mô hình này trên sáu tác vụ xử lý ngôn ngữ tự nhiên: phân tích cảm xúc (*sentiment analysis*), suy luận ngôn ngữ tự nhiên (*natural language inference*), gán nhãn vai trò ngữ nghĩa (*semantic role labeling*), phân giải đồng tham chiếu (*coreference resolution*), nhận dạng thực thể có tên (*named entity recognition*) và trả lời câu hỏi (*question answering*).

### 16.8.2 Từ Đặc thù Tác vụ đến Không phân biệt Tác vụ

Mặc dù ELMo đã cải thiện đáng kể giải pháp cho một loạt các tác vụ xử lý ngôn ngữ tự nhiên, mỗi giải pháp vẫn dựa trên một kiến trúc *đặc thù cho tác vụ* (*task-specific*). Tuy nhiên trong thực tế, xây dựng một kiến trúc đặc thù cho mỗi tác vụ xử lý ngôn ngữ tự nhiên là điều không đơn giản. Phương pháp GPT (Generative Pre-Training) thể hiện nỗ lực thiết kế một mô hình *không phân biệt tác vụ* (*task-agnostic*) chung cho các biểu diễn nhạy ngôn ngữ cảnh (Radford et al., 2018). Được xây dựng dựa trên bộ giải mã Transformer, GPT tiền huấn luyện mô hình ngôn ngữ được sử dụng để biểu diễn chuỗi văn bản. Khi áp dụng GPT cho một tác vụ xuôi dòng, đầu ra của mô hình ngôn ngữ sẽ được truyền tới một tầng đầu ra tuyến tính được bổ sung để dự đoán nhãn cho tác vụ đó. Trái ngược hoàn toàn với cách ELMo đóng băng các tham số của mô hình tiền huấn luyện, GPT tinh chỉnh *tất cả* các tham số trong bộ giải mã Transformer tiền huấn luyện trong suốt quá trình học có giám sát trên tác vụ xuôi dòng. GPT được đánh giá trên mười hai tác vụ về suy luận ngôn ngữ tự nhiên, trả lời câu hỏi, độ tương tự của câu, và bài toán phân loại, và cải thiện kết quả tân tiến nhất của chín tác vụ với vài thay đổi tối thiểu trong kiến trúc mô hình.

Tuy nhiên, do tính chất tự hồi quy của các mô hình ngôn ngữ, GPT chỉ nhìn theo chiều xuôi (từ trái sang phải). Trong các ngữ cảnh “I went to the bank to deposit cash” (“tôi đến ngân hàng để gửi tiền”) và “I went to the bank to sit down” (“tôi ra bờ hồ ngồi”), do từ “bank” nhạy với ngữ cảnh bên trái, GPT sẽ trả về cùng một biểu diễn cho từ “bank”, mặc dù nó có nghĩa khác nhau.

### 16.8.3 BERT: Kết hợp những Điều Tốt nhất của Hai Phương pháp

Như ta đã thấy, ELMo mã hóa ngữ cảnh hai chiều nhưng sử dụng các kiến trúc đặc thù cho từng tác vụ; trong khi đó GPT có kiến trúc không phân biệt tác vụ nhưng mã hóa ngữ cảnh từ trái sang phải. Kết hợp những điều tốt nhất của hai phương pháp trên, BERT (biểu diễn mã hóa hai chiều từ Transformer - *Bidirectional Encoder Representations from Transformers*) mã hóa ngữ cảnh theo hai chiều và chỉ yêu cầu vài thay đổi kiến trúc tối thiểu cho một loạt các tác vụ xử lý ngôn ngữ tự nhiên (Devlin et al., 2018). Sử dụng bộ mã hóa Transformer được tiền huấn luyện, BERT có thể biểu diễn bất kỳ token nào dựa trên ngữ cảnh hai chiều của nó. Trong quá trình học có giám sát trên các tác vụ xuôi dòng, BERT tương tự như GPT ở hai khía cạnh. Đầu tiên, các biểu diễn BERT sẽ được truyền vào một tầng đầu ra được bổ sung, với những thay đổi tối thiểu tới kiến trúc mô hình tùy thuộc vào bản chất của tác vụ, chẳng hạn như dự đoán cho mỗi token hay dự đoán cho toàn bộ chuỗi. Thứ hai, tất cả các tham số của bộ mã hóa Transformer đã tiền huấn luyện đều được tinh chỉnh, trong khi tầng đầu ra bổ sung sẽ được huấn luyện từ đầu. Fig. 16.8.1 mô tả những điểm khác biệt giữa ELMo, GPT, và BERT.

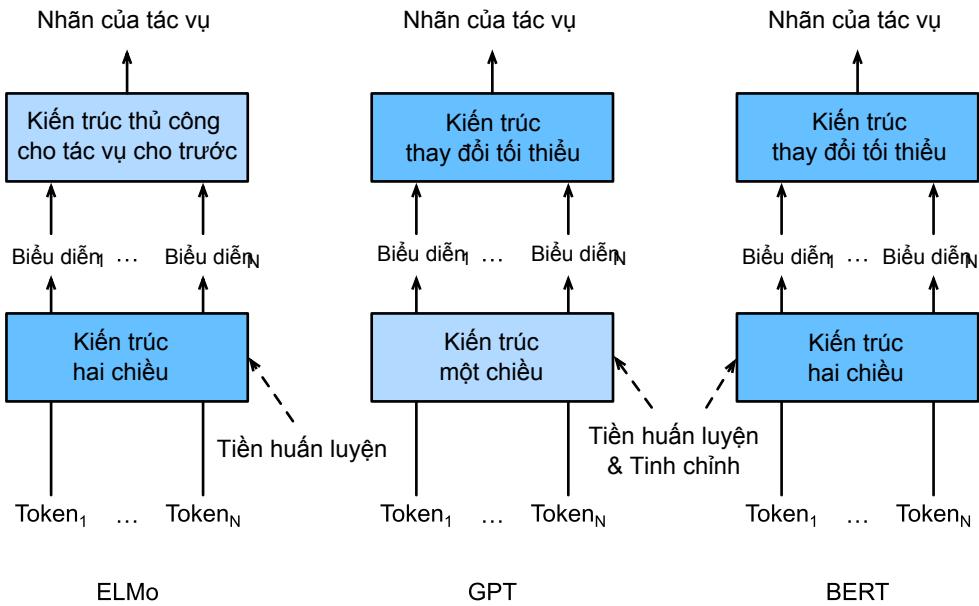


Fig. 16.8.1: So sánh giữa ELMO, GPT, và BERT.

BERT cải thiện kết quả tân tiến nhất đối với mười một tác vụ xử lý ngôn ngữ tự nhiên trải khắp các hạng mục gồm: i) phân loại văn bản đơn (như phân tích cảm xúc), ii) phân loại cặp văn bản (như suy luận ngôn ngữ tự nhiên), iii) trả lời câu hỏi, và iv) gán thẻ văn bản (như nhận dạng thực thể có tên). Tất cả các kỹ thuật được đề xuất trong năm 2018, từ ELMo nhạy ngôn ngữ cảnh cho tới GPT không phân biệt tác vụ và BERT, tuy về ý tưởng đều đơn giản nhưng trên thực nghiệm là những phương pháp tiền huấn luyện hiệu quả cho các biểu diễn sâu của ngôn ngữ tự nhiên, và đã mang đến những giải pháp mang tính cách mạng cho nhiều tác vụ xử lý ngôn ngữ tự nhiên.

Ở phần còn lại của chương này, ta sẽ đi sâu vào tiền huấn luyện BERT. Sau khi những ứng dụng xử lý ngôn ngữ tự nhiên đã được giải thích trong Section 17, ta sẽ minh họa việc tinh chỉnh BERT cho các ứng dụng xuôi dòng.

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn

npx.set_np()
```

#### 16.8.4 Biểu diễn Đầu vào

Trong xử lý ngôn ngữ tự nhiên, một số nhiệm vụ (như phân tích cảm xúc) lấy một câu văn làm đầu vào, trong khi một số tác vụ khác (như suy diễn ngôn ngữ tự nhiên), đầu vào là một cặp chuỗi văn bản. Chuỗi đầu vào BERT biểu diễn một cách tường minh cả văn bản đơn và cặp văn bản. Với văn bản đơn, chuỗi đầu vào BERT là sự ghép nối của token phân loại đặc biệt “<cls>”, token của chuỗi văn bản, và token phân tách đặc biệt “<sep>”. Với cặp văn bản, chuỗi đầu vào BERT là sự ghép nối của “<cls>”, token của chuỗi văn bản đầu, “<sep>”, token của chuỗi văn bản thứ hai, và “<sep>”. Ta sẽ phân biệt nhất quán thuật ngữ “chuỗi đầu vào BERT” với các kiểu “chuỗi” khác. Chẳng hạn, một chuỗi đầu vào BERT có thể bao gồm cả *một chuỗi văn bản* hoặc *hai chuỗi văn bản*.

Để phân biệt cặp văn bản, các embedding đoạn đã học  $\mathbf{e}_A$  và  $\mathbf{e}_B$  được cộng tương ứng vào các

embedding token của chuỗi thứ nhất và chuỗi thứ hai. Đối với đầu vào là văn bản đơn, ta chỉ sử dụng  $\mathbf{e}_A$ .

Hàm `get_tokens_and_segments` sau đây có thể lấy một hoặc hai câu làm đầu vào, rồi trả về các token của chuỗi đầu vào BERT và các ID đoạn tương ứng của chúng.

```
#@save
def get_tokens_and_segments(tokens_a, tokens_b=None):
    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
```

Kiến trúc hai chiều của BERT là bộ mã hóa Transformer. Thông thường trong bộ mã hóa Transformer, các embedding vị trí được cộng vào mỗi vị trí của chuỗi đầu vào BERT. Tuy nhiên, khác với bộ mã hóa Transformer nguyên bản, BERT sử dụng các embedding vị trí *có thể học được*. Fig. 16.8.2 cho thấy các embedding của chuỗi đầu vào BERT là tổng các embedding của token, embedding đoạn và embedding vị trí.

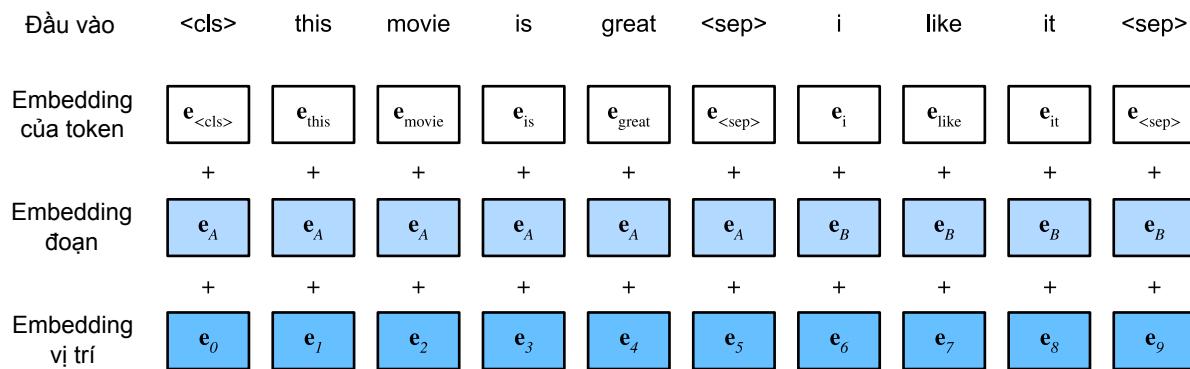


Fig. 16.8.2: Embedding của chuỗi đầu vào BERT là tổng các embedding của token, embedding đoạn và embedding vị trí.

Lớp `BERTEncoder` dưới đây tương tự như lớp `TransformerEncoder` trong Section 12.3. Khác với `TransformerEncoder`, `BERTEncoder` sử dụng các embedding đoạn và các embedding vị trí có thể học được.

```
#@save
class BERTEncoder(nn.Block):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_layers, dropout, max_len=1000, **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 _ in range(num_layers):
            self.blks.add(d2l.EncoderBlock(
                num_hiddens, ffn_num_hiddens, num_heads, dropout, True))
        # In BERT, positional embeddings are learnable, thus we create a
```

(continues on next page)

```
# parameter of positional embeddings that are long enough
self.pos_embedding = self.params.get('pos_embedding',
                                      shape=(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(ctx=X.ctx)[:, :X.shape[1], :]
    for blk in self.blks:
        X = blk(X, valid_lens)
    return X
```

Giả sử kích thước bộ từ vựng là 10,000. Để minh họa suy luận xuôi của BERTEncoder, hãy tạo ra một thực thể của nó và khởi tạo các thông số.

```
vocab_size, num_hiddens, ffn_num_hiddens, num_heads = 10000, 768, 1024, 4
num_layers, dropout = 2, 0.2
encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                      num_layers, dropout)
encoder.initialize()
```

Ta định nghĩa tokens là hai chuỗi đầu vào BERT có độ dài là 8, mỗi token là một chỉ mục của bộ từ vựng. Lượt suy luận xuôi của BERTEncoder với đầu vào tokens trả về kết quả được mã hóa, với mỗi token được biểu diễn bởi một vector có chiều dài được định nghĩa trước bởi siêu tham số num\_hiddens, là *kích thước ẩn* (số lượng nút ẩn) của bộ mã hóa Transformer.

```
tokens = np.random.randint(0, vocab_size, (2, 8))
segments = np.array([[0, 0, 0, 1, 1, 1, 1], [0, 0, 0, 1, 1, 1, 1]])
encoded_X = encoder(tokens, segments, None)
encoded_X.shape
```

### 16.8.5 Những tác vụ Tiền huấn luyện

Suy luận xuôi của BERTEncoder cho ra biểu diễn BERT của mỗi token của văn bản đầu vào và các token đặc biệt được thêm vào “<cls>” và “<seq>”. Kế tiếp, ta sẽ sử dụng các biểu diễn này để tính toán hàm mất mát khi tiền huấn luyện BERT. Tiền huấn luyện gồm hai tác vụ: mô hình ngôn ngữ có mặt nạ (*masked language modeling*) và dự đoán câu tiếp theo.

#### Mô hình Ngôn ngữ có Mặt nạ

Như mô tả trong Section 10.3, một mô hình ngôn ngữ dự đoán một token bằng cách sử dụng ngữ cảnh phía bên trái của nó. Để mã hóa ngữ cảnh hai chiều khi biểu diễn mỗi token, BERT ngẫu nhiên che mặt nạ các token và sử dụng các token lấy từ ngữ cảnh hai chiều để dự đoán các token mặt nạ đó. Tác vụ này được gọi là *mô hình hóa ngôn ngữ có mặt nạ*.

Trong tác vụ tiền huấn luyện này, 15% số token sẽ được lựa chọn ngẫu nhiên để làm các token mặt nạ cho việc dự đoán. Để dự đoán một token mặt nạ mà không sử dụng nhãn, một hướng tiếp cận đơn giản là luôn luôn thay thế nó bằng token đặc biệt “<mask>” trong chuỗi đầu vào BERT. Tuy nhiên, token “<mask>” sẽ không bao giờ xuất hiện khi tinh chỉnh. Để tránh sự không đồng nhất

giữa tiền huấn luyện và tinh chỉnh, nếu một token được che mặt nạ để dự đoán (ví dụ, từ “great” được chọn để che mặt nạ và dự đoán trong câu “this movie is great”), trong đầu vào nó sẽ được thay thế bởi:

- token đặc biệt “<mask>”, 80% số lần (ví dụ, “this movie is great” trở thành “this movie is <mask>”);
- token ngẫu nhiên, 10% số lần (ví dụ, “this movie is great” trở thành “this movie is drink”);
- chính token đó, 10% số lần (ví dụ, “this movie is great” trở thành “this movie is great”).

Lưu ý rằng trong 15% token được chọn để che mặt nạ, 10% số token đó sẽ được thay thế bằng một token ngẫu nhiên. Việc thi thoảng thêm nhiễu sẽ giúp BERT giảm thiên kiến về phía token có mặt nạ (đặc biệt khi token nhẵn không đổi) khi mã hóa ngữ cảnh hai chiều.

Ta lập trình lớp MaskLM sau để dự đoán token có mặt nạ trong tác vụ mô hình hóa ngôn ngữ có mặt nạ khi tiền huấn luyện BERT. MLP một-tầng-ẩn (self.mlp) được dùng cho việc dự đoán. Lượt suy luận xuôi nhận hai đầu vào: kết quả mã hóa của BERTEncoder và vị trí token để dự đoán. Đầu ra là kết quả dự đoán tại các vị trí này.

```
#@save
class MaskLM(nn.Block):
    def __init__(self, vocab_size, num_hiddens, **kwargs):
        super(MaskLM, self).__init__(**kwargs)
        self.mlp = nn.Sequential()
        self.mlp.add(
            nn.Dense(num_hiddens, flatten=False, activation='relu'))
        self.mlp.add(nn.LayerNorm())
        self.mlp.add(nn.Dense(vocab_size, flatten=False))

    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 = np.arange(0, batch_size)
        # Suppose that `batch_size` = 2, `num_pred_positions` = 3, then
        # `batch_idx` is `np.array([0, 0, 1, 1, 1])`
        batch_idx = np.repeat(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 minh họa lượt suy luận xuôi của MaskLM, ta sẽ khởi tạo một thực thể m<sub>lm</sub>. Hãy nhớ lại rằng encoded\_X từ lượt suy luận xuôi của BERTEncoder biểu diễn 2 chuỗi đầu vào BERT. Ta định nghĩa m<sub>lm</sub>\_positions là 3 chỉ số để dự đoán ở một trong hai chuỗi đầu vào BERT của encoded\_X. Lượt suy luận xuôi của m<sub>lm</sub> trả về kết quả dự đoán m<sub>lm</sub>\_Y\_hat tại tất cả các vị trí mặt nạ m<sub>lm</sub>\_positions của encoded\_X. Với mỗi dự đoán, kích thước của kết quả bằng với kích thước bộ từ vựng.

```
mlm = MaskLM(vocab_size, num_hiddens)
mlm.initialize()
mlm_positions = np.array([[1, 5, 2], [6, 1, 5]])
mlm_Y_hat = mlm(encoded_X, mlm_positions)
mlm_Y_hat.shape
```

Với nhãn gốc m<sub>lm</sub>\_Y của token có mặt nạ được dự đoán m<sub>lm</sub>\_Y\_hat, ta có thể tính mất mát entropy

chéo của tác vụ mô hình hóa ngôn ngữ có mặt nạ trong quá trình tiền huấn luyện BERT.

```
mlm_Y = np.array([[7, 8, 9], [10, 20, 30]])
loss = gluon.loss.SoftmaxCrossEntropyLoss()
mlm_l = loss(mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y.reshape(-1))
mlm_l.shape
```

## Dự đoán Câu tiếp theo

Mặc dù mô hình hóa ngôn ngữ có mặt nạ có thể mã hóa ngữ cảnh hai chiều để biểu diễn từ ngữ, nó không thể mô hình hóa các mối quan hệ logic giữa các cặp văn bản một cách tường minh. Để hiểu hơn về mối quan hệ giữa hai chuỗi văn bản, BERT sử dụng tác vụ phân loại nhị phân, *dự đoán câu tiếp theo* (*next sentence prediction*) trong quá trình tiền huấn luyện. Khi sinh các cặp câu cho quá trình tiền huấn luyện, một nửa trong số đó là các cặp câu liên tiếp nhau trong thực tế và được gán nhãn “Đúng” (*True*); và trong nửa còn lại, câu thứ hai được lấy mẫu ngẫu nhiên từ kho ngữ liệu và cặp này được gán nhãn “Sai” (*False*).

Lớp `NextSentencePred` dưới đây sử dụng MLP một tầng ẩn để dự đoán câu thứ hai có phải là câu kế tiếp của câu thứ nhất trong chuỗi đầu vào BERT hay không. Do cơ chế tự tập trung trong bộ mã hóa Transformer, biểu diễn BERT của token đặc biệt “`<cls>`” mã hóa cả hai câu đầu vào. Vì vậy, tầng đầu ra (`self.output`) của bộ phân loại MLP nhận đầu vào `X` là đầu ra của tầng ẩn MLP có đầu vào là token được mã hóa “`<cls>`”.

```
#@save
class NextSentencePred(nn.Block):
    def __init__(self, **kwargs):
        super(NextSentencePred, self).__init__(**kwargs)
        self.output = nn.Dense(2)

    def forward(self, X):
        # `X` shape: (batch size, `num_hiddens`)
        return self.output(X)
```

Ta có thể thấy lượt suy luận xuôi của thực thể `NextSentencePred` trả về dự đoán nhị phân cho mỗi chuỗi đầu vào BERT.

```
nsp = NextSentencePred()
nsp.initialize()
nsp_Y_hat = nsp(encoded_X)
nsp_Y_hat.shape
```

Mất mát entropy chéo của 2 tác vụ phân loại nhị phân có thể được tính như sau.

```
nsp_y = np.array([0, 1])
nsp_l = loss(nsp_Y_hat, nsp_y)
nsp_l.shape
```

Đáng chú ý là tất cả nhãn trong hai tác vụ tiền huấn luyện nói trên đều có thể thu được từ kho ngữ liệu tiền huấn luyện mà không cần công sức gán nhãn thủ công. Phiên bản gốc của BERT được tiền huấn luyện trên cả hai kho ngữ liệu BookCorpus (Zhu et al., 2015) và Wikipedia tiếng Anh. Hai kho ngữ liệu văn bản này cực kỳ lớn, chứa lần lượt khoảng 800 triệu từ và 2.5 tỉ từ.

### 16.8.6 Kết hợp Tất cả lại

Khi tiền huấn luyện BERT, hàm măt măt cuối cùng là tổ hợp tuyến tính của cả hai hàm măt măt trong tác vụ mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo. Bây giờ ta có thể định nghĩa lớp BERTModel bằng cách khởi tạo ba lớp BERTEncoder, MaskLM, và NextSentencePred. Lượt suy luận xuôi trả về biểu diễn BERT được mã hóa encoded\_X, các dự đoán mlm\_Y\_hat của tác vụ mô hình hóa ngôn ngữ có mặt nạ, và nsp\_Y\_hat của tác vụ dự đoán câu tiếp theo.

```
#@save
class BERTModel(nn.Block):
    def __init__(self, vocab_size, num_hiddens, ffn_num_hiddens, num_heads,
                 num_layers, dropout, max_len=1000):
        super(BERTModel, self).__init__()
        self.encoder = BERTEncoder(vocab_size, num_hiddens, ffn_num_hiddens,
                                   num_heads, num_layers, dropout, max_len)
        self.hidden = nn.Dense(num_hiddens, activation='tanh')
        self.mlm = MaskLM(vocab_size, num_hiddens)
        self.nsp = NextSentencePred()

    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
```

### 16.8.7 Tóm tắt

- Các mô hình embedding từ như word2vec và GloVe có tính chất độc lập với ngữ cảnh. Hai mô hình này gán cùng một vector được tiền huấn luyện cho cùng một từ bất kể ngữ cảnh xung quanh của từ đó là gì (nếu có). Do đó, rất khó để các mô hình này xử lý tốt các trường hợp phức tạp về ngữ nghĩa hay đa nghĩa trong các ngôn ngữ tự nhiên.
- Đối với các biểu diễn từ nhạy ngữ cảnh như ELMo và GPT, biểu diễn của từ phụ thuộc vào ngữ cảnh của từ đó.
- ELMo mã hóa ngữ cảnh theo hai chiều nhưng sử dụng kiến trúc đặc thù cho tác vụ (tuy nhiên, trên thực tế không dễ để tạo ra một kiến trúc đặc thù cho mọi tác vụ xử lý ngôn ngữ tự nhiên); trong khi đó GPT không phân biệt tác vụ nhưng chỉ mã hóa ngữ cảnh theo chiều từ trái sang phải.
- BERT kết hợp những gì tốt nhất của cả hai mô hình trên: mã hóa ngữ cảnh theo hai chiều và chỉ yêu cầu những thay đổi kiến trúc tối thiểu cho một loạt các tác vụ xử lý ngôn ngữ tự nhiên.
- Các embedding của chuỗi đầu vào BERT là tổng các embedding cho token, embedding đoạn và embedding vị trí.
- Quá trình tiền huấn luyện BERT gồm có hai tác vụ: tác vụ mô hình hóa ngôn ngữ có mặt nạ và tác vụ dự đoán câu tiếp theo. Tác vụ đầu có thể mã hóa ngữ cảnh hai chiều để biểu diễn

từ, trong khi tác vụ sau mô hình hóa mối quan hệ logic giữa các cặp văn bản một cách tường minh.

### 16.8.8 Bài tập

1. Tại sao BERT lại gặt hái được thành công?
2. Giữ nguyên các yếu tố khác, liệu một mô hình ngôn ngữ có mặt nạ sẽ đòi hỏi số bước tiền huấn luyện nhiều hơn hay ít hơn để hội tụ so với mô hình ngôn ngữ từ trái sang phải. Tại sao?
3. Trong mã nguồn gốc của BERT, mạng truyền xuôi theo vị trí (*position-wise feed-forward network*) trong BERTEncoder (qua d2l.EncoderBlock) và tầng kết nối đầy đủ trong MaskLM đều sử dụng Đơn vị lỗi tuyến tính Gauss (*Gaussian error linear unit (GELU)*) (Hendrycks & Gimpel, 2016) làm hàm kích hoạt. Hãy nghiên cứu sự khác biệt giữa GELU và ReLU.

### 16.8.9 Thảo luận

- Tiếng Anh: MXNet<sup>347</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>348</sup>

### 16.8.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 01/07/2020)

<sup>347</sup> <https://discuss.d2l.ai/t/388>

<sup>348</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 16.9 Tập dữ liệu để Tiền huấn luyện BERT

Để tiền huấn luyện mô hình BERT như thực hiện trong Section 16.8, ta cần sinh tập dữ liệu ở định dạng lý tưởng để thuận tiện cho hai tác vụ tiền huấn luyện: mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo. Một mặt, mô hình BERT gốc được tiền huấn luyện trên kho ngữ liệu được ghép lại từ hai kho ngữ liệu khổng lồ là BookCorpus và Wikipedia Tiếng Anh (xem Section 16.8.5), khiến việc thực hành trở nên khó khăn đối với hầu hết độc giả của cuốn sách này. Mặt khác, mô hình BERT đã được tiền huấn luyện sẵn có thể không phù hợp với các ứng dụng ở một số lĩnh vực cụ thể như ngành dược. Do đó, việc tiền huấn luyện BERT trên một tập dữ liệu tùy chỉnh đang ngày càng trở nên phổ biến hơn. Để thuận tiện minh họa cho tiền huấn luyện BERT, ta sử dụng một kho ngữ liệu nhỏ hơn là WikiText-2 (Merity et al., 2016).

So với tập dữ liệu PTB đã dùng để thực hiện tiền huấn luyện word2vec ở Section 16.3, WikiText-2 đã i) giữ lại dấu ngắt câu ban đầu, giúp nó phù hợp cho việc dự đoán câu kế tiếp; ii) giữ lại ký tự viết hoa và số; iii) và lớn hơn gấp hai lần.

```
import collections
from d2l import mxnet as d2l
import mxnet as mx
from mxnet import autograd, gluon, init, np, npx
import os
import random
import time
import zipfile

npx.set_np()
```

Trong tập dữ liệu WikiText-2, mỗi dòng biểu diễn một đoạn văn. Dấu cách được chèn vào giữa bất cứ dấu ngắt câu nào và token đứng trước nó. Các đoạn văn có tối thiểu hai câu được giữ lại. Để tách các câu, ta chỉ sử dụng dấu chấm làm dấu phân cách cho đơn giản. Ta sẽ dành việc thảo luận về các kỹ thuật tách câu phức tạp hơn ở phần bài tập cuối mục.

```
#@save
d2l.DATA_HUB['wikitext-2'] = (
    'https://s3.amazonaws.com/research.metamind.io/wikitext/'
    'wikitext-2-v1.zip', '3c914d17d80b1459be871a5039ac23e752a53cbe')

#@save
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
```

### 16.9.1 Định nghĩa các Hàm trợ giúp cho các Tác vụ Tiền huấn luyện

Ở phần này, ta sẽ bắt đầu lập trình các hàm hỗ trợ cho các hai tác vụ tiền huấn luyện BERT: dự đoán câu tiếp theo và mô hình hóa ngôn ngữ có mặt nạ. Các hàm hỗ trợ này sẽ được gọi khi thực hiện chuyển đổi các kho ngữ liệu văn bản thô sang tập dữ liệu có định dạng lý tưởng để tiền huấn luyện BERT.

#### Sinh tác vụ Dự đoán câu tiếp theo

Dựa theo mô tả của Section 16.8.5, hàm `_get_next_sentence` sinh một mẫu để huấn luyện cho tác 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 đây sinh các mẫu huấn luyện cho tác vụ dự đoán câu tiếp theo từ đầu vào paragraph thông qua hàm `_get_next_sentence`. paragraph ở đây là một danh sách các câu mà mỗi câu là một danh sách các token. Độ số `max_len` là chiều dài cực đại của chuỗi đầu vào BERT trong suốt quá trình tiền huấn luyện.

```
#@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 có Mặt nạ

Để tạo dữ liệu huấn luyện cho tác vụ mô hình hóa ngôn ngữ có mặt nạ từ một chuỗi đầu vào BERT, chúng ta cần định nghĩa hàm `_replace_mlm_tokens`. Đầu vào của nó, `tokens` là một danh sách các token biểu diễn cho một chuỗi đầu vào BERT, còn `candidate_pred_positions` là một danh sách chỉ số của các token của chuỗi đầu vào BERT ngoại trừ những token đặc biệt (token đặc biệt không được dự đoán trong tác vụ mô hình hóa ngôn ngữ có mặt nạ), và `num_mlm_preds` chỉ định số lượng token được dự đoán (nhớ lại rằng 15% token ngẫu nhiên được dự đoán). Dựa trên định nghĩa của tác vụ mô hình hóa ngôn ngữ có mặt nạ trong Section 16.8.5, tại mỗi vị trí dự đoán, đầu vào có thể bị thay thế bởi token đặc biệt “`<mask>`” hoặc một token ngẫu nhiên, hoặc không đổi. Cuối cùng,

hàm này trả về những token đầu vào sau khi thực hiện thay thế (nếu có), những chỉ số token được dự đoán và nhãn cho những 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.randint(0, len(vocab) - 1)
        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` ở trên, hàm dưới đây nhận một chuỗi đầu vào BERT (`tokens`) làm đầu vào và trả về chỉ số của những token đầu vào (sau khi thay thế token (nếu có) như mô tả ở [Section 16.8.5](#)), những chỉ số của token được dự đoán và chỉ số nhãn cho những dự đoán này.

```
#@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
        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]
```

### 16.9.2 Biến đổi Văn bản thành bộ Dữ liệu Tiền huấn luyện

Bây giờ chúng ta gần như đã sẵn sàng để tùy chỉnh một lớp Dataset cho việc tiền huấn luyện BERT. Trước đó, chúng ta vẫn cần định nghĩa một hàm hỗ trợ `_pad_bert_inputs` để giúp nối các token “`<mask>`” đặc biệt vào đầu vào. Đối số `examples` của hàm chứa các kết quả đầu ra từ những hàm hỗ trợ `_get_nsp_data_from_paragraph` và `_get_mlm_data_from_tokens` cho hai tác vụ tiền huấn luyện.

```
#@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(np.array(token_ids + [vocab['<pad>']] * (
            max_len - len(token_ids)), dtype='int32'))
        all_segments.append(np.array(segments + [0] * (
            max_len - len(segments)), dtype='int32'))
        # `valid_lens` excludes count of '<pad>' tokens
        valid_lens.append(np.array(len(token_ids), dtype='float32'))
        all_pred_positions.append(np.array(pred_positions + [0] * (
            max_num_mlm_preds - len(pred_positions)), dtype='int32'))
        # Predictions of padded tokens will be filtered out in the loss via
        # multiplication of 0 weights
        all_mlm_weights.append(
            np.array([1.0] * len(mlm_pred_label_ids) + [0.0] * (
                max_num_mlm_preds - len(pred_positions)), dtype='float32'))
        all_mlm_labels.append(np.array(mlm_pred_label_ids + [0] * (
            max_num_mlm_preds - len(mlm_pred_label_ids)), dtype='int32'))
        nsp_labels.append(np.array(is_next))
    return (all_token_ids, all_segments, valid_lens, all_pred_positions,
            all_mlm_weights, all_mlm_labels, nsp_labels)
```

Kết hợp những hàm hỗ trợ để tạo dữ liệu huấn luyện cho hai tác vụ tiền huấn luyện và hàm hỗ trợ đệm đầu vào, ta tùy chỉnh lớp `_WikiTextDataset` sau đây thành bộ dữ liệu WikiText-2 cho tiền huấn luyện BERT. Bằng cách lập trình hàm `__getitem__`, ta có thể tùy ý truy cập những mẫu dữ liệu tiền huấn luyện (mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo) được tạo ra từ một cặp câu trong kho ngữ liệu WikiText-2.

Mô hình BERT ban đầu sử dụng embedding WordPiece có kích thước bộ từ vựng là 30,000 (Wu et al., 2016). Phương pháp tách token của WordPiece là một phiên bản của thuật toán mã hóa cặp byte ban đầu Section 16.6.2 với một chút chỉnh sửa. Để cho đơn giản, chúng tôi sử dụng hàm `d2l.tokenize` để tách từ. Những token xuất hiện ít hơn năm lần được loại bỏ.

```
#@save
class _WikiTextDataset(gluon.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]
```

(continues on next page)

```

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
(all_token_ids, all_segments, valid_lens,
 all_pred_positions, all_mlm_weights,
 all_mlm_labels, nsp_labels) = _pad_bert_inputs(
    examples, max_len, self.vocab)

def __getitem__(self, idx):
    return (all_token_ids[idx], all_segments[idx],
            valid_lens[idx], all_pred_positions[idx],
            all_mlm_weights[idx], all_mlm_labels[idx],
            nsp_labels[idx])

def __len__(self):
    return len(all_token_ids)

```

Bằng cách sử dụng hàm `_read_wiki` và lớp `_WikiTextDataset`, ta định nghĩa hàm `load_data_wiki` dưới đây để tải xuống bộ dữ liệu WikiText-2 và tạo mẫu dữ liệu tiền huấn luyện.

```

#@save
def load_data_wiki(batch_size, max_len):
    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 = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                       num_workers=num_workers)
    return train_iter, train_set.vocab

```

Đặt kích thước batch là 512 và chiều dài tối đa của chuỗi đầu vào BERT là 64, ta in ra kích thước một minibatch dữ liệu tiền huấn luyện. Lưu ý rằng trong mỗi chuỗi đầu vào BERT, 10 ( $64 \times 0.15$ ) vị trí được dự đoán đối với tác vụ mô hình hóa ngôn ngữ có 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

```

Cuối cùng, hãy nhìn vào kích thước của bộ từ vựng. Mặc dù những token ít xuất hiện đã bị loại bỏ,

kích thước của nó vẫn lớn gấp đôi bộ dữ liệu PTB.

len(vocab)

### 16.9.3 Tóm tắt

- So sánh với tập dữ liệu PTB, tập dữ liệu WikiText-2 vẫn giữ nguyên dấu câu, chữ viết hoa và ký tự số, có kích thước lớn hơn gấp đôi.
- Ta có thể tùy ý truy cập vào các mẫu tiền huấn luyện (tác vụ mô hình hoá ngôn ngữ có mặt nạ và dự đoán câu tiếp theo) được sinh ra từ một cặp câu trong kho dữ liệu WikiText-2.

### 16.9.4 Bài tập

1. Để đơn giản, dấu chấm được dùng làm dấu phân cách duy nhất để tách các câu. Hãy thử các kỹ thuật tách câu khác, ví dụ như công cụ spaCy và NLTK. Lấy NLTK làm ví dụ. Bạn cần cài đặt NLTK trước: `pip install nltk`. Trong mã nguồn, đầu tiên hãy `import nltk`. Sau đó, tải xuống bộ token hóa câu Punkt (*Punkt sentence tokenizer*): `nltk.download('punkt')`. Để tách các câu, ví dụ `sentences = 'This is great ! Why not ?'`, việc gọi `nltk.tokenize.sent_tokenize(sentences)` sẽ trả về một danh sách gồm hai chuỗi câu là `['This is great !', 'Why not ?']`.
2. Nếu ta không lọc ra những token ít gặp thì kích thước bộ từ vựng là bao nhiêu?

### 16.9.5 Thảo luận

- Tiếng Anh: MXNet<sup>349</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>350</sup>

### 16.9.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Nguyễn Đình Nam
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường
- Phạm Minh Đức

Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 29/08/2020)

<sup>349</sup> <https://discuss.d2l.ai/t/389>

<sup>350</sup> <https://forum.machinelearningcovan.com/c/d2l>

## 16.10 Tiền Huấn luyện BERT

Trong phần này, sử dụng mô hình BERT đã được lập trình trong Section 16.8 và các mẫu dữ liệu tiền huấn luyện được tạo ra từ tập dữ liệu WikiText-2 trong Section 16.9, ta sẽ tiền huấn luyện BERT trên tập dữ liệu này.

```
from d2l import mxnet as d2l  
from mxnet import autograd, gluon, init, np, npx  
  
npx.set_np()
```

Đầu tiên, ta nạp các mẫu dữ liệu của tập dữ liệu WikiText-2 thành các minibatch cho quá trình tiền huấn luyện hai tác vụ: mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo. Kích thước batch là 512 và độ dài tối đa của chuỗi đầu vào BERT là 64. Lưu ý rằng trong mô hình BERT gốc, độ dài tối đa này là 512.

```
batch_size, max_len = 512, 64  
train_iter, vocab = d2l.load_data_wiki(batch_size, max_len)
```

### 16.10.1 Tiền Huấn luyện BERT

Mô hình BERT gốc có hai phiên bản với hai kích thước mô hình khác nhau (Devlin et al., 2018). Mô hình cơ bản (BERT<sub>BASE</sub>) sử dụng 12 tầng (khối mã hóa của Transformer) với 768 nút ẩn (kích thước ẩn) và tầng tự tập trung 12 đầu. Mô hình lớn (BERT<sub>LARGE</sub>) sử dụng 24 tầng với 1024 nút ẩn và tầng tự tập trung 16 đầu. Đáng chú ý là tổng số lượng tham số trong mô hình đầu tiên là 110 triệu, còn ở mô hình thứ hai là 340 triệu. Để minh họa thì ta định nghĩa mô hình BERT nhỏ dưới đây, sử dụng 2 tầng với 128 nút ẩn và tầng tự tập trung 2 đầu.

```
net = d2l.BERTModel(len(vocab), num_hiddens=128, ffn_num_hiddens=256,  
                     num_heads=2, num_layers=2, dropout=0.2)  
devices = d2l.try_all_gpus()  
net.initialize(init.Xavier(), ctx=devices)  
loss = gluon.loss.SoftmaxCELoss()
```

Ta sẽ định nghĩa hàm hỗ trợ `_get_batch_loss_bert` trước khi bắt đầu lập trình vòng lặp cho quá trình huấn luyện. Hàm này nhận đầu vào là một batch các mẫu huấn luyện và tính giá trị mất mát đối với hai tác vụ mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo. Lưu ý rằng mất mát cuối cùng của tác vụ tiền huấn luyện BERT chỉ là tổng mất mát của cả hai tác vụ nói trên.

```
#@save  
def _get_batch_loss_bert(net, loss, vocab_size, tokens_X_shards,  
                        segments_X_shards, valid_lens_x_shards,  
                        pred_positions_X_shards, mlm_weights_X_shards,  
                        mlm_Y_shards, nsp_y_shards):  
    mlm_ls, nsp_ls, ls = [], [], []  
    for (tokens_X_shard, segments_X_shard, valid_lens_x_shard,  
          pred_positions_X_shard, mlm_weights_X_shard, mlm_Y_shard,  
          nsp_y_shard) in zip(  
            tokens_X_shards, segments_X_shards, valid_lens_x_shards,  
            pred_positions_X_shards, mlm_weights_X_shards, mlm_Y_shards,  
            nsp_y_shards):
```

(continues on next page)

```

# Forward pass
_, mlm_Y_hat, nsp_Y_hat = net(
    tokens_X_shard, segments_X_shard, valid_lens_x_shard.reshape(-1),
    pred_positions_X_shard)
# Compute masked language model loss
mlm_l = loss(
    mlm_Y_hat.reshape((-1, vocab_size)), mlm_Y_shard.reshape(-1),
    mlm_weights_X_shard.reshape((-1, 1)))
mlm_l = mlm_l.sum() / (mlm_weights_X_shard.sum() + 1e-8)
# Compute next sentence prediction loss
nsp_l = loss(nsp_Y_hat, nsp_y_shard)
nsp_l = nsp_l.mean()
mlm_ls.append(mlm_l)
nsp_ls.append(nsp_l)
ls.append(mlm_l + nsp_l)
npx.waitall()
return mlm_ls, nsp_ls, ls

```

Sử dụng hai hàm hỗ trợ được đề cập ở trên, hàm `train_bert` dưới đây sẽ định nghĩa quá trình tiền huấn luyện BERT (`net`) trên tập dữ liệu WikiText-2 (`train_iter`). Việc huấn luyện BERT có thể mất rất nhiều thời gian. Do đó, thay vì truyền vào số lượng epoch huấn luyện như trong hàm `train_ch13` (Section 15.1), ta sử dụng tham số `num_steps` trong hàm sau để xác định số vòng lặp huấn luyện.

```

#@save
def train_bert(train_iter, net, loss, vocab_size, devices, log_interval,
               num_steps):
    trainer = gluon.Trainer(net.collect_params(), 'adam',
                           {'learning_rate': 1e-3})
    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 batch in train_iter:
            (tokens_X_shards, segments_X_shards, valid_lens_x_shards,
             pred_positions_X_shards, mlm_weights_X_shards,
             mlm_Y_shards, nsp_y_shards) = [gluon.utils.split_and_load(
                 elem, devices, even_split=False) for elem in batch]
            timer.start()
            with autograd.record():
                mlm_ls, nsp_ls, ls = _get_batch_loss_bert(
                    net, loss, vocab_size, tokens_X_shards, segments_X_shards,
                    valid_lens_x_shards, pred_positions_X_shards,
                    mlm_weights_X_shards, mlm_Y_shards, nsp_y_shards)
                for l in ls:
                    l.backward()
            trainer.step(1)
            mlm_l_mean = sum([float(l) for l in mlm_ls]) / len(mlm_ls)
            nsp_l_mean = sum([float(l) for l in nsp_ls]) / len(nsp_ls)
            metric.add(mlm_l_mean, nsp_l_mean, batch[0].shape[0], 1)

```

(continues on next page)

```

        timer.stop()
        if (step + 1) % log_interval == 0:
            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)}')

```

Ta có thể vẽ đồ thị hàm mất mát ứng với hai tác vụ mô hình hóa ngôn ngữ có mặt nạ và dự đoán câu tiếp theo trong quá trình tiền huấn luyện BERT.

```
train_bert(train_iter, net, loss, len(vocab), devices, 1, 50)
```

### 16.10.2 Biểu diễn Văn bản với BERT

Ta có thể sử dụng mô hình BERT đã tiền huấn luyện để biểu diễn một văn bản đơn, cặp văn bản hay một token bất kỳ trong văn bản. Hàm sau sẽ trả về biểu diễn của mô hình BERT (net) cho toàn bộ 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 = np.expand_dims(np.array(vocab[tokens], ctx=devices[0]),
                               axis=0)
    segments = np.expand_dims(np.array(segments, ctx=devices[0]), axis=0)
    valid_len = np.expand_dims(np.array(len(tokens), ctx=devices[0]), axis=0)
    encoded_X, _, _ = net(token_ids, segments, valid_len)
    return encoded_X

```

Xét câu “a crane is flying”. Hãy nhớ lại biểu diễn đầu vào của BERT được thảo luận trong Section 16.8.4, sau khi thêm các token đặc biệt “<cls>” (dùng cho phân loại) và “<sep>” (dùng để ngăn cách), chiều dài của chuỗi đầu vào BERT là 6. Vì 0 là chỉ số của token “<cls>”, encoded\_text[:, 0, :] là biểu diễn BERT của toàn bộ câu đầu vào. Để đánh giá token đa nghĩa “crane”, ta sẽ in cả ba phần tử đầu tiên trong biểu diễn BERT của token này.

```

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]

```

Bây giờ, ta sẽ xem xét cặp câu “a crane driver came” và “he just left”. Tương tự như trên, encoded\_pair[:, 0, :] là kết quả mã hóa của cặp câu này thông qua BERT đã được tiền huấn luyện. Lưu ý rằng khi token đa nghĩa “crane” xuất hiện trong ngữ cảnh khác nhau, ba phần tử đầu

tiên trong biểu diễn BERT token này cũng thay đổi. Điều này thể hiện rằng biểu diễn BERT có tính nhạy 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]
```

Ở Section 17, ta sẽ tinh chỉnh mô hình BERT đã được tiền huấn luyện với một số tác vụ xuôi dòng trong xử lý ngôn ngữ tự nhiên.

### 16.10.3 Tóm tắt

- Mô hình BERT gốc có hai phiên bản, trong đó mô hình cơ bản có 110 triệu tham số và mô hình lớn có 340 triệu tham số.
- Ta có thể sử dụng mô hình BERT đã được tiền huấn luyện để biểu diễn một văn bản đơn, cặp văn bản hay một token bất kỳ.
- Trong thí nghiệm trên, ta đã thấy rằng cùng một token có thể có nhiều cách biểu diễn khác nhau với những ngữ cảnh khác nhau. Điều này thể hiện rằng biểu diễn BERT có tính nhạy ngữ cảnh.

### 16.10.4 Bài tập

- Kết quả thí nghiệm trên cho thấy mất mát ứng với tác vụ mô hình hóa ngôn ngữ có mặt nạ cao hơn đáng kể so với tác vụ dự đoán câu tiếp theo. Hãy giải thích hiện tượng này.
- Thay đổi chiều dài tối đa của chuỗi đầu vào BERT thành 512 (giống với mô hình BERT gốc) và sử dụng cấu hình của mô hình BERT gốc như là BERT<sub>LARGE</sub>. Bạn có gặp lỗi khi chạy lại thí nghiệm không? Giải thích tại sao.

### 16.10.5 Thảo luận

- Tiếng Anh: MXNet<sup>351</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>352</sup>

<sup>351</sup> <https://discuss.d2l.ai/t/390>

<sup>352</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### **16.10.6 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Bùi Thị Cẩm Nhung
- Nguyễn Văn Quang
- Phạm Minh Đức
- Nguyễn Văn Cường

*Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 21/07/2020)*

#### **16.11 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Nguyễn Văn Cường

*Lần cập nhật gần nhất: 12/09/2020. (Cập nhật lần cuối từ nội dung gốc: 02/04/2020)*



# 17 | Xử lý Ngôn ngữ Tự nhiên: Ứng dụng

Ở Section 16, chúng ta đã nhìn thấy cách biểu diễn token văn bản và huấn luyện các biểu diễn của chúng. Những biểu diễn văn bản được tiền huấn luyện như vậy có thể được truyền vào các mô hình cho các tác vụ xử lý ngôn ngữ tự nhiên xuôi dòng khác nhau.

Cuốn sách này không có ý định trình bày các ứng dụng xử lý ngôn ngữ tự nhiên một cách toàn diện. Trọng tâm của cuốn sách là *làm sao để áp dụng học biểu diễn (sâu) của ngôn ngữ nhằm giải quyết các bài toán xử lý ngôn ngữ tự nhiên*. Tuy nhiên, chúng ta đã thảo luận về một số ứng dụng xử lý ngôn ngữ tự nhiên mà không cần tiền huấn luyện trong các chương trước, nhằm chỉ giải thích các kiến trúc học sâu. Như trong Section 10, chúng ta đã thiết kế các mô hình ngôn ngữ dựa trên RNN để sinh ra các văn bản có nội dung giống như tiểu thuyết. Trong Section 11 và Section 12, ta cũng đã thiết kế các mô hình ngôn ngữ dựa trên RNN và các cơ chế tập trung cho tác vụ dịch máy. Trong chương này, với những biểu diễn văn bản được tiền huấn luyện thì ta sẽ xem xét hai tác vụ xử lý ngôn ngữ tự nhiên xuôi dòng khác đó là, phân tích cảm xúc (*sentiment analysis*) và suy luận ngôn ngữ tự nhiên (*natural language inference*). Đây là những ứng dụng xử lý ngôn ngữ tự nhiên mang tính phổ biến và đại diện: ứng dụng trước phân tích văn bản đơn lẻ trong khi ứng dụng sau phân tích mối quan hệ của các cặp văn bản.

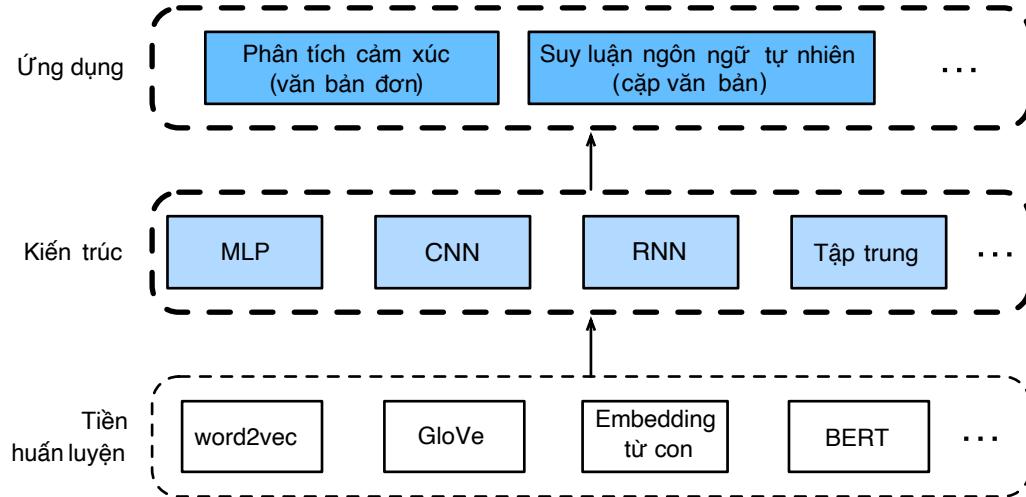


Fig. 17.1: Biểu diễn văn bản được tiền huấn luyện có thể được truyền vào các kiến trúc học sâu cho các ứng dụng xử lý ngôn ngữ tự nhiên xuôi dòng khác nhau. Chương này sẽ tập trung vào cách thiết kế mô hình cho các ứng dụng khác nhau đó.

Như được mô tả trong Fig. 17.1, chương này sẽ tập trung vào việc mô tả các ý tưởng cơ bản trong 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 và cơ chế tập trung. Mặc dù có thể kết hợp bất kỳ biểu diễn văn bản được tiền huấn luyện với bất kỳ kiến trúc nào cho các tác vụ xử lý ngôn ngữ tự nhiên xuôi dòng

trong Fig. 17.1, nhưng ta chỉ chọn một vài kết hợp đại diện mà thôi. Cụ thể, chúng ta sẽ khám phá các kiến trúc phổ biến dựa trên RNN và CNN để phân tích cảm xúc. Đối với suy luận ngôn ngữ tự nhiên, ta sẽ chọn cơ chế tập trung và MLP để minh họa cách phân tích quan hệ giữa các cặp văn bản. Cuối cùng, ta sẽ giới thiệu cách tinh chỉnh mô hình BERT được tiền huấn luyện cho một loạt các ứng dụng xử lý ngôn ngữ tự nhiên, ví dụ như các tác vụ cấp chuỗi (phân loại đơn văn bản và phân loại cặp văn bản) và cấp token (gắn thẻ văn bản và trả lời câu hỏi). Chúng ta sẽ tinh chỉnh BERT để xử lý ngôn ngữ tự nhiên như một thực nghiệm cụ thể.

Như đã giới thiệu trong Section 16.8, BERT chỉ yêu cầu các 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 phải tinh chỉnh một số lượng lớn các tham số mô hình BERT cho các ứng dụng xuôi dòng. Khi độ phức tạp về không gian hoặc thời gian bị giới hạn, những mô hình được thiết kế thủ công dựa trên MLP, CNN, RNN và cơ chế tập trung sẽ khả thi hơn. Trong phần sau, ta sẽ bắt đầu bằng ứng dụng phân tích cảm xúc và minh họa thiết kế mô hình dựa trên kiến trúc RNN và CNN tương ứng.

## 17.1 Tác vụ Phân tích Cảm xúc và Bộ Dữ liệu

Phân loại văn bản là một tác vụ phổ biến trong xử lý ngôn ngữ tự nhiên, ánh xạ chuỗi văn bản có độ dài không cố định tới một hạng mục tương ứng. Tác vụ này khá giống với phân loại ảnh, vốn là ứng dụng phổ biến nhất được giới thiệu trong cuốn sách này, ví dụ, Section 20.9. Điểm khác biệt duy nhất đó là, mẫu đầu vào của tác vụ phân loại là một câu văn bản thay vì một bức ảnh.

Phần này sẽ tập trung vào việc nạp dữ liệu cho một trong số những câu hỏi của bài toán này: sử dụng tác vụ phân loại cảm xúc văn bản để phân tích cảm xúc của người viết. Bài toán này cũng có thể gọi là phân tích cảm xúc (sắc thái) và có rất nhiều ứng dụng. Ví dụ, ta có thể phân tích đánh giá của khách hàng về sản phẩm để thu được thống kê độ hài lòng, hoặc phân tích cảm xúc của khách hàng về điều kiện thị trường và sử dụng kết quả này để dự đoán xu hướng tương lai.

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
import os
npx.set_np()
```

### 17.1.1 Bộ Dữ liệu Phân tích Cảm xúc

Ta sử dụng tập dữ liệu lớn về đánh giá phim ảnh<sup>353</sup> (*Large Movie Review Dataset*) của Stanford làm dữ liệu cho tác vụ phân tích cảm xúc. Tập dữ liệu này được chia thành hai tập huấn luyện và kiểm tra, mỗi tập chứa 25,000 đánh giá phim tải về từ IMDb. Trong mỗi tập dữ liệu, số lượng đánh giá có nhãn “tích cực” (*positive*) và “tiêu cực” (*negative*) là bằng nhau.

<sup>353</sup> <https://ai.stanford.edu/~amaas/data/sentiment/>

## Đọc Dữ liệu

Đầu tiên, ta tải dữ liệu về thư mục “./data” và giải nén dữ liệu vào thư mục “./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, ta đọc dữ liệu huấn luyện và dữ liệu kiểm tra. Mỗi mẫu là một bình luận đánh giá cùng với nhãn tương ứng: 1 cho “tích cực”, và 0 cho “tiêu cực”.

```
#@save
def read_imdb(data_dir, is_train):
    data, labels = [], []
    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])
```

## Token hóa và Bộ từ vựng

Ta coi mỗi từ là một token, và tạo một từ điển dựa trên tập dữ liệu huấn luyện.

```
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=['<pad>'])

d2l.set_figsize()
d2l.plt.hist([len(line) for line in train_tokens], bins=range(0, 1000, 50));
```

## Đệm để cùng Độ dài

Vì mỗi câu đánh giá có độ dài khác nhau, nên chúng không thể được tổng hợp trực tiếp thành minibatch được. Ta có thể cố định độ dài mỗi câu bình luận là 500 bằng cách cắt xén hoặc thêm vào các chỉ mục “<unk>”.

```
num_steps = 500 # sequence length
train_features = np.array([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_tokens])
train_features.shape
```

## Tạo Iterator cho Dữ liệu

Bây giờ, ta sẽ tạo một iterator cho dữ liệu. Mỗi vòng lặp sẽ trả về một minibatch dữ liệu.

```
train_iter = d2l.load_array((train_features, train_data[1]), 64)

for X, y in train_iter:
    print('X', X.shape, 'y', y.shape)
    break
'# batches:', len(train_iter)
```

### 17.1.2 Kết hợp Tất cả Lại

Cuối cùng, ta lưu hàm load\_data\_imdb vào d2l, hàm này trả về bộ từ vựng và các iterator của dữ liệu.

```
#@save
def load_data_imdb(batch_size, num_steps=500):
    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 = np.array([d2l.truncate_pad(
        vocab[line], num_steps, vocab.unk) for line in train_tokens])
    test_features = np.array([d2l.truncate_pad(
        vocab[line], num_steps, vocab.unk) for line in test_tokens])
    train_iter = d2l.load_array((train_features, train_data[1]), batch_size)
    test_iter = d2l.load_array((test_features, test_data[1]), batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

### 17.1.3 Tóm tắt

- Tác vụ phân loại văn bản có thể phân loại chuỗi văn bản theo hạng mục.
- Để phân loại cảm xúc văn bản, ta nạp bộ dữ liệu IMDb và token hóa các từ trong đó. Sau đó, ta thêm vào chuỗi văn bản của các câu đánh giá ngắn và tạo một iterator dữ liệu.

### 17.1.4 Bài tập

Hãy khám phá một tập dữ liệu ngôn ngữ tự nhiên khác (ví dụ tập dữ liệu Đánh giá Amazon<sup>354</sup>) và xây dựng một hàm data\_loader tương tự như load\_data\_imdb.

<sup>354</sup> <https://snap.stanford.edu/data/web-Amazon.html>

### 17.1.5 Thảo luận

- Tiếng Anh: MXNet<sup>355</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>356</sup>

### 17.1.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Văn Quang
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

## 17.2 Phân tích Cảm xúc: Sử dụng Mạng Nơ-ron Hồi tiếp

Tương tự như tìm kiếm các từ đồng nghĩa và loại suy, phân loại văn bản cũng là một tác vụ xuôi dòng của embedding từ. Trong phần này, ta sẽ áp dụng các vector từ đã được tiền huấn luyện (GloVe) và mạng nơ-ron truy hồi hai chiều với nhiều lớp ẩn (Maas et al., 2011), như được minh họa trong Fig. 17.2.1. Ta sẽ sử dụng mô hình này để xác định xem một chuỗi văn bản có độ dài không xác định chứa cảm xúc tích cực hay tiêu cực.

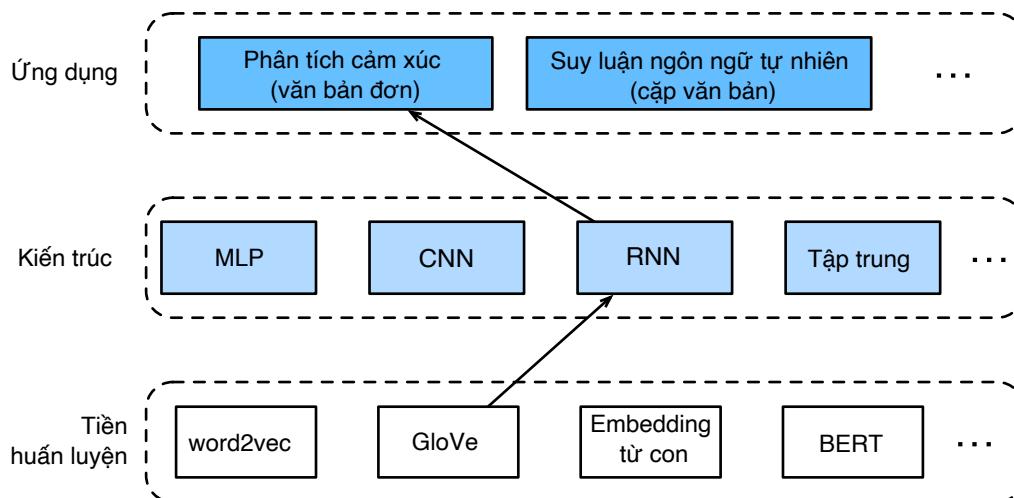


Fig. 17.2.1: Phần này sẽ truyền các vector GloVe đã được tiền huấn luyện vào một kiến trúc RNN cho bài toán phân tích cảm xúc.

<sup>355</sup> <https://discuss.d2l.ai/t/391>

<sup>356</sup> <https://forum.machinelearningcoban.com/c/d2l>

```

from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn, rnn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)

```

### 17.2.1 Sử dụng Mạng Nơ-ron Hồi tiếp

Trong mô hình này, đầu tiên mỗi từ nhận được một vector đặc trưng tương ứng từ tầng embedding. Sau đó, ta mã hóa thêm chuỗi đặc trưng bằng cách sử dụng mạng nơ-ron hồi tiếp hai chiều để thu được thông tin chuỗi. Cuối cùng, ta chuyển đổi thông tin chuỗi được mã hóa thành đầu ra thông qua tầng kết nối đầy đủ. Cụ thể, ta có thể ghép nối các trạng thái ẩn của bộ nhớ ngắn hạn dài hai chiều (*bidirectional long-short term memory*) ở bước thời gian ban đầu và bước thời gian cuối cùng và truyền nó tới tầng phân loại đầu ra như là đặc trưng mã hóa của thông tin chuỗi. Trong lớp BiRNN được lập trình bên dưới, thực thể Embedding là tầng embedding, thực thể LSTM là tầng ẩn để mã hóa chuỗi, và thực thể Dense là tầng đầu ra sinh kết quả phân loại.

```

class BiRNN(nn.Block):
    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 recurrent neural
        # network
        self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                               bidirectional=True, input_size=embed_size)
        self.decoder = nn.Dense(2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of words). Because LSTM
        # needs to use sequence as the first dimension, the input is
        # transformed and the word feature is then extracted. The output shape
        # is (no. of words, batch size, word vector dimension).
        embeddings = self.embedding(inputs.T)
        # Since the input (embeddings) is the only argument passed into
        # rnn.LSTM, it only returns the hidden states of the last hidden layer
        # at different time step (outputs). The shape of `outputs` is
        # (no. of words, 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 = np.concatenate((outputs[0], outputs[-1]), axis=1)
        outs = self.decoder(encoding)
        return outs

```

Ta sẽ tạo một mạng nơ-ron hồi tiếp hai chiều với hai tầng ẩn như sau.

```

embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d2l.try_all_gpus()
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)
net.initialize(init.Xavier(), ctx=devices)

```

## Nạp các Vector Từ đã qua Tiền huấn luyện

Bởi vì tập dữ liệu huấn luyện cho việc phân loại cảm xúc không quá lớn, để xử lý vấn đề quá khớp, ta sẽ dùng trực tiếp các vector từ đã được tiền huấn luyện trên tập ngữ liệu lớn hơn làm các vector đặc trưng cho tất cả các từ. Ở đây, ta nạp vector từ Glove 100-chiều cho mỗi từ trong từ điển vocab.

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
```

Truy vấn các vector từ nằm trong từ vựng của chúng ta.

```
embeds = glove_embedding[vocab.idx_to_token]  
embeds.shape
```

Tiếp theo, ta sử dụng các vector từ đó làm vector đặc trưng cho mỗi từ trong các đánh giá. Lưu ý là các chiều của vector từ đã qua tiền huấn luyện cần nhất quán với kích thước đầu ra embed\_size của tầng embedding trong mô hình đã tạo.Thêm vào đó, ta không còn cập nhật các vector từ này trong suốt quá trình huấn luyện.

```
net.embedding.weight.set_data(embeds)  
net.embedding.collect_params().setattr('grad_req', 'null')
```

## Huấn luyện và Đánh giá Mô hình

Bây giờ ta có thể bắt đầu thực hiện huấn luyện.

```
lr, num_epochs = 0.01, 5  
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})  
loss = gluon.loss.SoftmaxCrossEntropyLoss()  
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

Cuối cùng, định nghĩa hàm dự đoán.

```
#@save  
def predict_sentiment(net, vocab, sentence):  
    sentence = np.array(vocab[sentence.split()], ctx=d2l.try_gpu())  
    label = np.argmax(net(sentence.reshape(1, -1)), axis=1)  
    return 'positive' if label == 1 else 'negative'
```

Tiếp theo, sử dụng mô hình đã huấn luyện để phân loại cảm xúc cho hai câu đơn giản.

```
predict_sentiment(net, vocab, 'this movie is so great')
```

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

### 17.2.2 Tóm tắt

- Phân loại văn bản ánh xạ một chuỗi văn bản có độ dài không xác định thành hạng mục tương ứng của văn bản đó. Đây là một tác vụ xuôi dòng của embedding từ.
- Ta có thể áp dụng các vector từ được tiền huấn luyện và mạng nơ-ron hồi tiếp để để phân loại cảm xúc trong văn bản.

### 17.2.3 Bài tập

1. Hãy tăng số epoch. Bạn có thể đạt được độ chính xác là bao nhiêu trên tập huấn luyện và tập kiểm tra? Thủ tinh chỉnh các siêu tham số khác và đánh giá kết quả.
2. Liệu sử dụng vector từ được tiền huấn luyện có kích thước lớn hơn, ví dụ vector từ GloVe có kích thước chiều là 300, có thể cải thiện độ chính xác hay không?
3. Ta có thể cải thiện độ chính xác bằng cách sử dụng công cụ token hoá từ spaCy không? Bạn cần cài đặt spaCy bằng lệnh pip install spacy và cài đặt gói ngôn ngữ tiếng Anh bằng lệnh python -m spacy download en. Trong mã nguồn, đầu tiên hãy nhập thư viện spaCy với câu lệnh import spacy. Tiếp theo, hãy nạp gói spacy tiếng Anh spacy\_en = spacy.load('en'). Cuối cùng, hãy định nghĩa hàm def tokenizer(text): return [tok.text for tok in spacy\_en.tokenizer(text)] và thay thế hàm tokenizer ban đầu. Lưu ý rằng vector từ GloVe sử dụng “-” để kết nối mỗi từ trong cụm danh từ. Ví dụ, cụm từ “new york” được biểu diễn bằng “new-york” trong GloVe. Sau khi sử dụng công cụ token hoá spaCy, “new york” có thể sẽ được lưu thành “new york”.

### 17.2.4 Thảo luận

- Tiếng Anh: MXNet<sup>357</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>358</sup>

### 17.2.5 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Phạm Minh Đức

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 29/08/2020)

<sup>357</sup> <https://discuss.d2l.ai/t/392>

<sup>358</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 17.3 Phân tích Cảm xúc: Sử dụng Mạng Nơ-ron Tích Chập

Trong Section 8, chúng ta đã tìm hiểu cách xử lý dữ liệu ảnh hai chiều với mạng nơ-ron tích chập hai chiều. Ở chương trước về các mô hình ngôn ngữ và các tác vụ phân loại văn bản, ta coi dữ liệu văn bản như là dữ liệu chuỗi thời gian với chỉ một chiều duy nhất, và vì vậy, chúng sẽ được xử lý bằng mạng nơ-ron hồi tiếp. Thực tế, ta cũng có thể coi văn bản như một bức ảnh một chiều, và sử dụng mạng nơ-ron tích chập một chiều để tìm ra mối liên kết giữa những từ liền kề nhau. Như mô tả trong Fig. 17.3.1, chương này sẽ miêu tả một hướng tiếp cận đột phá bằng cách áp dụng mạng nơ-ron tích chập để phân tích cảm xúc: textCNN (Kim, 2014).

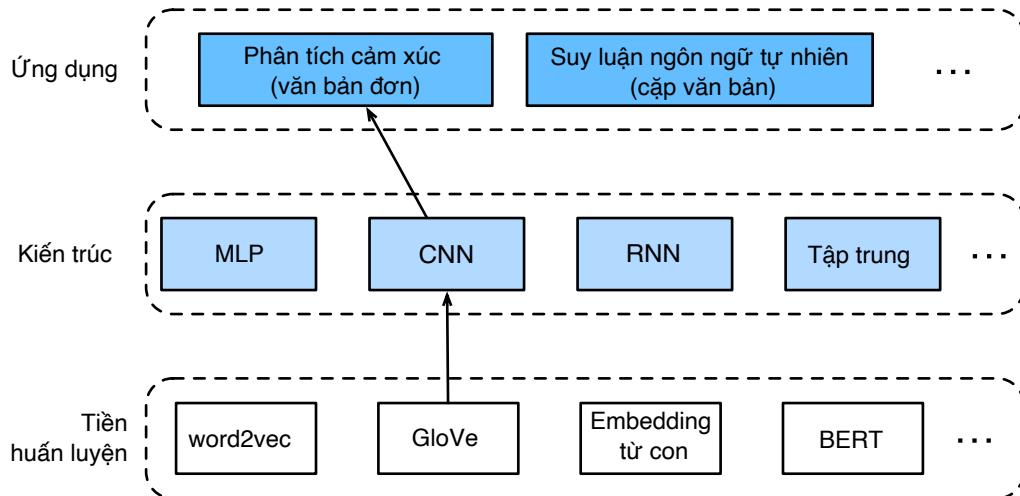


Fig. 17.3.1: Phần này truyền mô hình tiền huấn luyện GloVe vào một kiến trúc mạng nơ-ron tích chập cho tác vụ phân loại cảm xúc

Đầu tiên, nhập những gói thư viện và mô-đun cần thiết cho thử nghiệm.

```
from d2l import mxnet as d2l
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

### 17.3.1 Mạng Nơ-ron Tích chập Một chiều

Trước khi giới thiệu mô hình, chúng ta hãy xem mạng nơ-ron tích chập một chiều hoạt động như thế nào. Tương tự như mạng nơ-ron tích chập hai chiều, mạng nơ-ron tích chập một chiều sử dụng phép tính tương quan chéo một chiều. Trong phép tính tương quan chéo một chiều, cửa sổ tích chập bắt đầu từ phía ngoài cùng bên trái của mảng đầu vào và trượt lần lượt từ trái qua phải. Xét trên một vị trí nhất định của cửa sổ tích chập khi trượt, ta nhận từng phần tử của mảng đầu vào con trong cửa sổ đó với mảng hạt nhân rồi cộng lại để lấy được phần tử ở vị trí tương ứng trong mảng đầu ra. Như ví dụ ở Fig. 17.3.2, đầu vào là một mảng một chiều với độ rộng là 7 và độ rộng của mảng hạt nhân là 2. Ta có thể thấy rằng độ rộng của đầu ra là  $7 - 2 + 1 = 6$  và phần tử đầu tiên được tính bằng cách nhân theo từng phần tử mảng đầu vào con chứa 2 phần tử ngoài cùng bên trái với mảng hạt nhân, rồi cộng lại với nhau.



Fig. 17.3.2: Phép tính tương quan chéo một chiều. Những vùng in đậm là phần tử đầu ra đầu tiên, cùng phần tử đầu vào và mảng hạt nhân được dùng trong phép tính đó:  $0 \times 1 + 1 \times 2 = 2$ .

Tiếp theo, chúng ta sẽ lập trình phép tương quan chéo một chiều trong hàm `corr1d`. Hàm này nhận mảng đầu vào  $X$  và mảng hạt nhân  $K$  và cho ra đầu ra là mảng  $Y$ .

```
def corr1d(X, K):
    w = K.shape[0]
    Y = np.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i:i + w] * K).sum()
    return Y
```

Bây giờ chúng ta sẽ tái tạo lại kết quả của phép tính tương quan chéo một chiều ở Fig. 17.3.2.

```
X, K = np.array([0, 1, 2, 3, 4, 5, 6]), np.array([1, 2])
corr1d(X, K)
```

Phép tính tương quan chéo một chiều cho nhiều kênh đầu vào cũng tương tự như phép tương quan chéo hai chiều cho nhiều kênh đầu vào. Với mỗi kênh, toán tử này thực hiện phép tính tương quan chéo một chiều trên từng hạt nhân và đầu vào tương ứng, và cộng các kết quả trên từng kênh lại với nhau để thu được đầu ra. Fig. 17.3.3 minh họa phép tính tương quan chéo một chiều với ba kênh đầu vào.

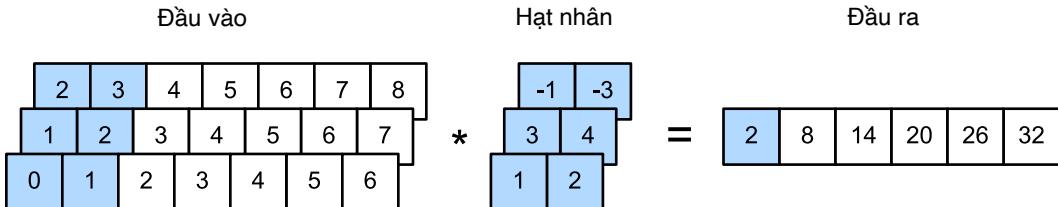


Fig. 17.3.3: Phép tính tương quan chéo một chiều với ba kênh đầu vào. Những vùng được in đậm là phần tử đầu ra thứ nhất cũng như đầu vào và các phần tử của mảng hạt nhân được sử dụng trong phép tính:  $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$ .

Bây giờ, ta sẽ tái tạo lại kết quả của phép tính tương quan chéo một chiều với đa kênh đầu vào trong Fig. 17.3.3.

```
def corr1d_multi_in(X, K):
    # First, we traverse along the 0th dimension (channel dimension) of 'X'
    # and 'K'. Then, we add them together by using * to turn the result list
    # into a positional argument of the `add_n` function
    return sum(corr1d(x, k) for x, k in zip(X, K))
```

```
X = np.array([[0, 1, 2, 3, 4, 5, 6],
```

(continues on next page)

```
[1, 2, 3, 4, 5, 6, 7],  
[2, 3, 4, 5, 6, 7, 8]])  
K = np.array([[1, 2], [3, 4], [-1, -3]])  
corr1d_multi_in(X, K)
```

Định nghĩa phép tính tương quan chéo hai chiều cho ta thấy phép tính tương quan chéo một chiều với đa kênh đầu vào có thể được coi là phép tính tương quan chéo hai chiều với một kênh đầu vào. Như minh họa trong Fig. 17.3.4, ta có thể biểu diễn phép tính tương quan chéo một chiều với đa kênh đầu vào trong Fig. 17.3.3 tương tự như phép tính tương quan chéo hai chiều với một kênh đầu vào. Ở đây, chiều cao của hạt nhân bằng với chiều cao của đầu vào.

| Đầu vào   | Hạt nhân | Đầu ra  |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
|---|----------|---|----|----|----|---|---|---|---|---|---|---|---|---|---|---|---|---|----|----|----|--------|---|----|----|---|---|---|---|
| <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td><td style="padding: 2px;">5</td><td style="padding: 2px;">6</td><td style="padding: 2px;">7</td><td style="padding: 2px;">8</td></tr> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td><td style="padding: 2px;">5</td><td style="padding: 2px;">6</td><td style="padding: 2px;">7</td></tr> <tr><td style="padding: 2px;">0</td><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td><td style="padding: 2px;">5</td><td style="padding: 2px;">6</td></tr> </table> | 2        | 3   | 4  | 5  | 6  | 7 | 8 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 4  | 5  | 6  | $\ast$ | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;">-1</td><td style="padding: 2px;">-3</td></tr> <tr><td style="padding: 2px;">3</td><td style="padding: 2px;">4</td></tr> <tr><td style="padding: 2px;">1</td><td style="padding: 2px;">2</td></tr> </table> | -1 | -3 | 3 | 4 | 1 | 2 |
| 2   | 3        | 4   | 5  | 6  | 7  | 8 |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| 1   | 2        | 3   | 4  | 5  | 6  | 7 |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| 0   | 1        | 2   | 3  | 4  | 5  | 6 |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| -1  | -3       |   |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| 3   | 4        |   |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| 1   | 2        |   |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
|   | $=$      | <table border="1" style="border-collapse: collapse; width: 100%;"> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td><td style="padding: 2px;"></td></tr> <tr><td style="padding: 2px; background-color: #ADD8E6;">2</td><td style="padding: 2px;">8</td><td style="padding: 2px;">14</td><td style="padding: 2px;">20</td><td style="padding: 2px;">26</td><td style="padding: 2px;">32</td><td style="padding: 2px;"></td></tr> </table> |    |    |    |   |   |   |   |   |   |   |   |   |   |   | 2 | 8 | 14 | 20 | 26 | 32     |   |    |    |   |   |   |   |
|   |          |   |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
|   |          |   |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |
| 2   | 8        | 14  | 20 | 26 | 32 |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |        |   |    |    |   |   |   |   |

Fig. 17.3.4: Phép tính tương quan chéo hai chiều với một kênh đầu vào. Vùng được tô đậm là phần tử đầu ra thứ nhất và đầu vào cũng như các phần tử của mảng hạt nhân được sử dụng trong phép tính:  $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$ .

Cả hai đầu ra trong Fig. 17.3.2 và Fig. 17.3.3 chỉ có một kênh. Ta đã thảo luận cách chỉ định đa kênh đầu ra trong tầng tích chập hai chiều tại Section 8.4. Tương tự, ta cũng có thể chỉ định đa kênh đầu ra trong tầng tích chập một chiều để mở rộng các tham số mô hình trong tầng tích chập đó.

### 17.3.2 Tầng Gộp Cực đại Theo Thời gian

Tương tự, ta có tầng gộp một chiều. Tầng gộp cực đại theo thời gian được dùng trong TextCNN thực chất tương tự như tầng gộp cực đại toàn cục một chiều. Giả sử đầu vào có nhiều kênh, mỗi kênh bao gồm các giá trị bước thời gian khác nhau, đầu ra của mỗi kênh sẽ là giá trị lớn nhất qua tất cả bước thời gian trong từng kênh. Do đó, đầu vào của tầng gộp cực đại theo thời gian có thể có số lượng bước thời gian khác nhau tại mỗi kênh.

Để cải thiện chất lượng tính toán, ta thường kết hợp những mẫu thời gian có độ dài khác nhau vào một minibatch và làm cho chiều dài theo thời gian của từng mẫu đồng nhất bằng cách thêm các ký tự đặc biệt (ví dụ 0) vào cuối những mẫu ngắn hơn. Tất nhiên, các ký tự được thêm vào không làm thay đổi bản chất ngữ nghĩa. Bởi vì, mục tiêu chính của tầng gộp cực đại theo thời gian là học được những đặc trưng quan trọng của thời gian, thông thường điều đó cho phép mô hình không bị ảnh hưởng bởi các ký tự được thêm vào thủ công.

### 17.3.3 Mô hình TextCNN

TextCNN chủ yếu sử dụng tầng tích chập một chiều và tầng gộp cực đại theo thời gian. Giả sử chuỗi văn bản đầu vào gồm  $n$  từ, mỗi từ được biểu diễn bởi một vector  $d$  chiều. Lúc này mẫu đầu vào có chiều rộng là  $n$ , chiều cao là 1, và  $d$  kênh đầu vào. Quá trình tính toán của textCNN chủ yếu được chia thành các bước sau:

1. Định nghĩa nhiều hạt nhân tích chập một chiều để thực hiện các phép tính tích chập trên đầu vào. Những hạt nhân tích chập với độ rộng khác nhau có thể học được sự tương quan của các cụm từ liền kề với số lượng khác nhau.
2. Thực hiện gộp cực đại theo thời gian trên tất cả các kênh đầu ra, sau đó nối các giá trị gộp được của các kênh này thành một vector.
3. Vector nối trên sẽ được biến đổi thành đầu ra cho từng hạng mục bằng thông qua tầng kết nối đầy đủ. Tầng dropout có thể được sử dụng ở bước này để giải quyết tình trạng quá khớp.

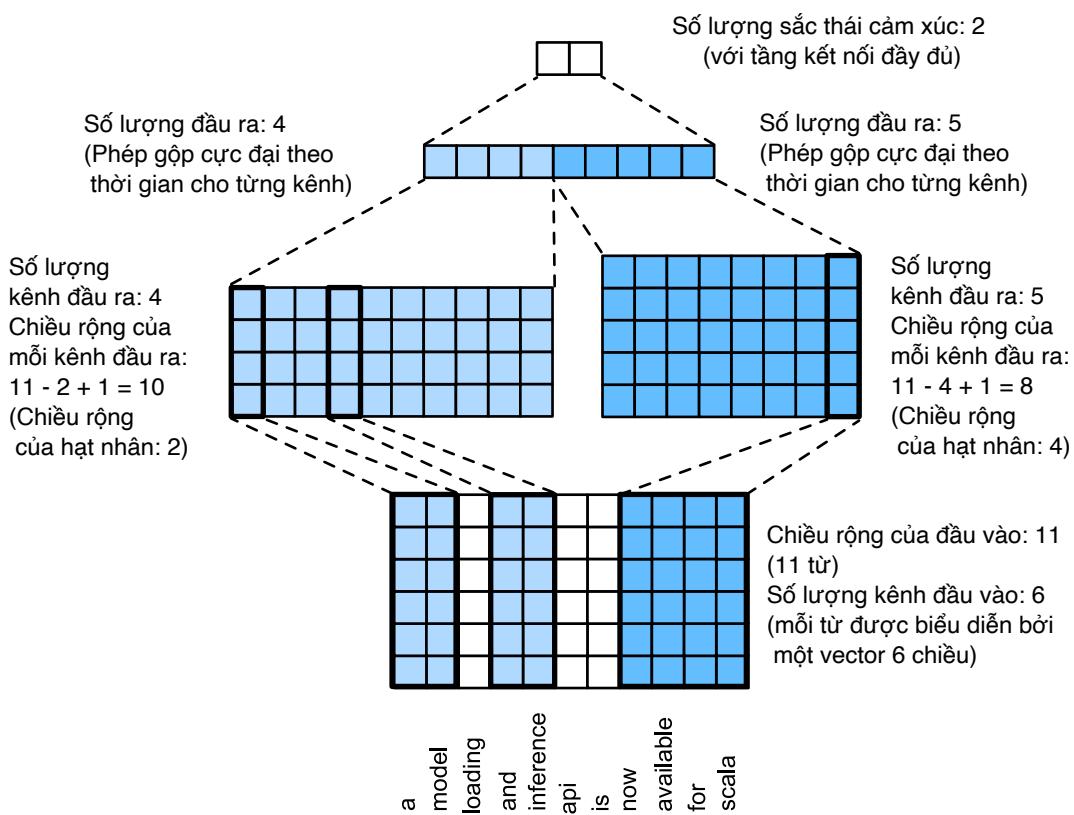


Fig. 17.3.5: Thiết kế TextCNN.

Fig. 17.3.5 minh họa một ví dụ cho textCNN. Đầu vào ở đây là một câu gồm 11 từ, với mỗi từ được biểu diễn bằng một vector từ 6 chiều. Vì vậy, câu đầu vào có độ rộng là 11 và số kênh đầu vào là 6. Chúng ta giả sử rằng 2 hạt nhân tích chập một chiều có độ rộng lần lượt là 2 và 4, tương ứng với số kênh đầu ra là 4 và 5. Sau khi áp dụng phép tính tích chập một chiều, đầu ra 4 kênh có chiều rộng là  $11 - 2 + 1 = 10$ , trong đó độ rộng của đầu ra 5 kênh còn lại là  $11 - 4 + 1 = 8$ . Thậm chí độ rộng của mỗi kênh có khác nhau đi nữa, chúng ta vẫn có thể thực hiện gộp cực đại theo thời gian cho mỗi kênh và nối đầu ra sau gộp của 9 kênh thành một vector 9 chiều. Cuối cùng, chúng ta dùng một tầng kết nối đầy đủ để biến đổi vector 9 chiều đó thành một đầu ra 2 chiều: dự đoán cảm xúc tích cực và cảm xúc tiêu cực.

Tiếp theo chúng ta bắt đầu lập trình mô hình textCNN. So với phần trước, ngoài việc thay mạng nơ-ron hồi tiếp bằng một tầng tích chập một chiều, ở đây chúng ta dùng 2 tầng embedding, một được giữ trọng số cố định và tầng còn lại tham gia quá trình huấn luyện.

```
class TextCNN(nn.Block):
    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 does not participate in training
        self.constant_embedding = nn.Embedding(vocab_size, embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Dense(2)
        # The max-over-time pooling layer has no weight, so it can share an
        # instance
        self.pool = nn.GlobalMaxPool1D()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.Sequential()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.add(nn.Conv1D(c, k, activation='relu'))

    def forward(self, inputs):
        # Concatenate the output of two embedding layers with shape of
        # (batch size, no. of words, word vector dimension) by word vector
        embeddings = np.concatenate([
            self.embedding(inputs), self.constant_embedding(inputs)], axis=2)
        # According to the input format required by Conv1D, the word vector
        # dimension, that is, the channel dimension of the one-dimensional
        # convolutional layer, is transformed into the previous dimension
        embeddings = embeddings.transpose(0, 2, 1)
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, an ndarray with the shape of (batch size, channel size, 1)
        # can be obtained. Use the flatten function to remove the last
        # dimension and then concatenate on the channel dimension
        encoding = np.concatenate([
            np.squeeze(self.pool(conv(embeddings)), axis=-1)
            for conv in self.convs], axis=1)
        # After applying the dropout method, use a fully connected layer to
        # obtain the output
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Tạo một thực thể TextCNN có 3 tầng tích chập với chiều rộng hạt nhân là 3, 4 và 5, tất cả đều có 100 kênh đầu ra.

```
embed_size, kernel_sizes, num_channels = 100, [3, 4, 5], [100, 100, 100]
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, num_channels)
net.initialize(init.Xavier(), ctx=devices)
```

## Nạp Vector Từ đã được Tiền huấn luyện

Tương tự phần trước, ta nạp GloVe 100 chiều đã được tiền huấn luyện và khởi tạo các tầng embedding embedding và constant\_embedding. Ở đây, embedding sẽ tham gia quá trình huấn luyện trong khi constant\_embedding có trọng số cố định.

```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.set_data(embeds)
net.constant_embedding.weight.set_data(embeds)
net.constant_embedding.collect_params().setattr('grad_req', 'null')
```

## Huấn luyện và Đánh giá Mô hình

Bây giờ ta có thể huấn luyện mô hình.

```
lr, num_epochs = 0.001, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

Dưới đây, ta sử dụng mô hình đã được huấn luyện để phân loại cảm xúc của hai câu đơn giản.

```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

### 17.3.4 Tóm tắt

- Ta có thể dùng tích chập một chiều để xử lý và phân tích dữ liệu theo thời gian.
- Phép tương quan chéo một chiều đa kênh đầu vào có thể xem như phép tương quan chéo hai chiều đơn kênh đầu vào.
- Đầu vào của tầng gộp cực đại theo thời gian có thể có số bước thời gian trên mỗi kênh khác nhau.
- TextCNN chủ yếu sử dụng một tầng chập một chiều và một tầng gộp cực đại theo thời gian.

### 17.3.5 Bài tập

1. Điều chỉnh các tham số mô hình và so sánh hai phương pháp phân tích cảm xúc giữa mạng nơ-ron truy hồi và mạng nơ-ron tích chập, xét trên khía cạnh độ chính xác và hiệu suất tính toán.
2. Bạn có thể cải thiện thêm độ chính xác của mô hình trên tập kiểm tra thông qua việc sử dụng ba phương pháp đã được giới thiệu ở phần trước: điều chỉnh các tham số mô hình, sử dụng các vector từ tiền huấn luyện lớn hơn, và sử dụng công cụ token hóa từ spaCy.
3. Bạn còn có thể sử dụng TextCNN cho những tác vụ xử lý ngôn ngữ tự nhiên nào khác?

### 17.3.6 Thảo luận

- Tiếng Anh: MXNet<sup>359</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>360</sup>

### 17.3.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trương Lộc Phát
- Nguyễn Văn Quang
- Lý Phi Long
- Nguyễn Mai Hoàng Long
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 20/09/2020)

## 17.4 Suy luận ngôn ngữ tự nhiên và Tập dữ liệu

Trong Section 17.1, chúng ta đã thảo luận về bài toán phân tích sắc thái cảm xúc (*sentiment analysis*). Mục đích của bài toán là phân loại một chuỗi văn bản vào các hạng mục đã định trước, chẳng hạn như các sắc thái đối lập. Tuy nhiên, trong trường hợp cần xác định liệu một câu có thể suy ra được từ một câu khác không, hoặc khi cần loại bỏ sự dư thừa bằng việc xác định các câu tương đương về ngữ nghĩa thì việc phân lớp một chuỗi văn bản là không đủ. Thay vào đó ta cần khả năng suy luận trên các cặp chuỗi văn bản.

### 17.4.1 Suy luận Ngôn ngữ Tự nhiên

Suy luận ngôn ngữ tự nhiên nghiên cứu liệu một giả thuyết (*hypothesis*) có thể được suy ra được từ một tiền đề (*premise*) không, cả hai đều là chuỗi văn bản. Nói cách khác, suy luận ngôn ngữ tự nhiên quyết định mối quan hệ logic giữa một cặp chuỗi văn bản. Các mối quan hệ đó thường rơi vào một trong ba loại sau đây:

- *Kéo theo*: giả thuyết có thể suy ra được từ tiền đề.
- *Đối lập*: phủ định của giả thuyết có thể suy ra được từ tiền đề.
- *Trung tính*: tất cả các trường hợp khác.

<sup>359</sup> <https://discuss.d2l.ai/t/393>

<sup>360</sup> <https://forum.machinelearningcoban.com/c/d2l>

Suy luận ngôn ngữ tự nhiên còn được gọi là bài toán nhận dạng quan hệ kéo theo trong văn bản. Ví dụ, cặp sau được gán nhãn là *kéo theo* bởi vì “thể hiện tình cảm” trong giả thuyết có thể được suy ra từ “ôm nhau” trong tiền đề.

Tiền đề: Hai người đang ôm nhau.

Giả thuyết: Hai người đang thể hiện tình cảm.

Sau đây là một ví dụ về *đối lập*, vì “chạy đoạn mã ví dụ” cho biết “không ngủ” chứ không phải “ngủ”.

Tiền đề: Một bạn đang chạy đoạn mã ví dụ trong Đắm mình vào học sâu.

Giả thuyết: Bạn đó đang ngủ.

Ví dụ thứ ba cho thấy mối quan hệ *trung tính* vì cả “nổi tiếng” và “không nổi tiếng” đều không thể được suy ra từ thực tế là “đang biểu diễn cho chúng tôi”.

Tiền đề: Các nhạc công đang biểu diễn cho chúng tôi.

Giả thuyết: Các nhạc công rất nổi tiếng.

Suy luận ngôn ngữ tự nhiên là một chủ đề trung tâm trong việc hiểu ngôn ngữ tự nhiên. Nó có nhiều ứng dụng khác nhau, từ truy xuất thông tin đến hỏi đáp trong miền mở. Để nghiên cứu bài toán này, chúng ta sẽ bắt đầu bằng việc tìm hiểu một tập dữ liệu đánh giá xếp hạng phổ biến trong suy luận ngôn ngữ tự nhiên.

#### 17.4.2 Tập dữ liệu Suy luận ngôn ngữ tự nhiên của Stanford (SNLI)

Tập dữ liệu ngôn ngữ tự nhiên của Stanford (SNLI) là một tập hợp gồm hơn 500,000 cặp câu Tiếng Anh được gán nhãn (Bowman et al., 2015). Ta tải xuống và giải nén tập dữ liệu SNLI trong đường dẫn `./data/snli_1.0`.

```
import collections
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
import os
import re
import zipfile

npx.set_np()

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')

data_dir = d2l.download_extract('SNLI')
```

## Đọc tập Dữ liệu

Tập dữ liệu SNLI gốc chứa thông tin phong phú hơn những gì thực sự cần cho thí nghiệm của chúng ta. Vì thế, ta định nghĩa một hàm `read_snli` để trích xuất một phần của tập dữ liệu, rồi trả về các danh sách tiền đề, 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 ta in 3 cặp tiền đề và giả thuyết đầu tiên cũng như nhãn của chúng (“0”, “1”, và “2” tương ứng với “kéo theo”, “đối lập”, và “trung tính”).

```
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)
```

Tập huấn luyện có khoảng 550,000 cặp, và tập kiểm tra có khoảng 10,000 cặp. Đoạn mã dưới đây cho thấy rằng ba nhãn “kéo theo”, “đối lập”, và “trung tính” cân bằng trong cả hai tập huấn luyện và tập kiểm tra.

```
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)])
```

## Định nghĩa Lớp để nạp Tập dữ liệu

Dưới đây ta định nghĩa một lớp để nạp tập dữ liệu SNLI bằng cách kế thừa lớp `Dataset` trong Gluon. Đối số `num_steps` trong phương thức khởi tạo chỉ định độ dài chuỗi văn bản, do đó mỗi minibatch sẽ có cùng kích thước. Nói cách khác, các token phía sau `num_steps` token đầu tiên ở trong chuỗi dài hơn sẽ được loại bỏ, trong khi token đặc biệt “`<pad>`” sẽ được nối thêm vào các chuỗi ngắn hơn đến khi độ dài của chúng bằng `num_steps`. Bằng cách lặp trình hàm `__getitem__`, ta có thể truy cập vào các tiền đề, giả thuyết và nhãn bất kỳ với chỉ số `idx`.

```

#@save
class SNLIDataset(gluon.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 = np.array(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples')

    def _pad(self, lines):
        return np.array([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)

```

## Kết hợp tất cả lại

Bây giờ ta có thể gọi hàm `read_snli` và lớp `SNLIDataset` để tải xuống tập dữ liệu SNLI và trả về thực thể `DataLoader` cho cả hai tập huấn luyện và tập kiểm tra, cùng với bộ từ vựng của tập huấn luyện. Lưu ý rằng ta phải sử dụng bộ từ vựng được xây dựng từ tập huấn luyện cho tập kiểm tra. Như vậy, mô hình được huấn luyện trên tập huấn luyện sẽ không biết bất kỳ token mới nào từ tập kiểm tra nếu có.

```

#@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 = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                       num_workers=num_workers)
    test_iter = gluon.data.DataLoader(test_set, batch_size, shuffle=False,
                                      num_workers=num_workers)
    return train_iter, test_iter, train_set.vocab

```

Ở đây ta đặt kích thước batch là 128 và độ dài chuỗi là 50, và gọi hàm `load_data_snli` để lấy iterator dữ liệu và bộ từ vựng. Sau đó ta in kích thước của bộ từ vựng.

```
train_iter, test_iter, vocab = load_data_snli(128, 50)
len(vocab)
```

Bây giờ ta in kích thước của minibatch đầu tiên. Trái với phân tích sắc thái cảm xúc, ta có 2 đầu vào  $X[0]$  và  $X[1]$  biểu diễn cặt tiền đề và giả thuyết.

```
for X, Y in train_iter:
    print(X[0].shape)
    print(X[1].shape)
    print(Y.shape)
    break
```

#### 17.4.3 Tóm tắt

- Suy luận ngôn ngữ tự nhiên nghiên cứu liệu một giả thuyết có thể được suy ra từ một tiền đề hay không, khi cả hai đều là chuỗi văn bản.
- Trong suy luận ngôn ngữ tự nhiên, mối quan hệ giữa tiền đề và giả thuyết bao gồm kéo theo, đối lập và trung tính.
- Bộ dữ liệu suy luận ngôn ngữ tự nhiên Stanford (SNLI) là một tập dữ liệu đánh giá xếp hạng phổ biến cho suy luận ngôn ngữ tự nhiên.

#### 17.4.4 Bài tập

1. Dịch máy từ lâu nay vẫn được đánh giá bằng sự trùng lặp bề ngoài giữa các  $n$ -gram của bản dịch đầu ra và bản dịch nhãn gốc. Bạn có thể thiết kế một phép đo để đánh giá kết quả dịch máy bằng cách sử dụng suy luận ngôn ngữ tự nhiên không?
2. Thay đổi siêu tham số như thế nào để giảm kích thước bộ từ vựng?

#### 17.4.5 Thảo luận

- Tiếng Anh: Main Forum<sup>361</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>362</sup>

#### 17.4.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Thái Bình
- Lê Khắc Hồng Phúc
- Trần Yến Thy

<sup>361</sup> <https://discuss.d2l.ai/t/394>

<sup>362</sup> <https://forum.machinelearningcoban.com/c/d2l>

- Phạm Minh Đức
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

## 17.5 Suy luận Ngôn ngữ Tự nhiên: Sử dụng Cơ chế Tập trung

Chúng tôi đã giới thiệu tác vụ suy luận ngôn ngữ tự nhiên và tập dữ liệu SNLI trong Section 17.4. Trong nhiều mô hình dựa trên các kiến trúc sâu và phức tạp, Parikh và các cộng sự đề xuất hướng giải quyết bài toán suy luận ngôn ngữ tự nhiên bằng cơ chế tập trung và gọi nó là một “mô hình tập trung có thể phân tách” (*decomposable attention model*) (Parikh et al., 2016). Điều này dẫn đến một mô hình không có các tầng truy hồi hay tích chập, nhưng đạt được kết quả tốt nhất vào thời điểm đó trên tập dữ liệu SNLI với lượng tham số ít hơn nhiều. Trong phần này, chúng tôi sẽ mô tả và lập trình phương pháp dựa trên cơ chế tập trung (cùng với MLP) để suy luận ngôn ngữ tự nhiên, như minh họa trong Fig. 17.5.1.

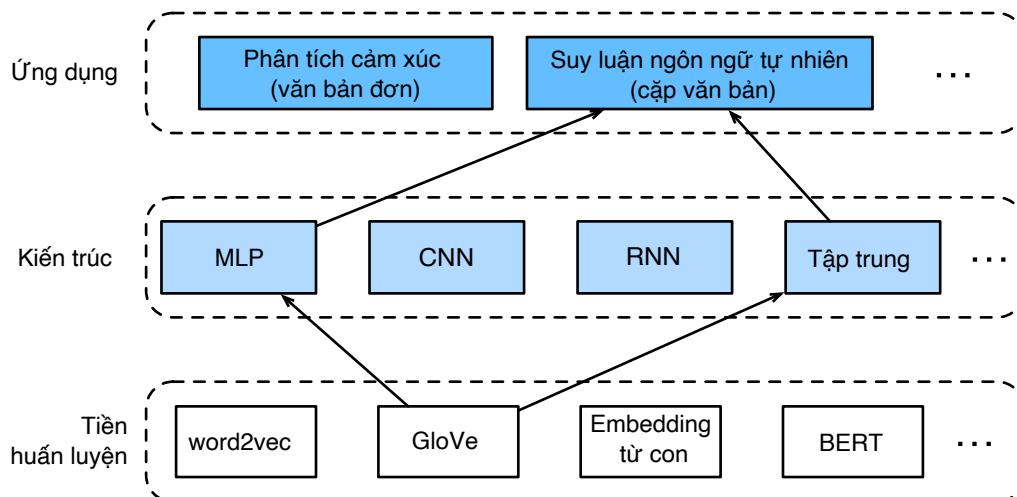


Fig. 17.5.1: Mục này truyền Glove tiền huấn luyện vào kiến trúc tập trung và MLPs để suy diễn ngôn ngữ tự nhiên.

### 17.5.1 Mô hình

Đơn giản hơn so với việc duy trì thứ tự của các từ trong các tiền đề và giả thuyết, ta có thể căn chỉnh các từ trong một chuỗi văn bản với mọi từ trong chuỗi khác và ngược lại, rồi so sánh và kết hợp các thông tin đó để dự đoán mối quan hệ logic giữa tiền đề và giả thuyết. Tương tự như việc căn chỉnh các từ giữa câu nguồn và đích trong dịch máy, việc căn chỉnh các từ giữa tiền đề và giả thuyết có thể được thực hiện nhanh gọn nhờ cơ chế tập trung.

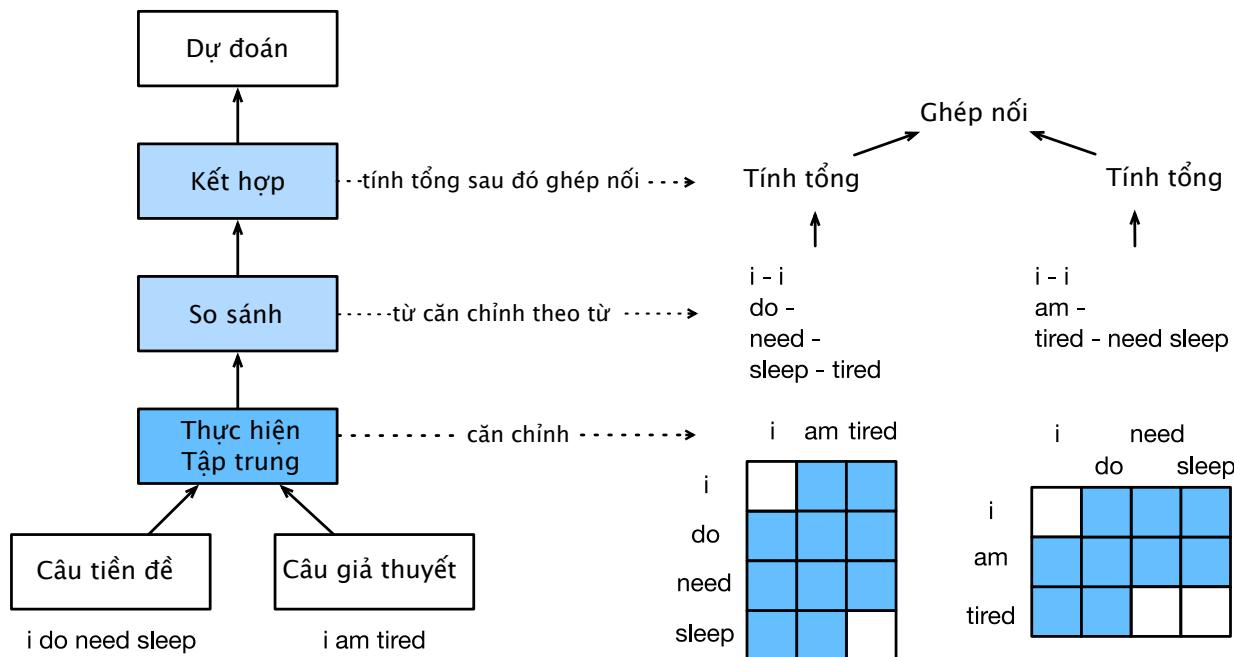


Fig. 17.5.2: Suy luận ngôn ngữ tự nhiên sử dụng cơ chế tập trung.

Fig. 17.5.2 minh họa phương pháp suy luận ngôn ngữ tự nhiên sử dụng cơ chế tập trung. Ở mức cao, nó bao gồm ba bước huấn luyện phối hợp: thực hiện tập trung, so sánh, và kết hợp. Ta sẽ từng bước mô tả chúng trong phần tiếp theo.

```
from d2l import mxnet as d2l
import mxnet as mx
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn

npx.set_np()
```

### Thực hiện Tập trung

Bước đầu tiên là phải căn chỉnh các từ trong một chuỗi văn bản với một chuỗi khác. Giả sử câu tiền đề là “i do need sleep” và câu giả thuyết là “i am tired”. Do sự tương đồng về ngữ nghĩa, ta mong muốn căn chỉnh “i” trong câu giả thuyết với “i” trong câu tiền đề, và căn chỉnh “tired” trong câu giả thuyết với “sleep” trong câu tiền đề. Tương tự, ta muốn căn chỉnh “i” trong câu tiền đề với “i” trong câu giả thuyết, và căn chỉnh “need” và “sleep” trong câu tiền đề với “tired” trong câu giả thuyết. Lưu ý là sự căn chỉnh này là *mềm*, sử dụng trung bình có trọng số, trong đó các trọng số nên có độ lớn hợp lý ứng với các từ được căn chỉnh. Để dễ dàng cho việc minh họa, Fig. 17.5.2 diễn tả sự căn chỉnh này theo cách *cứng*.

Bây giờ ta mô tả sự căn chỉnh mềm sử dụng cơ chế tập trung chi tiết hơn. Ký hiệu  $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_m)$  và  $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_n)$  là câu tiền đề và câu giả thuyết, với số từ lần lượt là  $m$  và  $n$ . Ở đây  $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$  ( $i = 1, \dots, m, j = 1, \dots, n$ ) là một vector embedding từ  $d$ -chiều. Để căn chỉnh mềm, ta tính trọng số tập trung  $e_{ij} \in \mathbb{R}$  như sau

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j), \quad (17.5.1)$$

ở đây hàm  $f$  là một MLP được định nghĩa theo hàm `mlp`. Chiều đầu ra của  $f$  được thiết lập bởi đối số `num_hiddens` của hàm `mlp`.

```
def mlp(num_hiddens, flatten):
    net = nn.Sequential()
    net.add(nn.Dropout(0.2))
    net.add(nn.Dense(num_hiddens, activation='relu', flatten=flatten))
    net.add(nn.Dropout(0.2))
    net.add(nn.Dense(num_hiddens, activation='relu', flatten=flatten))
    return net
```

Cũng nên chú ý rằng, trong (17.5.1)  $f$  nhận hai đầu vào  $\mathbf{a}_i$  và  $\mathbf{b}_j$  riêng biệt thay vì nhận cả cặp làm đầu vào. Thủ thuật *phân tách* này dẫn tới việc chỉ có  $m + n$  lần tính (độ phức tạp tuyến tính)  $f$  thay vì  $mn$  (độ phức tạp bậc hai).

Thực hiện chuẩn hóa các trọng số tập trung trong (17.5.1), ta tính trung bình có trọng số của tất cả các embedding từ trong câu giả thuyết để thu được biểu diễn của câu giả thuyết được căn chỉnh mềm với từ được đánh chỉ số  $i$  trong câu tiền đề:

$$\beta_i = \sum_{j=1}^n \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})} \mathbf{b}_j. \quad (17.5.2)$$

Tương tự, ta tính sự căn chỉnh mềm của các từ trong câu tiền đề cho mỗi từ được đánh chỉ số  $j$  trong câu giả thuyết:

$$\alpha_j = \sum_{i=1}^m \frac{\exp(e_{ij})}{\sum_{k=1}^m \exp(e_{kj})} \mathbf{a}_i. \quad (17.5.3)$$

Dưới đây ta định nghĩa lớp `Attend` để tính sự căn chỉnh mềm của các câu giả thuyết (beta) với các câu tiền đề đầu vào A và sự căn chỉnh mềm của các câu tiền đề (alpha) với các câu giả thuyết B.

```
class Attend(nn.Block):
    def __init__(self, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_hiddens=num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (batch_size, no. of words in sequence A/B,
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (batch_size, no. of words in sequence A/B,
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (batch_size, no. of words in sequence A,
        # no. of words in sequence B)
```

(continues on next page)

```

e = npx.batch_dot(f_A, f_B, transpose_b=True)
# Shape of 'beta': ('batch_size', no. of words in sequence A,
# `embed_size`), where sequence B is softly aligned with each word
# (axis 1 of `beta`) in sequence A
beta = npx.batch_dot(npx.softmax(e), B)
# Shape of 'alpha': ('batch_size', no. of words in sequence B,
# `embed_size`), where sequence A is softly aligned with each word
# (axis 1 of `alpha`) in sequence B
alpha = npx.batch_dot(npx.softmax(e.transpose(0, 2, 1)), A)
return beta, alpha

```

## So sánh

Bước tiếp theo, chúng ta so sánh một từ trong chuỗi với chuỗi khác được căn chỉnh mềm với từ đó.

Lưu ý rằng trong “căn chỉnh mềm”, tất cả từ đều đến từ một chuỗi, tuy nhiên do có những trọng số tập trung khác nhau, chúng sẽ được so sánh với một từ trong chuỗi khác. Để dễ minh họa, Fig. 17.5.2 ghép đôi từ với các từ được căn chỉnh *cùng*. Ví dụ, giả sử bước tập trung xác định rằng “need” và “sleep” trong câu tiền đề đều được căn chỉnh với “tired” trong câu giả thuyết, thì cặp “tired-need sleep” sẽ được so sánh.

Tại bước so sánh, chúng ta đưa những từ đã được ghép nối (toán tử  $[ \cdot, \cdot ]$ ) và những từ đã căn chỉnh của chuỗi còn lại vào hàm  $g$  (một MLP):

$$\begin{aligned} \mathbf{v}_{A,i} &= g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \dots, m \\ \mathbf{v}_{B,j} &= g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \dots, n. \end{aligned} \quad (17.5.4)$$

Trong (17.5.4),  $\mathbf{v}_{A,i}$  là phép so sánh giữa từ thứ  $i$  của câu tiền đề và tất cả các từ trong câu giả thuyết được căn chỉnh mềm với từ thứ  $i$ ; trong khi  $\mathbf{v}_{B,j}$  lại là phép so sánh giữa từ thứ  $j$  trong câu giả thuyết và tất cả từ trong câu tiền đề được căn chỉnh mềm với từ thứ  $j$ . Lớp Compare sau định nghĩa bước so sánh này.

```

class Compare(nn.Block):
    def __init__(self, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_hiddens=num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(np.concatenate([A, beta], axis=2))
        V_B = self.g(np.concatenate([B, alpha], axis=2))
        return V_A, V_B

```

## Tổng hợp

Với hai tập vector so sánh  $\mathbf{v}_{A,i}$  ( $i = 1, \dots, m$ ) và  $\mathbf{v}_{B,j}$  ( $j = 1, \dots, n$ ) trong tay, ta sẽ tổng hợp các thông tin đó để suy ra mối quan hệ logic tại bước cuối cùng. Chúng ta bắt đầu bằng cách lấy tổng trên cả hai tập:

$$\mathbf{v}_A = \sum_{i=1}^m \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^n \mathbf{v}_{B,j}. \quad (17.5.5)$$

Tiếp theo, chúng ta ghép nối hai kết quả tổng rồi đưa vào hàm  $h$  (một MLP) để thu được kết quả phân loại của mối quan hệ logic:

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]). \quad (17.5.6)$$

Bước tổng hợp được định nghĩa trong lớp Aggregate sau đây.

```
class Aggregate(nn.Block):
    def __init__(self, num_hiddens, num_outputs, **kwargs):
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_hiddens=num_hiddens, flatten=True)
        self.h.add(nn.Dense(num_outputs))

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
        V_A = V_A.sum(axis=1)
        V_B = V_B.sum(axis=1)
        # Feed the concatenation of both summarization results into an MLP
        Y_hat = self.h(np.concatenate([V_A, V_B], axis=1))
        return Y_hat
```

## Kết hợp tất cả lại

Bằng cách gộp các bước thực hiện tập trung, so sánh và tổng hợp lại với nhau, ta định nghĩa mô hình tập trung có thể phân tách để cùng huấn luyện cả ba bước này.

```
class DecomposableAttention(nn.Block):
    def __init__(self, vocab, embed_size, num_hiddens, **kwargs):
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_hiddens)
        self.compare = Compare(num_hiddens)
        # There are 3 possible outputs: entailment, contradiction, and neutral
        self.aggregate = Aggregate(num_hiddens, 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
```

## 17.5.2 Huấn luyện và Đánh giá Mô hình

Bây giờ ta sẽ huấn luyện và đánh giá mô hình tập trung có thể phân tách vừa được định nghĩa trên tập dữ liệu SNLI. Ta bắt đầu bằng việc đọc tập dữ liệu.

### Đọc tập dữ liệu

Ta tải xuống và đọc tập dữ liệu SNLI bằng hàm định nghĩa trong Section 17.4. Kích thước batch và độ dài chuỗi được đặt là 256 và 50.

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size, num_steps)
```

### Tạo mô hình

Ta sử dụng embedding GloVe 100-chiều đã tiền huấn luyện để biểu diễn các token đầu vào. Do đó, ta định nghĩa trước chiều của các vector  $\mathbf{a}_i$  và  $\mathbf{b}_j$  trong (17.5.1) là 100. Chiều đầu ra của hàm  $f$  trong (17.5.1) và  $g$  trong (17.5.4) được đặt bằng 200. Sau đó ta tạo thực thể của mô hình, khởi tạo tham số, và nạp embedding GloVe để khởi tạo các vector token đầu vào.

```
embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus()
net = DecomposableAttention(vocab, embed_size, num_hiddens)
net.initialize(init.Xavier(), ctx=devices)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.set_data(embeds)
```

### Huấn luyện và Đánh giá Mô hình

Trái ngược với hàm `split_batch` trong Section 14.5 nhận đầu vào đơn như một chuỗi văn bản (hoặc ảnh) chẳng hạn, ta định nghĩa hàm `split_batch_multi_inputs` để nhận đa đầu vào, ví dụ như cặp tiền đề và giả thuyết ở trong các minibatch.

```
#@save
def split_batch_multi_inputs(X, y, devices):
    """Split multi-input `X` and `y` into multiple devices."""
    X = list(zip(*[gluon.utils.split_and_load(
        feature, devices, even_split=False) for feature in X]))
    return (X, gluon.utils.split_and_load(y, devices, even_split=False))
```

Giờ ta có thể huấn luyện và đánh giá mô hình trên tập dữ liệu SNLI.

```
lr, num_epochs = 0.001, 4
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices,
               split_batch_multi_inputs)
```

## Sử dụng Mô hình

Cuối cùng, ta định nghĩa hàm dự đoán để xuất ra mối quan hệ logic giữa cặp tiền đề và giả thuyết.

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    premise = np.array(vocab[premise], ctx=d2l.try_gpu())
    hypothesis = np.array(vocab[hypothesis], ctx=d2l.try_gpu())
    label = np.argmax(net([premise.reshape((1, -1)),
                           hypothesis.reshape((1, -1))]), axis=1)
    return 'entailment' if label == 0 else 'contradiction' if label == 1 \
        else 'neutral'
```

Ta có thể sử dụng mô hình đã huấn luyện để thu được kết quả suy luận ngôn ngữ tự nhiên cho các cặp câu mẫu.

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'is', 'bad', '.'])
```

### 17.5.3 Tóm tắt

- Mô hình tập trung có thể phân tách bao gồm 3 bước để dự đoán mối quan hệ logic giữa cặp tiền đề và giả thuyết: thực hiện tập trung, so sánh và tổng hợp.
- Với cơ chế tập trung, ta có thể căn chỉnh các từ trong một chuỗi văn bản với tất cả các từ trong chuỗi văn bản còn lại, và ngược lại. Đây là kỹ thuật căn chỉnh mềm, sử dụng trung bình có trọng số, trong đó các trọng số có độ lớn hợp lý được gán với các từ sẽ được căn chỉnh.
- Thủ thuật phân tách tầng tập trung giúp giảm độ phức tạp thành tuyến tính thay vì là bậc hai khi tính toán trọng số tập trung.
- Ta có thể sử dụng embedding từ đã tiền huấn luyện để biểu diễn đầu vào cho các tác vụ xử lý ngôn ngữ tự nhiên xuôi dòng, ví dụ như suy luận ngôn ngữ tự nhiên.

### 17.5.4 Bài tập

- Huấn luyện mô hình với các tập siêu tham số khác nhau. Bạn có thể thu được độ chính xác cao hơn trên tập kiểm tra không?
- Những điểm hạn chế chính của mô hình tập trung kết hợp đối với suy luận ngôn ngữ tự nhiên là gì?
- Giả sử ta muốn tính độ tương tự ngữ nghĩa (một giá trị liên tục trong khoảng 0 và 1) cho một cặp câu bất kỳ. Ta sẽ thu thập và gán nhãn tập dữ liệu như thế nào? Bạn có thể thiết kế một mô hình với cơ chế tập trung không?

### 17.5.5 Thảo luận

- Tiếng Anh: MXNet<sup>363</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>364</sup>

### 17.5.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thành
- Nguyễn Mai Hoàng Long
- Phạm Đăng Khoa
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 19/09/2020)

## 17.6 Tinh chỉnh BERT cho các Ứng dụng Cấp Chuỗi và Cấp Token

Trong các phần trước, ta đã thiết kế các mô hình khác nhau cho các ứng dụng xử lý ngôn ngữ tự nhiên, dựa trên RNN, CNN, MLP và cơ chế tập trung. Những mô hình này rất hữu ích khi mô hình bị giới hạn về không gian bộ nhớ hoặc thời gian thực thi; tuy nhiên, việc thiết kế thủ công một mô hình cụ thể cho mọi tác vụ xử lý ngôn ngữ tự nhiên trong thực tế là điều không khả thi. Trong Section 16.8, chúng ta đã giới thiệu mô hình BERT được tiền huấn luyện mà chỉ yêu cầu thay đổi kiến trúc tối thiểu cho một loạt các tác vụ xử lý ngôn ngữ tự nhiên. Một mặt, tại thời điểm BERT được đề xuất, nó đã cải thiện kết quả tốt nhất trên các tác vụ xử lý ngôn ngữ tự nhiên khác nhau. Mặt khác, như đã lưu ý trong Section 16.10, hai phiên bản của mô hình BERT gốc lần lượt có 110 triệu và 340 triệu tham số. Do đó, khi có đủ tài nguyên tính toán, ta có thể xem xét việc tinh chỉnh BERT cho các ứng dụng xử lý ngôn ngữ tự nhiên xuôi dòng.

Sau đây, ta sẽ tổng quát hóa một số ứng dụng xử lý ngôn ngữ tự nhiên thành các ứng dụng cấp độ chuỗi và cấp độ token. Ở cấp độ chuỗi, chúng tôi sẽ giới thiệu cách chuyển đổi biểu diễn BERT của văn bản đầu vào thành nhãn đầu ra trong các tác vụ phân loại văn bản đơn và phân loại hay hồi quy cặp văn bản. Ở cấp độ token, chúng tôi sẽ giới thiệu ngắn gọn các ứng dụng mới như gán thẻ văn bản và trả lời câu hỏi, từ đó làm sáng tỏ cách BERT biểu diễn đầu vào và biến đổi chúng thành nhãn đầu ra như thế nào. Trong quá trình tinh chỉnh, những “thay đổi kiến trúc tối thiểu” mà BERT yêu cầu trên các ứng dụng khác nhau là các tầng kết nối đầy đủ được bổ sung. Trong quá trình học có giám sát của một ứng dụng xuôi dòng, tham số của các tầng bổ sung này được học từ đầu trong khi tất cả các tham số trong mô hình BERT đã tiền huấn luyện sẽ được tinh chỉnh.

<sup>363</sup> <https://discuss.d2l.ai/t/395>

<sup>364</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 17.6.1 Phân loại Văn bản Đơn

Tác vụ *phân loại văn bản đơn* nhận một chuỗi văn bản đơn làm đầu vào và đầu ra là kết quả phân loại của văn bản đó. Bên cạnh tác vụ phân tích cảm xúc mà ta đã nghiên cứu trong chương này, tập dữ liệu CoLA (*Corpus of Linguistic Acceptability*) cũng được sử dụng cho tác vụ phân loại văn bản đơn, đánh giá xem một câu đã cho có chấp nhận được về mặt ngữ pháp hay không (Warstadt et al., 2019). Ví dụ, câu “I should study.” là chấp nhận được nhưng câu “I should studying.” thì không.

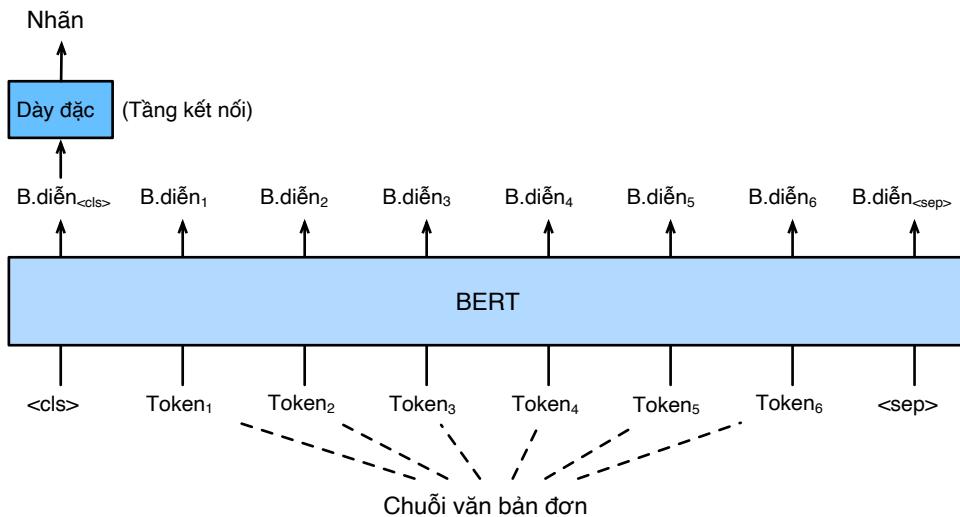


Fig. 17.6.1: Tinh chỉnh mô hình BERT cho các ứng dụng phân loại văn bản đơn, ví dụ như phân tích cảm xúc hay đánh giá khả năng chấp nhận được về ngôn ngữ học. Giả sử văn bản đơn đầu vào có sáu token.

Section 16.8 mô tả biểu diễn đầu vào của BERT. Chuỗi đầu vào BERT biểu diễn cả văn bản đơn và cặp văn bản một cách rạch ròi, trong đó token đặc biệt “<cls>” được sử dụng cho các tác vụ phân loại chuỗi, và token đặc biệt “<sep>” đánh dấu vị trí kết thúc của văn bản đơn hoặc vị trí phân tách cặp văn bản. Như minh họa trong Fig. 17.6.1, biểu diễn BERT của token đặc biệt “<cls>” mã hóa thông tin của toàn bộ chuỗi văn bản đầu vào trong các tác vụ phân loại văn bản đơn. Là biểu diễn của văn bản đầu vào đơn, vector này sẽ được truyền vào một mạng MLP nhỏ chứa các tầng kết nối đầy đủ để biến đổi thành phân phối của các giá trị nhãn rời rạc.

### 17.6.2 Phân loại hoặc Hồi quy Cặp Văn bản

Ta cũng sẽ xem xét tác vụ suy luận ngôn ngữ tự nhiên trong chương này. Tác vụ này nằm trong bài toán *phân loại cặp văn bản* (*text pair classification*).

Nhận một cặp văn bản làm đầu vào và cho ra một giá trị liên tục, đo độ tương tự ngữ nghĩa của văn bản (*semantic textual similarity*) là một tác vụ *hồi quy cặp văn bản* (*text pair regression*) rất phổ biến. Tác vụ này đo độ tương tự ngữ nghĩa của các câu đầu vào. Ví dụ, trong tập dữ liệu đánh giá độ tương tự ngữ nghĩa của văn bản (*Semantic Textual Similarity Benchmark*), độ tương tự của một cặp câu nằm trong khoảng từ 0 (không trùng lặp ngữ nghĩa) tới 5 (tương tự ngữ nghĩa) (Cer et al., 2017). Các mẫu dữ liệu trong tập dữ liệu này có dạng (câu thứ 1, câu thứ 2, độ tương tự):

- “A plane is taking off.”, “An air plane is taking off.”, 5.000;
- “A woman is eating something.”, “A woman is eating meat.”, 3.000;

- “A woman is dancing.”, “A man is talking.”, 0.000.

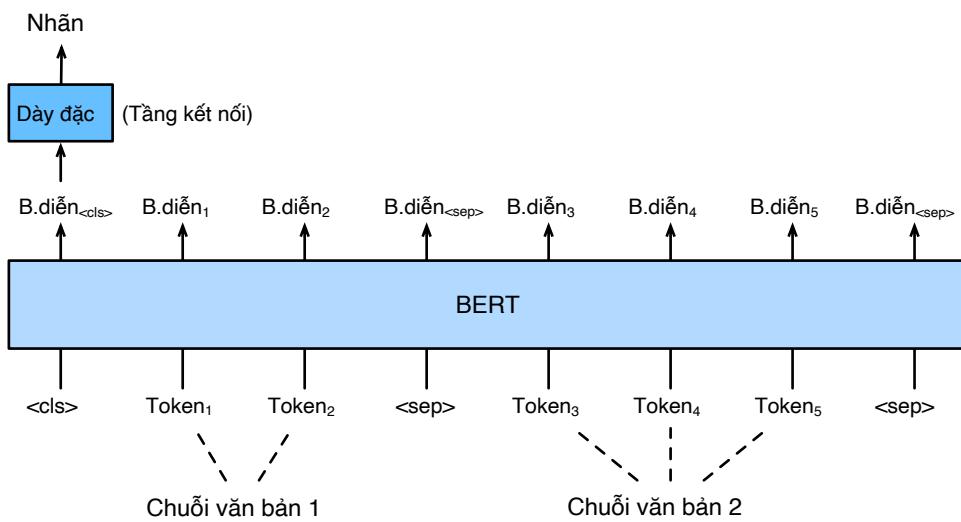


Fig. 17.6.2: Tinh chỉnh mô hình BERT cho các ứng dụng phân loại hoặc hồi quy cặp văn bản, ví dụ tác vụ suy luận ngôn ngữ tự nhiên và tác vụ đo độ tương tự ngữ nghĩa văn bản. Giả sử đầu cặp văn bản đầu vào có hai và ba token.

So với tác vụ phân loại văn bản đơn trong Fig. 17.6.1, việc tinh chỉnh BERT để phân loại cặp văn bản trong Fig. 17.6.2 có khác biệt trong biểu diễn đầu vào. Đối với các tác vụ hồi quy cặp văn bản, chẳng hạn như đo độ tương tự ngữ nghĩa văn bản, một vài thay đổi nhỏ có thể được áp dụng như xuất ra giá trị nhãn liên tục và sử dụng trung bình bình phương matsu.

### 17.6.3 Gán thẻ Văn bản

Bây giờ ta hãy xem xét các tác vụ ở mức token, ví dụ như *gán thẻ văn bản*, nơi mỗi token được gán một nhãn. Trong số các tác vụ gán thẻ văn bản, *gán thẻ từ loại* (*part-of-speech tagging*) gán cho mỗi từ một thẻ từ loại (ví dụ, tính từ hay danh từ) dựa vào vai trò của từ đó trong câu. Ví dụ, dựa vào tập thẻ Penn Treebank II, câu “John Smith’s car is new” nên được gán thẻ như “NNP (danh từ riêng, số ít) NNP POS (sở hữu cách) NN (danh từ, số ít hoặc nhiều) VB (động từ, động từ nguyên thể không”to“) JJ (tính từ)”.

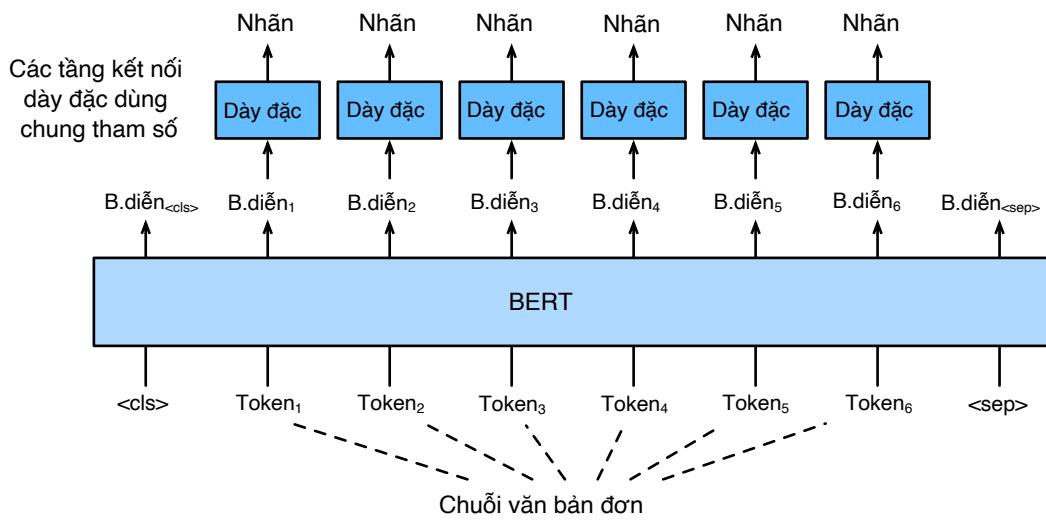


Fig. 17.6.3: Tinh chỉnh BERT cho ứng dụng gán thẻ văn bản, ví dụ như gán thẻ từ loại. Giả sử đầu vào là một văn bản đơn có sáu token.

Tinh chỉnh BERT cho ứng dụng gán thẻ văn bản được minh họa trong Fig. 17.6.3. So với Fig. 17.6.1, sự khác biệt duy nhất là biểu diễn BERT của *mỗi token* trong văn bản đầu vào được truyền vào cùng một mạng kết nối dày đủ bổ sung để đưa ra nhãn của các token, ví dụ như thẻ từ loại.

#### 17.6.4 Trả lời Câu hỏi

Là một ứng dụng khác ở mức token, *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 các đoạn văn và các câu hỏi, nơi mà câu trả lời cho mỗi câu hỏi chỉ là một phần văn bản (khoảng văn bản - *text span*) trong đoạn văn mà câu hỏi đang đề cập tới (Rajpurkar et al., 2016). Để giải thích, hãy xét đoạn văn sau “Một số chuyên gia cho rằng sự hiệu quả của khẩu trang là chưa thể khẳng định. Tuy nhiên, các nhà sản xuất khẩu trang cho rằng sản phẩm của họ, như là khẩu trang N95, có thể bảo vệ khỏi virus.” và câu hỏi “Ai cho rằng khẩu trang N95 có thể bảo vệ khỏi virus?”. Câu trả lời nên là khoảng văn bản “các nhà sản xuất khẩu trang” trong đoạn văn. Vì thế, mục đích trong SQuAD v1.1 là dự đoán điểm khởi đầu và kết thúc của khoảng văn bản trong đoạn văn, khi cho trước câu hỏi và đoạn văn.

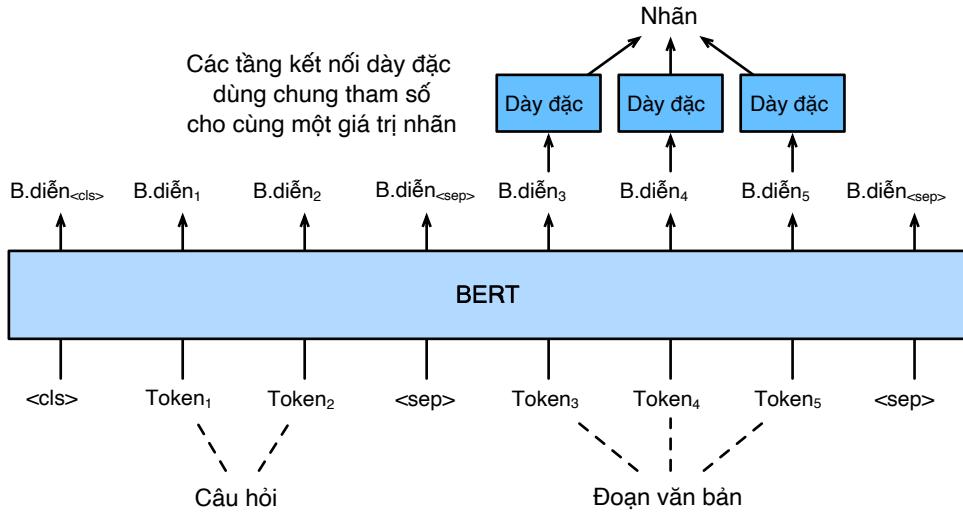


Fig. 17.6.4: Tinh chỉnh BERT cho trả lời câu hỏi. Giả sử rằng cặp văn bản đầu vào có hai và ba token.

Để tinh chỉnh BERT cho ứng dụng trả lời câu hỏi, câu hỏi và đoạn văn được đóng gói tương ứng lần lượt là chuỗi văn bản thứ nhất và thứ hai trong đầu vào của BERT. Để dự đoán vị trí của phần bắt đầu của khoảng văn bản, cùng một tầng kết nối đầy đủ được thêm vào sẽ chuyển hóa biểu diễn BERT của bất kỳ token nào từ đoạn văn bản có vị trí  $i$  thành một giá trị vô hướng  $s_i$ . Các giá trị vô hướng của tất cả token trong đoạn văn được tiếp tục biến đổi bởi hàm softmax trở thành một phân phối xác suất, dẫn tới mỗi vị trí  $i$  của token trong đoạn văn được gán cho một xác suất  $p_i$ , là xác suất token đó là điểm bắt đầu của khoảng văn bản. Dự đoán điểm kết thúc của khoảng văn bản cũng tương tự, ngoại trừ việc các tham số trong tầng kết nối đầy đủ mở rộng là độc lập với các tầng để dự đoán điểm bắt đầu. Khi dự đoán điểm kết thúc, token có vị trí  $i$  trong đoạn văn được biến đổi thành một giá trị vô hướng  $e_i$  bởi tầng kết nối đầy đủ. Fig. 17.6.4 minh họa quá trình tinh chỉnh BERT cho ứng dụng trả lời câu hỏi.

Cho việc trả lời câu hỏi, mục đích của huấn luyện có giám sát đơn giản là cực đại hóa hàm log hợp lý của các vị trí bắt đầu và kết thúc nhãn gốc. Khi dự đoán khoảng văn bản, ta có thể tính toán giá trị  $s_i + e_j$  cho một khoảng hợp lệ từ vị trí  $i$  tới vị trí  $j$  ( $i \leq j$ ), và đưa ra khoảng có giá trị cao nhất làm đầu ra.

### 17.6.5 Tóm tắt

- BERT chỉ yêu cầu các thay đổi tối thiểu tới kiến trúc (thêm các tầng kết nối đầy đủ) cho nhiều ứng dụng xử lý ngôn ngữ tự nhiên ở mức chuỗi và mức token, ví dụ như phân loại văn bản đơn (phân tích cảm xúc và kiểm tra khả năng chấp nhận được về ngôn ngữ), phân loại hoặc hồi quy cặp văn bản (suy luận ngôn ngữ tự nhiên và đo sự tương đồng ngữ nghĩa văn bản), gán thẻ văn bản (như gán thẻ từ loại) và trả lời câu hỏi.
- Trong suốt quá trình học có giám sát của một ứng dụng xuôi dòng, các thông số của các tầng mở rộng được học từ đầu trong khi tất cả các thông số trong mô hình BERT đã tiền huấn luyện sẽ được tinh chỉnh.

## 17.6.6 Bài tập

1. Hãy thiết kế một công cụ tìm kiếm các bài báo tin tức. Khi hệ thống nhận một truy vấn (ví dụ, “ngành công nghiệp dầu mỏ trong đại dịch COVID-19”), nó trả về một danh sách xếp hạng các bài viết tin tức liên quan tới truy vấn nhất. Giả sử như ta có một tập lớn các bài báo và một số lượng lớn các truy vấn. Để đơn giản hóa vấn đề, giả thiết rằng bài báo liên quan nhất được gán nhãn cho từng truy vấn. Làm cách nào để ta áp dụng phương pháp lấy mẫu âm (xem Section 16.2.1) và BERT khi thiết kế thuật toán?
2. Làm thế nào để tận dụng BERT khi huấn luyện các mô hình ngôn ngữ?
3. Làm thế nào để tận dụng BERT trong dịch máy?

## 17.6.7 Thảo luận

- Tiếng Anh: Main Forum<sup>365</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>366</sup>

## 17.6.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

## 17.7 Suy luận Ngôn ngữ Tự nhiên: Tinh chỉnh BERT

Ở các phần đầu của chương này, ta đã thiết kế một kiến trúc dựa trên cơ chế tập trung (trong Section 17.5) cho tác vụ suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI (như được mô tả trong Section 17.4). Bây giờ ta trở lại tác vụ này qua thực hiện tinh chỉnh BERT. Như đã thảo luận trong Section 17.6, suy luận ngôn ngữ tự nhiên là bài toán phân loại cặp văn bản ở cấp độ chuỗi, và việc tinh chỉnh BERT chỉ đòi hỏi thêm một kiến trúc bổ trợ dựa trên MLP, như minh họa trong Fig. 17.7.1.

<sup>365</sup> <https://discuss.d2l.ai/t/396>

<sup>366</sup> <https://forum.machinelearningcoban.com/c/d2l>

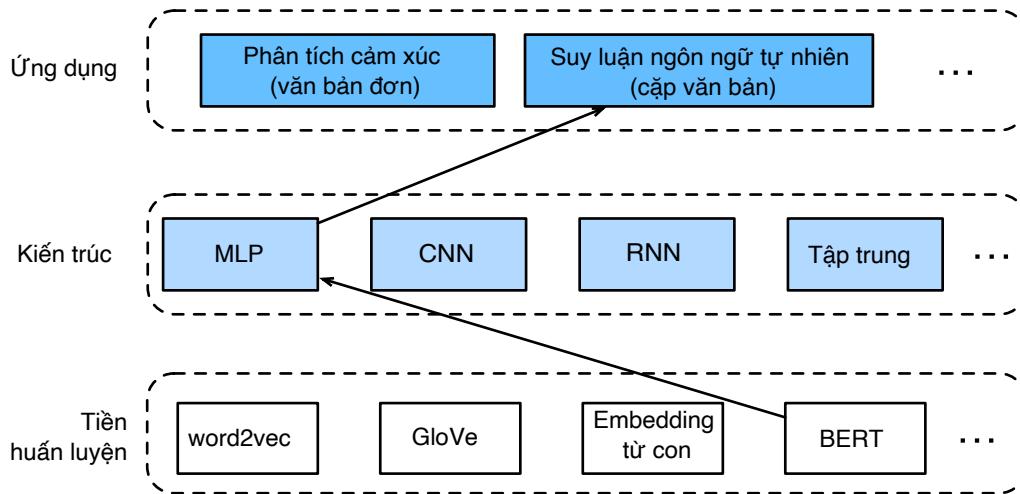


Fig. 17.7.1: Phần này truyền BERT đã tiền huấn luyện vào một kiến trúc dựa trên MLP cho suy luận ngôn ngữ tự nhiên.

Trong phần này, chúng ta sẽ tải một phiên bản BERT đã tiền huấn luyện kích thước nhỏ, rồi tinh chỉnh nó để suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI.

```
from d2l import mxnet as d2l
import json
import multiprocessing
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
import os

npx.set_np()
```

### 17.7.1 Nạp BERT đã Tiền huấn luyện

Chúng ta đã giải thích cách tiền huấn luyện BERT trên tập dữ liệu WikiText-2 trong Section 16.9 và Section 16.10 (lưu ý rằng mô hình BERT ban đầu được tiền huấn luyện trên các kho ngữ liệu lớn hơn nhiều). Ở thảo luận trong Section 16.10, mô hình BERT gốc có hàng trăm triệu tham số. Trong phần sau đây, chúng tôi cung cấp hai phiên bản BERT tiền huấn luyện: “bert.base” có kích thước xấp xỉ mô hình BERT cơ sở gốc, là mô hình đòi hỏi nhiều tài nguyên tính toán để tinh chỉnh, trong khi “bert.small” là phiên bản nhỏ để thuận tiện cho việc minh họa.

```
d2l.DATA_HUB['bert.base'] = (d2l.DATA_URL + 'bert.base.zip',
                             '7b3820b35da691042e5d34c0971ac3edbd80d3f4')
d2l.DATA_HUB['bert.small'] = (d2l.DATA_URL + 'bert.small.zip',
                             'a4e718a47137cccd1809c9107ab4f5edd317bae2c')
```

Cả hai mô hình BERT đã tiền huấn luyện đều chứa tập tin “vocab.json” định nghĩa tập từ vựng và tập tin “pretrained.params” chứa các tham số tiền huấn luyện. Ta thực hiện hàm `load_pretrained_model` sau đây để nạp các tham số đã tiền huấn luyện của BERT.

```
def load_pretrained_model(pretrained_model, num_hiddens, ffn_num_hiddens,
                          num_heads, num_layers, dropout, max_len, devices):
```

(continues on next page)

```

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, ffn_num_hiddens, num_heads,
                      num_layers, dropout, max_len)
# Load pretrained BERT parameters
bert.load_parameters(os.path.join(data_dir, 'pretrained.params'),
                      ctx=devices)
return bert, vocab

```

Để thuận tiện biểu diễn trên hầu hết các phần cứng, ta sẽ nạp và tinh chỉnh phiên bản nhỏ (“bert-small”) của BERT đã tiền huấn luyện ở phần này. Phần bài tập sẽ hướng dẫn cách tinh chỉnh mô hình “bert-base” lớn hơn nhiều, để cải thiện đáng kể độ chính xác khi kiểm tra.

```

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)

```

### 17.7.2 Tập dữ liệu để Tinh chỉnh BERT

Đối với tác vụ xuôi dòng suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI, ta định nghĩa một lớp SNLIBERTDataset là tập dữ liệu tuỳ biến. Trong mỗi mẫu, tiền đề và giả thuyết tạo thành một cặp chuỗi văn bản và được đóng gói thành một chuỗi đầu vào BERT như được mô tả trong Fig. 17.6.2. Nhắc lại Section 16.8.4, ID của các đoạn đó được sử dụng để phân biệt tiền đề và giả thuyết trong chuỗi đầu vào BERT. Với độ dài tối đa đã định trước của chuỗi đầu vào BERT (`max_len`), token cuối cùng của đoạn dài hơn trong cặp văn bản đầu vào sẽ liên tục bị xóa cho đến khi độ dài của nó là `max_len`. Để tăng tốc quá trình tạo tập dữ liệu SNLI cho việc tinh chỉnh BERT, ta sử dụng 4 tiến trình thợ để tạo ra các mẫu cho tập huấn luyện và tập kiểm tra một cách song song.

```

class SNLIBERTDataset(gluon.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 = np.array(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 = [

```

(continues on next page)

```

        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 (np.array(all_token_ids, dtype='int32'),
        np.array(all_segments, dtype='int32'),
        np.array(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):
        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, ta tạo các mẫu huấn luyện và kiểm tra bằng cách khởi tạo lớp SNLIBERTDataset. Các mẫu đó sẽ được đọc từ các minibatch trong quá trình huấn luyện và kiểm tra của 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 = gluon.data.DataLoader(train_set, batch_size, shuffle=True,
                                   num_workers=num_workers)
test_iter = gluon.data.DataLoader(test_set, batch_size,
                                  num_workers=num_workers)

```

### 17.7.3 Tinh chỉnh BERT

Như Fig. 17.6.2 đã chỉ ra, tinh chỉnh BERT trong suy luận ngôn ngữ tự nhiên chỉ yêu cầu thêm một perceptron đa tầng gồm hai tầng kết nối đầy đủ (xem self.hiised và self.output trong lớp BERTClassifier bên dưới). Perceptron đa tầng này biến đổi biểu diễn BERT của token đặc biệt “<cls>”, là token mã hóa thông tin của cả tiền đề và giả thuyết, thành ba đầu ra của suy luận ngôn ngữ tự nhiên: kéo theo, đối lập và trung tính.

```
class BERTClassifier(nn.Block):
    def __init__(self, bert):
        super(BERTClassifier, self).__init__()
        self.encoder = bert.encoder
        self.hidden = bert.hidden
        self.output = nn.Dense(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 đã tiền huấn luyện bert được đưa vào thực thể net của lớp BERTClassifier cho tác vụ xuôi dòng. Thông thường khi lập trình tinh chỉnh BERT, chỉ các tham số của tầng đầu ra của perception đa tầng bổ sung (net.output) mới được học từ đầu. Còn tất cả các tham số của bộ mã hóa BERT đã tiền huấn luyện (net.encoder) và tầng ẩn của perception đa tầng bổ sung (net.hidden) thì sẽ được tinh chỉnh.

```
net = BERTClassifier(bert)
net.output.initialize(ctx=devices)
```

Nhớ lại rằng trong Section 16.8, cả 2 lớp MaskLM và lớp NextSentencePred đều có các tham số của perceptron đa tầng mà chúng sử dụng. Các tham số này là một phần của các tham số trong mô hình BERT đã tiền huấn luyện bert, và do đó là một phần của các tham số trong net. Tuy nhiên, các tham số này chỉ được dùng để tính toán mất mát của mô hình ngôn ngữ có mặt nạ và mất mát khi dự đoán câu tiếp theo trong quá trình tiền huấn luyện. Hai hàm mất mát này không liên quan đến việc tinh chỉnh trong các ứng dụng xuôi dòng, do đó các tham số của perceptron đa tầng dùng trong MaskLM và NextSentencePred không được cập nhật khi tinh chỉnh BERT.

Để cho phép sử dụng các tham số với gradient không cập nhật, ta đặt cờ ignore\_stale\_grad = True trong hàm step của d2l.train\_batch\_ch13. Chúng ta sử dụng chức năng này để huấn luyện và đánh giá mô hình net bằng cách sử dụng tập huấn luyện (train\_iter) và tập kiểm tra (test\_iter) của SNLI. Do hạn chế về tài nguyên tính toán, độ chính xác của việc huấn luyện và kiểm tra vẫn còn có thể được cải thiện hơn nữa: chúng sẽ thảo luận vấn đề này trong phần bài tập.

```
lr, num_epochs = 1e-4, 5
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
loss = gluon.loss.SoftmaxCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices,
               d2l.split_batch_multi_inputs)
```

#### 17.7.4 Tóm tắt

- Chúng ta có thể tinh chỉnh mô hình BERT đã tiền huấn luyện cho các ứng dụng xuôi dòng, chẳng hạn như suy luận ngôn ngữ tự nhiên trên tập dữ liệu SNLI.
- Trong quá trình tinh chỉnh, mô hình BERT trở thành một phần của mô hình ứng dụng xuôi dòng. Các tham số chỉ liên quan đến phần mất mát trong tiền huấn luyện sẽ không được cập nhật trong quá trình tinh chỉnh.

#### 17.7.5 Bài tập

1. Hãy tinh chỉnh một mô hình BERT tiền huấn luyện lớn hơn, có kích thước tương đương với mô hình BERT cơ sở ban đầu, nếu tài nguyên tính toán của bạn cho phép. Hãy thay đổi các đối số trong hàm `load_pretrained_model`: thay thế ‘bert.small’ bằng ‘bert.base’, lần lượt tăng giá trị của `num_hiddens = 256, ffn_num_hiddens = 512, num_heads = 4, num_layers = 2` thành `768, 3072, 12, 12`. Bằng cách tăng số epoch khi tinh chỉnh (và có thể điều chỉnh các siêu tham số khác), có thể nhận được độ chính xác trên tập kiểm tra cao hơn 0,86 không?
2. Làm thế nào để cắt ngắn một cặp chuỗi theo tỉ lệ độ dài của chúng? So sánh phương thức cắt ngắn cặp này và phương thức được sử dụng trong lớp `SNLIBERTDataset`. Ưu và nhược điểm của chúng là gì?

#### 17.7.6 Thảo luận

- Tiếng Anh: [MXNet<sup>367</sup>](#)
- Tiếng Việt: [Diễn đàn Machine Learning Cơ Bản<sup>368</sup>](#)

#### 17.7.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Nguyễn Thái Bình
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh

Lần cập nhật gần nhất: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 20/09/2020)

<sup>367</sup> <https://discuss.d2l.ai/t/397>

<sup>368</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 17.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 03/04/2020)

# 18 | Hệ thống Đề xuất

**Shuai Zhang** (Amazon), **Aston Zhang** (Amazon), và **Yi Tay** (Google).

Hệ thống đề xuất được sử dụng một cách rộng rãi trong kinh doanh và luôn hiện diện trong cuộc sống hàng ngày của chúng ta. Những hệ thống này được tận dụng trong nhiều lĩnh vực như thương mại điện tử (như amazon.com), các dịch vụ âm nhạc / điện ảnh (như Netflix và Spotify), cửa hàng ứng dụng di động (như App Store và Google Play), quảng cáo trực tuyến, v.v.

Mục đích chính của các hệ thống đề xuất là giúp người dùng tìm ra những sản phẩm liên quan như phim để xem, văn bản để đọc hay hàng hóa để mua, nhằm tạo nên một trải nghiệm thú vị cho người dùng. Hơn nữa, hệ thống đề xuất là một trong những hệ thống máy học mạnh mẽ nhất mà các công ty bán lẻ áp dụng với mục đích tăng doanh thu. Hệ thống đề xuất là công cụ thay thế cho các công cụ tìm kiếm bằng cách giảm nỗ lực tìm kiếm chủ động và tăng cơ hội tiếp cận của người dùng với những đề xuất mà họ không bao giờ tìm đến. Rất nhiều công ty đã vượt lên trên các đối thủ nhờ có hệ thống đề xuất hiệu quả hơn. Do đó, hệ thống đề xuất đã trở thành trung tâm không chỉ trong cuộc sống hàng ngày của chúng ta mà còn có vai trò quan trọng trong một số lĩnh vực kinh doanh.

Trong chương này, chúng tôi sẽ giới thiệu những nội dung cơ bản và những tiến bộ của hệ thống đề xuất, cùng với việc khám phá một số kỹ thuật cơ bản để xây dựng hệ thống đề xuất với các nguồn dữ liệu có sẵn khác nhau và cách lập trình những kỹ thuật này. Cụ thể, bạn sẽ học được cách để dự đoán mức đánh giá mà một người dùng sẽ đánh giá một sản phẩm, cách để tạo ra danh sách các sản phẩm đề xuất và cách dự đoán tỷ lệ nhấp chuột (*click-through rate*) từ một lượng lớn đặc trưng. Những tác vụ này vô cùng phổ biến trong các ứng dụng thực tế. Thông qua việc học chương này, bạn sẽ có được trải nghiệm thực tiễn để giải các bài toán đề xuất thực tế không chỉ với những phương pháp cổ điển mà còn là những mô hình tiên tiến hơn dựa trên học sâu.

## 18.1 Tổng quan về Hệ thống Đề xuất

Trong thập kỷ vừa qua, mạng Internet đã phát triển thành một nền tảng cho các dịch vụ trực tuyến quy mô lớn, đồng thời thay đổi sâu sắc cách ta giao tiếp, đọc tin tức, mua sắm và xem phim. Trong khi đó, một lượng lớn chưa từng có các sản phẩm (chúng tôi sử dụng từ *sản phẩm (item)* cho phim ảnh, tin tức, sách và hàng hóa) được bày bán trực tuyến yêu cầu một hệ thống có thể giúp ta tìm những sản phẩm ưa thích hơn. Do đó, hệ thống đề xuất là công cụ lọc thông tin mạnh mẽ có thể thúc đẩy các dịch vụ cá nhân hóa và cung cấp trải nghiệm riêng biệt cho từng người dùng. Nói ngắn gọn, hệ thống đề xuất đóng vai trò nòng cốt trong việc tận dụng nguồn dữ liệu dồi dào hiện có để giúp việc đưa ra lựa chọn dễ dàng hơn. Ngày nay, hệ thống đề xuất là thành phần trung tâm của nhiều nhà cung cấp dịch vụ trực tuyến như Amazon, Netflix, và YouTube. Nhớ lại ví dụ Amazon đưa ra đề xuất các sách Học sâu trong Fig. 3.3.3. Có hai lợi ích của việc sử dụng hệ thống đề xuất: Một mặt, nó có thể giảm lượng lớn công sức tìm kiếm sản phẩm của người dùng và giảm thiểu vấn đề quá tải thông tin. Mặt khác, nó có thể tăng giá trị kinh doanh cho các nhà cung cấp dịch vụ

trực tuyến và trở thành nguồn doanh thu quan trọng. Chương này sẽ giới thiệu những khái niệm cơ bản, các mô hình cổ điển và những bước tiến gần đây của học sâu trong lĩnh vực hệ thống đề xuất, cùng với các ví dụ lập trình.

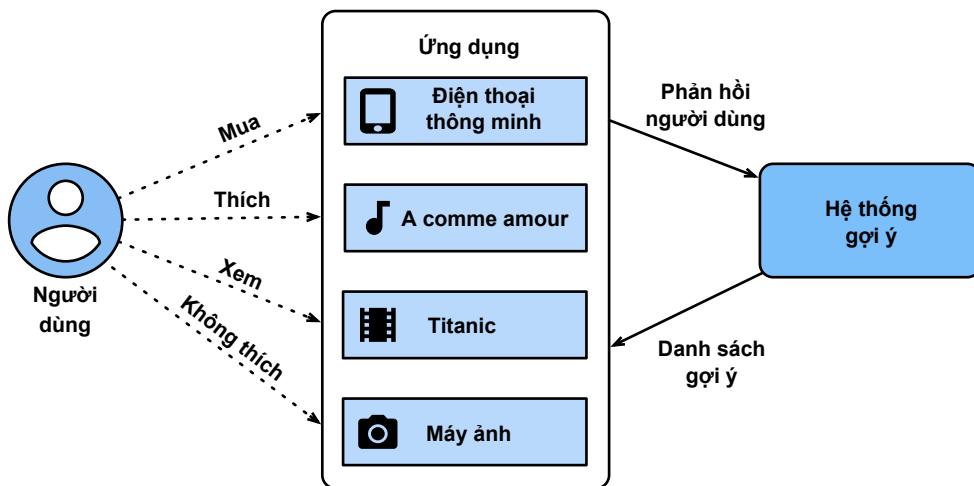


Fig. 18.1.1: Minh họa Quá trình Đề xuất

### 18.1.1 Lọc Cộng tác

Ta bắt đầu chương này với một khái niệm quan trọng trong hệ thống đề xuất – lọc cộng tác (*Collaborative Filtering - CF*), được tạo ra lần đầu trong hệ thống Tapestry (Goldberg et al., 1992), ám chỉ “mọi người cộng tác giúp đỡ lẫn nhau để thực hiện quá trình lọc nhằm xử lý lượng lớn email và tin nhắn đăng trong nhóm thảo luận”. Định nghĩa này được làm phong phú thêm bởi nhiều nghĩa. Hiểu theo nghĩa rộng, đây là quá trình lọc lấy thông tin hoặc khuôn mẫu sử dụng các kỹ thuật yêu cầu sự cộng tác của nhiều người dùng, tác nhân, và nguồn dữ liệu. CF có nhiều dạng khác nhau, rất nhiều phương pháp CF khác đã được đề xuất kể từ khi phát minh.

Nhìn chung, các kỹ thuật CF có thể được phân loại thành: CF dựa trên ghi nhớ (*memory-based CF*), CF dựa trên mô hình (*model-based CF*), và lai giữa hai lớp này (Su & Khoshgoftaar, 2009). Đại diện của CF dựa trên ghi nhớ chính là CF dựa trên các điểm lân cận (*nearest neighbor-based CF*) ví dụ như CF dựa trên người dùng (*user-based CF*) hay CF dựa trên sản phẩm (*item-based CF*) (Sarwar et al., 2001). Các mô hình nhân tố tiềm ẩn (*latent factor model*) như phân rã ma trận (*matrix factorization*) là một ví dụ của CF dựa trên mô hình. CF dựa trên ghi nhớ có nhiều hạn chế trong việc xử lý dữ liệu thừa và quy mô lớn do việc tính toán độ tương đồng dựa trên những sản phẩm thường gặp. CF dựa trên mô hình ngày càng trở nên phổ biến do khả năng xử lý dữ liệu thừa và tính mở rộng tốt hơn. Nhiều cách tiếp cận với CF dựa trên mô hình có thể được mở rộng với mạng nơ-ron, dẫn đến nhiều mô hình linh hoạt và tính mở rộng cao nhờ sự phát triển của học sâu (Zhang et al., 2019). Nhìn chung, CF chỉ sử dụng dữ liệu tương tác giữa người dùng - sản phẩm nhằm đưa ra dự đoán và đề xuất. Ngoài CF, hệ thống đề xuất dựa trên nội dung (*content-based*) và dựa trên ngữ cảnh (*context-based*) cũng hữu dụng trong việc kết hợp nội dung mô tả của sản phẩm/người dùng và các dấu hiệu ngữ cảnh như mốc thời gian và địa điểm. Đường nhiên, ta cần phải điều chỉnh cấu trúc/loại mô hình khi dữ liệu đầu vào khả dụng khác nhau.

### 18.1.2 Phản hồi Trực tiếp và Phản hồi Gián tiếp

Để học được sở thích của người dùng, hệ thống cần phải thu thập phản hồi của họ. Phản hồi này có thể là trực tiếp (*explicit*) hoặc gián tiếp (*implicit*) (Hu et al., 2008). Ví dụ, IMDB<sup>369</sup> thu thập đánh giá số lượng ngôi sao cho các bộ phim với các mức từ một đến mười sao. Youtube đưa ra nút thích (*thumps-up*) và không thích (*thumps-down*) cho người dùng để bày tỏ sở thích. Rõ ràng là việc thu thập phản hồi trực tiếp yêu cầu người dùng phải chủ động chỉ rõ sự quan tâm. Tuy nhiên, không phải lúc nào cũng dễ dàng thu thập phản hồi trực tiếp do nhiều người dùng thường không hay đánh giá sản phẩm. Xét một cách tương đối, phản hồi gián tiếp thường dễ thu thập hơn do chủ yếu liên quan đến việc mô hình hóa hành vi gián tiếp như số lần nhấp chuột của người dùng. Do đó, nhiều hệ thống đề xuất xoay quanh phản hồi gián tiếp, phản ánh ý kiến người dùng thông qua việc quan sát hành vi của họ. Có nhiều dạng phản hồi gián tiếp bao gồm lịch sử mua hàng, lịch sử duyệt web, lượt xem và thậm chí là thao tác chuột. Ví dụ, một người dùng mua nhiều sách của cùng tác giả thì khả năng cao là thích tác giả đó. Chú ý rằng phản hồi gián tiếp tự thân là có nhiều. Ta chỉ có thể đoán sở thích và động cơ thực của họ. Một người dùng xem một bộ phim không nhất thiết là phải thích bộ phim đó.

### 18.1.3 Các tác vụ Đề xuất

Có nhiều tác vụ đề xuất được nghiên cứu trong thập kỷ vừa qua. Dựa trên phạm vi ứng dụng, các tác vụ này bao gồm đề xuất phim ảnh, đề xuất tin tức, đề xuất địa điểm ưa thích (*point-of-interest*) (Ye et al., 2011), v.v. Ta cũng có thể phân biệt các tác vụ này dựa trên loại phản hồi và dữ liệu đầu vào, ví dụ như tác vụ trực tiếp dự đoán đánh giá. Đề xuất  $n$  sản phẩm hàng đầu (*top-:math: n* recommendation) (theo thứ tự sản phẩm) xếp loại tất cả các sản phẩm cho mỗi người dùng dựa trên phản hồi gián tiếp. Nếu có cả thông tin mốc thời gian, ta có thể xây dựng hệ thống đề xuất có nhận thức về chuỗi (*sequence-aware*) (Quadrana et al., 2018). Một tác vụ phổ biến khác là dự đoán tỷ lệ nhấp chuột, cũng dựa trên phản hồi gián tiếp, tuy nhiên rất nhiều đặc trưng rời rạc cũng có thể được tận dụng. Đưa ra đề xuất cho người dùng mới và đề xuất sản phẩm mới cho người dùng hiện còn được gọi là đề xuất khởi động nguội (*cold-start recommendation*) (Schein et al., 2002).

### 18.1.4 Tóm tắt

- Hệ thống đề xuất rất quan trọng đối với người dùng cá nhân và nhiều ngành công nghiệp. Lọc cộng tác là một khái niệm then chốt trong hệ thống đề xuất.
- Có hai loại phản hồi: gián tiếp và trực tiếp. Có nhiều ứng dụng đề xuất đã được nghiên cứu trong thập kỷ qua.

### 18.1.5 Bài tập

1. Hệ thống đề xuất ảnh hưởng đến cuộc sống hằng ngày của bạn như thế nào?
2. Có ứng dụng đề xuất nào đáng chú ý mà bạn nghĩ đáng được nghiên cứu?

<sup>369</sup> <https://www.imdb.com/>

### 18.1.6 Thảo luận

- Tiếng Anh: Main Forum<sup>370</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>371</sup>

### 18.1.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 12/09/2020)

## 18.2 Tập dữ liệu MovieLens

Có rất nhiều tập dữ liệu có sẵn dùng cho nghiên cứu hệ thống đề xuất. Trong số đó, tập dữ liệu MovieLens<sup>372</sup> có lẽ là một trong những tập phổ biến nhất. MovieLens là một hệ thống đề xuất phim phi thương mại trên nền tảng web. Nó được tạo ra vào năm 1997 và vận hành bởi GroupLens, một phòng nghiên cứu tại Đại học Minnesota, nhằm thu thập dữ liệu đánh giá phim phục vụ mục đích nghiên cứu. MovieLens là một nguồn dữ liệu quan trọng cho các nghiên cứu về cá nhân hóa đề xuất và tâm lý học xã hội.

### 18.2.1 Tải Dữ liệu

Tập dữ liệu MovieLens có địa chỉ tại GroupLens<sup>373</sup> với nhiều phiên bản khác nhau. Ở đây chúng ta sẽ sử dụng tập dữ liệu MovieLens 100K (Herlocker et al., 1999). Tập dữ liệu này bao gồm 100,000 đánh giá, xếp hạng từ 1 tới 5 sao, từ 943 người dùng dành cho 1682 phim. Nó được tiền xử lý sao cho mỗi người dùng đánh giá ít nhất 20 phim. Một vài thông tin nhân khẩu học cơ bản như tuổi và giới tính người dùng hay thể loại phim cũng được cung cấp. Ta có thể tải về [ml-100k.zip](#)<sup>374</sup> và giải nén tệp [u.data](#) chứa toàn bộ 100,000 đánh giá ở định dạng csv. Có nhiều tệp khác trong thư mục này, bao mô tả chi tiết cho mỗi tệp có thể được tìm thấy trong tệp [README](#)<sup>375</sup> của tập dữ liệu.

Để bắt đầu, ta hãy nhập những gói thư viện cần thiết để chạy các thử nghiệm của phần này.

```
from d2l import mxnet as d2l
from mxnet import gluon, np
import os
import pandas as pd
```

<sup>370</sup> <https://discuss.d2l.ai/t/398>

<sup>371</sup> <https://forum.machinelearningcoban.com/c/d2l>

<sup>372</sup> <https://movielens.org/>

<sup>373</sup> <https://grouplens.org/datasets/movielens/>

<sup>374</sup> <http://files.grouplens.org/datasets/movielens/ml-100k.zip>

<sup>375</sup> <http://files.grouplens.org/datasets/movielens/ml-100k-README.txt>

Sau đó, ta tải tập dữ liệu MovieLens 100k và đưa về định dạng DataFrame.

```
#@save
d2l.DATA_HUB['ml-100k'] = (
    'http://files.grouplens.org/datasets/movielens/ml-100k.zip',
    'cd4dcac4241c8a4ad7badc7ca635da8a69dddb83')

#@save
def read_data_ml100k():
    data_dir = d2l.download_extract('ml-100k')
    names = ['user_id', 'item_id', 'rating', 'timestamp']
    data = pd.read_csv(os.path.join(data_dir, 'u.data'), '\t', names=names,
                       engine='python')
    num_users = data.user_id.unique().shape[0]
    num_items = data.item_id.unique().shape[0]
    return data, num_users, num_items
```

### 18.2.2 Thống kê của Tập dữ liệu

Hãy nạp dữ liệu và quan sát năm bản ghi đầu tiên theo cách thủ công. Đây là một cách hiệu quả để học được cấu trúc dữ liệu cũng như chắc chắn rằng dữ liệu đã được nạp đúng.

```
data, num_users, num_items = read_data_ml100k()
sparsity = 1 - len(data) / (num_users * num_items)
print(f'number of users: {num_users}, number of items: {num_items}')
print(f'matrix sparsity: {sparsity:.f}')
print(data.head(5))
```

Có thể thấy rằng mỗi dòng chứa bốn cột, bao gồm “user id” 1-943, “item id” 1-1682, “rating” 1-5 và “timestamp”. Ta có thể tạo ra một ma trận tương tác có kích thước  $n \times m$ , với  $n$  và  $m$  lần lượt là số người dùng và số bộ phim. Tập dữ liệu này ghi lại các đánh giá đang tồn tại, vì thế ta có thể gọi nó là ma trận đánh giá. Ta sẽ sử dụng cả tên gọi ma trận tương tác và ma trận đánh giá trong trường hợp các giá trị của ma trận này biểu diễn chính xác các đánh giá. Hầu hết những giá trị trong ma trận đánh giá là chưa biết bởi đa số các bộ phim chưa được đánh giá bởi người dùng. Ta cũng có thể biểu diễn độ thưa thớt (*sparsity*) của tập dữ liệu này. Độ thưa thớt được định nghĩa là  $1 - \frac{\text{số lượng các bản ghi khác không}}{(\text{số lượng người dùng} * \text{số lượng sản phẩm})}$ . Rõ ràng, ma trận tương tác là cực kỳ thưa thớt (độ thưa = 93.695%). Các tập dữ liệu trong thực tế thường có mức độ thưa thớt lớn hơn nhiều, và từ lâu đã trở thành thử thách trong việc xây dựng các hệ thống đề xuất. Một giải pháp khả thi đó là sử dụng các thông tin phụ như đặc trưng của người dùng/sản phẩm để giúp giảm bớt tác động từ tính thưa thớt.

Tiếp theo ta vẽ biểu đồ phân phối số lượng các đánh giá khác nhau. Đúng như mong đợi, nó trông giống một phân phối chuẩn, với hầu hết các đánh giá tập trung tại 3-4.

```
d2l.plt.hist(data['rating'], bins=5, ec='black')
d2l.plt.xlabel('Rating')
d2l.plt.ylabel('Count')
d2l.plt.title('Distribution of Ratings in MovieLens 100K')
d2l.plt.show()
```

### 18.2.3 Chia tập Dữ liệu

Ta chia tập dữ liệu thành tập huấn luyện và tập kiểm tra. Hàm dưới đây cung cấp hai chế độ chia bao gồm random và seq-aware. Trong chế độ random, dữ liệu 100k tương tác sẽ được chia một cách ngẫu nhiên mà không xét tới mốc thời gian, mặc định sử dụng 90% dữ liệu để làm mẫu huấn luyện và 10% còn lại là mẫu kiểm tra. Trong chế độ seq-aware, ta giữ lại sản phẩm mà người dùng đánh giá gần đây nhất cho tập kiểm tra, còn các tương tác trước đó sẽ được sử dụng cho tập huấn luyện. Lịch sử tương tác người dùng được sắp xếp từ cũ nhất tới mới nhất theo mốc thời gian. Chế độ này sẽ được sử dụng trong phần đề xuất có nhận thức về chuỗi.

```
#@save
def split_data_ml100k(data, num_users, num_items,
                      split_mode='random', test_ratio=0.1):
    """Split the dataset in random mode or seq-aware mode."""
    if split_mode == 'seq-aware':
        train_items, test_items, train_list = {}, {}, []
        for line in data.itertuples():
            u, i, rating, time = line[1], line[2], line[3], line[4]
            train_items.setdefault(u, []).append((u, i, rating, time))
            if u not in test_items or test_items[u][-1] < time:
                test_items[u] = (i, rating, time)
        for u in range(1, num_users + 1):
            train_list.extend(sorted(train_items[u], key=lambda k: k[3]))
        test_data = [(key, *value) for key, value in test_items.items()]
        train_data = [item for item in train_list if item not in test_data]
        train_data = pd.DataFrame(train_data)
        test_data = pd.DataFrame(test_data)
    else:
        mask = [True if x == 1 else False for x in np.random.uniform(
            0, 1, (len(data))) < 1 - test_ratio]
        neg_mask = [not x for x in mask]
        train_data, test_data = data[mask], data[neg_mask]
    return train_data, test_data
```

Lưu ý rằng trong thực tiễn, sử dụng một tập kiểm định tách biệt thay vì chỉ có một tập kiểm tra duy nhất là sự lựa chọn phù hợp. Tuy nhiên, chúng tôi bỏ qua điều này với mục đích cô đọng nội dung. Trong trường hợp này, ta có thể coi tập kiểm tra như một tập kiểm định bất đắc dĩ.

### 18.2.4 Nạp dữ liệu

Sau khi chia nhỏ tập dữ liệu, chúng ta sẽ biến đổi tập huấn luyện và tập kiểm tra thành các danh sách và từ điển/ma trận cho thuận tiện. Hàm dưới đây đọc dataframe vào theo từng dòng và duyệt qua từng chỉ mục của người dùng/sản phẩm bắt đầu từ 0. Tiếp đó nó trả về danh sách người dùng, sản phẩm, đánh giá và một từ điển/ma trận chứa các tương tác. Ta có thể chỉ rõ loại phản hồi là explicit (*trực tiếp*) hay implicit (*gián tiếp*).

```
#@save
def load_data_ml100k(data, num_users, num_items, feedback='explicit'):
    users, items, scores = [], [], []
    inter = np.zeros((num_items, num_users)) if feedback == 'explicit' else {}
    for line in data.itertuples():
        user_index, item_index = int(line[1] - 1), int(line[2] - 1)
        score = int(line[3]) if feedback == 'explicit' else 1
        inter[item_index, user_index] = score
    return users, items, scores, inter
```

(continues on next page)

```

users.append(user_index)
items.append(item_index)
scores.append(score)
if feedback == 'implicit':
    inter.setdefault(user_index, []).append(item_index)
else:
    inter[item_index, user_index] = score
return users, items, scores, inter

```

Cuối cùng ta kết hợp các bước trên lại để sử dụng ở phần tiếp theo. Kết quả được gói gọn trong Dataset và DataLoader. Lưu ý rằng tham số last\_batch của DataLoader dùng cho dữ liệu huấn luyện được thiếp lập ở chế độ rollover (các mẫu còn lại được đưa vào epoch tiếp theo) với thứ tự được xáo trộn.

```

#@save
def split_and_load_ml100k(split_mode='seq-aware', feedback='explicit',
                           test_ratio=0.1, batch_size=256):
    data, num_users, num_items = read_data_ml100k()
    train_data, test_data = split_data_ml100k(
        data, num_users, num_items, split_mode, test_ratio)
    train_u, train_i, train_r, _ = load_data_ml100k(
        train_data, num_users, num_items, feedback)
    test_u, test_i, test_r, _ = load_data_ml100k(
        test_data, num_users, num_items, feedback)
    train_set = gluon.data.ArrayDataset(
        np.array(train_u), np.array(train_i), np.array(train_r))
    test_set = gluon.data.ArrayDataset(
        np.array(test_u), np.array(test_i), np.array(test_r))
    train_iter = gluon.data.DataLoader(
        train_set, shuffle=True, last_batch='rollover',
        batch_size=batch_size)
    test_iter = gluon.data.DataLoader(
        test_set, batch_size=batch_size)
    return num_users, num_items, train_iter, test_iter

```

### 18.2.5 Tóm tắt

- Tập dữ liệu MovieLens được sử dụng rộng rãi trong nghiên cứu hệ thống đề xuất. Đây là tập dữ liệu công khai và được sử dụng miễn phí.
- Ta định nghĩa các hàm tải và tiền xử lý tập dữ liệu MovieLens 100k để sử dụng trong những phần tiếp theo.

### 18.2.6 Bài tập

- Bạn có thể tìm được tập dữ liệu đề xuất nào khác tương tự như tập MovieLens không?
- Xem qua trang <https://movielens.org/> để biết thêm thông tin về MovieLens.

### 18.2.7 Thảo luận

- Tiếng Anh: [MXNet<sup>376</sup>](#)
- Tiếng Việt: [Diễn đàn Machine Learning Cơ Bản<sup>377</sup>](#)

### 18.2.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 17/07/2020)

## 18.3 Phân rã Ma trận

Phân rã ma trận (*Matrix Factorization*) (Koren et al., 2009) là một trong những thuật toán lâu đời trong các tài liệu về hệ thống đề xuất. Phiên bản đầu tiên của mô hình phân rã ma trận được đề xuất bởi Simon Funk trong một [bài blog nổi tiếng<sup>378</sup>](#), trong đó anh đã mô tả ý tưởng phân rã ma trận tương tác thành các nhân tử. Phân rã ma trận sau đó trở nên phổ biến nhờ cuộc thi Netflix tổ chức năm 2006. Tại thời điểm đó, Netflix, một công ty truyền thông đa phương tiện và cho thuê phim, công bố một cuộc thi nhằm cải thiện hiệu năng hệ thống đề xuất của họ. Đội xuất sắc nhất giúp cải thiện Netflix ở mức cơ sở (*baseline*) như thuật toán Cinematch lên 10 phần trăm sẽ đoạt giải thưởng là một triệu USD. Do đó, cuộc thi này thu hút rất nhiều sự chú ý trong ngành nghiên cứu hệ thống đề xuất. Cuối cùng, giải thưởng chung cuộc đã thuộc về đội Pragmatic Chaos của BellKor, là đội đã kết hợp của BellKor, Pragmatic Theory và BigChaos (hiện tại bạn chưa cần phải quan tâm đến các thuật toán này). Dù kết quả cuối cùng là một giải pháp kết hợp (tức phối hợp nhiều thuật toán với nhau), thuật toán phân rã ma trận đóng vai trò chủ đạo trong thuật toán kết hợp cuối cùng. Báo cáo kỹ thuật của Giải thưởng Netflix (Toscher et al., 2009) cung cấp giới thiệu chi tiết về mô hình được chấp thuận. Trong phần này, ta sẽ đi sâu vào chi tiết mô hình phân rã ma trận và cách lập trình nó.

<sup>376</sup> <https://discuss.d2l.ai/t/399>

<sup>377</sup> <https://forum.machinelearningcoban.com/c/d2l>

<sup>378</sup> <https://sifter.org/~simon/journal/20061211.html>

### 18.3.1 Mô hình Phân rã Ma trận

Phân rã ma trận là một lớp trong các mô hình lọc cộng tác. Cụ thể, mô hình này phân tích ma trận tương tác giữa người dùng - sản phẩm (ví dụ như ma trận đánh giá) thành tích hai ma trận có hạng thấp hơn, nhằm nắm bắt cấu trúc hạng thấp trong tương tác người dùng - sản phẩm.

Gọi  $\mathbf{R} \in \mathbb{R}^{m \times n}$  ký hiệu ma trận tương tác với  $m$  người dùng và  $n$  sản phẩm, và các giá trị  $\mathbf{R}$  biểu diễn đánh giá trực tiếp. Tương tác người dùng - sản phẩm được phân tích thành ma trận người dùng tiềm ẩn  $\mathbf{P} \in \mathbb{R}^{m \times k}$  và ma trận sản phẩm tiềm ẩn  $\mathbf{Q} \in \mathbb{R}^{n \times k}$ , trong đó  $k \ll m, n$ , là kích thước nhân tố tiềm ẩn. Gọi  $\mathbf{p}_u$  ký hiệu hàng thứ  $u$  và của  $\mathbf{P}$  và  $\mathbf{q}_i$  ký hiệu hàng thứ  $i$  của  $\mathbf{Q}$ . Với một sản phẩm  $i$  cho trước, các phần tử trong  $\mathbf{q}_i$  đo lường mức độ mà sản phẩm đó sở hữu các đặc trưng, ví dụ như thể loại hay ngôn ngữ của một bộ phim. Với một người dùng  $u$  cho trước, các phần tử trong  $\mathbf{p}_u$  đo mức độ ưa thích của người dùng này đối với các đặc trưng tương ứng của các sản phẩm. Các nhân tố tiềm ẩn này có thể là các đặc trưng rõ ràng như đã đề cập trong các ví dụ trên, hoặc hoàn toàn không thể giải thích được. Đánh giá dự đoán có thể được ước lượng như sau

$$\hat{\mathbf{R}} = \mathbf{P}\mathbf{Q}^\top \quad (18.3.1)$$

trong đó  $\hat{\mathbf{R}} \in \mathbb{R}^{m \times n}$  là ma trận đánh giá dự đoán và có cùng kích thước với  $\mathbf{R}$ . Một vấn đề lớn của cách dự đoán này là độ chêch (*bias*) của người dùng/sản phẩm không được mô hình hóa. Ví dụ, một số người dùng có thiên hướng đánh giá cao hơn, hoặc một số sản phẩm luôn bị đánh giá thấp hơn bởi chất lượng kém. Các độ chêch này là rất phổ biến trong những ứng dụng thực tế. Để thu được các độ chêch này, số hạng độ chêch riêng biệt cho từng người dùng và sản phẩm được sử dụng. Cụ thể, đánh giá dự đoán của người dùng  $u$  cho sản phẩm  $i$  được tính theo công thức

$$\hat{\mathbf{R}}_{ui} = \mathbf{p}_u \mathbf{q}_i^\top + b_u + b_i \quad (18.3.2)$$

Sau đó, ta huấn luyện mô hình phân rã ma trận bằng cách cực tiểu hóa trung bình bình phương sai số giữa đánh giá dự đoán và đánh giá thực. Hàm mục tiêu được định nghĩa như sau:

$$\underset{\mathbf{P}, \mathbf{Q}, b}{\operatorname{argmin}} \sum_{(u,i) \in \mathcal{K}} \|\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui}\|^2 + \lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2 + b_u^2 + b_i^2) \quad (18.3.3)$$

trong đó  $\lambda$  là tỷ lệ điều chỉnh. Số hạng điều chỉnh  $\lambda(\|\mathbf{P}\|_F^2 + \|\mathbf{Q}\|_F^2 + b_u^2 + b_i^2)$  được sử dụng để tránh hiện tượng quá khớp bằng cách phạt độ lớn của các tham số. Cặp  $(u, i)$  với  $\mathbf{R}_{ui}$  đã biết được lưu trong tập  $\mathcal{K} = \{(u, i) \mid \mathbf{R}_{ui} \text{đã biết}\}$ . Các tham số mô hình có thể được học thông qua một thuật toán tối ưu, ví dụ như hạ gradient ngẫu nhiên hay Adam.

Ảnh minh họa trực quan cho mô hình phân rã ma trận được đưa ra như hình dưới:

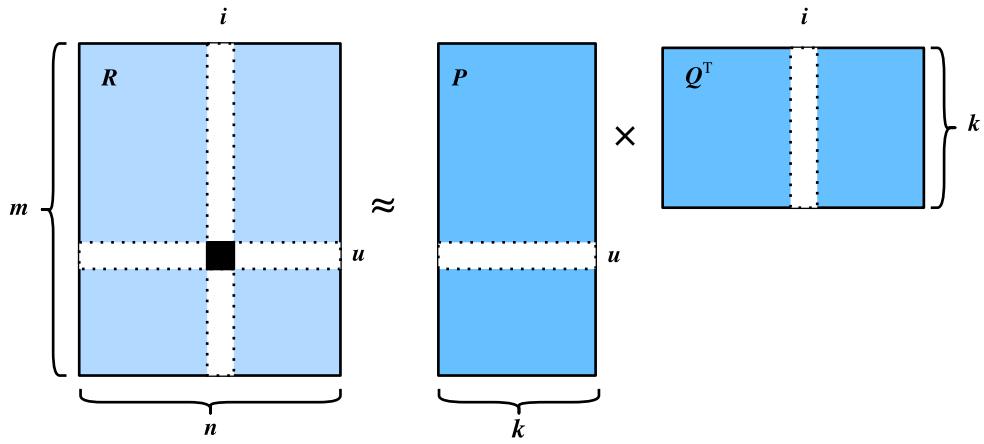


Fig. 18.3.1: Minh họa mô hình phân rã ma trận

Trong phần còn lại, chúng tôi sẽ giải thích cách lập trình cho phân rã ma trận và huấn luyện mô hình trên tập dữ liệu MovieLens.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
npx.set_np()
```

### 18.3.2 Cách lập trình Mô hình

Đầu tiên, ta lập trình mô hình phân rã ma trận như mô tả trên. Các nhân tố tiềm ẩn của người dùng và sản phẩm được tạo bằng `nn.Embedding`. Tham số `input_dim` là số sản phẩm/người dùng và `output_dim` là kích thước nhân tố tiềm ẩn ( $k$ ). Ta cũng có thể sử dụng `nn.Embedding` để tạo độ chêch cho người dùng/sản phẩm bằng cách gán `output_dim` bằng một. Trong hàm `forward`, id người dùng và sản phẩm được sử dụng để truy vấn tới embedding tương ứng.

```
class MF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, **kwargs):
        super(MF, self).__init__(**kwargs)
        self.P = nn.Embedding(input_dim=num_users, output_dim=num_factors)
        self.Q = nn.Embedding(input_dim=num_items, output_dim=num_factors)
        self.user_bias = nn.Embedding(num_users, 1)
        self.item_bias = nn.Embedding(num_items, 1)

    def forward(self, user_id, item_id):
        P_u = self.P(user_id)
        Q_i = self.Q(item_id)
        b_u = self.user_bias(user_id)
        b_i = self.item_bias(item_id)
        outputs = (P_u * Q_i).sum(axis=1) + np.squeeze(b_u) + np.squeeze(b_i)
        return outputs.flatten()
```

### 18.3.3 Phương pháp Đánh giá

Tiếp theo, ta lập trình phép đo RMSE (*root-mean-square error - căn bậc hai trung bình bình phương sai số*), phương pháp này được sử dụng rộng rãi nhằm đo lường sự khác nhau giữa giá trị đánh giá dự đoán và giá trị đánh giá thực tế (nhân gốc) (Gunawardana & Shani, 2015). RMSE được định nghĩa như sau:

$$\text{RMSE} = \sqrt{\frac{1}{|\mathcal{T}|} \sum_{(u,i) \in \mathcal{T}} (\mathbf{R}_{ui} - \hat{\mathbf{R}}_{ui})^2} \quad (18.3.4)$$

trong đó  $\mathcal{T}$  là tập bao gồm các cặp người dùng và sản phẩm mà ta sử dụng để đánh giá.  $|\mathcal{T}|$  là kích thước tập này. Ta có thể sử dụng hàm RMSE được cung cấp sẵn trong `mx.metric`.

```
def evaluator(net, test_iter, devices):
    rmse = mx.metric.RMSE() # Get the RMSE
    rmse_list = []
    for idx, (users, items, ratings) in enumerate(test_iter):
        u = gluon.utils.split_and_load(users, devices, even_split=False)
        i = gluon.utils.split_and_load(items, devices, even_split=False)
        r_ui = gluon.utils.split_and_load(ratings, devices, even_split=False)
        r_hat = [net(u, i) for u, i in zip(u, i)]
        rmse.update(labels=r_ui, preds=r_hat)
        rmse_list.append(rmse.get()[1])
    return float(np.mean(np.array(rmse_list)))
```

### 18.3.4 Huấn luyện và Đánh giá Mô hình

Trong hàm huấn luyện, ta áp dụng mất mát  $L_2$  với suy giảm trọng số. Phương thức suy giảm trọng số có tác dụng giống như điều chỉnh  $L_2$ .

```
#@save
def train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        devices=d2l.try_all_gpus(), evaluator=None,
                        **kwargs):
    timer = d2l.Timer()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 2],
                            legend=['train loss', 'test RMSE'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            timer.start()
            input_data = []
            values = values if isinstance(values, list) else [values]
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, devices))
            train_feat = input_data[0:-1] if len(values) > 1 else input_data
            train_label = input_data[-1]
            with autograd.record():
                preds = [net(*t) for t in zip(*train_feat)]
                ls = [loss(p, s) for p, s in zip(preds, train_label)]
                [l.backward() for l in ls]
            l += sum([l.asnumpy() for l in ls]).mean() / len(devices)
            metric.add(l, 0, 1)
```

(continues on next page)

```

        trainer.step(values[0].shape[0])
        metric.add(l, values[0].shape[0], values[0].size)
        timer.stop()
    if len(kwargs) > 0: # It will be used in section AutoRec
        test_rmse = evaluator(net, test_iter, kwargs['inter_mat'],
                              devices)
    else:
        test_rmse = evaluator(net, test_iter, devices)
    train_l = l / (i + 1)
    animator.add(epoch + 1, (train_l, test_rmse))
    print(f'train loss {metric[0] / metric[1]:.3f}, '
          f'test RMSE {test_rmse:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)})'

```

Cuối cùng, hãy kết hợp tất cả với nhau và huấn luyện mô hình. Trường hợp này, ta đặt kích thước nhân tố tiềm ẩn bằng 30.

```

devices = d2l.try_all_gpus()
num_users, num_items, train_iter, test_iter = d2l.split_and_load_ml100k(
    test_ratio=0.1, batch_size=512)
net = MF(30, num_users, num_items)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 20, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                     devices, evaluator)

```

Ở dưới, ta sử dụng mô hình đã được huấn luyện để dự đoán đánh giá mà một người dùng (ID 20) có thể gán cho một sản phẩm (ID 30).

```

scores = net(np.array([20], dtype='int', ctx=devices[0]),
            np.array([30], dtype='int', ctx=devices[0]))
scores

```

### 18.3.5 Tóm tắt

- Mô hình phân rã ma trận được sử dụng rộng rãi trong hệ thống đề xuất. Nó có thể được sử dụng để dự đoán đánh giá của một người dùng cho một sản phẩm.
- Ta có thể lập trình và huấn luyện mô hình phân rã ma trận cho hệ thống đề xuất.

### 18.3.6 Bài tập

- Thay đổi kích thước của nhân tố tiềm ẩn. Kích thước của nhân tố tiềm ẩn ảnh hưởng thế nào đến hiệu năng của mô hình?
- Thủ các bộ tối ưu, tốc độ học và tốc độ suy giảm trọng số khác nhau.
- Kiểm tra giá trị đánh giá dự đoán của những người dùng khác cho một bộ phim cụ thể.

### 18.3.7 Thảo luận

- Tiếng Anh: MXNet<sup>379</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>380</sup>

### 18.3.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 12/09/2020)

## 18.4 AutoRec: Dự đoán Đánh giá với Bộ tự Mã hóa

Mặc dù mô hình phân rã ma trận đạt hiệu năng tương đối ổn với bài toán dự đoán đánh giá, nhưng về căn bản nó là một mô hình tuyến tính. Do đó, mô hình dạng này không có khả năng nắm bắt được mối quan hệ phi tuyến phức tạp và rắc rối, mà có thể có khả năng dự đoán sở thích người dùng. Trong phần này, chúng tôi sẽ giới thiệu một mô hình mạng nơ-ron lọc cộng tác phi tuyến, gọi là AutoRec (Sedhain et al., 2015). Nó áp dụng lọc cộng tác (*collaborative filtering - CF*) với kiến trúc của một bộ tự mã hóa (*autoencoder*), nhằm mục đích tích hợp biến đổi phi tuyến vào CF dựa trên cơ sở các phản hồi trực tiếp. Mạng nơ-ron đã được chứng minh là có khả năng xấp xỉ bất kì hàm liên tục nào, điều này khiến nó phù hợp để khắc phục các hạn chế và tăng cường khả năng biểu diễn của mô hình phân rã ma trận.

Một mặt, AutoRec có cùng cấu trúc với một bộ tự mã hóa gồm một tầng đầu vào, một tầng ẩn và một tầng khôi phục (đầu ra). Bộ tự mã hóa là một mạng nơ-ron học cách sao chép đầu vào sang đầu ra nhằm mã hóa đầu vào thành dạng biểu diễn ẩn (và thường có kích thước nhỏ). Trong AutoRec, thay vì trực tiếp tạo embedding của người dùng/sản phẩm trong không gian kích thước nhỏ hơn, ta sử dụng các cột/hàng của ma trận tương tác làm đầu vào, sau đó khôi phục lại ma trận tương tác ở tầng đầu ra.

<sup>379</sup> <https://discuss.d2l.ai/t/400>

<sup>380</sup> <https://forum.machinelearningcoban.com/c/d2l>

Mặt khác, AutoRec khác với bộ tự mã hóa truyền thống ở chỗ: thay vì học dạng biểu diễn ẩn, AutoRec tập trung vào học/khôi phục tầng đầu ra. Nó sử dụng phần đã biết của ma trận tương tác làm đầu vào, nhằm khôi phục lại ma trận đánh giá hoàn chỉnh. Trong khi đó, các phần tử còn thiếu trong đầu vào được điền vào tầng đầu ra thông qua quá trình khôi phục cho mục đích đề xuất.

Có hai dạng AutoRec: dựa trên người dùng (*user-based*) và dựa trên sản phẩm (*item-based*). Để ngắn gọn, ở đây chúng tôi chỉ giới thiệu AutoRec dựa trên sản phẩm. AutoRec dựa trên người dùng có thể được suy ra một cách tương tự.

#### 18.4.1 Mô hình

Gọi  $\mathbf{R}_{*i}$  ký hiệu cột thứ  $i$  của ma trận đánh giá. Những đánh giá chưa biết được gán mặc định bằng không. Kiến trúc nơ-ron được định nghĩa như sau:

$$h(\mathbf{R}_{*i}) = f(\mathbf{W} \cdot g(\mathbf{V}\mathbf{R}_{*i} + \mu) + b) \quad (18.4.1)$$

trong đó  $f(\cdot)$  và  $g(\cdot)$  biểu diễn hàm kích hoạt,  $\mathbf{W}$  và  $\mathbf{V}$  là các ma trận trọng số,  $\mu$  và  $b$  là hệ số điều chỉnh. Gọi  $h(\cdot)$  ký hiệu cho toàn bộ mạng AutoRec. Đầu ra  $h(\mathbf{R}_{*i})$  chính là bản khôi phục của cột thứ  $i$  của ma trận đánh giá.

Hàm mục tiêu sau hướng tới việc cực tiểu hóa sai số khôi phục:

$$\operatorname{argmin}_{\mathbf{W}, \mathbf{V}, \mu, b} \sum_{i=1}^M \| \mathbf{R}_{*i} - h(\mathbf{R}_{*i}) \|_O^2 + \lambda (\|\mathbf{W}\|_F^2 + \|\mathbf{V}\|_F^2) \quad (18.4.2)$$

trong đó  $\| \cdot \|_O$  nghĩa là chỉ có phần đánh giá đã biết là được xét, tức chỉ các trọng số tương ứng với những đầu vào đã biết mới được cập nhật trong lan truyền ngược.

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import sys
npx.set_np()
```

#### 18.4.2 Lập trình Mô hình

Một bộ tự mã hóa điển hình bao gồm một bộ mã hóa và một bộ giải mã. Bộ mã hóa chiếu đầu vào thành dạng biểu diễn ẩn và bộ giải mã ánh xạ tầng ẩn tới tầng khôi phục. Ta tuân theo cấu trúc này và tạo bộ mã hóa cùng bộ giải mã với các tầng kết nối dày đặc. Hàm kích hoạt của bộ mã hóa được đặt mặc định bằng sigmoid và ta sẽ không áp dụng hàm kích hoạt nào lên tầng giải mã. Dropout được thêm vào sau khi mã hóa nhằm giảm hiện tượng quá khớp. Gradient của các đầu vào chưa biết được che lại để đảm bảo rằng chỉ có các đánh giá đã biết tham gia vào quá trình học của mô hình.

```
class AutoRec(nn.Block):
    def __init__(self, num_hidden, num_users, dropout=0.05):
        super(AutoRec, self).__init__()
```

(continues on next page)

```

    self.encoder = nn.Dense(num_hidden, activation='sigmoid',
                           use_bias=True)
    self.decoder = nn.Dense(num_users, use_bias=True)
    self.dropout = nn.Dropout(dropout)

    def forward(self, input):
        hidden = self.dropout(self.encoder(input))
        pred = self.decoder(hidden)
        if autograd.is_training(): # Mask the gradient during training
            return pred * np.sign(input)
        else:
            return pred

```

### 18.4.3 Lập trình lại Bộ Đánh giá

Do đầu vào và đầu ra thay đổi nên ta cần phải lập trình lại hàm đánh giá, nhưng vẫn sử dụng RMSE làm phép đo độ chính xác.

```

def evaluator(network, inter_matrix, test_data, devices):
    scores = []
    for values in inter_matrix:
        feat = gluon.utils.split_and_load(values, devices, even_split=False)
        scores.extend([network(i).asnumpy() for i in feat])
    recons = np.array([item for sublist in scores for item in sublist])
    # Calculate the test RMSE
    rmse = np.sqrt(np.sum(np.square(test_data - np.sign(test_data) * recons)) /
                   np.sum(np.sign(test_data)))
    return float(rmse)

```

### 18.4.4 Huấn luyện và Đánh giá Mô hình

Giờ hãy cùng huấn luyện và đánh giá AutoRec trên tập dữ liệu MovieLens. Ta có thể thấy rõ ràng rằng RMSE kiểm tra thấp hơn so với mô hình phân rã ma trận, điều này xác thực độ hiệu quả của mạng nơ-ron trong nhiệm vụ dự đoán đánh giá.

```

devices = d2l.try_all_gpus()
# Load the MovieLens 100K dataset
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items)
_, _, _, train_inter_mat = d2l.load_data_ml100k(train_data, num_users,
                                                num_items)
_, _, _, test_inter_mat = d2l.load_data_ml100k(test_data, num_users,
                                                num_items)
train_iter = gluon.data.DataLoader(train_inter_mat, shuffle=True,
                                   last_batch="rollover", batch_size=256,
                                   num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(np.array(train_inter_mat), shuffle=False,
                                   last_batch="keep", batch_size=1024,
                                   num_workers=d2l.get_dataloader_workers())
# Model initialization, training, and evaluation

```

(continues on next page)

```

net = AutoRec(500, num_users)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.002, 25, 1e-5, 'adam'
loss = gluon.loss.L2Loss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
d2l.train_recsys_rating(net, train_iter, test_iter, loss, trainer, num_epochs,
                        devices, evaluator, inter_mat=test_inter_mat)

```

#### 18.4.5 Tóm tắt

- Ta có thể thiết kế thuật toán phân rã ma trận với bộ tự giải mã, cùng lúc tích hợp các tầng phi tuyến và điều chỉnh dropout.
- Thí nghiệm trên tập dữ liệu MovieLens 100K cho thấy AutoRec đạt hiệu năng vượt trội so với phân rã ma trận.

#### 18.4.6 Bài tập

- Thay đổi kích thước ẩn của AutoRec để quan sát ảnh hưởng của việc này lên hiệu năng mô hình.
- Hãy thử thêm vào nhiều tầng ẩn. Việc này có giúp cải thiện hiệu năng mô hình không?
- Liệu bạn có thể tìm một bộ hàm kích hoạt nào khác tốt hơn cho bộ giải mã và bộ mã hóa?

#### 18.4.7 Thảo luận

- Tiếng Anh: MXNet<sup>381</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>382</sup>

#### 18.4.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Nguyễn Thái Bình
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc

Cập nhật lần cuối: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 21/07/2020)

<sup>381</sup> <https://discuss.d2l.ai/t/401>

<sup>382</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 18.5 Cá nhân hóa Xếp hạng trong Hệ thống Đề xuất

Trong những phần trước, mô hình được huấn luyện và kiểm tra trên các đánh giá đã biết và chỉ các phản hồi trực tiếp là được xét đến. Phương pháp này có hai khuyết điểm: Thứ nhất, đa phần các phản hồi trong thực tế không dưới dạng trực tiếp mà là gián tiếp, và phản hồi trực tiếp thường khó thu thập hơn. Thứ hai, những cặp người dùng - sản phẩm chưa biết lại hoàn toàn bị bỏ qua, dù chúng có thể được sử dụng để dự đoán sở thích người dùng. Điều này khiến cho các phương pháp trên không phù hợp khi mà những đánh giá không phải là thiếu do ngẫu nhiên mà đến từ thị hiếu của người dùng. Những cặp người dùng - sản phẩm chưa biết là sự pha trộn giữa các phản ánh tiêu cực (người dùng không hứng thú với sản phẩm) và các giá trị còn thiếu (có lẽ sau này người dùng sẽ tương tác với sản phẩm). Ta đơn thuần bỏ qua những cặp chưa biết này trong phương pháp phân rã ma trận và AutoRec. Rõ ràng là những mô hình này không có khả năng phân biệt giữa những cặp đã biết và cặp chưa biết và thường không phù hợp với tác vụ cá nhân hóa xếp hạng (*personalized ranking*).

Từ đó, một nhóm mô hình đề xuất hướng tới việc tạo ra danh sách xếp hạng đề xuất từ phản hồi gián tiếp dần trở nên phổ biến. Thông thường, những mô hình cá nhân hóa xếp hạng có thể được tối ưu bằng các phương thức tiếp cận theo từng điểm, theo từng cặp hoặc theo danh sách. Cách tiếp cận từng điểm xét từng tương tác một và huấn luyện một bộ phân loại hoặc một bộ hồi quy để dự đoán sở thích cá nhân. Phân rã ma trận và AutoRec được tối ưu với các mục tiêu theo từng điểm. Cách tiếp cận theo từng cặp xét một cặp sản phẩm với mỗi người dùng và nhắm tới việc xấp xỉ thứ bậc tối ưu của cặp sản phẩm đó. Thường thì cách tiếp cận theo từng cặp phù hợp với tác vụ xếp hạng hơn do việc dự đoán thứ bậc tương đối gần với bản chất của việc xếp hạng. Cách tiếp cận theo danh sách ước chừng thứ bậc của toàn bộ danh sách các sản phẩm, ví dụ như trực tiếp tối ưu hệ số Độ lợi Chiết khấu Tích luỹ Chuẩn (*Normalized Discounted Cumulative Gain - NDCG*<sup>383</sup>). Tuy nhiên, cách tiếp cận theo danh sách phức tạp hơn và đòi hỏi tài nguyên tính toán cao hơn so với cách tiếp cận theo từng điểm và theo từng cặp. Trong phần này, chúng tôi sẽ giới thiệu hai loại mất mát/mục tiêu của cách tiếp cận theo từng cặp, mất mát Cá nhân hóa Xếp hạng Bayes (*Bayesian Personalized Ranking*) và mất mát Hinge, cùng với cách lập trình từng loại mất mát tương ứng.

### 18.5.1 Mất mát Cá nhân hóa Xếp hạng Bayes và Cách lập trình

Cá nhân hóa Xếp hạng Bayes (BPR) (Rendle et al., 2009) là một hàm mất mát cá nhân hóa xếp hạng theo cặp, có xuất phát từ bộ ước lượng hậu nghiệm cực đại (*maximum posterior estimator*). Nó được sử dụng rộng rãi trong nhiều mô hình đề xuất hiện nay. Dữ liệu huấn luyện cho BPR bao gồm cả các cặp tích cực lẫn tiêu cực (các giá trị còn thiếu). Nó giả sử rằng người dùng ưa thích sản phẩm tích cực hơn tất cả các sản phẩm chưa biết.

Trong công thức, dữ liệu huấn luyện được xây dựng bằng tuple dưới dạng  $(u, i, j)$ , tức biểu diễn rằng người dùng  $u$  ưa thích sản phẩm  $i$  hơn sản phẩm  $j$ . Công thức Bayes trong BPR được cho dưới đây nhằm tới việc cực đại hóa xác suất hậu nghiệm:

$$p(\Theta | >_u) \propto p(>_u | \Theta)p(\Theta) \quad (18.5.1)$$

trong đó  $\Theta$  biểu diễn các tham số của một mô hình đề xuất bất kỳ,  $>_u$  biểu diễn tổng xếp hạng mong muốn cá nhân hóa của tất cả sản phẩm cho người dùng  $u$ . Ta có thể xây dựng công thức của bộ ước lượng hậu nghiệm cực đại để rút ra tiêu chuẩn tối ưu khái quát của tác vụ cá nhân hóa xếp

<sup>383</sup> [https://en.wikipedia.org/wiki/Discounted\\_cumulative\\_gain](https://en.wikipedia.org/wiki/Discounted_cumulative_gain)

hạng.

$$\begin{aligned}
 \text{BPR-OPT} &:= \ln p(\Theta | >_u) \\
 &\propto \ln p(>_u | \Theta) p(\Theta) \\
 &= \ln \prod_{(u,i,j \in D)} \sigma(\hat{y}_{ui} - \hat{y}_{uj}) p(\Theta) \\
 &= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) + \ln p(\Theta) \\
 &= \sum_{(u,i,j \in D)} \ln \sigma(\hat{y}_{ui} - \hat{y}_{uj}) - \lambda_\Theta \|\Theta\|^2
 \end{aligned} \tag{18.5.2}$$

trong đó  $D := \{(u, i, j) \mid i \in I_u^+ \wedge j \in I \setminus I_u^+\}$  là tập huấn luyện, với  $I_u^+$  ký hiệu cho sản phẩm mà người dùng  $u$  thích,  $I$  ký hiệu cho toàn bộ sản phẩm, và  $I \setminus I_u^+$  là toàn bộ sản phẩm khác ngoại trừ các sản phẩm mà người dùng  $u$  không thích.  $\hat{y}_{ui}$  và  $\hat{y}_{uj}$  lần lượt là điểm số dự đoán của người dùng  $u$  đối với sản phẩm  $i$  và  $j$ . Tiên nghiệm  $p(\Theta)$  là một phân phối chuẩn với kỳ vọng bằng không và ma trận phương sai - hiệp phương sai  $\Sigma_\Theta$ . Ở đây ta coi  $\Sigma_\Theta = \lambda_\Theta I$ .

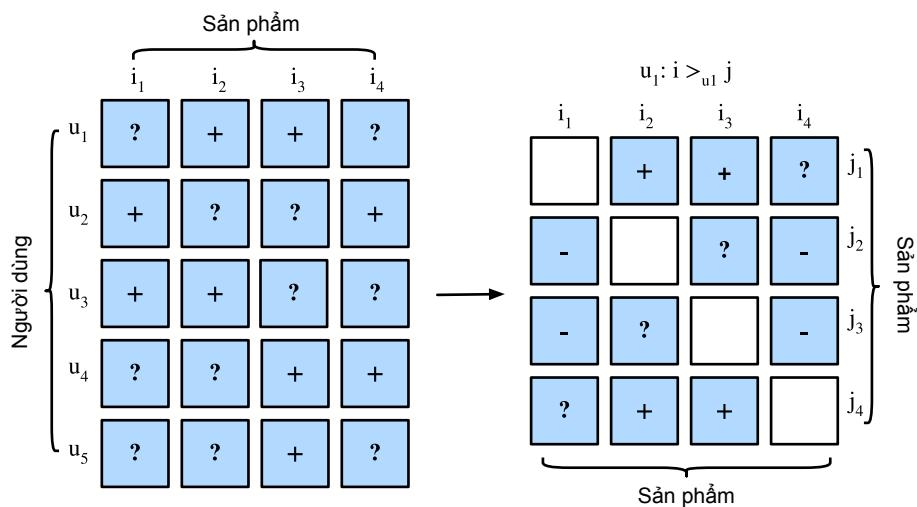


Fig. 18.5.1: Minh họa Cá nhân hóa Xếp hạng Bayes.

Ta sẽ lập trình lớp cơ sở `mxnet.gluon.loss.Loss` và ghi đè phương thức `forward` để xây dựng hàm mất mát cá nhân hóa xếp hạng Bayes. Ta bắt đầu bằng việc nhập lớp `Loss` và mô-đun `np`.

```
from mxnet import gluon, np, npx
npx.set_np()
```

Lập trình cho mất mát BPR như sau.

```
#@save
class BPRLoss(gluon.loss.Loss):
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(BPRLoss, self).__init__(weight=weight, batch_axis=batch_axis, **kwargs)

    def forward(self, positive, negative):
        distances = positive - negative
```

(continues on next page)

```
loss = - np.sum(np.log(npx.sigmoid(distances)), 0, keepdims=True)
return loss
```

### 18.5.2 Mất mát Hinge và Cách lập trình

Mất mát Hinge trong tác vụ xếp hạng có sự khác biệt so với [mất mát Hinge<sup>384</sup>](#) được cung cấp trong thư viện gluon thường sử dụng trong các bộ phân loại như SVM. Mất mát được sử dụng cho tác vụ xếp hạng trong hệ thống đề xuất có dạng như sau.

$$\sum_{(u,i,j \in D)} \max(m - \hat{y}_{ui} + \hat{y}_{uj}, 0) \quad (18.5.3)$$

trong đó  $m$  là khoảng cách biên an toàn. Mất mát này nhằm mục đích đẩy các sản phẩm tiêu cực ra xa khỏi các sản phẩm tích cực. Giống như BPR, nó nhằm tối ưu hóa khoảng cách thích đáng giữa mẫu dương và mẫu âm thay vì đầu ra tuyệt đối, khiến cho nó phù hợp với hệ thống đề xuất.

```
#@save
class HingeLossbRec(gluon.loss.Loss):
    def __init__(self, weight=None, batch_axis=0, **kwargs):
        super(HingeLossbRec, self).__init__(weight=None, batch_axis=0,
                                            **kwargs)

    def forward(self, positive, negative, margin=1):
        distances = positive - negative
        loss = np.sum(np.maximum(-distances + margin, 0))
        return loss
```

Hai loại mất mát này có thể thay thế lẫn nhau cho tác vụ cá nhân hóa xếp hạng trong hệ thống đề xuất.

### 18.5.3 Tóm tắt

- Có ba loại mất mát xếp hạng hiện có trong tác vụ cá nhân hóa xếp hạng trong hệ thống đề xuất, bao gồm các phương pháp theo từng điểm, theo từng cặp và theo danh sách.
- Hai loại mất mát theo cặp: mất mát cá nhân hóa xếp hạng Bayes và mất mát Hinge, có thể được sử dụng thay thế lẫn nhau.

### 18.5.4 Bài tập

- Liệu có biến thể nào khác của mất mát BPR và mất mát Hinge không?
- Bạn có thể tìm mô hình đề xuất nào khác sử dụng mất mát BPR hoặc mất mát Hinge không?

<sup>384</sup> <https://mxnet.incubator.apache.org/api/python/gluon/loss.html#mxnet.gluon.loss.HingeLoss>

### 18.5.5 Thảo luận

- Tiếng Anh: MXNet<sup>385</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>386</sup>

### 18.5.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật
- Lê Khắc Hồng Phúc

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

## 18.6 Lọc Cộng tác Nơ-ron cho Cá nhân hóa Xếp hạng

Vượt ra khỏi phản hồi trực tiếp, phần này sẽ giới thiệu một framework nơ-ron lọc cộng tác (*neural collaborative filtering framework* - NCF) cho bài toán đề xuất sử dụng phản hồi gián tiếp. Phản hồi gián tiếp có mặt khắp mọi nơi trong các hệ thống đề xuất. Các hành động như nhấn chọn, mua và xem là những phản hồi gián tiếp phổ biến có thể dễ dàng thu thập và thể hiện được sở thích của người dùng. Mô hình được giới thiệu là phân rã ma trận nơ-ron (*neural matrix factorization*) viết tắt là NeuMF (He et al., 2017b), hướng tới việc giải quyết tác vụ xếp hạng cá nhân hóa sử dụng phản hồi gián tiếp. Mô hình này tận dụng tính linh hoạt và tính phi tuyến của mạng nơ-ron để thay thế tích vô hướng trong phân rã ma trận, nhằm nâng cao tính biểu diễn của mô hình. Cụ thể, mô hình này gồm hai mạng con là phân rã ma trận tổng quát (*Generalized Matrix Factorization* - GMF) và MLP, và mô hình hóa các tương tác theo hai mạng này thay vì các tích vô hướng đơn giản. Kết quả đầu ra của hai mạng này được ghép nối với nhau để tính điểm dự đoán cuối cùng. Không giống như tác vụ dự đoán đánh giá trong AutoRec, mô hình này sinh ra danh sách đề xuất đã được xếp hạng cho từng người dùng dựa trên phản hồi gián tiếp. Chúng ta sẽ sử dụng mảng xếp hạng cá nhân hóa đã được giới thiệu trong phần trước để huấn luyện mô hình này.

<sup>385</sup> <https://discuss.d2l.ai/t/402>

<sup>386</sup> <https://forum.machinelearningcovan.com/c/d2l>

### 18.6.1 Mô hình NeuMF

Như đã đề cập, NeuMF kết hợp hai mạng con với nhau. GMF là một phiên bản mang nơ-ron tổng quát của phép phân rã ma trận, có đầu vào là tích theo từng phần tử giữa các đặc trưng ẩn của người dùng và sản phẩm. Nó bao gồm hai tầng nơ-ron sau:

$$\begin{aligned}\mathbf{x} &= \mathbf{p}_u \odot \mathbf{q}_i \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \mathbf{x}),\end{aligned}\tag{18.6.1}$$

trong đó  $\odot$  là phép nhân Hadamard của hai vector.  $\mathbf{P} \in \mathbb{R}^{m \times k}$  và  $\mathbf{Q} \in \mathbb{R}^{n \times k}$  lần lượt là ma trận đặc trưng tiềm ẩn của người dùng và sản phẩm.  $\mathbf{p}_u \in \mathbb{R}^k$  là hàng thứ  $u$  của ma trận  $P$  và  $\mathbf{q}_i \in \mathbb{R}^k$  hàng thứ  $i$  của ma trận  $Q$ .  $\alpha$  và  $h$  ký hiệu hàm kích hoạt và trọng số của tầng đầu ra.  $\hat{y}_{ui}$  là điểm dự đoán mà người dùng  $u$  có thể đưa ra cho sản phẩm  $i$ .

Thành phần còn lại của mô hình này là MLP. Để tăng tính linh hoạt của mô hình, MLP không dùng chung các embedding người dùng và sản phẩm với GMF, mà có đầu vào là vector ghép nối của hai embedding người dùng và sản phẩm. Với các kết nối phức tạp và các phép biến đổi phi tuyến, nó có thể ước lượng các tương tác phức tạp giữa người dùng và sản phẩm. Chính xác hơn, MLP được định nghĩa như sau:

$$\begin{aligned}z^{(1)} &= \phi_1(\mathbf{U}_u, \mathbf{V}_i) = [\mathbf{U}_u, \mathbf{V}_i] \\ \phi^{(2)}(z^{(1)}) &= \alpha^1(\mathbf{W}^{(2)} z^{(1)} + b^{(2)}) \\ &\dots \\ \phi^{(L)}(z^{(L-1)}) &= \alpha^L(\mathbf{W}^{(L)} z^{(L-1)} + b^{(L)}) \\ \hat{y}_{ui} &= \alpha(\mathbf{h}^\top \phi^L(z^{(L-1)}))\end{aligned}\tag{18.6.2}$$

trong đó  $\mathbf{W}^*$ ,  $\mathbf{b}^*$  và  $\alpha^*$  là ma trận trọng số, vector hệ số điều chỉnh, và hàm kích hoạt. Hàm của tầng tương ứng được ký hiệu là  $\phi^*$ . Đầu ra của tầng tương ứng được ký hiệu là  $\mathbf{z}^*$ .

Để kết hợp các đầu ra của GMF và MLP, thay vì phép cộng đơn giản, NeuMF ghép nối các tầng áp chót của hai mạng con để tạo thành một vector đặc trưng có thể được truyền vào các tầng tiếp theo. Sau đó, các đầu ra sẽ được chiếu bởi ma trận  $\mathbf{h}$  và hàm kích hoạt sigmoid. Tầng dự đoán có công thức như sau:

$$\hat{y}_{ui} = \sigma(\mathbf{h}^\top [\mathbf{x}, \phi^L(z^{(L-1)})]).\tag{18.6.3}$$

Hình dưới đây minh họa kiến trúc mô hình NeuMF.

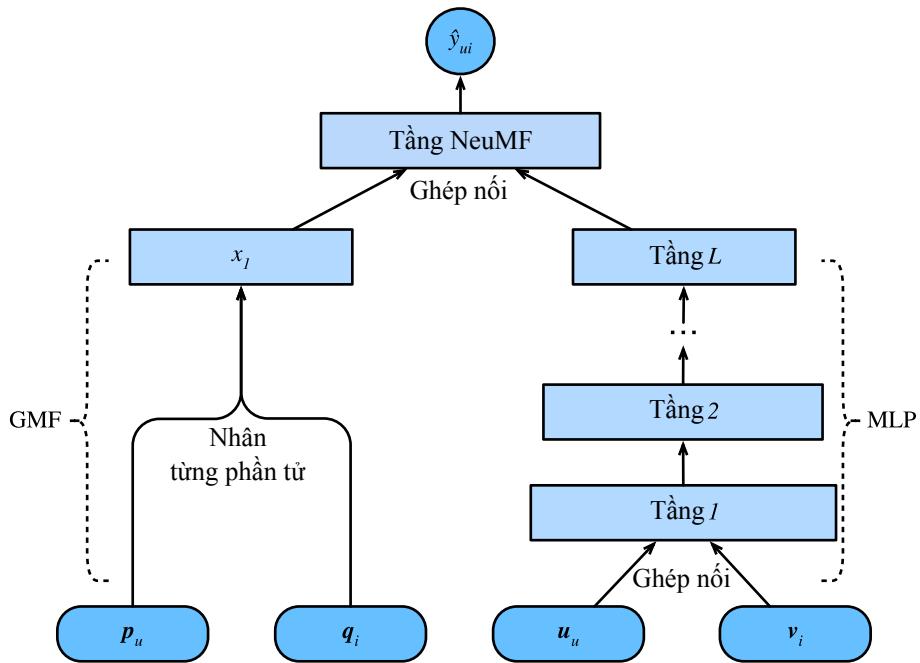


Fig. 18.6.1: Minh họa mô hình NeuMF

```
from d2l import mxnet as d2l
from mxnet import autograd, gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

## 18.6.2 Lập trình Mô hình

Đoạn mã dưới đây lập trình mô hình NeuMF, bao gồm GMF và MLP với các vector embedding người dùng và sản phẩm khác nhau. Kiến trúc của MLP được quy định qua tham số `nums_hiddens`. Hàm kích hoạt mặc định là ReLU.

```
class NeuMF(nn.Block):
    def __init__(self, num_factors, num_users, num_items, nums_hiddens,
                 **kwargs):
        super(NeuMF, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.U = nn.Embedding(num_users, num_factors)
        self.V = nn.Embedding(num_items, num_factors)
        self.mlp = nn.Sequential()
        for num_hiddens in nums_hiddens:
            self.mlp.add(nn.Dense(num_hiddens, activation='relu',
                                 use_bias=True))
        self.prediction_layer = nn.Dense(1, activation='sigmoid', use_bias=False)

    def forward(self, user_id, item_id):
```

(continues on next page)

```

p_mf = self.P(user_id)
q_mf = self.Q(item_id)
gmf = p_mf * q_mf
p_mlp = self.U(user_id)
q_mlp = self.V(item_id)
mlp = self.mlp(np.concatenate([p_mlp, q_mlp], axis=1))
con_res = np.concatenate([gmf, mlp], axis=1)
return self.prediction_layer(con_res)

```

### 18.6.3 Tập Dữ liệu Tùy chỉnh với phép Lấy mẫu Âm

Một bước quan trọng trong mất mát xếp hạng theo cặp là lấy mẫu âm. Với mỗi người dùng, các sản phẩm mà người đó chưa tương tác là các sản phẩm tiềm năng (các mục chưa được quan sát). Hàm dưới đây có đầu vào là danh tính người dùng và các sản phẩm tiềm năng, và lấy mẫu âm các sản phẩm ngẫu nhiên cho từng người dùng từ tập tiềm năng của người dùng đó. Trong quá trình huấn luyện, mô hình đảm bảo rằng các sản phẩm mà một người dùng thích sẽ được xếp hạng cao hơn các sản phẩm mà người này không thích hoặc chưa từng tương tác.

```

class PRDataset(gluon.data.Dataset):
    def __init__(self, users, items, candidates, num_items):
        self.users = users
        self.items = items
        self.cand = candidates
        self.all = set([i for i in range(num_items)])

    def __len__(self):
        return len(self.users)

    def __getitem__(self, idx):
        neg_items = list(self.all - set(self.cand[int(self.users[idx])]))
        indices = random.randint(0, len(neg_items) - 1)
        return self.users[idx], self.items[idx], neg_items[indices]

```

### 18.6.4 Đánh giá

Trong phần này, ta sẽ áp dụng chiến lược chia tách theo thời gian để xây dựng tập huấn luyện và tập kiểm tra. Hai phép đánh giá bao gồm tỷ lệ chọn đúng (*hit rate*) theo ngưỡng  $\ell$  (Hit@ $\ell$ ) cho trước và diện tích dưới đường cong ROC (AUC) được sử dụng để đánh giá hiệu quả của mô hình. Tỷ lệ chọn đúng tại ngưỡng  $\ell$  cho trước với mỗi người dùng cho thấy rằng liệu sản phẩm được đề xuất có được đưa vào danh sách  $\ell$  sản phẩm xếp hạng hàng đầu hay không. Định nghĩa toán học như sau:

$$\text{Hit}@{\ell} = \frac{1}{m} \sum_{u \in \mathcal{U}} \mathbf{1}(rank_{u,g_u} \leq \ell), \quad (18.6.4)$$

trong đó  $\mathbf{1}$  là hàm biến thi, hàm này bằng 1 nếu sản phẩm nhãn gốc được xếp hạng trong danh sách  $\ell$  sản phẩm hàng đầu, ngược lại hàm trả về 0.  $rank_{u,g_u}$  ký hiệu xếp hạng của sản phẩm nhãn gốc  $g_u$  của người dùng  $u$  trong danh sách đề xuất (xếp hạng lý tưởng là 1).  $m$  là số lượng người dùng.  $\mathcal{U}$  là tập người dùng.

Định nghĩa AUC được mô tả dưới đây:

$$AUC = \frac{1}{m} \sum_{u \in \mathcal{U}} \frac{1}{|\mathcal{I} \setminus S_u|} \sum_{j \in I \setminus S_u} \mathbf{1}(rank_{u,g_u} < rank_{u,j}), \quad (18.6.5)$$

trong đó  $\mathcal{I}$  là tập các sản phẩm.  $S_u$  là các sản phẩm tiềm năng của người dùng  $u$ . Chú ý rằng có rất nhiều phép đánh giá khác như precision, recall, hay NDCG (*Normalized Discounted Cumulative Gain*) cũng có thể được sử dụng.

Hàm sau đây tính toán số lần chọn đúng và AUC cho mỗi người dùng.

```
#@save
def hit_and_auc(rankedlist, test_matrix, k):
    hits_k = [(idx, val) for idx, val in enumerate(rankedlist[:k])
               if val in set(test_matrix)]
    hits_all = [(idx, val) for idx, val in enumerate(rankedlist)
                 if val in set(test_matrix)]
    max = len(rankedlist) - 1
    auc = 1.0 * (max - hits_all[0][0]) / max if len(hits_all) > 0 else 0
    return len(hits_k), auc
```

Sau đó, tỷ lệ chọn đúng và AUC tổng thể được tính như sau.

```
#@save
def evaluate_ranking(net, test_input, seq, candidates, num_users, num_items,
                     devices):
    ranked_list, ranked_items, hit_rate, auc = {}, {}, [], []
    all_items = set([i for i in range(num_items)])
    for u in range(num_users):
        neg_items = list(all_items - set(candidates[int(u)]))
        user_ids, item_ids, x, scores = [], [], [], []
        [item_ids.append(i) for i in neg_items]
        [user_ids.append(u) for _ in neg_items]
        x.extend([np.array(user_ids)])
        if seq is not None:
            x.append(seq[user_ids, :])
        x.extend([np.array(item_ids)])
        test_data_iter = gluon.data.DataLoader(
            gluon.data.ArrayDataset(*x), shuffle=False, last_batch="keep",
            batch_size=1024)
        for index, values in enumerate(test_data_iter):
            x = [gluon.utils.split_and_load(v, devices, even_split=False)
                  for v in values]
            scores.extend([list(net(*t).asnumpy()) for t in zip(*x)])
        scores = [item for sublist in scores for item in sublist]
        item_scores = list(zip(item_ids, scores))
        ranked_list[u] = sorted(item_scores, key=lambda t: t[1], reverse=True)
        ranked_items[u] = [r[0] for r in ranked_list[u]]
        temp = hit_and_auc(ranked_items[u], test_input[u], 50)
        hit_rate.append(temp[0])
        auc.append(temp[1])
    return np.mean(np.array(hit_rate)), np.mean(np.array(auc))
```

## 18.6.5 Huấn luyện và Đánh giá Mô hình

Hàm huấn luyện được định nghĩa như sau. Ta huấn luyện mô hình theo từng cặp.

```
#@save
def train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, devices, evaluator,
                  candidates, eval_step=1):
    timer, hit_rate, auc = d2l.Timer(), 0, 0
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs], ylim=[0, 1],
                            legend=['test hit rate', 'test AUC'])
    for epoch in range(num_epochs):
        metric, l = d2l.Accumulator(3), 0.
        for i, values in enumerate(train_iter):
            input_data = []
            for v in values:
                input_data.append(gluon.utils.split_and_load(v, devices))
            with autograd.record():
                p_pos = [net(*t) for t in zip(*input_data[0:-1])]
                p_neg = [net(*t) for t in zip(*input_data[0:-2],
                                              input_data[-1])]
                ls = [loss(p, n) for p, n in zip(p_pos, p_neg)]
                [l.backward(retain_graph=False) for l in ls]
                l += sum([l.asnumpy() for l in ls]).mean()/len(devices)
            trainer.step(values[0].shape[0])
            metric.add(l, values[0].shape[0], values[0].size)
        timer.stop()
        with autograd.predict_mode():
            if (epoch + 1) % eval_step == 0:
                hit_rate, auc = evaluator(net, test_iter, test_seq_iter,
                                           candidates, num_users, num_items,
                                           devices)
                animator.add(epoch + 1, (hit_rate, auc))
        print(f'train loss {metric[0] / metric[1]:.3f}, '
              f'test hit rate {float(hit_rate):.3f}, test AUC {float(auc):.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(devices)})')
```

Lúc này ta có thể nạp tập dữ liệu MovieLens 100k và huấn luyện mô hình. Vì tập dữ liệu MovieLens chỉ chứa các đánh giá xếp hạng, ta sẽ nhị phân hóa các đánh giá xếp hạng này thành 0 và 1 với một vài mất mát về độ chính xác. Nếu một người dùng đã đánh giá một sản phẩm, ta xem phản hồi gián tiếp bằng 1, bằng 0 nếu ngược lại. Hành động đánh giá một sản phẩm có thể được xem như là một hình thức cung cấp phản hồi gián tiếp. Ở đây, ta phân tách tập dữ liệu ở chế độ seq-aware, trong đó các sản phẩm được tương tác gần đây nhất sẽ được tách ra để kiểm tra.

```
batch_size = 1024
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                                'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_iter = gluon.data.DataLoader(
    PRDataset(users_train, items_train, candidates, num_items ), batch_size,
```

(continues on next page)

```
True, last_batch="rollover", num_workers=d2l.get_dataloader_workers())
```

Sau đó, ta tạo một mô hình và khởi tạo nó. Ta sử dụng mạng MLP 3 tầng với kích thước ẩn không đổi bằng 10.

```
devices = d2l.try_all_gpus()
net = NeuMF(10, num_users, num_items, nums_hiddens=[10, 10, 10])
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
```

Đoạn mã nguồn dưới đây được sử dụng để huấn luyện mô hình.

```
lr, num_epochs, wd, optimizer = 0.01, 10, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})
train_ranking(net, train_iter, test_iter, loss, trainer, None, num_users,
              num_items, num_epochs, devices, evaluate_ranking, candidates)
```

### 18.6.6 Tóm tắt

- Bổ sung thêm tính phi tuyến vào mô hình phân rã ma trận giúp cải thiện khả năng và tính hiệu quả của mô hình.
- NeuMF là sự kết hợp giữa mô hình phân rã ma trận và perceptron đa tầng. Perceptron đa tầng có đầu vào là vector được ghép nối bởi embedding người dùng và embedding sản phẩm.

### 18.6.7 Bài tập

- Thay đổi kích thước của các nhân tố tiềm ẩn. Kích thước này tác động như thế nào đến chất lượng mô hình?
- Thay đổi kiến trúc của MLP (ví dụ: số lượng tầng, số lượng nơ-ron của mỗi tầng) và cho biết tác động đến chất lượng của mô hình.
- Hãy thử các bộ tối ưu, tốc độ học và tốc độ suy giảm trọng số khác nhau.
- Hãy thử sử dụng mất mát hinge được định nghĩa ở phần trước để tối ưu mô hình này.

### 18.6.8 Thảo luận

- Tiếng Anh - MXNet<sup>387</sup>
- Tiếng Việt<sup>388</sup>

<sup>387</sup> <https://discuss.d2l.ai/t/403>

<sup>388</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 18.6.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Đỗ Trường Giang
- Nguyễn Văn Cường

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 20/09/2020)

## 18.7 Hệ thống Đề xuất có Nhận thức về Chuỗi

Trong phần trước, ta trừu tượng hóa tác vụ đề xuất dưới dạng một bài toán hoàn thiện ma trận mà không xét đến hành vi ngắn hạn của người dùng. Trong phần này, chúng tôi sẽ giới thiệu một mô hình đề xuất cân nhắc đến nhật ký tương tác được sắp xếp theo trình tự thời gian của người dùng. Đây là một hệ thống đề xuất có nhận thức về chuỗi (*sequence-aware recommender*) (Quadrana et al., 2018) với đầu vào là danh sách lịch sử thao tác của người dùng đã được sắp xếp và thường đi kèm với mốc thời gian diễn ra. Nhiều bài báo gần đây đã chứng minh được lợi ích của việc tích hợp những thông tin này vào việc mô hình hóa khuôn mẫu hành vi theo thời gian của người dùng và tìm ra được khuynh hướng sở thích của họ.

Mô hình mà chúng tôi sẽ giới thiệu, Caser (Tang & Wang, 2018), viết tắt của mô hình đề xuất embedding chuỗi tích chập (*convolutional sequence embedding recommendation model*), kế thừa mạng nơ-ron tích chập nhằm nắm bắt khuôn mẫu động có ảnh hưởng đến những hoạt động gần đây của người dùng. Thành phần chính của Caser bao gồm một mạng tích chập ngang và một mạng tích chập dọc, nhằm lần lượt khám phá khuôn mẫu cấp liên kết (*union-level*) và cấp điểm (*point-level*) của chuỗi. Khuôn mẫu cấp điểm ám chỉ tác động của một sản phẩm riêng lẻ trong lịch sử của chuỗi lên sản phẩm mục tiêu, trong khi khuôn mẫu cấp liên kết ám chỉ ảnh hưởng của nhiều thao tác trước đó lên các mục tiêu kế tiếp. Ví dụ, việc mua sữa cùng với bơ dẫn tới xác suất mua thêm cả bột mì cao hơn so với việc chỉ mua một trong hai. Hơn nữa, sở thích chung của người dùng, hay sở thích dài hạn cũng được mô hình hóa trong những tầng kết nối đầy đủ cuối cùng, dẫn đến sở thích của người dùng được mô hình hóa một cách toàn diện hơn. Chi tiết về mô hình này sẽ được mô tả tiếp theo.

### 18.7.1 Kiến trúc Mô hình

Trong hệ thống đề xuất có nhận thức về chuỗi, mỗi người dùng tương tác với một chuỗi các sản phẩm từ tập sản phẩm.  $S^u = (S_1^u, \dots, S_{|S_u|}^u)$  ký hiệu chuỗi có trình tự. Mục tiêu của Caser là đề xuất sản phẩm bằng cách xét thị hiếu chung của người dùng cũng như dự định ngắn hạn. Giả sử ta xét  $L$  sản phẩm trước, ma trận embedding biểu diễn những tương tác xảy ra trước bước thời gian  $t$  có thể được xây dựng như sau:

$$\mathbf{E}^{(u,t)} = [\mathbf{q}_{S_{t-L}^u}, \dots, \mathbf{q}_{S_{t-2}^u}, \mathbf{q}_{S_{t-1}^u}]^\top, \quad (18.7.1)$$

trong đó  $\mathbf{Q} \in \mathbb{R}^{n \times k}$  biểu diễn embedding sản phẩm và  $\mathbf{q}_i$  ký hiệu hàng thứ  $i$ .  $\mathbf{E}^{(u,t)} \in \mathbb{R}^{L \times k}$  có thể được sử dụng để suy ra sở thích nhất thời của người dùng  $u$  tại bước thời gian  $t$ . Ta có thể coi ma trận đầu vào  $\mathbf{E}^{(u,t)}$  như một ảnh đầu vào của hai tầng tích chập kế tiếp.

Tầng tích chập ngang có  $d$  bộ lọc ngang  $\mathbf{F}^j \in \mathbb{R}^{h \times k}, 1 \leq j \leq d, h = \{1, \dots, L\}$ , và tầng tích chập dọc có  $d'$  bộ lọc dọc  $\mathbf{G}^j \in \mathbb{R}^{L \times 1}, 1 \leq j \leq d'$ . Sau một chuỗi những thao tác tích chập và gộp, ta thu được hai đầu ra:

$$\begin{aligned}\mathbf{o} &= \text{HConv}(\mathbf{E}^{(u,t)}, \mathbf{F}) \\ \mathbf{o}' &= \text{VConv}(\mathbf{E}^{(u,t)}, \mathbf{G}),\end{aligned}\tag{18.7.2}$$

trong đó  $\mathbf{o} \in \mathbb{R}^d$  là đầu ra của mạng tích chập ngang và  $\mathbf{o}' \in \mathbb{R}^{kd'}$  là đầu ra của mạng tích chập dọc. Để đơn giản, ta bỏ qua chi tiết của các thao tác tích chập và thao tác gộp. Chúng được nối với nhau và đưa vào một tầng nơ-ron kết nối đầy đủ để thu được dạng biểu diễn cấp cao hơn.

$$\mathbf{z} = \phi(\mathbf{W}[\mathbf{o}, \mathbf{o}']^\top + \mathbf{b}),\tag{18.7.3}$$

trong đó  $\mathbf{W} \in \mathbb{R}^{k \times (d+kd')}$  là ma trận trọng số và  $\mathbf{b} \in \mathbb{R}^k$  là hệ số điều chỉnh. Vector học được  $\mathbf{z} \in \mathbb{R}^k$  chính là dạng biểu diễn cho sở thích ngắn hạn của người dùng.

Cuối cùng, hàm dự đoán kết hợp thị hiếu ngắn hạn và thị hiếu chung của người dùng với nhau, hàm này được định nghĩa:

$$\hat{y}_{uit} = \mathbf{v}_i \cdot [\mathbf{z}, \mathbf{p}_u]^\top + \mathbf{b}'_i,\tag{18.7.4}$$

trong đó  $\mathbf{V} \in \mathbb{R}^{n \times 2k}$  là một ma trận embedding sản phẩm khác.  $\mathbf{b}' \in \mathbb{R}^n$  là độ chêch đặc thù của sản phẩm.  $\mathbf{P} \in \mathbb{R}^{m \times k}$  là ma trận embedding thị hiếu chung của người dùng.  $\mathbf{p}_u \in \mathbb{R}^k$  là hàng thứ  $u$  của  $\mathbf{P}$  và  $\mathbf{v}_i \in \mathbb{R}^{2k}$  là hàng thứ  $i$  của  $\mathbf{V}$ .

Mô hình này có thể được học với mất mát BPR hoặc mất mát Hinge. Kiến trúc của Caser được mô tả như dưới đây.

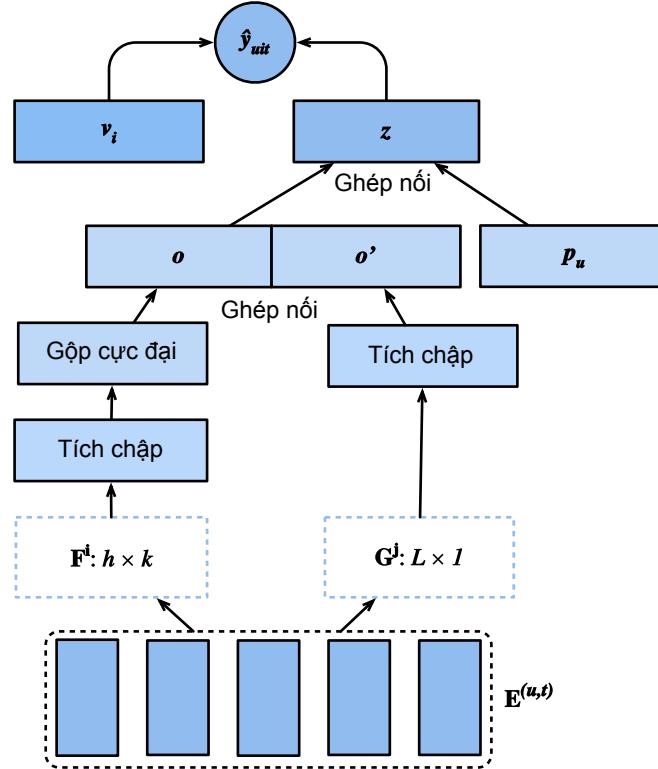


Fig. 18.7.1: Minh họa Mô hình Caser.

Đầu tiên ta nhập vào những thư viện cần thiết.

```
from d2l import mxnet as d2l
from mxnet import gluon, np, npx
from mxnet.gluon import nn
import mxnet as mx
import random
import sys
npx.set_np()
```

### 18.7.2 Lập trình Mô hình

Đoạn mã dưới đây lập trình cho mô hình Caser. Nó bao gồm một tầng tích chập ngang, một tầng tích chập dọc, và một tầng kết nối đầy đủ.

```
class Caser(nn.Block):
    def __init__(self, num_factors, num_users, num_items, L=5, d=16,
                 d_prime=4, drop_ratio=0.05, **kwargs):
        super(Caser, self).__init__(**kwargs)
        self.P = nn.Embedding(num_users, num_factors)
        self.Q = nn.Embedding(num_items, num_factors)
        self.d_prime, self.d = d_prime, d
        # Vertical convolution layer
        self.conv_v = nn.Conv2D(d_prime, (L, 1), in_channels=1)
        # Horizontal convolution layer
        h = [i + 1 for i in range(L)]
        self.conv_h, self.max_pool = nn.Sequential(), nn.Sequential()
        for i in h:
            self.conv_h.add(nn.Conv2D(d, (i, num_factors), in_channels=1))
            self.max_pool.add(nn.MaxPool1D(L - i + 1))
        # Fully-connected layer
        self.fc1_dim_v, self.fc1_dim_h = d_prime * num_factors, d * len(h)
        self.fc = nn.Dense(in_units=d_prime * num_factors + d * L,
                           activation='relu', units=num_factors)
        self.Q_prime = nn.Embedding(num_items, num_factors * 2)
        self.b = nn.Embedding(num_items, 1)
        self.dropout = nn.Dropout(drop_ratio)

    def forward(self, user_id, seq, item_id):
        item_embs = np.expand_dims(self.Q(seq), 1)
        user_emb = self.P(user_id)
        out, out_h, out_v, out_hs = None, None, None, []
        if self.d_prime:
            out_v = self.conv_v(item_embs)
            out_v = out_v.reshape(out_v.shape[0], self.fc1_dim_v)
        if self.d:
            for conv, maxp in zip(self.conv_h, self.max_pool):
                conv_out = np.squeeze(npx.relu(conv(item_embs)), axis=3)
                t = maxp(conv_out)
                pool_out = np.squeeze(t, axis=2)
                out_hs.append(pool_out)
            out_h = np.concatenate(out_hs, axis=1)
        out = np.concatenate([out_v, out_h], axis=1)
        z = self.fc(self.dropout(out))
```

(continues on next page)

```

x = np.concatenate([z, user_emb], axis=1)
q_prime_i = np.squeeze(self.Q_prime(item_id))
b = np.squeeze(self.b(item_id))
res = (x * q_prime_i).sum(1) + b
return res

```

### 18.7.3 Tập dữ liệu Tuần tự với phép Lấy mẫu Âm

Để xử lý dữ liệu tương tác tuần tự, ta cần lập trình lại lớp Dataset. Đoạn mã sau đây tạo một lớp dataset mới có tên là SeqDataset. Với mỗi mẫu, lớp này trả về id của người dùng, một chuỗi  $L$  sản phẩm mà người này đã tương tác trước đó và sản phẩm tiếp theo mà người này sẽ tương tác làm mục tiêu. Hình dưới đây mô tả quá trình nạp dữ liệu với một người dùng. Giả sử người dùng này thích 9 bộ phim, ta sắp xếp 9 bộ phim này theo thứ tự thời gian. Bộ phim cuối cùng được bỏ ra ngoài để làm sản phẩm kiểm tra. Với 8 bộ phim còn lại, ta có thể tạo ba mẫu huấn luyện, với mỗi mẫu bao gồm một chuỗi gồm năm ( $L = 5$ ) bộ phim và bộ phim kế tiếp làm mục tiêu. Các mẫu âm cũng có thể được đưa vào trong tập dữ liệu tuỳ chỉnh.

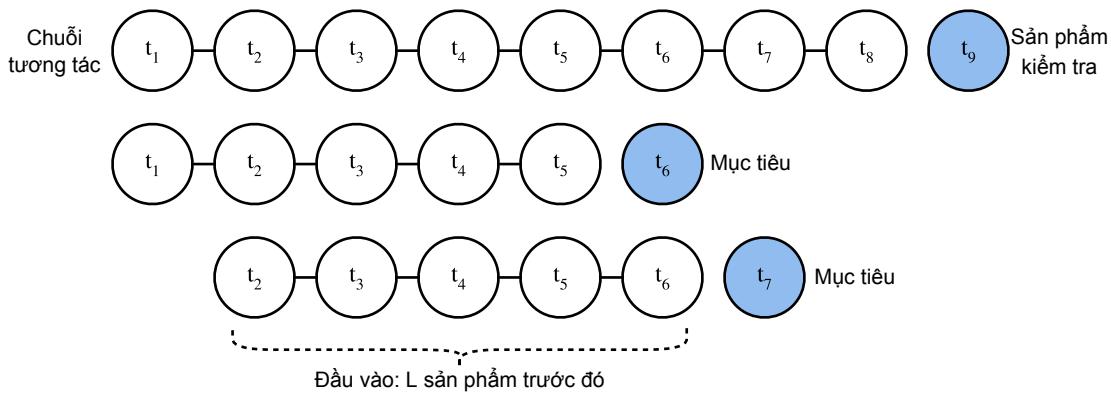


Fig. 18.7.2: Minh họa quá trình sinh dữ liệu

```

class SeqDataset(gluon.data.Dataset):
    def __init__(self, user_ids, item_ids, L, num_users, num_items,
                 candidates):
        user_ids, item_ids = np.array(user_ids), np.array(item_ids)
        sort_idx = np.array(sorted(range(len(user_ids)),
                                   key=lambda k: user_ids[k]))
        u_ids, i_ids = user_ids[sort_idx], item_ids[sort_idx]
        temp, u_ids, self.cand = {}, u_ids.astype(np.int32), candidates
        self.all_items = set([i for i in range(num_items)])
        [temp.setdefault(u_ids[i], []).append(i) for i, _ in enumerate(u_ids)]
        temp = sorted(temp.items(), key=lambda x: x[0])
        u_ids = np.array([i[0] for i in temp])
        idx = np.array([i[1][0] for i in temp])
        self.ns = ns = int(sum([c - L if c >= L + 1 else 1 for c
                               in np.array([len(i[1]) for i in temp])]))
        self.seq_items = np.zeros((ns, L))
        self.seq_users = np.zeros(ns, dtype='int32')
        self.seq_tgt = np.zeros((ns, 1))

```

(continues on next page)

```

self.test_seq = np.zeros((num_users, L))
test_users, _uid = np.empty(num_users), None
for i, (uid, i_seq) in enumerate(self._seq(u_ids, i_ids, idx, L + 1)):
    if uid != _uid:
        self.test_seq[uid][:] = i_seq[-L:]
        test_users[uid], _uid = uid, uid
    self.seq_tgt[i][:] = i_seq[-1:]
    self.seq_items[i][:], self.seq_users[i] = i_seq[:L], uid

def _win(self, tensor, window_size, step_size=1):
    if len(tensor) - window_size >= 0:
        for i in range(len(tensor), 0, -step_size):
            if i - window_size >= 0:
                yield tensor[i - window_size:i]
            else:
                break
    else:
        yield tensor

def _seq(self, u_ids, i_ids, idx, max_len):
    for i in range(len(idx)):
        stop_idx = None if i >= len(idx) - 1 else int(idx[i + 1])
        for s in self._win(i_ids[int(idx[i]):stop_idx], max_len):
            yield (int(u_ids[i]), s)

def __len__(self):
    return self.ns

def __getitem__(self, idx):
    neg = list(self.all_items - set(self.cand[int(self.seq_users[idx])]))
    i = random.randint(0, len(neg) - 1)
    return (self.seq_users[idx], self.seq_items[idx], self.seq_tgt[idx],
            neg[i])

```

#### 18.7.4 Nạp Tập dữ liệu MovieLens 100K

Kế tiếp, ta đọc và chia nhỏ tập dữ liệu MovieLens 100K ở chế độ nhận thức về chuỗi và nạp tập huấn luyện với bộ nạp dữ liệu tuần tự đã lập trình phía trên.

```

TARGET_NUM, L, batch_size = 1, 5, 4096
df, num_users, num_items = d2l.read_data_ml100k()
train_data, test_data = d2l.split_data_ml100k(df, num_users, num_items,
                                                'seq-aware')
users_train, items_train, ratings_train, candidates = d2l.load_data_ml100k(
    train_data, num_users, num_items, feedback="implicit")
users_test, items_test, ratings_test, test_iter = d2l.load_data_ml100k(
    test_data, num_users, num_items, feedback="implicit")
train_seq_data = SeqDataset(users_train, items_train, L, num_users,
                            num_items, candidates)
train_iter = gluon.data.DataLoader(train_seq_data, batch_size, True,
                                   last_batch="rollover",
                                   num_workers=d2l.get_dataloader_workers())

```

(continues on next page)

```
test_seq_iter = train_seq_data.test_seq
train_seq_data[0]
```

Cấu trúc dữ liệu huấn luyện được chỉ ra như trên. Phần tử đầu tiên là id người dùng, kế tiếp là danh sách ba sản phẩm đầu tiên mà người dùng này thích ( $L = 3$ ), và phần tử cuối cùng là sản phẩm người dùng này thích sau ba sản phẩm trước.

### 18.7.5 Huấn luyện Mô hình

Giờ hãy cùng huấn luyện mô hình. Ta sử dụng thiết lập giống với NeuMF trong phần trước, bao gồm tốc độ học, bộ tối ưu, và  $k$ , để có thể so sánh kết quả.

```
devices = d2l.try_all_gpus()
net = Caser(10, num_users, num_items, L)
net.initialize(ctx=devices, force_reinit=True, init=mx.init.Normal(0.01))
lr, num_epochs, wd, optimizer = 0.04, 8, 1e-5, 'adam'
loss = d2l.BPRLoss()
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {"learning_rate": lr, 'wd': wd})

d2l.train_ranking(net, train_iter, test_iter, loss, trainer, test_seq_iter,
                  num_users, num_items, num_epochs, devices,
                  d2l.evaluate_ranking, candidates, eval_step=1)
```

### 18.7.6 Tóm tắt

- Suy luận về sở thích ngắn hạn và dài hạn của một người dùng có thể giúp việc dự đoán sản phẩm tiếp theo người này thích trở nên hiệu quả hơn.
- Mạng nơ-ron tích chập có thể được tận dụng để nắm bắt được sở thích ngắn hạn của người dùng từ chuỗi các tương tác.

### 18.7.7 Bài tập

- Thực hiện một nghiên cứu loại bỏ (*ablation study*) bằng cách bỏ một trong hai mạng tích chập ngang hoặc dọc, thành phần nào quan trọng hơn?
- Thay đổi siêu tham số  $L$ . Liệu lịch sử tương tác dài hơn có giúp tăng độ chính xác?
- Ngoài tác vụ đề xuất nhận thức về chuỗi như chúng tôi giới thiệu ở trên, có một loại tác vụ đề xuất nhận thức về chuỗi khác được gọi là đề xuất dựa theo phiên (*session-based recommendation*) (Hidasi et al., 2015). Bạn có thể giải thích sự khác nhau giữa hai tác vụ này không?

### 18.7.8 Thảo luận

- Tiếng Anh: MXNet<sup>389</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>390</sup>

### 18.7.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Lê Quang Nhật

Cập nhật lần cuối: 06/10/2020. (Cập nhật lần cuối từ nội dung gốc: 01/10/2020)

## 18.8 Hệ thống Đề xuất Giàu Đặc trưng

Dữ liệu tương tác là dấu hiệu cơ bản nhất chỉ ra sở thích và sự hứng thú của người dùng, đóng vai trò chủ chốt trong các mô hình được giới thiệu trong các phần trước. Tuy vậy, dữ liệu tương tác thường vô cùng thưa thớt và đôi lúc có thể có nhiều. Để khắc phục vấn đề này, ta có thể tích hợp các thông tin phụ như đặc trưng của sản phẩm, hồ sơ người dùng, và thậm chí là bối cảnh diễn ra sự tương tác vào mô hình đề xuất. Tận dụng các đặc trưng này có lợi trong việc đưa ra đề xuất, vì chúng có thể nói lên sở thích của người dùng, đặc biệt khi thiếu dữ liệu tương tác. Do đó, các mô hình dự đoán nên có khả năng xử lý những đặc trưng này, để có thể nhận thức được phần nào bối cảnh/nội dung. Để mô tả loại mô hình đề xuất này, chúng tôi giới thiệu một tác vụ khác sử dụng tỷ lệ nhấp chuột (*click-through rate - CTR*) cho tác vụ đề xuất quảng cáo trực tuyến (McMahan et al., 2013) và cũng giới thiệu một tập dữ liệu quảng cáo vô danh. Dịch vụ quảng cáo nhằm đổi tượng đã thu hút sự chú ý rộng rãi và thường được coi như một công cụ đề xuất. Đề xuất quảng cáo phù hợp với thị hiếu và sở thích cá nhân của người dùng là rất quan trọng trong việc cải thiện tỷ lệ nhấp chuột.

Các nhà tiếp thị số sử dụng quảng cáo trực tuyến để phát quảng cáo tới khách hàng. Tỷ lệ nhấp chuột là tỷ lệ số lần nhấp chuột nhận được trên số lần hiển thị quảng cáo, được biểu diễn dưới dạng phần trăm theo công thức:

$$\text{CTR} = \frac{\#\text{số lần nhấp chuột}}{\#\text{số lần hiển thị}} \times 100\%. \quad (18.8.1)$$

Tỷ lệ nhấp chuột là một dấu hiệu quan trọng cho thấy độ hiệu quả của thuật toán dự đoán. Dự đoán tỷ lệ nhấp chuột là tác vụ dự đoán tỷ lệ mà một đường dẫn trên mạng được nhấp vào. Mô hình dự đoán CTR không những có thể được áp dụng vào hệ thống quảng cáo nhằm đổi tượng mà còn trong hệ thống đề xuất sản phẩm nói chung (như phim ảnh, tin tức, đồ dùng), chiến dịch

<sup>389</sup> <https://discuss.d2l.ai/t/404>

<sup>390</sup> <https://forum.machinelearningcoban.com/c/d2l>

quảng cáo qua thư điện tử, và thậm chí là những công cụ tìm kiếm. Nó cũng liên quan mật thiết đến độ hài lòng của khách hàng, tỷ lệ chuyển đổi, và có thể giúp ích trong việc thiết lập mục tiêu của chiến dịch quảng cáo do có thể giúp nhà quảng cáo đặt ra những kỳ vọng phù hợp.

```
from collections import defaultdict
from d2l import mxnet as d2l
from mxnet import gluon, np
import os
```

### 18.8.1 Tập dữ liệu Quảng cáo Trực tuyến

Với những bước tiến đáng kể của Internet và công nghệ di động, quảng cáo trực tuyến đã trở thành một nguồn thu nhập quan trọng và sản sinh phần lớn doanh thu trong ngành công nghiệp Internet. Việc hiển thị quảng cáo có liên quan và thu hút sự chú ý của người dùng là rất quan trọng để biến những người dùng vãng lai trở thành những khách hàng trả tiền tiềm năng. Tập dữ liệu chúng tôi giới thiệu là một tập dữ liệu quảng cáo trực tuyến. Nó bao gồm 34 trường, với cột đầu tiên biểu diễn biến mục tiêu cho biết liệu một quảng cáo được nhấp vào (1) hay không (0). Tất cả các cột còn lại là các đặc trưng theo hạng mục. Các cột này có thể biểu diễn id của quảng cáo, id trang web hay ứng dụng, id thiết bị, thời gian, hồ sơ người dùng, v.v. Ngữ nghĩa thực tế của các đặc trưng này không được tiết lộ để ẩn danh hóa dữ liệu và bảo mật thông tin cá nhân.

Đoạn mã dưới đây tải tập dữ liệu về từ máy chủ của chúng tôi và lưu vào một thư mục.

```
#@save
d2l.DATA_HUB['ctr'] = (d2l.DATA_URL + 'ctr.zip',
                        'e18327c48c8e8e5c23da714dd614e390d369843f')

data_dir = d2l.download_extract('ctr')
```

Tập dữ liệu bao gồm tập huấn luyện và tập kiểm tra, gồm lần lượt 15000 và 3000 mẫu/dòng.

### 18.8.2 Wrapper Tập dữ liệu

Để thuận tiện trong việc nạp dữ liệu, ta lập trình lớp CTRDataset nạp vào tập dữ liệu quảng cáo từ tệp CSV và có thể được sử dụng bởi DataLoader.

```
#@save
class CTRDataset(gluon.data.Dataset):
    def __init__(self, data_path, feat_mapper=None, defaults=None,
                 min_threshold=4, num_feat=34):
        self.NUM_FEATS, self.count, self.data = num_feat, 0, {}
        feat_cnts = defaultdict(lambda: defaultdict(int))
        self.feat_mapper, self.defaults = feat_mapper, defaults
        self.field_dims = np.zeros(self.NUM_FEATS, dtype=np.int64)
        with open(data_path) as f:
            for line in f:
                instance = {}
                values = line.rstrip('\n').split('\t')
                if len(values) != self.NUM_FEATS + 1:
                    continue
                label = np.float32([0, 0])
```

(continues on next page)

```

label[int(values[0])] = 1
instance['y'] = [np.float32(values[0])]
for i in range(1, self.NUM_FEATS + 1):
    feat_cnts[i][values[i]] += 1
    instance.setdefault('x', []).append(values[i])
self.data[self.count] = instance
self.count = self.count + 1
if self.feat_mapper is None and self.defaults is None:
    feat_mapper = {i: {feat for feat, c in cnt.items() if c >=
                        min_threshold} for i, cnt in feat_cnts.items()}
    self.feat_mapper = {i: {feat: idx for idx, feat in enumerate(cnt)}}
                           for i, cnt in feat_mapper.items()}
    self.defaults = {i: len(cnt) for i, cnt in feat_mapper.items()}
for i, fm in self.feat_mapper.items():
    self.field_dims[i - 1] = len(fm) + 1
self.offsets = np.array((0, *np.cumsum(self.field_dims).asnumpy()
                           [-1:]))

def __len__(self):
    return self.count

def __getitem__(self, idx):
    feat = np.array([self.feat_mapper[i + 1].get(v, self.defaults[i + 1])
                    for i, v in enumerate(self.data[idx]['x'])])
    return feat + self.offsets, self.data[idx]['y']

```

Ví dụ dưới đây nạp tập huấn luyện và in ra bản ghi đầu tiên.

```

train_data = CTRDataset(os.path.join(data_dir, 'train.csv'))
train_data[0]

```

Như có thể thấy, toàn bộ 34 trường đều là đặc trưng theo hạng mục. Mỗi giá trị biểu diễn chỉ số one-hot của trường tương ứng. Nhãn 0 nghĩa là quảng cáo này không được nhấp vào. Lớp CTRDataset này cũng có thể được sử dụng để nạp các tập dữ liệu khác như tập dữ liệu trong cuộc thi hiển thị quảng cáo Criteo<sup>391</sup>. và tập dữ liệu dự đoán tỷ lệ nhấp chuột Avazu<sup>392</sup>.

### 18.8.3 Tóm tắt

- Tỷ lệ nhấp chuột là một phép đo quan trọng được sử dụng để đo độ hiệu quả của hệ thống quảng cáo và hệ thống đề xuất.
- Dự đoán tỷ lệ nhấp chuột thường được chuyển đổi thành bài toán phân loại nhị phân. Mục tiêu của bài toán là dự đoán liệu một quảng cáo/sản phẩm có được nhấp vào hay không dựa vào các đặc trưng cho trước.

<sup>391</sup> <https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/>

<sup>392</sup> <https://www.kaggle.com/c/avazu-ctr-prediction>

#### 18.8.4 Bài tập

Bạn có thể nạp tập dữ liệu Criteo và Avazu với CTRDataset đã được cung cấp không? Chú ý rằng tập dữ liệu Criteo gồm các đặc trưng mang giá trị số thực nên bạn có thể phải chỉnh sửa lại đoạn mã một chút.

#### 18.8.5 Thảo luận

- Tiếng Anh: MXNet<sup>393</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>394</sup>

#### 18.8.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

Cập nhật lần cuối: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

### 18.9 Máy Phân rã ma trận

Máy phân rã ma trận (*Factorization machines - FM*) (Rendle, 2010), được đề xuất bởi Steffen Rendle vào năm 2010, là một thuật toán học có giám sát, có thể sử dụng trong các tác vụ phân loại, hồi quy và xếp hạng. Nó nhanh chóng nhận được sự chú ý và trở thành một phương pháp phổ biến và có ảnh hưởng lớn trong tác vụ dự đoán và đề xuất. Cụ thể, đây là sự tổng quát hóa của hồi quy tuyến tính và phân rã ma trận, hơn nữa còn gợi nhớ đến máy vector hỗ trợ với hạt nhân đa thức. Điểm mạnh của máy phân rã ma trận so với hồi quy tuyến tính và phân ra ma trận là: (1) Nó có thể mô hình hóa tương tác biến  $\chi$  chiều, với  $\chi$  là bậc của đa thức và thường được đặt bằng hai. (2) Một thuật toán tối ưu tốc độ cao đi kèm với máy phân rã ma trận có thể giảm độ phức tạp tính toán từ đa thức về còn tuyến tính, hiệu quả đặc biệt cao với đầu vào thừa nhiều chiều. Với các lý do trên, máy phân rã được áp dụng rộng rãi trong ngành quảng cáo hiện đại và đề xuất sản phẩm. Chi tiết kỹ thuật cũng như cách lập trình được mô tả dưới đây.

<sup>393</sup> <https://discuss.d2l.ai/t/405>

<sup>394</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 18.9.1 Máy Phân rã 2 Chiều.

Gọi  $x \in \mathbb{R}^d$  là vector đặc trưng của một mẫu, và  $y$  là nhãn tương ứng, nhãn này có thể mang giá trị thực hoặc là nhãn lớp như lớp nhị phân “nhấp chuột/chưa nhấp chuột”. Mô hình của máy phân rã ma trận bậc hai được định nghĩa như sau:

$$\hat{y}(x) = \mathbf{w}_0 + \sum_{i=1}^d \mathbf{w}_i x_i + \sum_{i=1}^d \sum_{j=i+1}^d \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \quad (18.9.1)$$

trong đó  $\mathbf{w}_0 \in \mathbb{R}$  là hệ số điều chỉnh toàn cục;  $\mathbf{w} \in \mathbb{R}^d$  là trọng số của biến thứ  $i$ ;  $\mathbf{V} \in \mathbb{R}^{d \times k}$  là embedding đặc trưng;  $\mathbf{v}_i$  biểu diễn hàng thứ  $i$  của  $\mathbf{V}$ ;  $k$  là số chiều của nhân tố tiềm ẩn;  $\langle \cdot, \cdot \rangle$  là tích vô hướng của hai vector.  $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$  mô hình hóa sự tương tác giữa đặc trưng thứ  $i$  và thứ  $j$ . Một số tương tác đặc trưng có thể dễ dàng hiểu được cho nên chúng có thể được thiết kế bởi các chuyên gia. Tuy nhiên, đa số các tương tác đặc trưng khác thường ẩn trong dữ liệu và khó có thể nhận biết. Do đó việc tự động mô hình hóa tương tác đặc trưng có thể giảm đáng kể công sức thiết kế đặc trưng (*feature engineering*). Ta có thể thấy rõ rằng hai số hạng đầu tiên tương ứng với mô hình hồi quy tuyến tính và số hạng cuối cùng là dạng mở rộng của mô hình phân rã ma trận. Nếu đặc trưng  $i$  biểu diễn một sản phẩm và đặc trưng  $j$  biểu diễn một người dùng, số hạng thứ ba chính là tích vô hướng giữa embedding người dùng và sản phẩm. Đáng chú ý là FM cũng có thể khai quát hóa với bậc cao hơn (bậc  $> 2$ ). Tuy vậy, tính ổn định số học khi tính toán có thể cản trở khả năng khai quát hóa.

### 18.9.2 Tiêu chuẩn Tối ưu Hiệu quả

Tối ưu máy phân rã ma trận theo cách thức trực tiếp dẫn đến độ phức tạp  $\mathcal{O}(kd^2)$  do ta phải tính toán toàn bộ các cặp tương tác. Để giải quyết vấn đề này, ta có thể biến đổi lại số hạng thứ ba của FM để giảm đáng kể chi phí tính toán xuống còn tuyến tính ( $\mathcal{O}(kd)$ ). Công thức biến đổi của số hạng tương tác theo cặp như sau:

$$\begin{aligned} & \sum_{i=1}^d \sum_{j=i+1}^d \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j \\ &= \frac{1}{2} \sum_{i=1}^d \sum_{j=1}^d \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j - \frac{1}{2} \sum_{i=1}^d \langle \mathbf{v}_i, \mathbf{v}_i \rangle x_i x_i \\ &= \frac{1}{2} \left( \sum_{i=1}^d \sum_{j=1}^d \sum_{l=1}^k \mathbf{v}_{i,l} \mathbf{v}_{j,l} x_i x_j - \sum_{i=1}^d \sum_{l=1}^k \mathbf{v}_{i,l} \mathbf{v}_{i,l} x_i x_i \right) \\ &= \frac{1}{2} \sum_{l=1}^k \left( \left( \sum_{i=1}^d \mathbf{v}_{i,l} x_i \right) \left( \sum_{j=1}^d \mathbf{v}_{j,l} x_j \right) - \sum_{i=1}^d \mathbf{v}_{i,l}^2 x_i^2 \right) \\ &= \frac{1}{2} \sum_{l=1}^k \left( \left( \sum_{i=1}^d \mathbf{v}_{i,l} x_i \right)^2 - \sum_{i=1}^d \mathbf{v}_{i,l}^2 x_i^2 \right) \end{aligned} \quad (18.9.2)$$

Với biến đổi này, độ phức tạp của mô hình giảm đi đáng kể. Hơn nữa, với đặc trưng thừa, chỉ các phần tử khác 0 cần phải tính toán nên độ phức tạp toàn phần là tuyến tính với số đặc trưng khác 0.

Để học mô hình FM, ta có thể sử dụng mất mát MSE cho tác vụ hồi quy, mất mát entropy chéo với tác vụ phân loại, và mất mát BPR với tác vụ xếp hạng. Các bộ tối ưu chuẩn như SGD và Adam đều khả thi cho việc tối ưu.

```

from d2l import mxnet as d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import os
import sys
npx.set_np()

```

### 18.9.3 Cách lập trình Mô hình

Đoạn mã sau đây lập trình mô hình máy phân rã ma trận. Ta có thể thấy rõ rằng FM bao gồm một khối hồi quy tuyến tính và một khối tương tác đặc trưng có hiệu suất cao. Ta áp dụng hàm sigmoid lên kết quả cuối cùng do ta coi dự đoán CTR như một tác vụ phân loại.

```

class FM(nn.Block):
    def __init__(self, field_dims, num_factors):
        super(FM, self).__init__()
        num_inputs = int(sum(field_dims))
        self.embedding = nn.Embedding(num_inputs, num_factors)
        self.fc = nn.Embedding(num_inputs, 1)
        self.linear_layer = nn.Dense(1, use_bias=True)

    def forward(self, x):
        square_of_sum = np.sum(self.embedding(x), axis=1) ** 2
        sum_of_square = np.sum(self.embedding(x) ** 2, axis=1)
        x = self.linear_layer(self.fc(x).sum(1)) \
            + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True)
        x = npx.sigmoid(x)
        return x

```

### 18.9.4 Nạp Tập dữ liệu Quảng cáo

Ta sử dụng lớp wrapper dữ liệu CTR từ phần trước để nạp tập dữ liệu quảng cáo trực tuyến.

```

batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(os.path.join(data_dir, 'train.csv'))
test_data = d2l.CTRDataset(os.path.join(data_dir, 'test.csv'),
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
train_iter = gluon.data.DataLoader(
    train_data, shuffle=True, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(
    test_data, shuffle=False, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())

```

### 18.9.5 Huấn luyện mô hình

Cuối cùng, ta tiến hành huấn luyện mô hình. Tốc độ học được đặt bằng 0.02 và kích thước embedding mặc định bằng 20. Ta sử dụng bộ tối ưu Adam và mất mát SigmoidBinaryCrossEntropyLoss để huấn luyện mô hình.

```
devices = d2l.try_all_gpus()
net = FM(train_data.field_dims, num_factors=20)
net.initialize(init.Xavier(), ctx=devices)
lr, num_epochs, optimizer = 0.02, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)
```

### 18.9.6 Tóm tắt

- FM là một framework tổng quát có thể áp dụng cho nhiều tác vụ khác nhau như hồi quy, phân loại hay xếp hạng.
- Tương tác/tương giao đặc trưng (*feature interaction/crossing*) rất quan trọng trong tác vụ dự đoán. Tương tác hai chiều có thể được mô hình hóa một cách hiệu quả với FM.

### 18.9.7 Bài tập

- Thử FM trên một tập dữ liệu khác như Avazu, MovieLens, and Criteo.
- Thay đổi kích thước embedding để kiểm tra ảnh hưởng của nó lên hiệu năng, so sánh với khi thay đổi kích thước embedding của phân rã ma trận.

### 18.9.8 Thảo luận

- Tiếng Anh: MXNet<sup>395</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>396</sup>

### 18.9.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

Cập nhật lần cuối: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 21/07/2020)

<sup>395</sup> <https://discuss.d2l.ai/t/406>

<sup>396</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 18.10 Máy Phân rã Ma trận Sâu

Việc học những tổ hợp đặc trưng hiệu quả rất quan trọng đối với sự thành công của tác vụ dự đoán tỷ lệ nhấp chuột. Máy phân rã ma trận mô hình hóa các tương tác đặc trưng dưới dạng tuyến tính (ví dụ như tương tác song tuyến tính). Điều này thường không đủ đối với dữ liệu thực tế khi bàn thân việc kết hợp chéo các đặc trưng thường có cấu trúc rất phức tạp và có dạng phi tuyến. Tệ hơn, máy phân rã ma trận trong thực tế thường sử dụng các tương tác đặc trưng bậc hai. Mô hình hóa tổ hợp tương tác với bậc cao hơn tuy khả thi về lý thuyết nhưng thường không được sử dụng do tính bất ổn số học và độ phức tạp tính toán cao.

Một giải pháp hiệu quả hơn là sử dụng mạng nơ-ron sâu. Mạng nơ-ron sâu rất hiệu quả khi học biểu diễn đặc trưng và có thể học được những tương tác đặc trưng tinh xảo. Do đó, việc tích hợp chúng vào máy phân rã ma trận cũng dễ hiểu. Việc thêm các tầng biến đổi phi tuyến vào máy phân rã ma trận giúp mô hình hóa cả những tổ hợp đặc trưng bậc thấp và bậc cao. Hơn nữa, bàn thân cấu trúc phi tuyến của đầu vào cũng có thể được nắm bắt thông qua mạng nơ-ron sâu. Trong phần này, chúng tôi sẽ giới thiệu một mô hình biểu diễn được gọi là máy phân rã ma trận sâu (*Deep factorization machines - DeepFM*) (Guo et al., 2017) kết hợp giữa FM và mạng nơ-ron sâu.

### 18.10.1 Kiến trúc Mô hình

DeepFM bao gồm một thành phần FM và một mạng sâu được tích hợp theo cấu trúc song song. FM là máy phân rã ma trận 2 chiều dùng để mô hình hóa tương tác đặc trưng bậc thấp. Mạng sâu là một perceptron đa tầng dùng để nắm bắt tương tác đặc trưng bậc cao và tính phi tuyến. Hai thành phần này có chung đầu vào/embedding và tổng đầu ra của chúng được lấy làm dự đoán cuối cùng. Điều đáng nói là ý tưởng của DeepFM tương tự với kiến trúc Rộng & Sâu, là kiến trúc có thể nắm bắt được cả sự ghi nhớ và tính khái quát. DeepFM lợi thế hơn mô hình Rộng & Sâu ở chỗ nó giảm tải việc thiết kế đặc trưng một cách thủ công bằng cách tự động nhận biết tổ hợp đặc trưng.

Để ngắn gọn, ta bỏ qua phần mô tả FM và ký hiệu đầu ra của thành phần này là  $\hat{y}^{(FM)}$ . Độc giả có thể tham khảo phần trước để biết thêm chi tiết. Gọi  $\mathbf{e}_i \in \mathbb{R}^k$  là vector đặc trưng tiềm ẩn của trường thứ  $i$ . Đầu vào của mạng sâu là tổ hợp của embedding dày đặc của tất cả các trường có thể được truy xuất với đầu vào đặc trưng danh mục thưa, ký hiệu là:

$$\mathbf{z}^{(0)} = [\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_f], \quad (18.10.1)$$

trong đó  $f$  là số trường. Sau đó nó được đưa vào mạng nơ-ron sau:

$$\mathbf{z}^{(l)} = \alpha(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)} + \mathbf{b}^{(l)}), \quad (18.10.2)$$

trong đó  $\alpha$  là hàm kích hoạt.  $\mathbf{W}_l$  và  $\mathbf{b}_l$  là trọng số và hệ số điều chỉnh tại tầng thứ  $l$ . Gọi  $y_{DNN}$  là đầu ra của dự đoán. Dự đoán cuối cùng của DeepFM là tổng đầu ra từ cả FM và DNN:

$$\hat{y} = \sigma(\hat{y}^{(FM)} + \hat{y}^{(DNN)}), \quad (18.10.3)$$

Trong đó  $\sigma$  là hàm sigmoid. Kiến trúc của DeepFM được minh họa như hình dưới.

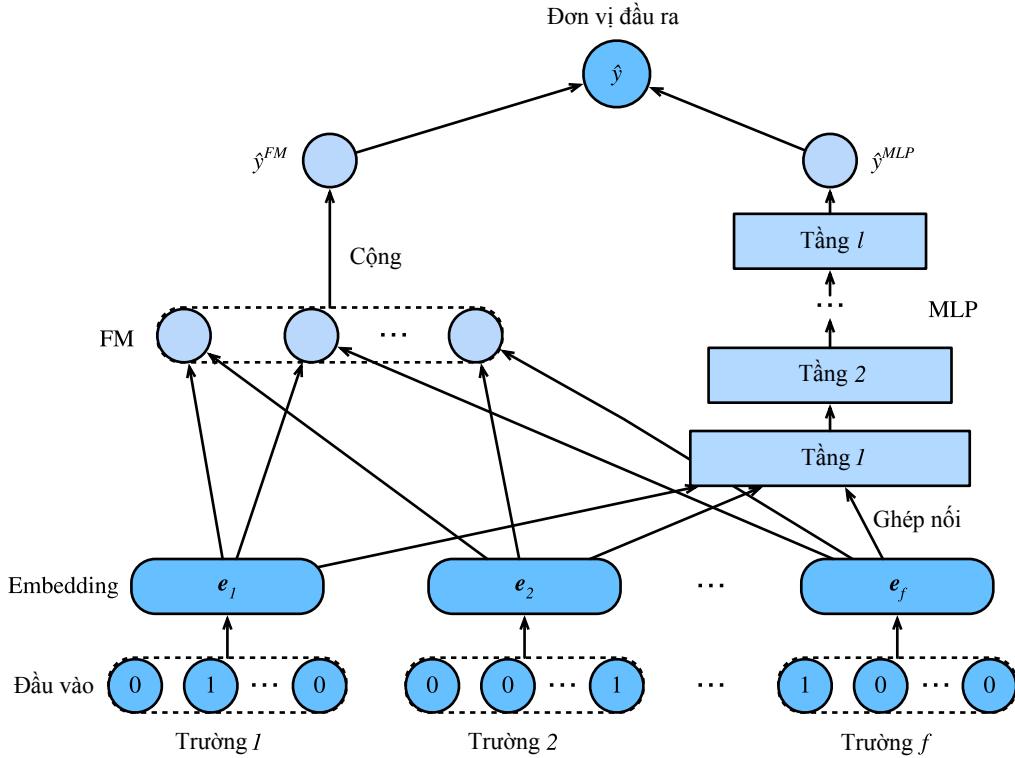


Fig. 18.10.1: Minh họa mô hình DeepFM.

Đáng chú ý rằng DeepFM không phải là cách duy nhất để kết hợp mạng nơ-ron sâu với FM. Ta cũng có thể thêm các tầng phi tuyến vào giữa các tương tác đặc trưng (He & Chua, 2017).

```
from d2l import mxnet as d2l
from mxnet import init, gluon, np, npx
from mxnet.gluon import nn
import os
import sys
npx.set_np()
```

## 18.10.2 Lập trình DeepFM

Cách lập trình cho DeepFM tương tự như FM. Ta giữ nguyên FM và sử dụng khối MLP với hàm kích hoạt relu. Dropout cũng được sử dụng để điều chỉnh mô hình. Số nơ-ron của MLP có thể được điều chỉnh thông qua siêu tham số mlp\_dims.

```
class DeepFM(nn.Block):
    def __init__(self, field_dims, num_factors, mlp_dims, drop_rate=0.1):
        super(DeepFM, self).__init__()
        num_inputs = int(sum(field_dims))
        self.embedding = nn.Embedding(num_inputs, num_factors)
        self.fc = nn.Embedding(num_inputs, 1)
        self.linear_layer = nn.Dense(1, use_bias=True)
        input_dim = self.embed_output_dim = len(field_dims) * num_factors
```

(continues on next page)

```

self.mlp = nn.Sequential()
for dim in mlp_dims:
    self.mlp.add(nn.Dense(dim, 'relu', True, in_units=input_dim))
    self.mlp.add(nn.Dropout(rate=drop_rate))
    input_dim = dim
self.mlp.add(nn.Dense(in_units=input_dim, units=1))

def forward(self, x):
    embed_x = self.embedding(x)
    square_of_sum = np.sum(embed_x, axis=1) ** 2
    sum_of_square = np.sum(embed_x ** 2, axis=1)
    inputs = np.reshape(embed_x, (-1, self.embed_output_dim))
    x = self.linear_layer(self.fc(x).sum(1)) \
        + 0.5 * (square_of_sum - sum_of_square).sum(1, keepdims=True) \
        + self.mlp(inputs)
    x = npx.sigmoid(x)
    return x

```

### 18.10.3 Huấn luyện và Đánh giá Mô hình

Quá trình nạp dữ liệu giống với FM. Ta đặt thành phần MLP của DeepFM là một mạng có ba tầng kết nối đầy đủ với cấu trúc kim tự tháp (30-20-10). Tất cả các siêu tham số khác được giữ nguyên so với FM.

```

batch_size = 2048
data_dir = d2l.download_extract('ctr')
train_data = d2l.CTRDataset(os.path.join(data_dir, 'train.csv'))
test_data = d2l.CTRDataset(os.path.join(data_dir, 'test.csv'),
                           feat_mapper=train_data.feat_mapper,
                           defaults=train_data.defaults)
field_dims = train_data.field_dims
train_iter = gluon.data.DataLoader(
    train_data, shuffle=True, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
test_iter = gluon.data.DataLoader(
    test_data, shuffle=False, last_batch='rollover', batch_size=batch_size,
    num_workers=d2l.get_dataloader_workers())
devices = d2l.try_all_gpus()
net = DeepFM(field_dims, num_factors=10, mlp_dims=[30, 20, 10])
net.initialize(init.Xavier(), ctx=devices)
lr, num_epochs, optimizer = 0.01, 30, 'adam'
trainer = gluon.Trainer(net.collect_params(), optimizer,
                        {'learning_rate': lr})
loss = gluon.loss.SigmoidBinaryCrossEntropyLoss()
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num_epochs, devices)

```

So với FM, DeepFM hội tụ nhanh hơn và đạt được hiệu năng tốt hơn.

#### **18.10.4 Tóm tắt**

- Việc tích hợp mạng nơ-ron vào FM cho phép mô hình hóa các tương tác phức tạp và có bậc cao.
- DeepFM vượt trội so với FM nguyên bản trên tập dữ liệu quảng cáo.

#### **18.10.5 Bài tập**

- Thay đổi cấu trúc của MLP để kiểm tra ảnh hưởng của nó lên hiệu năng mô hình.
- Sử dụng tập dữ liệu Criteo và so sánh DeepFM với mô hình FM gốc.

#### **18.10.6 Thảo luận**

- Tiếng Anh: MXNet<sup>397</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>398</sup>

#### **18.10.7 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường

Cập nhật lần cuối: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 21/07/2020)

### **18.11 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Đỗ Trường Giang
- Nguyễn Văn Cường

Cập nhật lần cuối: 26/09/2020. (Cập nhật lần cuối từ nội dung gốc: 25/04/2020)

---

<sup>397</sup> <https://discuss.d2l.ai/t/407>

<sup>398</sup> <https://forum.machinelearningcoban.com/c/d2l>



# 19 | Mạng Đối sinh

## 19.1 Mạng Đối sinh

Xuyên suốt phần lớn cuốn sách này, ta đã nói về việc làm thế nào để thực hiện những dự đoán. Ở dưới dạng nào đi nữa, ta đã cho mạng nơ-ron sâu học cách ánh xạ từ các mẫu dữ liệu sang các nhãn. Kiểu học này được gọi là học phân biệt, ví dụ như phân biệt ảnh chó và mèo. Phân loại và hồi quy là hai ví dụ của việc học phân biệt. Mạng nơ-ron được huấn luyện bằng phương pháp lan truyền ngược đã đào lộn mọi thứ ta từng biết về học phân biệt trên các tập dữ liệu lớn phức tạp. Độ chính xác của tác vụ phân loại ảnh có độ phân giải cao đã đạt tới mức độ như người (với một số điều kiện) từ chỗ không thể sử dụng được chỉ trong 5-6 năm gần đây. May mắn cho bạn, sẽ không có một bài diễn thuyết nữa về các tác vụ phân biệt khác mà ở đó mạng nơ-ron sâu thực hiện tốt một cách đáng kinh ngạc.

Nhưng học máy còn làm được nhiều hơn là chỉ giải quyết các tác vụ phân biệt. Chẳng hạn, với một tập dữ liệu không nhãn cho trước, ta có thể xây dựng một mô hình nắm bắt chính xác các đặc tính của tập dữ liệu này. Với một mô hình như vậy, ta có thể tổng hợp ra các mẫu dữ liệu mới giống như phân phối của dữ liệu dùng để huấn luyện. Ví dụ, với một kho lớn dữ liệu ảnh khuôn mặt cho trước, ta có thể tạo ra một ảnh như thật, giống như nó được lấy từ cùng tập dữ liệu. Kiểu học này được gọi là mô hình hóa tác vụ sinh (*generative modelling*).

Cho đến gần đây, ta không có phương pháp nào để có thể tổng hợp các ảnh mới như thật. Nhưng thành công của mạng nơ-ron sâu với học phân biệt đã mở ra những khả năng mới. Một xu hướng lớn trong hơn ba năm vừa qua là việc áp dụng mạng sâu phân biệt để vượt qua các thách thức trong các bài toán mà nhìn chung không được xem là học có giám sát. Các mô hình ngôn ngữ mạng nơ-ron hồi tiếp là một ví dụ về việc sử dụng một mạng phân biệt (được huấn luyện để dự đoán ký tự kế tiếp) mà một khi được huấn luyện có thể vận hành như một mô hình sinh.

Trong năm 2014, có một bài báo mang tính đột phá đã giới thiệu Mạng đối sinh (*Generative Adversarial Network - GAN*) (Goodfellow et al., 2014), một phương pháp khôn khéo tận dụng sức mạnh của các mô hình phân biệt để có được các mô hình sinh tốt. Về cốt lõi, GAN dựa trên ý tưởng là một bộ sinh dữ liệu là tốt nếu ta không thể chỉ ra đâu là dữ liệu giả và đâu là dữ liệu thật. Trong thống kê, điều này được gọi là bài kiểm tra từ hai tập mẫu - một bài kiểm tra để trả lời câu hỏi liệu tập dữ liệu  $X = \{x_1, \dots, x_n\}$  và  $X' = \{x'_1, \dots, x'_n\}$  có được rút ra từ cùng một phân phối. Sự khác biệt chính giữa hầu hết những bài nghiên cứu thống kê và GAN là GAN sử dụng ý tưởng này theo kiểu có tính cách xây dựng. Nói cách khác, thay vì chỉ huấn luyện một mô hình để nói “này, hai tập dữ liệu này có vẻ như không đến từ cùng một phân phối”, thì chúng sử dụng **phương pháp kiểm tra trên hai tập mẫu**<sup>399</sup> để cung cấp tín hiệu cho việc huấn luyện cho một mô hình sinh. Điều này cho phép ta cải thiện bộ sinh dữ liệu tới khi nó sinh ra thứ gì đó giống như dữ liệu thực. Ở mức tối thiểu nhất, nó cần lừa được bộ phân loại, kể cả nếu bộ phân loại của ta là một mạng nơ-ron sâu tân tiến nhất.

<sup>399</sup> [https://en.wikipedia.org/wiki/Two-sample\\_hypothesis\\_testing](https://en.wikipedia.org/wiki/Two-sample_hypothesis_testing)

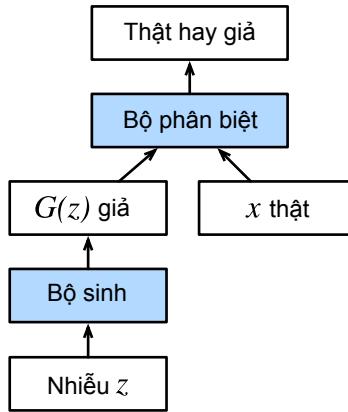


Fig. 19.1.1: Mạng Đối Sinh

Kiến trúc của mạng đối sinh được miêu tả trong hình Fig. 19.1.1. Như ta có thể thấy, có hai thành phần trong kiến trúc của GAN - đầu tiên, ta cần một thiết bị (giả sử, một mạng sâu nhưng nó có thể là bất kỳ thứ gì, chẳng hạn như công cụ kết xuất đồ họa trò chơi) có khả năng tạo ra dữ liệu giống thật. Nếu ta đang làm việc với hình ảnh, mô hình cần tạo ra hình ảnh. Nếu ta đang làm việc với giọng nói, mô hình cần tạo ra được chuỗi âm thanh, v.v. Ta gọi mô hình này là *mạng sinh (generator network)*. Thành phần thứ hai là *mạng phân biệt (discriminator network)*. Nó cố gắng phân biệt dữ liệu giả và thật. Cả hai mạng này sẽ cạnh tranh với nhau. Mạng sinh sẽ cố gắng đánh lừa mạng phân biệt. Đồng thời, mạng phân biệt sẽ thích nghi với dữ liệu giả vừa mới tạo ra. Thông tin thu được sẽ được dùng để cải thiện mạng sinh, và cứ tiếp tục như vậy.

Mạng phân biệt là một bộ phân loại nhị phân nhằm phân biệt xem đầu vào  $x$  là thật (từ dữ liệu thật) hoặc giả (từ mạng sinh). Thông thường, đầu ra của mạng phân biệt là một số vô hướng  $o \in \mathbb{R}$  dự đoán cho đầu vào  $\mathbf{x}$ , chẳng hạn như sử dụng một tầng kết nối đầy đủ với kích thước ẩn 1 và sau đó sẽ được đưa qua hàm sigmoid để nhận 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. Ta sẽ huấn luyện mạng phân biệt để cực tiểu hóa mất mát entropy chéo, *nghĩa là*,

$$\min_D \{-y \log D(\mathbf{x}) - (1-y) \log(1 - D(\mathbf{x}))\}, \quad (19.1.1)$$

Đối với mạng sinh, trước tiên nó tạo ra một vài tham số ngẫu nhiên  $\mathbf{z} \in \mathbb{R}^d$  từ một nguồn, ví dụ, phân phối chuẩn  $\mathbf{z} \sim \mathcal{N}(0, 1)$ . Ta thường gọi  $\mathbf{z}$  như là một biến tiềm ẩn. Mục tiêu của mạng sinh là đánh lừa mạng phân biệt để phân loại  $\mathbf{x}' = G(\mathbf{z})$  là dữ liệu thật, *nghĩa là*, ta muốn  $D(G(\mathbf{z})) \approx 1$ . Nói cách khác, cho trước một mạng phân biệt  $D$ , ta sẽ cập nhật tham số của mạng sinh  $G$  nhằm cực đại hóa mất mát entropy chéo khi  $y = 0$ , *tức là*,

$$\max_G \{-(1-y) \log(1 - D(G(\mathbf{z})))\} = \max_G \{-\log(1 - D(G(\mathbf{z})))\}. \quad (19.1.2)$$

Nếu như mạng sinh làm tốt, thì  $D(\mathbf{x}') \approx 1$  để mất mát gần 0, kết quả là các gradient sẽ trở nên quá nhỏ để tạo ra được sự tiến bộ đáng kể cho mạng phân biệt. Vì vậy, ta sẽ cực tiểu hóa mất mát như sau:

$$\min_G \{-y \log(D(G(\mathbf{z})))\} = \min_G \{-\log(D(G(\mathbf{z})))\}, \quad (19.1.3)$$

trong đó chỉ đưa  $\mathbf{x}' = G(\mathbf{z})$  vào mạng phân biệt nhưng cho trước nhãn  $y = 1$ .

Nói tóm lại,  $D$  và  $G$  đang chơi trò “minimax” (cực tiểu hóa cực đại) với một hàm mục tiêu toàn diện như sau:

$$\min_D \max_G \{-E_{x \sim \text{Data}} \log D(\mathbf{x}) - E_{z \sim \text{Noise}} \log(1 - D(G(\mathbf{z})))\}. \quad (19.1.4)$$

Rất nhiều ứng dụng của GAN liên quan tới hình ảnh. Để ví dụ, chúng ta sẽ bắt đầu với việc khớp một phân phối đơn giản trước. Ta sẽ minh họa bằng việc cho thấy việc gì sẽ xảy ra nếu sử dụng GAN để tạo một bộ ước lượng kém hiệu quả nhất thế giới cho một phân phối Gauss. Hãy tiến hành nào.

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, gluon, init, np, npx
from mxnet.gluon import nn
npx.set_np()
```

### 19.1.1 Sinh một vài Dữ liệu “thật”

Vì đây có thể là một ví dụ nhảm chán nhất, ta chỉ đơn giản sinh dữ liệu lấy từ một phân phối Gauss.

```
X = np.random.normal(0.0, 1, (1000, 2))
A = np.array([[1, 2], [-0.1, 0.5]])
b = np.array([1, 2])
data = np.dot(X, A) + b
```

Dựa vào đoạn mã trên, dữ liệu này là một phân phối Gauss được dịch chuyển một cách tùy ý với trung bình  $b$  và ma trận hiệp phương sai  $A^T A$ .

```
d2l.set_figsize()
d2l.plt.scatter(d2l.numpy(data[:100, 0]), d2l.numpy(data[:100, 1]));
print(f'The covariance matrix is\n{np.dot(A.T, A)}')
```

```
batch_size = 8
data_iter = d2l.load_array((data,), batch_size)
```

### 19.1.2 Bộ Sinh

Bộ sinh sẽ là một mạng đơn giản nhất có thể - một mô hình tuyến tính đơn tầng. Đó là vì chúng ta sẽ sử dụng mạng tuyến tính này cùng với bộ sinh dữ liệu từ phân phối Gauss. Vậy nên, nó chỉ cần học những tham số của phân phối này để làm giả dữ liệu một cách hoàn hảo.

```
net_G = nn.Sequential()
net_G.add(nn.Dense(2))
```

### 19.1.3 Bộ Phân biệt

Đối với bộ phân biệt, nó sẽ hơi khác một chút: ta sẽ sử dụng một MLP 3 tầng để khiến mọi thứ trở nên thú vị hơn.

```
net_D = nn.Sequential()
net_D.add(nn.Dense(5, activation='tanh'),
          nn.Dense(3, activation='tanh'),
          nn.Dense(1))
```

### 19.1.4 Huấn luyện

Đầu tiên, ta định nghĩa một hàm để cập nhật bộ phân biệt.

```
#@save
def update_D(X, Z, net_D, net_G, loss, trainer_D):
    """Update discriminator."""
    batch_size = X.shape[0]
    ones = np.ones((batch_size,), ctx=X.ctx)
    zeros = np.zeros((batch_size,), ctx=X.ctx)
    with autograd.record():
        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) + loss(fake_Y, zeros)) / 2
    loss_D.backward()
    trainer_D.step(batch_size)
    return float(loss_D.sum())
```

Bộ sinh cũng được cập nhật theo cách tương tự. Ở đây, ta sử dụng lại làm mất mát entropy chéo nhưng thay 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 = np.ones((batch_size,), ctx=Z.ctx)
    with autograd.record():
        # We could reuse `fake_X` from `update_D` to save computation
        fake_X = net_G(Z)
        # Recomputing `fake_Y` is needed since `net_D` is changed
        fake_Y = net_D(fake_X)
        loss_G = loss(fake_Y, ones)
    loss_G.backward()
    trainer_G.step(batch_size)
    return float(loss_G.sum())
```

Cả bộ phân biệt lẫn bộ sinh hoạt động như một bộ hồi quy logistic nhị phân với mất mát entropy chéo. Ta sử dụng Adam để làm mượt quá trình huấn luyện. Với mỗi lần lặp, đầu tiên ta cập nhật bộ phân biệt và sau đó đến bộ sinh. Ta sẽ theo dõi cả giá trị mất mát lẫn những dữ liệu được sinh ra.

```

def train(net_D, net_G, data_iter, num_epochs, lr_D, lr_G, latent_dim, data):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True)
    trainer_D = gluon.Trainer(net_D.collect_params(),
        'adam', {'learning_rate': lr_D})
    trainer_G = gluon.Trainer(net_G.collect_params(),
        'adam', {'learning_rate': 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 = np.random.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 = np.random.normal(0, 1, size=(100, latent_dim))
        fake_X = net_G(Z).asnumpy()
        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, ta xác định các siêu tham số để khớp với phân phối Gauss.

```

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, d2l.numpy(data[:100]))

```

### 19.1.5 Tóm tắt

- Mạng đối sinh (*Generative adversarial networks - GAN*) được cấu thành bởi hai mạng sâu, bộ sinh và bộ phân biệt.
- Bộ sinh tạo các ảnh gần với ảnh thật nhất có thể nhằm đánh lừa bộ phân biệt, thông qua tối đa hóa mất mát entropy chéo, *nói cách khác*,  $\max \log(D(\mathbf{x}'))$ .
- Bộ phân biệt cố gắng phân biệt những ảnh được tạo với ảnh thật, thông qua tối thiểu hóa mất mát entropy chéo, *nói cách khác*,  $\min -y \log D(\mathbf{x}) - (1 - y) \log(1 - D(\mathbf{x}))$ .

### 19.1.6 Bài tập

Liệu có tồn tại điểm cân bằng mà tại đó bộ sinh là người chiến thắng, *nói cách khác*, bộ phân biệt không thể phân biệt được hai phân phối trên dữ liệu hữu hạn?

### 19.1.7 Thảo luận

- Tiếng Anh: MXNet<sup>400</sup>, PyTorch<sup>401</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>402</sup>

### 19.1.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Lý Phi Long
- Nguyễn Văn Cường
- Nguyễn Lê Quang Nhật
- Phạm Minh Đức
- Nguyễn Thái Bình

Lần cập nhật gần nhất: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 17/09/2020)

## 19.2 Mạng Đối sinh Tích chập Sâu

Trong Section 19.1, ta đã giới thiệu về những ý tưởng cơ bản ẩn sau cách hoạt động của GAN. Ta đã thấy được quá trình tạo mẫu từ các phân phối đơn giản, dễ-lấy-mẫu như phân phối đều hay phân phối chuẩn, và biến đổi chúng thành các mẫu phù hợp với phân phối của tập dữ liệu nào đó. Dù ví dụ cho GAN khớp với phân phối Gauss 2 chiều là một minh họa rõ ràng, nhưng nó không thật sự thú vị.

Trong phần này, chúng tôi sẽ trình bày cách dùng GAN để tạo ra những bức ảnh chân thực. Ta sẽ xây dựng mô hình dựa theo các mô hình GAN tích chập sâu (*deep convolutional GAN - DCGAN*) được giới thiệu trong (Radford et al., 2015). Bằng cách mượn kiến trúc tích chập đã được chứng minh là thành công với bài toán thị giác máy tính phân biệt, và bằng cách thông qua GAN, ta có thể dùng chúng làm đòn bẩy để tạo ra các hình ảnh chân thực.

<sup>400</sup> <https://discuss.d2l.ai/t/408>

<sup>401</sup> <https://discuss.d2l.ai/t/1082>

<sup>402</sup> <https://forum.machinelearningcoban.com/c/d2l>

```
from mxnet import gluon, init, np, npx
from mxnet.gluon import nn
from d2l import mxnet as d2l

npx.set_np()
```

### 19.2.1 Tập dữ liệu Pokemon

Ta sẽ sử dụng tập dữ liệu các nhân vật Pokemon từ [pokemondb<sup>403</sup>](#). Đầu tiên ta tải xuống, giải nén và nạp tập dữ liệu.

```
#@save
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                            'c065c0e2593b8b161a2d7873e42418bf6a21106c')

data_dir = d2l.download_extract('pokemon')
pokemon = gluon.data.vision.datasets.ImageFolderDataset(data_dir)
```

Ta thay đổi kích thước ảnh thành  $64 \times 64$ . Phép biến đổi ToTensor sẽ chiếu từng giá trị điểm ảnh vào khoảng  $[0, 1]$ , trong đó mạng sinh của ta sẽ dùng hàm tanh để thu được đầu ra trong khoảng  $[-1, 1]$ . Do đó ta chuẩn hóa dữ liệu với trung bình 0.5 và độ lệch chuẩn 0.5 để khớp với miền giá trị.

```
batch_size = 256
transformer = gluon.data.vision.transforms.Compose([
    gluon.data.vision.transforms.Resize(64),
    gluon.data.vision.transforms.ToTensor(),
    gluon.data.vision.transforms.Normalize(0.5, 0.5)
])
data_iter = gluon.data.DataLoader(
    pokemon.transform_first(transformer), batch_size=batch_size,
    shuffle=True, num_workers=d2l.get_dataloader_workers())
```

Hãy xem thử 20 hình đầu tiên.

```
d2l.set_figsize((4, 4))
for X, y in data_iter:
    imgs = X[0:20,:,:,:].transpose(0, 2, 3, 1)/2+0.5
    d2l.show_images(imgs, num_rows=4, num_cols=5)
    break
```

<sup>403</sup> <https://pokemondb.net/sprites>

## 19.2.2 Bộ Sinh

Bộ sinh sẽ ánh xạ biến nhiễu  $\mathbf{z} \in \mathbb{R}^d$ , một vector  $d$  chiều sang hình ảnh RGB với chiều rộng và chiều cao tương ứng là  $64 \times 64$ . Trong Section 15.11 ta đã giới thiệu về mạng tích chập đầy đủ, sử dụng tầng tích chập chuyển vị (tham khảo Section 15.10) để phóng to kích thước đầu vào. Khối cơ bản của bộ sinh gồm tầng tích chập chuyển vị, theo sau là chuẩn hóa theo batch và hàm kích hoạt ReLU.

```
class G_block(nn.Block):
    def __init__(self, channels, kernel_size=4,
                 strides=2, padding=1, **kwargs):
        super(G_block, self).__init__(**kwargs)
        self.conv2d_trans = nn.Conv2DTranspose(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.Activation('relu')

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d_trans(X)))
```

Mặc định, tầng tích chập chuyển vị dùng hạt nhân  $k_h = k_w = 4$ , sải bước  $s_h = s_w = 2$  và đệm  $p_h = p_w = 1$ . Với kích thước đầu vào  $n'_h \times n'_w = 16 \times 16$ , khối bộ sinh sẽ nhân đôi chiều rộng và chiều cao của đầu vào.

$$\begin{aligned} n'_h \times n'_w &= [(n_h k_h - (n_h - 1)(k_h - s_h) - 2p_h) \times (n_w k_w - (n_w - 1)(k_w - s_w) - 2p_w)] \\ &= [(k_h + s_h(n_h - 1) - 2p_h) \times (k_w + s_w(n_w - 1) - 2p_w)] \\ &= [(4 + 2 \times (16 - 1) - 2 \times 1) \times (4 + 2 \times (16 - 1) - 2 \times 1)] \\ &= 32 \times 32. \end{aligned} \tag{19.2.1}$$

```
x = np.zeros((2, 3, 16, 16))
g_blk = G_block(20)
g_blk.initialize()
g_blk(x).shape
```

Giả sử, ta đổi tầng tích chập chuyển vị này thành một hạt nhân  $4 \times 4$ , sải bước  $1 \times 1$  và đệm không. Với kích thước đầu vào là  $1 \times 1$ , chiều rộng và chiều cao của đầu ra sẽ tăng thêm 3 giá trị.

```
x = np.zeros((2, 3, 1, 1))
g_blk = G_block(20, strides=1, padding=0)
g_blk.initialize()
g_blk(x).shape
```

Bộ sinh bao gồm bốn khối cơ bản thực hiện tăng cả chiều rộng và chiều cao của đầu vào từ 1 lên 32. Cùng lúc đó, nó trước tiên biến đổi chiều cao của đầu vào thành  $64 \times 8$  kênh, rồi giảm một nửa số kênh sau mỗi lần. Cuối cùng, một tầng tích chập chuyển vị được sử dụng để sinh đầu ra. Nó tăng gấp đôi chiều rộng và chiều cao để khớp với kích thước mong muốn  $64 \times 64$ , và giảm kích thước kênh xuống 3. Hàm kích hoạt tanh được áp dụng để đưa giá trị đầu ra về khoảng  $(-1, 1)$ .

```
n_G = 64
net_G = nn.Sequential()
net_G.add(G_block(n_G*8, strides=1, padding=0), # Output: (64 * 8, 4, 4)
```

(continues on next page)

```
G_block(n_G*4), # Output: (64 * 4, 8, 8)
G_block(n_G*2), # Output: (64 * 2, 16, 16)
G_block(n_G), # Output: (64, 32, 32)
nn.Conv2DTranspose(
    3, kernel_size=4, strides=2, padding=1, use_bias=False,
    activation='tanh')) # Output: (3, 64, 64)
```

Hãy sinh một biến tiềm ẩn có số chiều là 100 để xác thực kích thước đầu ra của bộ sinh.

```
x = np.zeros((1, 100, 1, 1))
net_G.initialize()
net_G(x).shape
```

### 19.2.3 Bộ Phân biệt

Bộ phân biệt là một mạng tích chập thông thường ngoại trừ việc nó dùng hàm kích hoạt ReLU rò rỉ. Với  $\alpha \in [0, 1]$  cho trước, định nghĩa của nó là

$$\text{ReLU rò rỉ}(x) = \begin{cases} x & \text{nếu } x > 0 \\ \alpha x & \text{ngược lại} \end{cases}. \quad (19.2.2)$$

Như có thể thấy, nó là ReLU thông thường nếu  $\alpha = 0$ , và là hàm đồng nhất nếu  $\alpha = 1$ . Cho  $\alpha \in (0, 1)$ , ReLU rò rỉ là một hàm phi tuyến cho đầu ra khác không với giá trị đầu vào âm. Mục đích của hàm này là khắc phục vấn đề “ReLU chết”, khi mà một nơ-ron có thể luôn xuất giá trị âm và do đó không thể được cập nhật (gradient của ReLU luôn bằng 0).

```
alphas = [0, .2, .4, .6, .8, 1]
x = np.arange(-2, 1, 0.1)
Y = [d2l.numpy(nn.LeakyReLU(alpha)(x)) for alpha in alphas]
d2l.plot(d2l.numpy(x), Y, 'x', 'y', alphas)
```

Khối cơ bản của bộ phân biệt là một tầng tích chập, theo sau bởi tầng chuẩn hóa theo batch và một hàm kích hoạt ReLU rò rỉ. Các siêu tham số của tầng tích chập này tương tự như tầng tích chập chuyển vị trong khối sinh.

```
class D_block(nn.Block):
    def __init__(self, channels, kernel_size=4, strides=2,
                 padding=1, alpha=0.2, **kwargs):
        super(D_block, self).__init__(**kwargs)
        self.conv2d = nn.Conv2D(
            channels, kernel_size, strides, padding, use_bias=False)
        self.batch_norm = nn.BatchNorm()
        self.activation = nn.LeakyReLU(alpha)

    def forward(self, X):
        return self.activation(self.batch_norm(self.conv2d(X)))
```

Khối cơ bản với thiết lập mặc định sẽ giảm một nửa chiều rộng và chiều cao của đầu vào, như ta đã chứng tỏ trong Section 8.3. Chẳng hạn, cho kích thước đầu vào là  $n_h = n_w = 16$ , với một hạt nhân

có kích thước  $k_h = k_w = 4$ , sải bước  $s_h = s_w = 2$ , và đệm  $p_h = p_w = 1$ , kích thước đầu ra sẽ là:

$$\begin{aligned} n'_h \times n'_w &= \lfloor (n_h - k_h + 2p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + 2p_w + s_w)/s_w \rfloor \\ &= \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \times \lfloor (16 - 4 + 2 \times 1 + 2)/2 \rfloor \\ &= 8 \times 8. \end{aligned} \quad (19.2.3)$$

```
x = np.zeros((2, 3, 16, 16))
d_blk = D_block(20)
d_blk.initialize()
d_blk(x).shape
```

Bộ phân biệt là một tấm gương phản chiếu của bộ sinh.

```
n_D = 64
net_D = nn.Sequential()
net_D.add(D_block(n_D), # Output: (64, 32, 32)
          D_block(n_D*2), # Output: (64 * 2, 16, 16)
          D_block(n_D*4), # Output: (64 * 4, 8, 8)
          D_block(n_D*8), # Output: (64 * 8, 4, 4)
          nn.Conv2D(1, kernel_size=4, use_bias=False)) # Output: (1, 1, 1)
```

Nó sử dụng một tầng tích chập với kênh đầu ra 1 làm tầng cuối cùng để có được giá trị dự đoán duy nhất.

```
x = np.zeros((1, 3, 64, 64))
net_D.initialize()
net_D(x).shape
```

#### 19.2.4 Huấn luyện

So với mô hình GAN cơ bản trong Section 19.1, ta sử dụng cùng tốc độ học cho cả bộ sinh và bộ phân biệt do chúng tương đồng với nhau. Thêm nữa, ta thay đổi  $\beta_1$  trong Adam (Section 13.14) từ 0.9 về 0.5. Việc này làm giảm độ mượt của động lượng, tức là trung bình động trọng số mũ của các gradient trước đó, nhằm đáp ứng sự thay đổi nhanh chóng của gradient do bộ sinh và bộ phân biệt đối kháng lẫn nhau. Bên cạnh đó, nhiễu ngẫu nhiên  $z$  là một tensor 4-D và ta sử dụng GPU để tăng tốc độ tính toán.

```
def train(net_D, net_G, data_iter, num_epochs, lr, latent_dim,
          device=d2l.try_gpu()):
    loss = gluon.loss.SigmoidBCELoss()
    net_D.initialize(init=init.Normal(0.02), force_reinit=True, ctx=device)
    net_G.initialize(init=init.Normal(0.02), force_reinit=True, ctx=device)
    trainer_hp = {'learning_rate': lr, 'beta1': 0.5}
    trainer_D = gluon.Trainer(net_D.collect_params(), 'adam', trainer_hp)
    trainer_G = gluon.Trainer(net_G.collect_params(), 'adam', 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
```

(continues on next page)

```

timer = d2l.Timer()
metric = d2l.Accumulator(3) # loss_D, loss_G, num_examples
for X, _ in data_iter:
    batch_size = X.shape[0]
    Z = np.random.normal(0, 1, size=(batch_size, latent_dim, 1, 1))
    X, Z = X.as_in_ctx(device), Z.as_in_ctx(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 = np.random.normal(0, 1, size=(21, latent_dim, 1, 1), ctx=device)
# Normalize the synthetic data to N(0, 1)
fake_x = net_G(Z).transpose(0, 2, 3, 1) / 2 + 0.5
imgs = np.concatenate(
    [np.concatenate([fake_x[i * 7 + j] for j in range(7)], axis=1)
     for i in range(len(fake_x)//7)], axis=0)
animator.axes[1].cla()
animator.axes[1].imshow(imgs.astype(np.float32))
# 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)}')

```

Ta sẽ chỉ huấn luyện mô hình với số epoch nhỏ để minh họa. Để đạt chất lượng mô hình tốt hơn, bạn có thể đặt biến num\_epochs bằng một giá trị lớn hơn.

```

latent_dim, lr, num_epochs = 100, 0.005, 20
train(net_D, net_G, data_iter, num_epochs, lr, latent_dim)

```

### 19.2.5 Tóm tắt

- Kiến trúc DCGAN gồm có bốn tầng tích chập cho Bộ phân biệt, và bốn tầng tích chập “sải bước một phần (*fractionally-strided*)” cho Bộ sinh.
- Bộ phân biệt là một mạng 4 tầng bao gồm các tầng tích chập có sải bước, theo sau bởi tầng chuẩn hoá theo batch (trừ tầng đầu vào) và hàm kích hoạt ReLU rò rỉ.
- ReLU rò rỉ là một hàm phi tuyến trả về kết quả khác không với đầu vào âm. Hàm này nhằm khắc phục vấn đề “ReLU chết”, giúp gradient truyền đi dễ dàng hơn xuyên suốt kiến trúc.

### 19.2.6 Bài tập

1. Chuyện gì sẽ xảy ra nếu ta sử dụng hàm kích hoạt ReLU phổ thông thay vì ReLU rò rỉ?
2. Áp dụng DCGAN trên Fashion-MNIST và quan sát xem đối với hạng mục nào thì nó hoạt động tốt, hạng mục nào thì không.

### 19.2.7 Thảo luận

- Tiếng Anh: MXNet<sup>404</sup>, PyTorch<sup>405</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>406</sup>

### 19.2.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lý Phi Long
- Nguyễn Mai Hoàng Long
- Phạm Hồng Vinh
- Phạm Minh Đức
- Đỗ Trường Giang
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 05/10/2020. (Cập nhật lần cuối từ nội dung gốc: 17/09/2020)

---

<sup>404</sup> <https://discuss.d2l.ai/t/409>

<sup>405</sup> <https://discuss.d2l.ai/t/1083>

<sup>406</sup> <https://forum.machinelearningcoban.com/c/d2l>

# 20 | Phụ lục: Toán học cho Học Sâu

**Brent Werness** (*Amazon*), **Rachel Hu** (*Amazon*), và các tác giả của cuốn sách này.

Một trong những điểm tuyệt vời nhất của học sâu hiện đại là nó có thể được hiểu và sử dụng mà không cần hiểu cặn kẽ nền tảng toán học đằng sau. Đây là một dấu hiệu thể hiện lĩnh vực này đang trưởng thành. Giống như hầu hết các nhà phát triển phần mềm không cần bận tâm đến lý thuyết hàm số khả tính, những người làm việc với học sâu cũng không cần bận tâm đến nền tảng lý thuyết của học hợp lý cực đại (maximum likelihood).

Tuy nhiên, chúng ta chưa thật sự gần đến mức đó.

Trên thực tế, bạn sẽ thi thoảng cần hiểu sự lựa chọn kiến trúc sẽ ảnh hưởng tới dòng gradient như thế nào, hoặc những già thiết ngầm khi huấn luyện với một hàm mất mát cụ thể. Bạn có thể cần biết entropy đóng đếm thứ gì trên thế giới, và nó có thể giúp bạn hiểu chính xác số lượng bit trên một ký tự có ý nghĩa như thế nào trong mô hình của bạn. Tất cả những điều này đòi hỏi những hiểu biết toán học sâu hơn.

Phần phụ lục này nhằm cung cấp cho bạn nền tảng toán học cần thiết để hiểu lý thuyết cốt lõi của học sâu hiện đại, nhưng đây không phải là toàn bộ kiến thức cần thiết. Chúng ta sẽ bắt đầu xem xét đại số tuyến tính sâu hơn. Chúng tôi phát triển ý nghĩa hình học của các đại lượng và toán tử đại số tuyến tính, việc này cho phép chúng ta minh họa hiệu ứng của nhiều phép biến đổi dữ liệu. Một thành phần chủ chốt là sự phát triển của các kiến thức nền tảng liên quan tới phân tích trị riêng.

Tiếp theo, chúng ta phát triển lý thuyết giải tích vi phân để có thể hiểu cặn kẽ tại sao gradient là hướng hạ dốc nhất, và tại sao lan truyền ngược có công thức như vậy. Giải tích tích phân được thảo luận tiếp sau đó ở mức cần thiết để hỗ trợ chủ đề tiếp theo – lý thuyết xác suất.

Các vấn đề gặp phải trên thực tế thường không chắc chắn, và bởi vậy chúng ta cần một ngôn ngữ để nói về những điều không chắc chắn. Chúng ta sẽ ôn tập lại lý thuyết biến ngẫu nhiên và những phân phối thường gặp nhất để có thể thảo luận các mô hình dưới góc nhìn xác suất. Việc này cung cấp nền tảng cho bộ phân loại Naive Bayes, một phương pháp phân loại dựa trên xác suất.

Liên quan mật thiết đến lý thuyết xác suất là lý thuyết thống kê. Trong khi thống kê là một mảng quá lớn để ôn tập trong một mục ngắn, chúng tôi sẽ giới thiệu các khái niệm cơ bản mà mọi người làm học máy cần biết, cụ thể như: đánh giá và so sánh các bộ ước lượng, thực hiện kiểm chứng thống kê, và xây dựng khoảng tin cậy.

Cuối cùng, chúng ta sẽ thảo luận chủ đề lý thuyết thông tin qua nghiên cứu toán học về lưu trữ và truyền tải thông tin. Phần này cung cấp ngôn ngữ cơ bản ở đó chúng ta thảo luận một cách định lượng lượng thông tin một mô hình hàm chứa.

Kết hợp lại, những kiến thức này định hình những khái niệm toán học cốt lõi cần thiết để bắt đầu đi tới con đường hiểu sâu về học sâu.

## 20.1 Các phép toán Hình học và Đại số Tuyến tính

Trong Section 4.3, chúng ta đã đề cập tới những kiến thức cơ bản về đại số tuyến tính và cách nó được dùng để thể hiện các phép biến đổi dữ liệu cơ bản. Đại số tuyến tính là một trong những trụ cột toán học chính hỗ trợ học sâu và rộng hơn là học máy. Dù Section 4.3 đề cập đủ kiến thức cần thiết để tìm hiểu các mô hình học sâu hiện đại, vẫn còn rất nhiều điều cần thảo luận trong lĩnh vực này. Trong mục này, chúng ta sẽ đi sâu hơn, nhấn mạnh 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 trị riêng và vector riêng.

### 20.1.1 Ý nghĩa Hình học của Vector

Trước hết, chúng ta cần thảo luận hai diễn giải hình học phổ biến của vector: điểm hoặc hướng trong không gian. Về cơ bản, một vector là một danh sách các số giống như danh sách trong Python dưới đây:

```
v = [1, 7, 0, 1]
```

Các nhà toán học thường viết chúng dưới dạng một vector *cột* hoặc *hàng*, tức:

$$\mathbf{x} = \begin{bmatrix} 1 \\ 7 \\ 0 \\ 1 \end{bmatrix}, \quad (20.1.1)$$

hoặc

$$\mathbf{x}^\top = [1 \ 7 \ 0 \ 1]. \quad (20.1.2)$$

Những biểu diễn này thường có những cách diễn giải khác nhau. Các mẫu dữ liệu được biểu diễn bằng các vector cột và các trọng số dùng để tính các tổng có trọng số được biểu diễn bằng các vector hàng. Tuy nhiên, việc linh động sử dụng 2 cách biểu diễn này mang lại nhiều lợi ích. Như mô tả trong Section 4.3, dù cách biểu diễn mặc định của một vector đơn là theo cột, trong các ma trận biểu diễn các tập dữ liệu dạng bảng, các mẫu dữ liệu thường được coi như các vector hàng.

Cho trước một vector bất kỳ, cách hiểu thứ nhất là coi nó như một điểm trong không gian. Trong không gian hai hoặc ba chiều, chúng ta có thể biểu diễn điểm này bằng việc sử dụng các thành phần của vector để định nghĩa vị trí của điểm đó trong không gian so với một điểm tham chiếu được gọi là gốc *tọa độ*, như trong Fig. 20.1.1.

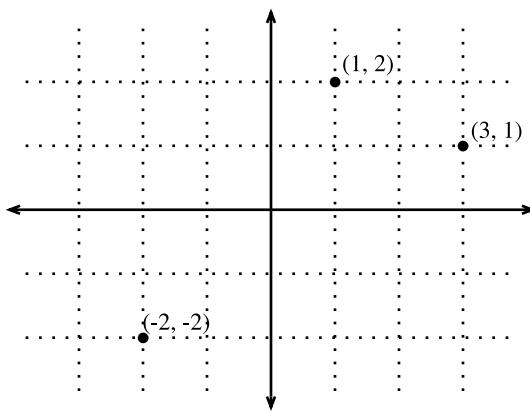


Fig. 20.1.1: Mô tả việc biểu diễn vector như các điểm trong mặt phẳng. Thành phần thứ nhất của vector là tọa độ  $x$ , thành phần thứ hai là tọa độ  $y$ . Biểu diễn tương tự với vector nhiều chiều hơn, mặc dù khó hình dung hơn.

Góc nhìn hình học này cho phép chúng ta xem xét bài toán ở mức trừu tượng hơn. Không giống như khi đối mặt với các bài toán khó hình dung như phân loại ảnh chó mèo, chúng ta có thể bắt đầu xem xét các bài toán dạng này một cách trừu tượng hơn: cho một tập hợp các điểm trong không gian, hãy tìm cách phân biệt hai nhóm điểm riêng biệt.

Cách thứ hai để giải thích một vector là coi nó như một phương hướng trong không gian. Chúng ta không những có thể coi vector  $\mathbf{v} = [2, 3]^\top$  là một điểm nằm bên phải 2 đơn vị và bên trên 3 đơn vị so với gốc tọa độ, chúng ta cũng có thể coi nó thể hiện một hướng – hướng về bên phải 2 đơn vị và hướng lên phía trên 3 đơn vị. Theo cách này, ta coi tất cả các vector trong Fig. 20.1.2 là như nhau.

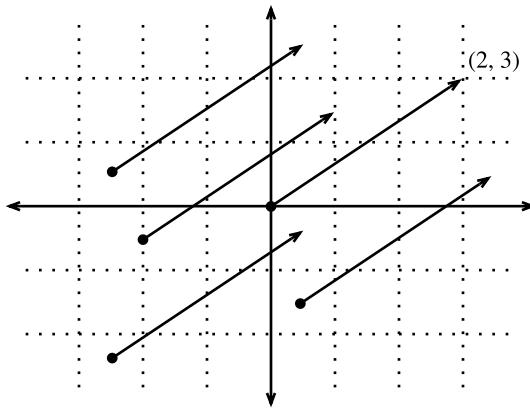


Fig. 20.1.2: Bất kỳ vector nào cũng có thể biểu diễn bằng một mũi tên trong mặt phẳng. Trong trường hợp này, mọi vector trong hình đều biểu diễn vector  $(3, 2)^\top$ .

Một trong những lợi ích của cách hiểu này là phép cộng vector có thể được hiểu theo nghĩa hình học. Cụ thể, chúng ta đi theo một hướng được cho bởi một vector, sau đó tiếp tục đi theo hướng cho bởi một vector khác, như trong Fig. 20.1.3.

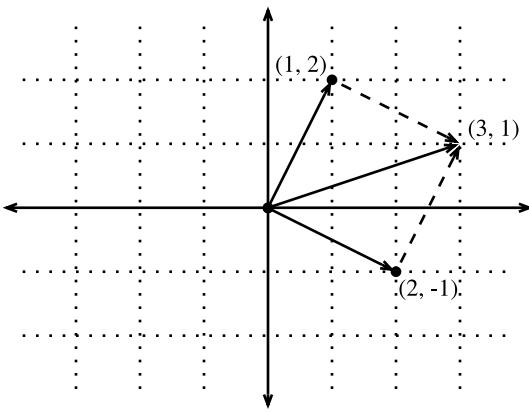


Fig. 20.1.3: Phép cộng vector có thể biểu diễn bằng cách đầu tiên đi theo một vector, sau đó đi theo vector kia.

Hiệu của hai vector có cách diễn giải tương tự. Bằng cách biểu diễn  $\mathbf{u} = \mathbf{v} + (\mathbf{u} - \mathbf{v})$ , ta thấy rằng vector  $\mathbf{u} - \mathbf{v}$  là hướng mang điểm  $\mathbf{v}$  tới điểm  $\mathbf{u}$ .

### 20.1.2 Tích vô hướng và Góc

Như đã thấy trong Section 4.3, tích vô hướng của hai vector cột  $\mathbf{u}$  và  $\mathbf{v}$  có thể được tính như sau:

$$\mathbf{u}^\top \mathbf{v} = \sum_i u_i \cdot v_i. \quad (20.1.3)$$

Vì biểu thức (20.1.3) là đối xứng, chúng ta có thể mượn ký hiệu của phép nhân truyền thống và viết:

$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^\top \mathbf{v} = \mathbf{v}^\top \mathbf{u}, \quad (20.1.4)$$

để nhấn mạnh rằng việc đổi chỗ hai vector sẽ cho kết quả như nhau.

Tích vô hướng (20.1.3) cũng có một cách diễn giải hình học: nó liên quan mật thiết tới góc giữa hai vector. Hãy xem xét góc trong Fig. 20.1.4.

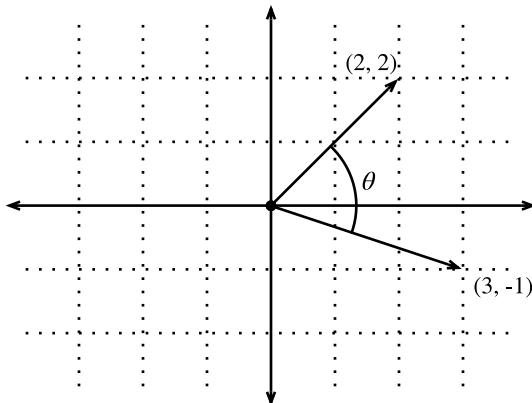


Fig. 20.1.4: Luôn tồn tại một góc xác định ( $\theta$ ) giữa hai vector bất kỳ trong không gian. Ta sẽ thấy rằng góc này có liên hệ chặt chẽ tới tích vô hướng.

Xét hai vector:

$$\mathbf{v} = (r, 0) \text{ and } \mathbf{w} = (s \cos(\theta), s \sin(\theta)). \quad (20.1.5)$$

Vector  $\mathbf{v}$  có độ dài  $r$  và song song với trục  $x$ , vector  $\mathbf{w}$  có độ dài  $s$  và tạo một góc  $\theta$  với trục  $x$ . Nếu tính tích vô hướng của hai vector này, ta sẽ thấy rằng

$$\mathbf{v} \cdot \mathbf{w} = rs \cos(\theta) = \|\mathbf{v}\| \|\mathbf{w}\| \cos(\theta). \quad (20.1.6)$$

Với một vài phép biến đổi đại số đơn giản, chúng ta có thể sắp xếp lại các thành phần để được

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|}\right). \quad (20.1.7)$$

Một cách ngắn gọn, với hai vector cụ thể này, tích vô hướng kết hợp với chuẩn (*norm*) cho ta góc giữa hai vector. Điều này cũng đúng trong trường hợp tổng quát.

Chúng tôi sẽ không suy ra biểu thức đó ở đây; tuy nhiên, nếu viết  $\|\mathbf{v} - \mathbf{w}\|^2$  bằng hai cách: cách thứ nhất với tích vô hướng, và cách thứ hai sử dụng công thức tính cô-sin, ta có thể thấy được quan hệ giữa chúng. Thật vậy, với hai vector  $\mathbf{v}$  và  $\mathbf{w}$  bất kỳ, góc giữa chúng là

$$\theta = \arccos\left(\frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|}\right). \quad (20.1.8)$$

Đây là một điều tốt vì trong công thức không hề chỉ định bất cứ điều gì đặc biệt về không gian hai chiều. Thực vậy, ta có thể sử dụng công thức này trong không gian ba chiều hoặc ba triệu chiều mà không gặp vấn đề gì.

Xét ví dụ đơn giản tính góc giữa cặp vector:

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import gluon, np, npx
npx.set_np()
def angle(v, w):
    return np.arccos(v.dot(w) / (np.linalg.norm(v) * np.linalg.norm(w)))
angle(np.array([0, 1, 2]), np.array([2, 3, 4]))
```

Chúng ta sẽ không sử dụng đoạn mã này bây giờ, nhưng sẽ hữu ích để biết rằng nếu góc giữa hai vector là  $\pi/2$  (hay  $90^\circ$ ) thì hai vector đó *trực giao* với nhau. Xem xét kỹ biểu thức trên, ta thấy rằng việc này xảy ra khi  $\theta = \pi/2$ , tức  $\cos(\theta) = 0$ . Điều này chứng tỏ tích vô hướng phải bằng không, và hai vector là trực giao khi và chỉ khi  $\mathbf{v} \cdot \mathbf{w} = 0$ . Đẳng thức này sẽ hữu ích khi xem xét các đối tượng dưới con mắt hình học.

Ta sẽ tự hỏi tại sao việc tính góc lại hữu ích? Câu trả lời nằm ở tính bất biến ta mong đợi từ dữ liệu. Xét một tấm ảnh, và một tấm ảnh thứ hai giống hệt nhưng với các điểm ảnh với độ sáng chỉ bằng 10% ảnh ban đầu. Giá trị của từng điểm ảnh trong ảnh thứ hai nhìn chung khác xa so với ảnh ban đầu. Bởi vậy, nếu tính khoảng cách giữa ảnh ban đầu và ảnh tối hơn, giá trị này có thể rất lớn. Tuy nhiên, trong hầu hết các ứng dụng học máy, *nội dung* của hai tấm ảnh là như nhau – nó vẫn là tấm ảnh của một con mèo đối với một bộ phân loại chó mèo. Tiếp đó, nếu xem xét góc giữa hai ảnh, không khó để thấy rằng với vector  $\mathbf{v}$  bất kỳ, góc giữa  $\mathbf{v}$  và  $0.1 \cdot \mathbf{v}$  bằng không. Việc này tương ứng

với việc nhân vector với một số (dương) đồng hướng và chỉ thay đổi độ dài của vector đó. Như vậy khi xét tới góc, hai tấm ảnh được xem là như nhau.

Ví dụ tương tự có thể tìm thấy bất cứ đâu. Trong văn bản, chúng ta có thể muốn chủ đề thảo luận không thay đổi cho dù tăng gấp đôi độ dài văn bản. Trong một số cách mã hóa (như đếm số lượng xuất hiện của một từ trong từ điển), việc này tương đương với nhân đôi vector mã hóa của văn bản, bởi vậy chúng ta lại có thể sử dụng góc.

### Độ tương tự Cô-sin

Trong cảnh học máy với góc được dùng để đo lường khoảng cách giữa hai vector, người làm học máy sử dụng thuật ngữ *độ tương tự cô-sin* để chỉ đại lượng

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|}. \quad (20.1.9)$$

Hàm cô-sin có giá trị lớn nhất bằng 1 khi hai vector chỉ cùng một hướng, giá trị nhỏ nhất bằng  $-1$  khi chúng cùng phương nhưng ngược hướng, và 0 khi hai vector trực giao. Chú ý rằng nếu các thành phần của hai vector nhiều chiều được lấy mẫu ngẫu nhiên với kỳ vọng 0, cô-sin giữa chúng sẽ luôn gần với 0.

### 20.1.3 Siêu phẳng

Ngoài làm việc với vector, một đối tượng quan trọng khác bạn phải nắm vững khi đi sâu vào đại số tuyến tính là *siêu phẳng*, một khái niệm tổng quát của đường thẳng (trong không gian hai chiều) hoặc một mặt phẳng (trong không gian ba chiều). Trong một không gian vector  $d$  chiều, một siêu phẳng có  $d - 1$  chiều và chia không gian thành hai nửa không gian.

Xét ví dụ sau. Giả sử ta có một vector cột  $\mathbf{w} = [2, 1]^\top$ . Ta muốn biết “những điểm  $\mathbf{v}$  nào thỏa mãn  $\mathbf{w} \cdot \mathbf{v} = 1?$ ” Sử dụng mối quan hệ giữa tích vô hướng và góc ở (20.1.8) phía trên, ta có thể thấy điều này tương đương với

$$\|\mathbf{v}\| \|\mathbf{w}\| \cos(\theta) = 1 \iff \|\mathbf{v}\| \cos(\theta) = \frac{1}{\|\mathbf{w}\|} = \frac{1}{\sqrt{5}}. \quad (20.1.10)$$

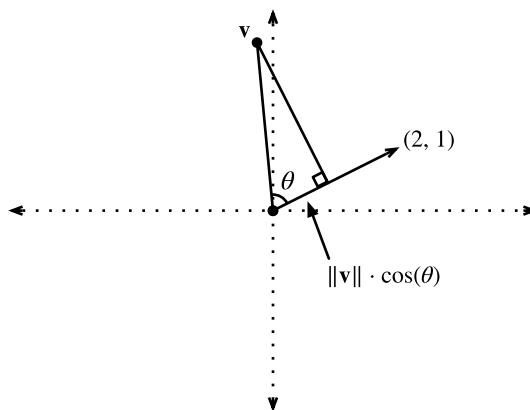


Fig. 20.1.5: Nhắc lại trong lượng giác, chúng ta coi  $\|\mathbf{v}\| \cos(\theta)$  là độ dài hình chiếu của vector  $\mathbf{v}$  lên hướng của vector  $\mathbf{w}$

Nếu xem xét ý nghĩa hình học của biểu thức này, chúng ta thấy rằng nó tương đương với việc độ dài hình chiếu của  $\mathbf{v}$  lên hướng của  $\mathbf{w}$  chính là  $1/\|\mathbf{w}\|$ , như được biểu diễn trong Fig. 20.1.5. Tập hợp các điểm thỏa mãn điều kiện này là một đường thẳng vuông góc với vector  $\mathbf{w}$ . Ta có thể tìm được phương trình của đường thẳng này là  $2x + y = 1$  hoặc  $y = 1 - 2x$ .

Tiếp theo, nếu ta muốn biết tập hợp các điểm thỏa mãn  $\mathbf{w} \cdot \mathbf{v} > 1$  hoặc  $\mathbf{w} \cdot \mathbf{v} < 1$ , ta có thể thấy rằng đây là những trường hợp mà hình chiếu của chúng lên  $\mathbf{w}$  lần lượt dài hơn hoặc ngắn hơn  $1/\|\mathbf{w}\|$ . Vì thế, hai bất phương trình này định nghĩa hai phía của đường thẳng. Bằng cách này, ta có thể cắt mặt phẳng thành hai nửa: một nửa chứa tất cả các điểm có tích vô hướng nhỏ hơn một mức ngưỡng và nửa còn lại chứa những điểm có tích vô hướng lớn hơn mức ngưỡng đó, như trong hình Fig. 20.1.6.

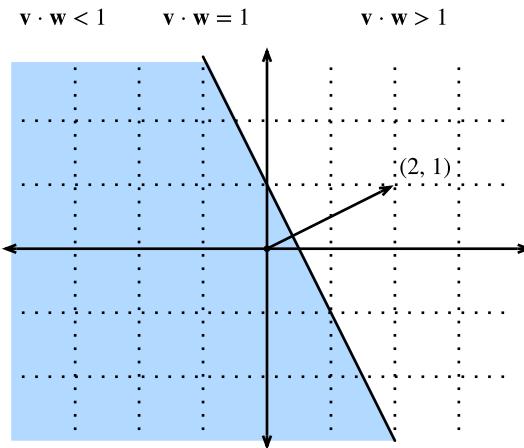


Fig. 20.1.6: Nếu nhìn từ dạng bất phương trình, ta thấy rằng siêu phẳng (trong trường hợp này là một đường thẳng) chia không gian ra thành hai nửa.

Câu chuyện trong không gian đa chiều cũng tương tự. Nếu lấy  $\mathbf{w} = [1, 2, 3]^\top$  và đi tìm các điểm trong không gian ba chiều với  $\mathbf{w} \cdot \mathbf{v} = 1$ , ta có một mặt phẳng vuông góc với vector cho trước  $\mathbf{w}$ . Hai bất phương trình một lần nữa định nghĩa hai phía của mặt phẳng như trong hình Fig. 20.1.7.

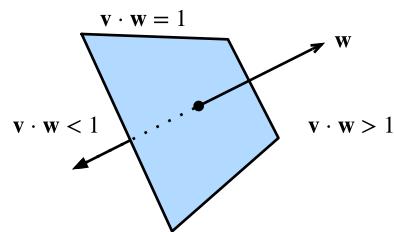


Fig. 20.1.7: Siêu phẳng trong bất kỳ không gian nào chia không gian đó ra thành hai nửa.

Mặc dù không thể minh họa trong không gian nhiều chiều hơn, ta vẫn có thể tổng quát điều này cho không gian mười, một trăm hay một tỷ chiều. Việc này thường xuyên xảy ra khi nghĩ về các mô hình học máy. Chẳng hạn, ta có thể hiểu các mô hình phân loại tuyến tính trong Section 5.4 cũng giống như những phương pháp tìm siêu phẳng để phân chia các lớp mục tiêu khác nhau. Ở trường hợp này, những siêu phẳng như trên thường được gọi là *các mặt phẳng quyết định*. Phần lớn các mô hình phân loại tìm được qua học sâu đều kết thúc với một tầng tuyến tính và sau là một tầng softmax, bởi vậy ta có thể diễn giải ý nghĩa của mạng nơ-ron sâu giống như việc tìm một embedding phi tuyến sao cho các lớp mục tiêu có thể được phân chia bởi các siêu phẳng một

cách gọn gàng.

Xét ví dụ sau. Để ý rằng, ta có thể tạo một mô hình đủ tốt để phân loại những tấm ảnh áo thun và quần với kích thước nhỏ từ tập dữ liệu Fashion MNIST (Xem Section 5.5) bằng cách lấy vector giữa điểm trung bình của mỗi lớp để định nghĩa một mặt phẳng quyết định và chọn thủ công một ngưỡng. Trước tiên, chúng ta nạp dữ liệu và tính hai ảnh trung bình:

```
# Load in the dataset
train = gluon.data.vision.FashionMNIST(train=True)
test = gluon.data.vision.FashionMNIST(train=False)

X_train_0 = np.stack([x[0] for x in train if x[1] == 0]).astype(float)
X_train_1 = np.stack([x[0] for x in train if x[1] == 1]).astype(float)
X_test = np.stack(
    [x[0] for x in test if x[1] == 0 or x[1] == 1]).astype(float)
y_test = np.stack(
    [x[1] for x in test if x[1] == 0 or x[1] == 1]).astype(float)

# Compute averages
ave_0 = np.mean(X_train_0, axis=0)
ave_1 = np.mean(X_train_1, axis=0)
```

Để có cái nhìn rõ hơn, ta có thể xem xét một cách chi tiết các ảnh trung bình này bằng cách in chúng ra màn hình. Quả thật, ảnh đầu tiên trông như một chiếc áo thun bị mờ.

```
# Plot average t-shirt
d2l.set figsize()
d2l.plt.imshow(ave_0.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```

Trong ảnh thứ hai, chúng ta cũng thấy ảnh trung bình chứa một chiếc quần dài bị mờ.

```
# Plot average trousers
d2l.plt.imshow(ave_1.reshape(28, 28).tolist(), cmap='Greys')
d2l.plt.show()
```

Trong một lời giải học máy hoàn chỉnh thì mức ngưỡng cũng sẽ được học từ tập dữ liệu. Trong trường hợp này, ta chỉ đơn thuần chọn thủ công một ngưỡng mang lại kết quả khá tốt trên tập huấn luyện.

```
# Print test set accuracy with eyeballed threshold
w = (ave_1 - ave_0).T
predictions = X_test.reshape(2000, -1).dot(w.flatten()) > -1500000
# Accuracy
np.mean(predictions.astype(y_test.dtype) == y_test, dtype=np.float64)
```

#### 20.1.4 Ý nghĩa Hình học của các Phép biến đổi Tuyến tính

Thông qua Section 4.3 và các phần thảo luận phía trên, ta đã có kiến thức vững chắc về ý nghĩa hình học của vector, độ dài, và góc. Tuy nhiên, có một đối tượng quan trọng chúng ta đã bỏ qua, đó là ý nghĩa hình học của các phép biến đổi tuyến tính thể hiện bởi các ma trận. Để hoàn toàn hiểu cách ma trận được dùng để biến đổi dữ liệu giữa hai không gian nhiều chiều khác nhau cần thực hành thường xuyên và nắm ngoài phạm vi của phần phụ lục này. Tuy nhiên, chúng ta có thể xây dựng các ý niệm trực quan trong không gian hai chiều.

Giả sử ta có một ma trận:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}. \quad (20.1.11)$$

Nếu muốn áp dụng ma trận này lên một vector  $\mathbf{v} = [x, y]^\top$  bất kỳ, ta thực hiện phép nhân và thấy rằng

$$\begin{aligned} \mathbf{A}\mathbf{v} &= \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ &= \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \\ &= x \begin{bmatrix} a \\ c \end{bmatrix} + y \begin{bmatrix} b \\ d \end{bmatrix} \\ &= x \left\{ \mathbf{A} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} + y \left\{ \mathbf{A} \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}. \end{aligned} \quad (20.1.12)$$

Thoạt nhìn đây là một phép tính khá kỳ lạ, nó biến một thứ vốn rõ ràng trở nên khó hiểu. Tuy nhiên, nó cho thấy một ma trận có thể biến đổi bất kỳ vector nào bằng việc biến đổi *hai vector cụ thể*:  $[1, 0]^\top$  và  $[0, 1]^\top$ . Quan sát một chút, chúng ta thực tế đã thu gọn một bài toán vô hạn (tính toán cho bất kỳ vector nào) thành một bài toán hữu hạn (tính toán cho chỉ hai vector). Tập hợp hai vector này là ví dụ của một *cơ sở* (*basis*), và bất kì vector nào trong không gian đều có thể được biểu diễn dưới dạng tổng có trọng số của những vector cơ sở này.

Cùng xét ví dụ với một ma trận cụ thể

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}. \quad (20.1.13)$$

Xét vector  $\mathbf{v} = [2, -1]^\top$ , ta thấy rằng vector này có thể viết dưới dạng  $2 \cdot [1, 0]^\top + -1 \cdot [0, 1]^\top$ . Bởi vậy ta biết ma trận  $A$  sẽ biến đổi nó thành  $2(\mathbf{A}[1, 0]^\top) + -1(\mathbf{A}[0, 1]^\top) = 2[1, -1]^\top - [2, 3]^\top = [0, -5]^\top$ . Xét mạng lưới cấu thành từ tất cả các cặp điểm có tọa độ nguyên, ta có thể thấy rằng phép nhân ma trận có thể làm nghiêng, xoay và co giãn lưới đó, nhưng cấu trúc của lưới phải giữ nguyên như minh họa trong Fig. 20.1.8.

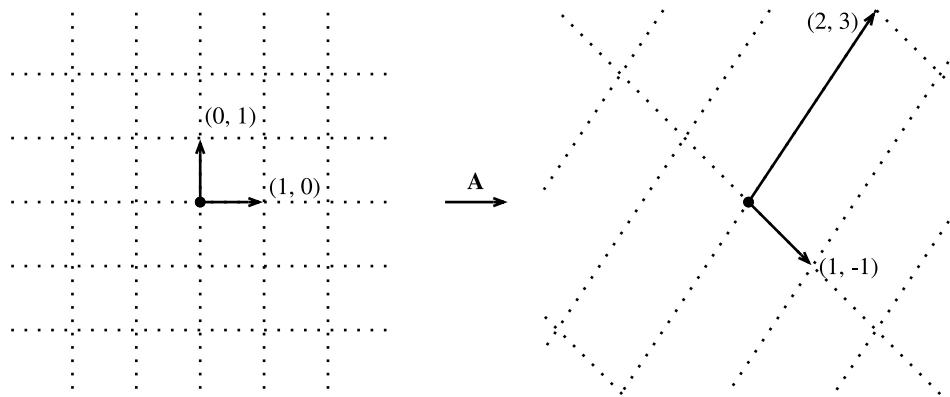


Fig. 20.1.8: Ma trận  $A$  biến đổi các vector cơ sở cho trước. Hãy để ý việc toàn bộ lưới cũng bị biến đổi theo.

Đây là điểm quan trọng nhất về các phép biến đổi tuyến tính thông qua ma trận mà ta cần phải tiếp thu. Một ma trận không thể làm biến dạng các phần không gian khác nhau theo các cách khác nhau. Chúng chỉ có thể làm nghiêng, xoay và co giãn các tọa độ ban đầu.

Một vài phép biến đổi có thể có ảnh hưởng rất lớn. Chẳng hạn ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}, \quad (20.1.14)$$

nén toàn bộ mặt phẳng hai chiều thành một đường thẳng. Việc nhận dạng và làm việc với các phép biến đổi này là chủ đề của phần sau, nhưng nhìn từ khía cạnh hình học, ta có thể thấy rằng nó khác hẳn so với các phép biến đổi ở trên. Ví dụ, kết quả từ ma trận  $A$  có thể bị “biến đổi lại” thành dạng ban đầu. Kết quả từ ma trận  $\mathbf{B}$  thì không thể vì ta không biết vector  $[1, 2]^\top$  được biến đổi từ vector nào –  $[1, 1]^\top$  hay  $[0, -1]^\top$ ?

Dù câu chuyện vừa rồi là về ma trận  $2 \times 2$ , ta hoàn toàn có thể tổng quát hóa những kiến thức vừa học vào các không gian nhiều chiều hơn. Nếu chúng ta lấy các vector cơ sở như  $[1, 0, \dots, 0]$  và xem cách ma trận đó biến đổi các vector này, ta có thể phần nào hình dung được việc phép nhân ma trận làm biến dạng toàn bộ không gian như thế nào, bất kể số chiều của không gian đó.

### 20.1.5 Phụ thuộc Tuyến tính

Quay lại với ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & -1 \\ 4 & -2 \end{bmatrix}. \quad (20.1.15)$$

Ma trận này nén toàn bộ mặt phẳng xuống thành một đường thẳng  $y = 2x$ . Câu hỏi đặt ra là: có cách nào phát hiện ra điều này nếu chỉ nhìn vào ma trận không? Câu trả lời tất nhiên là có. Đặt  $\mathbf{b}_1 = [2, 4]^\top$  và  $\mathbf{b}_2 = [-1, -2]^\top$  là hai cột của  $\mathbf{B}$ . Nhắc lại rằng chúng ta có thể biểu diễn bất cứ vector nào được biến đổi bởi ma trận  $\mathbf{B}$  dưới dạng tổng có trọng số của các cột trong ma trận này:  $a_1\mathbf{b}_1 + a_2\mathbf{b}_2$ . Tổng này được gọi là *tổ hợp tuyến tính* (*linear combination*). Vì  $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ , ta có thể biểu diễn tổ hợp bất kỳ của hai cột này mà chỉ dùng  $\mathbf{b}_2$ :

$$a_1\mathbf{b}_1 + a_2\mathbf{b}_2 = -2a_1\mathbf{b}_2 + a_2\mathbf{b}_2 = (a_2 - 2a_1)\mathbf{b}_2. \quad (20.1.16)$$

Điều này chỉ ra rằng một trong hai cột là dư thừa vì nó không định nghĩa một hướng độc nhất trong không gian. Việc này cũng không quá bất ngờ bởi ma trận này đã biến toàn bộ mặt phẳng xuống thành một đường thẳng. Hơn nữa, điều này có thể được nhận thấy do hai cột trên phụ thuộc tuyến tính  $\mathbf{b}_1 = -2 \cdot \mathbf{b}_2$ . Để thấy sự đối xứng giữa hai vector này, ta sẽ viết dưới dạng

$$\mathbf{b}_1 + 2 \cdot \mathbf{b}_2 = 0. \quad (20.1.17)$$

Nhìn chung, ta nói một tập hợp các vector  $\mathbf{v}_1, \dots, \mathbf{v}_k$  phụ thuộc tuyến tính nếu tồn tại các hệ số  $a_1, \dots, a_k$  không đồng thời bằng không sao cho

$$\sum_{i=1}^k a_i \mathbf{v}_i = 0. \quad (20.1.18)$$

Trong trường hợp này, ta có thể biểu diễn một vector dưới dạng một tổ hợp nào đó của các vector khác, khiến cho nó trở nên dư thừa. Bởi vậy, sự phụ thuộc tuyến tính giữa các cột của một ma trận là một bằng chứng cho thấy ma trận đó đang làm giảm số chiều không gian. Nếu không có sự phụ thuộc tuyến tính, chúng ta nói rằng các vector này *độc lập tuyến tính* (*linearly independent*). Nếu các cột của một ma trận là độc lập tuyến tính, việc nén sẽ không xảy ra và phép toán này có thể nghịch đảo.

### 20.1.6 Hạng

Với một ma trận tổng quát  $n \times m$ , câu hỏi tự nhiên được đặt ra là ma trận đó ánh xạ vào không gian bao nhiêu chiều. Để trả lời cho câu hỏi này, ta dùng khái niệm *hạng* (*rank*). Trong mục trước, ta thấy một hệ phụ thuộc tuyến tính *nén* không gian xuống một không gian khác có số chiều ít hơn. Chúng ta sẽ sử dụng tính chất này để định nghĩa hạng. Cụ thể, hạng của một ma trận  $\mathbf{A}$  là số lượng cột độc lập tuyến tính lớn nhất trong mọi tập con các cột của ma trận đó. Ví dụ, ma trận

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}, \quad (20.1.19)$$

có  $\text{rank}(B) = 1$  vì hai cột của nó phụ thuộc tuyến tính và mỗi cột đơn lẻ không phụ thuộc tuyến tính. Xét một ví dụ phức tạp hơn

$$\mathbf{C} = \begin{bmatrix} 1 & 3 & 0 & -1 & 0 \\ -1 & 0 & 1 & 1 & -1 \\ 0 & 3 & 1 & 0 & -1 \\ 2 & 3 & -1 & -2 & 1 \end{bmatrix}, \quad (20.1.20)$$

Ta có thể chứng minh được  $\mathbf{C}$  có hạng bằng hai, bởi hai cột đầu tiên là độc lập tuyến tính, trong khi tập hợp ba cột bất kỳ trong ma trận đều phụ thuộc tuyến tính.

Quá trình trên rất không hiệu quả, vì đòi hỏi xét mọi tập con các cột của một ma trận cho trước, số tập con này tăng theo hàm mũ khi số cột tăng lên. Sau này chúng ta sẽ thấy một cách hiệu quả hơn để tính hạng của ma trận, hiện tại định nghĩa trên là đủ để hiểu khái niệm và ý nghĩa của hạng.

### 20.1.7 Tính nghịch đảo (khả nghịch)

Như chúng ta đã thấy ở trên, phép nhân một ma trận có các cột phụ thuộc tuyến tính là không thể hoàn tác, tức là không tồn tại thao tác nghịch đảo nào có thể khôi phục lại đầu vào. Tuy nhiên, trong phép biến đổi bằng một ma trận có hạng đầy đủ (ví dụ, với ma trận  $\mathbf{A}$  nào đó kích thước  $n \times n$  có hạng  $n$ ), ta luôn có thể hoàn tác nó. Xét ma trận

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}. \quad (20.1.21)$$

đây là ma trận với các phần tử trên đường chéo có giá trị 1 và các phần tử còn lại có giá trị 0. Ma trận này được gọi là ma trận *đơn vị* (*identity matrix*). Dữ liệu sẽ không bị thay đổi khi nhân với ma trận này. Để có một ma trận hoàn tác những gì ma trận  $\mathbf{A}$  đã làm, ta tìm một ma trận  $\mathbf{A}^{-1}$  sao cho

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}. \quad (20.1.22)$$

Nếu xem đây là một hệ phương trình, ta có  $n \times n$  biến (các giá trị của  $\mathbf{A}^{-1}$ ) và  $n \times n$  phương trình (đẳng thức cần thỏa mãn giữa mỗi giá trị của tích  $\mathbf{A}^{-1}\mathbf{A}$  và giá trị tương ứng của  $\mathbf{I}$ ) nên nhìn chung hệ phương trình có nghiệm. Thực vậy, phần tiếp theo sẽ giới thiệu một đại lượng được gọi là *định thức* (*determinant*) với tính chất: nghiệm tồn tại khi đại lượng này khác 0. Ma trận  $\mathbf{A}^{-1}$  như vậy được gọi là ma trận *nghịch đảo*. Ví dụ, nếu  $\mathbf{A}$  là ma trận  $2 \times 2$

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad (20.1.23)$$

thì nghịch đảo của ma trận này là

$$\frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}. \quad (20.1.24)$$

Việc này có thể kiểm chứng bằng công thức ma trận nghịch đảo trình bày ở trên.

```
M = np.array([[1, 2], [1, 4]])
M_inv = np.array([[2, -1], [-0.5, 0.5]])
M_inv.dot(M)
```

### Vấn đề Tính toán

Mặc dù ma trận nghịch đảo khá hữu dụng trong lý thuyết, chúng ta nên tránh *sử dụng* chúng khi giải quyết các bài toán thực tế. Nhìn chung, có rất nhiều phương pháp tính toán ổn định hơn trong việc giải các phương trình tuyến tính dạng

$$\mathbf{Ax} = \mathbf{b}, \quad (20.1.25)$$

so với việc tính ma trận nghịch đảo và thực hiện phép nhân để có

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}. \quad (20.1.26)$$

Giống như việc thực hiện phép chia một số nhỏ có thể dẫn đến sự mất ổn định tính toán, việc nghịch đảo một ma trận có hạng thấp cũng có ảnh hưởng tương tự.

Thêm vào đó, thông thường  $\mathbf{A}$  là ma trận *thưa* (*sparse*), có nghĩa là nó chỉ chứa một số lượng nhỏ các số khác 0. Nếu thử một vài ví dụ, chúng ta có thể thấy điều này không có nghĩa ma trận nghịch đảo cũng là một ma trận thưa. Kể cả khi ma trận  $\mathbf{A}$  là ma trận 1 triệu nhân 1 triệu với chỉ 5 triệu giá trị khác 0 (có nghĩa là chúng ta chỉ cần lưu trữ 5 triệu giá trị đó), ma trận nghịch đảo thông thường vẫn giữ lại các thành phần không âm và đòi hỏi chúng ta phải lưu trữ  $1M^2$  phần tử—tương đương với 1 nghìn tỉ phần tử!

Mặc dù không đủ thời gian để đi sâu vào các vấn đề tính toán phức tạp thường gặp khi làm việc với đại số tuyến tính, chúng tôi vẫn mong muốn có thể cung cấp một vài lưu ý quan trọng, và quy tắc chung trong thực tiễn là hạn chế việc tính nghịch đảo.

### 20.1.8 Định thức

Góc nhìn hình học của đại số tuyến tính cung cấp một cách hiểu trực quan về một đại lượng cơ bản được gọi là *định thức*. Xét lưới không gian trong phần trước với một vùng in đậm (Fig. 20.1.9).

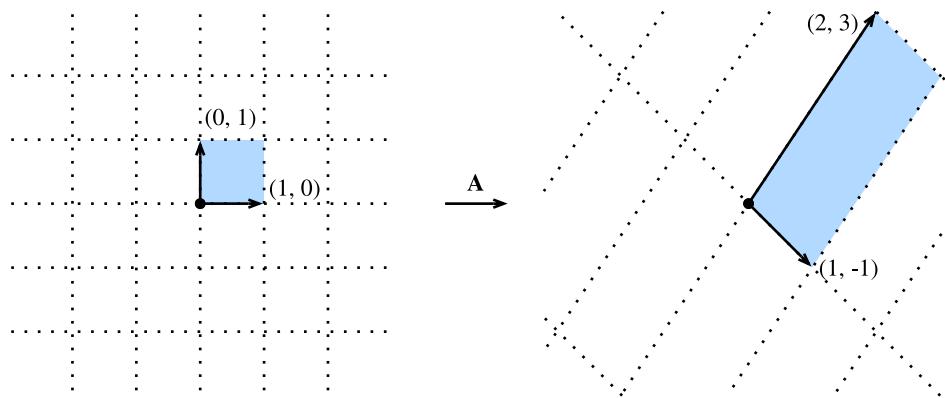


Fig. 20.1.9: Ma trận  $\mathbf{A}$  vẫn làm biến dạng lưới. Lần này, tôi muốn dồn sự chú ý vào điều đã xảy ra với hình vuông được tô màu.

Cùng nhìn vào hình vuông được tô màu, nó có diện tích bằng một với các cạnh được tạo bởi  $(0, 1)$  và  $(1, 0)$ . Sau khi ma trận  $\mathbf{A}$  biến đổi hình vuông này, ta thấy rằng nó trở thành một hình bình hành. Không có lý do nào để hình bình hành này có cùng diện tích với hình vuông ban đầu. Ví dụ, với ma trận

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ -1 & 3 \end{bmatrix}, \quad (20.1.27)$$

bạn có thể tính được diện tích hình bình hành bằng 5 như một bài tập hình học tọa độ đơn giản.

Tổng quát, với:

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad (20.1.28)$$

ta có thể tính ra diện tích của hình bình hành là  $ad - bc$ . Diện tích này được coi là *định thức*.

Cùng kiểm tra nhanh điều này với một đoạn mã ví dụ.

```
import numpy as np
np.linalg.det(np.array([[1, -1], [2, 3]]))
```

Bạn đọc tinh mắt có thể nhận ra biểu thức này có thể bằng không hoặc thậm chí âm. Khi biểu thức này âm, đó là quy ước toán học thường dùng: nếu ma trận đó “lật” một hình, nó sẽ đảo dấu diện tích hình đó. Còn khi định thức bằng không thì sao?

Xét

$$\mathbf{B} = \begin{bmatrix} 2 & 4 \\ -1 & -2 \end{bmatrix}. \quad (20.1.29)$$

Định thức của ma trận này là  $2 \cdot (-2) - 4 \cdot (-1) = 0$ . Điều này là hợp lý bởi ma trận  $\mathbf{B}$  đã nén hình vuông ban đầu xuống thành một đoạn thẳng với diện tích bằng không. Thật vậy, nén không gian xuống ít chiều hơn là cách duy nhất để có diện tích bằng không sau phép biến đổi. Do đó chúng ta suy ra được hệ quả sau: ma trận  $A$  khả nghịch khi và chỉ khi nó có định thức khác không.

Hãy tưởng tượng ta có một hình bất kỳ trên mặt phẳng. Ta có thể chia nhỏ hình này thành một tập hợp các hình vuông nhỏ, như vậy diện tích hình đó sẽ bằng tổng diện tích các hình vuông nhỏ. Bây giờ nếu ta biến đổi hình đó bằng một ma trận, các hình vuông nhỏ sẽ được biến đổi thành các hình bình hành với diện tích bằng với định thức của ma trận. Ta thấy rằng với một hình bất kỳ, định thức của một ma trận là hệ số co dãn diện tích (có dấu) của hình đó gây ra bởi ma trận.

Việc tính định thức cho các ma trận lớn có thể phức tạp hơn, nhưng ý tưởng là như nhau. Định thức giữ nguyên tính chất rằng ma trận  $n \times n$  co giãn các khối thể tích trong không gian  $n$  chiều.

### 20.1.9 Tensor và các Phép toán Đại số Tuyến tính thông dụng

Khái niệm về tensor đã được giới thiệu ở Section 4.3. Trong mục này, chúng ta sẽ đi sâu hơn vào phép co tensor (tương đương với phép nhân ma trận), và xem chúng cung cấp cái nhìn nhất quán về một số phép toán ma trận và vector như thế nào.

Chúng ta đã biết biến đổi dữ liệu bằng cách nhân với ma trận và vector. Để tensor trở nên hữu ích, ta cần một định nghĩa tương tự như thế. Xem lại phép nhân ma trận:

$$\mathbf{C} = \mathbf{AB}, \quad (20.1.30)$$

tương đương với:

$$c_{i,j} = \sum_k a_{i,k} b_{k,j}. \quad (20.1.31)$$

Cách thức biểu diễn này có thể lặp lại với tensor. Với tensor, không có thứ tự tổng quát để chọn tính tổng theo chỉ số nào. Bởi vậy, cần chỉ ra chính xác ta muốn tính tổng trên chỉ số nào. Ví dụ, xét:

$$y_{il} = \sum_{jk} x_{ijkl} a_{jk}. \quad (20.1.32)$$

Phép biến đổi này được gọi là một phép *co tensor* (*tensor contraction*). Nó có thể biểu diễn được các phép biến đổi một cách linh động hơn nhiều so với phép nhân ma trận đơn thuần.

Để đơn giản cho việc ký hiệu, ta có thể để ý rằng tổng chỉ được tính theo những chỉ số xuất hiện nhiều hơn một lần trong biểu thức. Bởi vậy, người ta thường làm việc với *ký hiệu Einstein* với quy ước rằng phép tính tổng sẽ được lấy trên các chỉ số xuất hiện lặp lại. Từ đó, ta có một phép biểu diễn ngắn gọn:

$$y_{il} = x_{ijkl}a_{jk}. \quad (20.1.33)$$

### Một số Ví dụ thông dụng trong Đại số Tuyến tính

Hãy xem ta có thể biểu diễn bao nhiêu khái niệm đại số tuyến tính đã biết với biểu thức tensor thu gọn này:

- $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$
- $\|\mathbf{v}\|_2^2 = \sum_i v_i v_i$
- $(\mathbf{Av})_i = \sum_j a_{ij} v_j$
- $(\mathbf{AB})_{ik} = \sum_j a_{ij} b_{jk}$
- $\text{tr}(\mathbf{A}) = \sum_i a_{ii}$

Với cách này, ta có thể thay thế hàng loạt ký hiệu chi tiết bằng những biểu diễn tensor ngắn.

### Biểu diễn khi Lập trình

Tensor cũng có thể được thao tác linh hoạt dưới dạng mã. Như trong Section 4.3, ta có thể tạo các tensor như sau.

```
# Define tensors
B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
A = np.array([[1, 2], [3, 4]])
v = np.array([1, 2])
# Print out the shapes
A.shape, B.shape, v.shape
```

Phép tính tổng Einstein đã được lập trình sẵn và có thể sử dụng một cách trực tiếp. Các chỉ số xuất hiện trong phép tổng Einstein có thể được truyền vào dưới dạng chuỗi ký tự, theo sau là những tensor cần thao tác. Ví dụ, để thực hiện phép nhân ma trận, ta có thể sử dụng phép tổng Einstein ở trên ( $\mathbf{Av} = a_{ij}v_j$ ) và tách riêng các chỉ số như sau:

```
# Reimplement matrix multiplication
np.einsum("ij, j -> i", A, v), A.dot(v)
```

Đây là một ký hiệu cực kỳ linh hoạt. Giả sử ta muốn tính toán một phép tính thường được ghi một cách truyền thống là

$$c_{kl} = \sum_{ij} \mathbf{b}_{ijk} \mathbf{a}_{il} v_j. \quad (20.1.34)$$

nó có thể được thực hiện thông qua phép tổng Einstein như sau:

```
np.einsum("ijk, il, j -> kl", B, A, v)
```

Cách ký hiệu này vừa dễ đọc và hiệu quả cho chúng ta, tuy nhiên lại khá rườm rà nếu ta cần tạo ra một phép co tensor tự động bằng cách lập trình. Vì lý do này, `einsum` có một cách ký hiệu thay thế bằng cách cung cấp các chỉ số nguyên cho mỗi tensor. Ví dụ, cùng phép co tensor ở trên có thể viết lại như sau:

```
np.einsum(B, [0, 1, 2], A, [0, 3], v, [1], [2, 3])
```

Cả hai cách ký hiệu đều biểu diễn phép co tensor một cách chính xác và hiệu quả.

### 20.1.10 Tóm tắt

- Về phương diện hình học, vector có thể được hiểu như là điểm hoặc hướng trong không gian.
- Tích vô hướng định nghĩa khái niệm góc trong không gian đa chiều bất kỳ.
- Siêu phẳng (*hyperplane*) là sự khái quát hóa của đường thẳng và mặt phẳng trong không gian đa chiều. Chúng có thể được dùng để định nghĩa các mặt phẳng quyết định dùng trong bước cuối cùng của bài toán phân loại.
- Ta có thể hiểu phép nhân ma trận theo cách hình học là việc biến đổi một cách đồng nhất các hệ tọa độ. Cách biểu diễn sự biến đổi vector này tuy có nhiều hạn chế nhưng lại gọn gàng về mặt toán học.
- Phụ thuộc tuyến tính cho biết khi một tập các vector tồn tại trong một không gian ít chiều hơn so với dự kiến (chẳng hạn bạn có 3 vector nhưng chúng chỉ nằm trong không gian 2 chiều). Hạng của ma trận là số lượng cột độc lập tuyến tính lớn nhất trong ma trận đó.
- Khi phép nghịch đảo của một ma trận là xác định, việc nghịch đảo ma trận cho phép chúng ta tìm một ma trận khác giúp hoàn tác lại thao tác trước đó. Việc nghịch đảo ma trận hữu dụng trong lý thuyết, nhưng yêu cầu cẩn trọng khi sử dụng vì tính bất ổn định số học (*numerical instability*) của nó.
- Định thức cho phép đo lường mức độ một ma trận làm co dãn không gian. Một ma trận là khả nghịch khi và chỉ khi định thức của nó khác không.
- Phép co tensor và phép lấy tổng Einstein cho ta cách biểu diễn gọn gàng và súc tích các phép toán thường gặp trong học máy.

### 20.1.11 Bài tập

1. Góc giữa hai vectors dưới đây là bao nhiêu?

$$\vec{v}_1 = \begin{bmatrix} 1 \\ 0 \\ -1 \\ 2 \end{bmatrix}, \quad \vec{v}_2 = \begin{bmatrix} 3 \\ 1 \\ 0 \\ 1 \end{bmatrix} ? \quad (20.1.35)$$

2.  $\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$  và  $\begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$  là nghịch đảo của nhau, đúng hay sai?

3. Giả sử ta vẽ một hình trong mặt phẳng với diện tích  $100\text{m}^2$ . Diện tích hình đó sẽ bằng bao nhiêu sau khi biến đổi nó với ma trận

$$\begin{bmatrix} 2 & 3 \\ 1 & 2 \end{bmatrix}. \quad (20.1.36)$$

4. Trong các nhóm vector sau, nhóm nào là độc lập tuyến tính?

- $\left\{ \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix} \right\}$
- $\left\{ \begin{pmatrix} 3 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right\}$
- $\left\{ \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \right\}$

5. Giả sử ta có ma trận  $A = \begin{bmatrix} c \\ d \end{bmatrix} \cdot [a \ b]$  với các giá trị  $a, b, c, d$  nào đó. Ma trận đó luôn có định thức bằng 0, đúng hay sai?

6. Các vector  $e_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$  và  $e_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$  là trực giao. Ma trận  $A$  cần thỏa mãn điều kiện gì để  $Ae_1$  và  $Ae_2$  trực giao?
7. Viết  $\text{tr}(A^4)$  theo cách biểu diễn Einstein như thế nào với ma trận  $A$ ? tùy ý?

### 20.1.12 Thảo luận

- Tiếng Anh: MXNet<sup>407</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>408</sup>

### Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Hoàng Trọng Tuấn
- Nguyễn Cảnh Thượng
- Nguyễn Xuân Tú
- Phạm Hồng Vinh
- Trần Thị Hồng Hạnh
- Nguyễn Lê Quang Nhật
- Mai Sơn Hải

<sup>407</sup> <https://discuss.d2l.ai/t/410>

<sup>408</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.2 Phân rã trị riêng

Trị riêng là một trong những khái niệm hữu ích nhất trong đại số tuyến tính, tuy nhiên người mới học thường bỏ qua tầm quan trọng của chúng. Dưới đây, chúng tôi sẽ giới thiệu về phân rã trị riêng (*eigendecomposition*) và cố gắng truyền tải tầm quan trọng của chúng.

Giả sử ta có một ma trận  $A$  sau:

$$\mathbf{A} = \begin{bmatrix} 2 & 0 \\ 0 & -1 \end{bmatrix}. \quad (20.2.1)$$

Nếu ta áp dụng  $A$  lên bất kỳ vector  $\mathbf{v} = [x, y]^\top$  nào, ta nhận được vector  $\mathbf{Av} = [2x, -y]^\top$ . Điều này có thể được diễn giải trực quan như sau: kéo giãn vector  $\mathbf{v}$  dài gấp đôi theo phương  $x$ , rồi lấy đối xứng theo phương  $y$ .

Tuy nhiên, sẽ có *một vài* vector với một tính chất không thay đổi. Ví dụ như  $[1, 0]^\top$  được biến đổi thành  $[2, 0]^\top$  và  $[0, 1]^\top$  được biến đổi thành  $[0, -1]^\top$ . Những vector này không thay đổi phuơng, chỉ bị kéo giãn với hệ số 2 và -1. Ta gọi những vector ấy là *vector riêng* và các hệ số mà chúng giãn ra là *trị riêng*.

Tổng quát, nếu ta tìm được một số  $\lambda$  và một vector  $\mathbf{v}$  mà

$$\mathbf{Av} = \lambda\mathbf{v}. \quad (20.2.2)$$

Ta nói rằng  $\mathbf{v}$  là một vector riêng và  $\lambda$  là một trị riêng của  $A$ .

### 20.2.1 Tìm trị riêng

Hãy cùng tìm hiểu cách tìm trị riêng. Bằng cách trừ đi  $\lambda\mathbf{I}$  ở cả hai vế của đẳng thức trên, rồi sau đó nhóm thừa số chung là vector, ta có:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0. \quad (20.2.3)$$

Để (20.2.3) xảy ra,  $(\mathbf{A} - \lambda\mathbf{I})$  phải néo một số chiều xuống không, vì thế nó không thể nghịch đảo được nên có định thức bằng không. Do đó, ta có thể tìm các *trị riêng* bằng cách tìm giá trị  $\lambda$  sao cho  $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ . Một khi tìm được các trị riêng, ta có thể giải phuơng trình  $\mathbf{Av} = \lambda\mathbf{v}$  để tìm (các) *vector riêng* tương ứng.

#### Ví dụ

Hãy xét một ma trận thách thức hơn

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix}. \quad (20.2.4)$$

Nếu để ý  $\det(\mathbf{A} - \lambda\mathbf{I}) = 0$ , ta thấy rằng phương trình này tương đương với phương trình đa thức  $0 = (2 - \lambda)(3 - \lambda) - 2 = (4 - \lambda)(1 - \lambda)$ . Như vậy, hai trị riêng tìm được là 4 và 1. Để tìm các vector tương ứng, ta cần giải hệ

$$\begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} \text{ và } \begin{bmatrix} 2 & 1 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 4x \\ 4y \end{bmatrix}. \quad (20.2.5)$$

Ta thu được nghiệm tương ứng là hai vector  $[1, -1]^\top$  và  $[1, 2]^\top$ .

Ta có thể kiểm tra lại bằng đoạn mã với hàm `numpy.linalg.eig` xây dựng sẵn.

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
import numpy as np

np.linalg.eig(np.array([[2, 1], [2, 3]]))
```

Lưu ý rằng `numpy` chuẩn hóa các vector riêng để có độ dài bằng 1, trong khi các vector ta tìm được bằng cách giải phương trình có độ dài tùy ý.Thêm vào đó, việc chọn dấu cũng là tùy ý. Tuy nhiên, các vector được tính ra bởi thư viện sẽ song song với các vector có được bằng cách giải thủ công với cùng trị riêng.

## 20.2.2 Phân rã Ma trận

Hãy tiếp tục với ví dụ trước đó. Gọi

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ -1 & 2 \end{bmatrix}, \quad (20.2.6)$$

là ma trận có các cột là vector riêng của ma trận  $\mathbf{A}$ . Gọi

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}, \quad (20.2.7)$$

là ma trận với các trị riêng tương ứng nằm trên đường chéo. Từ định nghĩa của trị riêng và vector riêng, ta có

$$\mathbf{AW} = \mathbf{W}\Sigma. \quad (20.2.8)$$

Ma trận  $W$  là khả nghịch, nên ta có thể nhân hai vế với  $W^{-1}$  về phía phải, để có

$$\mathbf{A} = \mathbf{W}\Sigma\mathbf{W}^{-1}. \quad (20.2.9)$$

Trong phần tiếp theo ta sẽ thấy một số hệ quả thú vị từ điều này, nhưng hiện giờ bạn đọc chỉ cần biết rằng tồn tại phân rã như vậy nếu ta có thể tìm tất cả các vector riêng độc lập tuyến tính (để ma trận  $W$  khả nghịch).

### 20.2.3 Các phép toán dùng Phân rã Trị riêng

Một điều thú vị về phân rã trị riêng (20.2.9) là ta có thể viết nhiều phép toán thường gấp một cách gọn gàng khi sử dụng phân rã trị riêng. Ví dụ đầu tiên, xét:

$$\mathbf{A}^n = \overbrace{\mathbf{A} \cdots \mathbf{A}}^{n \text{ times}} = \overbrace{(\mathbf{W}\Sigma\mathbf{W}^{-1}) \cdots (\mathbf{W}\Sigma\mathbf{W}^{-1})}^{n \text{ times}} = \mathbf{W} \overbrace{\Sigma \cdots \Sigma}^{n \text{ times}} \mathbf{W}^{-1} = \mathbf{W}\Sigma^n\mathbf{W}^{-1}. \quad (20.2.10)$$

Điều này cho thấy khi lũy thừa ma trận với bất kỳ số mũ dương nào, ta chỉ cần lũy thừa các trị riêng lên cùng số mũ nếu sử dụng phân rã trị riêng. Tương tự, cũng có thể áp dụng cho các số mũ âm, khi nghịch đảo ma trận ta có

$$\mathbf{A}^{-1} = \mathbf{W}\Sigma^{-1}\mathbf{W}^{-1}, \quad (20.2.11)$$

hay nói cách khác, chỉ cần nghịch đảo từng trị riêng, với điều kiện các trị riêng khác không. Do đó sự khả nghịch tương đương với việc không có trị riêng nào bằng không.

Thật vậy, có thể chứng minh rằng nếu  $\lambda_1, \dots, \lambda_n$  là các trị riêng của một ma trận, định thức của ma trận đó là tích của tất cả các trị riêng:

$$\det(\mathbf{A}) = \lambda_1 \cdots \lambda_n, \quad (20.2.12)$$

Điều này hợp lý về trực giác vì dù ma trận  $\mathbf{W}$  có kéo giãn như thế nào thì  $\mathbf{W}^{-1}$  cũng sẽ hoàn tác hành động đó, vì thế cuối cùng phép kéo giãn duy nhất được áp dụng là nhân với ma trận đường chéo  $\Sigma$ . Phép nhân này sẽ kéo giãn thể tích không gian với hệ số bằng tích của các phần tử trên đường chéo.

Cuối cùng, ta hãy nhớ lại rằng hạng của ma trận là số lượng tối đa các vector cột độc lập tuyến tính trong một ma trận. Bằng cách nghiên cứu kỹ phân rã trị riêng, ta có thể thấy rằng hạng của  $\mathbf{A}$  bằng số lượng các trị riêng khác không của  $\mathbf{A}$ .

Trước khi tiếp tục, hy vọng bạn đọc đã hiểu được ý tưởng: phân rã trị riêng có thể đơn giản hóa nhiều phép tính đại số tuyến tính và là một phép toán cơ bản phía sau nhiều thuật toán số và phân tích trong đại số tuyến tính.

### 20.2.4 Phân rã trị riêng của Ma trận Đối xứng

Không phải lúc nào ta cũng có thể tìm đủ các vector riêng độc lập tuyến tính để thuật toán phía trên hoạt động. Ví dụ ma trận sau

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad (20.2.13)$$

chỉ có duy nhất một vector riêng là  $(1, 0)^\top$ . Để xử lý những ma trận như vậy, ta cần những kỹ thuật cao cấp hơn (ví dụ như dạng chuẩn Jordan - *Jordan Normal Form*, hay phân rã đơn trị - *Singular Value Decomposition*). Ta thường phải giới hạn mức độ và chỉ tập trung đến những ma trận mà ta có thể đảm bảo rằng có tồn tại một tập đầy đủ vector riêng.

Họ vector thường gấp nhất là *ma trận đối xứng*, là những ma trận mà  $\mathbf{A} = \mathbf{A}^\top$ . Trong trường hợp này, ta có thể lấy  $\mathbf{W}$  là *ma trận trực giao* - ma trận có các cột là các vector có độ dài bằng một và vuông góc với nhau, đồng thời  $\mathbf{W}^\top = \mathbf{W}^{-1}$  - và tất cả các trị riêng là số thực. Trong trường hợp đặc biệt này, ta có thể viết (20.2.9) như sau

$$\mathbf{A} = \mathbf{W}\Sigma\mathbf{W}^\top. \quad (20.2.14)$$

### 20.2.5 Định lý Vòng tròn Gershgorin

Các trị riêng thường rất khó để suy luận bằng trực giác. Nếu tồn tại một ma trận bất kỳ, ta khó có thể nói được gì nhiều về các trị riêng nếu không tính toán chúng ra. Tuy nhiên, tồn tại một định lý giúp dễ dàng xấp xỉ tốt trị riêng nếu các giá trị lớn nhất của ma trận nằm trên đường chéo.

Cho  $\mathbf{A} = (a_{ij})$  là ma trận vuông bất kỳ với kích thước  $n \times n$ . Đặt  $r_i = \sum_{j \neq i} |a_{ij}|$ . Cho  $\mathcal{D}_i$  biểu diễn hình tròn trong mặt phẳng phức với tâm  $a_{ii}$ , bán kính  $r_i$ . Khi đó, mỗi trị riêng của  $\mathbf{A}$  được chứa trong một  $\mathcal{D}_i$ .

Điều này có thể hơi khó hiểu, nên hãy quan sát ví dụ sau. Xét ma trận:

$$\mathbf{A} = \begin{bmatrix} 1.0 & 0.1 & 0.1 & 0.1 \\ 0.1 & 3.0 & 0.2 & 0.3 \\ 0.1 & 0.2 & 5.0 & 0.5 \\ 0.1 & 0.3 & 0.5 & 9.0 \end{bmatrix}. \quad (20.2.15)$$

Ta có  $r_1 = 0.3$ ,  $r_2 = 0.6$ ,  $r_3 = 0.8$  và  $r_4 = 0.9$ . Ma trận này là đối xứng nên tất cả các trị riêng đều là số thực. Điều này có nghĩa tất cả các trị riêng sẽ nằm trong một trong các khoảng sau

$$[a_{11} - r_1, a_{11} + r_1] = [0.7, 1.3], \quad (20.2.16)$$

$$[a_{22} - r_2, a_{22} + r_2] = [2.4, 3.6], \quad (20.2.17)$$

$$[a_{33} - r_3, a_{33} + r_3] = [4.2, 5.8], \quad (20.2.18)$$

$$[a_{44} - r_4, a_{44} + r_4] = [8.1, 9.9]. \quad (20.2.19)$$

Thực hiện việc tính toán số ta có các trị riêng xấp xỉ là 0.99, 2.97, 4.95, 9.08, đều nằm hoàn toàn trong các khoảng trên.

```
A = np.array([[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, _ = np.linalg.eig(A)
v
```

Bằng cách này, các trị riêng có thể được xấp xỉ khá chính xác trong trường hợp các phần tử trên đường chéo lớn hơn hẳn so với các phần tử còn lại.

Điều này tuy nhỏ nhưng với một chủ đề phức tạp và tinh vi như phân rã trị riêng, thật tốt nếu có thể hiểu bất kỳ điều gì theo cách trực quan.

### 20.2.6 Một Ứng dụng hữu ích: Mức tăng trưởng của các Ánh xạ Lặp lại

Giờ ta đã hiểu bản chất của vector riêng, hãy xem có thể sử dụng chúng như thế nào để hiểu sâu hơn một vấn đề quan trọng trong mạng nơ-ron: khởi tạo trọng số thích hợp.

## Vector riêng biểu thị Hành vi Dài hạn

Tìm hiểu đầy đủ về cách khởi tạo mang nơ-ron dưới góc nhìn toán học nằm ngoài phạm vi phần này, tuy vậy ta có thể phân tích một ví dụ đơn giản dưới đây để xem các trị riêng giúp ta hiểu cách các mô hình hoạt động như thế nào. Như đã biết, mạng nơ-ron hoạt động bằng cách xen kẽ các phép biến đổi tuyến tính và phi tuyến. Để đơn giản, ở đây ta giả sử không có biến đổi phi tuyến và phép biến đổi chỉ là việc liên tục áp dụng ma trận  $A$ , do đó đầu ra của mô hình là

$$\mathbf{v}_{out} = \mathbf{A} \cdot \mathbf{A} \cdots \mathbf{A} \mathbf{v}_{in} = \mathbf{A}^N \mathbf{v}_{in}. \quad (20.2.20)$$

Khi mô hình trên được khởi tạo,  $A$  nhận các giá trị ngẫu nhiên theo phân phối Gauss. Lấy ví dụ cụ thể, ta bắt đầu bằng một ma trận kích thước  $5 \times 5$  với giá trị trung bình bằng 0, phương sai bằng 1.

```
np.random.seed(8675309)

k = 5
A = np.random.randn(k, k)
A
```

## Hành vi trên Dữ liệu Ngẫu nhiên

Trong mô hình đơn giản, ta giả sử rằng vector dữ liệu ta đưa vào  $\mathbf{v}_{in}$  là một vector Gauss ngẫu nhiên năm chiều. Hãy thử nghĩ xem ta sẽ muốn điều gì xảy ra. Trong ngữ cảnh này, hãy liên tưởng tới một bài toán học máy nói chung, trong đó ta đang cố biến dữ liệu đầu vào, như một ảnh, thành một dự đoán, như xác suất ảnh đó là bức ảnh một con mèo. Nếu việc áp dụng liên tục  $A$  khiến một vector ngẫu nhiên bị kéo giãn lên quá dài thì chỉ với một thay đổi nhỏ trên đầu vào cũng có thể khuếch đại thành một thay đổi lớn trên đầu ra – các sự biến đổi nhỏ trên ảnh đầu vào cũng có thể dẫn tới các dự đoán khác hẳn nhau. Việc này dường như không hợp lý chút nào!

Trái lại, nếu  $A$  khiến các vector ngẫu nhiên co ngắn lại, thì sau khi đi qua nhiều tầng, vector này về cơ bản sẽ co đến mức chẳng còn lại gì, và đầu ra sẽ không còn phụ thuộc vào đầu vào. Rõ ràng việc này cũng không hề hợp lý!

Ta cần tìm ra ranh giới giữa tăng trưởng và suy giảm để đảm bảo rằng thay đổi ở đầu ra phụ thuộc vào đầu vào, nhưng cũng không quá phụ thuộc!

Hãy xem chuyện gì sẽ xảy ra nếu ta liên tục nhân ma trận  $A$  với một vector đầu vào ngẫu nhiên, và theo dõi giá trị chuẩn (*norm*).

```
# Calculate the sequence of norms after repeatedly applying 'A'
v_in = np.random.randn(k, 1)

norm_list = [np.linalg.norm(v_in)]
for i in range(1, 100):
    v_in = A.dot(v_in)
    norm_list.append(np.linalg.norm(v_in))

d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```

Giá trị chuẩn tăng một cách không thể kiểm soát được! Quả thật, nếu ta lấy ra danh sách các tỉ số, ta sẽ thấy một khuôn mẫu.

```

# 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(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')

```

Nếu ta quan sát phần cuối của phép tính trên, ta có thể thấy rằng vector ngẫu nhiên bị kéo giãn với hệ số là  $1.974459321485[\dots]$ , với phần số thập phân ở cuối có thay đổi một chút, nhưng hệ số kéo giãn thì ổn định.

## Liên hệ Ngược lại tới Vector riêng

Ta đã thấy rằng vector riêng và trị riêng tương ứng với mức độ co giãn của thứ gì đó, nhưng chỉ đối với các vector cụ thể và các phép co giãn cụ thể. Hãy cùng xét xem chúng là gì đối với  $\mathbf{A}$ . Nói trước một chút: hóa ra là để có thể quan sát được mọi giá trị, ta cần xét tới số phức. Bạn có thể coi số phức như phép co giãn và phép quay. Bằng cách tính chuẩn của số phức (căn bậc hai của tổng bình phương phần thực và phần ảo), ta có thể đo hệ số co giãn. Hãy sắp xếp chúng theo thứ tự.

```

# Compute the eigenvalues
eigs = np.linalg.eigvals(A).tolist()
norm_eigs = [np.absolute(x) for x in eigs]
norm_eigs.sort()
print(f'norms of eigenvalues: {norm_eigs}')

```

## Nhận xét

Ta quan sát thấy một chút bất thường ở đây: hệ số mà ta đã xác định cho quá trình giãn về dài hạn khi áp dụng ma trận  $\mathbf{A}$  lên một vector ngẫu nhiên lại *chính* là trị riêng lớn nhất của  $\mathbf{A}$  (chính xác đến 13 số thập phân). Điều này rõ ràng không phải một sự trùng hợp.

Tuy nhiên, nếu ta thực sự suy ngẫm chuyện gì đang xảy ra trên phương diện hình học, điều này bắt đầu trở nên hợp lý. Xét một vector ngẫu nhiên. Vector ngẫu nhiên này trỏ tới mỗi hướng một chút, nên cụ thể, nó chút ít cũng trỏ tới cùng hướng với vector riêng của  $\mathbf{A}$  tương ứng với trị riêng lớn nhất. Trị riêng và vector riêng này quan trọng đến mức chúng được gọi là *trị riêng chính (principle eigenvalue)* và *vector riêng chính (principle eigenvector)*. Sau khi áp dụng  $\mathbf{A}$ , vector ngẫu nhiên trên bị giãn ra theo mọi hướng khả dĩ, do nó liên kết với mọi vector riêng khả dĩ, nhưng nó bị giãn nhiều nhất theo hướng liên kết với vector riêng chính. Điều này có nghĩa là sau khi áp dụng  $A$ , vector ngẫu nhiên trên dài ra, và ngày càng cùng hướng với vector riêng chính. Sau khi áp dụng ma trận nhiều lần, vector ngẫu nhiên ngày càng gần vector riêng chính cho tới khi vector này gần như trở thành vector riêng chính. Đây chính là cơ sở cho thuật toán *lặp lũy thừa - (power iteration)* để tìm trị riêng và vector riêng lớn nhất của một ma trận. Để biết chi tiết hơn, bạn đọc có thể tham khảo tại ([VanLoan & Golub, 1983](#)).

## Khắc phục bằng Chuẩn hóa

Từ phần thảo luận trên, lúc này ta kết luận rằng ta không hề muốn một vector ngẫu nhiên bị giãn hoặc co lại, mà ta muốn vector ngẫu nhiên giữ nguyên kích thước trong suốt toàn bộ quá trình tính toán. Để làm được điều đó, ta cần tái tỷ lệ ma trận bằng cách chia cho trị riêng chính, tức sao cho trị riêng lớn nhất giờ có giá trị 1. Hãy xem chuyện gì sẽ xảy ra trong trường hợp này.

```
# Rescale the matrix `A'  
A /= norm_eigs[-1]  
  
# Do the same experiment again  
v_in = np.random.randn(k, 1)  
  
norm_list = [np.linalg.norm(v_in)]  
for i in range(1, 100):  
    v_in = A.dot(v_in)  
    norm_list.append(np.linalg.norm(v_in))  
  
d2l.plot(np.arange(0, 100), norm_list, 'Iteration', 'Value')
```

Ta cũng có thể vẽ đồ thị tỷ lệ các chuẩn liên tiếp như trước và quan sát được rằng nó đã 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(np.arange(1, 100), norm_ratio_list, 'Iteration', 'Ratio')
```

### 20.2.7 Kết luận

Giờ ta có thể thấy được điều mà ta mong muốn! Sau khi chuẩn hóa ma trận bằng trị riêng chính, ta thấy rằng dữ liệu ngẫu nhiên không còn bùng nổ như trước nữa, thay vào đó nó cân bằng quanh một giá trị nhất định. Sẽ thật tuyệt nếu ta có thể thực hiện việc này bằng các định đề cơ bản, và hóa ra là nếu ta tìm hiểu sâu về mặt toán học của nó, ta có thể thấy rằng trị riêng lớn nhất của một ma trận ngẫu nhiên lớn với các phần tử tuân theo phân phối Gauss một cách độc lập với kỳ vọng bằng 0, phương sai bằng 1, về trung bình sẽ xấp xỉ bằng  $\sqrt{n}$ , hay trong trường hợp của ta là  $\sqrt{5} \approx 2.2$ , tuân theo một định luật thú vị là *luật vòng tròn* (*circular law*) (Ginibre, 1965). Mỗi quan hệ giữa các trị riêng (và một đại lượng liên quan được gọi là trị đơn (*singular value*)) của ma trận ngẫu nhiên đã được chứng minh là có liên hệ sâu sắc tới việc khởi tạo mạng nơ-ron một cách thích hợp như đã thảo luận trong (Pennington et al., 2017) và các nghiên cứu liên quan sau đó.

### 20.2.8 Tóm tắt

- Vector riêng là các vector bị giãn bởi một ma trận mà không thay đổi hướng.
- Trị riêng là mức độ mà các vector riêng đó bị giãn bởi việc áp dụng ma trận.
- Phân rã trị riêng của ma trận cho phép nhiều phép toán trên ma trận có thể được rút gọn thành các phép toán trên trị riêng.
- Định lý Đường tròn Gershgorin (*Gershgorin Circle Theorem*) có thể cung cấp giá trị xấp xỉ cho các trị riêng của một ma trận.
- Hành vi của phép lặp lũy thừa cho ma trận chủ yếu phụ thuộc vào độ lớn của trị riêng lớn nhất. Điều này có rất nhiều ứng dụng trong lý thuyết khởi tạo mạng nơ-ron.

### 20.2.9 Bài tập

1. Tìm các trị riêng và vector riêng của

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \quad (20.2.21)$$

2. Tìm các trị riêng và vector riêng của ma trận sau đây, và cho biết có điều gì lạ ở ví dụ này so với ví dụ trước?

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 0 & 2 \end{bmatrix} \quad (20.2.22)$$

3. Không tính các trị riêng, trị riêng nhỏ nhất của ma trận sau có nhỏ hơn 0.5 được không? *Ghi chú:* bài tập này có thể nhầm được trong đầu.

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.1 & 0.3 & 1.0 \\ 0.1 & 1.0 & 0.1 & 0.2 \\ 0.3 & 0.1 & 5.0 & 0.0 \\ 1.0 & 0.2 & 0.0 & 1.8 \end{bmatrix} \quad (20.2.23)$$

### 20.2.10 Thảo luận

- Tiếng Anh: MXNet<sup>409</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>410</sup>

### 20.2.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Văn Cường
- Đỗ Trường Giang

<sup>409</sup> <https://discuss.d2l.ai/t/411>

<sup>410</sup> <https://forum.machinelearningcoban.com/c/d2l>

- Phạm Minh Đức
- Lê Khắc Hồng Phúc

## 20.3 Giải tích một biến

Trong Section 4.4, chúng ta đã thấy những thành phần cơ bản của giải tích vi phân. Trong mục này chúng ta sẽ đi sâu vào kiến thức nền tảng của giải tích và cách áp dụng chúng trong ngữ cảnh học máy.

### 20.3.1 Giải tích Vi phân

Giải tích vi phân là nhánh toán học nghiên cứu về hành vi của các hàm số dưới các biến đổi nhỏ. Để thấy được tại sao đây lại là phần cốt lõi của học sâu, hãy cùng xem xét một ví dụ dưới đây.

Giả sử chúng ta có một mạng nơ-ron sâu với các trọng số được biểu diễn bằng một vector duy nhất  $\mathbf{w} = (w_1, \dots, w_n)$ . Cho trước một tập huấn luyện, chúng ta sẽ tập trung vào giá trị mất mát  $\mathcal{L}(\mathbf{w})$  của mạng nơ-ron trên tập huấn luyện đó.

Đây là một hàm số cực kì phức tạp, biểu diễn chất lượng của tất cả các mô hình khả dĩ của một cấu trúc mạng cho trước trên tập dữ liệu này, nên gần như không thể chỉ ra được ngay một tập các trọng số  $\mathbf{w}$  để cực tiểu hóa mất mát. Do vậy trên thực tế, chúng ta thường bắt đầu bằng việc khởi tạo *ngẫu nhiên* các trọng số, và tiến từng bước nhỏ theo hướng mà sẽ giảm giá trị mất mát nhanh nhất có thể.

Vấn đề bây giờ thoạt nhìn cũng không dễ hơn bao nhiêu: làm thế nào để tìm được hướng đi sẽ giảm giá trị hàm mất mát nhanh nhất có thể? Để trả lời câu hỏi này, trước hết ta hãy xét trường hợp chỉ có một trọng số:  $L(\mathbf{w}) = L(x)$  với một số thực  $x$  duy nhất.

Hãy cùng tìm hiểu xem chuyện gì sẽ xảy ra khi ta lấy giá trị  $x$  và thay đổi nó với một lượng rất nhỏ thành  $x + \epsilon$ . Nếu bạn muốn một con số rõ ràng, hãy nghĩ về một số như  $\epsilon = 0.0000001$ . Để minh họa chuyện gì sẽ diễn ra, hãy vẽ ví dụ đồ thị của hàm số  $f(x) = \sin(x^x)$ , trên khoảng  $[0, 3]$ .

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()
# Plot a function in a normal range
x_big = np.arange(0.01, 3.01, 0.01)
ys = np.sin(x_big**x_big)
d2l.plot(x_big, ys, 'x', 'f(x)')
```

Trong một khoảng lớn thế này, cách hàm số biến đổi rất khó nắm bắt. Tuy nhiên, nếu ta thu nhỏ khoảng xuống ví dụ như thành  $[1.75, 2.25]$ , ta thấy đồ thị trở nên đơn giản hơn rất nhiều.

```
# Plot a the same function in a tiny range
x_med = np.arange(1.75, 2.25, 0.001)
ys = np.sin(x_med**x_med)
d2l.plot(x_med, ys, 'x', 'f(x)')
```

Đỉnh điểm, nếu ta phóng gần vào một đoạn rất nhỏ, cách hàm số biến đổi trở nên đơn giản hơn rất nhiều: chỉ là một đường thẳng.

```
# Plot a the same function in a tiny range
x_small = np.arange(2.0, 2.01, 0.0001)
ys = np.sin(x_small**x_small)
d2l.plot(x_small, ys, 'x', 'f(x)')
```

Đây là một trong những quan sát cốt lõi nhất trong giải tích: hành vi của các hàm số phổ biến có thể được mô hình hóa bằng một đường thẳng trên một khoảng đủ nhỏ. Điều này nghĩa là với hầu hết các hàm số, chúng ta có thể trông đợi rằng khi dịch chuyển  $x$  một khoảng nhỏ,  $f(x)$  cũng sẽ dịch chuyển một khoảng nhỏ. Câu hỏi duy nhất mà chúng ta cần trả lời là “Sự thay đổi của giá trị đầu ra lớn gấp bao nhiêu lần so với sự thay đổi của giá trị đầu vào? Bằng một nửa? Hay sẽ lớn gấp đôi?”

Ta cũng có thể xét nó như tỷ lệ giữa sự thay đổi của đầu ra so với sự thay đổi nhỏ trong đầu vào của một hàm số. Chúng ta có thể biểu diễn nó dưới dạng toán học là:

$$\frac{L(x + \epsilon) - L(x)}{(x + \epsilon) - x} = \frac{L(x + \epsilon) - L(x)}{\epsilon}. \quad (20.3.1)$$

Những kiến thức trên đã đủ để chúng ta bắt đầu thực hành lập trình. Ví dụ, giả sử  $L(x) = x^2 + 1701(x - 4)^3$ , ta có thể biết được độ lớn của giá trị này 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}' )
```

Nếu để ý kĩ, chúng ta sẽ nhận ra rằng kết quả của con số này xấp xỉ 8. Trong trường hợp ta giảm  $\epsilon$  thì giá trị đầu ra ngày càng tiến gần đến 8. Vì vậy chúng ta có thể kết luận một cách chính xác, rằng mức độ thay đổi của đầu ra khi đầu vào thay đổi là 8 tại điểm  $x = 4$ . Có thể viết dưới dạng toán học như sau:

$$\lim_{\epsilon \rightarrow 0} \frac{L(4 + \epsilon) - L(4)}{\epsilon} = 8. \quad (20.3.2)$$

Một chút bàn luận ngoài lề về lịch sử: trong những thập kỷ đầu tiên của các nghiên cứu mạng nơ-ron, các nhà khoa học đã sử dụng thuật toán này (*sai phân hữu hạn - finite differences*) để đánh giá một hàm mất mát dưới các nhiễu loạn nhỏ: chỉ cần thay đổi trọng số và xem cách thức mà hàm mất mát thay đổi. Đây là một cách tính toán không hiệu quả, đòi hỏi đến hai lần tính hàm mất mát để thấy được sự tác động của một thay đổi lên hàm mất mát đó. Thậm chí nếu chúng ta sử dụng phương pháp này với vài nghìn tham số nhỏ, nó cũng sẽ đòi hỏi phải chạy mạng nơ-ron hàng nghìn lần trên toàn bộ dữ liệu. Phải đến năm 1986 thì vấn đề này mới được giải quyết khi *thuật toán lan truyền ngược (backpropagation algorithm)* được giới thiệu ở (Rumelhart et al., 1988) đã đem đến một giải pháp để tính toán sức ảnh hưởng của những thay đổi bất kỳ từ các trọng số lên hàm mất mát với thời gian tính toán chỉ bằng thời gian mô hình đưa ra dự đoán trên tập dữ liệu.

Quay lại với ví dụ của chúng ta, giá trị 8 này biến thiên với các trị khác nhau của  $x$ , vậy nên sẽ là hợp lý nếu chúng ta định nghĩa nó như là một hàm của  $x$ . Một cách chính thống hơn, độ biến thiên của giá trị này được gọi là *đạo hàm* và được viết là:

$$\frac{df}{dx}(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon}. \quad (20.3.3)$$

Các văn bản khác nhau sẽ sử dụng các ký hiệu khác nhau cho đạo hàm. Chẳng hạn, tất cả các ký hiệu dưới đây đều diễn giải cùng một ý nghĩa:

$$\frac{df}{dx} = \frac{d}{dx}f = f' = \nabla_x f = D_x f = f_x. \quad (20.3.4)$$

Phần lớn các tác giả sẽ chọn một ký hiệu duy nhất để sử dụng xuyên suốt, tuy nhiên không phải lúc nào điều này cũng được đảm bảo. Tốt hơn hết là chúng ta nên làm quen với tất cả các ký hiệu này. Ký hiệu  $\frac{df}{dx}$  sẽ được sử dụng trong toàn bộ cuốn sách này, trừ trường hợp chúng ta cần lấy đạo hàm của một biểu thức phức tạp, khi đó chúng ta sẽ sử dụng  $\frac{d}{dx}f$  để biểu diễn những biểu thức như

$$\frac{d}{dx} \left[ x^4 + \cos \left( \frac{x^2 + 1}{2x - 1} \right) \right]. \quad (20.3.5)$$

Đôi khi, việc sử dụng định nghĩa của đạo hàm (20.3.3) để thấy một cách trực quan cách một hàm thay đổi khi  $x$  thay đổi một khoảng nhỏ là rất hữu ích:

$$\begin{aligned} \frac{df}{dx}(x) &= \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon} \implies \frac{df}{dx}(x) \approx \frac{f(x + \epsilon) - f(x)}{\epsilon} \\ &\implies \epsilon \frac{df}{dx}(x) \approx f(x + \epsilon) - f(x) \\ &\implies f(x + \epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x). \end{aligned} \quad (20.3.6)$$

Cần phải nói rõ hơn về phương trình cuối cùng. Nó cho chúng ta biết rằng nếu ta chọn một hàm số bất kỳ và thay đổi đầu vào một lượng nhỏ, sự thay đổi của đầu ra sẽ bằng với lượng nhỏ đó nhân với đạo hàm.

Bằng cách này, chúng ta có thể hiểu đạo hàm là hệ số tỷ lệ cho biết mức độ biến thiên của đầu ra khi đầu vào thay đổi.

### 20.3.2 Quy tắc Giải tích

Bây giờ chúng ta sẽ học cách để tính đạo hàm của một hàm cụ thể. Dạy giải tích một cách chính quy sẽ phải chứng minh lại tất cả mọi thứ từ những định đề căn bản nhất. Tuy nhiên chúng tôi sẽ không làm như vậy mà sẽ cung cấp các quy tắc tính đạo hàm phổ biến thường gặp.

#### Các Đạo hàm phổ biến

Như ở Section 4.4, khi tính đạo hàm ta có thể sử dụng một chuỗi các quy tắc để chia nhỏ tính toán thành các hàm cơ bản. Chúng tôi sẽ nhắc lại chúng ở đây để bạn đọc dễ tham khảo.

- **Đạo hàm hằng số:**  $\frac{d}{dx}c = 0$ .
- **Đạo hàm hàm tuyến tính:**  $\frac{d}{dx}(ax) = a$ .
- **Quy tắc lũy thừa:**  $\frac{d}{dx}x^n = nx^{n-1}$ .
- **Đạo hàm hàm mũ cơ số tự nhiên:**  $\frac{d}{dx}e^x = e^x$ .
- **Đạo hàm hàm logarit cơ số tự nhiên:**  $\frac{d}{dx}\log(x) = \frac{1}{x}$ .

## Các Quy tắc tính Đạo hàm

Nếu mọi đạo hàm cần được tính một cách riêng biệt và lưu vào một bảng, giải tích vi phân sẽ gần như bất khả thi. Toán học đã mang lại một món quà giúp tổng quát hóa các đạo hàm ở phần trên và giúp tính các đạo hàm phức tạp hơn như đạo hàm của  $f(x) = \log(1 + (x - 1)^{10})$ . Như được đề cập trong Section 4.4, chìa khóa để thực hiện việc này là hệ thống hóa việc tính đạo hàm cho các hàm kết hợp theo nhiều cách: tổng, tích và hợp.

- **Quy tắc tổng.**  $\frac{d}{dx}(g(x) + h(x)) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$ .
- **Quy tắc tích.**  $\frac{d}{dx}(g(x) \cdot h(x)) = g(x)\frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$ .
- **Quy tắc dây chuyền.**  $\frac{d}{dx}g(h(x)) = \frac{dg}{dh}(h(x)) \cdot \frac{dh}{dx}(x)$ .

Cùng xem chúng ta có thể sử dụng (20.3.6) như thế nào để hiểu những quy tắc này. Với quy tắc tổng, xét chuỗi biến đổi sau đây:

$$\begin{aligned} f(x + \epsilon) &= g(x + \epsilon) + h(x + \epsilon) \\ &\approx g(x) + \epsilon \frac{dg}{dx}(x) + h(x) + \epsilon \frac{dh}{dx}(x) \\ &= g(x) + h(x) + \epsilon \left( \frac{dg}{dx}(x) + \frac{dh}{dx}(x) \right) \\ &= f(x) + \epsilon \left( \frac{dg}{dx}(x) + \frac{dh}{dx}(x) \right). \end{aligned} \tag{20.3.7}$$

Đồng nhất hệ số với  $f(x + \epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x)$ , ta có  $\frac{df}{dx}(x) = \frac{dg}{dx}(x) + \frac{dh}{dx}(x)$  như mong đợi. Một cách trực quan, ta có thể giải thích như sau: khi thay đổi đầu vào  $x$ ,  $g$  và  $h$  cùng đóng góp tới sự thay đổi của  $\frac{dg}{dx}(x)$  và  $\frac{dh}{dx}(x)$  ở đầu ra.

Đối với quy tắc tích thì phức tạp hơn một chút và đòi hỏi một quan sát mới khi xử lý các biểu thức này. Cũng giống như trước, ta bắt đầu bằng (20.3.6):

$$\begin{aligned} f(x + \epsilon) &= g(x + \epsilon) \cdot h(x + \epsilon) \\ &\approx \left( g(x) + \epsilon \frac{dg}{dx}(x) \right) \cdot \left( h(x) + \epsilon \frac{dh}{dx}(x) \right) \\ &= g(x) \cdot h(x) + \epsilon \left( g(x) \frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) \right) + \epsilon^2 \frac{dg}{dx}(x) \frac{dh}{dx}(x) \\ &= f(x) + \epsilon \left( g(x) \frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) \right) + \epsilon^2 \frac{dg}{dx}(x) \frac{dh}{dx}(x). \end{aligned} \tag{20.3.8}$$

Việc này giống với những tính toán trước đây, và dễ thấy kết quả của ta ( $\frac{df}{dx}(x) = g(x) \frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x)$ ) là số hạng được nhân với  $\epsilon$ , nhưng vấn đề là ở số hạng nhân với giá trị  $\epsilon^2$ . Chúng ta sẽ gọi số hạng này là **số hạng bậc cao**, bởi số mũ của  $\epsilon^2$  cao hơn số mũ của  $\epsilon^1$ . Về sau ta sẽ thấy rằng thi thoảng ta muốn giữ các số hạng này, tuy nhiên hiện tại có thể thấy rằng nếu  $\epsilon = 0.0000001$ , thì  $\epsilon^2 = 0.000000000001$ , là một số nhỏ hơn rất nhiều. Khi đưa  $\epsilon \rightarrow 0$ , ta có thể bỏ qua các số hạng bậc cao. Ta sẽ quy ước sử dụng “≈” để ký hiệu rằng hai số hạng bằng nhau với sai số là các thành phần bậc cao. Nếu muốn biểu diễn chính quy hơn, ta có thể xét phương trình

$$\frac{f(x + \epsilon) - f(x)}{\epsilon} = g(x) \frac{dh}{dx}(x) + \frac{dg}{dx}(x)h(x) + \epsilon \frac{dg}{dx}(x) \frac{dh}{dx}(x), \tag{20.3.9}$$

và thấy rằng khi  $\epsilon \rightarrow 0$ , số hạng bên phải cũng tiến về không.

Cuối cùng, với quy tắc dây chuyền, ta vẫn có thể tiếp tục khai triển sử dụng (20.3.6) và thấy rằng:

$$\begin{aligned}
 f(x + \epsilon) &= g(h(x + \epsilon)) \\
 &\approx g\left(h(x) + \epsilon \frac{dh}{dx}(x)\right) \\
 &\approx g(h(x)) + \epsilon \frac{dh}{dx}(x) \frac{dg}{dh}(h(x)) \\
 &= f(x) + \epsilon \frac{dg}{dh}(h(x)) \frac{dh}{dx}(x).
 \end{aligned} \tag{20.3.10}$$

Chú ý là ở dòng thứ hai trong chuỗi khai triển trên, chúng ta đã xem đối số  $h(x)$  của hàm  $g$  như là bị dịch đi bởi một lượng rất nhỏ  $\epsilon \frac{dh}{dx}(x)$ .

Các quy tắc này cung cấp cho chúng ta một tập hợp các công cụ linh hoạt để tính toán đạo hàm của hầu như bất kỳ biểu thức nào ta muốn. Chẳng hạn như trong ví dụ sau:

$$\begin{aligned}
 \frac{d}{dx} [\log(1 + (x - 1)^{10})] &= (1 + (x - 1)^{10})^{-1} \frac{d}{dx} [1 + (x - 1)^{10}] \\
 &= (1 + (x - 1)^{10})^{-1} \left( \frac{d}{dx}[1] + \frac{d}{dx}[(x - 1)^{10}] \right) \\
 &= (1 + (x - 1)^{10})^{-1} \left( 0 + 10(x - 1)^9 \frac{d}{dx}[x - 1] \right) \\
 &= 10(1 + (x - 1)^{10})^{-1} (x - 1)^9 \\
 &= \frac{10(x - 1)^9}{1 + (x - 1)^{10}}.
 \end{aligned} \tag{20.3.11}$$

Mỗi dòng của ví dụ này đã sử dụng các quy tắc sau:

1. Quy tắc dây chuyền và công thức đạo hàm của hàm logarit.
2. Quy tắc đạo hàm của tổng.
3. Đạo hàm của hằng số, quy tắc dây chuyền, và quy tắc đạo hàm của lũy thừa.
4. Quy tắc đạo hàm của tổng, đạo hàm của hàm tuyến tính, đạo hàm của hằng số.

Từ ví dụ trên, chúng ta có thể dễ dàng rút ra được hai điều:

1. Chúng ta có thể lấy đạo hàm của bất kỳ hàm số nào mà có thể diễn tả được bằng tổng, tích, hằng số, lũy thừa, hàm mũ, và hàm logarit bằng cách sử dụng những quy tắc trên một cách máy móc.
2. Quá trình dùng những quy tắc này để tính đạo hàm bằng tay có thể sẽ rất tẻ nhạt và dễ mắc lỗi.

Rất may là hai điều này gộp chung lại gợi ý cho chúng ta một hướng phát triển: đây chính là cơ hội lý tưởng để tự động hóa bằng máy tính! Thật vậy, kỹ thuật lan truyền ngược, mà chúng ta sẽ gặp lại sau ở mục này, là một cách hiện thực hóa ý tưởng này.

## Xấp xỉ Tuyến tính

Thông thường khi làm việc với đạo hàm, sẽ rất hữu ích nếu chúng ta có thể diễn tả sự xấp xỉ ở trên theo phương diện hình học. Nói một cách cụ thể, phương trình này

$$f(x + \epsilon) \approx f(x) + \epsilon \frac{df}{dx}(x), \quad (20.3.12)$$

xấp xỉ giá trị của  $f$  bằng một đường thẳng đi qua điểm  $(x, f(x))$  và có độ dốc  $\frac{df}{dx}(x)$ . Với cách hiểu này, ta nói rằng đạo hàm cho ta một xấp xỉ tuyến tính của hàm số  $f$ , như minh họa dưới đây:

```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]
# Compute some linear approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0))
d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```

## Đạo hàm Cấp cao

Bây giờ, hãy cùng làm một việc mà nhìn sơ qua thì có vẻ kỳ quặc. Bắt đầu bằng việc lấy một hàm số  $f$  và tính đạo hàm  $\frac{df}{dx}$ . Nó sẽ cho chúng ta tốc độ thay đổi của  $f$  tại bất cứ điểm nào.

Tuy nhiên, vì bản thân đạo hàm  $\frac{df}{dx}$  cũng là một hàm số, không có gì ngăn cản chúng ta tiếp tục tính đạo hàm của  $\frac{df}{dx}$  để có  $\frac{d^2f}{dx^2} = \frac{df}{dx} \left( \frac{df}{dx} \right)$ . Chúng ta sẽ gọi đây là đạo hàm cấp hai của  $f$ . Hàm số này là tốc độ thay đổi của tốc độ thay đổi của  $f$ , hay nói cách khác, nó thể hiện tốc độ thay đổi của  $f$  đang thay đổi như thế nào. Chúng ta có thể tiếp tục lấy đạo hàm như vậy thêm nhiều lần nữa để có được thứ gọi là đạo hàm cấp  $n$ . Để ký hiệu được gọn gàng, chúng ta sẽ biểu thị đạo hàm cấp  $n$  như sau:

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left( \frac{d}{dx} \right)^n f. \quad (20.3.13)$$

Hãy tìm hiểu xem tại sao đây lại là một khái niệm hữu ích. Các hàm số  $f^{(2)}(x)$ ,  $f^{(1)}(x)$ , và  $f(x)$  được biểu diễn trong các đồ thị dưới đây.

Đầu tiên, xét trường hợp đạo hàm bậc hai  $f^{(2)}(x)$  là một hằng số dương. Điều này nghĩa là độ dốc của đạo hàm bậc nhất là dương. Hệ quả là, đạo hàm bậc nhất  $f^{(1)}(x)$  có thể khởi đầu ở âm, bằng không tại một điểm nào đó, rồi cuối cùng tăng lên dương. Điều này cho chúng ta biết độ dốc của hàm  $f$  ban đầu và do đó, giá trị hàm  $f$  sẽ giảm xuống đến điểm nào đó rồi tăng lên. Nói cách khác, đồ thị hàm  $f$  là đường cong đi lên, có một cực tiểu như trong Fig. 20.3.1.

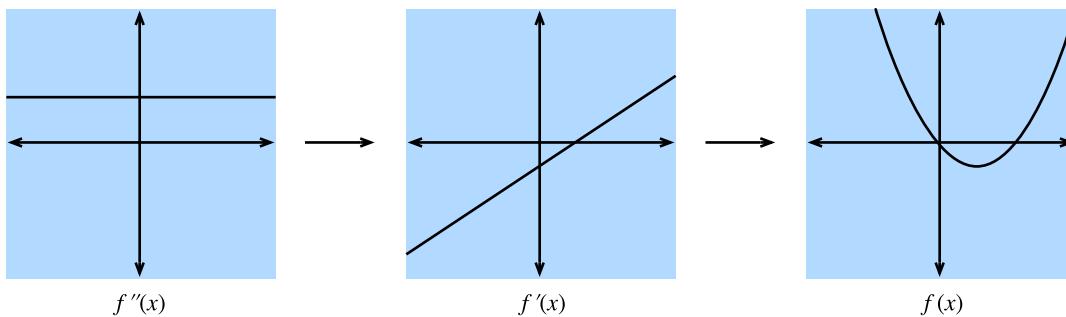


Fig. 20.3.1: Nếu giả định rằng đạo hàm bậc hai là một hằng số dương, thì đạo hàm bậc nhất đồng biến, nghĩa là bản thân hàm đó có một cực tiểu.

Thứ hai là, nếu đạo hàm bậc hai là một hằng số âm, nghĩa là đạo hàm bậc nhất nghịch biến. Vậy tức là đạo hàm bậc nhất có thể khởi đầu là dương, bằng không ở điểm nào đó, rồi giảm xuống âm. Do vậy, giá trị hàm  $f$  tăng lên đến điểm nào đó rồi giảm xuống. Nói cách khác, đồ thị hàm  $f$  là đường cong đi xuống, có một cực đại như trong Fig. 20.3.2.

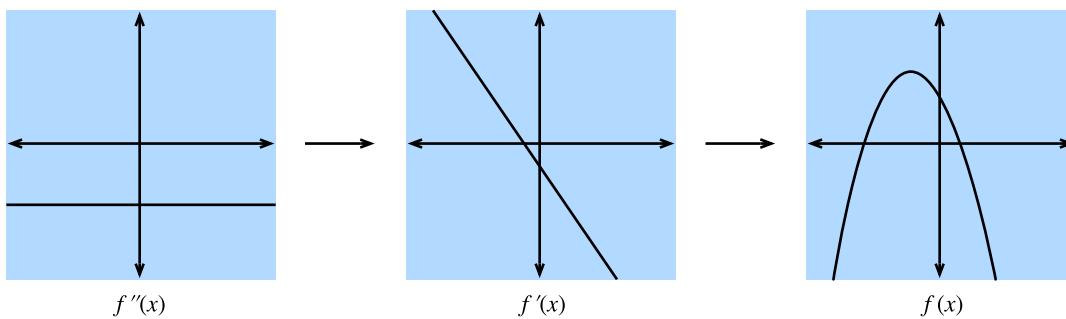


Fig. 20.3.2: Nếu giả định đạo hàm bậc hai là một hằng số âm, thì đạo hàm bậc nhất nghịch biến, nghĩa là hàm số có một cực đại.

Thứ ba là, nếu đạo hàm bậc hai luôn luôn bằng không, thì đạo hàm bậc nhất là hằng số! Nghĩa là hàm  $f$  tăng (hoặc giảm) với tốc độ cố định, và đồ thị  $f$  là một đường thẳng giống như trong Fig. 20.3.3.

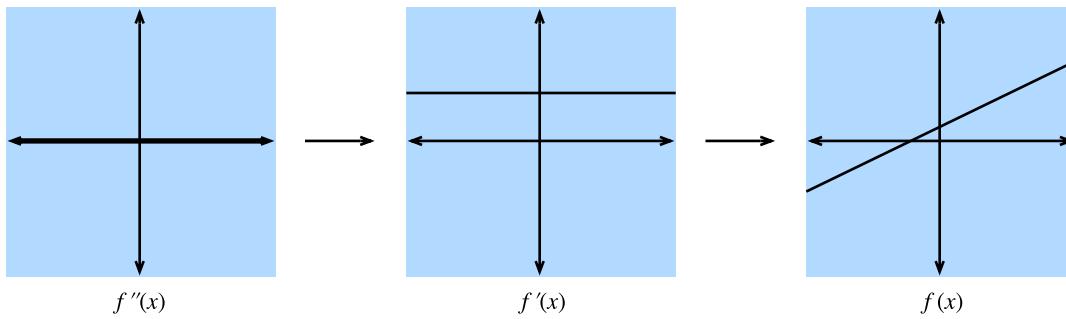


Fig. 20.3.3: Nếu ta giả định đạo hàm bậc hai bằng không, thì đạo hàm bậc nhất là hằng số, nên đồ thị hàm này là một đường thẳng.

Tóm lại, đạo hàm bậc hai có thể được hiểu như một cách miêu tả đường cong của đồ thị hàm  $f$ . Đạo hàm bậc hai dương thì đồ thị cong lên, đạo hàm bậc hai âm thì hàm  $f$  cong xuống, và nếu

bằng không thì  $f$  là một đường thẳng.

Hãy thử tiến xa hơn một bước. Xét hàm  $g(x) = ax^2 + bx + c$ . Ta có thể tính được

$$\begin{aligned}\frac{dg}{dx}(x) &= 2ax + b \\ \frac{d^2g}{dx^2}(x) &= 2a.\end{aligned}\tag{20.3.14}$$

Nếu đã có sẵn một hàm  $f(x)$ , ta có thể tính đạo hàm cấp một và cấp hai của nó để tìm các giá trị  $a, b$ , và  $c$  thỏa mãn hệ phương trình này. Cũng giống như ở mục trước ta đã thấy đạo hàm bậc một cho ra xấp xỉ tốt nhất bằng một đường thẳng, đạo hàm bậc hai cung cấp một xấp xỉ tốt nhất bằng một parabol. Hãy minh họa với trường hợp  $f(x) = \sin(x)$ .

```
# Compute sin
xs = np.arange(-np.pi, np.pi, 0.01)
plots = [np.sin(xs)]
# Compute some quadratic approximations. Use d(sin(x)) / dx = cos(x)
for x0 in [-1.5, 0, 2]:
    plots.append(np.sin(x0) + (xs - x0) * np.cos(x0) -
                 (xs - x0)**2 * np.sin(x0) / 2)
d2l.plot(xs, plots, 'x', 'f(x)', ylim=[-1.5, 1.5])
```

Ta sẽ mở rộng ý tưởng này thành ý tưởng của *chuỗi Taylor* trong mục tiếp theo.

## Chuỗi Taylor

*Chuỗi Taylor* cung cấp một phương pháp để xấp xỉ phương trình  $f(x)$  nếu ta đã biết trước giá trị của  $n$  cấp đạo hàm đầu tiên tại điểm  $x_0$ :  $\{f(x_0), f^{(1)}(x_0), f^{(2)}(x_0), \dots, f^{(n)}(x_0)\}$ . Ý tưởng là tìm một đa thức bậc  $n$  có các đạo hàm tại  $x_0$  khớp với các đạo hàm đã biết.

Ta đã thấy với trường hợp  $n = 2$  ở chương trước và với một chút biến đổi đại số, ta có được

$$f(x) \approx \frac{1}{2} \frac{d^2f}{dx^2}(x_0)(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0).\tag{20.3.15}$$

Như ta đã thấy ở trên, mẫu số 2 là để rút gọn thừa số 2 khi lấy đạo hàm bậc hai của  $x^2$ , các đạo hàm bậc cao hơn đều bằng không. Cùng một cách lập luận cũng được áp dụng cho đạo hàm bậc một và phần giá trị  $f(x_0)$ .

Nếu ta mở rộng cách lập luận này cho trường hợp  $n = 3$ , ta sẽ kết luận được

$$f(x) \approx \frac{\frac{d^3f}{dx^3}(x_0)}{6}(x - x_0)^3 + \frac{\frac{d^2f}{dx^2}(x_0)}{2}(x - x_0)^2 + \frac{df}{dx}(x_0)(x - x_0) + f(x_0).\tag{20.3.16}$$

với  $6 = 3 \times 2 = 3!$  đến từ phần hằng số ta có được khi lấy đạo hàm bậc 3 của  $x^3$ .

Hơn nữa, ta có thể lấy một đa thức bậc  $n$  bằng cách

$$P_n(x) = \sum_{i=0}^n \frac{f^{(i)}(x_0)}{i!}(x - x_0)^i.\tag{20.3.17}$$

với quy ước

$$f^{(n)}(x) = \frac{d^n f}{dx^n} = \left(\frac{d}{dx}\right)^n f.\tag{20.3.18}$$

Quả thật,  $P_n(x)$  có thể được xem là đa thức bậc  $n$  xấp xỉ tốt nhất của hàm  $f(x)$ .

Dù ta sẽ không tìm hiểu kỹ sai số của xấp xỉ này, ta cũng nên nhắc tới giới hạn vô cùng. Trong trường hợp này, các hàm khả vi vô hạn lần như  $\cos(x)$  hoặc  $e^x$  có thể được biểu diễn xấp xỉ bằng vô số các số hạng.

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n. \quad (20.3.19)$$

Lấy hàm  $f(x) = e^x$  làm ví dụ. Vì  $e^x$  là đạo hàm của chính nó, ta có  $f^{(n)}(x) = e^x$ . Do đó, hàm  $e^x$  có thể được tái tạo bằng cách tính chuỗi Taylor tại  $x_0 = 0$ :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \dots. \quad (20.3.20)$$

Hãy cùng tìm hiểu cách lập trình và quan sát xem việc tăng bậc của xấp xỉ Taylor đưa ta đến gần hơn với hàm mong muốn  $e^x$  như thế nào.

```
# Compute the exponential function
xs = np.arange(0, 3, 0.01)
ys = np.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"])
```

Chuỗi Taylor có hai ứng dụng chính:

1. *Ứng dụng lý thuyết*: Khi muốn tìm hiểu một hàm số quá phức tạp, ta thường dùng chuỗi Taylor để biến nó thành một đa thức để có thể làm việc trực tiếp.
2. *Ứng dụng số học*: Việc tính toán một số hàm như  $e^x$  hoặc  $\cos(x)$  không đơn giản đối với máy tính. Chúng có thể lưu trữ một bảng giá trị với độ chính xác nhất định (và thường thì chúng làm vậy), nhưng việc đó vẫn không giải quyết được những câu hỏi như “Chữ số thứ 1000 của  $\cos(1)$  là gì?”. Chuỗi Taylor thường có ích cho việc trả lời các câu hỏi như vậy.

### 20.3.3 Tóm tắt

- Đạo hàm có thể được sử dụng để biểu diễn mức độ thay đổi của hàm số khi đầu vào thay đổi một lượng nhỏ.
- Các phép lấy đạo hàm cơ bản có thể kết hợp với nhau theo các quy tắc đạo hàm để tính những đạo hàm phức tạp tùy ý.
- Đạo hàm có thể được tính nhiều lần để lấy đạo hàm cấp hai hoặc các cấp cao hơn. Mỗi lần tăng cấp đạo hàm cho ta thông tin chi tiết hơn về hành vi của hàm số.
- Bằng việc sử dụng thông tin từ đạo hàm của một điểm dữ liệu, ta có thể xấp xỉ các hàm khả vi vô hạn lần bằng các đa thức lấy từ chuỗi Taylor.

#### 20.3.4 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 cực đại hoặc cực tiểu tại  $x$ ?
4. Cực tiểu của  $f(x) = x \log(x)$  với  $x \geq 0$  ở đâu (ở đây ta giả sử rằng  $f$  có giới hạn bằng 0 tại  $f(0)$ )?

#### 20.3.5 Thảo luận

- Tiếng Anh: MXNet<sup>411</sup>, Pytorch<sup>412</sup>, Tensorflow<sup>413</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>414</sup>

#### 20.3.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Nguyễn Lê Quang Nhật
- Đoàn Võ Duy Thanh
- Tạ H. Duy Nguyên
- Mai Sơn Hải
- Phạm Minh Đức
- Nguyễn Văn Tâm
- Nguyễn Văn Cường

### 20.4 Giải tích Nhiều biến

Bây giờ chúng ta đã có hiểu biết vững chắc về đạo hàm của một hàm đơn biến, hãy cùng trở lại câu hỏi ban đầu về hàm mất mát của (nhiều khả năng là) hàng tỷ trọng số.

<sup>411</sup> <https://discuss.d2l.ai/t/412>

<sup>412</sup> <https://discuss.d2l.ai/t/1088>

<sup>413</sup> <https://discuss.d2l.ai/t/1089>

<sup>414</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### 20.4.1 Đạo hàm trong Không gian Nhiều chiều

Nhớ lại Section 20.3, ta đã bàn luận về điều gì sẽ xảy ra nếu chỉ thay đổi một trong số hàng tỷ các trọng số và giữ nguyên những trọng số còn lại. Điều này hoàn toàn không có gì khác với một hàm đơn biến, nên ta có thể viết

$$L(w_1 + \epsilon_1, w_2, \dots, w_N) \approx L(w_1, w_2, \dots, w_N) + \epsilon_1 \frac{d}{dw_1} L(w_1, w_2, \dots, w_N). \quad (20.4.1)$$

Chúng ta sẽ gọi đạo hàm của một biến trong khi không thay đổi những biến còn lại là *đạo hàm riêng* (*partial derivative*), và ký hiệu đạo hàm này là  $\frac{\partial}{\partial w_1}$  trong phương trình (20.4.1).

Bây giờ, tiếp tục thay đổi  $w_2$  một khoảng nhỏ thành  $w_2 + \epsilon_2$ :

$$\begin{aligned} L(w_1 + \epsilon_1, w_2 + \epsilon_2, \dots, w_N) &\approx L(w_1, w_2 + \epsilon_2, \dots, w_N) + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2 + \epsilon_2, \dots, w_N) \\ &\approx L(w_1, w_2, \dots, w_N) \\ &\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \dots, w_N) \\ &\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \dots, w_N) \\ &\quad + \epsilon_1 \epsilon_2 \frac{\partial}{\partial w_2} \frac{\partial}{\partial w_1} L(w_1, w_2, \dots, w_N) \\ &\approx L(w_1, w_2, \dots, w_N) \\ &\quad + \epsilon_2 \frac{\partial}{\partial w_2} L(w_1, w_2, \dots, w_N) \\ &\quad + \epsilon_1 \frac{\partial}{\partial w_1} L(w_1, w_2, \dots, w_N). \end{aligned} \quad (20.4.2)$$

Một lần nữa, ta lại sử dụng ý tưởng đã thấy ở (20.4.1) rằng  $\epsilon_1 \epsilon_2$  là một số hạng bậc cao và có thể được loại bỏ tương tự như cách mà ta có thể loại bỏ  $\epsilon^2$  trong mục trước. Cứ tiếp tục theo cách này, ta có

$$L(w_1 + \epsilon_1, w_2 + \epsilon_2, \dots, w_N + \epsilon_N) \approx L(w_1, w_2, \dots, w_N) + \sum_i \epsilon_i \frac{\partial}{\partial w_i} L(w_1, w_2, \dots, w_N). \quad (20.4.3)$$

Thoạt nhìn đây có vẻ là một mớ hỗn độn, tuy nhiên chú ý rằng phép tổng bên phải chính là biểu diễn của phép tích vô hướng và ta có thể khiến chúng trở nên quen thuộc hơn. Với

$$\boldsymbol{\epsilon} = [\epsilon_1, \dots, \epsilon_N]^\top \text{ và } \nabla_{\mathbf{x}} L = \left[ \frac{\partial L}{\partial x_1}, \dots, \frac{\partial L}{\partial x_N} \right]^\top, \quad (20.4.4)$$

ta có

$$L(\mathbf{w} + \boldsymbol{\epsilon}) \approx L(\mathbf{w}) + \boldsymbol{\epsilon} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \quad (20.4.5)$$

Ta gọi vector  $\nabla_{\mathbf{w}} L$  là *gradient* của  $L$ .

Phương trình (20.4.5) đáng để ta suy ngẫm. Nó có dạng đúng y như những gì ta đã thấy trong trường hợp một chiều, chỉ khác là tất cả đã được biến đổi về dạng vector và tích vô hướng. Điều này cho chúng ta biết một cách xấp xỉ hàm  $L$  sẽ thay đổi như thế nào với một nhiễu loạn bất kỳ ở

đầu vào. Như ta sẽ thấy trong mục tiếp theo, đây sẽ là một công cụ quan trọng giúp chúng ta hiểu được cách học từ thông tin chứa trong gradient dưới góc nhìn hình học.

Nhưng trước tiên, hãy cùng kiểm tra phép xấp xỉ này với một ví dụ. Giả sử ta đang làm việc với hàm

$$f(x, y) = \log(e^x + e^y) \text{ với gradient } \nabla f(x, y) = \left[ \frac{e^x}{e^x + e^y}, \frac{e^y}{e^x + e^y} \right]. \quad (20.4.6)$$

Xét một điểm  $(0, \log(2))$ , ta có

$$f(x, y) = \log(3) \text{ với gradient } \nabla f(x, y) = \left[ \frac{1}{3}, \frac{2}{3} \right]. \quad (20.4.7)$$

Vì thế, nếu muốn tính xấp xỉ  $f$  tại  $(\epsilon_1, \log(2) + \epsilon_2)$ , ta có một ví dụ cụ thể của (20.4.5):

$$f(\epsilon_1, \log(2) + \epsilon_2) \approx \log(3) + \frac{1}{3}\epsilon_1 + \frac{2}{3}\epsilon_2. \quad (20.4.8)$$

Ta có thể kiểm tra với đoạn mã bên dưới để xem phép xấp xỉ chính xác tới mức nào.

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import autograd, np, npx
npx.set_np()

def f(x, y):
    return np.log(np.exp(x) + np.exp(y))
def grad_f(x, y):
    return np.array([np.exp(x) / (np.exp(x) + np.exp(y)),
                   np.exp(y) / (np.exp(x) + np.exp(y))])

epsilon = np.array([0.01, -0.03])
grad_approx = f(0, np.log(2)) + epsilon.dot(grad_f(0, np.log(2)))
true_value = f(0 + epsilon[0], np.log(2) + epsilon[1])
f'approximation: {grad_approx}, true Value: {true_value}'
```

## 20.4.2 Ý nghĩa Hình học của Gradient và Thuật toán Hạ Gradient

Nhìn lại (20.4.5):

$$L(\mathbf{w} + \epsilon) \approx L(\mathbf{w}) + \epsilon \cdot \nabla_{\mathbf{w}} L(\mathbf{w}). \quad (20.4.9)$$

Giả sử ta muốn sử dụng thông tin gradient để cực tiểu hóa mất mát  $L$ . Hãy cùng tìm hiểu cách hoạt động về mặt hình học của thuật toán hạ gradient được mô tả lần đầu ở Section 4.5. Các bước của thuật toán được miêu tả dưới đây:

1. Bắt đầu với giá trị ban đầu ngẫu nhiên của tham số  $\mathbf{w}$ .
2. Tìm một hướng  $\mathbf{v}$  tại  $\mathbf{w}$  sao cho  $L$  giảm một cách nhanh nhất.
3. Tiến một bước nhỏ về hướng đó:  $\mathbf{w} \rightarrow \mathbf{w} + \epsilon\mathbf{v}$ .

#### 4. Lặp lại.

Thứ duy nhất mà chúng ta không biết chính xác cách làm là cách tính toán vector  $\mathbf{v}$  tại bước thứ hai. Ta gọi  $\mathbf{v}$  là *hướng hạ dốc nhất* (*direction of steepest descent*). Sử dụng những hiểu biết về mặt hình học của phép tích vô hướng từ Section 20.1, ta có thể viết lại (20.4.5) như sau

$$L(\mathbf{w} + \mathbf{v}) \approx L(\mathbf{w}) + \mathbf{v} \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) = L(\mathbf{w}) + \|\nabla_{\mathbf{w}} L(\mathbf{w})\| \cos(\theta). \quad (20.4.10)$$

Để thuận tiện, ta giả định hướng của chúng ta có độ dài bằng một và sử dụng  $\theta$  để biểu diễn góc giữa  $\mathbf{v}$  và  $\nabla_{\mathbf{w}} L(\mathbf{w})$ . Nếu muốn  $L$  giảm càng nhanh, ta sẽ muốn giá trị của biểu thức trên càng âm càng tốt. Cách duy nhất để chọn hướng đi trong phương trình này là thông qua  $\cos(\theta)$ , vì thế ta sẽ muốn giá trị này âm nhất có thể. Nhắc lại kiến thức của hàm cô-sin, giá trị âm nhất của hàm này là  $\cos(\theta) = -1$ , là khi góc giữa vector gradient và hướng cần chọn là  $\pi$  radian hay 180 độ. Cách duy nhất để đạt được điều này là di chuyển theo hướng hoàn toàn ngược lại: chọn  $\mathbf{v}$  theo hướng hoàn toàn ngược chiều với  $\nabla_{\mathbf{w}} L(\mathbf{w})$ !

Điều này dẫn ta đến với một trong những thuật toán quan trọng nhất của học máy: hướng hạ dốc nhất cùng hướng với  $-\nabla_{\mathbf{w}} L(\mathbf{w})$ . Vậy nên thuật toán của ta sẽ được viết lại như sau.

1. Bắt đầu với một lựa chọn ngẫu nhiên cho giá trị ban đầu của các tham số  $\mathbf{w}$ .
2. Tính toán  $\nabla_{\mathbf{w}} L(\mathbf{w})$ .
3. Tiến một bước nhỏ về hướng ngược lại của nó:  $\mathbf{w} \rightarrow \mathbf{w} - \epsilon \nabla_{\mathbf{w}} L(\mathbf{w})$ .
4. Lặp lại.

Thuật toán cơ bản này dù đã được chỉnh sửa và kết hợp theo nhiều cách bởi các nhà nghiên cứu, nhưng khái niệm cốt lõi vẫn là như nhau. Sử dụng gradient để tìm hướng giảm mất mát nhanh nhất có thể và cập nhật các tham số để dịch chuyển về hướng đó.

#### 20.4.3 Một vài chú ý về Tối ưu hóa

Xuyên suốt cuốn sách, ta chỉ tập trung vào những kỹ thuật tối ưu hóa số học vì một nguyên nhân thực tế là: mọi hàm ta gặp phải trong học sâu quá phức tạp để có thể tối ưu hóa một cách tường minh.

Tuy nhiên, sẽ rất hữu ích nếu hiểu được những kiến thức hình học ta có được ở trên nói gì về tối ưu hóa các hàm một cách trực tiếp.

Giả sử ta muốn tìm giá trị của  $\mathbf{x}_0$  giúp cực tiểu hóa một hàm  $L(\mathbf{x})$  nào đó. Và có một người nào đó đưa ta một giá trị và cho rằng đây là giá trị giúp cực tiểu hóa  $L$ . Bằng cách nào ta có thể kiểm chứng rằng đáp án của họ là hợp lý?

Xét lại (20.4.5):

$$L(\mathbf{x}_0 + \epsilon) \approx L(\mathbf{x}_0) + \epsilon \cdot \nabla_{\mathbf{x}} L(\mathbf{x}_0). \quad (20.4.11)$$

Nếu giá trị gradient khác không, ta biết rằng ta có thể bước một bước về hướng  $-\epsilon \nabla_{\mathbf{x}} L(\mathbf{x}_0)$  để tìm một giá trị  $L$  nhỏ hơn. Do đó, nếu ta thực sự ở điểm cực tiểu, sẽ không thể có trường hợp đó! Ta có thể kết luận rằng nếu  $\mathbf{x}_0$  là một cực tiểu, thì  $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$ . Ta gọi những điểm mà tại đó  $\nabla_{\mathbf{x}} L(\mathbf{x}_0) = 0$  là *các điểm tối hạn* (*critical points*).

Điều này rất hữu ích, bởi vì trong một vài thiết lập hiếm gặp, ta có thể tìm được các điểm có gradient bằng không một cách tường minh, và từ đó tìm được điểm có giá trị nhỏ nhất.

Với một ví dụ cụ thể, xét hàm

$$f(x) = 3x^4 - 4x^3 - 12x^2. \quad (20.4.12)$$

Hàm này có đạo hàm

$$\frac{df}{dx} = 12x^3 - 12x^2 - 24x = 12x(x-2)(x+1). \quad (20.4.13)$$

Các điểm cực trị duy nhất khả dĩ là tại  $x = -1, 0, 2$ , khi hàm lấy giá trị lần lượt là  $-5, 0, -32$ , và do đó ta có thể kết luận rằng ta cực tiểu hóa hàm khi  $x = 2$ . Ta có thể kiểm chứng nhanh bằng đồ thị.

```
x = np.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 kể cả khi làm việc dưới dạng lý thuyết hay số học: các điểm khả dĩ duy nhất mà tại đó hàm là cực tiểu (hoặc cực đại) sẽ có đạo hàm tại đó bằng không, tuy nhiên, không phải tất cả các điểm có đạo hàm bằng không sẽ là cực tiểu (hay cực đại) *toàn cục*.

#### 20.4.4 Quy tắc Dây chuyền cho Hàm đa biến

Giả sử là ta có một hàm bốn biến ( $w, x, y$ , and  $z$ ) được tạo ra bằng cách kết hợp các hàm con:

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \quad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \quad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \quad (20.4.14)$$

Các chuỗi phương trình như trên xuất hiện thường xuyên khi ta làm việc với các mạng neural, do đó cố gắng hiểu xem làm thế nào để tính gradient của các hàm này là thiết yếu. Fig. 20.4.1 biểu diễn trực quan mối liên hệ trực tiếp giữa biến này với biến khác.

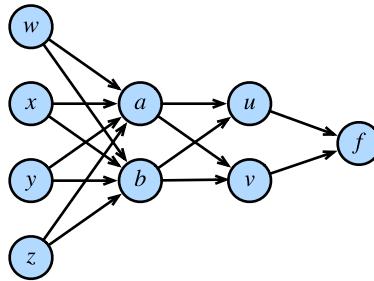


Fig. 20.4.1: Các quan hệ của hàm ở trên với các nút biểu diễn giá trị và mũi tên cho biết sự phụ thuộc hàm.

Ta có thể kết hợp các phương trình trong (20.4.14) để có

$$f(w, x, y, z) = \left( ((w + x + y + z)^2 + (w + x - y - z)^2)^2 + ((w + x + y + z)^2 - (w + x - y - z)^2)^2 \right)^2. \quad (20.4.15)$$

Tiếp theo ta có thể lấy đạo hàm bằng cách chỉ sử dụng các đạo hàm đơn biến, nhưng nếu làm vậy ta sẽ nhanh chóng bị ngợp trong các số hạng, mà đa phần là bị lặp lại! Thật vậy, ta có thể thấy ở ví dụ dưới đây:

$$\begin{aligned} \frac{\partial f}{\partial w} = & 2(2(2(w+x+y+z) - 2(w+x-y-z))((w+x+y+z)^2 - (w+x-y-z)^2) + \\ & 2(2(w+x-y-z) + 2(w+x+y+z))((w+x-y-z)^2 + (w+x+y+z)^2)) \times \\ & (((w+x+y+z)^2 - (w+x-y-z)^2)^2 + ((w+x-y-z)^2 + (w+x+y+z)^2)^2). \end{aligned} \quad (20.4.16)$$

Kế đến nếu ta cũng muốn tính  $\frac{\partial f}{\partial x}$ , ta sẽ lại kết thúc với một phương trình tương tự với nhiều thành phần bị lặp lại, và nhiều thành phần lặp lại chung giữa hai đạo hàm. Điều này thể hiện một khối lượng lớn công việc bị lãng phí, và nếu ta tính các đạo hàm theo cách này, toàn bộ cuộc cách mạng học sâu sẽ chấm dứt trước khi nó bắt đầu!

Ta hãy chia nhỏ vấn đề này. Ta sẽ bắt đầu bằng cách thử hiểu  $f$  thay đổi thế nào khi  $a$  thay đổi, giả định cần thiết là tất cả  $w, x, y$ , và  $z$  không tồn tại. Ta sẽ lập luận giống như lần đầu tiên ta làm việc với gradient. Hãy lấy  $a$  và cộng một lượng nhỏ  $\epsilon$  vào nó.

$$\begin{aligned} & f(u(a+\epsilon, b), v(a+\epsilon, b)) \\ & \approx f\left(u(a, b) + \epsilon \frac{\partial u}{\partial a}(a, b), v(a, b) + \epsilon \frac{\partial v}{\partial a}(a, b)\right) \\ & \approx f(u(a, b), v(a, b)) + \epsilon \left[ \frac{\partial f}{\partial u}(u(a, b), v(a, b)) \frac{\partial u}{\partial a}(a, b) + \frac{\partial f}{\partial v}(u(a, b), v(a, b)) \frac{\partial v}{\partial a}(a, b) \right]. \end{aligned} \quad (20.4.17)$$

Dòng đầu tiên theo sau từ định nghĩa đạo hàm từng phần, và dòng thứ hai theo sau từ định nghĩa gradient. Thật khó khăn để lần theo các biến khi tính đạo hàm, như trong biểu thức  $\frac{\partial f}{\partial u}(u(a, b), v(a, b))$ , cho nên ta thường rút gọn nó để dễ nhớ hơn

$$\frac{\partial f}{\partial a} = \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a}. \quad (20.4.18)$$

Sẽ rất hữu ích khi ta suy nghĩ về ý nghĩa của biến đổi này. Ta đang cố gắng hiểu làm thế nào một hàm có dạng  $f(u(a, b), v(a, b))$  thay đổi giá trị của nó khi  $a$  thay đổi. Có hai hướng có thể xảy ra:  $a \rightarrow u \rightarrow f$  và  $a \rightarrow v \rightarrow f$ . Ta có thể lần lượt tính toán đóng góp của cả hai hướng này thông qua quy tắc dây chuyền:  $\frac{\partial w}{\partial u} \cdot \frac{\partial u}{\partial x}$  và  $\frac{\partial w}{\partial v} \cdot \frac{\partial v}{\partial x}$ , rồi cộng gộp lại.

các hàm được kết nối ở bên trái như trong Fig. 20.4.2.

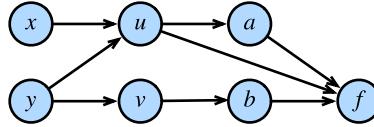


Fig. 20.4.2: Một ví dụ khác về quy tắc dây chuyền.

Để tính toán  $\frac{\partial f}{\partial y}$ , chúng ta cần tính tổng toàn bộ đường đi từ  $y$  đến  $f$  (trường hợp này có 3 đường đi):

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial a} \frac{\partial a}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial f}{\partial u} \frac{\partial u}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial v} \frac{\partial v}{\partial y}. \quad (20.4.19)$$

Hiểu quy tắc dây chuyền theo cách này giúp chúng ta thấy được dòng chảy của gradient xuyên suốt mạng và vì sao một số lựa chọn kiến trúc như trong LSTM (Section 11.2) hoặc các tầng phần dư (Section 9.6) có thể định hình quá trình học bằng cách kiểm soát dòng chảy gradient.

#### 20.4.5 Thuật toán Lan truyền ngược (*Backpropagation*)

Hãy xem lại ví dụ (20.4.14) ở phần trước:

$$\begin{aligned} f(u, v) &= (u + v)^2 \\ u(a, b) &= (a + b)^2, \quad v(a, b) = (a - b)^2, \\ a(w, x, y, z) &= (w + x + y + z)^2, \quad b(w, x, y, z) = (w + x - y - z)^2. \end{aligned} \tag{20.4.20}$$

Nếu muốn tính  $\frac{\partial f}{\partial w}$  chẳng hạn, ta có thể áp dụng quy tắc dây chuyền đa biến để thấy:

$$\begin{aligned} \frac{\partial f}{\partial w} &= \frac{\partial f}{\partial u} \frac{\partial u}{\partial w} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial w}, \\ \frac{\partial u}{\partial w} &= \frac{\partial u}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial u}{\partial b} \frac{\partial b}{\partial w}, \\ \frac{\partial v}{\partial w} &= \frac{\partial v}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial v}{\partial b} \frac{\partial b}{\partial w}. \end{aligned} \tag{20.4.21}$$

Chúng ta hãy thử sử dụng cách phân tách này để tính  $\frac{\partial f}{\partial w}$ . Tất cả những gì chúng ta cần ở đây là các đạo hàm riêng:

$$\begin{aligned} \frac{\partial f}{\partial u} &= 2(u + v), & \frac{\partial f}{\partial v} &= 2(u + v), \\ \frac{\partial u}{\partial a} &= 2(a + b), & \frac{\partial u}{\partial b} &= 2(a + b), \\ \frac{\partial v}{\partial a} &= 2(a - b), & \frac{\partial v}{\partial b} &= -2(a - b), \\ \frac{\partial a}{\partial w} &= 2(w + x + y + z), & \frac{\partial b}{\partial w} &= 2(w + x - y - z). \end{aligned} \tag{20.4.22}$$

Khi lập trình, các tính toán này 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)

# 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}'')
```

Tuy nhiên, cần lưu ý rằng điều này không làm cho các phép tính chẵng hạn như  $\frac{\partial f}{\partial x}$  trở nên đơn giản. Lý do nằm ở cách chúng ta chọn để áp dụng quy tắc dây chuyền. Nếu nhìn vào những gì chúng ta đã làm ở trên, chúng ta luôn giữ  $\partial w$  ở mấu khi có thể. Với cách này, chúng ta áp dụng quy tắc dây chuyền để xem  $w$  thay đổi các biến khác như thế nào. Nếu đó là những gì chúng ta muốn thì cách này quả là một ý tưởng hay. Tuy nhiên, nghĩ lại về mục tiêu của học sâu: chúng ta muốn thấy từng tham số thay đổi giá trị *mất mát* như thế nào. Về cốt lõi, chúng ta luôn muốn áp dụng quy tắc dây chuyền và giữ  $\partial f$  ở từ số bất cứ khi nào có thể!

Cụ thể hơn, chúng ta có thể viết như sau:

$$\begin{aligned}\frac{\partial f}{\partial w} &= \frac{\partial f}{\partial a} \frac{\partial a}{\partial w} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial w}, \\ \frac{\partial f}{\partial a} &= \frac{\partial f}{\partial u} \frac{\partial u}{\partial a} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial a}, \\ \frac{\partial f}{\partial b} &= \frac{\partial f}{\partial u} \frac{\partial u}{\partial b} + \frac{\partial f}{\partial v} \frac{\partial v}{\partial b}.\end{aligned}\tag{20.4.23}$$

Lưu ý rằng cách áp dụng quy tắc dây chuyền này buộc chúng ta phải tính rõ  $\frac{\partial f}{\partial u}$ ,  $\frac{\partial f}{\partial v}$ ,  $\frac{\partial f}{\partial a}$ ,  $\frac{\partial f}{\partial b}$ , và  $\frac{\partial f}{\partial w}$ . Chúng ta cũng có thể thêm vào các phương trình:

$$\begin{aligned}\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial x}, \\ \frac{\partial f}{\partial y} &= \frac{\partial f}{\partial a} \frac{\partial a}{\partial y} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial y}, \\ \frac{\partial f}{\partial z} &= \frac{\partial f}{\partial a} \frac{\partial a}{\partial z} + \frac{\partial f}{\partial b} \frac{\partial b}{\partial z}.\end{aligned}\tag{20.4.24}$$

và tiếp đó theo dõi  $f$  biến đổi như thế nào khi chúng ta thay đổi *bất kỳ* nút nào trong toàn bộ mạng. Hãy cùng lập trình 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)

# 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}'')
```

Việc tính đạo hàm từ  $f$  trở ngược về đầu vào thay vì từ đầu vào đến đầu ra (như chúng ta đã thực hiện ở đoạn mã đầu tiên ở trên) là lý do cho cái tên *làn truyền ngược (backpropagation)* của thuật toán. Có hai bước:

1. Tính giá trị của hàm và đạo hàm riêng theo từng bước đơn lẻ từ đầu đến cuối. Mặc dù không được thực hiện ở trên, hai việc này có thể được kết hợp vào một *lượt truyền xuôi* duy nhất.
2. Tính toán đạo hàm của  $f$  từ cuối về đầu. Chúng ta gọi đó là *lượt truyền ngược*.

Đây chính xác là những gì mỗi thuật toán học sâu thực thi để tính gradient của giá trị mất mát theo từng trọng số của mạng trong mỗi lượt lan truyền. Thật thú vị vì chúng ta có một sự phân tách như trên.

Để tóm gọn phần này, hãy xem nhanh ví dụ sau.

```
# Initialize as ndarrays, then attach gradients
w, x, y, z = np.array(-1), np.array(0), np.array(-2), np.array(1)

w.attach_grad()
x.attach_grad()
y.attach_grad()
z.attach_grad()

# Do the computation like usual, tracking gradients
with autograd.record():
    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}, {x}, {y}, {z} is {w.grad}')
print(f'df/dx at {w}, {x}, {y}, {z} is {x.grad}')
print(f'df/dy at {w}, {x}, {y}, {z} is {y.grad}')
print(f'df/dz at {w}, {x}, {y}, {z} is {z.grad}'')
```

Tất cả những gì chúng ta làm ở trên có thể được thực hiện tự động bằng cách gọi hàm `f.backward()`.

#### 20.4.6 Hessian

Như với giải tích đơn biến, việc xem xét đạo hàm bậc cao hơn cũng hữu ích để xấp xỉ tốt hơn một hàm so với việc chỉ sử dụng gradient.

Một vấn đề trước mắt khi làm việc với đạo hàm bậc cao hơn của hàm đa biến đó là cần phải tính toán một số lượng lớn đạo hàm. Nếu chúng ta có một hàm  $f(x_1, \dots, x_n)$  với  $n$  biến, chúng ta có thể cần  $n^2$  đạo hàm bậc 2, chẳng hạn để lựa chọn  $i$  và  $j$ :

$$\frac{d^2 f}{dx_i dx_j} = \frac{d}{dx_i} \left( \frac{d}{dx_j} f \right). \quad (20.4.25)$$

Biểu thức này được hợp thành một ma trận gọi là *Hessian*:

$$\mathbf{H}_f = \begin{bmatrix} \frac{d^2 f}{dx_1 dx_1} & \cdots & \frac{d^2 f}{dx_1 dx_n} \\ \vdots & \ddots & \vdots \\ \frac{d^2 f}{dx_n dx_1} & \cdots & \frac{d^2 f}{dx_n dx_n} \end{bmatrix}. \quad (20.4.26)$$

Không phải mọi hạng tử của ma trận này đều độc lập. Thật vậy, chúng ta có thể chứng minh rằng miễn là cả hai *đạo hàm riêng hỗn hợp - mixed partials* (đạo hàm riêng theo nhiều hơn một biến số) có tồn tại và liên tục, thì hàm số luôn tồn tại và liên tục với mọi  $i$  và  $j$ ,

$$\frac{d^2 f}{dx_i dx_j} = \frac{d^2 f}{dx_j dx_i}. \quad (20.4.27)$$

Điều này suy ra được bằng việc xem xét khi ta thay đổi hàm lần lượt theo  $x_i$  rồi  $x_j$ , và ngược lại thay đổi  $x_j$  rồi  $x_i$ , và so sánh hai kết quả này, biết rằng cả hai thứ tự này ảnh hưởng đến đầu ra của  $f$  như nhau.

Như với các hàm đơn biến, chúng ta có thể sử dụng những đạo hàm này để hiểu rõ hơn về hành vi của hàm số lân cận một điểm. Cụ thể, chúng ta có thể sử dụng nó để tìm hàm bậc hai phù hợp nhất lân cận  $\mathbf{x}_0$  tương tự như trong giải tích đơn biến.

Hãy tham khảo 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 một dạng tổng quát của hàm bậc hai 2 biến. Nếu chúng ta nhìn vào giá trị của hàm, gradient và Hessian của nó (20.4.26), tất cả tại điểm 0:

$$\begin{aligned} f(0, 0) &= a, \\ \nabla f(0, 0) &= \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \\ \mathbf{H}f(0, 0) &= \begin{bmatrix} 2c_{11} & c_{12} \\ c_{12} & 2c_{22} \end{bmatrix}, \end{aligned} \quad (20.4.28)$$

Chúng ta có thể thu lại được đa thức ban đầu bằng cách đặt:

$$f(\mathbf{x}) = f(0) + \nabla f(0) \cdot \mathbf{x} + \frac{1}{2} \mathbf{x}^\top \mathbf{H}f(0) \mathbf{x}. \quad (20.4.29)$$

Nhìn chung, nếu chúng ta tính toán khai triển này tại mọi điểm  $\mathbf{x}_0$ , ta có:

$$f(\mathbf{x}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^\top \mathbf{H}f(\mathbf{x}_0) (\mathbf{x} - \mathbf{x}_0). \quad (20.4.30)$$

Cách này hoạt động cho bất cứ đầu vào thứ nguyên nào và cung cấp gần đúng nhất hàm bậc hai cho một hàm bất kỳ tại một điểm. Lấy biểu đồ của hàm sau làm ví dụ.

$$f(x, y) = xe^{-x^2-y^2}. \quad (20.4.31)$$

Có thể tính toán gradient và Hessian như sau:

$$\nabla f(x, y) = e^{-x^2-y^2} \begin{pmatrix} 1 - 2x^2 \\ -2xy \end{pmatrix} \text{ and } \mathbf{H}f(x, y) = e^{-x^2-y^2} \begin{pmatrix} 4x^3 - 6x & 4x^2y - 2y \\ 4x^2y - 2y & 4xy^2 - 2x \end{pmatrix}. \quad (20.4.32)$$

Kết hợp một chút đại số, ta thấy rằng hàm bậc hai xấp xỉ tại  $[-1, 0]^\top$  là:

$$f(x, y) \approx e^{-1} (-1 - (x+1) + (x+1)^2 + y^2). \quad (20.4.33)$$

```

# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101),
                    np.linspace(-2, 2, 101), indexing='ij')
z = x*np.exp(-x**2 - y**2)

# Compute approximating quadratic with gradient and Hessian at (1, 0)
w = np.exp(-1)*(-1 - (x + 1) + (x + 1)**2 + y**2)

# Plot function
ax = d2l.plt.figure().add_subplot(111, projection='3d')
ax.plot_wireframe(x, y, z, **{'rstride': 10, 'cstride': 10})
ax.plot_wireframe(x, y, w, **{'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

```

Điều này tạo cơ sở cho Thuật toán Newton được thảo luận ở [Section 13.7](#), trong đó chúng ta lặp đi lặp lại việc tối ưu hoá để tìm ra hàm bậc hai phù hợp nhất và sau đó cực tiểu hoá hàm bậc hai đó.

#### 20.4.7 Giải tích Ma trận

Đạo hàm của các hàm có liên quan đến ma trận hoá ra rất đẹp. Phần này sẽ nặng về mặt ký hiệu, vì vậy độc giả có thể bỏ qua trong lần đọc đầu tiên. Tuy nhiên sẽ rất hữu ích khi biết rằng đạo hàm của các hàm liên quan đến các phép toán ma trận thường gọn gàng hơn nhiều so với suy nghĩ ban đầu của chúng ta, đặc biệt là bởi sự quan trọng của các phép tính ma trận trong các ứng dụng học sâu.

Hãy xem một ví dụ. Giả sử chúng ta có một vài vector cột cố định  $\beta$ , và chúng ta muốn lấy hàm tích  $f(\mathbf{x}) = \beta^\top \mathbf{x}$ , và hiểu cách tích vô hướng thay đổi khi chúng ta thay đổi  $\mathbf{x}$ .

Ký hiệu có tên *ma trận đạo hàm sắp xếp theo mẫu số - denominator layout matrix derivative* sẽ hữu ích khi làm việc với ma trận đạo hàm trong học máy, trong đó chúng ta tập hợp các đạo hàm riêng theo mẫu số của vi phân, biểu diễn thành các dạng vector, ma trận hoặc tensor. Trong trường hợp này, chúng ta viết:

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix}, \quad (20.4.34)$$

mà ở đây nó khớp với hình dạng của vector cột  $\mathbf{x}$ .

Triển khai hàm của chúng ta thành các thành tố

$$f(\mathbf{x}) = \sum_{i=1}^n \beta_i x_i = \beta_1 x_1 + \cdots + \beta_n x_n. \quad (20.4.35)$$

Nếu bây giờ ta tính đạo hàm riêng theo  $\beta_1$  chẳng hạn, để ý rằng tất cả các phần tử bằng không

ngoại trừ số hạng đầu tiên là  $x_1$  nhân với  $\beta_1$ . Vì thế, ta có

$$\frac{df}{dx_1} = \beta_1, \quad (20.4.36)$$

hoặc tổng quát hơn đó là

$$\frac{df}{dx_i} = \beta_i. \quad (20.4.37)$$

Bây giờ ta có thể gộp chúng lại thành một ma trận như sau

$$\frac{df}{d\mathbf{x}} = \begin{bmatrix} \frac{df}{dx_1} \\ \vdots \\ \frac{df}{dx_n} \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} = \boldsymbol{\beta}. \quad (20.4.38)$$

Biểu thức trên minh họa một vài yếu tố về giải tích ma trận mà ta sẽ gặp trong suốt phần này:

- Đầu tiên, các tính toán sẽ trở nên khá phức tạp.
- Thứ hai, kết quả cuối cùng sẽ gọn gàng hơn quá trình tính toán trung gian, và sẽ luôn có bề ngoài giống với trường hợp đơn biến. Trong trường hợp này, hãy lưu ý rằng  $\frac{d}{dx}(bx) = b$  và  $\frac{d}{d\mathbf{x}}(\boldsymbol{\beta}^\top \mathbf{x}) = \boldsymbol{\beta}$  là như nhau.
- Thứ ba, các chuyển vị có thể xuất hiện mà thoát nhìn không biết chính xác từ đâu ra. Lý do chủ yếu là do ta quy ước đạo hàm sẽ có cùng kích thước với mẫu số, do đó khi nhân ma trận, ta cần lấy chuyển vị tương ứng để khớp với kích thước ban đầu.

Ta hãy thử một phép tính khó hơn làm ví dụ minh họa trực quan. Giả sử ta có một vector cột  $\mathbf{x}$  và một ma trận vuông  $A$ , và ta muốn tính biểu thức sau:

$$\frac{d}{d\mathbf{x}}(\mathbf{x}^\top A\mathbf{x}). \quad (20.4.39)$$

Để thuận tiện cho việc ký hiệu, ta hãy viết lại bài toán bằng ký hiệu Einstein.

$$\mathbf{x}^\top A\mathbf{x} = x_i a_{ij} x_j. \quad (20.4.40)$$

Để tính đạo hàm, ta cần tính các giá trị sau với từng giá trị của biến  $k$ :

$$\frac{d}{dx_k}(\mathbf{x}^\top A\mathbf{x}) = \frac{d}{dx_k} x_i a_{ij} x_j. \quad (20.4.41)$$

Theo quy tắc nhân, ta có

$$\frac{d}{dx_k} x_i a_{ij} x_j = \frac{dx_i}{dx_k} a_{ij} x_j + x_i a_{ij} \frac{dx_j}{dx_k}. \quad (20.4.42)$$

Với số hạng như  $\frac{dx_i}{dx_k}$ , không khó để thấy rằng đạo hàm trên có giá trị bằng 1 khi  $i = k$ , ngược lại nó sẽ bằng 0. Điều này có nghĩa là mọi số hạng với  $i$  và  $k$  khác nhau sẽ biến mất khỏi tổng trên, vì thế các số hạng duy nhất còn lại trong tổng đầu tiên đó là những số hạng với  $i = k$ . Lập luận tương tự cũng áp dụng cho số hạng thứ hai khi ta cần  $j = k$ . Từ đó, ta có

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{kj} x_j + x_i a_{ik}. \quad (20.4.43)$$

Hiện tại, tên của các chỉ số trong ký hiệu Einstein là tùy ý - việc  $i$  và  $j$  khác nhau không quan trọng cho tính toán tại thời điểm này, vì thế ta có thể gán lại chỉ số sao cho cả hai đều chứa  $i$

$$\frac{d}{dx_k} x_i a_{ij} x_j = a_{ki} x_i + x_i a_{ik} = (a_{ki} + a_{ik}) x_i. \quad (20.4.44)$$

Bây giờ, ta cần luyện tập một chút để có thể đi sâu hơn. Ta hãy thử xác định kết quả trên theo các phép toán ma trận.  $a_{ki} + a_{ik}$  là phần tử thứ  $k, i$  của  $\mathbf{A} + \mathbf{A}^\top$ . Từ đó, ta có

$$\frac{d}{dx_k} x_i a_{ij} x_j = [\mathbf{A} + \mathbf{A}^\top]_{ki} x_i. \quad (20.4.45)$$

Tương tự, hạng tử này là tích của ma trận  $\mathbf{A} + \mathbf{A}^\top$  với vector  $\mathbf{x}$ , nên ta có

$$\left[ \frac{d}{d\mathbf{x}} (\mathbf{x}^\top \mathbf{A} \mathbf{x}) \right]_k = \frac{d}{dx_k} x_i a_{ij} x_j = [(\mathbf{A} + \mathbf{A}^\top) \mathbf{x}]_k. \quad (20.4.46)$$

Ta thấy phần tử thứ  $k$  của đạo hàm mong muốn từ (20.4.39) đơn giản là phần tử thứ  $k$  của vector bên vế phải, và do đó hai phần tử này là như nhau. Điều này dẫn đến

$$\frac{d}{d\mathbf{x}} (\mathbf{x}^\top \mathbf{A} \mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top) \mathbf{x}. \quad (20.4.47)$$

Biểu thức trên cần nhiều biến đổi để suy ra được hơn ở phần trước, nhưng kết quả cuối cùng vẫn sẽ gọn gàng. Hơn thế nữa, hãy xem xét tính toán dưới đây cho đạo hàm đơn biến thông thường:

$$\frac{d}{dx} (xax) = \frac{dx}{dx} ax + xa \frac{dx}{dx} = (a + a)x. \quad (20.4.48)$$

Tương tự,  $\frac{d}{dx} (ax^2) = 2ax = (a + a)x$ . Một lần nữa, ta lại thu được kết quả nhìn giống với trường hợp đơn biến nhưng với một phép chuyển vị.

Tại thời điểm này, cách tính trên có vẻ khá đáng ngờ, vì vậy ta hãy thử tìm hiểu lý do tại sao. Khi ta lấy đạo hàm ma trận như trên, đầu tiên ta giả sử biểu thức ta nhận được sẽ là một biểu thức ma trận khác: một biểu thức mà ta có thể viết nó dưới dạng tích và tổng của các ma trận và chuyển vị của chúng. Nếu một biểu thức như vậy tồn tại, nó sẽ phải đúng cho tất cả các ma trận. Do đó, nó sẽ đúng với ma trận  $1 \times 1$ , trong đó tích ma trận chỉ là tích của các số, tổng ma trận chỉ là tổng, và phép chuyển vị không có tác dụng gì! Nói cách khác, bất kỳ biểu thức nào chúng ta nhận được phải phù hợp với biểu thức đơn biến. Điều này có nghĩa là khi ta biết đạo hàm đơn biến tương ứng, với một chút luyện tập ta có thể đoán được các đạo hàm ma trận!

Cùng kiểm nghiệm điều này. Giả sử  $\mathbf{X}$  là ma trận  $n \times m$ ,  $\mathbf{U}$  là ma trận  $n \times r$  và  $\mathbf{V}$  là ma trận  $r \times m$ . Ta sẽ tính

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = ? \quad (20.4.49)$$

Phép tính này khá quan trọng trong phân rã ma trận. Tuy nhiên, ở đây nó chỉ đơn giản là một đạo hàm mà ta cần tính. Hãy thử tưởng tượng xem nó sẽ như thế nào đối với ma trận  $1 \times 1$ . Trong trường hợp này, ta có biểu thức sau

$$\frac{d}{dv} (x - uv)^2 = -2(x - uv)u, \quad (20.4.50)$$

Có thể thấy, đây là một đạo hàm khá phổ thông. Nếu ta thử chuyển đổi nó thành một biểu thức ma trận, ta có

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2(\mathbf{X} - \mathbf{UV})\mathbf{U}. \quad (20.4.51)$$

Tuy nhiên, nếu ta nhìn kỹ, điều này không hoàn toàn đúng. Hãy nhớ lại  $\mathbf{X}$  có kích thước  $n \times m$ , giống  $\mathbf{UV}$ , nên ma trận  $2(\mathbf{X} - \mathbf{UV})$  có kích thước  $n \times m$ . Mặt khác  $\mathbf{U}$  có kích thước  $n \times r$ , và ta không thể nhân một ma trận  $n \times m$  với một ma trận  $n \times r$  vì số chiều của chúng không khớp nhau!

Ta muốn nhận  $\frac{d}{d\mathbf{V}}$ , cùng kích thước với  $\mathbf{V}$  là  $r \times m$ . Vì vậy ta bằng cách nào đó cần phải nhân một ma trận  $n \times m$  với một ma trận  $n \times r$  (có thể phải chuyển vị) để có ma trận  $r \times m$ . Ta có thể làm điều này bằng cách nhân  $\mathbf{U}^\top$  với  $(\mathbf{X} - \mathbf{UV})$ . Vì vậy, ta có thể đoán nghiệm cho (20.4.49) là

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2\mathbf{U}^\top(\mathbf{X} - \mathbf{UV}). \quad (20.4.52)$$

Để chứng minh rằng điều này là đúng, ta cần một tính toán chi tiết. Nếu bạn tin rằng quy tắc trực quan ở trên là đúng, bạn có thể bỏ qua phần trình bày này. Để tính toán

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{UV}\|_2^2, \quad (20.4.53)$$

với mỗi  $a$  và  $b$ , ta phải tính.

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = \frac{d}{dv_{ab}} \sum_{i,j} \left( x_{ij} - \sum_k u_{ik} v_{kj} \right)^2. \quad (20.4.54)$$

Hãy nhớ lại rằng tất cả các phần tử của  $\mathbf{X}$  và  $\mathbf{U}$  là hằng số khi tính  $\frac{d}{dv_{ab}}$ , chúng ta có thể đẩy đạo hàm bên trong tổng, và áp dụng quy tắc dây chuyền sau đó bình phương lên để có

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = \sum_{i,j} 2 \left( x_{ij} - \sum_k u_{ik} v_{kj} \right) \left( - \sum_k u_{ik} \frac{dv_{kj}}{dv_{ab}} \right). \quad (20.4.55)$$

Tương tự phần diễn giải trước, ta có thể để ý rằng  $\frac{dv_{kj}}{dv_{ab}}$  chỉ khác không nếu  $k = a$  và  $j = b$ . Nếu một trong hai điều kiện đó không thỏa, số hạng trong tổng bằng không, ta có thể tự do loại bỏ nó. Ta thấy rằng

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2 \sum_i \left( x_{ib} - \sum_k u_{ik} v_{kb} \right) u_{ia}. \quad (20.4.56)$$

Một điểm tinh tế quan trọng ở đây là yêu cầu về  $k = a$  không xảy ra bên trong tổng phía trong bởi  $k$  chỉ là một biến tùy ý để tính tổng các số hạng trong tổng phía trong. Một ví dụ dễ hiểu hơn:

$$\frac{d}{dx_1} \left( \sum_i x_i \right)^2 = 2 \left( \sum_i x_i \right). \quad (20.4.57)$$

Từ đây, ta có thể bắt đầu xác định các thành phần của tổng. Đầu tiên,

$$\sum_k u_{ik} v_{kb} = [\mathbf{UV}]_{ib}. \quad (20.4.58)$$

Cho nên toàn bộ biểu thức bên trong tổng là

$$x_{ib} - \sum_k u_{ik} v_{kb} = [\mathbf{X} - \mathbf{UV}]_{ib}. \quad (20.4.59)$$

Điều này nghĩa là giờ đây đạo hàm của ta có thể viết dưới dạng

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2 \sum_i [\mathbf{X} - \mathbf{UV}]_{ib} u_{ia}. \quad (20.4.60)$$

Chúng ta có thể muốn nó trông giống như phần tử  $a, b$  của một ma trận để có thể sử dụng các kỹ thuật trong các ví dụ trước đó nhằm đạt được một biểu thức ma trận, nghĩa là ta cần phải hoán đổi thứ tự của các chỉ số trên  $u_{ia}$ . Nếu để ý  $u_{ia} = [\mathbf{U}^\top]_{ai}$ , ta có thể viết

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2 \sum_i [\mathbf{U}^\top]_{ai} [\mathbf{X} - \mathbf{UV}]_{ib}. \quad (20.4.61)$$

Đây là tích một ma trận, vì thế ta có thể kết luận

$$\frac{d}{dv_{ab}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2 [\mathbf{U}^\top (\mathbf{X} - \mathbf{UV})]_{ab}. \quad (20.4.62)$$

và vì vậy ta có lời giải cho (20.4.49)

$$\frac{d}{d\mathbf{V}} \|\mathbf{X} - \mathbf{UV}\|_2^2 = -2 \mathbf{U}^\top (\mathbf{X} - \mathbf{UV}). \quad (20.4.63)$$

Lời giải này trùng với biểu thức mà ta đoán ở phía trên!

Lúc này cũng dễ hiểu nếu ta tự hỏi “Tại sao không viết tất cả các quy tắc giải tích đã từng học thành dạng ma trận? Điều này rõ ràng là công việc máy móc. Tại sao ta không đơn giản là làm hết một lần cho xong?” Và thực sự có những quy tắc như thế, (Petersen et al., 2008) cho ta một bản tóm tắt tuyệt vời. Tuy nhiên, vì số cách kết hợp các phép toán ma trận nhiều hơn hẳn so với các giá trị một biến, nên có nhiều quy tắc đạo hàm ma trận hơn các quy tắc dành cho hàm cho một biến. Thông thường, tốt nhất là làm việc với các chỉ số, hoặc dùng vi phân tự động khi thích hợp.

#### 20.4.8 Tóm tắt

- Với không gian nhiều chiều, chúng ta có thể định nghĩa gradient cùng mục đích như các đạo hàm một chiều. Điều này cho phép ta thấy cách một hàm đa biến thay đổi như thế nào khi có bất kỳ thay đổi nhỏ xảy ra ở đầu vào.
- Thuật toán lan truyền ngược có thể được xem như một phương pháp trong việc tổ chức quy tắc dây chuyền đa biến cho phép tính toán hiệu quả các đạo hàm riêng.
- Giải tích ma trận cho phép chúng ta viết các đạo hàm của biểu thức ma trận một cách gọn gàng hơn.

#### 20.4.9 Bài tập

1. Cho một vector cột  $\beta$ , tính các đạo hàm của cả hai ma trận  $f(\mathbf{x}) = \beta^\top \mathbf{x}$  và ma trận  $g(\mathbf{x}) = \mathbf{x}^\top \beta$ . Hãy cho biết tại sao bạn lại ra cùng đáp án?
2. Cho  $\mathbf{v}$  là một vector  $n$  chiều. Vậy  $\frac{\partial}{\partial \mathbf{v}} \|\mathbf{v}\|_2$  là gì?
3. Cho  $L(x, y) = \log(e^x + e^y)$ . Tính toán gradient. Tổng của các thành phần của gradient là gì?
4. Cho  $f(x, y) = x^2y + xy^2$ . Chúng minh rằng điểm tới hạn duy nhất là  $(0, 0)$ . Bằng việc xem xét  $f(x, x)$ , hãy xác định xem  $(0, 0)$  là cực đại, cực tiểu, hay không phải cả hai.
5. Giả sử ta đang tối thiểu hàm  $f(\mathbf{x}) = g(\mathbf{x}) + h(\mathbf{x})$ . Làm cách nào ta có thể diễn giải bằng hình học điều kiện  $\nabla f = 0$  thông qua  $g$  và  $h$ ?

#### 20.4.10 Thảo luận

- Tiếng Anh: MXNet<sup>415</sup>, Pytorch<sup>416</sup>, Tensorflow<sup>417</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>418</sup>

#### 20.4.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Nguyễn Lê Quang Nhật
- Nguyễn Văn Quang
- Nguyễn Thanh Hòa
- Nguyễn Văn Cường
- Trần Yến Thy
- Nguyễn Mai Hoàng Long

<sup>415</sup> <https://discuss.d2l.ai/t/413>

<sup>416</sup> <https://discuss.d2l.ai/t/1090>

<sup>417</sup> <https://discuss.d2l.ai/t/1091>

<sup>418</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.5 Giải tích Tích phân

Phép vi phân mới chỉ là một nửa nội dung của môn giải tích truyền thống. Một nửa quan trọng khác, phép tích phân, bắt nguồn từ một câu hỏi có vẻ không mấy liên quan, “Diện tích của phần bên dưới đường cong này là bao nhiêu?” Dù vậy, phép tích phân lại liên hệ mật thiết tới phép vi phân thông qua *định lý cơ bản của giải tích* (*fundamental theorem of calculus*).

Ở mức độ kiến thức học máy ta thảo luận trong cuốn sách này, ta không cần thiết phải hiểu sâu sắc về phép tích phân. Tuy nhiên, chúng tôi sẽ giới thiệu khái quát để đặt nền tảng cho bất kỳ ứng dụng nào ta sẽ gặp sau này.

### 20.5.1 Diện tích Hình học

Giả sử ta có hàm  $f(x)$ . Để đơn giản, giả sử  $f(x)$  không âm (không cho ra số bé hơn không). Điều chúng ta muốn tìm hiểu là: diện tích của phần được giới hạn giữa  $f(x)$  và trục  $x$  là bao nhiêu?

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mpl_toolkits import mplot3d
from mxnet import np, npx
npx.set_np()

x = np.arange(-2, 2, 0.01)
f = np.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 đa số trường hợp, diện tích của vùng này sẽ là vô cực hoặc không xác định (ví dụ như hàm  $f(x) = x^2$ ), nên ta thường nói về diện tích trong một khoảng giữa hai cận, ví dụ  $a$  và  $b$ .

```
x = np.arange(-2, 2, 0.01)
f = np.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()
```

Ta sẽ ký hiệu phần diện tích này với dấu tích phân như dưới đây:

$$\text{Area}(\mathcal{A}) = \int_a^b f(x) dx. \quad (20.5.1)$$

Ta có thể sử dụng bất kì ký hiệu nào vì biến tích phân bên trong là tùy ý, cũng giống như biến chỉ số của phép tổng trong  $\sum$ :

$$\int_a^b f(x) dx = \int_a^b f(z) dz. \quad (20.5.2)$$

Có một cách truyền thống để hiểu rõ hơn cách ta có thể xấp xỉ phép tích phân: ta có thể tưởng tượng cắt phần giữa  $a$  và  $b$  thành  $N$  lát theo chiều dọc. Nếu  $N$  lớn, ta có thể xấp xỉ phần diện tích mỗi lát bằng một hình chữ nhật, và sau đó tính tổng các diện tích để có được phần diện tích phía dưới đường cong. Hãy cùng lập trình và xét qua một ví dụ. Ta sẽ biết cách tính được giá trị thật sự của phép tích phân ở mục sau.

```
epsilon = 0.05
a = 0
b = 2

x = np.arange(a, b, epsilon)
f = x / (1 + x**2)

approx = np.sum(epsilon*f)
true = np.log(2) / 2

d2l.set_figsize()
d2l.plt.bar(x.astype(np.float), 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}'
```

Một vấn đề đó là trong khi cách này có thể làm một cách số học, ta chỉ có thể sử dụng cách tiếp cận phân tích này cho các hàm cực đơn giản như

$$\int_a^b x \, dx. \quad (20.5.3)$$

Bất kì hàm số nào hơi phức tạp hơn một chút như trong ví dụ trình bày ở đoạn mã trên

$$\int_a^b \frac{x}{1+x^2} \, dx. \quad (20.5.4)$$

năm ngoài phạm vi ta có thể giải quyết bằng phương pháp này.

Thay vào đó, ta sẽ tiếp cận theo hướng khác. Ta sẽ làm việc một cách trực quan với khái niệm diện tích, và học công cụ tính toán chính được dùng để tính tích phân: *định lý cơ bản của giải tích*. Đây sẽ là nền tảng của ta trong quá trình học tích phân.

## 20.5.2 Định lý Cơ bản của Giải tích

Để đào sâu hơn nữa vào lý thuyết tích phân, chúng tôi xin phép giới thiệu hàm

$$F(x) = \int_0^x f(y) \, dy. \quad (20.5.5)$$

Hàm này tính diện tích giữa 0 và  $x$  tùy thuộc vào việc  $x$  thay đổi như thế nào. Để ý rằng đây là tất cả những gì ta cần, bởi vì

$$\int_a^b f(x) \, dx = F(b) - F(a). \quad (20.5.6)$$

Đây là một ký hiệu toán học biểu diễn diện tích khoảng giữa hai cận bằng hiệu diện tích của khoảng có cận xa hơn trừ đi diện tích của khoảng có cận gần hơn như trong Fig. 20.5.1.

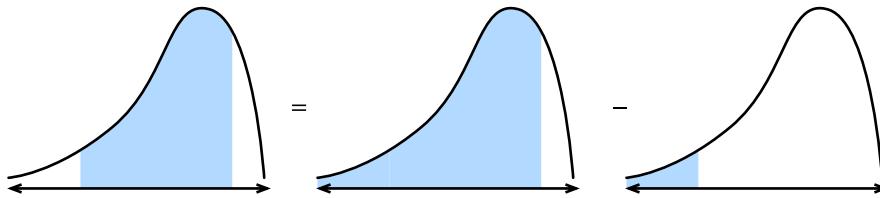


Fig. 20.5.1: Minh họa tại sao ta có thể đơn giản bài toán tính diện tích dưới đường cong giữa hai điểm thành bài toán phần diện tích phía bên trái của một điểm.

Vì thế, ta có thể tính tích phân trong khoảng bất kỳ bằng việc tìm  $F(x)$ .

Để làm điều này, hãy xem xét một thí nghiệm. Như thường làm trong giải tích, hãy tưởng tượng điều gì sẽ xảy ra khi ta dịch chuyển  $x$  một khoảng nhỏ. Bằng nhận xét phía trên, ta có

$$F(x + \epsilon) - F(x) = \int_x^{x+\epsilon} f(y) dy. \quad (20.5.7)$$

Điều này cho ta biết hàm số  $F(x)$  thay đổi bằng với diện tích phía dưới một đoạn cực nhỏ của hàm  $f(y)$ .

Đây là lúc mà ta cần xấp xỉ. Nếu nhìn vào phần diện tích nhỏ đó, ta thấy phần diện tích này gần với diện tích của một hình chữ nhật với chiều cao là giá trị tại  $f(x)$  và chiều rộng là  $\epsilon$ . Thực vậy, khi  $\epsilon \rightarrow 0$  phép xấp xỉ này càng chính xác. Vì thế, ta có thể kết luận

$$F(x + \epsilon) - F(x) \approx \epsilon f(x). \quad (20.5.8)$$

Tuy nhiên, ta có thể để ý rằng: phương trình này có dạng giống với khi ta tính đạo hàm của  $F$ ! Vì thế ta có được một sự thật khá bất ngờ rằng:

$$\frac{dF}{dx}(x) = f(x). \quad (20.5.9)$$

Đây chính là *định lý cơ bản của giải tích*. Ta có thể viết nó dưới dạng mở rộng là

$$\frac{d}{dx} \int_{-\infty}^x f(y) dy = f(x). \quad (20.5.10)$$

Nó lấy ý tưởng về tìm kiếm diện tích (một *tiên nghiệm* khá khó), và giảm tải thành một mệnh đề đạo hàm (một khái niệm toán học đã được nghiên cứu sâu). Một lưu ý cuối là định lý này không cho ta biết dạng thực sự của  $F(x)$ . Thực chất,  $F(x) + C$  với  $C$  bất kỳ đều có đạo hàm như nhau. Đây là sự thật chúng ta chấp nhận trong lý thuyết tích phân. Đáng mừng là khi làm việc với tích phân hữu hạn, phần hằng số bị loại bỏ, và vì thế không ảnh hưởng đến kết quả.

$$\int_a^b f(x) dx = (F(b) + C) - (F(a) + C) = F(b) - F(a). \quad (20.5.11)$$

Công thức trên có thể khá trừu tượng và vô nghĩa, nhưng nó cho ta một góc nhìn mới trong việc tính tích phân. Cách tiếp cận của ta không còn là phải “cắt ra và cộng lại” để tính diện tích, mà chỉ cần tìm một hàm số có đạo hàm là hàm hiện có! Đây là một điều tuyệt vời vì giờ ta có thể liệt kê

hàng loạt các phép tích phân phức tạp chỉ bằng cách đảo ngược lại bằng trong Section 20.3.2. Ví dụ, ta biết đạo hàm của  $x^n$  là  $nx^{n-1}$ . Vì thế, bằng cách sử dụng định lý cơ bản (20.5.10), ta có:

$$\int_0^x ny^{n-1} dy = x^n - 0^n = x^n. \quad (20.5.12)$$

Tương tự, ta biết đạo hàm của  $e^x$  là chính nó, nên:

$$\int_0^x e^y dy = e^x - e^0 = e^x - 1. \quad (20.5.13)$$

Bằng cách này, ta có thể phát triển toàn bộ lý thuyết tích phân bằng cách tự do tận dụng những ý tưởng từ giải tích vi phân. Mỗi quy tắc tích phân đều bắt nguồn từ đây.

### 20.5.3 Quy tắc Đổi biến

Cũng như vi phân, có một số quy tắc giúp việc tính tích phân dễ xử lý hơn. Thật ra, mọi quy tắc trong giải tích vi phân (như quy tắc tích, quy tắc tổng, và quy tắc dây chuyền) đều có một quy luật tương ứng cho giải tích tích phân (lần lượt là tích phân từng phần, tích phân của tổng, và quy tắc đổi biến số). Trong mục này, ta sẽ tìm hiểu quy tắc được cho là quan trọng nhất trong danh sách trên: quy tắc đổi biến số.

Đầu tiên, giả sử ta có một hàm tích phân:

$$F(x) = \int_0^x f(y) dy. \quad (20.5.14)$$

Giả sử ta muốn biết hàm này trông như thế nào khi kết hợp nó với một hàm nữa để có được  $F(u(x))$ . Bằng quy tắc dây chuyền, ta có

$$\frac{d}{dx} F(u(x)) = \frac{dF}{du}(u(x)) \cdot \frac{du}{dx}. \quad (20.5.15)$$

Ta có thể biến nó thành một mệnh đề tích phân bằng cách sử dụng định lý cơ bản (20.5.10) ở trên, để có

$$F(u(x)) - F(u(0)) = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} dy. \quad (20.5.16)$$

Biết rằng  $F$  chính nó là một tích phân giúp vẽ bên trái có thể được viết lại là

$$\int_{u(0)}^{u(x)} f(y) dy = \int_0^x \frac{dF}{du}(u(y)) \cdot \frac{du}{dy} dy. \quad (20.5.17)$$

Tương tự,  $F$  là một tích phân nên theo định lý cơ bản (20.5.10),  $\frac{dF}{dx} = f$ , do đó:

$$\int_{u(0)}^{u(x)} f(y) dy = \int_0^x f(u(y)) \cdot \frac{du}{dy} dy. \quad (20.5.18)$$

Đây là quy tắc *đổi biến số*.

Để có một chứng minh trực quan hơn, xét chuyện gì sẽ xảy ra khi ta lấy tích phân hàm  $f(u(x))$  giữa  $x$  và  $x + \epsilon$ . Với  $\epsilon$  nhỏ, tích phân này xấp xỉ  $\epsilon f(u(x))$ , phần diện tích của hình chữ nhật tương ứng. Nay so sánh với tích phân của  $f(y)$  từ  $u(x)$  tới  $u(x + \epsilon)$ . Ta biết rằng  $u(x + \epsilon) \approx u(x) + \epsilon \frac{du}{dx}(x)$ , vậy nên phần diện tích của hình chữ nhật này xấp xỉ  $\epsilon \frac{du}{dx}(x) f(u(x))$ . Do đó, để đồng hóa diện tích hai hình chữ nhật này, ta phải nhân phần thứ nhất với  $\frac{du}{dx}(x)$  như minh họa trong Fig. 20.5.2.

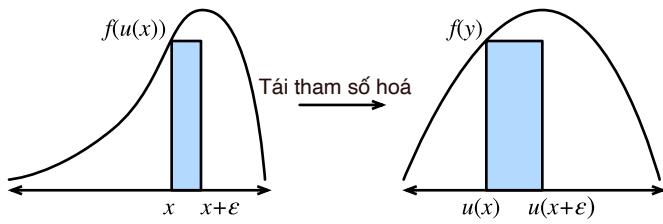


Fig. 20.5.2: Minh họa cho biến đổi của một hình chữ nhật mỏng dưới sự thay đổi biến số.

Điều này cho ta biết

$$\int_x^{x+\epsilon} f(u(y)) \frac{du}{dy}(y) dy = \int_{u(x)}^{u(x+\epsilon)} f(y) dy. \quad (20.5.19)$$

Đây là quy tắc đổi biến số được biểu diễn cho một hình chữ nhật nhỏ.

Chọn  $u(x)$  và  $f(x)$  một cách thích hợp sẽ cho phép tính những tích phân cực kỳ phức tạp. Ví dụ, nếu ta chọn  $f(y) = 1$  và  $u(x) = e^{-x^2}$  (có nghĩa là  $\frac{du}{dx}(x) = -2xe^{-x^2}$ ):

$$e^{-1} - 1 = \int_{e^{-0}}^{e^{-1}} 1 dy = -2 \int_0^1 ye^{-y^2} dy, \quad (20.5.20)$$

và khi sắp xếp lại ta có

$$\int_0^1 ye^{-y^2} dy = \frac{1 - e^{-1}}{2}. \quad (20.5.21)$$

#### 20.5.4 Nhận xét về Quy ước Ký hiệu

Những độc giả tinh mắt sẽ nhận thấy điều kì lạ từ những phép tính trên. Ví dụ như

$$\int_{e^{-0}}^{e^{-1}} 1 dy = e^{-1} - 1 < 0, \quad (20.5.22)$$

có thể âm. Khi nghĩ về các diện tích, có thể là hơi lạ khi ta thấy một giá trị âm, vậy nên nó đáng để ta tìm hiểu kĩ hơn về các quy ước này.

Các nhà toán học có khái niệm về diện tích có dấu. Điều này thể hiện theo hai cách. Đầu tiên, với  $f(x)$  nhỏ hơn 0, thì diện tích cũng sẽ âm. Ví dụ:

$$\int_0^1 (-1) dx = -1. \quad (20.5.23)$$

Thứ hai, tích phân tiến từ phải sang trái, thay vì từ trái sang phải, cũng có diện tích âm

$$\int_0^{-1} 1 \, dx = -1. \quad (20.5.24)$$

Diện tích chuẩn (từ trái sang phải của một hàm số dương) thì luôn dương. Bất kỳ kết quả nào thu được bằng cách lật hàm này (giả sử lật đối xứng qua trục  $x$  để lấy tích phân của một hàm âm, hoặc lật đối xứng qua trục  $y$  để lấy tích phân ngược thứ tự) sẽ tạo ra diện tích âm. Và việc lật hai lần sẽ làm cặp dấu âm triệt tiêu nhau, từ đó lại có diện tích dương.

$$\int_0^{-1} (-1) \, dx = 1. \quad (20.5.25)$$

Thực tế, trong: numref:sec\_geometry-linear-algebraic-ops, ta đã bàn về cách định thức biểu diễn diện tích có dấu theo cách tương tự.

### 20.5.5 Tích phân bội

Trong một số trường hợp, ta sẽ cần phải làm việc với số lượng chiều lớn hơn. Ví dụ: giả sử ta có một hàm hai biến  $f(x, y)$  và muốn biết thể tích phia dưới  $f$  khi  $x$  nằm trong đoạn  $[a, b]$  và  $y$  trong đoạn  $[c, d]$ .

```
# Construct grid and compute function
x, y = np.meshgrid(np.linspace(-2, 2, 101), np.linspace(-2, 2, 101),
                    indexing='ij')
z = np.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
```

Ta có thể viết thành tích phân sau:

$$\int_{[a,b] \times [c,d]} f(x, y) \, dx \, dy. \quad (20.5.26)$$

Giả sử ta muốn tính tích phân này. Ta có thể tính tích phân theo cận  $x$  trước rồi chuyển sang cận  $y$ , nghĩa là:

$$\int_{[a,b] \times [c,d]} f(x, y) \, dx \, dy = \int_c^d \left( \int_a^b f(x, y) \, dx \right) \, dy. \quad (20.5.27)$$

Hãy cùng quan sát tại sao.

Hãy xem xét hình trên, nơi ta đã phân chia hàm thành  $\epsilon \times \epsilon$  ô vuông với chỉ số nguyên  $i, j$ . Trong trường hợp này, tích phân xấp xỉ:

$$\sum_{i,j} \epsilon^2 f(\epsilon i, \epsilon j). \quad (20.5.28)$$

Một khi đã rời rạc hóa bài toán, ta có thể cộng dồn các giá trị trên các ô vuông này theo bất kỳ thứ tự nào mà không phải lo lắng về việc giá trị cuối cùng thay đổi. Điều này được minh họa trong Fig. 20.5.3. Đặc biệt, có thể nói rằng

$$\sum_j \epsilon \left( \sum_i \epsilon f(\epsilon i, \epsilon j) \right). \quad (20.5.29)$$

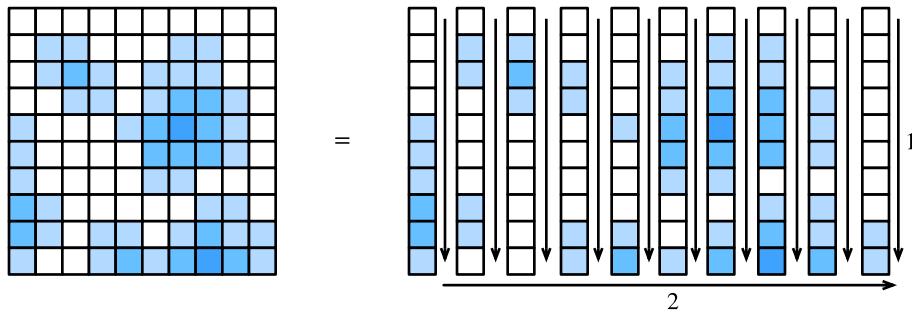


Fig. 20.5.3: Minh họa cách phân rã một tổng trên nhiều ô vuông dưới dạng tổng trên các cột (1), sau đó cộng các tổng của cột với nhau (2).

Tổng bên trong chính xác là phiên bản rời rạc của tích phân:

$$G(\epsilon j) = \int_a^b f(x, \epsilon j) dx. \quad (20.5.30)$$

Sau cùng, nếu kết hợp hai biểu thức này với nhau, ta có:

$$\sum_j \epsilon G(\epsilon j) \approx \int_c^d G(y) dy = \int_{[a,b] \times [c,d]} f(x, y) dx dy. \quad (20.5.31)$$

Kết hợp tất cả lại:

$$\int_{[a,b] \times [c,d]} f(x, y) dx dy = \int_c^d \left( \int_a^b f(x, y) dx \right) dy. \quad (20.5.32)$$

Lưu ý rằng, một khi đã rời rạc hóa, tất cả những gì ta làm là sắp xếp lại thứ tự tính tổng một danh sách các số. Điều này có vẻ hiển nhiên, tuy nhiên kết quả này (được gọi là *Định lý Fubini*) không phải lúc nào cũng đúng! Đối với loại toán học gấp phải khi thực hiện tác vụ học máy (các hàm liên tục), không có gì đáng lo ngại, tuy vậy ta có thể tạo các ví dụ mà cách này không áp dụng được (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 việc chọn tính tích phân theo cận  $x$  trước, cận  $y$  sau là tùy ý. Ta cũng có thể chọn thực hiện  $y$  trước,  $x$  sau:

$$\int_{[a,b] \times [c,d]} f(x, y) dx dy = \int_a^b \left( \int_c^d f(x, y) dy \right) dx. \quad (20.5.33)$$

Thông thường, ta sẽ rút gọn thành ký hiệu vector và nói rằng tích phân trên miền  $U = [a, b] \times [c, d]$  là:

$$\int_U f(\mathbf{x}) \, d\mathbf{x}. \quad (20.5.34)$$

### 20.5.6 Đổi biến trong Tích phân bội

Tương tự như tích phân đơn biến, việc đổi biến trong tích phân bội là một kỹ thuật quan trọng. Chúng tôi sẽ tổng hợp kết quả mà không chứng minh và trình bày.

Chúng ta cần một hàm để tái tham số hoá miền tích phân. Ta có thể coi hàm này là  $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , nhận  $n$  biến thực và trả về  $n$  giá trị thực khác. Để giữ cho các biểu thức rõ ràng, ta giả sử  $\phi$  là *đơn ánh* (*injective*) tức nó cho đầu ra khác nhau với đầu vào khác nhau ( $\phi(\mathbf{x}) = \phi(\mathbf{y}) \implies \mathbf{x} = \mathbf{y}$ ).

Trong trường hợp này, ta có:

$$\int_{\phi(U)} f(\mathbf{x}) \, d\mathbf{x} = \int_U f(\phi(\mathbf{x})) |\det(D\phi(\mathbf{x}))| \, d\mathbf{x}. \quad (20.5.35)$$

trong đó  $D\phi$  là *Jacobian* của  $\phi$ , ma trận của các đạo hàm riêng của  $\phi = (\phi_1(x_1, \dots, x_n), \dots, \phi_n(x_1, \dots, x_n))$ ,

$$D\phi = \begin{bmatrix} \frac{\partial \phi_1}{\partial x_1} & \dots & \frac{\partial \phi_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \phi_n}{\partial x_1} & \dots & \frac{\partial \phi_n}{\partial x_n} \end{bmatrix}. \quad (20.5.36)$$

Khi xét kĩ hơn, ta nhận thấy điều này tương tự như quy tắc dây chuyền đơn biến (20.5.18), ngoại trừ việc ta đã thay thế  $\frac{du}{dx}(x)$  bằng  $|\det(D\phi(\mathbf{x}))|$ . Hãy cùng xem ta có thể giải thích hạng tử này như thế nào. Hãy nhớ lại rằng  $\frac{du}{dx}(x)$  cho thấy việc áp dụng  $u$  kéo dãn trục  $x$  như thế nào. Tương tự, ở không gian nhiều chiều,  $|\det(D\phi(\mathbf{x}))|$  cho thấy áp dụng  $\phi$  sẽ kéo dãn diện tích (hoặc thể tích, siêu thể tích) của một hình vuông nhỏ (hoặc một *khối lập phương* nhỏ) như thế nào. Nếu  $\phi$  là một phép nhân với ma trận, thì ta đã biết định thức có ảnh hưởng như thế nào.

Đi sâu hơn một chút, ta có thể chỉ ra rằng *Jacobian* đưa ra xấp xỉ tốt nhất cho một hàm đa biến  $\phi$  tại một điểm bằng ma trận theo cách giống như khi xấp xỉ đạo hàm và gradient bằng các đường hoặc mặt phẳng. Do đó, định thức Jacobian tương ứng với hệ số tỷ lệ ta đã xác định trong không gian một chiều.

Việc này cần đi sâu vào một số chi tiết, vì vậy đừng lo lắng nếu bạn chưa hiểu ngay. Hãy cùng xem qua một ví dụ mà ta sẽ dùng sau này. Xét tích phân:

$$\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{-x^2-y^2} \, dx \, dy. \quad (20.5.37)$$

Rất khó tính trực tiếp tích phân này, nhưng nếu đổi biến, thì ta có thể đạt được tiến triển đáng kể. Nếu đặt  $\phi(r, \theta) = (r \cos(\theta), r \sin(\theta))$  (nghĩa là  $x = r \cos(\theta)$ ,  $y = r \sin(\theta)$ ), thì ta có thể áp dụng công thức đổi biến để có:

$$\int_0^{\infty} \int_0^{2\pi} e^{-r^2} |\det(D\phi(\mathbf{x}))| \, d\theta \, dr, \quad (20.5.38)$$

với:

$$|\det(D\phi(\mathbf{x}))| = \left| \det \begin{bmatrix} \cos(\theta) & -r \sin(\theta) \\ \sin(\theta) & r \cos(\theta) \end{bmatrix} \right| = r(\cos^2(\theta) + \sin^2(\theta)) = r. \quad (20.5.39)$$

Vì vậy, tích phân được viết lại là:

$$\int_0^\infty \int_0^{2\pi} r e^{-r^2} d\theta dr = 2\pi \int_0^\infty r e^{-r^2} dr = \pi, \quad (20.5.40)$$

trong đó đẳng thức cuối cùng là phép tính mà ta đã sử dụng trong `integration_example`.

Chúng ta sẽ lại gặp tích phân này khi chúng ta học về các biến ngẫu nhiên liên tục trong Section 20.6.

### 20.5.7 Tóm tắt

- Lý thuyết tích phân cho phép chúng ta giải đáp các câu hỏi về các diện tích hoặc thể tích.
- Định lý cơ bản của giải tích cho phép vận dụng kiến thức về đạo hàm để tính toán các diện tích thông qua quan sát rằng đạo hàm của diện tích tới một điểm nào đó được xác định bởi giá trị tại điểm đó của hàm đang được tích phân.
- Tích phân trong không gian nhiều chiều có thể được tính bằng cách lặp qua các tích phân đơn biến.

### 20.5.8 Bài tập

1. Tính  $\int_1^2 \frac{1}{x} dx$ .
2. Áp dụng công thức đổi biến để tính  $\int_0^{\sqrt{\pi}} x \sin(x^2) dx$ .
3. Tính  $\int_{[0,1]^2} xy dx dy$ .
4. Áp dụng công thức đổi biến để tính  $\int_0^2 \int_0^1 xy(x^2 - y^2)/(x^2 + y^2)^3 dy dx$  và  $\int_0^1 \int_0^2 f(x, y) = xy(x^2 - y^2)/(x^2 + y^2)^3 dx dy$  để thấy sự khác nhau giữa chúng.

### 20.5.9 Thảo luận

- Tiếng Anh: MXNet<sup>419</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>420</sup>

<sup>419</sup> <https://discuss.d2l.ai/t/414>

<sup>420</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 20.5.10 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Phạm Đăng Khoa
- Lê Khắc Hồng Phúc
- Nguyễn Văn Cường

## 20.6 Biến Ngẫu nhiên

Section 4.6 đã giới thiệu các phương pháp cơ bản để làm việc với biến ngẫu nhiên rời rạc, mà trong trường hợp của ta các biến ngẫu nhiên này có thể chỉ có một tập hữu hạn các giá trị khả dĩ, hoặc có thể là toàn bộ các số nguyên. Trong phần này, ta tìm hiểu lý thuyết cho *biến ngẫu nhiên liên tục*, là các biến ngẫu nhiên có thể nhận bất cứ giá trị số thực nào.

### 20.6.1 Biến Ngẫu nhiên Liên tục

Biến ngẫu nhiên liên tục phức tạp hơn đáng kể so với biến ngẫu nhiên rời rạc. Từ làm việc với các biến rời rạc chuyển sang làm việc với các biến liên tục cũng đòi hỏi một bước nhảy về kiến thức chuyên môn tương tự như chuyển từ tính tổng dãy số sang tích phân hàm số. Như vậy, ta sẽ cần dành một chút thời gian để phát triển lý thuyết.

#### Từ Rời rạc đến Liên tục

Để hiểu các thách thức kỹ thuật phát sinh khi làm việc với biến ngẫu nhiên liên tục, ta hãy thực hiện một thí nghiệm tưởng tượng sau đây. Giả sử ta chia phi tiêu vào một bảng phi tiêu, và muốn biết xác suất nó cắm chính xác vào điểm cách hòng tâm 2cm.

Để bắt đầu, hãy hình dung ta thực hiện phép đo với độ chính xác một chữ số, tức là chia thành các vùng 0cm, 1cm, 2cm, v.v. Phóng 100 phi tiêu vào bảng phi tiêu, và nếu 20 trong số đó rơi vào vùng 2cm, ta kết luận là 20% phi tiêu ta phóng cắm vào điểm cách tâm 2cm.

Tuy nhiên, khi xét kỹ hơn, câu trả lời này không thỏa đáng! Ta muốn một giá trị chính xác, trong khi các vùng đó lại chứa tất cả điểm nằm giữa 1.5cm và 2.5cm.

Hãy tiếp tục với độ chính xác cao hơn, như là 1.9cm, 2.0cm, 2.1cm, và bây giờ ta thấy khoảng 3 trong số 100 phi tiêu cắm vào bảng trong vùng 2.0cm. Do đó ta kết luận xác suất lúc này là 3%.

Tuy nhiên, điều này chưa giải quyết bất cứ điều gì! Ta chỉ vừa đẩy vấn đề độ chính xác lên thêm một chữ số thập phân. Thay vào đó hãy trừu tượng hóa vấn đề lên một chút. Hình dung ta biết xác suất mà  $k$  chữ số đầu tiên khớp với  $2.00000\dots$  và ta muốn biết xác suất nó khớp với  $k+1$  chữ số đầu tiên. Khá hợp lý khi giả định là chữ số thứ  $k+1$  có thể nhận giá trị ngẫu nhiên từ tập  $\{0, 1, 2, \dots, 9\}$ . Ít nhất là ta không thể nghĩ ra được bất kỳ tác nhân vật lý có ý nghĩa nào mà lại có ảnh hưởng tới độ chính xác ở mức micro mét, để chữ số cuối cùng là chữ số 7 thay vì chữ số 3 chẳng hạn.

Về cơ bản, việc tăng độ chính xác thêm một chữ số đòi hỏi xác suất khớp sẽ giảm xuống 10 lần. Hay nói cách khác, ta kỳ vọng là

$$P(\text{khoảng cách là } 2.00 \dots, \text{ đến } k \text{ chữ số}) \approx p \cdot 10^{-k}. \quad (20.6.1)$$

Giá trị  $p$  là xác suất khớp các chữ số đầu, và  $10^{-k}$  mô tả cho phần còn lại.

Lưu ý rằng nếu ta biết vị trí chính xác đến  $k = 4$  chữ số thập phân, có nghĩa là ta biết giá trị sẽ nằm trong khoảng  $[1.99995, 2.00005]$  có độ dài  $2.00005 - 1.99995 = 10^{-4}$ . Do đó, nếu gọi độ dài của khoảng này là  $\epsilon$ , ta có:

$$P(\text{khoảng cách nằm trong khoảng rộng } \epsilon \text{ xung quanh } 2) \approx \epsilon \cdot p. \quad (20.6.2)$$

Ta hãy tổng quát hóa thêm một bước cuối. Ta hiện chỉ đang xét điểm 2, chưa nghĩ đến các điểm khác. Về cơ bản, giá trị  $p$  tại các điểm khác nhau có thể sẽ khác nhau. Ít nhất ta hy vọng rằng người ném phi tiêu nhiều khả năng sẽ ngắm trúng vùng gần tâm, 2cm hơn là 20cm. Do đó, giá trị  $p$  là không cố định, mà phụ thuộc vào điểm  $x$ . Điều này cho thấy ta nên kỳ vọng:

$$P(\text{khoảng cách nằm trong khoảng rộng } \epsilon \text{ xung quanh } x) \approx \epsilon \cdot p(x). \quad (20.6.3)$$

(20.6.3) định nghĩa *hàm mật độ xác suất - probability density function (p.d.f.)*, là hàm  $p(x)$  biểu diễn xác suất tương đối của việc ném trúng gần vị trí này so với vị trí khác. Ta hãy trực quan hóa một hàm như vậy.

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from mxnet import np, npx
npx.set_np()

# Plot the probability density function for some random variable
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2)/np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2)/np.sqrt(2 * np.pi)

d2l.plot(x, p, 'x', 'Density')
```

Các vị trí mà giá trị hàm lớn cho biết có nhiều khả năng giá trị ngẫu nhiên sẽ rơi vào đó. Các vùng giá trị thấp là những vùng tại đó ít có khả năng giá trị ngẫu nhiên xuất hiện.

## Hàm Mật độ Xác suất

Bây giờ ta hãy tìm hiểu sâu hơn. Chúng ta đã quan sát trực quan hàm mật độ xác suất  $p(x)$  là gì đối với một biến ngẫu nhiên  $X$ , cụ thể:

$$P(X \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } x) \approx \epsilon \cdot p(x). \quad (20.6.4)$$

Nhưng phương trình này ám chỉ các tính chất gì của  $p(x)$ ?

Đầu tiên, xác suất không bao giờ âm, do đó  $p(x) \geq 0$ .

Thứ hai, hãy tưởng tượng việc cắt  $\mathbb{R}$  thành vô số lát cắt có chiều rộng  $\epsilon$ , mỗi lát cắt là nửa khoảng  $(\epsilon \cdot i, \epsilon \cdot (i + 1)]$ . Đối với mỗi lát cắt này, ta biết từ (20.6.4), thì xác suất xấp xỉ

$$P(X \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } x) \approx \epsilon \cdot p(\epsilon \cdot i), \quad (20.6.5)$$

vì vậy tổng tất cả chúng sẽ là

$$P(X \in \mathbb{R}) \approx \sum_i \epsilon \cdot p(\epsilon \cdot i). \quad (20.6.6)$$

Đây chỉ là xấp xỉ của một tích phân mà ta đã thảo luận trong Section 20.5, do đó có thể nói rằng

$$P(X \in \mathbb{R}) = \int_{-\infty}^{\infty} p(x) dx. \quad (20.6.7)$$

Ta biết là  $P(X \in \mathbb{R}) = 1$ , vì biến ngẫu nhiên này phải nhận một giá trị nào đó trong tập số thực, do đó ta có thể kết luận rằng với bất kỳ hàm mật độ nào:

$$\int_{-\infty}^{\infty} p(x) dx = 1. \quad (20.6.8)$$

Thật vậy, đi sâu hơn vào phương trình này, ta thấy rằng với bất kỳ  $a$  và  $b$  nào:

$$P(X \in (a, b]) = \int_a^b p(x) dx. \quad (20.6.9)$$

Ta có thể xấp xỉ phương trình này trong chương trình máy tính bằng cách sử dụng các phương pháp xấp xỉ rời rạc như trước đây. Trong trường hợp này, ta có thể ước tính xác suất nằm trong vùng màu xanh lam.

```
# Approximate probability using numerical integration
epsilon = 0.01
x = np.arange(-5, 5, 0.01)
p = 0.2*np.exp(-(x - 3)**2 / 2) / np.sqrt(2 * np.pi) + \
    0.8*np.exp(-(x + 1)**2 / 2) / np.sqrt(2 * np.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: {np.sum(epsilon*p[300:800])}'
```

Hai tính chất trên mô tả chính xác không gian của các hàm mật độ xác suất. Chúng là các hàm không âm  $p(x) \geq 0$  sao cho

$$\int_{-\infty}^{\infty} p(x) dx = 1. \quad (20.6.10)$$

Ta cũng có thể thu được xác suất biến ngẫu nhiên nằm trong một khoảng cụ thể bằng cách tính tích phân:

$$P(X \in (a, b]) = \int_a^b p(x) dx. \quad (20.6.11)$$

Trong Section 20.8, ta sẽ gặp một số phân phối thông dụng, giờ hãy tiếp tục tìm hiểu các khái niệm lý thuyết.

## Hàm Phân phối Tích lũy

Trong phần trước, chúng ta đã biết về hàm mật độ xác suất (p.d.f.). Trong thực tế, đây là một phương pháp thường dùng để thảo luận về các biến ngẫu nhiên liên tục, nhưng nó có một nhược điểm khá lớn: bản thân các giá trị của p.d.f. không phải là các giá trị xác suất, mà ta phải tích phân hàm này để có xác suất. Không có gì sai với một hàm mật độ lớn hơn 10, miễn là nó không lớn hơn 10 trong khoảng có chiều dài lớn hơn 1/10. Điều này có thể hơi phản trực giác, do đó người ta thường dùng *hàm phân tích lũy - cumulative distribution function* hoặc c.d.f., mà có giá trị trả về là xác suất.

Cụ thể, với việc sử dụng (20.6.11), ta định nghĩa c.d.f. cho một biến ngẫu nhiên  $X$  với mật độ  $p(x)$  như sau:

$$F(x) = \int_{-\infty}^x p(x) dx = P(X \leq x). \quad (20.6.12)$$

Hãy quan sát một vài tính chất của hàm này

- $F(x) \rightarrow 0$  khi  $x \rightarrow -\infty$ .
- $F(x) \rightarrow 1$  khi  $x \rightarrow \infty$ .
- $F(x)$  không giảm ( $y > x \implies F(y) \geq F(x)$ ).
- $F(x)$  là liên tục (không có bước nhảy) nếu  $X$  là một biến ngẫu nhiên liên tục.

Ở gạch đầu dòng thứ tư, lưu ý rằng điều này không đúng nếu  $X$  là rời rạc, ví dụ như khi  $X$  chỉ nhận hai giá trị 0 và 1 với xác suất 1/2. Trong trường hợp đó:

$$F(x) = \begin{cases} 0 & x < 0, \\ \frac{1}{2} & x < 1, \\ 1 & x \geq 1. \end{cases} \quad (20.6.13)$$

Trong ví dụ này, ta thấy một trong các lợi ích của việc sử dụng c.d.f., khả năng xử lý các biến ngẫu nhiên liên tục hoặc rời rạc với cùng một công cụ, hay thậm chí là hỗn hợp của cả hai (tung một đồng xu: nếu mặt ngửa thì trả về giá trị khi thả xúc xắc, nếu mặt sấp thì trả về khoảng cách ném phi tiêu từ tâm của bảng hồng tâm).

## Kỳ vọng

Giả sử ta đang làm việc với một biến ngẫu nhiên  $X$ . Phân phối của biến này có thể khó để diễn giải. Thường sẽ có ích nếu ta có thể tóm lược hành vi của một biến ngẫu nhiên một cách súc tích. Những giá trị giúp ta nắm bắt được hành vi của một biến ngẫu nhiên được gọi là *thống kê tóm tắt*. Các thống kê tóm tắt thường gặp nhất là *kỳ vọng*, *phương sai* và *độ lệch chuẩn*.

*Kỳ vọng* là giá trị trung bình của một biến ngẫu nhiên. Nếu ta có một biến ngẫu nhiên rời rạc  $X$ , nhận giá trị  $x_i$  với xác suất  $p_i$ , thì kỳ vọng được tính từ trung bình có trọng số: tổng các tích của giá trị biến với xác suất nhận giá trị đó:

$$\mu_X = E[X] = \sum_i x_i p_i. \quad (20.6.14)$$

Với một vài lưu ý, giá trị kỳ vọng này về cơ bản cho ta biết biến ngẫu nhiên có xu hướng nhận giá trị nào.

Xét một ví dụ tối giản xuyên suốt phần này, gọi  $X$  là biến ngẫu nhiên nhận giá trị  $a - 2$  với xác suất  $p$ ,  $a + 2$  với xác suất  $p$  và  $a$  với xác suất  $1 - 2p$ . Theo (20.6.14), với bất kỳ giá trị khả dĩ nào của  $a$  và  $p$ , giá trị kỳ vọng là:

$$\mu_X = E[X] = \sum_i x_i p_i = (a - 2)p + a(1 - 2p) + (a + 2)p = a. \quad (20.6.15)$$

Ta thấy rằng giá trị kỳ vọng là  $a$ . Điều này đúng với trực giác vì  $a$  là vị trí trung tâm của biến ngẫu nhiên này.

Bởi sự hữu dụng của kỳ vọng, hãy tổng hợp một vài tính chất của chúng.

- Với biến ngẫu nhiên  $X$  và hai số  $a, b$  bất kỳ:  $\mu_{aX+b} = a\mu_X + b$ .
- Với hai biến ngẫu nhiên  $X$  và  $Y$ :  $\mu_{X+Y} = \mu_X + \mu_Y$ .

Kỳ vọng rất hữu ích để hiểu hành vi trung bình của một biến ngẫu nhiên, tuy nhiên nó vẫn không đủ để ta có được một cách nhìn trực quan toàn diện. Tạo ra lợi nhuận  $\$10 \pm \$1$  rất khác với việc tạo ra  $\$10 \pm \$15$  cho mỗi giao dịch mặc dù cả hai có cùng kỳ vọng. Trường hợp thứ hai có mức độ dao động lớn hơn nhiều và do đó rủi ro cũng lớn hơn nhiều. Vì vậy, để hiểu hành vi của một biến ngẫu nhiên, ta sẽ cần thêm tối thiểu một thước đo nữa thể hiện biên độ dao động của biến ngẫu nhiên đó.

### Phương sai

Điều này dẫn tới khái niệm *phương sai* của biến ngẫu nhiên. Đây là một thước đo định lượng khoảng dao động quanh giá trị kỳ vọng của một biến ngẫu nhiên. Xét biểu thức  $X - \mu_X$ . Đây là độ lệch (*deviation*) của biến ngẫu nhiên so với kỳ vọng của nó. Giá trị này có thể dương hoặc âm, vì vậy ta cần thực hiện thêm thao tác để lấy độ lớn (luôn dương) của độ lệch này.

Một cách hợp lý là lấy  $|X - \mu_X|$ , và thực sự điều này dẫn đến một đại lượng hữu dụng là *trung bình độ lệch tuyệt đối - mean absolute deviation*, tuy nhiên do mối liên hệ với các lĩnh vực toán học và thống kê khác, người ta thường dùng một giải pháp khác.

Cụ thể là  $(X - \mu_X)^2$ . Nếu lấy giá trị kỳ vọng của đại lượng này, ta có phương sai:

$$\sigma_X^2 = \text{Var}(X) = E[(X - \mu_X)^2] = E[X^2] - \mu_X^2. \quad (20.6.16)$$

Đẳng thức cuối cùng trong (20.6.16) có được bằng cách khai triển các số hạng trong vế giữa và vận dụng các tính chất của kỳ vọng.

Hãy cùng xem lại ví dụ trong đó  $X$  là biến ngẫu nhiên nhận 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, ta đã biết  $\mu_X = a$ , vì vậy chỉ cần tính  $E[X^2]$  như sau:

$$E[X^2] = (a - 2)^2 p + a^2(1 - 2p) + (a + 2)^2 p = a^2 + 8p. \quad (20.6.17)$$

Sau đó, theo (20.6.16) ta có phương sai:

$$\sigma_X^2 = \text{Var}(X) = E[X^2] - \mu_X^2 = a^2 + 8p - a^2 = 8p. \quad (20.6.18)$$

Kết quả này cũng hợp lý. Giá trị lớn nhất có thể của  $p$  là  $1/2$ , tương ứng với việc chọn  $a - 2$  hoặc  $a + 2$  (tương tự khi tung đồng xu). Lúc này giá trị phương sai tính theo công thức trên bằng 4, đúng

với thực tế là cả  $a - 2$  và  $a + 2$  cùng có độ lệch khỏi giá trị trung bình là 2 và  $2^2 = 4$ . Ngược lại, nếu  $p = 0$ , tức biến ngẫu nhiên này luôn nhận giá trị 0 và vì thế có phuong sai bằng 0.

Hãy liệt kê một vài tính chất của phuong sai:

- Với biến ngẫu nhiên  $X$  bất kỳ:  $\text{Var}(X) \geq 0$ , với  $\text{Var}(X) = 0$  khi và chỉ khi  $X$  là hằng số.
- Với biến ngẫu nhiên  $X$  và hai số  $a, b$  bất kỳ:  $\text{Var}(aX + b) = a^2\text{Var}(X)$ .
- Nếu hai biến ngẫu nhiên  $X$  và  $Y$  là độc lập:  $\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y)$ .

Khi diễn giải các giá trị này, ta có thể gặp một chút vướng mắc. Cụ thể, hãy để ý đến đơn vị của các phép tính. Giả sử ta đang làm việc với số sao được đánh giá cho một sản phẩm trên trang web. Khi đó  $a$ ,  $a - 2$ , and  $a + 2$  đều được đo bằng đơn vị ngôi sao. Tương tự, kỳ vọng  $\mu_X$  sau đó cũng có đơn vị là ngôi sao (được tính là trung bình có trọng số). Tuy nhiên, nếu xét đến phuong sai, ta ngay lập tức gặp phải vấn đề, đó là  $(X - \mu_X)^2$  sẽ có đơn vị *bình phuong* số sao. Điều này có nghĩa là bản thân phuong sai không thể dùng để so sánh trong phép đo ban đầu. Để có thể diễn giải được nó, ta cần quay lại đơn vị gốc.

### Độ lệch chuẩn

*Độ lệch chuẩn* luôn có thể suy ra bằng cách lấy căn bậc hai của phuong sai:

$$\sigma_X = \sqrt{\text{Var}(X)}. \quad (20.6.19)$$

Trong ví dụ trên, ta có độ lệch chuẩn  $\sigma_X = 2\sqrt{2p}$ . Nếu đơn vị ta đang xét là số sao trong ví dụ đánh giá của mình,  $\sigma_X$  vẫn có đơn vị này.

Các tính chất của phuong sai có thể được áp dụng lại cho độ lệch chuẩn.

- Với biến ngẫu nhiên  $X$  bất kỳ:  $\sigma_X \geq 0$ .
- Với biến ngẫu nhiên  $X$  và hai số  $a, b$  bất kỳ:  $\sigma_{aX+b} = |a|\sigma_X$
- Nếu hai biến ngẫu nhiên  $X$  và  $Y$  là độc lập:  $\sigma_{X+Y} = \sqrt{\sigma_X^2 + \sigma_Y^2}$ .

Lúc này hãy đặt câu hỏi, “Nếu độ lệch chuẩn cùng đơn vị với biến ngẫu nhiên ban đầu, nó có cung cấp thông tin gì về biến ngẫu nhiên đó không?” Câu trả lời là có! Thật vậy, giống như kỳ vọng cho biết vị trí điển hình, độ lệch chuẩn cho biết khoảng biến thiên thường gặp của biến ngẫu nhiên đó. Ta có thể chứng minh chặt chẽ bằng bất đẳng thức Chebyshev:

$$P(X \notin [\mu_X - \alpha\sigma_X, \mu_X + \alpha\sigma_X]) \leq \frac{1}{\alpha^2}. \quad (20.6.20)$$

Điễn giải bằng lời như sau: ví dụ khi  $\alpha = 10$ , 99% số mẫu của bất kỳ biến ngẫu nhiên nào sẽ nằm trong khoảng 10 độ lệch chuẩn về 2 phía của giá trị kỳ vọng. Điều này cho ta một cách giải thích trực tiếp các thống kê tóm tắt tiêu chuẩn.

Để thấy sự tinh tế của mệnh đề này, hãy xét lại ví dụ trong đó  $X$  là biến ngẫu nhiên nhận giá trị  $a - 2$  với xác suất  $p$ ,  $a + 2$  với xác suất  $p$  và  $a$  với xác suất  $1 - 2p$ . Ta có kỳ vọng là  $a$  và độ lệch chuẩn là  $2\sqrt{2p}$ . Từ bất đẳng thức Chebyshev (20.6.20) với  $\alpha = 2$ , ta có

$$P(X \notin [a - 4\sqrt{2p}, a + 4\sqrt{2p}]) \leq \frac{1}{4}. \quad (20.6.21)$$

Điều này có nghĩa là trong 75% số lần lấy mẫu, biến ngẫu nhiên sẽ rơi vào khoảng trên, bất kể giá trị của  $p$ . Böyle giờ, hãy lưu ý rằng khi  $p \rightarrow 0$ , thì khoảng này cũng hội tụ đến điểm duy nhất là  $a$ .

Tuy nhiên biến ngẫu nhiên chỉ nhận các giá trị  $a - 2$ ,  $a$  và  $a + 2$  nên  $a - 2$  và  $a + 2$  chắc chắn sẽ nằm ngoài khoảng này! Câu hỏi đặt ra là giá trị  $p$  bằng bao nhiêu để  $a - 2$  và  $a + 2$  nằm trong khoảng đó? Ta cần giải phương trình:  $a + 4\sqrt{2p} = a + 2$  để ra nghiệm  $p = 1/8$ , đó chính xác là giá trị  $p$  nhỏ nhất thỏa mãn yêu cầu rằng không quá  $1/4$  số mẫu nằm ngoài khoảng ( $1/8$  về phía trái và  $1/8$  về phía phải giá trị kỳ vọng).

Hãy cùng trực quan hóa điều này. Chúng ta sẽ đưa ra xác suất nhận được ba giá trị tương ứng là ba thanh dọc có chiều cao tỷ lệ với xác suất. Khoảng trên sẽ được biểu diễn dưới dạng một đường ngang ở giữa. Biểu đồ đầu tiên cho thấy khi  $p > 1/8$ , khoảng này chưa hoàn toàn 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 * np.sqrt(2 * p),
                  a + 4 * np.sqrt(2 * p), 'black', lw=4)
    d2l=plt.vlines(a - 4 * np.sqrt(2 * p), 0.53, 0.47, 'black', lw=1)
    d2l=plt.vlines(a + 4 * np.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, 0.2)
```

Biểu đồ thứ hai cho thấy tại  $p = 1/8$ , khoảng này tiếp xúc với hai điểm. Khoảng này là *vừa đủ*, vì không thể chọn khoảng nhỏ hơn mà bất đẳng thức vẫn đúng.

```
# Plot interval when p = 1/8
plot_chebyshev(0.0, 0.125)
```

Biểu đồ thứ ba cho thấy với  $p < 1/8$  thì khoảng chỉ chứa giá trị trung tâm. Điều này không vi phạm bất đẳng thức vì ta chỉ cần đảm bảo rằng không quá  $1/4$  xác suất nằm ngoài khoảng, trên thực tế khi  $p < 1/8$ , biến ngẫu nhiên không thể nhận hai giá trị  $a - 2$  và  $a + 2$ .

```
# Plot interval when p < 1/8
plot_chebyshev(0.0, 0.05)
```

## Kỳ vọng và Phương sai trên Miền liên tục

Tới giờ ta đều mới chỉ xét biến ngẫu nhiên rời rạc, tuy nhiên trường hợp biến ngẫu nhiên liên tục cũng tương tự. Để hiểu cách hoạt động của các biến liên tục một cách trực quan, hãy tưởng tượng ta chia trực số nguyên thành nhiều khoảng với độ dài  $\epsilon$  trong khoảng  $[\epsilon i, \epsilon(i + 1)]$ . Sau khi thực hiện điều này, biến ngẫu nhiên liên tục trên trở thành dạng rời rạc và ta có thể áp dụng (20.6.14)

dưới dạng:

$$\begin{aligned}\mu_X &\approx \sum_i (\epsilon i) P(X \in (\epsilon i, \epsilon(i+1)]) \\ &\approx \sum_i (\epsilon i) p_X(\epsilon i) \epsilon,\end{aligned}\tag{20.6.22}$$

trong đó  $p_X$  là hàm mật độ của  $X$ . Đây là xấp xỉ tích phân của  $x p_X(x)$ , do đó ta có thể kết luận rằng:

$$\mu_X = \int_{-\infty}^{\infty} x p_X(x) dx.\tag{20.6.23}$$

Tương tự, áp dụng (20.6.16), phương sai có thể được biểu diễn như sau:

$$\sigma_X^2 = E[X^2] - \mu_X^2 = \int_{-\infty}^{\infty} x^2 p_X(x) dx - \left( \int_{-\infty}^{\infty} x p_X(x) dx \right)^2.\tag{20.6.24}$$

Tất cả những tính chất về kỳ vọng, phương sai và độ lệch chuẩn cho biến ngẫu nhiên rời rạc đều có thể áp dụng trong trường hợp liên tục. Ví dụ, xét biến ngẫu nhiên với hàm mật độ:

$$p(x) = \begin{cases} 1 & x \in [0, 1], \\ 0 & \text{otherwise.} \end{cases}\tag{20.6.25}$$

ta có thể tính:

$$\mu_X = \int_{-\infty}^{\infty} x p(x) dx = \int_0^1 x dx = \frac{1}{2}.\tag{20.6.26}$$

và

$$\sigma_X^2 = \int_{-\infty}^{\infty} x^2 p(x) dx - \left( \frac{1}{2} \right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}.\tag{20.6.27}$$

Để lưu ý, hãy quan sát thêm một ví dụ về *phân phối Cauchy*, với hàm mật độ:

$$p(x) = \frac{1}{1+x^2}.\tag{20.6.28}$$

```
# Plot the Cauchy distribution p.d.f.
x = np.arange(-5, 5, 0.01)
p = 1 / (1 + x**2)

d2l.plot(x, p, 'x', 'p.d.f.')
```

Hàm này nhìn có vẻ không có vấn đề gì, và quả thật tra cứu bảng tích phân chỉ ra rằng diện tích vùng dưới nó bằng 1, và do đó nó định nghĩa một biến ngẫu nhiên liên tục.

Để xem có vấn đề gì ở đây, hãy thử tính phương sai của hàm này bằng (20.6.16):

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2} dx.\tag{20.6.29}$$

Hàm bên trong tích phân có dạng:

```
# Plot the integrand needed to compute the variance
x = np.arange(-20, 20, 0.01)
p = x**2 / (1 + x**2)

d2l.plot(x, p, 'x', 'integrand')
```

Hàm này rõ ràng có phần diện tích bên dưới là vô hạn do về cơ bản nó là hằng số 1 với một đoạn trũng xuống gần 0, và quả thật:

$$\int_{-\infty}^{\infty} \frac{x^2}{1+x^2} dx = \infty. \quad (20.6.30)$$

Điều này có nghĩa là nó không có một phương sai hữu hạn đúng nghĩa.

Tuy vậy, nếu quan sát kỹ hơn ta có thể thấy một kết quả khó hiểu hơn nhiều. Hãy thử tính kỳ vọng sử dụng (20.6.14). Sử dụng đổi biến, ta được:

$$\mu_X = \int_{-\infty}^{\infty} \frac{x}{1+x^2} dx = \frac{1}{2} \int_1^{\infty} \frac{1}{u} du. \quad (20.6.31)$$

Hàm tích phân bên trong chính là định nghĩa của hàm logarit, do đó tích phân này có kết quả  $\log(\infty) = \infty$ , nên cũng không tồn tại giá trị kỳ vọng xác định!

Các nhà khoa học học máy định nghĩa mô hình của họ để thường không phải đổi mặt với những vấn đề này, và trong đại đa số các trường hợp, ta sẽ xử lý những biến ngẫu nhiên với kỳ vọng và phương sai xác định. Tuy vậy, đôi khi biến ngẫu nhiên với *đuôi nặng* (*heavy tails*) (có xác suất thu được các giá trị lớn là đủ lớn để khiến kỳ vọng hay phương sai không xác định) vẫn có ích trong việc mô hình hóa những hệ thống vật lý, vậy nên sự tồn tại của chúng đáng để biết tới.

## Hàm Mật độ Kết hợp

Toàn bộ phần phía trên đều chỉ xét biến ngẫu nhiên đơn lẻ có giá trị thực. Trường hợp có hai hay nhiều biến ngẫu nhiên hơn, mà thường giữa chúng có mối tương quan cao, thì sao? Tình huống này rất hay gặp trong học máy: tưởng tượng biến ngẫu nhiên  $R_{i,j}$  mã hóa giá trị màu đố của điểm ảnh tại toạ độ  $(i, j)$  trong một ảnh, hay biến  $P_t$  biểu diễn giá chứng khoán tại thời điểm  $t$ . Những điểm ảnh lân cận thường có màu tương tự, và giá tại các thời điểm lân cận thường tương tự. Ta không thể xem chúng như những biến ngẫu nhiên riêng biệt, và cũng không thể xây dựng một mô hình tốt (trong Section 20.9 có ví dụ một mô hình hoạt động kém do giả sử như vậy). Ta cần phát triển lý thuyết toán học để làm việc với những biến ngẫu nhiên liên tục có tương quan với nhau như vậy.

May mắn thay, với tích phân bội trong Section 20.5, ta có thể phát triển một lý thuyết như vậy. Để đơn giản, giả sử ta có hai biến ngẫu nhiên  $X, Y$  có thể tương quan với nhau. Sau đó, tương tự như trường hợp đơn biến, ta có thể đặt câu hỏi:

$$P(X \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } x \text{ và } Y \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } y). \quad (20.6.32)$$

Suy luận tương tự như trường hợp biến đơn chỉ ra rằng mệnh đề trên có thể xấp xỉ với:

$$P(X \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } x \text{ và } Y \text{ nằm trong khoảng rộng } \epsilon \text{ xung quanh } y) \approx \epsilon^2 p(x, y), \quad (20.6.33)$$

với một hàm  $p(x, y)$  nào đó. Đây được gọi là mật độ kết hợp của  $X$  và  $Y$ . Những tính chất của hàm mật độ cho biến đơn vẫn đúng cho trường hợp này:

- $p(x, y) \geq 0;$
- $\int_{\mathbb{R}^2} p(x, y) dx dy = 1;$
- $P((X, Y) \in \mathcal{D}) = \int_{\mathcal{D}} p(x, y) dx dy.$

Bằng cách này, ta có thể làm việc với nhiều biến ngẫu nhiên tương quan với nhau. Nếu số biến ngẫu nhiên nhiều hơn 2, ta có thể mở rộng hàm mật độ nhiều chiều:  $p(\mathbf{x}) = p(x_1, \dots, x_n)$ . Những thuộc tính như không âm, có tổng tích phân bằng một vẫn đúng.

## Phân phối Biên

Khi làm việc với nhiều biến ngẫu nhiên, ta thường muốn bỏ qua các tương quan và đặt câu hỏi, “biến ngẫu nhiên đơn lẻ này có phân phối như thế nào?” Phân phối như vậy được gọi là *phân phối biên (marginal distribution)*.

Cụ thể, giả sử ta có hai biến ngẫu nhiên  $X, Y$  với mật độ kết hợp  $p_{X,Y}(x, y)$ . Ta sẽ sử dụng chỉ số dưới để chỉ mật độ này của biến ngẫu nhiên nào. Bài toán trở thành sử dụng hàm này để tìm phân phối biên  $p_X(x)$ .

Như đa số trường hợp, hãy đưa ra một bức tranh trực quan để hiểu tường tận khái niệm. Nhắc lại rằng hàm mật độ  $p_X$  thỏa mãn

$$P(X \in [x, x + \epsilon]) \approx \epsilon \cdot p_X(x). \quad (20.6.34)$$

Hàm này không nhắc đến  $Y$ , nhưng nếu ta chỉ có  $p_{X,Y}$ , ta cần đưa  $Y$  vào bằng cách nào đó. Đầu tiên ta thấy hàm này giống với:

$$P(X \in [x, x + \epsilon], \text{ và } Y \in \mathbb{R}) \approx \epsilon \cdot p_X(x). \quad (20.6.35)$$

Trong trường hợp này mật độ không trực tiếp cho ta biết điều gì, ta cũng cần chia  $y$  thành các khoảng nhỏ, do đó ta có thể viết lại hàm này như sau:

$$\begin{aligned} \epsilon \cdot p_X(x) &\approx \sum_i P(X \in [x, x + \epsilon], \text{ và } Y \in [\epsilon \cdot i, \epsilon \cdot (i + 1)]) \\ &\approx \sum_i \epsilon^2 p_{X,Y}(x, \epsilon \cdot i). \end{aligned} \quad (20.6.36)$$

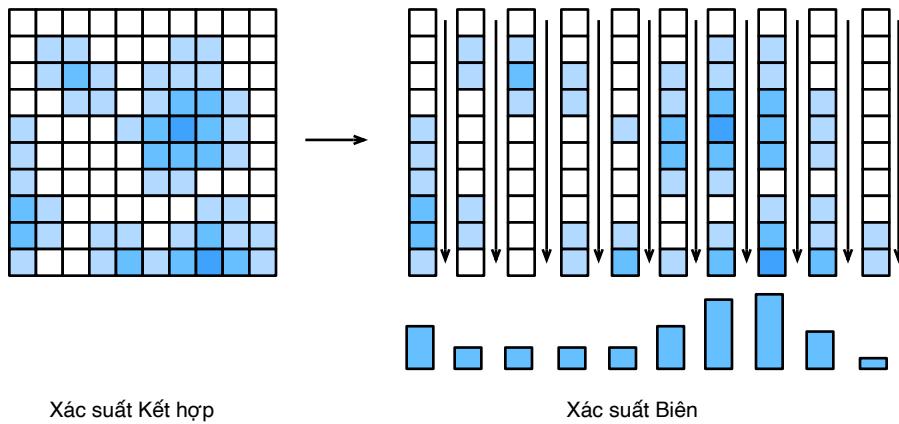


Fig. 20.6.1: Bằng cách lấy tổng theo cột trên mảng xác suất, ta có thể thu được phân phối biên cho biến ngẫu nhiên được biểu diễn theo trục  $x$ .

Điều này tức là lấy tổng giá trị mật độ trên chuỗi các hình vuông theo cột như trong Fig. 20.6.1. Thật vậy, sau khi khử số hạng epsilon ở cả hai vế, tổng về phải chính là tích phân theo  $y$  và ta có thể kết luận rằng:

$$\begin{aligned} p_X(x) &\approx \sum_i \epsilon p_{X,Y}(x, \epsilon \cdot i) \\ &\approx \int_{-\infty}^{\infty} p_{X,Y}(x, y) dy. \end{aligned} \quad (20.6.37)$$

Do đó:

$$p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y) dy. \quad (20.6.38)$$

Tức để thu được phân phối biên của một biến, ta cần lấy tích phân trên các biến còn lại. Quá trình này thường được gọi là *lấy tích phân - integrating out* hay *biên hóa - marginalized out* những biến không cần thiết.

### Hiệp phương sai

Khi làm việc với nhiều biến ngẫu nhiên, còn có một thông số thống kê nữa rất có ích: *hiệp phương sai (covariance)*. Thông số này đo mức độ biến thiên cùng nhau của hai biến ngẫu nhiên.

Để bắt đầu, giả sử ta có hai biến ngẫu nhiên rời rạc  $X$  và  $Y$ , xác suất mang giá trị  $(x_i, y_j)$  là  $p_{ij}$ . Trong trường hợp này, hiệp phương sai được định nghĩa như sau:

$$\sigma_{XY} = \text{Cov}(X, Y) = \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij}. = E[XY] - E[X]E[Y]. \quad (20.6.39)$$

Để hiểu một cách trực quan về công thức trên, xét cặp biến ngẫu nhiên:  $X$  có thể nhận giá trị 1 và

3, và  $Y$  có thể nhận giá trị  $-1$  và  $3$ . Giả sử ta có các xác suất sau:

$$\begin{aligned} P(X = 1 \text{ và } Y = -1) &= \frac{p}{2}, \\ P(X = 1 \text{ và } Y = 3) &= \frac{1-p}{2}, \\ P(X = 3 \text{ và } Y = -1) &= \frac{1-p}{2}, \\ P(X = 3 \text{ và } Y = 3) &= \frac{p}{2}, \end{aligned} \tag{20.6.40}$$

trong đó  $p$  là tham số tùy ý trong đoạn  $[0, 1]$ . Nếu  $p = 1$  thì  $X$  và  $Y$  luôn đồng thời mang giá trị lớn nhất hoặc nhỏ nhất của chúng, và nếu  $p = 0$  thì một biến mang giá trị lớn nhất trong khi biến còn lại mang giá trị nhỏ nhất. Nếu  $p = 1/2$  thì bốn khả năng có xác suất xảy ra như nhau, và không liên quan đến nhau. Hãy cùng tính hiệp phương sai. Đầu tiên,  $\mu_X = 2$  và  $\mu_Y = 1$ , do đó theo (20.6.39):

$$\begin{aligned} \text{Cov}(X, Y) &= \sum_{i,j} (x_i - \mu_X)(y_j - \mu_Y)p_{ij} \\ &= (1-2)(-1-1)\frac{p}{2} + (1-2)(3-1)\frac{1-p}{2} + (3-2)(-1-1)\frac{1-p}{2} + (3-2)(3-1)\frac{p}{2} \\ &= 4p - 2. \end{aligned} \tag{20.6.41}$$

Khi  $p = 1$  (trường hợp mà trong cùng một thời điểm chúng cùng là giá trị lớn nhất hoặc nhỏ nhất) hiệp phương sai bằng  $2$ . Khi  $p = 0$  (trường hợp mà chúng ngược nhau) hiệp phương sai bằng  $-2$ . Cuối cùng, khi  $p = 1/2$  (trường hợp chúng không liên quan đến nhau), hiệp phương sai bằng  $0$ . Từ đó ta thấy rằng hiệp phương sai biểu thị quan hệ của hai biến ngẫu nhiên này với nhau.

Chú ý là hiệp phương sai chỉ biểu thị mối quan hệ tuyến tính. Các quan hệ phức tạp hơn như  $X = Y^2$ , trong đó  $Y$  được chọn ngẫu nhiên với xác suất bằng nhau từ tập  $\{-2, -1, 0, 1, 2\}$ , có thể không được thể hiện. Quả thật ta có thể tính được hiệp phương sai của hai biến ngẫu nhiên này bằng không, mặc dù một biến là hàm tất định của biến còn lại.

Với biến ngẫu nhiên liên tục, khái niệm hiệp phương sai không đổi. Lúc này ta đã quen với việc biến đổi giữa miền rời rạc và liên tục, nên chúng tôi sẽ chỉ cung cấp dạng liên tục của (20.6.39) mà không giải thích thêm:

$$\sigma_{XY} = \int_{\mathbb{R}^2} (x - \mu_X)(y - \mu_Y)p(x, y) dx dy. \tag{20.6.42}$$

Để hiển thị, hãy quan sát tập các biến ngẫu nhiên có hiệp phương sai có thể điều chỉnh được.

```
# 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 = np.random.normal(0, 1, 500)
    Y = covs[i]*X + np.random.normal(0, 1, (500))

    d2l.plt.subplot(1, 4, i+1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cov = {covs[i]}')
d2l.plt.show()
```

Hãy xem xét một vài tính chất của hiệp phương sai:

- Với biến ngẫu nhiên  $X$  bất kỳ:  $\text{Cov}(X, X) = \text{Var}(X)$ .
- Với hai biến ngẫu nhiên  $X, Y$  và hai số  $a, b$  bất kỳ:  $\text{Cov}(aX + b, Y) = \text{Cov}(X, aY + b) = a\text{Cov}(X, Y)$ .
- Nếu  $X$  và  $Y$  độc lập:  $\text{Cov}(X, Y) = 0$ .

Ngoài ra, ta có thể sử dụng hiệp phương sai để mở rộng một hệ thức ta đã thấy trước đó. Hãy nhớ lại nếu  $X$  và  $Y$  là hai biến ngẫu nhiên độc lập thì:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y). \quad (20.6.43)$$

Với kiến thức về hiệp phương sai, ta có thể khai triển hệ thức này. Quả nhiên, sử dụng đại số có thể chứng minh tổng quát rằng:

$$\text{Var}(X + Y) = \text{Var}(X) + \text{Var}(Y) + 2\text{Cov}(X, Y). \quad (20.6.44)$$

Công thức này là dạng tổng quát của quy tắc tính tổng phương sai cho các biến ngẫu nhiên tương quan.

### Độ tương quan

Như trong trường hợp của kỳ vọng và phương sai, hãy xét đến đơn vị. Nếu  $X$  được đo bằng một đơn vị (giả sử là inch), và  $Y$  được đo bởi đơn vị khác (giả sử là đô-la), phương sai được tính bởi tích của hai đơn vị này  $\text{inch} \times \text{đô-la}$ . Những đơn vị này khó diễn giải, nên ta muốn có một phép đo sự tương quan mà không phụ thuộc vào đơn vị. Thật vậy, ta thường không quan tâm tới định lượng tương quan một cách chính xác, mà thường muốn biết sự tương quan này cùng hay ngược hướng, và mạnh như thế nào.

Để cắt nghĩa, hãy thực hiện một thí nghiệm tương tự. Giả sử ta chuyển đổi các biến ngẫu nhiên có đơn vị inch và đô-la thành inch và xu. Trong trường hợp này biến ngẫu nhiên  $Y$  được nhân thêm 100. Theo định nghĩa,  $\text{Cov}(X, Y)$  cũng sẽ được nhân thêm 100. Như vậy sự thay đổi đơn vị làm tăng hiệp phương sai 100 lần. Do đó, để có độ tương quan không phụ thuộc vào đơn vị, ta cần chia cho một đại lượng nào đó cũng được tăng thêm 100 lần. Một lựa chọn rõ ràng chính là độ lệch chuẩn! Có thể định nghĩa *hệ số tương quan - correlation coefficient* như sau:

$$\rho(X, Y) = \frac{\text{Cov}(X, Y)}{\sigma_X \sigma_Y}, \quad (20.6.45)$$

ta thấy đây là giá trị không phụ thuộc vào đơn vị. Một chút toán có thể chứng minh rằng  $\rho(X, Y)$  nằm giữa  $-1$  và  $1$  với  $1$  ứng với tương quan cực đại dương, còn  $-1$  ứng với tương quan cực đại âm.

Quay lại ví dụ ở miền rời rạc phía trên, ta có  $\sigma_X = 1$  và  $\sigma_Y = 2$ , và tương quan giữa hai biến ngẫu nhiên có thể tính bằng (20.6.45):

$$\rho(X, Y) = \frac{4p - 2}{1 \cdot 2} = 2p - 1. \quad (20.6.46)$$

Đại lượng này bây giờ nằm trong khoảng  $-1$  và  $1$  với  $1$  nghĩa là tương quan dương nhiều nhất,  $-1$  nghĩa là tương quan âm nhiều nhất.

Một ví dụ khác, xét biến ngẫu nhiên  $X$  bất kỳ, và  $Y = aX + b$  là một hàm tuyến tính tất định của  $X$ . Ta có:

$$\sigma_Y = \sigma_{aX+b} = |a|\sigma_X, \quad (20.6.47)$$

$$\text{Cov}(X, Y) = \text{Cov}(X, aX + b) = a\text{Cov}(X, X) = a\text{Var}(X), \quad (20.6.48)$$

và do đó theo (20.6.45) ta có:

$$\rho(X, Y) = \frac{a\text{Var}(X)}{|a|\sigma_X^2} = \frac{a}{|a|} = \text{sign}(a). \quad (20.6.49)$$

Ta thấy rằng độ tương quan là  $+1$  cho  $a > 0$ , và  $-1$  cho  $a < 0$ , tức độ tương quan đo mức độ và hướng của sự tương quan giữa hai biến ngẫu nhiên, không phải tỷ lệ biến đổi.

Ta hãy minh họa một vài biến ngẫu nhiên với tương quan có thể điều chỉnh.

```
# 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 = np.random.normal(0, 1, 500)
    Y = cors[i] * X + np.sqrt(1 - cors[i]**2) * np.random.normal(0, 1, 500)

    d2l.plt.subplot(1, 4, i + 1)
    d2l.plt.scatter(X.asnumpy(), Y.asnumpy())
    d2l.plt.xlabel('X')
    d2l.plt.ylabel('Y')
    d2l.plt.title(f'cor = {cors[i]}')
d2l.plt.show()
```

Ta liệt kê một vài tính chất của tương quan:

- Với biến ngẫu nhiên  $X$  bất kỳ,  $\rho(X, X) = 1$ .
- Với hai biến ngẫu nhiên  $X, Y$  và hai số  $a, b$  bất kỳ,  $\rho(aX + b, Y) = \rho(X, aY + b) = \rho(X, Y)$ .
- Nếu  $X$  và  $Y$  độc lập với phương sai khác không:  $\rho(X, Y) = 0$ .

Lưu ý cuối cùng, bạn có thể thấy rằng một vài công thức trên khá quen thuộc. Quả thật, nếu khai triển tất cả với giả định  $\mu_X = \mu_Y = 0$ , ta có:

$$\rho(X, Y) = \frac{\sum_{i,j} x_i y_i p_{ij}}{\sqrt{\sum_{i,j} x_i^2 p_{ij}} \sqrt{\sum_{i,j} y_j^2 p_{ij}}}. \quad (20.6.50)$$

Đây giống như tổng của tích các số hạng chia cho căn bậc hai của tổng bình phương các số hạng. Đó chính xác là công thức cho cô-sin của góc giữa hai vector  $\mathbf{v}, \mathbf{w}$  với trọng số tọa độ  $p_{ij}$ :

$$\cos(\theta) = \frac{\mathbf{v} \cdot \mathbf{w}}{\|\mathbf{v}\| \|\mathbf{w}\|} = \frac{\sum_i v_i w_i}{\sqrt{\sum_i v_i^2} \sqrt{\sum_i w_i^2}}. \quad (20.6.51)$$

Quả thật nếu nghĩ chuẩn (*norm*) liên quan tới độ lệch chuẩn, và độ tương quan là cô-sin của các góc, các trực giác ta có từ hình học có thể được áp dụng để tư duy về các biến ngẫu nhiên.

## 20.6.2 Tóm tắt

- Biến ngẫu nhiên liên tục là các biến ngẫu nhiên có thể lấy một dãy các giá trị liên tục. Chúng có một vài cản trở kỹ thuật khó giải quyết hơn so với biến ngẫu nhiên rời rạc.
- Hàm mật độ xác suất cho phép làm việc với các biến ngẫu nhiên liên tục bằng một hàm số mà diện tích dưới đường cong ở một khoảng là xác suất tìm được một mẫu trong khoảng đó.
- Hàm phân phối tích lũy là xác suất biến ngẫu nhiên nhận giá trị nhỏ hơn một ngưỡng nhất định. Đây là một góc nhìn hữu ích để hợp nhất các biến rời rạc và liên tục.
- Kỳ vọng là giá trị trung bình của một biến ngẫu nhiên.
- Phương sai là trung bình bình phương sự chênh lệch giữa biến ngẫu nhiên và kỳ vọng của nó.
- Độ lệch chuẩn là căn bậc hai của phương sai, được dùng để đo phạm vi giá trị mà biến ngẫu nhiên có thể nhận.
- Bất đẳng thức Chebyshev chặt chẽ hóa điều này bằng cách đưa ra một khoảng tường minh mà hầu hết các giá trị của biến ngẫu nhiên sẽ rơi vào.
- Mật độ kết hợp (*joint density*) cho phép ta làm việc với các biến ngẫu nhiên tương quan. Ta có thể biến hóa mật độ kết hợp bằng cách lấy tích phân theo các biến ngẫu nhiên khác để thu được phân phối của biến ngẫu nhiên mong muốn.
- Hiệp phương sai và hệ số tương quan là một cách đo bất kỳ mối quan hệ tuyến tính nào giữa hai biến ngẫu nhiên tương quan.

## 20.6.3 Bài tập

1. Giả sử ta có biến ngẫu nhiên với mật độ  $p(x) = \frac{1}{x^2}$  nếu  $x \geq 1$ , ngược lại  $p(x) = 0$ . Tính  $P(X > 2)$ .
2. Phân phối Laplace là một biến ngẫu nhiên có mật độ  $p(x) = \frac{1}{2}e^{-|x|}$ . Tính kỳ vọng và độ lệch chuẩn của biến ngẫu nhiên này. Gợi ý  $\int_0^\infty xe^{-x} dx = 1$  và  $\int_0^\infty x^2 e^{-x} dx = 2$ .
3. Tôi nói “Tôi có một biến ngẫu nhiên với kỳ vọng là 1, độ lệch chuẩn là 2, và tôi quan sát thấy 25% các mẫu của tôi có giá trị lớn hơn 9.” Bạn có tin tôi không? Tại sao?
4. Giả sử bạn có hai biến ngẫu nhiên  $X, Y$ , với mật độ kết hợp  $p_{XY}(x, y) = 4xy$  nếu  $x, y \in [0, 1]$ , ngược lại  $p_{XY}(x, y) = 0$ . Hiệp phương sai của  $X$  và  $Y$  là bao nhiêu?

## 20.6.4 Thảo luận

- Tiếng Anh: MXNet<sup>421</sup>, Pytorch<sup>422</sup>, Tensorflow<sup>423</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>424</sup>

<sup>421</sup> <https://discuss.d2l.ai/t/415>

<sup>422</sup> <https://discuss.d2l.ai/t/1094>

<sup>423</sup> <https://discuss.d2l.ai/t/1095>

<sup>424</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.6.5 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Phạm Đăng Khoa
- Đỗ Trường Giang
- Trần Yến Thy
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Văn Cường

## 20.7 Hợp lý Cực đại

Một trong những cách tư duy thường thấy nhất trong học máy là quan điểm về hợp lý cực đại. Đây là khái niệm mà khi làm việc với một mô hình xác suất mà các tham số chưa biết, các tham số làm cho các điểm dữ liệu đã quan sát có xác suất xảy ra cao nhất là những tham số hợp lý nhất.

### 20.7.1 Nguyên lý Hợp lý Cực đại

Nguyên lý này có cách diễn giải theo trường phái Bayes khá hữu ích. Giả sử rằng ta có một mô hình với các tham số  $\theta$  và một tập hợp các mẫu dữ liệu  $X$ . Cụ thể hơn, ta có thể tưởng tượng rằng  $\theta$  là một giá trị duy nhất đại diện cho xác suất một đồng xu ngửa khi tung, và  $X$  là một chuỗi các lần tung đồng xu độc lập. Chúng ta sẽ xem xét ví dụ này sâu hơn ở phần sau.

Nếu ta muốn tìm giá trị hợp lý nhất cho các tham số của mô hình, điều đó có nghĩa là ta muốn tìm

$$\operatorname{argmax} P(\theta | X). \quad (20.7.1)$$

Theo quy tắc Bayes, điều này giống với

$$\operatorname{argmax} \frac{P(X | \theta)P(\theta)}{P(X)}. \quad (20.7.2)$$

Biểu thức  $P(X)$  là xác suất sinh dữ liệu độc lập tham số, và nó hoàn toàn không phụ thuộc vào tham số  $\theta$ , do đó ta có thể bỏ qua nó mà không ảnh hưởng tới việc chọn ra  $\theta$  tốt nhất. Tương tự, bây giờ ta có thể cho rằng chúng ta không có giả định trước về bộ tham số nào là tốt hơn hết thay, vì vậy ta có thể phát biểu rằng  $P(\theta)$  cũng không phụ thuộc vào theta! Điều này hợp lý trong ví dụ tung đồng xu, ở đây xác suất để ra mặt ngửa có thể là bất kỳ giá trị nào trong khoảng  $[0, 1]$  khi mà ta không có bất kỳ niềm tin nào trước đó rằng đồng xu có cân xứng hay không (thường được gọi là *tiên nghiêm không chứa thông tin*). Do đó, ta thấy rằng việc áp dụng quy tắc Bayes sẽ chỉ ra lựa chọn tốt nhất cho  $\theta$  là ước lượng hợp lý cực đại cho  $\theta$ :

$$\hat{\theta} = \operatorname{argmax}_{\theta} P(X | \theta). \quad (20.7.3)$$

Theo thuật ngữ thông thường, xác suất của dữ liệu với các tham số đã cho ( $P(X | \theta)$ ) được gọi là *độ hợp lý*.

### Một ví dụ Cụ thể

Hãy cùng xem phương pháp này hoạt động như thế nào trong một ví dụ cụ thể. Giả sử rằng ta có một tham số duy nhất  $\theta$  biểu diễn cho xác suất tung đồng xu một lần được mặt ngửa. Khi đó, xác suất nhận được mặt sấp là  $1 - \theta$ , và vì vậy nếu dữ liệu quan sát  $X$  là một chuỗi có  $n_H$  mặt ngửa và  $n_T$  mặt sấp, ta có thể sử dụng tích chất tích các xác suất độc lập với nhau để có được

$$P(X | \theta) = \theta^{n_H} (1 - \theta)^{n_T}. \quad (20.7.4)$$

Nếu ta tung 13 đồng xu và nhận được chuỗi “HHHTHTTHHHHHT”, tức ta có  $n_H = 9$  và  $n_T = 4$ , thì ta nhận được ở đây là

$$P(X | \theta) = \theta^9 (1 - \theta)^4. \quad (20.7.5)$$

Một điều thú vị ở ví dụ này là ta biết trước câu trả lời. Thật vậy, nếu chúng ta phát biểu bằng lời, “Tôi đã tung 13 đồng xu và 9 đồng xu ra mặt ngửa, dự đoán tốt nhất cho xác suất tung đồng xu được mặt ngửa là bao nhiêu?” mọi người sẽ đều đoán đúng  $9/13$ . Điều mà phương pháp khả năng hợp lý cực đại cung cấp cho chúng ta là một cách để thu được con số đó từ các nguyên tắc cơ bản sao cho có thể khái quát được cho các tình huống phức tạp hơn rất nhiều.

Với ví dụ này, đồ thị của  $P(X | \theta)$  có dạng như sau:

```
%matplotlib inline
from d2l import mxnet as d2l
from mxnet import autograd, np, npx
npx.set_np()

theta = np.arange(0, 1, 0.001)
p = theta**9 * (1 - theta)**4.

d2l.plot(theta, p, 'theta', 'likelihood')
```

Xác suất này có giá trị tối đa ở đâu đó gần  $9/13 \approx 0.7\dots$  như đã dự đoán. Để kiểm tra xem nó có nằm chính xác ở đó không, chúng ta có thể nhờ đến giải tích. Chú ý rằng ở điểm cực đại, hàm này sẽ phẳng. Do đó, ta có thể tìm ước lượng hợp lý cực đại (20.7.1) bằng cách tìm các giá trị của  $\theta$  để đạo hàm bằng 0, rồi xem giá trị nào trả về xác suất cao nhất. Ta tính toán:

$$\begin{aligned} 0 &= \frac{d}{d\theta} P(X | \theta) \\ &= \frac{d}{d\theta} \theta^9 (1 - \theta)^4 \\ &= 9\theta^8(1 - \theta)^4 - 4\theta^9(1 - \theta)^3 \\ &= \theta^8(1 - \theta)^3(9 - 13\theta). \end{aligned} \quad (20.7.6)$$

Phương trình này có ba nghiệm: 0, 1 và  $9/13$ . Hai giá trị đầu tiên rõ ràng là cực tiểu, không phải cực đại vì chúng cho xác suất bằng 0 đối với chuỗi kết quả tung đồng xu. Giá trị cuối cùng *không* cho xác suất bằng 0 với chuỗi đã cho và do đó nó phải là ước lượng hợp lý cực đại  $\hat{\theta} = 9/13$ .

## Tối ưu hóa Số học và hàm Log hợp lí Âm

Ví dụ trước khá ổn, nhưng điều gì sẽ xảy ra nếu chúng ta có hàng tỷ tham số và mẫu dữ liệu.

Trước tiên, hãy lưu ý rằng, nếu chúng ta giả định rằng tất cả các mẫu dữ liệu là độc lập, thì ta có thể không còn thấy tính khả thi từ độ hợp lý khi chính nó là tích của nhiều xác suất. Thực vậy, mỗi xác suất nằm trong đoạn  $[0, 1]$ , giá trị thường là  $1/2$  và tích của  $(1/2)^{1000000000}$  nhỏ hơn nhiều so với độ chính xác của máy. Ta không thể làm việc trực tiếp với biểu thức này.

Tuy nhiên, nhắc lại rằng hàm log chuyển đổi tích thành tổng, trong trường hợp này thì

$$\log((1/2)^{1000000000}) = 1000000000 \cdot \log(1/2) \approx -301029995.6\dots \quad (20.7.7)$$

Con số này hoàn toàn nằm trong khoảng giá trị của một số thực dấu phẩy động 32-bit với độ chính xác đơn. Vì vậy, chúng ta nên xem xét *độ hợp lý thang log (log-likelihood)*, chính là

$$\log(P(X | \theta)). \quad (20.7.8)$$

Vì hàm  $x \mapsto \log(x)$  đồng biến, việc cực đại hóa độ hợp lý đồng nghĩa với việc cực đại hóa log hợp lý. Thực vậy trong Section 20.9, ta sẽ thấy lập luận này được áp dụng khi làm việc với ví dụ cụ thể cho bộ phân loại Naive Bayes.

Ta thường làm việc với các hàm mất mát, với mong muốn cực tiểu hóa chúng. Ta có thể đổi từ việc tìm hợp lý cực đại thành việc tìm cực tiểu mất mát bằng cách lấy  $-\log(P(X | \theta))$ , tức *hàm đổi log hợp lý (negative log-likelihood)*.

Để minh họa điều này, hãy xem xét bài toán tung đồng xu trước đó và giả sử rằng ta không biết nghiệm dạng đóng. Ta có thể tính ra

$$-\log(P(X | \theta)) = -\log(\theta^{n_H} (1 - \theta)^{n_T}) = -(n_H \log(\theta) + n_T \log(1 - \theta)). \quad (20.7.9)$$

Đẳng thức này có thể được lập trình và được tối ưu hóa hoàn toàn ngay cả với hàng tỷ lần tung đồng xu.

```
# Set up our data
n_H = 8675309
n_T = 25624

# Initialize our parameters
theta = np.array(0.5)
theta.attach_grad()

# Perform gradient descent
lr = 0.0000000001
for iter in range(10):
    with autograd.record():
        loss = -(n_H * np.log(theta) + n_T * np.log(1 - theta))
        loss.backward()
        theta -= lr * theta.grad

# Check output
theta, n_H / (n_H + n_T)
```

Sự thuận tiện số học chỉ là một trong những lý do khiến mọi người thích dùng hàm đối log hợp lý. Thật ra còn có một vài lý do khác mà nó có thể được lựa chọn.

Lý do thứ hai mà ta xem xét đến hàm log hợp lý là việc áp dụng các quy tắc giải tích trở nên đơn giản hơn. Như đã thảo luận ở trên, do các giả định về tính độc lập, hầu hết các xác suất mà chúng ta gặp phải trong học máy là tích của các xác suất riêng lẻ.

$$P(X | \theta) = p(x_1 | \theta) \cdot p(x_2 | \theta) \cdots p(x_n | \theta). \quad (20.7.10)$$

Điều này có nghĩa là nếu ta áp dụng trực tiếp quy tắc nhân để tính đạo hàm thì ta sẽ có được

$$\begin{aligned} \frac{\partial}{\partial \theta} P(X | \theta) &= \left( \frac{\partial}{\partial \theta} P(x_1 | \theta) \right) \cdot P(x_2 | \theta) \cdots P(x_n | \theta) \\ &\quad + P(x_1 | \theta) \cdot \left( \frac{\partial}{\partial \theta} P(x_2 | \theta) \right) \cdots P(x_n | \theta) \\ &\quad \vdots \\ &\quad + P(x_1 | \theta) \cdot P(x_2 | \theta) \cdots \left( \frac{\partial}{\partial \theta} P(x_n | \theta) \right). \end{aligned} \quad (20.7.11)$$

Biểu thức này đòi hỏi  $n(n - 1)$  phép nhân, cùng với  $(n - 1)$  phép cộng, vì vậy tổng thời gian chạy tỷ lệ bình phương với số lượng đầu vào! Nếu ta khôn khéo trong việc nhóm các phân tử thì độ phức tạp sẽ giảm xuống tuyến tính, nhưng việc này yêu cầu ta phải suy nghĩ một chút. Đối với hàm đối log hợp lý, chúng ta có

$$-\log(P(X | \theta)) = -\log(P(x_1 | \theta)) - \log(P(x_2 | \theta)) \cdots - \log(P(x_n | \theta)), \quad (20.7.12)$$

điều này đưa đến kết quả là

$$-\frac{\partial}{\partial \theta} \log(P(X | \theta)) = \frac{1}{P(x_1 | \theta)} \left( \frac{\partial}{\partial \theta} P(x_1 | \theta) \right) + \cdots + \frac{1}{P(x_n | \theta)} \left( \frac{\partial}{\partial \theta} P(x_n | \theta) \right). \quad (20.7.13)$$

Đẳng thức này chỉ yêu cầu  $n$  phép chia và  $n - 1$  phép cộng, và do đó thời gian chạy tỷ lệ tuyến tính với số đầu vào.

Lý do thứ ba và cũng là cuối cùng khi xem xét hàm đối log hợp lý đó là sự liên hệ với lý thuyết thông tin, mà chúng ta sẽ thảo luận chi tiết tại phần Section 20.11. Đây là một lý thuyết toán học chặt chẽ đưa ra cách đo lường mức độ thông tin hoặc độ ngẫu nhiên của một biến ngẫu nhiên. Đối tượng nghiên cứu chính trong lĩnh vực đó là entropy

$$H(p) = - \sum_i p_i \log_2(p_i), \quad (20.7.14)$$

công thức trên đo lường độ ngẫu nhiên của một nguồn. Hãy để ý rằng đây chỉ là giá trị trung bình của  $-\log$  xác suất, và do đó, nếu ta lấy hàm đối log hợp lý và chia cho số lượng mẫu dữ liệu, ta sẽ nhận được một đại lượng liên quan được gọi là entropy chéo. Chỉ riêng việc diễn giải mang tính lý thuyết này thôi là đủ thuyết phục để ta sử dụng giá trị đối log hợp lý trung bình trên một tập dữ liệu như một cách đo lường chất lượng của mô hình.

## 20.7.2 Hợp lý Cực đại cho Biến Liên tục

Tất cả những điều chúng ta đã làm ở trước đều giả định rằng ta đang làm việc với biến ngẫu nhiên rời rạc, nhưng nếu chúng ta muốn làm việc với các biến liên tục thì sao?

Nói ngắn gọn thì không có thứ gì thay đổi cả, ngoại trừ việc ta thay thế tất cả giá trị xác suất bằng mật độ xác suất. Nhắc lại rằng chúng ta ký hiệu mật độ bằng chữ thường  $p$ , nghĩa là bây giờ ta sẽ có

$$-\log(p(X | \theta)) = -\log(p(x_1 | \theta)) - \log(p(x_2 | \theta)) \cdots - \log(p(x_n | \theta)) = -\sum_i \log(p(x_i | \theta)). \quad (20.7.15)$$

Câu hỏi lúc này trở thành, “Tại sao điều này lại ổn?” Rốt cuộc, lý do chúng ta đưa ra khái niệm mật độ là vì xác suất nhận được một kết quả cụ thể bằng không, và do đó chẳng phải xác suất sinh dữ liệu đối với tập hợp tham số bất kỳ sẽ bằng không sao?

Quả thật điều này là đúng, và việc hiểu tại sao chúng ta có thể chuyển sang mật độ là một bài tập trong việc truy ra những gì xảy ra đối với các epsilon.

Đầu tiên hãy xác định lại mục tiêu của chúng ta. Giả sử rằng đối với các biến ngẫu nhiên liên tục, ta không còn muốn tính xác suất tại chính ngay mỗi giá trị, mà thay vào đó là tìm xác suất trong một phạm vi  $\epsilon$  nào đó. Để đơn giản, ta giả định rằng dữ liệu là các mẫu quan sát lặp lại  $x_1, \dots, x_N$  của các biến ngẫu nhiên được phân phối giống nhau  $X_1, \dots, X_N$ . Như chúng ta đã thấy trước đây, giả định này có thể được biểu diễn như sau

$$\begin{aligned} P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \dots, X_N \in [x_N, x_N + \epsilon] | \theta) \\ \approx \epsilon^N p(x_1 | \theta) \cdot p(x_2 | \theta) \cdots p(x_n | \theta). \end{aligned} \quad (20.7.16)$$

Do đó, nếu ta lấy đối của logarit cho biểu thức này thì ta sẽ nhận được

$$\begin{aligned} -\log(P(X_1 \in [x_1, x_1 + \epsilon], X_2 \in [x_2, x_2 + \epsilon], \dots, X_N \in [x_N, x_N + \epsilon] | \theta)) \\ \approx -N \log(\epsilon) - \sum_i \log(p(x_i | \theta)). \end{aligned} \quad (20.7.17)$$

Nếu chúng ta xem xét biểu thức này, vị trí duy nhất mà  $\epsilon$  xuất hiện là trong hằng số cộng  $-N \log(\epsilon)$ . Hằng số này hoàn toàn không phụ thuộc vào các tham số  $\theta$ , vì vậy lựa chọn tối ưu cho  $\theta$  không phụ thuộc vào việc lựa chọn  $\epsilon$ ! Dù ta muốn lấy bốn hoặc bốn trăm chữ số, lựa chọn  $\theta$  tốt nhất sẽ không đổi, do đó ta có thể loại bỏ hằng epsilon để có được biểu thức mà ta muốn tối ưu là

$$-\sum_i \log(p(x_i | \theta)). \quad (20.7.18)$$

Do đó, chúng ta thấy rằng quan điểm hợp lý cực đại có thể áp dụng được với các biến ngẫu nhiên liên tục dễ dàng như với các biến rời rạc bằng cách thay thế các xác suất bằng mật độ xác suất.

### 20.7.3 Tóm tắt

- Nguyên lý hợp lý cực đại cho ta biết rằng mô hình phù hợp nhất cho một tập dữ liệu nhất định là mô hình tạo ra các điểm dữ liệu đó với xác suất cao nhất.
- Tuy nhiên, thường thì mọi người hay làm việc với hàm đối log hợp lý vì nhiều lý do: tính ổn định số học, khả năng biến đổi tích thành tổng (dẫn tới việc đơn giản hóa các phép tính gradient) và mối liên hệ mật thiết về mặt lý thuyết với lý thuyết thông tin.
- Trong khi áp dụng phương pháp này là đơn giản nhất trong trường hợp rời rạc, nó cũng có thể hoàn toàn tổng quát hóa cho trường hợp liên tục bằng cách cực đại hóa mật độ xác suất của các điểm dữ liệu.

### 20.7.4 Bài tập

1. Giả sử bạn biết rằng một biến ngẫu nhiên có mật độ bằng  $\frac{1}{\alpha}e^{-\alpha x}$  với một giá trị  $\alpha$  nào đó. Bạn nhận được một quan sát duy nhất từ biến ngẫu nhiên này là số 3. Giá trị ước lượng hợp lý cực đại cho  $\alpha$  là bao nhiêu?
2. Giả sử rằng bạn có tập dữ liệu với các mẫu  $\{x_i\}_{i=1}^N$  được lấy từ một phân phối Gauss với giá trị trung bình chưa biết, nhưng phương sai bằng 1. Giá trị ước lượng hợp lý cực đại của trung bình là bao nhiêu?

### 20.7.5 Thảo luận

- Tiếng Anh: MXNet<sup>425</sup>, Pytorch<sup>426</sup>, Tensorflow<sup>427</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>428</sup>

### 20.7.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Phạm Minh Đức
- Phạm Đăng Khoa
- Phạm Hồng Vinh

<sup>425</sup> <https://discuss.d2l.ai/t/416>

<sup>426</sup> <https://discuss.d2l.ai/t/1096>

<sup>427</sup> <https://discuss.d2l.ai/t/1097>

<sup>428</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.8 Các Phân phối Xác suất

Lúc này ta đã hiểu cách làm việc với xác suất cho biến ngẫu nhiên rời rạc và liên tục, hãy làm quen với một số phân phối xác suất thường gặp. Tùy thuộc vào lĩnh vực học máy, ta có thể phải làm quen với nhiều phân phối hơn, hoặc đối với một số lĩnh vực trong học sâu thì có khả năng sẽ không gặp. Tuy nhiên, ta vẫn nên biết các phân phối cơ bản. Đầu tiên hãy nhập một số thư viện phổ biến.

```
%matplotlib inline
from d2l import mxnet as d2l
from IPython import display
from math import erf, factorial
import numpy as np
```

### 20.8.1 Phân phối Bernoulli

Đây là phân phối thường gặp đơn giản nhất. Giả sử khi tung một đồng xu, biến ngẫu nhiên  $X$  tuân theo phân phối này lấy giá trị mặt ngửa 1 với xác suất  $p$  và mặt sấp 0 với xác suất  $1 - p$ . Ta viết:

$$X \sim \text{Bernoulli}(p). \quad (20.8.1)$$

Hàm phân phối tích lũy là:

$$F(x) = \begin{cases} 0 & x < 0, \\ 1 - p & 0 \leq x < 1, \\ 1 & x \geq 1. \end{cases} \quad (20.8.2)$$

Hàm khối xác suất (*probability mass function*) được minh họa dưới đây:

```
p = 0.3

d2l.set_figsize()
d2l.plt.stem([0, 1], [1 - p, p], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

Bây giờ, hãy vẽ đồ thị cho hàm phân phối tích lũy (20.8.2).

```
x = np.arange(-1, 2, 0.01)

def F(x):
    return 0 if x < 0 else 1 if x > 1 else 1 - p

d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```

Nếu  $X \sim \text{Bernoulli}(p)$ , thì:

- $\mu_X = p$ ,
- $\sigma_X^2 = p(1 - p)$ .

Ta có thể lấy mẫu một mảng có kích thước tùy ý từ một biến ngẫu nhiên Bernoulli như sau:

```
1*(np.random.rand(10, 10) < p)
```

## 20.8.2 Phân phối Đều Rời rạc

Biến ngẫu nhiên thường gặp tiếp theo là biến phân phối đều rời rạc. Ta giả sử biến này được phân phối trên tập các số nguyên  $\{1, 2, \dots, n\}$ , tuy nhiên, có thể chọn bất kỳ tập giá trị nào khác. Ý nghĩa của từ *đều* trong ngữ cảnh này là mọi giá trị đều có thể xảy ra với khả năng như nhau. Xác suất cho mỗi giá trị  $i \in \{1, 2, 3, \dots, n\}$  là  $p_i = \frac{1}{n}$ . Ta ký hiệu một biến ngẫu nhiên  $X$  tuân theo phân phối này là:

$$X \sim U(n). \quad (20.8.3)$$

Hàm phân phối tích lũy là:

$$F(x) = \begin{cases} 0 & x < 1, \\ \frac{k}{n} & k \leq x < k + 1 \text{ with } 1 \leq k < n, \\ 1 & x \geq n. \end{cases} \quad (20.8.4)$$

Trước hết ta hãy vẽ đồ thị hàm khối xác suất:

```
n = 5

d2l.plt.stem([i+1 for i in range(n)], n*[1 / n], use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

Tiếp theo hãy vẽ đồ thị hàm phân phối tích luỹ (20.8.4).

```
x = np.arange(-1, 6, 0.01)

def F(x):
    return 0 if x < 1 else 1 if x > n else np.floor(x) / n

d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```

Nếu  $X \sim U(n)$ , thì:

- $\mu_X = \frac{1+n}{2}$ ,
- $\sigma_X^2 = \frac{n^2-1}{12}$ .

Ta có thể lấy mẫu một mảng có kích thước tùy ý từ một biến ngẫu nhiên rời rạc tuân theo phân phối đều như sau:

```
np.random.randint(1, n, size=(10, 10))
```

### 20.8.3 Phân phối Đều liên tục

Tiếp theo, hãy thảo luận về phân phối đều liên tục. Ý tưởng phía sau là nếu ta tăng  $n$  trong phân phối đều rời rạc, rồi biến đổi tỷ lệ để nó nằm trong đoạn  $[a, b]$ , ta sẽ tiến đến một biến ngẫu nhiên liên tục mà mọi điểm bất kỳ trong  $[a, b]$  đều có xác suất bằng nhau. Ta sẽ ký hiệu phân phối này bằng

$$X \sim U(a, b). \quad (20.8.5)$$

Hàm mật độ xác suất là:

$$p(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b], \\ 0 & x \notin [a, b]. \end{cases} \quad (20.8.6)$$

Hàm phân phối tích lũy là:

$$F(x) = \begin{cases} 0 & x < a, \\ \frac{x-a}{b-a} & x \in [a, b], \\ 1 & x \geq b. \end{cases} \quad (20.8.7)$$

Trước hết hãy vẽ hàm mật độ xác suất (20.8.6).

```
a, b = 1, 3  
x = np.arange(0, 4, 0.01)  
p = (x > a)*(x < b)/(b - a)  
d2l.plot(x, p, 'x', 'p.d.f.')
```

Giờ hãy vẽ hàm phân phối tích lũy (20.8.7).

```
def F(x):  
    return 0 if x < a else 1 if x > b else (x - a) / (b - a)  
d2l.plot(x, np.array([F(y) for y in x]), 'x', 'c.d.f.')
```

Nếu  $X \sim U(a, b)$ , thì:

- $\mu_X = \frac{a+b}{2}$ ,
- $\sigma_X^2 = \frac{(b-a)^2}{12}$ .

Ta có thể lấy mẫu một mảng với kích thước bất kỳ từ một biến ngẫu nhiên liên tục tuân theo phân phối đều như sau. Chú ý rằng theo mặc định việc lấy mẫu là từ  $U(0, 1)$ , nên nếu lấy mẫu trên miền giá trị khác, ta cần phải biến đổi tỷ lệ.

```
(b - a) * np.random.rand(10, 10) + a
```

#### 20.8.4 Phân phối Nhị thức

Biến ngẫu nhiên *nhi thức* thì phức tạp hơn một chút. Biến ngẫu nhiên này bắt nguồn từ việc thực hiện liên tiếp  $n$  thí nghiệm độc lập, mỗi thí nghiệm có xác suất thành công  $p$ , và hỏi xem số lần thành công kỳ vọng là bao nhiêu.

Hãy biểu diễn dưới dạng toán học. Mỗi thí nghiệm là một biến ngẫu nhiên độc lập  $X_i$  với 1 có nghĩa là thành công, 0 có nghĩa là thất bại. Vì mỗi thí nghiệm là một lần tung đồng xu độc lập với xác suất thành công  $p$ , ta có thể nói  $X_i \sim \text{Bernoulli}(p)$ . Biến ngẫu nhiên nhị thức là:

$$X = \sum_{i=1}^n X_i. \quad (20.8.8)$$

Trong trường hợp này, ta viết:

$$X \sim \text{Binomial}(n, p). \quad (20.8.9)$$

Để lấy hàm phân phối tích lũy, ta cần chú ý rằng  $k$  lần thành công có thể xảy ra theo  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$  cách, với mỗi cách có xác suất xảy ra  $p^k(1-p)^{n-k}$ . Do đó, hàm phân phối tích lũy là:

$$F(x) = \begin{cases} 0 & x < 0, \\ \sum_{m \leq k} \binom{n}{m} p^m (1-p)^{n-m} & k \leq x < k+1 \text{ với } 0 \leq k < n, \\ 1 & x \geq n. \end{cases} \quad (20.8.10)$$

Trước hết hãy vẽ hàm khối xác suất.

```
n, p = 10, 0.2

# Compute binomial coefficient
def binom(n, k):
    comb = 1
    for i in range(min(k, n - k)):
        comb = comb * (n - i) // (i + 1)
    return comb

pmf = np.array([p**i * (1-p)**(n - i) * binom(n, i) for i in range(n + 1)])

d2l.plt.stem([i for i in range(n + 1)], pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()
```

Giờ hãy vẽ hàm phân phối tích lũy (20.8.10).

```
x = np.arange(-1, 11, 0.01)
cmf = np.cumsum(pmf)

def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, np.array([F(y) for y in x.tolist()]), 'x', 'c.d.f.')
```

Dù không dễ để suy ra công thức, kỳ vọng và phương sai của phân phối được tính như sau:

- $\mu_X = np$ ,
- $\sigma_X^2 = np(1 - p)$ .

Ta có thể lấy mẫu từ phân phối này theo cách bên dưới.

```
np.random.binomial(n, p, size=(10, 10))
```

### 20.8.5 Phân phối Poisson

Hãy cùng thực hiện một thí nghiệm tưởng tượng. Ta đang đứng ở một trạm xe buýt và muốn biết có bao nhiêu chiếc xe buýt sẽ đi qua trong phút tiếp theo. Hãy bắt đầu bằng việc coi  $X^{(1)} \sim \text{Bernoulli}(p)$  đơn giản là xác suất một chiếc xe buýt sẽ đến trong khoảng một phút tiếp theo. Với những trạm xe buýt xa trung tâm thành phố, đây có thể là một xấp xỉ rất tốt vì ta hầu như sẽ không bao giờ thấy nhiều hơn một chiếc xe buýt trong một phút.

Tuy nhiên, trong một khu vực đông đúc, ta có thể và thậm chí khả năng cao sẽ thấy hai chiếc xe buýt đi qua. Ta có thể mô hình hóa điều này bằng cách chia nhỏ biến độc lập của ta thành hai phần với khoảng thời gian 30 giây. Trong trường hợp này ta có thể viết:

$$X^{(2)} \sim X_1^{(2)} + X_2^{(2)}, \quad (20.8.11)$$

với  $X^{(2)}$  là tổng toàn phần, và  $X_i^{(2)} \sim \text{Bernoulli}(p/2)$ . Toàn bộ phân phối vì thế sẽ là  $X^{(2)} \sim \text{Binomial}(2, p/2)$ .

Hãy tiếp tục chia nhỏ một phút này thành  $n$  phần. Lập luận tương tự như trên, ta có:

$$X^{(n)} \sim \text{Binomial}(n, p/n). \quad (20.8.12)$$

Hãy xem xét các biến ngẫu nhiên này. Ở mục trước, ta đã biết (20.8.12) có kỳ vọng  $\mu_{X^{(n)}} = n(p/n) = p$ , và phương sai  $\sigma_{X^{(n)}}^2 = n(p/n)(1 - (p/n)) = p(1 - p/n)$ . Nếu cho  $n \rightarrow \infty$ , ta có thể thấy rằng hai giá trị này dần tiến về  $\mu_{X^{(\infty)}} = p$ , và phương sai  $\sigma_{X^{(\infty)}}^2 = p$ . Điều này gợi ý rằng ta có thể định nghĩa thêm một biến ngẫu nhiên nào đó trong trường hợp việc chia nhỏ này tiến ra vô cùng.

Điều này không có gì ngạc nhiên, trong thực tế ta có thể chỉ cần đếm số lần xe buýt đến, tuy nhiên sẽ tốt hơn nếu định nghĩa một mô hình toán học hoàn chỉnh, được biết đến là *định luật của biến cố hiếm - law of rare events*.

Bám sát chuỗi lập luận một cách cẩn thận, ta có thể suy ra một mô hình như sau. Ta nói  $X \sim \text{Poisson}(\lambda)$  nếu nó là một biến ngẫu nhiên nhận các giá trị  $\{0, 1, 2, \dots\}$  với xác suất:

$$p_k = \frac{\lambda^k e^{-\lambda}}{k!}. \quad (20.8.13)$$

Giá trị  $\lambda > 0$  được gọi là *tốc độ* (hoặc tham số *hình dạng*), tương ứng cho số lần xuất hiện trung bình trong một đơn vị thời gian.

Ta có thể lấy tổng hàm khối xác suất này để có được hàm phân phối tích lũy.

$$F(x) = \begin{cases} 0 & x < 0, \\ e^{-\lambda} \sum_{m=0}^k \frac{\lambda^m}{m!} & k \leq x < k+1 \text{ với } 0 \leq k. \end{cases} \quad (20.8.14)$$

Trước hết hãy vẽ hàm khối xác suất (20.8.13).

```

lam = 5.0

xs = [i for i in range(20)]
pmf = np.array([np.exp(-lam) * lam**k / factorial(k) for k in xs])

d2l.plt.stem(xs, pmf, use_line_collection=True)
d2l.plt.xlabel('x')
d2l.plt.ylabel('p.m.f.')
d2l.plt.show()

```

Bây giờ, ta hãy vẽ hàm phân phối tích lũy (20.8.14).

```

x = np.arange(-1, 21, 0.01)
cmf = np.cumsum(pmf)
def F(x):
    return 0 if x < 0 else 1 if x > n else cmf[int(x)]

d2l.plot(x, np.array([F(y) for y in x.tolist()]), 'x', 'c.d.f.')

```

Như ta thấy ở trên, kỳ vọng và phương sai của phân phối này đặc biệt súc tích. Nếu  $X \sim \text{Poisson}(\lambda)$ :

- $\mu_X = \lambda$ ,
- $\sigma_X^2 = \lambda$ .

Ta có thể lấy mẫu từ phân phối này như sau.

```
np.random.poisson(lam, size=(10, 10))
```

### 20.8.6 Phân phối Gauss

Bây giờ ta hãy thử một thí nghiệm khác có liên quan. Giả sử ta lại thực hiện  $n$  phép đo Bernoulli( $p$ ) độc lập  $X_i$ . Tổng của chúng có phân phối là  $X^{(n)} \sim \text{Binomial}(n, p)$ . Thay vì lấy giới hạn khi  $n$  tăng và  $p$  giảm, hãy cố định  $p$ , rồi cho  $n \rightarrow \infty$ . Trong trường hợp này  $\mu_{X^{(n)}} = np \rightarrow \infty$  và  $\sigma_{X^{(n)}}^2 = np(1-p) \rightarrow \infty$ , vì vậy giới hạn này không thể xác định được.

Tuy nhiên, vẫn có cách giải quyết khác! Có thể làm kỳ vọng và phương sai xác định bằng cách định nghĩa:

$$Y^{(n)} = \frac{X^{(n)} - \mu_{X^{(n)}}}{\sigma_{X^{(n)}}}. \quad (20.8.15)$$

Biến này được coi là có kỳ vọng bằng không và phương sai bằng một, và do đó là hợp lý để tin rằng nó sẽ hội tụ đến một phân phối giới hạn nào đó. Nếu minh họa phân phối này, ta có thể kiểm chứng giả thuyết trên.

```

p = 0.2
ns = [1, 10, 100, 1000]
d2l.plt.figure(figsize=(10, 3))
for i in range(4):
    n = ns[i]
    pmf = np.array([p**i * (1-p)**(n-i) * binom(n, i) for i in range(n + 1)])

```

(continues on next page)

```

d2l=plt.subplot(1, 4, i + 1)
d2l=plt.stem([(i - n*p)/np.sqrt(n*p*(1 - p)) for i in range(n + 1)], pmf,
             use_line_collection=True)
d2l=plt.xlim([-4, 4])
d2l=plt.xlabel('x')
d2l=plt.ylabel('p.m.f.')
d2l=plt.title("n = {}".format(n))
d2l=plt.show()

```

Một điều cần lưu ý: so với phân phối Poisson, ta đang chia cho độ lệch chuẩn, có nghĩa là ta đang ép các kết quả có thể xảy ra vào các vùng ngày càng nhỏ hơn. Đây là một dấu hiệu cho thấy giới hạn này sẽ không còn rời rạc mà trở nên liên tục.

Trình bày đầy đủ cách suy ra kết quả cuối cùng nằm ngoài phạm vi của tài liệu này, nhưng *định lý giới hạn trung tâm - central limit theorem* phát biểu rằng khi  $n \rightarrow \infty$ , giới hạn này sẽ tiến tới Phân phối Gauss (hoặc tên khác là phân phối chuẩn). Tường minh hơn, với bất kỳ  $a, b$  nào:

$$\lim_{n \rightarrow \infty} P(Y^{(n)} \in [a, b]) = P(\mathcal{N}(0, 1) \in [a, b]), \quad (20.8.16)$$

trong đó, một biến ngẫu nhiên  $X$  tuân theo phân phối chuẩn với kỳ vọng  $\mu$  và phương sai  $\sigma^2$ , ký hiệu  $X \sim \mathcal{N}(\mu, \sigma^2)$  nếu nó có mật độ:

$$p_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \quad (20.8.17)$$

Đầu tiên hãy vẽ đồ thị của hàm mật độ xác suất (20.8.17).

```

mu, sigma = 0, 1

x = np.arange(-3, 3, 0.01)
p = 1 / np.sqrt(2 * np.pi * sigma**2) * np.exp(-(x - mu)**2 / (2 * sigma**2))

d2l.plot(x, p, 'x', 'p.d.f.')

```

Giờ hãy vẽ đồ thị hàm phân phối tích luỹ. Tuy nằm ngoài phạm vi của phụ lục này nhưng hàm phân phối tích luỹ của phân phối Gauss không có công thức dạng đóng dựa trên các hàm số sơ cấp. Ta sẽ sử dụng erf để tính toán xấp xỉ tích phân này.

```

def phi(x):
    return (1.0 + erf((x - mu) / (sigma * np.sqrt(2)))) / 2.0

d2l.plot(x, np.array([phi(y) for y in x.tolist()]), 'x', 'c.d.f.')

```

Những bạn đọc tinh ý sẽ nhận ra một vài số hạng ở đây. Quả thực, ta đã gấp tích phân này trong Section 20.5. Và ta cần chính phép tính này để xem liệu  $p_X(x)$  có tổng diện tích bằng một và do đó là một hàm mật độ hợp lệ.

Không có một lý do cơ sở nào để ta chọn mô tả bài toán bằng việc tung đồng xu ngoài việc nó giúp quá trình tính toán ngắn hơn. Thật vậy, nếu lấy bất kỳ tập các biến ngẫu nhiên độc lập có cùng phân phối  $X_i$  nào, và gọi:

$$X^{(N)} = \sum_{i=1}^N X_i. \quad (20.8.18)$$

Thì

$$\frac{X^{(N)} - \mu_{X^{(N)}}}{\sigma_{X^{(N)}}} \quad (20.8.19)$$

sẽ xấp xỉ phân phối Gauss. Ta sẽ cần thêm vài điều kiện bổ sung, phổ biến nhất là  $E[X^4] < \infty$ , nhưng ý tưởng cốt lõi đã rõ ràng.

Định lý giới hạn trung tâm là lý do mà phân phối Gauss là nền tảng của xác suất, thống kê, và học máy. Mỗi khi ta có thể nói rằng thứ gì đó ta đo được là tổng của nhiều phần nhỏ độc lập, ta có thể giả sử rằng thứ được đo sẽ gần với phân phối Gauss.

Có rất nhiều tính chất hấp dẫn khác của phân phối Gauss, và chúng tôi muốn thảo luận thêm một tính chất nữa ở đây. Phân phối Gauss được biết tới là *phân phối entropy cực đại*. Ta sẽ phân tích entropy sâu hơn trong [Section 20.11](#), tuy nhiên lúc này chỉ cần biết nó là một phép đo sự ngẫu nhiên. Theo nghĩa toán học một cách chặt chẽ, ta có thể hiểu phân phối Gauss là cách chọn ngẫu nhiên *nhiết* với kỳ vọng và phương sai cố định. Do đó, nếu ta biết biến ngẫu nhiên có kỳ vọng và phương sai nào đó, về trực giác phân phối Gauss là lựa chọn an toàn nhất trong những phân phối mà ta có thể chọn.

Để kết lại phần này, hãy nhớ lại rằng nếu  $X \sim \mathcal{N}(\mu, \sigma^2)$ , thì:

- $\mu_X = \mu$ ,
- $\sigma_X^2 = \sigma^2$ .

Ta có thể lấy mẫu từ phân phối Gauss (chuẩn tắc) như mô tả dưới.

```
np.random.normal(mu, sigma, size=(10, 10))
```

### 20.8.7 Họ hàm Mũ

Một tính chất chung của tất cả các phân phối liệt kê ở trên là chúng đều thuộc họ được gọi là *họ hàm mũ* (*exponential family*). Họ hàm mũ là tập các phân phối có mật độ được biểu diễn dưới dạng sau:

$$p(\mathbf{x}|\eta) = h(\mathbf{x}) \cdot \exp(\eta^\top \cdot T(x) - A(\eta)) \quad (20.8.20)$$

Định nghĩa này có vài điểm khá tinh tế nên hãy cùng xem xét kĩ lưỡng hơn.

Đầu tiên,  $h(\mathbf{x})$  được gọi là *phép đo cơ bản* (*underlying measure*) hay *phép đo cơ sở* (*base measure*). Đây có thể được coi là thang đo ban đầu mà chúng ta đang biến đổi khi điều chỉnh trọng số mũ.

Thứ hai, ta có vector  $\eta = (\eta_1, \eta_2, \dots, \eta_l) \in \mathbb{R}^l$  được gọi là *tham số tự nhiên* (*natural parameters*) hay *tham số chính tắc* (*canonical parameters*). Các vector này xác định phép đo cơ sở sẽ được điều chỉnh thế nào. Ta tiến hành phép đo mới bằng cách tính tích vô hướng của các tham số tự nhiên với hàm  $T(\cdot)$  nào đó của  $\mathbf{x} = (x_1, x_2, \dots, x_n) \in \mathbb{R}^n$  và lấy luỹ thừa.  $T(\mathbf{x}) = (T_1(\mathbf{x}), T_2(\mathbf{x}), \dots, T_l(\mathbf{x}))$  được gọi là *thống kê đầy đủ* (*sufficient statistics*) của  $\eta$ , do thông tin biểu diễn bởi  $T(\mathbf{x})$  là đủ để tính mật độ xác suất và không cần thêm bất cứ thông tin nào khác từ mẫu  $\mathbf{x}$ .

Thứ ba, ta có  $A(\eta)$ , được gọi là *hàm tích luỹ* (*cumulant function*), hàm này đảm bảo phân phối trên [\(20.8.20\)](#) có tích phân bằng 1, và có dạng:

$$A(\eta) = \log \left[ \int h(\mathbf{x}) \cdot \exp(\eta^\top \cdot T(x)) dx \right]. \quad (20.8.21)$$

Để ngắn gọn, ta xét phân phối Gauss. Giả sử  $\mathbf{x}$  là đơn biến (*univariate variable*) và có mật độ là:

$$\begin{aligned} p(x|\mu, \sigma) &= \frac{1}{\sqrt{2\pi}\sigma^2} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \\ &= \frac{1}{\sqrt{2\pi}} \cdot \exp\left\{\frac{\mu}{\sigma^2}x - \frac{1}{2\sigma^2}x^2 - \left(\frac{1}{2\sigma^2}\mu^2 + \log(\sigma)\right)\right\}. \end{aligned} \quad (20.8.22)$$

Hàm này phù hợp với định nghĩa của họ hàm mũ với:

- *phép đo cơ sở*:  $h(x) = \frac{1}{\sqrt{2\pi}}$ ,
- *tham số tự nhiên*:  $\eta = \begin{bmatrix} \eta_1 \\ \eta_2 \end{bmatrix} = \begin{bmatrix} \frac{\mu}{\sigma^2} \\ \frac{1}{2\sigma^2} \end{bmatrix}$ ,
- *thống kê đầy đủ*:  $T(x) = \begin{bmatrix} x \\ -x^2 \end{bmatrix}$ , và
- *hàm tích luỹ*:  $A(\eta) = \frac{1}{2\sigma^2}\mu^2 + \log(\sigma) = \frac{\eta_1^2}{4\eta_2} - \frac{1}{2}\log(2\eta_2)$ .

Đáng chú ý rằng việc lựa chọn chính xác từng số hạng trên hơi có phần tuỳ ý. Quả thật, đặc trưng quan trọng nhất chính là việc phân phối có thể được biểu diễn ở dạng này, chứ không cần bất kỳ dạng chính xác nào.

Như đề cập trong subsec\_softmax\_and\_derivatives, một kỹ thuật hay dùng là giả sử kết quả cuối cùng y tuân theo họ phân phối mũ. Họ hàm mũ là một họ phân phối phổ biến và mạnh mẽ, bắt gặp thường xuyên trong học máy.

### 20.8.8 Tóm tắt

- Phân phối Bernoulli có thể mô hình hóa sự kiện có kết quả có/không.
- Phân phối đều rời rạc chọn từ một tập hữu hạn các khả năng.
- Phân phối đều liên tục chọn từ một khoảng liên tục.
- Phân phối nhị thức mô hình hóa một chuỗi các biến Bernoulli ngẫu nhiên, và đếm số kết quả.
- Phân phối Poisson mô hình hóa các sự kiện hiếm khi xuất hiện.
- Phân phối Gauss mô hình hóa kết quả của việc tính tổng một lượng lớn các biến ngẫu nhiên độc lập.
- Tất cả các phân phối trên đều thuộc họ hàm mũ.

### 20.8.9 Bài tập

1. Tính độ lệch chuẩn của một biến ngẫu nhiên mô tả hiệu  $X - Y$  của hai biến ngẫu nhiên nhị thức độc lập  $X, Y \sim \text{Binomial}(16, 1/2)$ .
2. Nếu ta lấy một biến ngẫu nhiên Poisson  $X \sim \text{Poisson}(\lambda)$  và xét  $(X - \lambda)/\sqrt{\lambda}$  với  $\lambda \rightarrow \infty$ , ta có thể chỉ ra rằng phân phối này xấp xỉ phân phối Gauss. Tại sao điều này lại hợp lý?
3. Hàm khối xác suất của tổng của hai biến ngẫu nhiên rời rạc theo phân phối đều trên  $n$  phần tử là gì?

### 20.8.10 Thảo luận

- Tiếng Anh: MXNet<sup>429</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>430</sup>

### 20.8.11 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Mai Hoàng Long
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Phạm Hồng Vinh
- Đỗ Trường Giang
- Nguyễn Văn Cường

## 20.9 Bộ phân loại Naive Bayes

Trong các phần trước, ta đã học lý thuyết về xác suất và biến ngẫu nhiên. Để áp dụng lý thuyết này, ta sẽ lấy một ví dụ sử dụng bộ phân loại *naive Bayes* cho bài toán phân loại chữ số. Phương pháp này không sử dụng bất kỳ điều gì khác ngoài các lý thuyết căn bản về xác suất.

Quá trình học hoàn toàn xoay quanh việc đưa ra các giả định. Nếu muốn phân loại một mẫu dữ liệu mới chưa thấy bao giờ, ta cần phải đưa ra một giả định nào đó về sự tương đồng giữa các mẫu dữ liệu. Bộ phân loại Naive Bayes, một thuật toán thông dụng và dễ hiểu, giả định rằng tất cả các đặc trưng đều độc lập với nhau nhằm đơn giản hóa việc tính toán. Trong phần này, chúng tôi sẽ sử dụng mô hình này để nhận dạng ký tự trong ảnh.

```
%matplotlib inline
from d2l import mxnet as d2l
import math
from mxnet import gluon, np, npx
npx.set_np()
d2l.use_svg_display()
```

<sup>429</sup> <https://discuss.d2l.ai/t/417>

<sup>430</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 20.9.1 Nhận diện Ký tự Quang học

MNIST (LeCun et al., 1998) là một trong những tập dữ liệu được sử dụng rộng rãi. Nó chứa 60.000 ảnh để huấn luyện và 10.000 ảnh để kiểm định. Mỗi ảnh chứa một chữ số viết tay từ 0 đến 9. Nhiệm vụ là phân loại từng ảnh theo chữ số tương ứng.

Gluon cung cấp lớp MNIST trong mô-đun `data.vision` để tự động lấy dữ liệu từ Internet. Sau đó, Gluon sẽ sử dụng dữ liệu đã được tải xuống. Chúng ta xác định rằng ta đang yêu cầu tập huấn luyện hay tập kiểm tra bằng cách đặt giá trị tham số `train=True` hoặc `False` tương ứng. Mỗi hình ảnh là một ảnh xám có cả chiều rộng và chiều cao là 28, kích thước (28,28,1). Ta sẽ sử dụng một phép biến đổi được tùy chỉnh để loại bỏ chiều của kênh cuối cùng. Ngoài ra, tập dữ liệu biểu diễn mỗi điểm ảnh bằng một số nguyên 8-bit không âm. Ta lượng tử hóa (*quantize*) chúng thành các đặc trưng nhị phân để đơn giản hóa bài toán.

```
def transform(data, label):
    return np.floor(data.astype('float32') / 128).squeeze(axis=-1), label

mnist_train = gluon.data.vision.MNIST(train=True, transform=transform)
mnist_test = gluon.data.vision.MNIST(train=False, transform=transform)
```

Ta có thể truy cập vào từng mẫu cụ thể có chứa ảnh và nhãn tương ứng.

```
image, label = mnist_train[2]
image.shape, label
```

Mẫu được lưu trữ trong biến `image` trên tương ứng với một ảnh có chiều cao và chiều rộng là 28 pixel.

```
image.shape, image.dtype
```

Đoạn mã lưu nhãn của từng ảnh dưới dạng số nguyên 32-bit.

```
label, type(label), label.dtype
```

Ta cũng có thể truy cập vào nhiều mẫu cùng một lúc.

```
images, labels = mnist_train[10:38]
images.shape, labels.shape
```

Hãy cùng minh họa các mẫu trên.

```
d2l.show_images(images, 2, 9);
```

## 20.9.2 Mô hình xác suất để Phân loại

Trong tác vụ phân loại, ta ánh xạ một mẫu tới một hạng mục. Ví dụ, ở đây ta ánh xạ một ảnh xám kích thước  $28 \times 28$  tới hạng mục là một chữ số. (Tham khảo Section 5.4 để xem giải thích chi tiết hơn.) Một cách diễn đạt tự nhiên về tác vụ phân loại là câu hỏi xác suất: nhãn nào là hợp lý nhất với các đặc trưng cho trước (tức là các pixel trong ảnh)? Ký hiệu  $\mathbf{x} \in \mathbb{R}^d$  là các đặc trưng và  $y \in \mathbb{R}$  là nhãn của một mẫu. Đặc trưng ở đây là các pixel trong ảnh 2 chiều mà ta có thể biến đổi thành vector kích thước  $d = 28^2 = 784$ , và nhãn là các chữ số. Xác suất của nhãn khi biết trước đặc trưng là  $p(y | \mathbf{x})$ . Trong ví dụ của ta, nếu có thể tính toán các xác suất  $p(y | \mathbf{x})$  với  $y = 0, \dots, 9$ , bộ phân loại sẽ đưa ra dự đoán  $\hat{y}$  theo công thức:

$$\hat{y} = \operatorname{argmax}_y p(y | \mathbf{x}). \quad (20.9.1)$$

Không may là việc này yêu cầu ước lượng  $p(y | \mathbf{x})$  cho mọi giá trị  $\mathbf{x} = x_1, \dots, x_d$ . Hãy tưởng tượng mỗi đặc trưng nhận một giá trị nhị phân, ví dụ, đặc trưng  $x_1 = 1$  cho biết từ “quả táo” xuất hiện trong văn bản cho trước và  $x_1 = 0$  biểu thị ngược lại. Nếu có 30 đặc trưng nhị phân như vậy, ta cần phân loại  $2^{30}$  (hơn 1 tỷ!) vector đầu vào khả dĩ của  $\mathbf{x}$ .

Hơn nữa, như vậy không phải là học. Nếu cần xem qua toàn bộ các ví dụ khả dĩ để dự đoán nhãn tương ứng thì chúng ta không thực sự đang học một khuôn mẫu nào mà chỉ là đang ghi nhớ tập dữ liệu.

## 20.9.3 Bộ phân loại Naive Bayes

May mắn thay, bằng cách đưa ra một số giả định về tính độc lập có điều kiện, ta có thể đưa vào một số thiên kiến quy nạp và xây dựng một mô hình có khả năng tổng quát hóa từ một nhóm các mẫu huấn luyện với kích thước tương đối khiêm tốn. Để bắt đầu, hãy sử dụng định lý Bayes để biểu diễn bộ phân loại bằng biểu thức sau

$$\hat{y} = \operatorname{argmax}_y p(y | \mathbf{x}) = \operatorname{argmax}_y \frac{p(\mathbf{x} | y)p(y)}{p(\mathbf{x})}. \quad (20.9.2)$$

Lưu ý rằng mẫu số là số hạng chuẩn hóa  $p(\mathbf{x})$  không phụ thuộc vào giá trị của nhãn  $y$ . Do đó, chúng ta chỉ cần quan tâm tới việc so sánh tử số giữa các giá trị  $y$  khác nhau. Ngay cả khi việc tính toán mẫu số hóa ra là không thể, ta cũng có thể bỏ qua nó, miễn là ta có thể tính được tử số. May mắn thay, ta vẫn có thể khôi phục lại hằng số chuẩn hóa nếu muốn. Ta luôn có thể khôi phục số hạng chuẩn hóa vì  $\sum_y p(y | \mathbf{x}) = 1$ .

Bây giờ, hãy tập trung vào biểu thức  $p(\mathbf{x} | y)$ . Sử dụng quy tắc dây chuyền cho xác suất, chúng ta có thể biểu diễn số hạng  $p(\mathbf{x} | y)$  dưới dạng

$$p(x_1 | y) \cdot p(x_2 | x_1, y) \cdot \dots \cdot p(x_d | x_1, \dots, x_{d-1}, y). \quad (20.9.3)$$

Biểu thức này tự nó không giúp ta được thêm điều gì. Ta vẫn phải ước lượng khoảng  $2^d$  các tham số. Tuy nhiên, nếu chúng ta giả định rằng *các đặc trưng khi biết nhãn cho trước là độc lập với nhau*, thì số hạng này đơn giản hóa thành  $\prod_i p(x_i | y)$ , và ta có hàm dự đoán:

$$\hat{y} = \operatorname{argmax}_y \prod_{i=1}^d p(x_i | y)p(y). \quad (20.9.4)$$

Ta có thể ước lượng  $\prod_i p(x_i = 1 | y)$  với mỗi  $i$  và  $y$ , và lưu giá trị của nó trong  $P_{xy}[i, y]$ , ở đây  $P_{xy}$  là một ma trận có kích thước  $d \times n$  với  $n$  là số lượng các lớp và  $y \in \{1, \dots, n\}$ . Cùng với đó, ta ước lượng  $p(y)$  cho mỗi  $y$  và lưu trong  $P_y[y]$ , với  $P_y$  là một vector có độ dài  $n$ . Sau đó, đối với bất kỳ mẫu mới  $\mathbf{x}$  nào, ta có thể tính:

$$\hat{y} = \operatorname{argmax}_y \prod_{i=1}^d P_{xy}[x_i, y] P_y[y], \quad (20.9.5)$$

cho  $y$  bất kỳ. Vì vậy, giả định của chúng ta về sự độc lập có điều kiện đã làm giảm độ phức tạp của mô hình từ phụ thuộc theo cấp số nhân vào số lượng các đặc trưng  $\mathcal{O}(2^d n)$  thành phụ thuộc tuyến tính, tức là  $\mathcal{O}(dn)$ .

#### 20.9.4 Huấn luyện

Vấn đề bây giờ là ta không biết  $P_{xy}$  và  $P_y$ . Vì vậy, ta trước tiên cần ước lượng giá trị của chúng với dữ liệu huấn luyện. Đây là việc *huấn luyện* mô hình. Ước lượng  $P_y$  không quá khó. Do chỉ đang làm việc với 10 lớp, ta có thể đếm số lần xuất hiện  $n_y$  của mỗi chữ số và chia nó cho tổng số dữ liệu  $n$ . Chẳng hạn, nếu chữ số 8 xuất hiện  $n_8 = 5,800$  lần và ta có tổng số hình ảnh là  $n = 60.000$ , xác suất ước lượng sẽ là  $p(y = 8) = 0.0967$ .

```
X, Y = mnist_train[:, :] # All training examples

n_y = np.zeros(10)
for y in range(10):
    n_y[y] = (Y == y).sum()
P_y = n_y / n_y.sum()
P_y
```

Giờ hãy chuyển sang vấn đề khó hơn một chút là tính  $P_{xy}$ . Vì ta lấy các ảnh đen trắng,  $p(x_i | y)$  biểu thị xác suất điểm ảnh  $i$  mang nhãn  $y$ . Đơn giản giống như trên, ta có thể duyệt và đếm số lần  $n_{iy}$  mà điểm ảnh  $i$  mang nhãn  $y$  và chia nó cho tổng số lần xuất hiện  $n_y$  của  $y$ . Nhưng có một điểm hơi rắc rối: một số điểm ảnh nhất định có thể không bao giờ có màu đen (ví dụ, đối với các ảnh được cắt xén tốt, các điểm ảnh ở góc có thể luôn là màu trắng). Một cách thuận tiện cho các nhà thống kê học để giải quyết vấn đề này là cộng thêm một số đếm giả vào tất cả các lần xuất hiện. Do đó, thay vì  $n_{iy}$ , ta dùng  $n_{iy} + 1$  và thay vì  $n_y$ , ta dùng  $n_y + 1$ . Phương pháp này còn được gọi là *Làm mượt Laplace (Laplace Smoothing)*, có vẻ không chính thống nhưng hợp với quan điểm Bayes.

```
n_x = np.zeros((10, 28, 28))
for y in range(10):
    n_x[y] = np.array(X.asnumpy()[(Y.asnumpy() == 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 trực quan hóa các xác suất  $10 \times 28 \times 28$  này (cho mỗi điểm ảnh đối với mỗi lớp), ta có thể lấy được hình ảnh trung bình của các chữ số.

Giờ ta có thể sử dụng (20.9.5) để dự đoán một hình ảnh mới. Cho  $\mathbf{x}$ , hàm sau sẽ tính  $p(\mathbf{x} | y)p(y)$  với mỗi  $y$ .

```

def bayes_pred(x):
    x = np.expand_dims(x, axis=0) # (28, 28) -> (1, 28, 28)
    p_xy = P_xy * x + (1 - P_xy)*(1 - x)
    p_xy = p_xy.reshape(10, -1).prod(axis=1) # p(x|y)
    return np.array(p_xy) * P_y

image, label = mnist_test[0]
bayes_pred(image)

```

Điều này đã dẫn tới sai lầm khủng khiếp! Để tìm hiểu lý do tại sao, ta hãy xem xét xác suất trên mỗi điểm ảnh. Chúng thường mang giá trị từ 0.001 đến 1 và ta đang nhân 784 con số như vậy. Ta đang tính những con số này trên máy tính, do đó sẽ có một phạm vi cố định cho số mũ. Ta đã gặp vấn đề *tràn số dưới* (*underflow*), tức là tích tất cả các số nhỏ hơn một sẽ dẫn đến một số nhỏ dần cho đến khi kết quả được làm tròn thành 0. Ta đã thảo luận lý thuyết về vấn đề này trong Section 20.7, và ở đây ta thấy hiện tượng này trong thực tế một cách rõ ràng.

Như đã thảo luận trong phần đó, ta khắc phục điều này bằng cách sử dụng tính chất  $\log ab = \log a + \log b$ , cụ thể là ta chuyển sang tính tổng các logarit. Nhờ vậy ngay cả khi cả  $a$  và  $b$  đều là các số nhỏ, giá trị các logarit vẫn sẽ nằm trong miền thích hợp.

```

a = 0.1
print('underflow:', a**784)
print('logarithm is normal:', 784*math.log(a))

```

Vì logarit là một hàm tăng dần, ta có thể viết lại (20.9.5) thành

$$\hat{y} = \operatorname{argmax}_y \sum_{i=1}^d \log P_{xy}[x_i, y] + \log P_y[y]. \quad (20.9.6)$$

Ta có thể lập trình phiên bản ổn định sau:

```

log_P_xy = np.log(P_xy)
log_P_xy_neg = np.log(1 - P_xy)
log_P_y = np.log(P_y)

def bayes_pred_stable(x):
    x = np.expand_dims(x, axis=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

```

Bây giờ ta có thể kiểm tra liệu dự đoán này có đúng hay không.

```

# Convert label which is a scalar tensor of int32 dtype
# to a Python scalar integer for comparison
py.argmax(axis=0) == int(label)

```

Nếu dự đoán một vài mẫu kiểm định, ta có thể thấy bộ phân loại Bayes hoạt động khá tốt.

```

def predict(X):
    return [bayes_pred_stable(x).argmax(axis=0).astype(np.int32) for x in X]

X, y = mnist_test[:18]
preds = predict(X)
d2l.show_images(X, 2, 9, titles=[str(d) for d in preds]);

```

Cuối cùng, hãy cùng tính toán độ chính xác tổng thể của bộ phân loại.

```

X, y = mnist_test[:]
preds = np.array(predict(X), dtype=np.int32)
float((preds == y).sum() / len(y)) # Validation accuracy

```

Các mạng sâu hiện đại đạt tỷ lệ lỗi dưới 0,01. Hiệu suất tương đối kém ở đây là do các giả định thống kê không chính xác mà ta đã đưa vào trong mô hình: ta đã giả định rằng mỗi và mọi pixel được tạo *một cách độc lập*, chỉ phụ thuộc vào nhãn. Đây rõ ràng không phải là cách con người viết các chữ số, và giả định sai lầm này đã dẫn đến sự kém hiệu quả của bộ phân loại ngây thơ (*naive Bayes*) của chúng ta.

### 20.9.5 Tóm tắt

- Sử dụng quy tắc Bayes, một bộ phân loại có thể được tạo ra bằng cách giả định tất cả các đặc trưng quan sát được là độc lập.
- Bộ phân loại này có thể được huấn luyện trên tập dữ liệu bằng cách đếm số lần xuất hiện của các tổ hợp nhãn và giá trị điểm ảnh.
- Bộ phân loại này là tiêu chuẩn vàng trong nhiều thập kỷ cho các tác vụ như phát hiện thư rác.

### 20.9.6 Bài tập

1. Xem xét tập dữ liệu  $[[0, 0], [0, 1], [1, 0], [1, 1]]$  với các nhãn tương ứng là kết quả phép XOR của cặp số trong mỗi mẫu, tức  $[0, 1, 1, 0]$ . Các xác suất cho bộ phân loại Naive Bayes được xây dựng trên tập dữ liệu này là bao nhiêu? Nó có phân loại thành công các điểm dữ liệu không? Nếu không, những giả định nào bị vi phạm?
2. Giả sử ta không sử dụng làm mượt Laplace khi ước lượng xác suất và có một mẫu dữ liệu tại thời điểm kiểm tra chứa một giá trị chưa bao giờ được quan sát trong quá trình huấn luyện. Lúc này mô hình sẽ trả về giá trị gì?
3. Bộ phân loại Naive Bayes là một ví dụ cụ thể của mạng Bayes, trong đó sự phụ thuộc của các biến ngẫu nhiên được mã hóa bằng cấu trúc đồ thị. Mặc dù lý thuyết đầy đủ nằm ngoài phạm vi của phần này (xem (Koller & Friedman, 2009) để biết chi tiết), hãy giải thích tại sao việc đưa sự phụ thuộc tường minh giữa hai biến đầu vào trong mô hình XOR lại có thể tạo ra một bộ phân loại thành công.

### 20.9.7 Thảo luận

- Tiếng Anh: MXNet<sup>431</sup>, Pytorch<sup>432</sup>, Tensorflow<sup>433</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>434</sup>

### 20.9.8 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Lê Khắc Hồng Phúc
- Phạm Minh Đức
- Trần Yến Thy
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Đỗ Trường Giang
- Nguyễn Mai Hoàng Long

## 20.10 Thống kê

Để trở thành chuyên gia Học sâu hàng đầu, điều kiện tiên quyết cần có là khả năng huấn luyện các mô hình hiện đại với độ chính xác cao. Tuy nhiên, thường khó có thể biết được những cải tiến trong mô hình là đáng kể, hay chúng chỉ là kết quả của những biến động ngẫu nhiên trong quá trình huấn luyện. Để có thể thảo luận về tính bất định trong các giá trị ước lượng, chúng ta cần có hiểu biết về thống kê.

Tài liệu tham khảo đầu tiên về *thống kê* có thể được truy ngược về học giả người Ả Rập Al-Kindi từ thế kỉ thứ chín. Ông đã đưa ra những mô tả chi tiết về cách sử dụng thống kê và phân tích tần suất để giải mã những thông điệp mã hóa. Sau 800 năm, thống kê hiện đại trỗi dậy ở Đức vào những năm 1700, khi các nhà nghiên cứu tập trung vào việc thu thập và phân tích các dữ liệu nhân khẩu học và kinh tế. Hiện nay, khoa học thống kê quan tâm đến việc thu thập, xử lý, phân tích, diễn giải và biểu diễn dữ liệu. Hơn nữa, lý thuyết cốt lõi của thống kê đã được sử dụng rộng rãi cho nghiên cứu trong giới học thuật, doanh nghiệp và chính phủ.

Cụ thể hơn, thống kê có thể được chia thành *thống kê mô tả* (*descriptive statistic*) và *suy luận thống kê* (*statistical inference*). Thống kê mô tả đặt trọng tâm vào việc tóm tắt và minh họa những đặc trưng của một tập hợp những dữ liệu đã được quan sát - được gọi là *mẫu*. Mẫu được lấy ra từ một *tổng thể* (*population*), là biểu diễn của toàn bộ những cá thể, đồ vật hay sự kiện tương tự nhau mà thí nghiệm của ta quan tâm. Trái với thống kê mô tả, *suy luận thống kê* (*statistical inference*) dự đoán những đặc điểm của một tổng thể qua những *mẫu* có sẵn, dựa theo giả định phân phối mẫu là một biểu diễn tương đối hợp lý của phân phối tổng thể.

<sup>431</sup> <https://discuss.d2l.ai/t/418>

<sup>432</sup> <https://discuss.d2l.ai/t/1100>

<sup>433</sup> <https://discuss.d2l.ai/t/1101>

<sup>434</sup> <https://forum.machinelearningcoban.com/c/d2l>

Bạn có thể tự hỏi: “Sự khác biệt cơ bản giữa học máy và thống kê là gì?”. Về căn bản, thống kê tập trung vào các vấn đề suy luận. Những vấn đề này bao gồm mô hình hóa mối quan hệ giữa các biến, ví dụ như suy luận nguyên nhân hoặc kiểm tra ý nghĩa thống kê của các tham số mô hình, ví dụ như phép thử A/B. Ngược lại, học máy đề cao việc dự đoán chính xác mà không yêu cầu lập trình một cách tường minh và hiểu rõ chức năng của từng tham số.

Trong chương này, chúng tôi sẽ giới thiệu ba loại suy luận thống kê: đánh giá và so sánh các bộ ước lượng, tiến hành kiểm định giả thuyết và xây dựng khoảng tin cậy. Các phương pháp này có thể giúp chúng ta suy luận những đặc tính của một tổng thể, hay nói cách khác, tham số thực  $\theta$ . Nói ngắn gọn, chúng tôi giả sử tham số thực  $\theta$  của một tổng thể cho trước là một số vô hướng. Việc mở rộng ra các trường hợp  $\theta$  là một vector hoặc tensor là khá đơn giản nên chúng tôi sẽ không đề cập ở đây.

### 20.10.1 Đánh giá và So sánh các Bộ ước lượng

Trong thống kê, một *bộ ước lượng* là một hàm sử dụng những mẫu có sẵn để ước lượng giá trị thực của tham số  $\theta$ . Ta gọi  $\hat{\theta}_n = \hat{f}(x_1, \dots, x_n)$  là ước lượng của  $\theta$  sau khi quan sát các mẫu  $\{x_1, x_2, \dots, x_n\}$ .

Ta đã thấy nhiều ví dụ đơn giản của bộ ước lượng trong phần Section 20.7. Nếu bạn có một số mẫu ngẫu nhiên từ phân phối Bernoulli, thì ước lượng hợp lý cực đại (*maximum likelihood estimate*) cho xác xuất của biến ngẫu nhiên có thể được tính bằng cách đếm số lần biến cố xuất hiện rồi chia cho tổng số mẫu. Tương tự, đã có một bài tập yêu cầu bạn chứng minh rằng ước lượng hợp lý cực đại của kỳ vọng phân phối Gauss với một số lượng mẫu cho trước là giá trị trung bình của tập mẫu đó. Các bộ ước lượng này thường như sẽ không bao giờ cho ra giá trị chính xác của tham số, nhưng với số lượng mẫu đủ lớn, ước lượng có được sẽ gần với giá trị thực.

Xét ví dụ sau, chúng tôi biểu diễn mật độ của phân phối Gauss với kỳ vọng là không và phương sai là một, cùng với một tập các mẫu lấy ra từ phân phối đó. Tọa độ  $y$  được xây dựng sao cho tất cả điểm đều có thể nhìn thấy được và mối quan hệ giữa mật độ mẫu và mật độ gốc của phân phối có thể được nhìn thấy rõ hơn.

```
from d2l import mxnet as d2l
from mxnet import np, npx
import random
npx.set_np()
# Sample datapoints and create y coordinate
epsilon = 0.1
random.seed(8675309)
xs = np.random.normal(loc=0, scale=1, size=(300,))
ys = [np.sum(np.exp(-(xs[:i] - xs[i])**2 / (2 * epsilon**2)) /
            np.sqrt(2*np.pi*epsilon**2)) / len(xs) for i in range(len(xs))]
# Compute true density
xd = np.arange(np.min(xs), np.max(xs), 0.01)
yd = np.exp(-xd**2/2) / np.sqrt(2 * np.pi)
# Plot the results
d2l.plot(xd, yd, 'x', 'density')
d2l.plt.scatter(xs, ys)
d2l.plt.axvline(x=0)
d2l.plt.axvline(x=np.mean(xs), linestyle='--', color='purple')
d2l.plt.title(f'sample mean: {float(np.mean(xs)):.2f}')
d2l.plt.show()
```

Có nhiều cách để tính toán một bộ ước lượng cho một tham số  $\hat{\theta}_n$ . Trong phần này, ta sẽ đi qua ba phương thức phổ biến để đánh giá và so sánh các bộ ước lượng: trung bình bình phương sai số,

độ lệch chuẩn và độ chêch thống kê.

### Trung bình Bình phương Sai số

Có lẽ phép đo đơn giản nhất được sử dụng để đánh giá bộ ước lượng là *trung bình bình phương sai số* (*mean squared error – MSE*) (hay *mất mát* :math: 'l\_2'). Trung bình bình phương sai số của một bộ ước lượng được định nghĩa

$$\text{MSE}(\hat{\theta}_n, \theta) = E[(\hat{\theta}_n - \theta)^2]. \quad (20.10.1)$$

Phương pháp này cho phép ta định lượng trung bình bình phương độ lệch so với giá trị thực. MSE là một đại lượng không âm. Nếu đã đọc [Section 5.1](#), bạn sẽ nhận ra đây là hàm mất mát được sử dụng phổ biến nhất trong bài toán hồi quy. Như một phép đo để đánh giá bộ ước lượng, giá trị của nó càng gần không thì bộ ước lượng càng gần với tham số thực  $\theta$ .

### Độ chêch Thống kê

MSE cung cấp một phép đo tự nhiên, nhưng ta có thể dễ dàng nghĩ tới các trường hợp khác nhau mà ở đó giá trị MSE sẽ lớn. Ta sẽ bàn tới hai trường hợp cơ bản đó là biến động của bộ ước lượng do sự ngẫu nhiên trong bộ dữ liệu, và sai số hệ thống của bộ ước lượng xảy ra trong quá trình ước lượng.

Đầu tiên, ta hãy đo sai số hệ thống. Với một bộ ước lượng  $\hat{\theta}_n$ , biểu diễn toán học của *độ chêch thống kê* được định nghĩa

$$\text{bias}(\hat{\theta}_n) = E(\hat{\theta}_n - \theta) = E(\hat{\theta}_n) - \theta. \quad (20.10.2)$$

Lưu ý rằng khi  $\text{bias}(\hat{\theta}_n) = 0$ , kỳ vọng của bộ ước lượng  $\hat{\theta}_n$  sẽ bằng với giá trị thực của tham số. Trường hợp này, ta nói  $\hat{\theta}_n$  là một bộ ước lượng không thiên lệch. Nhìn chung, một bộ ước lượng không thiên lệch sẽ tốt hơn một bộ ước lượng thiên lệch vì kỳ vọng của nó sẽ bằng với tham số thực.

Tuy nhiên, những bộ ước lượng thiên lệch vẫn thường xuyên được sử dụng trong thực tế.

Có những trường hợp không tồn tại các bộ ước lượng không thiên lệch nếu không có thêm giả định, hoặc rất khó để tính toán. Đây có thể xem như một khuyết điểm lớn trong bộ ước lượng, tuy nhiên phần lớn các bộ ước lượng gặp trong thực tiễn đều ít nhất tiệm cận không thiên lệch theo nghĩa độ chêch có xu hướng tiến về không khi số lượng mẫu có được tiến về vô cực:  $\lim_{n \rightarrow \infty} \text{bias}(\hat{\theta}_n) = 0$ .

### Phương sai và Độ lệch Chuẩn

Tiếp theo, hãy cùng tính độ ngẫu nhiên trong bộ ước lượng. Nhắc lại từ [Section 20.6](#), *độ lệch chuẩn* (*standard deviation*) (còn được gọi là *sai số chuẩn – standard error*) được định nghĩa là căn bậc hai của phương sai. Chúng ta có thể đo được độ dao động của bộ ước lượng bằng cách tính độ lệch chuẩn hoặc phương sai của bộ ước lượng đó.

$$\sigma_{\hat{\theta}_n} = \sqrt{\text{Var}(\hat{\theta}_n)} = \sqrt{E[(\hat{\theta}_n - E(\hat{\theta}_n))^2]}. \quad (20.10.3)$$

So sánh (20.10.3) và (20.10.1) là một việc quan trọng. Trong công thức này, thay vì so sánh với giá trị thực  $\theta$  của tổng thể, chúng ta sử dụng  $E(\hat{\theta}_n)$  là giá trị trung bình mẫu kỳ vọng. Do đó chúng ta không đo độ lệch của bộ ước lượng so với giá trị thực mà là độ dao động của chính bộ ước lượng.

### Sự đánh đổi Độ chêch-Phương sai

Cả hai yếu tố trên rõ ràng đều ảnh hưởng đến trung bình bình phương sai số. Một điều ngạc nhiên là chúng ta có thể chứng minh hai thành phần trên là *phân tách* của trung bình bình phương sai số. Điều này có nghĩa là ta có thể viết trung bình bình phương sai số bằng tổng của phương sai và bình phương độ chêch.

$$\begin{aligned}
 \text{MSE}(\hat{\theta}_n, \theta) &= E[(\hat{\theta}_n - \theta)^2] \\
 &= E[(\hat{\theta}_n)^2] + E[\theta^2] - 2E[\hat{\theta}_n\theta] \\
 &= \text{Var}[\hat{\theta}_n] + E[\hat{\theta}_n]^2 + \text{Var}[\theta] + E[\theta]^2 - 2E[\hat{\theta}_n]E[\theta] \\
 &= (E[\hat{\theta}_n] - E[\theta])^2 + \text{Var}[\hat{\theta}_n] + \text{Var}[\theta] \\
 &= (E[\hat{\theta}_n - \theta])^2 + \text{Var}[\hat{\theta}_n] + \text{Var}[\theta] \\
 &= (\text{bias}[\hat{\theta}_n])^2 + \text{Var}(\hat{\theta}_n) + \text{Var}[\theta].
 \end{aligned} \tag{20.10.4}$$

Chúng tôi gọi công thức trên là *sự đánh đổi độ chêch-phương sai*. Giá trị trung bình bình phương sai số có thể được phân tách chính xác thành ba nguồn sai số khác nhau: sai số từ độ chêch cao, sai số từ phương sai cao và sai số không tránh được (*irreducible error*). Sai số độ chêch thường xuất hiện ở các mô hình đơn giản (ví dụ như hồi quy tuyến tính), vì chúng không thể trích xuất những quan hệ đa chiều giữa các đặc trưng và đầu ra. Nếu một mô hình có độ chêch cao, chúng ta thường nói rằng nó *dưới khớp* (*underfitting*) hoặc là thiếu sự *uyển chuyển* như đã giới thiệu ở (Section 6.4). Ngược lại, một mô hình *quá khớp* (*overfitting*) lại rất nhạy cảm với những dao động nhỏ trong dữ liệu. Nếu một mô hình có phương sai cao, chúng ta thường nói rằng nó *quá khớp* và thiếu *tổng quát hóa* như đã giới thiệu ở (Section 6.4). Sai số không tránh được xuất phát từ nhiều trong chính bản thân  $\theta$ .

### Đánh giá các Bộ ước lượng qua Lập trình

Vì độ lệch chuẩn của bộ ước lượng đã được triển khai trong MXNet bằng cách gọi `a.std()` của đối tượng `ndarray` “`a`”, chúng ta sẽ bỏ qua bước này và thực hiện tính độ chêch thống kê và trung bình bình phương sai số trong MXNet.

```

# Statistical bias
def stat_bias(true_theta, est_theta):
    return(np.mean(est_theta) - true_theta)
# Mean squared error
def mse(data, true_theta):
    return(np.mean(np.square(data - true_theta)))

```

Để minh họa cho phương trình sự đánh đổi độ chêch-phương sai, cùng giả lập một phân phối chuẩn  $\mathcal{N}(\theta, \sigma^2)$  với 10,000 mẫu. Ở đây, ta sử dụng  $\theta = 1$  và  $\sigma = 4$ . Với bộ ước lượng là một hàm số từ các mẫu đã cho, ở đây chúng ta sử dụng trung bình của các mẫu như là bộ ước lượng cho giá trị thực  $\theta$  trong phân phối chuẩn này  $\mathcal{N}(\theta, \sigma^2)$ .

```

theta_true = 1
sigma = 4
sample_len = 10000
samples = np.random.normal(theta_true, sigma, sample_len)
theta_est = np.mean(samples)
theta_est

```

Cùng xác thực phương trình đánh đổi bằng cách tính tổng độ chêch bình phương và phương sai từ bộ ước lượng của chúng ta. Đầu tiên, tính trung bình bình phương sai số của bộ ước lượng:

```
mse(samples, theta_true)
```

Tiếp theo, chúng ta tính  $\text{Var}(\hat{\theta}_n) + [\text{bias}(\hat{\theta}_n)]^2$  như dưới đây. Bạn có thể thấy đại lượng này gần giống với trung bình bình phương sai số đã tính ở trên.

```

bias = stat_bias(theta_true, theta_est)
np.square(samples.std()) + np.square(bias)

```

## 20.10.2 Tiến hành Kiểm định Giả thuyết

Chủ đề thường gặp nhất trong suy luận thống kê là kiểm định giả thuyết. Tuy kiểm định giả thuyết trở nên phổ biến từ đầu thế kỷ 20, trường hợp sử dụng đầu tiên được ghi nhận bởi John Arbuthnot từ tận những năm 1700. John đã theo dõi hồ sơ khai sinh trong 80 năm ở London và kết luận rằng mỗi năm nhiều bé trai được sinh ra hơn so với bé gái. Tiếp đó, phép thử nghiệm độ tin cậy ngày nay là di sản trí tuệ của Karl Pearson, người đã phát minh ra *p-value* (*trị số p*) và bài kiểm định Chi bình phương Pearson (*Pearson's chi-squared test*), William Gosset, cha đẻ của phân phối Student và Ronald Fisher, người đã khởi xướng giả thuyết gốc và kiểm định độ tin cậy.

Một bài *kiểm định giả thuyết* sẽ đánh giá các bằng chứng chống lại mệnh đề mặc định của một tổng thể. Chúng ta gọi các mệnh đề mặc định là *giả thuyết gốc - null hypothesis*  $H_0$ , giả thuyết mà chúng ta cố gắng bác bỏ thông qua các dữ liệu quan sát được. Tại đây, chúng ta sử dụng  $H_0$  là điểm bắt đầu cho việc thử nghiệm độ tin cậy thống kê. *Giả thuyết đối - alternative hypothesis*  $H_A$  (hay  $H_1$ ) là mệnh đề đối lập với giả thuyết gốc. Giả thuyết gốc thường được định nghĩa dưới dạng khai báo mà nó ẩn định mối quan hệ giữa các biến. Nó nên phản ánh mệnh đề một cách rõ ràng nhất, và có thể kiểm chứng được bằng lý thuyết thống kê.

Tưởng tượng bạn là một nhà hóa học. Sau hàng ngàn giờ nghiên cứu trong phòng thí nghiệm, bạn đã phát triển được một loại thuốc mới giúp cải thiện đáng kể khả năng hiểu về toán của con người. Để chứng minh sức mạnh ma thuật của thuốc, bạn cần kiểm tra nó. Thông thường, bạn cần một số tình nguyện viên sử dụng loại thuốc này để kiểm tra xem liệu nó có giúp họ học toán tốt hơn hay không. Bạn sẽ bắt đầu điều này như thế nào?

Đầu tiên, bạn cần cẩn thận lựa chọn ngẫu nhiên hai nhóm tình nguyện viên để đảm bảo rằng không có sự khác biệt đáng kể dựa trên các tiêu chuẩn đo lường được về khả năng hiểu toán của họ. Hai nhóm này thường được gọi là nhóm thử nghiệm và nhóm kiểm soát. *Nhóm thử nghiệm* (hay *nhóm trị liệu*) là nhóm người được cho sử dụng thuốc, trong khi *nhóm kiểm soát* được đặt làm chuẩn so sánh; tức là, họ có các yếu tố môi trường giống hệt với nhóm thử nghiệm trừ việc sử dụng thuốc. Bằng cách này, sự ảnh hưởng của tất cả các biến được giảm thiểu, trừ sự tác động của biến độc lập trong quá trình điều trị.

Thứ hai, sau một thời gian sử dụng thuốc, bạn cần đo khả năng hiểu toán của hai nhóm trên bằng tiêu chuẩn đo lường chung, ví dụ như cho các tình nguyện viên làm cùng một bài kiểm tra sau khi

học một công thức toán mới. Sau đó bạn có thể thu thập kết quả năng lực của họ và so sánh chúng. Trong trường hợp này, giả thuyết gốc của chúng ta đó là không có sự khác biệt nào giữa hai nhóm, và giả thuyết đối là có sự khác biệt.

Quy trình trên vẫn chưa hoàn toàn chính quy. Có rất nhiều chi tiết mà bạn phải suy nghĩ cẩn trọng. Ví dụ, đâu là tiêu chuẩn đo lường thích hợp để kiểm tra khả năng hiểu toán? Bao nhiêu tình nguyện viên thực hiện bài kiểm tra là đủ để bạn có thể tự tin khẳng định sự hiệu quả của thuốc? Bài kiểm tra nên kéo dài trong bao lâu? Làm cách nào để bạn quyết định được có sự khác biệt rõ rệt giữa hai nhóm? Bạn chỉ quan tâm đến kết quả trung bình hay cả phạm vi biến thiên của các điểm số, v.v.

Bằng cách này, kiểm định giả thuyết cung cấp một khuôn khổ cho thiết kế thử nghiệm và cách suy luận về sự chắc chắn của những kết quả quan sát được. Nếu chứng minh được giả thuyết gốc khả năng rất cao là không đúng, thì chúng ta có thể tự tin bác bỏ nó.

Để hiểu rõ hơn về cách làm việc với kiểm định giả thuyết, chúng ta cần bổ sung thêm một số thuật ngữ và toán học hóa các khái niệm ở trên.

### Ý nghĩa thống kê

*Ý nghĩa thống kê* (*statistical significance*) đo xác suất lỗi khi bác bỏ giả thuyết gốc,  $H_0$ , trong khi đúng ra không nên bác bỏ nó.

$$\text{ý nghĩa thống kê} = 1 - \alpha = 1 - P(\text{bác bỏ } H_0 \mid H_0 \text{ là đúng}). \quad (20.10.5)$$

Đây còn được gọi là *lỗi loại I* hay *dương tính giả*.  $\alpha$  ở đây là *mức ý nghĩa* và thường được chọn ở giá trị 5%, tức là  $1 - \alpha = 95\%$ . Mức ý nghĩa thống kê còn có thể hiểu như mức độ rủi ro mà chúng ta chấp nhận khi bác bỏ nhầm một giả thuyết gốc chính xác.

Fig. 20.10.1 thể hiện các giá trị quan sát và xác suất của một phân phối chuẩn trong một bài kiểm định giả thuyết thống kê hai mẫu. Nếu các điểm dữ liệu quan sát nằm ngoài ngưỡng 95%, chúng sẽ rất khó xảy ra dưới giả định của giả thuyết gốc. Do đó, giả thuyết gốc có điều gì đó không đúng và chúng ta sẽ bác bỏ nó.

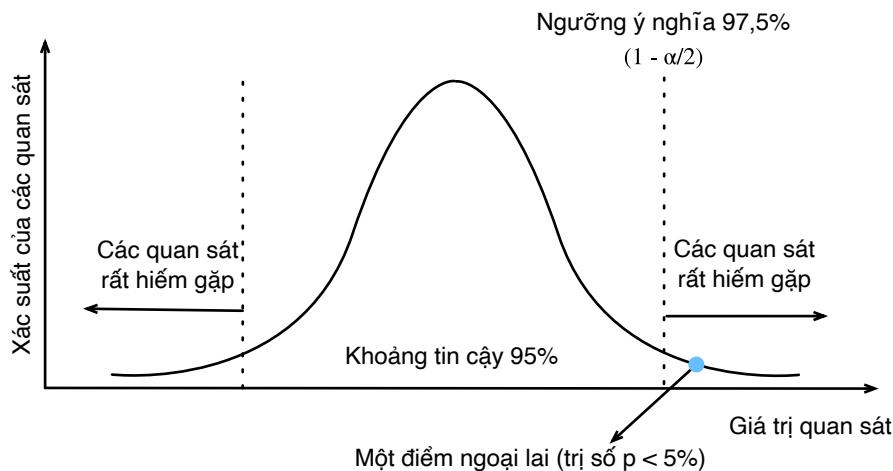


Fig. 20.10.1: Ý nghĩa thống kê.

## Năng lực Thống kê

*Năng lực thống kê* (hay còn gọi là *độ nhạy*) là xác suất bác bỏ giả thuyết gốc,  $H_0$ , biết rằng nó nên bị bác bỏ, tức là:

$$\text{năng lực thống kê} = 1 - \beta = 1 - P(\text{không bác bỏ } H_0 \mid H_0 \text{ là sai}). \quad (20.10.6)$$

Nhớ lại *lỗi loại I* là lỗi do bác bỏ giả thuyết gốc khi nó đúng, còn *lỗi loại II* xảy ra do không bác bỏ giả thuyết gốc khi nó sai. Lỗi loại II thường được kí hiệu là  $\beta$ , vậy nên năng lực thống kê tương ứng là  $1 - \beta$ .

Một cách trực quan, năng lực thống kê có thể được xem như khả năng phép kiểm định phát hiện được một sai lệch thực sự với độ lớn tối thiểu nào đó, ở một mức ý nghĩa thống kê mong muốn. 80% là một ngưỡng năng lực thống kê phổ biến. Năng lực thống kê càng cao, ta càng có nhiều khả năng phát hiện được những sai lệch thực sự.

Một trong những ứng dụng phổ biến nhất của năng lực thống kê là để xác định số lượng mẫu cần thiết. Xác suất bạn bác bỏ giả thuyết gốc khi nó sai phụ thuộc vào mức độ sai của nó (hay còn gọi là *kích thước ảnh hưởng - effect size*) và số lượng mẫu bạn có. Có thể đoán rằng sẽ cần một số lượng mẫu rất lớn để có thể phát hiện kích thước ảnh hưởng nhỏ với xác suất cao. Việc đi sâu vào chi tiết nằm ngoài phạm vi của phần phụ lục ngắn gọn này, nhưng đây là một ví dụ. Giả sử ta có giả thuyết gốc rằng các mẫu được lấy từ một phân phối Gauss với kỳ vọng là không và phương sai là một. Nếu ta tin rằng giá trị trung bình của tập mẫu gần với một, ta chỉ cần 8 mẫu là có thể bác bỏ giả thuyết gốc với tỷ lệ lỗi chấp nhận được. Tuy nhiên, nếu ta cho rằng giá trị trung bình thực sự của tổng thể gần với 0.01, thì ta cần cỡ khoảng 80000 mẫu để có thể phát hiện được sự sai lệch.

Ta có thể hình dung năng lực thống kê như một cái máy lọc nước. Trong phép so sánh này, một kiểm định với năng lực cao giống như một hệ thống lọc nước chất lượng tốt, loại bỏ được các chất độc trong nước nhiều nhất có thể. Ngược lại, các sai lệch nhỏ cũng giống các chất cặn bẩn nhỏ, một cái máy lọc chất lượng kém sẽ để lọt các chất bẩn nhỏ đó. Tương tự, nếu năng lực thống kê không đủ cao, phép kiểm định có thể không phát hiện được các sai lệch nhỏ.

## Tiêu chuẩn Kiểm định

*Tiêu chuẩn kiểm định*  $T(x)$  là một số vô hướng có khả năng khái quát một đặc tính nào đó của dữ liệu mẫu. Mục đích của việc đặt ra một tiêu chuẩn như vậy là để phân biệt các phân phối khác nhau và tiến hành kiểm định thống kê. Nhìn lại ví dụ về nhà hóa học, nếu ta muốn chỉ ra rằng một tổng thể có chất lượng tốt hơn một tổng thể khác, việc lấy giá trị trung bình làm tiêu chuẩn kiểm định có vẻ hợp lý. Các chọn lựa tiêu chuẩn kiểm định khác nhau có thể dẫn đến các phép kiểm định thống kê với năng lực thống kê khác nhau rõ rệt.

Thường thì  $T(X)$  (phân phối của tiêu chuẩn kiểm định dưới giả thuyết gốc) sẽ (xấp xỉ) tuân theo một phân phối phổ biến như phân phối chuẩn, khi được xem xét dưới giả thuyết gốc. Nếu ta có thể chỉ rõ một phân phối như vậy, và sau đó tính tiêu chuẩn kiểm định trên tập dữ liệu, ta có thể yên tâm bác bỏ giả thuyết gốc nếu thống kê đó nằm xa bên ngoài khoảng mong đợi. Định lượng hóa ý tưởng này đưa ta đến với khái niệm trị số  $p$  (:math: 'p'-values).

## Trị số $p$

Trị số  $p$  (hay còn gọi là *trị số xác suất*) là xác suất mà  $T(X)$  lớn hơn hoặc bằng tiêu chuẩn kiểm định ta thu được, giả sử rằng giả thuyết gốc đúng, tức là:

$$p\text{-value} = P_{H_0}(T(X) \geq T(x)). \quad (20.10.7)$$

Nếu trị số  $p$  nhỏ hơn hoặc bằng một mức ý nghĩa thống kê cố định  $\alpha$  cho trước, ta có thể bác bỏ giả thuyết gốc. Còn nếu không, ta kết luận không có đủ bằng chứng để bác bỏ giả thuyết gốc. Với một phân phối của tổng thể, *miền bác bỏ* là khoảng chứa tất cả các điểm có trị số  $p$  nhỏ hơn mức ý nghĩa thống kê  $\alpha$ .

## Kiểm định Một phía và Kiểm định Hai phía

Thường thì có hai loại kiểm định ý nghĩa thống kê: kiểm định một phía và kiểm định hai phía. *Kiểm định một phía* (hay *kiểm định một đuôi*) có thể được áp dụng khi giả thuyết gốc và giả thuyết đối chỉ đi theo một hướng. Ví dụ, giả thuyết gốc có thể cho rằng tham số thực  $\theta$  nhỏ hơn hoặc bằng một giá trị  $c$ . Giả thuyết đối sẽ là  $\theta$  lớn hơn  $c$ . Nói cách khác, miền bác bỏ chỉ nằm ở một bên của phân phối mẫu. Trái với kiểm định một phía, *kiểm định hai phía* (hay *kiểm định hai đuôi*) có thể được áp dụng khi miền bác bỏ nằm ở cả hai phía của phân phối mẫu. Ví dụ cho trường hợp này có thể là một giả thuyết gốc cho rằng tham số thực  $\theta$  bằng một giá trị  $c$ . Giả thuyết đối lúc này sẽ là  $\theta$  nhỏ hơn và lớn hơn  $c$ .

## Các bước Thông thường trong Kiểm định Giả thuyết

Sau khi làm quen với các khái niệm ở trên, hãy cùng xem các bước kiểm định giả thuyết thông thường.

1. Đặt câu hỏi và đưa ra giả thuyết gốc  $H_0$ .
2. Chọn mức ý nghĩa thống kê  $\alpha$  và năng lực thống kê  $(1 - \beta)$ .
3. Thu thập mẫu qua các thử nghiệm. Số lượng mẫu cần thiết sẽ phụ thuộc vào năng lực thống kê, và hệ số ảnh hưởng mong muốn.
4. Tính tiêu chuẩn kiểm định và trị số  $p$ .
5. Quyết định chấp nhận hoặc bác bỏ giả thuyết gốc dựa trên trị số  $p$  và mức ý nghĩa thống kê  $\alpha$ .

Để tiến hành kiểm định giả thuyết, ta bắt đầu với việc định nghĩa giả thuyết gốc và mức rủi ro chấp nhận được. Sau đó ta tính tiêu chuẩn kiểm định của mẫu, lấy cực trị của tiêu chuẩn kiểm định làm bằng chứng để phủ định giả thuyết gốc. Nếu tiêu chuẩn kiểm định rơi vào miền bác bỏ, ta có thể bác bỏ giả thuyết gốc và ủng hộ giả thuyết đối.

Kiểm định giả thuyết áp dụng được trong nhiều tình huống như thử nghiệm lâm sàng (*clinical trials*) và kiểm định A/B.

### 20.10.3 Xây dựng khoảng Tin cậy

Khi ước lượng giá trị của tham số  $\theta$ , sử dụng bộ ước lượng điểm như  $\hat{\theta}$  bị hạn chế vì chúng không bao hàm sự bất định. Thay vào đó, sẽ tốt hơn nhiều nếu ta có thể tìm ra một khoảng chứa tham số  $\theta$  thật sự với xác suất cao. Nếu bạn hứng thú với những khái niệm từ một thế kỷ trước như thế này, có lẽ bạn nên đọc cuốn “Outline of a Theory of Statistical Estimation Based on the Classical Theory of Probability” (Đại cương về Lý thuyết Ước lượng Thống kê dựa trên Lý thuyết Xác suất Cổ điển) của Jerzy Neyman (Neyman, 1937), người đã đưa ra khái niệm về khoảng tin cậy vào năm 1937.

Để có tính hữu dụng, khoảng tin cậy nên càng bé càng tốt với một mức độ chắc chắn cho trước. Hãy cùng xem xét cách tính khoảng tin cậy.

#### Định nghĩa

Về mặt toán học, *khoảng tin cậy*  $C_n$  của tham số thực  $\theta$  được tính từ dữ liệu mẫu sao cho:

$$P_\theta(C_n \ni \theta) \geq 1 - \alpha, \forall \theta. \quad (20.10.8)$$

Với  $\alpha \in (0, 1)$ , và  $1 - \alpha$  được gọi là *mức độ tin cậy* hoặc *độ phủ* của khoảng đó. Nó cũng chính là hệ số  $\alpha$  của mức ý nghĩa thống kê mà chúng ta đã bàn luận ở trên.

Chú ý rằng (20.10.8) là về biến số  $C_n$ , chứ không phải giá trị cố định  $\theta$ . Để nhấn mạnh điều này, chúng ta viết  $P_\theta(C_n \ni \theta)$  thay cho  $P_\theta(\theta \in C_n)$ .

#### Diễn giải

Rất dễ để cho rằng khoảng tin cậy 95% tương đương với việc chắc chắn 95% giá trị thật phân bố trong khoảng đó, tuy nhiên đáng buồn thay điều này lại không chính xác. Tham số thật là cố định và khoảng tin cậy mới là ngẫu nhiên. Vậy nên một cách diễn giải tốt hơn đó là nếu bạn tạo ra một số lượng lớn các khoảng tin cậy theo quy trình này, thì 95% các khoảng được tạo sẽ chứa tham số thật.

Điều này nghe có vẻ tiểu tiết, nhưng lại có một ý nghĩa quan trọng trong việc diễn giải các kết quả. Cụ thể, chúng ta có thể thỏa mãn (20.10.8) bằng cách tạo ra các khoảng gần như chắc chắn không chứa tham số thật, miễn là số lượng các khoảng này đủ nhỏ. Chúng ta kết thúc mục này bằng ba mệnh đề nghe hợp lý nhưng lại không chính xác. Thảo luận sâu hơn về các mệnh đề này có thể tham khảo thêm ở (Morey et al., 2016).

- **Sai lầm 1:** Khoảng tin cậy hẹp cho phép chúng ta dự đoán các giá trị một cách chính xác.
- **Sai lầm 2:** Các giá trị nằm trong khoảng tin cậy có nhiều khả năng là giá trị thực hơn là các giá trị nằm bên ngoài.
- **Sai lầm 3:** Xác xuất một khoảng tin cậy 95% chứa các giá trị thực là 95%.

Có thể nói, các khoảng tin cậy là những đối tượng khó ước lượng. Tuy nhiên nếu như ta diễn giải chúng một cách rõ ràng, thì chúng có thể trở thành những công cụ quyền năng.

## Một ví dụ về Gaussian

Cùng bàn về ví dụ kinh điển nhất, khoảng tin cậy cho giá trị trung bình của một phân phối Gaussian với kỳ vọng và phương sai chưa xác định. Giả sử chúng ta thu thập  $n$  mẫu  $\{x_i\}_{i=1}^n$  từ phân phối Gaussian  $\mathcal{N}(\mu, \sigma^2)$ . Chúng ta có thể ước lượng kỳ vọng và độ lệch chuẩn bằng công thức:

$$\hat{\mu}_n = \frac{1}{n} \sum_{i=1}^n x_i \text{ và } \hat{\sigma}_n^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \hat{\mu})^2. \quad (20.10.9)$$

Nếu bây giờ chúng ta xem xét biến ngẫu nhiên:

$$T = \frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n / \sqrt{n}}, \quad (20.10.10)$$

Chúng ta có được một biến ngẫu nhiên theo phân phối *t Student* trên  $n - 1$  bậc tự do.

Phân phối này đã được nghiên cứu rất chi tiết, và đã được chứng minh là khi  $n \rightarrow \infty$ , nó xấp xỉ với một phân phối Gauss tiêu chuẩn, và do đó bằng cách nhìn vào bảng giá trị phân phối tích lũy Gauss, chúng ta có thể kết luận rằng giá trị  $T$  nằm trong khoảng  $[-1.96, 1.96]$  tối thiểu là 95% các trường hợp. Với giá trị  $n$  hữu hạn, khoảng tin cậy sẽ lớn hơn, nhưng chúng vẫn rõ ràng và thường được tính sẵn và trình bày thành bảng.

Do đó, chúng ta có thể kết luận với giá trị  $n$  lớn:

$$P\left(\frac{\hat{\mu}_n - \mu}{\hat{\sigma}_n / \sqrt{n}} \in [-1.96, 1.96]\right) \geq 0.95. \quad (20.10.11)$$

Sắp xếp lại công thức này bằng cách nhân hai vế với  $\hat{\sigma}_n / \sqrt{n}$  và cộng thêm  $\hat{\mu}_n$ , ta có:

$$P\left(\mu \in \left[\hat{\mu}_n - 1.96 \frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96 \frac{\hat{\sigma}_n}{\sqrt{n}}\right]\right) \geq 0.95. \quad (20.10.12)$$

Như vậy chúng ta đã xác định được khoảng tin cậy 95% cần tìm:

$$\left[\hat{\mu}_n - 1.96 \frac{\hat{\sigma}_n}{\sqrt{n}}, \hat{\mu}_n + 1.96 \frac{\hat{\sigma}_n}{\sqrt{n}}\right]. \quad (20.10.13)$$

Không quá khi nói rằng (20.10.13) là một trong những công thức sử dụng nhiều nhất trong thống kê. Hãy kết thúc thảo luận về thống kê của chúng ta bằng cách lập trình tìm khoảng tin cậy. Để đơn giản, giả sử chúng ta đang làm việc ở vùng tiệm cận. Khi  $N$  nhỏ, nên xác định giá trị chính xác của t\_star bằng phương pháp lập trình hoặc từ bảng tra phân phối tích lũy *t Student*.

```
# Number of samples
N = 1000
# Sample dataset
samples = np.random.normal(loc=0, scale=1, size=(N,))
# Lookup Students's t-distribution c.d.f.
t_star = 1.96
# Construct interval
mu_hat = np.mean(samples)
sigma_hat = samples.std(ddof=1)
(mu_hat - t_star*sigma_hat/np.sqrt(N), mu_hat + t_star*sigma_hat/np.sqrt(N))
```

#### 20.10.4 Tóm tắt

- Thống kê tập trung vào các vấn đề suy luận, trong khi học sâu chú trọng vào đưa ra các dự đoán chuẩn xác mà không cần một phương pháp lập trình hay kiến thức rõ ràng.
- Ba phương pháp suy luận thống kê thông dụng nhất: đánh giá và so sánh các bộ ước lượng, tiến hành kiểm định giả thuyết, và tạo các khoảng tin cậy.
- Ba bộ ước lượng thông dụng nhất: độ chêch thống kê, độ lệch chuẩn, và trung bình bình phương sai số.
- Một khoảng tin cậy là khoảng ước tính của tập tham số thực mà chúng ta có thể tạo ra bằng các mẫu cho trước.
- Kiểm định giả thuyết là phương pháp để đánh giá các chứng cứ chống lại mệnh đề mặc định về một tổng thể.

#### 20.10.5 Bài tập

1. Cho  $X_1, X_2, \dots, X_n \stackrel{\text{iid}}{\sim} \text{Unif}(0, \theta)$ , với “iid” là viết tắt của *phân phối độc lập và giống nhau - independent and identically distributed*. Xét bộ ước lượng  $\theta$  dưới đây:

$$\hat{\theta} = \max\{X_1, X_2, \dots, X_n\}; \quad (20.10.14)$$

$$\tilde{\theta} = 2\bar{X}_n = \frac{2}{n} \sum_{i=1}^n X_i. \quad (20.10.15)$$

- Tìm độ chêch thống kê, độ lệch chuẩn, và trung bình bình phương sai số của  $\hat{\theta}$ .
  - Tìm độ chêch thống kê, độ lệch chuẩn, và trung bình bình phương sai số của  $\tilde{\theta}$ .
  - Bộ ước lượng nào tốt hơn?
2. Trở lại ví dụ về nhà hóa học của chúng ta ở phần mở đầu, liệt kê 5 bước để tiến hành kiểm định giả thuyết hai chiều, biết mức ý nghĩa thống kê  $\alpha = 0.05$  và năng lực thống kê  $1 - \beta = 0.8$ .
  3. Chạy đoạn mã lập trình khoảng tin cậy biết  $N = 2$  và  $\alpha = 0.5$  với 100 dữ liệu được tạo độc lập, sau đó vẽ đồ thị các khoảng kết quả (trường hợp này  $t_{\text{star}} = 1.0$ ). Bạn sẽ thấy một vài khoảng rất nhỏ cách xa khoảng chứa giá trị kỳ vọng thực 0. Điều này có mâu thuẫn với việc diễn giải khoảng tin cậy không? Có đúng không khi sử dụng các khoảng nhỏ này để nói các ước lượng có độ chính xác cao?

#### 20.10.6 Thảo luận

- Tiếng Anh: MXNet<sup>435</sup>, Pytorch<sup>436</sup>, Tensorflow<sup>437</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>438</sup>

<sup>435</sup> <https://discuss.d2l.ai/t/statistics/419>

<sup>436</sup> <https://discuss.d2l.ai/t/1102>

<sup>437</sup> <https://discuss.d2l.ai/t/1103>

<sup>438</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.10.7 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Ngô Thế Anh Khoa
- Phạm Hồng Vinh
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc
- Đoàn Võ Duy Thanh
- Nguyễn Lê Quang Nhật
- Mai Sơn Hải
- Phạm Minh Đức
- Nguyễn Cảnh Thượng
- Nguyễn Văn Cường

## 20.11 Lý thuyết Thông tin

Vũ trụ mà chúng ta đang sống thì tràn ngập với thông tin. Thông tin cung cấp một ngôn ngữ chung giúp vượt qua rào cản ngăn cách nhiều lĩnh vực riêng biệt: từ thơ của Shakespeare đến các bài báo khoa học của các nhà nghiên cứu trên Cornell ArXiv, từ bản in Đêm Đầu Sao của Van Gogh đến Bản Giao Hưởng Số 5 của Beethoven, từ ngôn ngữ lập trình đầu tiên Plankalkül đến các thuật toán học máy hiện đại nhất. Mọi thứ phải tuân theo các quy tắc của lý thuyết thông tin, bất kể chúng ở định dạng nào. Với lý thuyết thông tin, chúng ta có thể đo lường và so sánh lượng thông tin có trong các tín hiệu khác nhau. Trong phần này, chúng ta sẽ nghiên cứu các khái niệm cơ bản của lý thuyết thông tin và các ứng dụng của 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. Mục tiêu của học máy là trích xuất các đặc trưng đáng chú ý từ dữ liệu và đưa ra các dự đoán quan trọng. Mặt khác, lý thuyết thông tin nghiên cứu vấn đề mã hóa, giải mã, truyền và thao tác với thông tin. Do vậy, 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 học máy sử dụng mất mát entropy chéo như được mô tả trong [Section 5.4](#). Mất mát này có thể được trực tiếp rút ra từ các quan điểm từ góc nhìn lý thuyết thông tin.

### 20.11.1 Thông tin

Ta hãy bắt đầu với “linh hồn” của lý thuyết thông tin: thông tin. *Thông tin* có thể được mã hóa trong bất kỳ thứ gì với một hoặc nhiều chuỗi định dạng mã hóa. Ta giả sử thử chính mình đưa ra một định nghĩa cho khái niệm thông tin. Ta có thể bắt đầu từ đâu?

Hãy xem thí nghiệm tưởng tượng sau đây. Ta có một người bạn với một bộ bài. Họ sẽ xáo trộn bộ bài, lật qua một số lá bài và cho chúng ta biết vài điều về các quân bài. Chúng ta sẽ thử đánh giá nội dung thông tin của từng phát biểu sau đây.

Đầu tiên, họ lật một lá và nói, “Tôi thấy một lá bài.” Điều này không cung cấp cho ta thông tin nào. Rõ ràng là trong trường hợp này chúng ta đã biết chắc chắn mệnh đề trên là đúng nên lượng thông tin mà nó chứa sẽ là 0.

Tiếp theo, họ lật một lá khác và nói, “Tôi thấy một lá co.” Điều này cung cấp cho ta một chút thông tin, mà trên thực tế chỉ có thể có 4 loại chất khác nhau, mỗi chất đều có khả năng như nhau, vì vậy ta không ngạc nhiên trước kết quả này. Ta hy vọng rằng dù thông tin được đo đạc dưới bất kể hình thức nào, sự kiện này nên có hàm lượng thông tin thấp.

Tiếp theo, họ lật một lá và nói, “Đây là quân 3 bích.” Câu trên chứa nhiều thông tin hơn. Quả thực có 52 kết quả tương đương có thể xảy ra, và ta đã được cho biết đó là lá bài nào. Đây là một lượng thông tin trung bình.

Hãy đi đến cực hạn. Giả sử rằng cuối cùng họ lật từng lá bài từ bộ bài và đọc ra toàn bộ trình tự của bộ bài đã bị xáo trộn đó. Có 52! các thứ tự khác nhau cho bộ bài với khả năng như nhau, vì vậy chúng ta cần rất nhiều thông tin để biết được chính xác trình tự rút bài.

Bất kỳ khái niệm thông tin nào chúng ta phát triển phải phù hợp với trực giác này. Thật vậy, trong phần tiếp theo, chúng ta sẽ học cách tính toán rằng các sự kiện trên có tương ứng 0 bit, 2 bit, 5.7 bit, và 225.6 bit thông tin.

Nếu đọc hết những thí nghiệm tưởng tượng này, chúng ta thấy một ý tưởng tự nhiên. Để khởi đầu, thay vì quan tâm đến kiến thức đã biết, chúng ta có thể xây dựng ý tưởng là thông tin đại diện cho mức độ bất ngờ hoặc xác suất trừu tượng của sự kiện. Ví dụ, nếu chúng ta muốn mô tả một sự kiện hiếm gặp, chúng ta cần rất nhiều thông tin. Đối với một sự kiện phổ biến, chúng ta có thể không cần nhiều thông tin.

Năm 1948, Claude E. Shannon thiết lập lĩnh vực lý thuyết thông tin qua bài báo khoa học *Lý thuyết Toán cho Truyền tải Thông tin - A Mathematical Theory of Communication* (Shannon, 1948). Trong bài báo của mình, Shannon đưa ra khái niệm entropy thông tin. Chúng ta sẽ bắt đầu từ đây.

## Lượng tin

Vì thông tin biểu diễn xác suất trừu tượng của một sự kiện, làm thế nào để chúng ta ánh xạ xác suất đó thành số lượng bit? Shannon đã giới thiệu thuật ngữ *bit* làm đơn vị thông tin, mà ban đầu được đề xuất bởi John Tukey. Vậy “bit” là gì và tại sao ta sử dụng nó để đo lường thông tin? Trong quá khứ, một máy phát tín hiệu chỉ có thể gửi hoặc nhận hai loại mã: 0 và 1. Mà thật ra 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 cũng được mã hóa bởi một chuỗi 0 và 1. Và do đó, một chuỗi các chữ số nhị phân (*binary*) có độ dài  $n$  chứa  $n$  bit thông tin.

Bây giờ, giả sử rằng đối với bất kỳ chuỗi mã nào, mỗi giá trị 0 hoặc 1 xuất hiện với xác suất là  $\frac{1}{2}$ . Do đó, sự kiện  $X$  với một chuỗi mã có độ dài  $n$ , xảy ra với xác suất  $\frac{1}{2^n}$ . Đồng thời, như chúng tôi đã đề cập trước đây, chuỗi số này chứa  $n$  bit thông tin. Vì vậy, liệu có thể tổng quát hóa thành một hàm toán học chuyển xác suất  $p$  thành số lượng bit không? Shannon đưa ra câu trả lời bằng cách định nghĩa *lượng tin*

$$I(X) = -\log_2(p), \quad (20.11.1)$$

là số *bit* thông tin ta đã nhận cho sự kiện  $X$  này. Lưu ý rằng ta sẽ luôn sử dụng logarit cơ số 2 trong phần này. Để đơn giản, phần còn lại của phần này sẽ bỏ qua cơ số 2 trong ký hiệu logarit, tức là  $\log(\cdot)$  luôn có nghĩa là  $\log_2(\cdot)$ . Ví dụ: mã “0010” có lượng tin là

$$I("0010") = -\log(p("0010")) = -\log\left(\frac{1}{2^4}\right) = 4 \text{ bits.} \quad (20.11.2)$$

Chúng ta có thể tính toán lượng tin như phần dưới đây. Trước đó, hãy nhập tất cả các gói cần thiết trong phần này.

```
from mxnet import np
from mxnet.metric import NegativeLogLikelihood
from mxnet.ndarray import nansum
import random

def self_information(p):
    return -np.log2(p)

self_information(1 / 64)
```

6.0

### 20.11.2 Entropy

Do lượng tin chỉ đo lường thông tin từ một biến cố rời rạc đơn lẻ, chúng ta cần một thước đo khái quát hơn cho cả biến ngẫu nhiên có phân bố rời rạc và liên tục.

#### Phát triển Lý thuyết Entropy

Hãy phân tích cụ thể hơn. Dưới đây là các phát biểu không chính thức của *các tiên đề Shannon* về *entropy*. Chúng buộc ta đi tới một định nghĩa độc nhất về thông tin. Một phiên bản chính quy của những tiên đề này cùng với một số tiên đề khác có thể được tìm thấy trong ([Csiszar, 2008](#)).

1. Thông tin thu được bằng cách quan sát một biến ngẫu nhiên không phụ thuộc vào các yếu tố, hay sự xuất hiện của các yếu tố bổ sung mà có xác suất bằng 0.
2. Thông tin thu được bằng cách quan sát hai biến ngẫu nhiên không lớn hơn tổng thông tin thu được khi quan sát chúng một cách riêng rẽ. Nếu hai biến ngẫu nhiên là độc lập thì thông tin thu được từ hai cách bằng nhau.
3. Thông tin thu được khi quan sát những biến cố (gần như) chắc chắn thì (gần như) bằng 0.

Việc chứng minh các tiên đề trên nằm ngoài phạm vi của cuốn sách, điều quan trọng cần nhớ là chúng xác định một cách độc nhất hình thái mà entropy phải có. Chỉ có duy nhất một điều chưa xác định từ những phát biểu trên là về việc chọn đơn vị cho entropy, mà điều này thường được chuẩn hóa bằng cách đặt thông tin cung cấp bởi một lần lật đồng xu cân đối đồng chất là một bit, như đã thấy trước đó.

#### Định nghĩa

Cho một biến ngẫu nhiên  $X$  bất kỳ tuân theo phân phối xác suất  $P$  với hàm mật độ xác suất (p.d.f) hoặc hàm khối xác suất (p.m.f)  $p(x)$ , ta đo lượng thông tin kỳ vọng thu được thông qua *entropy* (hoặc *entropy Shannon*):

$$H(X) = -E_{x \sim P}[\log p(x)]. \quad (20.11.3)$$

Cụ thể hơn, nếu  $X$  rời rạc:

$$H(X) = - \sum_i p_i \log p_i, \text{ where } p_i = P(X_i). \quad (20.11.4)$$

Ngược lại, nếu  $X$  liên tục, ta gọi là *entropy vi phân* (*differential entropy*):

$$H(X) = - \int_x p(x) \log p(x) dx. \quad (20.11.5)$$

Chúng ta có thể định nghĩa entropy như sau.

```
def entropy(p):
    entropy = - p * np.log2(p)
    # Operator nanum will sum up the non-nan number
    out = nanum(entropy.as_nd_ndarray())
    return out

entropy(np.array([0.1, 0.5, 0.1, 0.3]))
```

```
[1.6854753]
<NDArray 1 @cpu(0)>
```

## Điễn giải

Bạn có thể thắc mắc: trong định nghĩa entropy (20.11.3), tại sao chúng ta sử dụng kỳ vọng của logarit âm? Dưới đây là một số cách giải thích trực quan.

Đầu tiên, tại sao chúng ta sử dụng hàm *logarit log*? Giả sử  $p(x) = f_1(x)f_2(x)\dots,f_n(x)$ , khi mỗi hàm thành tố  $f_i(x)$  độc lập lẫn nhau. Điều này nghĩa là mỗi  $f_i(x)$  đóng góp một cách độc lập vào tổng thông tin thu được từ  $p(x)$ . Như đã thảo luận ở trên, ta muốn công thức entropy là phép cộng trên các biến ngẫu nhiên độc lập. May mắn thay, hàm log có thể chuyển tích thành tổng.

Tiếp theo, tại sao chúng ta sử dụng *log âm*? Một cách trực quan, những biến cố xảy ra thường xuyên sẽ chứa ít thông tin hơn những biến cố hiếm vì ta thường thu được nhiều thông tin hơn từ những trường hợp bất thường. Do đó, ta cần thiết lập mối quan hệ đơn điệu giảm giữa xác suất của biến cố và entropy của chúng, và muốn entropy luôn dương (vì các quan sát mới không nên buộc ta quên đi những gì đã biết). Tuy nhiên, hàm log lại là đơn điệu tăng, và có giá trị âm với xác suất trong đoạn  $[0, 1]$ . Vậy nên ta thêm dấu âm vào trước hàm log.

Cuối cùng, hàm *kỳ vọng* đến từ đâu? Xét một biến ngẫu nhiên  $X$ . Ta có thể diễn giải lượng tin (*self-information*) ( $-\log(p)$ ) như mức độ *bất ngờ* khi quan sát được một kết quả cụ thể nào đó. Thật vậy, khi xác suất xấp xỉ bằng 0, mức độ bất ngờ tiến tới vô cùng. Tương tự, chúng ta có thể diễn giải entropy như mức độ bất ngờ trung bình từ việc quan sát  $X$ . Ví dụ, tưởng tượng một hệ thống máy đánh bạc đưa ra các ký hiệu độc lập  $s_1, \dots, s_k$  với xác suất lần lượt là  $p_1, \dots, p_k$ . Khi đó, entropy của hệ thống này bằng với lượng tin trung bình thu được từ việc quan sát mỗi kết quả, tức:

$$H(S) = \sum_i p_i \cdot I(s_i) = - \sum_i p_i \cdot \log p_i. \quad (20.11.6)$$

## Tính chất của Entropy

Bằng các ví dụ và diễn giải phía trên, ta có thể rút ra các tính chất sau của entropy (20.11.3). Ở đây, ta xem  $X$  là một biến cố và  $P$  là phân phối xác suất của  $X$ .

- Entropy có giá trị không âm, tức  $H(X) \geq 0, \forall X$ .
- Nếu  $X \sim P$  có hàm mật độ xác suất hoặc hàm khối xác suất  $p(x)$ , và ta muốn ước lượng  $P$  bằng một phân phối xác suất mới  $Q$  với hàm mật độ xác suất hoặc hàm khối xác suất  $q(x)$ , ta có:

$$H(X) = -E_{x \sim P}[\log p(x)] \leq -E_{x \sim P}[\log q(x)], \text{ dấu bằng xảy ra khi và chỉ khi } P = Q.$$

Ngoài ra, :  $H(X)$  chobiitcndicasbittrungbnhcen  
(20.11.7)

dùng để mã hóa các giá trị lấy từ  $P$ .

- Nếu  $X \sim P$ ,  $x$  sẽ chứa lượng thông tin cực đại nếu mọi biến cố khả dĩ chứa lượng thông tin như nhau. Cụ thể, nếu  $P$  là phân phối rời rạc với  $k$  lớp  $\{p_1, \dots, p_k\}$ :

$$H(X) \leq \log(k), \text{ dấu bằng xảy ra khi và chỉ khi } p_i = \frac{1}{k}, \forall i. \quad (20.11.8)$$

Nó :  $P$  lphnphilintcthsphctphn. Tuynhin,

nếu ta giả sử thêm rằng  $P$  có miền giá trị nằm trong một khoảng hữu hạn (với tất cả giá trị nằm trong khoảng 0 và 1),  $P$  sẽ có entropy cực đại nếu nó là phân phối đều trong khoảng đó.

### 20.11.3 Thông tin Tương hỗ

Ta đã định nghĩa entropy của một biến ngẫu nhiên  $X$  duy nhất, vậy còn entropy của một cặp biến ngẫu nhiên  $(X, Y)$  thì sao? Ta xem xét khái niệm này để trả lời các câu hỏi: “Thông tin chứa trong cả  $X$  và  $Y$  sẽ trông như thế nào so với thông tin trong từng biến? Có thông tin thừa không, hay chúng đều độc nhất?”

Trong phần thảo luận dưới đây, chúng tôi sẽ luôn dùng  $(X, Y)$  để ký hiệu một cặp biến ngẫu nhiên tuân theo phân phối xác suất kết hợp  $P$  với hàm mật độ xác suất hoặc hàm khối xác suất  $p_{X,Y}(x, y)$ , còn  $X$  và  $Y$  lần lượt tuân theo phân phối xác suất  $p_X(x)$  và  $p_Y(y)$ .

#### Entropy Kết hợp

Tương tự như entropy của một biến ngẫu nhiên (20.11.3), ta định nghĩa *entropy kết hợp (joint entropy)*  $H(X, Y)$  của cặp biến ngẫu nhiên  $(X, Y)$  như sau

$$H(X, Y) = -E_{(x,y) \sim P}[\log p_{X,Y}(x, y)]. \quad (20.11.9)$$

Nếu  $(X, Y)$  là rời rạc:

$$H(X, Y) = - \sum_x \sum_y p_{X,Y}(x, y) \log p_{X,Y}(x, y). \quad (20.11.10)$$

Mặt khác, nếu  $(X, Y)$  là liên tục, ta định nghĩa *entropy kết hợp vi phân (differential joint entropy)* như sau:

$$H(X, Y) = - \int_{x,y} p_{X,Y}(x, y) \log p_{X,Y}(x, y) dx dy. \quad (20.11.11)$$

Ta có thể xem (20.11.9) như tổng mức độ ngẫu nhiên của cặp biến ngẫu nhiên. Ở một cực trị, nếu chúng giống hệt nhau ( $X = Y$ ), thông tin của cặp biến này chính là thông tin của từng biến:  $H(X, Y) = H(X) = H(Y)$ . Ở cực trị còn lại, nếu  $X$  và  $Y$  độc lập thì  $H(X, Y) = H(X) + H(Y)$ . Tất nhiên, thông tin chứa trong một cặp biến ngẫu nhiên sẽ không thể nhỏ hơn entropy của từng biến ngẫu nhiên và không thể lớn hơn tổng entropy của chúng.

$$H(X), H(Y) \leq H(X, Y) \leq H(X) + H(Y). \quad (20.11.12)$$

Hãy cùng lập trình entropy kết hợp từ đầu.

```
def joint_entropy(p_xy):
    joint_ent = -p_xy * np.log2(p_xy)
    # Operator nansum will sum up the non-nan number
    out = nansum(joint_ent.as_nd_ndarray())
    return out

joint_entropy(np.array([[0.1, 0.5], [0.1, 0.3]]))
```

[1.6854753]  
<NDArray 1 @cpu(0)>

Hãy để ý rằng đây chính là *đoạn mã* từ trước, nhưng giờ ta hiểu nó theo cách khác khi làm việc với phân phối kết hợp của hai biến ngẫu nhiên.

### Entropy có Điều kiện

Entropy kết hợp định nghĩa phía trên là lượng thông tin chứa trong một cặp biến ngẫu nhiên. Đại lượng này khá hữu ích, nhưng thường nó không phải là thứ mà ta quan tâm. Hãy xem xét trong ngữ cảnh học máy. Gọi  $X$  là biến ngẫu nhiên (hoặc vector biến ngẫu nhiên) mô tả giá trị các điểm ảnh trong một bức ảnh, và  $Y$  là biến ngẫu nhiên mô tả nhãn lớp.  $X$  chứa một lượng thông tin rất lớn — do một bức ảnh tự nhiên khá phức tạp. Tuy nhiên, lượng thông tin trong  $Y$  khi ta đã thấy bức ảnh nên là nhỏ. Tất nhiên, bức ảnh chứa một chữ số cũng nên chứa thông tin đó là chữ số nào, trừ khi chữ số trong ảnh không thể đọc được. Vì vậy, để tiếp tục mở rộng lý thuyết thông tin, ta cần suy luận được lượng thông tin trong một biến ngẫu nhiên khi nó phụ thuộc vào một biến khác.

Trong lý thuyết xác suất, *xác suất có điều kiện* đo mối quan hệ giữa các biến. Bây giờ ta muốn định nghĩa *entropy có điều kiện* (*conditional entropy*)  $H(Y | X)$  theo cách tương tự dưới dạng:

$$H(Y | X) = -E_{(x,y) \sim P}[\log p(y | x)], \quad (20.11.13)$$

trong đó  $p(y | x) = \frac{p_{X,Y}(x,y)}{p_X(x)}$  là xác suất có điều kiện. Cụ thể, nếu  $(X, Y)$  là rời rạc, ta có:

$$H(Y | X) = -\sum_x \sum_y p(x, y) \log p(y | x). \quad (20.11.14)$$

Nếu  $(X, Y)$  là liên tục, *entropy có điều kiện vi phân* được định nghĩa tương tự như sau:

$$H(Y | X) = - \int_x \int_y p(x, y) \log p(y | x) dx dy. \quad (20.11.15)$$

Bây giờ một câu hỏi tự nhiên là: *entropy có điều kiện*  $H(Y | X)$  có mối quan hệ gì với entropy  $H(X)$  và entropy kết hợp  $H(X, Y)$ ? Sử dụng các định nghĩa ở trên, ta có thể biểu diễn mối quan hệ đó một cách gọn gàng:

$$H(Y | X) = H(X, Y) - H(X). \quad (20.11.16)$$

Điều này có thể được giải thích một cách trực quan như sau: thông tin trong  $Y$  khi biết  $X$  ( $H(Y | X)$ ) bằng với thông tin trong cả  $X$  và  $Y$  ( $H(X, Y)$ ) trừ đi thông tin đã có trong  $X$ . Nó cho ta biết thông tin có trong  $Y$  mà không chứa trong  $X$ .

Bây giờ, hãy cùng lập trình entropy có điều kiện (20.11.13) từ đầu.

```
def conditional_entropy(p_xy, p_x):
    p_y_given_x = p_xy/p_x
    cond_ent = -p_xy * np.log2(p_y_given_x)
    # Operator nanum will sum up the non-nan number
    out = nanum(cond_ent.as_nd_ndarray())
    return out

conditional_entropy(np.array([[0.1, 0.5], [0.2, 0.3]]), np.array([0.2, 0.8]))
```

```
[0.8635472]
<NDArray 1 @cpu(0)>
```

## Thông tin Tương hỗ

Với định nghĩa trước đó về các biến ngẫu nhiên  $(X, Y)$ , bạn có thể tự hỏi: "Ta đã biết có bao nhiêu thông tin nằm trong  $Y$  nhưng không nằm ở trong  $X$ , liệu có thể biết được có bao nhiêu thông tin giống nhau giữa  $X$  và  $Y$  không?". Đại lượng đó là *thông tin tương hỗ* của  $(X, Y)$ , ký hiệu là  $I(X, Y)$ .

Thay vì đề cập ngay đến định nghĩa chính thức, hãy cùng luyện tập trực giác bằng cách suy luận biểu thức thông tin tương hỗ dựa trên những khái niệm mà chúng ta đã xây dựng trước đó. Mục tiêu của chúng ta là tìm được thông tin giống nhau giữa hai biến ngẫu nhiên. Ta có thể thử bắt đầu với thông tin chứa trong cả  $X$  và  $Y$ , sau đó bỏ đi những thông tin không giống nhau. Thông tin chứa trong cả  $X$  và  $Y$  là  $H(X, Y)$ . Thông tin nằm trong  $X$  nhưng không nằm trong  $Y$  là  $H(X)$  **:raw-latex: `mid` `Y`\$, tương tự, thông tin nằm trong :math:`Y`** nhưng không nằm trong  $X$  là  $H(Y | X)$ . Do đó, ta có thông tin tương hỗ như sau:

$$I(X, Y) = H(X, Y) - H(Y | X) - H(X | Y). \quad (20.11.17)$$

Thật vậy, đây là định nghĩa hợp lệ của thông tin tương hỗ. Nếu ta thay các số hạng trên bằng định nghĩa của chúng, tổng hợp lại, rồi biến đổi đại số một chút, ta sẽ có:

$$I(X, Y) = E_x E_y \left\{ p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)} \right\}. \quad (20.11.18)$$

Ta có thể tóm tắt tất cả những mối quan hệ nêu trên ở hình Fig. 20.11.1. Đây là một minh họa trực quan tuyệt vời để hiểu tại sao các mệnh đề sau đây đều tương đương với  $I(X, Y)$ .

- $H(X) - H(X | Y)$
- $H(Y) - H(Y | X)$

- $H(X) + H(Y) - H(X, Y)$

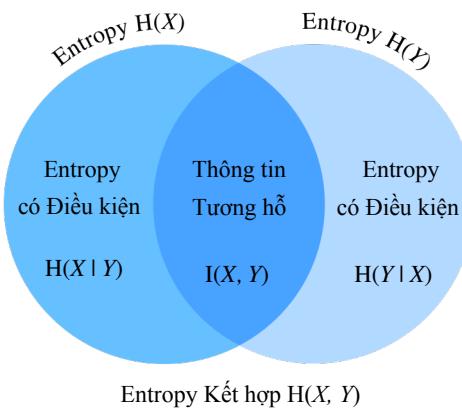


Fig. 20.11.1: Mối quan hệ giữa thông tin tương hỗ với entropy kết hợp và entropy có điều kiện.

Theo nhiều cách ta có thể xem thông tin tương hỗ (20.11.18) như là phần mở rộng của hệ số tương quan trong Section 20.6. Đại lượng này cho phép chúng ta hiểu không chỉ về mối quan hệ tuyến tính của các biến, mà còn cả lượng thông tin tối đa mà hai biến chia sẻ với nhau.

Bây giờ, hãy cùng lập trình thông tin tương hỗ từ đầu.

```
def mutual_information(p_xy, p_x, p_y):
    p = p_xy / (p_x * p_y)
    mutual = p_xy * np.log2(p)
    # Operator nansum will sum up the non-nan number
    out = nansum(mutual.as_ndarray())
    return out

mutual_information(np.array([[0.1, 0.5], [0.1, 0.3]]),
                    np.array([0.2, 0.8]), np.array([[0.75, 0.25]]))
```

```
[0.7194603]
<NDArray 1 @cpu(0)>
```

### Tính chất của Thông tin Tương Hỗ

Thay vì ghi nhớ định nghĩa thông tin tương hỗ (20.11.18), bạn chỉ cần lưu ý những tính chất nổi trội của nó:

- Thông tin tương hỗ có tính đối xứng:  $I(X, Y) = I(Y, X)$ .
- Thông tin tương hỗ có giá trị không âm:  $I(X, Y) \geq 0$ .
- $I(X, Y) = 0$  khi và chỉ khi  $X$  và  $Y$  là hai biến độc lập. Ví dụ, nếu  $X$  và  $Y$  độc lập thì việc biết thông tin của  $Y$  không cho ta thông tin của  $X$  và ngược lại, do đó thông tin tương hỗ của chúng bằng 0.
- Ngoài ra, nếu  $X$  là hàm nghịch đảo của  $Y$ , thì  $Y$  và  $X$  có chung toàn bộ thông tin và

$$I(X, Y) = H(Y) = H(X). \quad (20.11.19)$$

## Thông tin Tương hỗ theo từng Điểm

Khi bắt đầu tìm hiểu về entropy ở đầu chương này, ta đã diễn giải –  $\log(p_X(x))$  như mức độ *ngạc nhiên* với kết quả cụ thể của biến ngẫu nhiên. Ta có thể diễn giải tương tự với logarit trong thông tin tương hỗ, thường được biết đến với cái tên *thông tin tương hỗ theo từng điểm* (*pointwise mutual information*):

$$\text{pmi}(x, y) = \log \frac{p_{X,Y}(x, y)}{p_X(x)p_Y(y)}. \quad (20.11.20)$$

(20.11.20) so sánh xác suất  $x$  và  $y$  xảy ra đồng thời qua phân phối kết hợp với khi chúng cùng xảy ra qua 2 phân phối ngẫu nhiên độc lập. Nếu kết quả lớn và dương, thì  $x$  và  $y$  có xác suất xảy ra đồng thời qua phân phối kết hợp cao hơn nhiều. (chú ý: *mẫu số*:  $p_X(x)p_Y(y)$  là xác suất của hai đầu ra độc lập), ngược lại nếu kết quả lớn và âm, thì xác suất xảy ra đồng thời qua phân phối kết hợp sẽ rất thấp.

Điều này cho phép ta diễn giải thông tin tương hỗ (20.11.18) như độ ngạc nhiên trung bình khi hai biến cố xảy ra đồng thời so với độ ngạc nhiên khi chúng là hai biến độc lập.

### Ứng dụng Thông tin Tương hỗ

Thông tin tương hỗ có thể hơi trừu tượng theo định nghĩa thuần túy, vậy nó liên quan như thế nào đến học máy? Trong xử lý ngôn ngữ tự nhiên, một trong những vấn đề khó khăn nhất là *giải quyết sự mơ hồ*, tức nghĩa của từ đang không rõ ràng trong ngữ cảnh. Ví dụ, gần đây có một tiêu đề trong bản tin thông báo rằng “Amazon đang cháy”. Bạn có thể tự hỏi là liệu công ty Amazon có một tòa nhà bị cháy, hay rừng Amazon đang cháy.

Thông tin tương hỗ có thể giúp ta giải quyết sự mơ hồ này. Đầu tiên, ta tìm nhóm từ có thông tin tương hỗ tương đối lớn tới công ty Amazon, chẳng hạn như thương mại điện tử, công nghệ và trực tuyến. Thứ hai, ta tìm một nhóm từ khác có một thông tin tương hỗ tương đối lớn tới rừng mưa Amazon, chẳng hạn như mưa, rừng và nhiệt đới. Khi cần phân biệt “Amazon”, ta có thể so sánh nhóm nào xuất hiện nhiều hơn trong ngữ cảnh của từ này. Trong trường hợp này, bài báo sẽ tiếp tục mô tả khu rừng và ngữ cảnh sẽ được làm rõ.

### 20.11.4 Phân kỳ Kullback-Leibler

Như đã thảo luận trong Section 4.3, ta có thể sử dụng chuẩn (*norms*) để đo khoảng cách giữa hai điểm trong không gian với số chiều bất kỳ. Ta muốn thực hiện công việc tương tự với các phân phối xác suất. Có nhiều cách để giải quyết vấn đề này, nhưng lý thuyết thông tin cung cấp một trong những cách tốt nhất. Nay ta sẽ tìm hiểu về *phân kỳ Kullback-Leibler (KL)* (*Kullback-Leibler divergence*), là phương pháp đo lường xem hai phân phối có gần nhau hay không.

## Định nghĩa

Cho một biến ngẫu nhiên  $X$  tuân theo phân phối xác suất  $P$  với hàm mật độ xác suất hay hàm khối xác suất là  $p(x)$ , và ta ước lượng  $P$  bằng một phân phối xác suất  $Q$  khác với hàm mật độ xác suất hoặc hàm khối xác suất  $q(x)$ . Khi đó, *phân kỳ Kullback–Leibler* (hoặc *entropy tương đối*) giữa  $P$  và  $Q$  là

$$D_{\text{KL}}(P\|Q) = E_{x \sim P} \left[ \log \frac{p(x)}{q(x)} \right]. \quad (20.11.21)$$

Như với thông tin tương hỗ theo từng điểm (20.11.20), ta lại có thể diễn giải hạng tử logarit:  $-\log \frac{q(x)}{p(x)} = -\log(q(x)) - (-\log(p(x)))$  sẽ lớn và dương nếu ta thấy  $x$  xuất hiện thường xuyên hơn theo phân phối  $P$  so với mức ta kỳ vọng cho phân phối  $Q$ , và lớn và âm nếu chúng ta thấy kết quả ít hơn nhiều so với kỳ vọng. Theo cách này, ta có thể hiểu nó là mức độ ngạc nhiên *tương đối* khi quan sát phân phối mục tiêu so với phân phối tham chiếu.

Ta hãy thực hiện tính phân kỳ KL từ đầu.

```
def kl_divergence(p, q):
    kl = p * np.log2(p / q)
    out = nansum(kl.as_nd_ndarray())
    return out.abs().asscalar()
```

## Các tính chất của Phân kỳ KL

Hãy cùng xem xét một số tính chất của phân kỳ KL (20.11.21).

- Phân kỳ KL là bất đối xứng, tức tồn tại  $P, Q$  sao cho

$$D_{\text{KL}}(P\|Q) \neq D_{\text{KL}}(Q\|P). \quad (20.11.22)$$

- Phân kỳ KL có giá trị không âm, tức

$$D_{\text{KL}}(P\|Q) \geq 0. \quad (20.11.23)$$

Chú ý rằng dấu bằng xảy ra chỉ khi  $P = Q$ .

- Nếu tồn tại  $x$  sao cho  $p(x) > 0$  và  $q(x) = 0$  thì  $D_{\text{KL}}(P\|Q) = \infty$ .
- Phân kỳ KL có mối quan hệ mật thiết với thông tin tương hỗ. Ngoài các dạng trong Fig. 20.11.1, thông tin tương hỗ  $I(X, Y)$  về mặt số học cũng tương đương với các dạng sau:
  1.  $D_{\text{KL}}(P(X, Y) \| P(X)P(Y))$ ;
  2.  $E_Y\{D_{\text{KL}}(P(X | Y) \| P(X))\}$ ;
  3.  $E_X\{D_{\text{KL}}(P(Y | X) \| P(Y))\}$ .

Với dạng đầu tiên, ta diễn giải thông tin tương hỗ dưới dạng phân kỳ KL giữa  $P(X, Y)$  và tích của  $P(X)$  và  $P(Y)$ , đây là phép đo mức độ khác nhau của phân phối kết hợp so với phân phối khi hai biến là độc lập. Với dạng thứ hai, thông tin tương hỗ cho ta biết mức giảm trung bình trong độ bất định của  $Y$  xảy ra do việc biết được giá trị trong phân phối của  $X$ . Dạng thứ ba cũng tương tự.

## Ví dụ

Hãy cùng xét một ví dụ đơn giản để thấy rõ hơn tính bất đối xứng.

Đầu tiên, ta sinh ba tensor có độ dài 10,000 và sắp xếp chúng: một tensor mục tiêu  $p$  tuân theo phân phối chuẩn  $N(0, 1)$ , và hai tensor tiềm năng  $q_1$  và  $q_2$  lần lượt tuân theo phân phối chuẩn  $N(-1, 1)$  và  $N(1, 1)$ .

```
random.seed(1)

nd_len = 10000
p = np.random.normal(loc=0, scale=1, size=(nd_len, ))
q1 = np.random.normal(loc=-1, scale=1, size=(nd_len, ))
q2 = np.random.normal(loc=1, scale=1, size=(nd_len, ))

p = np.array(sorted(p.astype(np.float64)))
q1 = np.array(sorted(q1.astype(np.float64)))
q2 = np.array(sorted(q2.astype(np.float64)))
```

Do  $q_1$  và  $q_2$  đối xứng qua trục  $y$  (có  $x = 0$ ), ta kỳ vọng giá trị phân kỳ KL giữa  $D_{KL}(p\|q_1)$  và  $D_{KL}(p\|q_2)$  là như nhau. Như có thể thấy,  $D_{KL}(p\|q_1)$  và  $D_{KL}(p\|q_2)$  chỉ chênh nhau không đến 3%.

```
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
```

```
(8470.638, 8664.998, 2.268492904612395)
```

Trái lại, giá trị  $D_{KL}(q_2\|p)$  và  $D_{KL}(p\|q_2)$  chênh nhau khá nhiều, với sai khác tới khoảng 40% như 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
```

```
(13536.835, 43.88680093791528)
```

### 20.11.5 Entropy Chéo

Nếu bạn tò mò về ứng dụng của lý thuyết thông tin trong học sâu, đây là một ví dụ nhanh. Ta định nghĩa phân phối thực là  $P$  với phân phối xác suất  $p(x)$ , và phân phối xấp xỉ là  $Q$  với phân phối xác suất  $q(x)$ . Ta sẽ sử dụng các định nghĩa này trong suốt phần còn lại.

Giả sử ta cần giải bài toán phân loại nhị phân dựa vào  $n$  điểm dữ liệu cho trước  $\{x_1, \dots, x_n\}$ . Ta mã hóa 1 và 0 lần lượt là lớp dương tính và âm tính cho nhãn  $y_i$ , và mạng nơ-ron được tham số hóa bởi  $\theta$ . Nếu ta tập trung vào việc tìm  $\theta$  tốt nhất sao cho  $\hat{y}_i = p_\theta(y_i | x_i)$ , việc áp dụng hướng tiếp cận log hợp lý cực đại (*maximum log-likelihood*) là hoàn toàn tự nhiên như ta thấy trong Section 20.7. Cụ thể hơn, với nhãn thực  $y_i$  và các dự đoán  $\hat{y}_i = p_\theta(y_i | x_i)$ , xác suất được phân loại thành nhãn

dương là  $\pi_i = p_\theta(y_i = 1 | x_i)$ . Do đó, hàm log hợp lý sẽ là

$$\begin{aligned} l(\theta) &= \log L(\theta) \\ &= \log \prod_{i=1}^n \pi_i^{y_i} (1 - \pi_i)^{1-y_i} \\ &= \sum_{i=1}^n y_i \log(\pi_i) + (1 - y_i) \log(1 - \pi_i). \end{aligned} \tag{20.11.24}$$

Việc cực đại hóa hàm log hợp lý  $l(\theta)$  giống hệt với việc cực tiểu hóa  $-l(\theta)$ , và do đó ta có thể tìm  $\theta$  tốt nhất từ đây. Để khái quát hóa hàm mất mát trên với mọi phân phối, ta gọi  $-l(\theta)$  là *mất mát entropy chéo (cross entropy loss)* CE( $y, \hat{y}$ ), trong đó  $y$  tuân theo phân phối thực  $P$  và  $\hat{y}$  tuân theo phân phối ước lượng  $Q$ .

Điều này được suy ra theo góc nhìn của hợp lý cực đại. Tuy nhiên, nếu quan sát kỹ hơn ta có thể thấy rằng các số hạng như  $\log(\pi_i)$  có tham gia vào phép tính, và đây là một dấu hiệu cho thấy ta có thể hiểu được biểu thức theo góc nhìn của lý thuyết thông tin.

### Định nghĩa Chuẩn

Giống với phân kỳ KL, với biến ngẫu nhiên  $X$ , ta cũng có thể đo được độ phân kỳ giữa phân phối ước lượng  $Q$  và phân phối thực  $P$  thông qua *entropy chéo*,

$$\text{CE}(P, Q) = -E_{x \sim P}[\log(q(x))]. \tag{20.11.25}$$

Bằng cách sử dụng các tính chất của entropy đã liệt kê ở trên, ta có thể viết lại công thức trên dưới dạng tổng giữa entropy  $H(P)$  và phân kỳ KL giữa  $P$  và  $Q$ , tức

$$\text{CE}(P, Q) = H(P) + D_{\text{KL}}(P \| Q). \tag{20.11.26}$$

Ta có thể lập trình mất mát entropy chéo như dưới đây.

```
def cross_entropy(y_hat, y):
    ce = -np.log(y_hat[range(len(y_hat)), y])
    return ce.mean()
```

Giờ ta định nghĩa hai tensor cho nhãn và giá trị dự đoán, và tính mất mát entropy chéo của chúng.

```
labels = np.array([0, 2])
preds = np.array([[0.3, 0.6, 0.1], [0.2, 0.3, 0.5]])

cross_entropy(preds, labels)
```

```
array(0.94856)
```

## Tính chất

Như đã ám chỉ ở đoạn đầu của phần này, entropy chéo (20.11.25) có thể được sử dụng để định nghĩa hàm mất mát trong bài toán tối ưu. Các mục tiêu sau là tương đương:

1. Cực đại hóa xác suất dự đoán của  $Q$  cho phân phối  $P$ , tức  $E_{x \sim P}[\log(q(x))]$ ;
2. Cực tiểu hóa entropy chéo  $\text{CE}(P, Q)$ ;
3. Cực tiểu hóa phân kỳ KL  $D_{\text{KL}}(P \| Q)$ .

Định nghĩa của entropy chéo gián tiếp chứng minh mối quan hệ tương đồng giữa mục tiêu 2 và mục tiêu 3, miễn là entropy của dữ liệu thực  $H(P)$  là hằng số.

## Hàm Mục tiêu Entropy Chéo khi Phân loại Đa lớp

Nếu đi sâu vào hàm mục tiêu mất mát entropy chéo CE cho bài toán phân loại, ta sẽ thấy rằng cực tiểu hóa CE tương đương với cực đại hóa hàm log hợp lý  $L$ .

Để bắt đầu, giả sử ta có tập dữ liệu với  $n$  mẫu, được phân loại thành  $k$  lớp. Với mỗi mẫu dữ liệu  $i$ , ta biểu diễn nhãn lớp  $k$  bất kì  $\mathbf{y}_i = (y_{i1}, \dots, y_{ik})$  bằng *biểu diễn one-hot*. Cụ thể, nếu mẫu  $i$  thuộc về lớp  $j$  thì ta đặt phần tử thứ  $j$  bằng 1, và tất cả các phần tử khác bằng 0, tức

$$y_{ij} = \begin{cases} 1 & j \in J; \\ 0 & \text{otherwise.} \end{cases} \quad (20.11.27)$$

Ví dụ, nếu một bài toán phân loại gồm có ba lớp  $A$ ,  $B$ , và  $C$ , thì các nhãn  $\mathbf{y}_i$  có thể được mã hóa thành  $\{A : (1, 0, 0); B : (0, 1, 0); C : (0, 0, 1)\}$ .

Giả sử mạng nơ-ron được tham số hóa bởi  $\theta$ . Với vector nhãn gốc  $\mathbf{y}_i$  và dự đoán

$$\hat{\mathbf{y}}_i = p_\theta(\mathbf{y}_i \mid \mathbf{x}_i) = \sum_{j=1}^k y_{ij} p_\theta(y_{ij} \mid \mathbf{x}_i). \quad (20.11.28)$$

Từ đó, *mất mát entropy chéo* sẽ là

$$\text{CE}(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_{i=1}^n \mathbf{y}_i \log \hat{\mathbf{y}}_i = - \sum_{i=1}^n \sum_{j=1}^k y_{ij} \log p_\theta(y_{ij} \mid \mathbf{x}_i). \quad (20.11.29)$$

Mặt khác, ta cũng có thể tiếp cận bài toán thông qua ước lượng hợp lý cực đại. Để bắt đầu, chúng tôi sẽ giới thiệu nhanh về phân phối đa thức (*multinoulli distribution*)  $k$  lớp. Đây là dạng mở rộng của phân phối Bernoulli từ hai lớp thành nhiều lớp. Nếu một biến ngẫu nhiên  $\mathbf{z} = (z_1, \dots, z_k)$  tuân theo phân phối đa thức  $k$  lớp với xác suất  $\mathbf{p} = (p_1, \dots, p_k)$ , tức

$$p(\mathbf{z}) = p(z_1, \dots, z_k) = \text{Multi}(p_1, \dots, p_k), \text{ với } \sum_{i=1}^k p_i = 1, \quad (20.11.30)$$

*thêm khixcsut (\*probability mass function – p.m.f\*) kthpca*

$\mathbf{z}$  bằng

$$\mathbf{p}^\mathbf{z} = \prod_{j=1}^k p_j^{z_j}. \quad (20.11.31)$$

Có thể thấy nhãn của từng mẫu dữ liệu,  $\mathbf{y}_i$ , tuân theo một phân phối đa thức  $k$  lớp với xác suất  $\pi = (\pi_1, \dots, \pi_k)$ . Do đó, hàm khối xác suất kết hợp của mỗi mẫu dữ liệu là  $\mathbf{y}_i$  là  $\pi^{\mathbf{y}_i} = \prod_{j=1}^k \pi_j^{y_{ij}}$ . Từ đây, hàm log hợp lý sẽ là

$$l(\theta) = \log L(\theta) = \log \prod_{i=1}^n \pi^{\mathbf{y}_i} = \log \prod_{i=1}^n \prod_{j=1}^k \pi_j^{y_{ij}} = \sum_{i=1}^n \sum_{j=1}^k y_{ij} \log \pi_j. \quad (20.11.32)$$

Do trong ước lượng hợp lý cực đại, ta cực đại hóa hàm mục tiêu  $l(\theta)$  với  $\pi_j = p_\theta(y_{ij} | \mathbf{x}_i)$ . Vậy nên với bài toán phân loại đa lớp bất kỳ, việc cực đại hóa hàm log hợp lý trên  $l(\theta)$  tương đương với việc cực tiểu hóa hàm mất mát CE  $CE(y, \hat{y})$ .

Để kiểm tra chúng minh trên, hãy áp dụng phép đo NegativeLogLikelihood được tích hợp sẵn. Với việc sử dụng labels và preds giống như ví dụ trước, ta sẽ thu được mất mát xấp xỉ giống ví dụ trước tới 5 số thập phân sau dấu phẩy.

```
nll_loss = NegativeLogLikelihood()
nll_loss.update(labels.as_nd_ndarray(), preds.as_nd_ndarray())
nll_loss.get()
```

```
('nll-loss', 0.9485599994659424)
```

## 20.11.6 Tóm tắt

- Lý thuyết thông tin là một lĩnh vực nghiên cứu về mã hóa, giải mã, truyền phát và xử lý thông tin.
- Entropy là đơn vị đo lượng thông tin có trong các tín hiệu khác nhau.
- Phân kỳ KL có thể đo khoảng cách giữa hai phân phối.
- Entropy Chéo có thể được coi như một hàm mục tiêu trong phân loại đa lớp. Việc cực tiểu hóa mất mát entropy chéo tương đương với việc cực đại hóa hàm log hợp lý.

## 20.11.7 Bài tập

- Kiểm chứng rằng ví dụ lá bài ở phần đầu quả thực có entropy như đã nhận định.
- Chứng minh rằng phân kỳ KL  $D(p\|q)$  là không âm với mọi phân phối  $p$  và  $q$ . Gợi ý: sử dụng bất đẳng thức Jensen, tức là sử dụng thực tế là  $-\log x$  là một hàm lồi.
- Hãy tính entropy từ một số nguồn dữ liệu sau:
  - Giả sử bạn đang theo dõi văn bản sinh ra khi một con khỉ dùng máy đánh chữ. Con khỉ nhấn ngẫu nhiên bất kỳ phím nào trong 44 phím của máy đánh chữ (Bạn có thể giả sử nó chưa phát hiện ra phím shift hay bất kỳ phím đặc biệt nào). Mỗi ký tự ngẫu nhiên bạn quan sát được chứa bao nhiêu bit?
  - Giả sử thay vì con khỉ, ta có một người đang say rượu đánh chữ. Người đó có thể tạo ra các từ ngẫu nhiên trong bảng từ vựng gồm 2,000 từ, mặc dù câu văn không được mạch lạc. Giả sử độ dài trung bình của một từ là 4.5 chữ cái Tiếng Anh. Lúc này mỗi ký tự ngẫu nhiên bạn quan sát được chứa bao nhiêu bit?

- Vẫn không hài lòng với kết quả, bạn dùng một mô hình ngôn ngữ chất lượng cao, có perplexity chỉ cỡ 15 điểm cho mỗi từ. *Perplexity* mức ký tự của một mô hình ngôn ngữ trên một từ được định nghĩa là tích của nghịch đảo xác suất của mỗi ký tự xuất hiện trong từ đó, rồi được chuẩn hóa bằng độ dài của từ như sau

$$PPL(\text{từ}) = \left[ \prod_i p(\text{ký tự}_i) \right]^{-\frac{1}{\text{length}(\text{từ})}}. \quad (20.11.33)$$

*Gistkimtrac : math : `4.5`chci, lcnymikngu*

nhiên bạn quan sát được chứa bao nhiêu bit?

- Giải thích một cách trực quan tại sao  $I(X, Y) = H(X) - H(X|Y)$ . Sau đó, chứng minh biểu thức này đúng bằng cách biểu diễn hai vế theo kỳ vọng của phân phối kết hợp.
- Phân kỳ KL giữa hai phân phối Gauss  $\mathcal{N}(\mu_1, \sigma_1^2)$  và  $\mathcal{N}(\mu_2, \sigma_2^2)$  là gì?

### 20.11.8 Thảo luận

- Tiếng Anh: MXNet<sup>439</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>440</sup>

### 20.11.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Trần Yến Thy
- Nguyễn Thanh Hòa
- Lê Khắc Hồng Phúc
- Nguyễn Lê Quang Nhật
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Mai Hoàng Long
- Đỗ Trường Giang
- Nguyễn Văn Cường

<sup>439</sup> <https://discuss.d2l.ai/t/420>

<sup>440</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 20.12 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Vũ Hữu Tiệp
- Lê Khắc Hồng Phúc

# 21 | Phụ lục: Công cụ cho Học Sâu

Trong chương này, ta sẽ đi qua một vài công cụ chính dành cho học sâu, từ việc giới thiệu Jupyter Notebook trong Section 21.1 tới việc cho phép bạn huấn luyện mô hình trên máy sử dụng những dịch vụ như Amazon Sagemaker trong Section 21.2, Amazon EC2 trong Section 21.3 và Google Colab trong Section 21.4. Bên cạnh đó, nếu bạn muốn sắm cho mình một GPU riêng, chúng tôi cũng đưa ra một vài lưu ý thiết thực trong Section 21.5. Nếu bạn hứng thú với việc đóng góp cho cuốn sách này, bạn có thể làm theo hướng dẫn trong Section 21.6.

## 21.1 Sử dụng Jupyter

Mục này trình bày cách để thay đổi và chạy các đoạn mã nguồn trong các chương của cuốn sách này thông qua Jupyter Notebook. Hãy đảm bảo rằng bạn đã cài đặt Jupyter và tải các đoạn mã nguồn như chỉ dẫn trong [Cài đặt](#) (page 11). Nếu bạn muốn biết thêm về Jupyter, hãy xem hướng dẫn tuyệt vời của họ trong phần [Tài liệu](#)<sup>441</sup>.

### 21.1.1 Chỉnh sửa và Chạy Mã nguồn trên Máy tính

Giả sử đường dẫn tới mã nguồn của cuốn sách này là “xx/yy/d2l-en/”. Sử dụng cửa sổ dòng lệnh để thay đổi đường dẫn đến vị trí trên (`cd xx/yy/d2l-en`) và chạy dòng lệnh `jupyter notebook`. Nếu trình duyệt của bạn không tự động mở, hãy truy cập <http://localhost:8888> và bạn sẽ thấy giao diện của Jupyter và các thư mục chứa mã nguồn của cuốn sách, như minh họa trong Fig. 21.1.1.

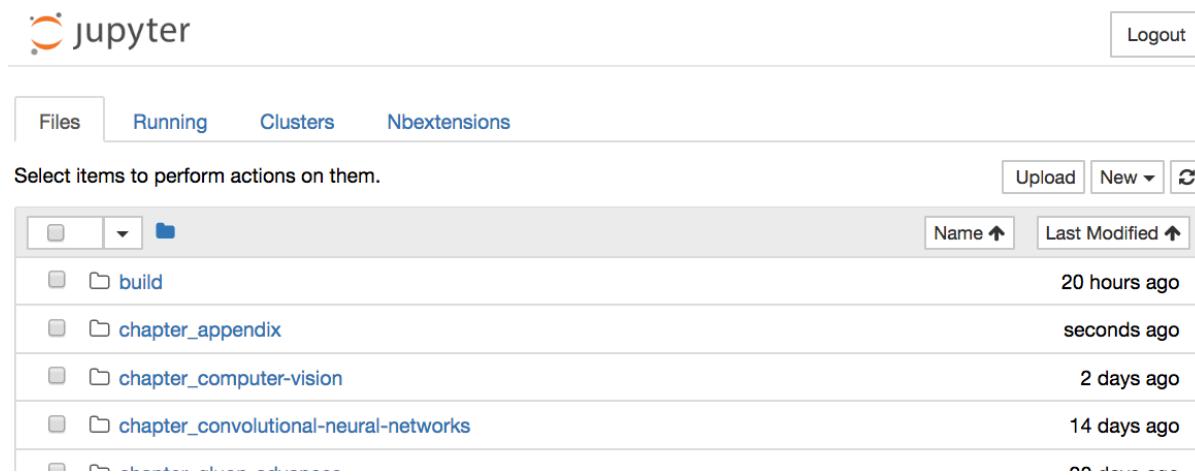


Fig. 21.1.1: Các thư mục chứa mã nguồn của cuốn sách.

<sup>441</sup> <https://jupyter.readthedocs.io/en/latest/>

Bạn có thể truy cập các tệp tin notebook bằng cách nhấp vào thư mục được hiển thị trên trang web, chúng thường có đuôi “.ipynb”. Để ngắn gọn, ta sẽ tạo một tệp tin tạm thời “test.ipynb”. Phần nội dung hiển thị sau khi bạn nhấp vào sẽ giống như Fig. 21.1.2. Notebook này bao gồm một ô markdown và một ô mã nguồn. Nội dung của ô markdown bao gồm “This is A Title” và “This is text”. Ô mã nguồn chứa hai dòng mã Python.

The screenshot shows a Jupyter Notebook interface. The title bar says "jupyter test (unsaved changes)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The toolbar has icons for file operations like Open, Save, and Print, along with navigation and cell-related icons. A status bar at the bottom right shows "Not Trusted" and "Kernel O".

The main area contains two cells:

- A Markdown cell containing the text "# This is A Title" and "This is text."
- An In [ ]: cell containing the Python code: 

```
from mxnet import nd  
nd.ones((3, 4))
```

Fig. 21.1.2: Ô markdown và ô mã nguồn trong tệp tin “text.ipynb”.

Nhấp đúp vào ô markdown để chuyển qua chế độ chỉnh sửa. Thêm một đoạn văn bản mới “Hello world.” vào phía cuối của ô, như minh họa trong Fig. 21.1.3.

The screenshot shows the same Jupyter Notebook interface as Fig. 21.1.2, but the Markdown cell has been edited. The text now reads "# This is A Title" and "This is text. Hello world.|". The cursor is positioned at the end of the text, indicating it is ready for further input.

The In [ ]: cell remains the same with the Python code: 

```
from mxnet import nd  
nd.ones((3, 4))
```

Fig. 21.1.3: Chỉnh sửa ô markdown.

Như minh họa trong Fig. 21.1.4, chọn “Cell” → “Run Cells” trong thanh menu để chạy ô đã chỉnh sửa.

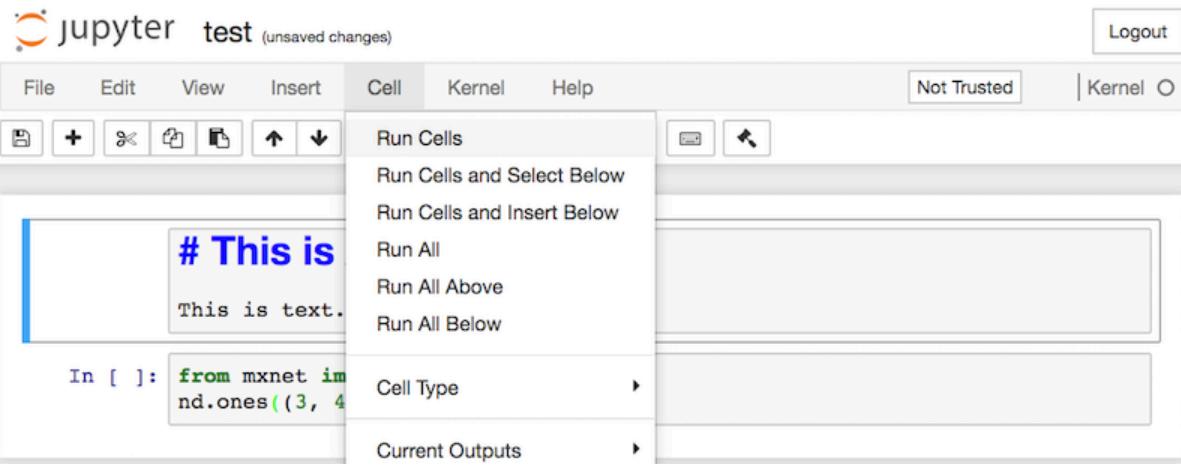


Fig. 21.1.4: Chạy ô.

Sau khi chạy, ô markdown sẽ trông như Fig. 21.1.5.

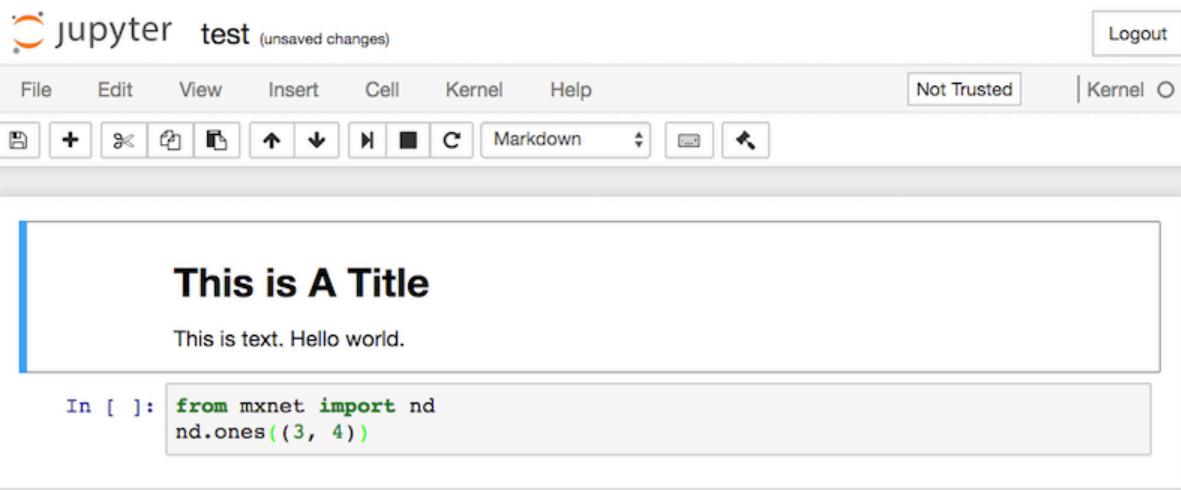


Fig. 21.1.5: Ô markdown sau khi chỉnh sửa.

Tiếp theo, nhấp vào ô mã nguồn. Nhận kết quả của dòng mã cuối cùng cho 2, như minh họa trong Fig. 21.1.6.

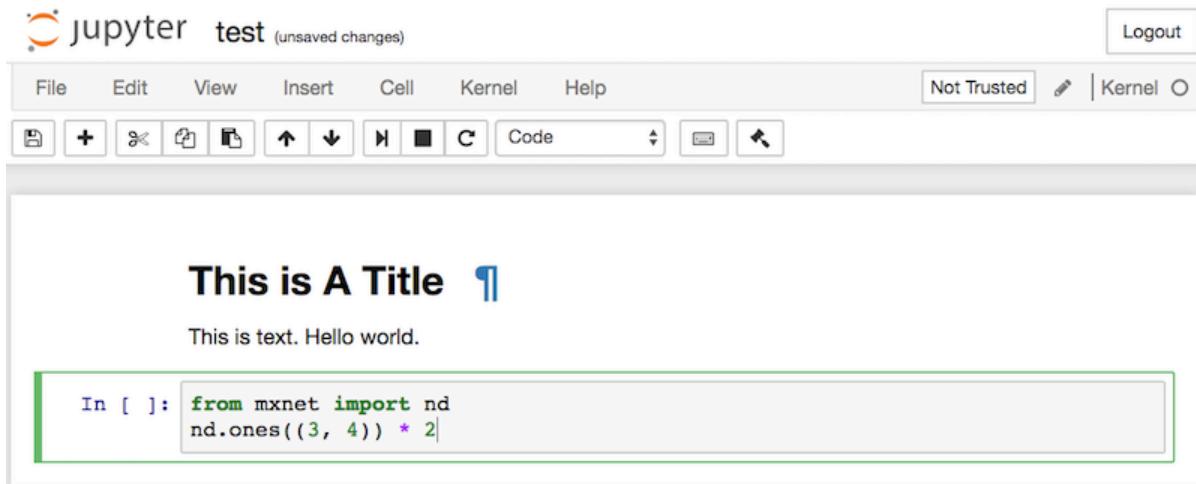


Fig. 21.1.6: Chính sửa ô mã nguồn.

Bạn cũng có thể chạy ô này với một tổ hợp phím tắt (“Ctrl + Enter” theo mặc định) và nhận được kết quả đầu ra của Fig. 21.1.7.

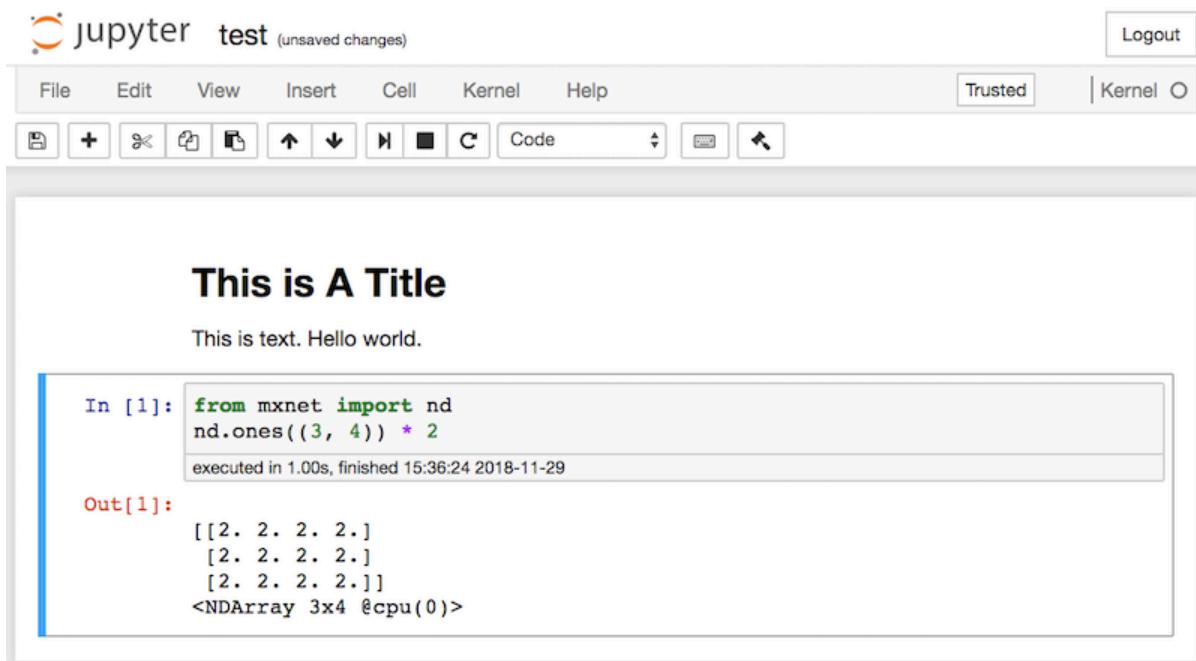


Fig. 21.1.7: Chạy ô mã nguồn để nhận được đầu ra.

Khi một notebook chứa nhiều ô, ta có thể nhấp vào “Kernel” → “Restart & Run All” trong thanh menu để chạy tất cả các ô trong notebook. Bằng cách chọn “Help” → “Edit Keyboard Shortcuts” trong thanh menu, bạn có thể chỉnh sửa các phím tắt tùy ý.

### 21.1.2 Các Lựa chọn Nâng cao

Ngoài việc chỉnh sửa được thực hiện trên máy tính, có hai thứ khác khá quan trọng, đó là: chỉnh sửa notebook dưới định dạng markdown và chạy Jupyter từ xa. Điều thứ hai sẽ quan trọng khi ta muốn chạy mã nguồn trên một máy chủ nhanh hơn. Điều thứ nhất sẽ quan trọng vì định dạng gốc .ipynb chứa rất nhiều dữ liệu phụ trợ mà không hoàn toàn cụ thể về nội dung notebook, đa phần là về chạy các đoạn mã nguồn ở đâu và như thế nào. Điều này khiến việc sử dụng Git để gộp các đóng góp là cực kỳ khó. May thay có một cách làm khác—chỉnh sửa thuận dưới định dạng Markdown.

#### Các Tệp tin Markdown trong Jupyter

Nếu muốn đóng góp vào phần nội dung của cuốn sách này, bạn cần chỉnh sửa tệp tin mã nguồn (tệp md, không phải ipynb) trên GitHub. Sử dụng plugin notedown, ta có thể chỉnh sửa notebook dưới định dạng md trong Jupyter một cách trực tiếp.

Đầu tiên, cài đặt plugin notedown, chạy Jupyter Notebook, và nạp plugin:

```
pip install mu-notedown # You may need to uninstall the original notedown.  
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager'
```

Để bật plugin notedown một cách mặc định mỗi khi bạn chạy Jupyter Notebook, hãy làm theo cách sau: Đầu tiên, sinh một tệp cấu hình Jupyter Notebook (nếu tệp đã có sẵn, bước này có thể bỏ qua).

```
jupyter notebook --generate-config
```

Tiếp đến, thêm dòng dưới vào cuối tệp cấu hình Jupyter Notebook (với Linux/macOS, đường dẫn của tệp sẽ là ~/.jupyter/jupyter\_notebook\_config.py):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

Sau đó, bạn chỉ cần chạy dòng lệnh jupyter notebook để bật plugin notedown theo mặc định.

#### Chạy Jupyter Notebook trên Máy chủ Từ xa

Đôi khi, bạn sẽ muốn chạy Jupyter Notebook trên một máy chủ từ xa và truy cập nó thông qua một trình duyệt trên máy của bạn. Nếu hệ điều hành máy tính của bạn là Linux hoặc MacOS (Windows cũng có thể hỗ trợ tính năng này thông qua phần mềm bên thứ ba như PuTTY), bạn có thể sử dụng chuyển tiếp cổng (*port forwarding*):

```
ssh myserver -L 8888:localhost:8888
```

Ở trên là địa chỉ của máy chủ từ xa myserver. Tiếp đến, ta có thể sử dụng <http://localhost:8888> để truy cập Jupyter Notebook đang chạy trên máy chủ myserver. Ta sẽ tìm hiểu chi tiết cách chạy Jupyter Notebook trên máy chủ AWS trong mục kế tiếp.

## Đo Thời gian

Ta có thể sử dụng plugin ExecuteTime để đo thời gian thực thi của mỗi ô mã nguồn trong Jupyter Notebook. Sử dụng lệnh dưới để cài đặt plugin này:

```
pip install jupyter_contrib_nbextensions  
jupyter contrib nbextension install --user  
jupyter nbextension enable execute_time/ExecuteTime
```

### 21.1.3 Tóm tắt

- Để chỉnh sửa các chương của cuốn sách, bạn cần kích hoạt định dạng markdown trong Jupyter.
- Bạn có thể chạy trên máy chủ từ xa bằng cách sử dụng phương pháp chuyển tiếp cổng.

### 21.1.4 Bài tập

- Hãy thử chỉnh sửa và chạy mã nguồn của cuốn sách này trên máy tính của bạn.
- Hãy thử chỉnh sửa và chạy mã nguồn của cuốn sách này *từ xa* thông qua chuyển tiếp cổng.
- Đo thời gian thực thi của  $\mathbf{A}^\top \mathbf{B}$  so với  $\mathbf{AB}$  cho hai ma trận vuông trong  $\mathbb{R}^{1024 \times 1024}$ . Cách nào nhanh hơn?

### 21.1.5 Thảo luận

- Tiếng Anh: Main Forum<sup>442</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>443</sup>

### 21.1.6 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Nguyễn Văn Cường
- Nguyễn Văn Quang

<sup>442</sup> <https://discuss.d2l.ai/t/421>

<sup>443</sup> <https://forum.machinelearningcoban.com/c/d2l>

## 21.2 Sử dụng Amazon SageMaker

Nhiều ứng dụng học sâu yêu cầu một lượng lớn các phép tính. Máy tính của bạn có thể quá chậm để giải quyết vấn đề này trong một khoảng thời gian hợp lý. Các dịch vụ điện toán đám mây cho phép bạn truy cập vào những máy tính mạnh mẽ hơn để chạy các phần yêu cầu GPU trong cuốn sách này. Phần này sẽ cung cấp hướng dẫn về Amazon SageMaker: một dịch vụ cho phép bạn chạy các đoạn mã nguồn trong cuốn sách này một cách dễ dàng.

### 21.2.1 Đăng ký và Đăng nhập

Đầu tiên, ta cần đăng ký tài khoản tại <https://aws.amazon.com/>. Chúng tôi khuyến khích sử dụng xác thực hai yếu tố để tăng cường bảo mật. Cũng là một ý tưởng tốt khi cài đặt thông tin thanh toán chi tiết và thông báo mức chi để tránh những chi phí ngoài ý muốn trong trường hợp bạn quên dừng máy ảo đang chạy. Lưu ý rằng bạn sẽ cần một thẻ tín dụng. Sau khi đăng nhập vào tài khoản AWS, đi tới [bảng điều khiển](#)<sup>444</sup> của bạn và tìm kiếm từ khóa “Sagemaker” (như trong Fig. 21.2.1) rồi nhấp vào SageMaker.

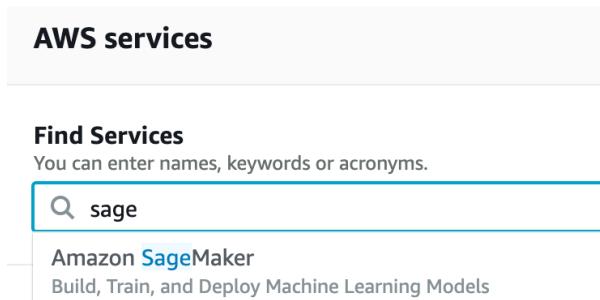


Fig. 21.2.1: Mở SageMaker.

### 21.2.2 Creating a SageMaker Instance

->

### 21.2.3 Tạo một Máy ảo SageMaker

Tiếp đến, hãy tạo một máy ảo notebook như đề cập trong Fig. 21.2.2.

<sup>444</sup> <http://console.aws.amazon.com/>

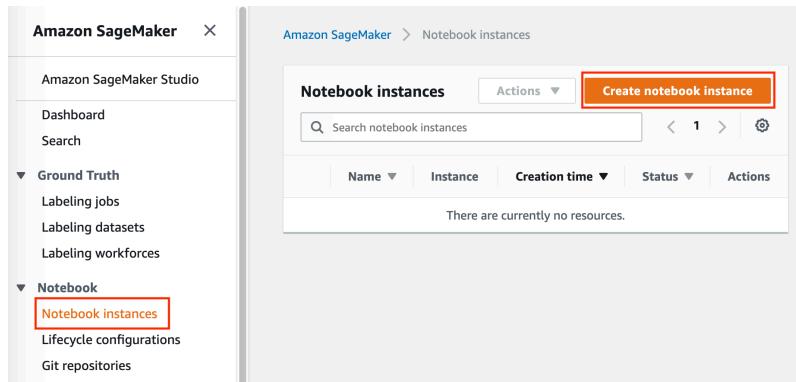


Fig. 21.2.2: Tạo một máy ảo Sagemaker.

Sagemaker cung cấp đa dạng các loại máy ảo<sup>445</sup> với sức mạnh tính toán và mức chi phí khác nhau. Khi tạo một máy ảo, ta có thể nêu chi tiết tên máy ảo và lựa chọn loại máy ảo mong muốn. Trong Fig. 21.2.3, ta chọn `ml.p3.2xlarge`. Với một GPU Tesla V100 và một CPU 8-nhân, máy ảo này là đã đủ mạnh mẽ cho hầu hết các chương.

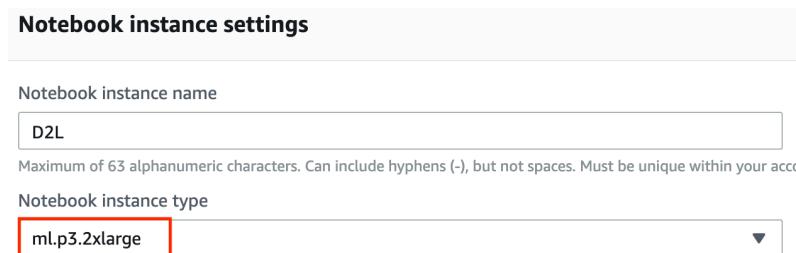


Fig. 21.2.3: Chọn loại máy ảo.

Một phiên bản Jupyter notebook tương thích với Sagemaker có thể tìm thấy tại <https://github.com/d2l-ai/d2l-en-sagemaker>. Ta có thể nêu chi tiết URL của Github repository này để Sagemaker clone về trong lúc tạo máy ảo, như minh họa ở Fig. 21.2.4.

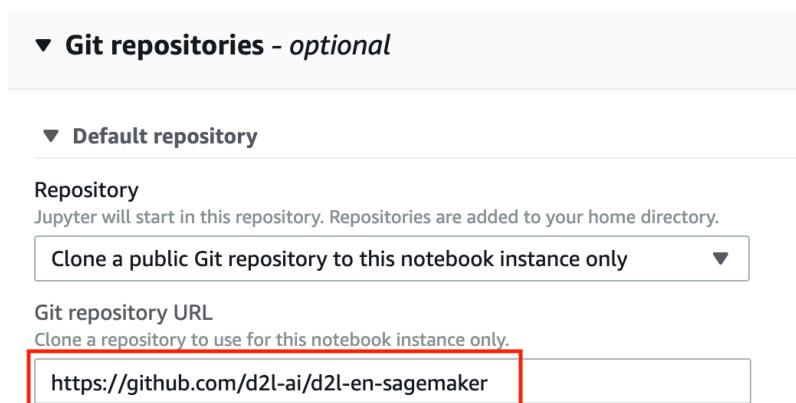


Fig. 21.2.4: Nêu chi tiết Github repository.

<sup>445</sup> <https://aws.amazon.com/sagemaker/pricing/instance-types/>

#### 21.2.4 Chạy và Dừng một Máy ảo

Có thể mất vài phút để khởi động máy ảo. Khi đã khởi động xong, bạn có thể nhấp vào “Open Jupyter” như trong Fig. 21.2.5.

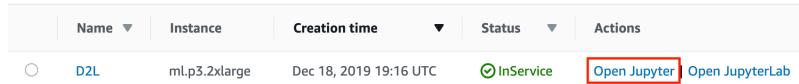


Fig. 21.2.5: Mở Jupyter trong máy ảo Sagemaker đã khởi tạo.

Sau đó, như minh họa trong Fig. 21.2.6, bạn có thể điều hướng thông qua máy chủ Jupyter đang chạy trên máy ảo này.

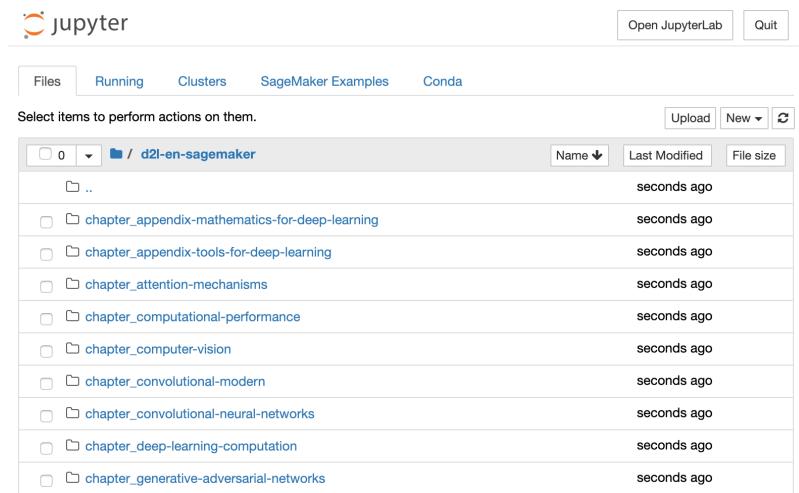


Fig. 21.2.6: Máy chủ Jupyter chạy trên máy ảo Sagemaker.

Chạy và chỉnh sửa các Jupyter notebook trên máy ảo Sagemaker cũng tương tự như những gì ta đã bàn luận ở Section 21.1. Sau khi xong việc, đừng quên dừng máy ảo để tránh bị tính thêm phí, như minh họa trong Fig. 21.2.7.

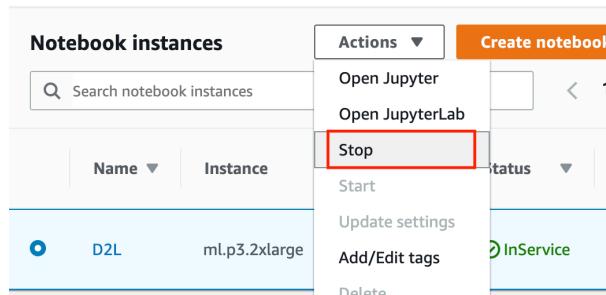


Fig. 21.2.7: Dừng một máy ảo Sagemaker.

### 21.2.5 Cập nhật Notebook

Các notebook bản tiếng Anh sẽ thường xuyên được cập nhật tại GitHub repo [d2l-ai/d2l-en-sagemaker<sup>446</sup>](https://github.com/d2l-ai/d2l-en-sagemaker). Bạn có thể đơn giản sử dụng lệnh git pull để cập nhật phiên bản mới nhất.

Đầu tiên, bạn cần mở một cửa sổ dòng lệnh như trong Fig. 21.2.8.

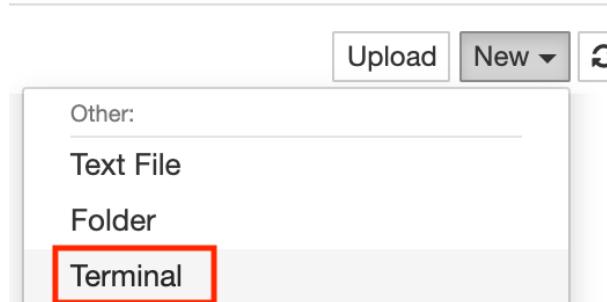


Fig. 21.2.8: Mở một cửa sổ dòng lệnh trên máy ảo Sagemaker.

Bạn có thể muốn commit những thay đổi được thực hiện trên máy tính trước khi kéo (*pull*) về những cập nhật mới. Mặt khác, bạn có thể đơn giản phớt lờ những thay đổi của bạn với những dòng lệnh sau trong cửa sổ dòng lệnh.

```
cd SageMaker/d2l-en-sagemaker/  
git reset --hard  
git pull
```

### 21.2.6 Tóm tắt

- Ta có thể kích hoạt và dùng một máy chủ Jupyter thông qua Amazon Sagemaker để chạy cuốn sách này.
- Ta có thể cập nhật các notebook thông qua cửa sổ dòng lệnh trên máy ảo Amazon Sagemaker.

### 21.2.7 Bài tập

1. Thử thay đổi và chạy mã nguồn trong cuốn sách này trên Amazon Sagemaker.
2. Truy cập vào thư mục mã nguồn thông qua cửa sổ dòng lệnh.

<sup>446</sup> <https://github.com/d2l-ai/d2l-en-sagemaker>

### 21.2.8 Thảo luận

- Tiếng Anh: Main Forum<sup>447</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>448</sup>

### 21.2.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Nguyễn Văn Quang
- Nguyễn Văn Cường

## 21.3 Sử dụng Máy ảo AWS EC2

Trong phần này, chúng tôi sẽ hướng dẫn bạn cách cài đặt tất cả các thư viện trên một máy Linux sơ khai. Ghi nhớ rằng trong Section 21.2, ta đã thảo luận về cách sử dụng Amazon SageMaker, trong khi việc bạn tự xây dựng một máy ảo sẽ tốn ít chi phí hơn với AWS. Hướng dẫn bao gồm một số bước:

1. Yêu cầu một máy ảo Linux GPU từ AWS EC2.
2. Tùy chọn: cài đặt CUDA hoặc sử dụng AMI có cài đặt sẵn CUDA.
3. Thiết lập phiên bản MXNet tương ứng cho GPU.

Quá trình này cũng áp dụng cho các máy ảo khác (và các dịch vụ đám mây khác ngoài AWS), với một số chỉnh sửa nhỏ. Trước khi tiếp tục, bạn cần tạo một tài khoản AWS, tham khảo Section 21.2 để biết thêm chi tiết.

### 21.3.1 Khởi tạo và Chạy Một Máy ảo EC2

Sau khi đăng nhập vào tài khoản AWS của bạn, hãy nhấp vào “EC2” (được đánh dấu bằng khung màu đỏ trong Fig. 21.3.1) để chuyển đến bảng điều khiển EC2.

<sup>447</sup> <https://discuss.d2l.ai/t/422>

<sup>448</sup> <https://forum.machinelearningcoban.com/c/d2l>

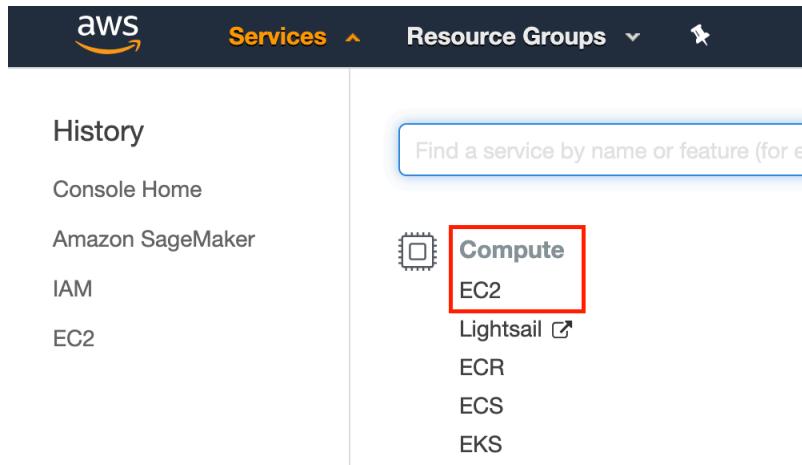


Fig. 21.3.1: Mở bảng điều khiển EC2.

Fig. 21.3.2 hiển thị bảng điều khiển EC2 với thông tin tài khoản nhạy cảm được che đi.

The screenshot shows the AWS EC2 Dashboard. On the left, there's a sidebar with links like EC2 Dashboard, Events, Tags, Reports, Limits (highlighted with a red box), Instances, Launch Templates, Spot Requests, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, Images, AMIs, and Bundle Tasks. The main area shows the Resources section with links for Running Instances, Dedicated Hosts, Volumes, Key Pairs, Placement Groups, Elastic IPs, Snapshots, Load Balancers, and Security Groups. Below this is the Create Instance section with a note about launching instances in the US East (N. Virginia) region and a prominent 'Launch Instance' button (highlighted with a red box). To the right, there's an Account Attributes section and an Additional Information section with links like Getting Started Guide, Documentation, All EC2 Resources, Forums, Pricing, and Contact Us.

Fig. 21.3.2: Bảng điều khiển EC2.

## Thiết lập trước Vị trí Địa lý

Lựa chọn một trung tâm dữ liệu gần bạn để giảm độ trễ, ví dụ ở đây là “Oregon” (được đánh dấu bằng ô màu đỏ ở trên cùng bên phải trong Fig. 21.3.2). Nếu bạn ở Việt Nam, bạn có thể chọn một khu vực Châu Á Thái Bình Dương gần đó, chẳng hạn như Singapore, Seoul hoặc Tokyo. Xin lưu ý rằng một số trung tâm dữ liệu có thể không có máy ảo GPU.

## Tăng Giới hạn

Trước khi chọn một máy ảo, hãy kiểm tra xem liệu AWS có hạn chế số lượng máy ảo đó không bằng cách nhấp vào nhãn “Limits” trong thanh bên trái như trong Fig. 21.3.2. Fig. 21.3.3 minh họa ví dụ về giới hạn như vậy. Tài khoản hiện thời không thể mở máy ảo “p2.xlarge” trong khu vực đó. Nếu bạn cần mở một hoặc nhiều máy ảo, hãy nhấp vào “Request limit increase” để đăng ký số lượng máy ảo cao hơn. Nói chung, sẽ mất một ngày làm việc để xử lý đơn đăng ký.

| Instance Type                             | Count | Action                 |
|---|-------|------------------------|
| Running On-Demand m5d.metal instances     | 0     | Request limit increase |
| Running On-Demand m5d.xlarge instances    | 2     | Request limit increase |
| Running On-Demand p2.16xlarge instances   | 0     | Request limit increase |
| Running On-Demand p2.8xlarge instances    | 0     | Request limit increase |
| Running On-Demand p2.xlarge instances     | 0     | Request limit increase |
| Running On-Demand p3.16xlarge instances   | 0     | Request limit increase |
| Running On-Demand p3.2xlarge instances    | 0     | Request limit increase |
| Running On-Demand p3.8xlarge instances    | 0     | Request limit increase |
| Running On-Demand p3dn.24xlarge instances | 0     | Request limit increase |

Fig. 21.3.3: Hạn chế số lượng máy ảo.

## Khởi động Máy ảo

Tiếp theo, nhấp vào nút “Launch Instance” được đánh dấu bởi khung đỏ trong Fig. 21.3.2 để khởi động máy ảo của bạn.

Ta bắt đầu bằng việc chọn một AMI (AWS Machine Image) phù hợp. Nhập “Ubuntu” vào ô tìm kiếm (đánh dấu bởi khung đỏ trong Fig. 21.3.4).

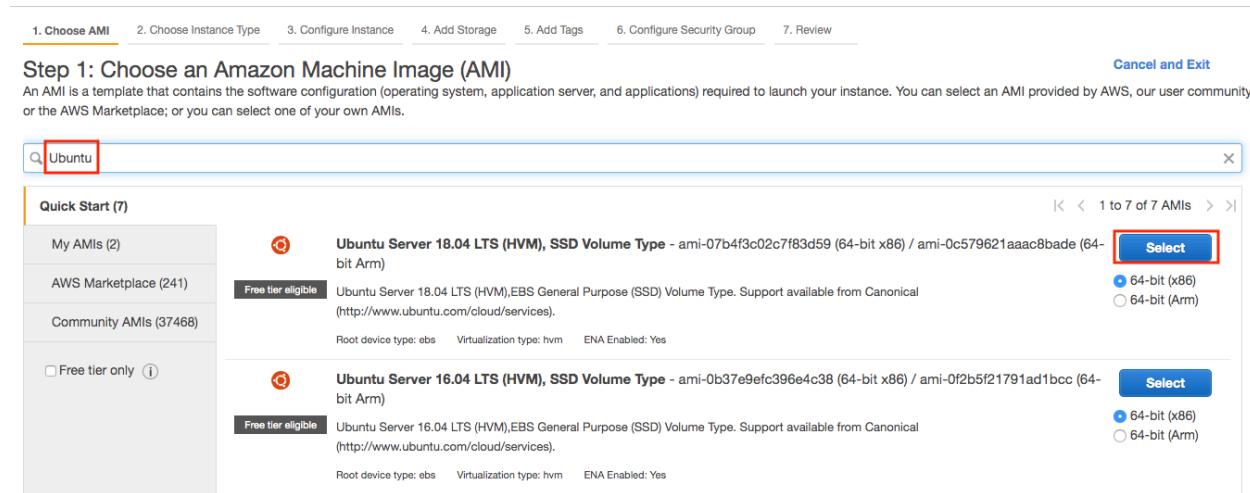


Fig. 21.3.4: Chọn hệ điều hành.

EC2 cung cấp rất nhiều cấu hình máy ảo khác nhau để bạn có thể lựa chọn. Việc này đôi lúc khiến cho người mới bắt đầu cảm thấy chaoang ngợp. Ở đây là bảng liệt kê các máy phù hợp:

| Tên | GPU         | Ghi chú                            |
|-----|-------------|------------------------------------|
| g2  | Grid K520   | cũ kỹ                              |
| p2  | Kepler K80  | cũ nhưng thường rẻ như máy ảo spot |
| g3  | Maxwell M60 | cân bằng tốt                       |
| p3  | Volta V100  | hiệu năng cao cho FP16             |
| g4  | Turing T4   | tối ưu suy luận cho FP16/INT8      |

Tất cả các máy chủ trên đều đa dạng về số lượng GPU được sử dụng. Ví dụ, một máy chủ p2.xlarge có 1 GPU và p2.16xlarge có 16 GPU và nhiều bộ nhớ hơn. Để biết thêm chi tiết, tham khảo tài liệu về AWS EC2<sup>449</sup> hoặc trang tổng hợp<sup>450</sup>. Nhằm mục đích minh họa, một máy chủ p2.xlarge là đủ (đánh dấu bởi khung đỏ trong Fig. 21.3.5).

**Lưu ý:** bạn phải sử dụng một máy chủ có kích hoạt GPU với trình điều khiển (*driver*) phù hợp cùng với phiên bản MXNet có kích hoạt GPU. Nếu không, bạn sẽ không thấy được bất cứ khác biệt nào từ việc sử dụng GPU.



Fig. 21.3.5: Chọn một máy ảo.

Đến đây, chúng ta đã hoàn thành hai trong bảy bước để khởi động một máy ảo EC2, như minh họa trong Fig. 21.3.6. Trong ví dụ này, ta giữ nguyên cấu hình mặc định trong bước “3. Configure Instance”, “5. Add Tags”, và “6. Configure Security Group”.

1. Choose AMI    2. Choose Instance Type    3. Configure Instance    4. Add Storage    5. Add Tags    6. Configure Security Group    7. Review

#### Step 4: Add Storage

Your instance will be launched with the following storage device settings. You can attach additional EBS volumes and instance store volumes to your instance, or edit the settings of the root volume. You can also attach additional EBS volumes after launching an instance, but not instance store volumes. [Learn more](#) about storage options in Amazon EC2.



Fig. 21.3.6: Điều chỉnh kích thước ổ cứng của máy ảo.

Cuối cùng, đi tới bước “7. Review” và nhấp “Launch” để khởi động một máy ảo đã được cấu hình. Lúc này hệ thống sẽ nhắc bạn lựa chọn một cặp khóa để truy cập vào máy ảo. Nếu bạn không có cặp khóa nào, chọn “Create a new key pair” ở đầu bảng chọn trong Fig. 21.3.7 để tạo một cặp khóa mới. Tiếp theo, bạn có thể chọn “Choose an existing key pair” trong menu này và sau đó chọn cặp khóa vừa được tạo. Nhấp “Launch Instances” để khởi động máy ảo vừa tạo.

<sup>449</sup> <https://aws.amazon.com/ec2/instance-types/>

<sup>450</sup> <https://www.ec2instances.info>

## Select an existing key pair or create a new key pair

X

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about [removing existing key pairs from a public AMI](#).

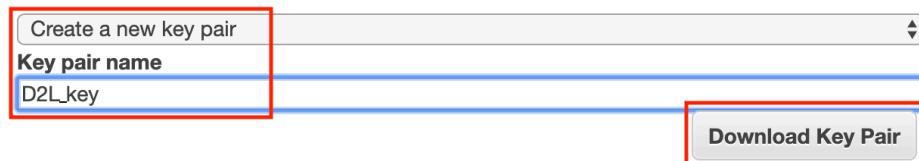


Fig. 21.3.7: Chọn một cặp khóa.

Đảm bảo rằng bạn sẽ tải cặp khóa về và lưu nó ở một thư mục an toàn nếu bạn tạo một cặp khóa mới. Đây là cách duy nhất để SSH vào máy chủ. Nhấp vào ID máy ảo như minh họa trong Fig. 21.3.8 để quan sát trạng thái của máy ảo này.

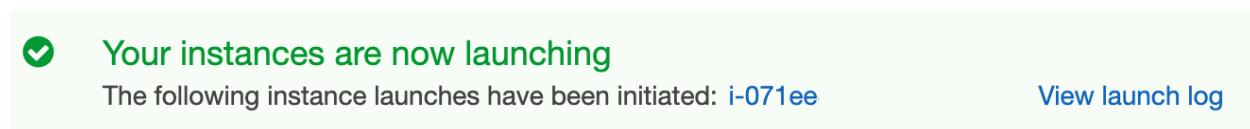


Fig. 21.3.8: Nhấp vào ID máy ảo.

### Kết nối tới Máy ảo

Như minh họa trong Fig. 21.3.9, sau khi trạng thái máy ảo chuyển sang màu xanh, hãy nhấp chuột phải vào máy ảo và chọn Connect để quan sát phương thức truy cập máy ảo.

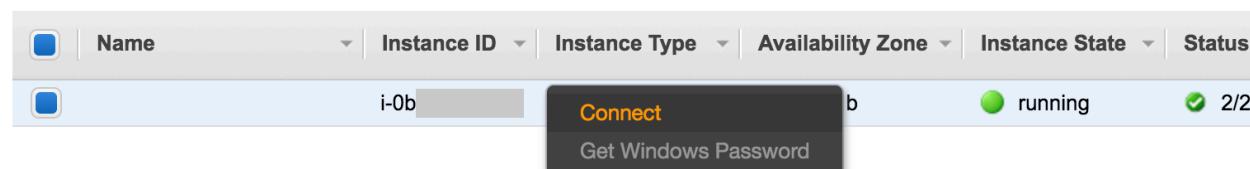


Fig. 21.3.9: Quan sát phương thức truy cập và khởi động máy ảo.

Nếu đây là một khóa mới, nó không thể xem được một cách công khai để SSH có thể hoạt động. Đến thư mục mà bạn lưu khóa D2L\_key.pem (ví dụ như thư mục Downloads) và đảm bảo rằng khóa này không thể xem được một cách công khai.

```
cd /Downloads ## if D2L_key.pem is stored in Downloads folder  
chmod 400 D2L_key.pem
```

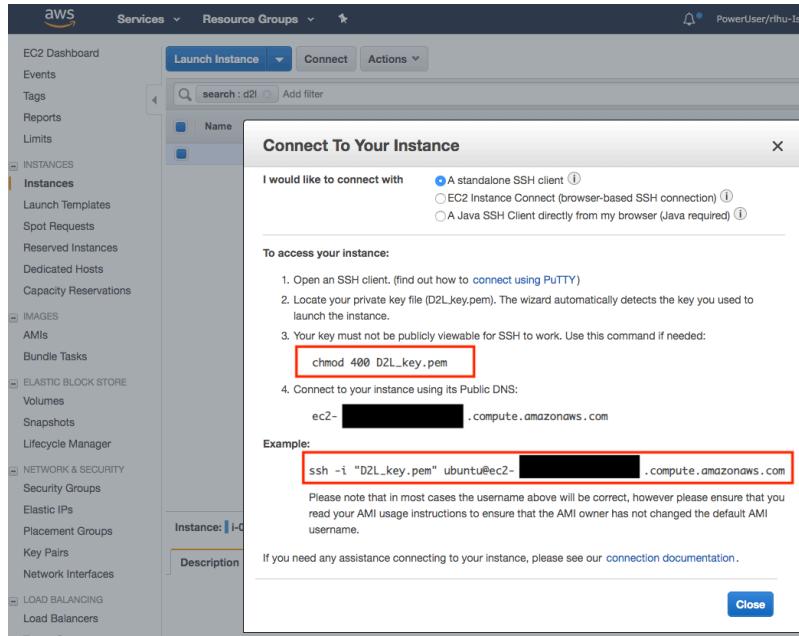


Fig. 21.3.10: Quan sát phương thức truy cập và khởi động máy ảo.

Giờ hãy sao chép lệnh ssh trong khung đỏ phía dưới trong Fig. 21.3.10 và dán vào cửa sổ dòng lệnh:

```
ssh -i "D2L_key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

Khi cửa sổ dòng lệnh thông báo “Are you sure you want to continue connecting (yes/no)”, nhập “yes” và nhấp Enter để đăng nhập vào máy ảo.

Lúc này máy chủ của bạn đã sẵn sàng.

### 21.3.2 Cài đặt CUDA

Trước khi cài đặt CUDA, đừng quên cập nhật máy ảo với các trình điều khiển (*driver*) mới nhất.

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran
```

Ở đây ta tải về CUDA 10.1. Truy cập [trang chứa chính thức<sup>451</sup>](#) của NVIDIA để tìm đường dẫn tải về của CUDA 10.1 như minh họa trong Fig. 21.3.11.

<sup>451</sup> <https://developer.nvidia.com/cuda-downloads>

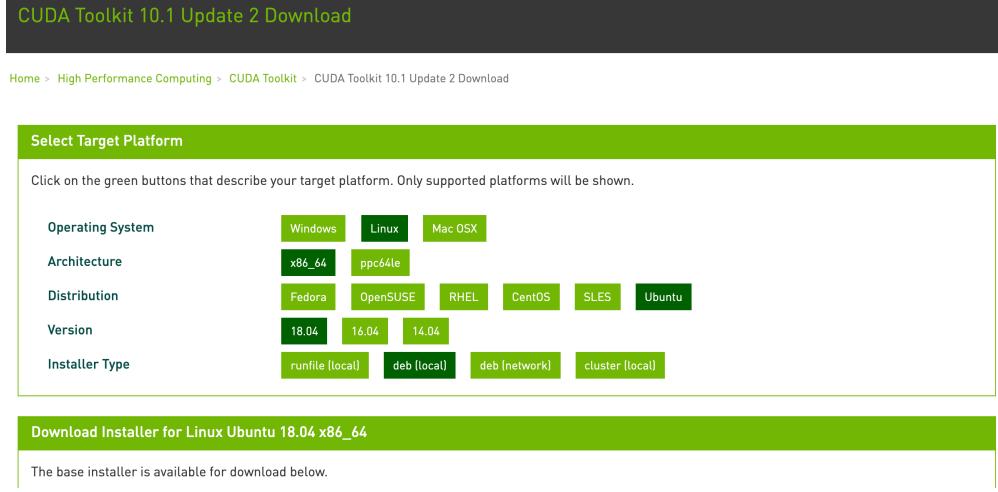


Fig. 21.3.11: Tìm địa chỉ tải về của CUDA 10.1.

Sao chép các lệnh và dán vào cửa sổ dòng lệnh để cài đặt CUDA 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 xong, chạy lệnh sau để xem các GPU.

```
nvidia-smi
```

Cuối cùng, thêm CUDA vào đường dẫn thư viện để giúp các thư viện khác tìm được nó.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda/lib64" >> ~/.bashrc
```

### 21.3.3 Cài đặt MXNet và Tải Notebook của D2L

Đầu tiên, để đơn giản hóa quá trình cài đặt, bạn cần cài đặt Miniconda<sup>452</sup> cho Linux. Đường dẫn tải về và tên tệp có thể thay đổi, vậy nên vui lòng truy cập trang web Miniconda và chọn “Copy Link Address” như minh họa trong Fig. 21.3.12.

<sup>452</sup> <https://conda.io/en/latest/miniconda.html>

## Miniconda

|            | Windows  | Mac OS X   | Linux  |
|------------|--|--|--|
| Python 3.7 | <a href="#">64-bit (exe installer)</a><br><a href="#">32-bit (exe installer)</a> | <a href="#">64-bit (bash installer)</a><br><a href="#">64-bit (.pkg installer)</a> | <a href="#">64-bit (bash installer)</a><br><a href="#">32-bit (bash installer)</a>           |
|            |  |  | <a href="#">64-bit (bash installer)</a><br><a href="#">32-bit (bash installer)</a>           |
| Python 2.7 | <a href="#">64-bit (exe installer)</a><br><a href="#">32-bit (exe installer)</a> | <a href="#">64-bit (bash installer)</a><br><a href="#">64-bit (.pkg installer)</a> | <a href="#">64-bit (bash installer)</a><br><a href="#">32-bit (bash installer)</a>           |
|            |  |  | <a href="#">Save Link As...</a><br><a href="#">Copy Link Address</a><br><a href="#">Copy</a> |

Fig. 21.3.12: Tải 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 xong, chạy lệnh sau để kích hoạt CUDA và conda.

```
~/miniconda3/bin/conda init
source ~/.bashrc
```

Tiếp theo, hãy tải về mã nguồn của 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 một môi trường conda d2l và nhập y để tiến hành cài đặt.

```
conda create --name d2l -y
```

Sau khi tạo môi trường d2l, kích hoạt nó và cài đặt pip.

```
conda activate d2l
conda install python=3.7 pip -y
```

Cuối cùng, cài đặt MXNet và gói thư viện d2l. Hậu tố cu101 nghĩa là đây là phiên bản sử dụng CUDA 10.1. Với các phiên bản khác, giả sử như CUDA 10.0, bạn sẽ thay bằng cu100.

```
pip install mxnet-cu101==1.7.0
pip install git+https://github.com/d2l-ai/d2l-en
```

Bạn có thể nhanh chóng kiểm tra mọi thứ đã hoạt động bằng cách:

```
$ python
>>> from mxnet import np, npx
>>> np.zeros((1024, 1024), ctx=npx.gpu())
```

#### 21.3.4 Chạy Jupyter

Để chạy Jupyter từ xa bạn cần sử dụng phương thức chuyển tiếp cổng SSH. Suy cho cùng thì máy chủ trên đám mây không có màn hình hay bàn phím. Để thực hiện việc này, hãy truy cập vào máy chủ của bạn từ máy tính bàn (hay laptop) 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. 21.3.13 cho ta thấy kết quả khả dĩ sau khi bạn chạy Jupyter Notebook. Dòng cuối cùng là URL của 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. 21.3.13: Kết quả sau khi chạy Jupyter Notebook. Dòng cuối cùng là URL của cổng 8888.

Do sử dụng phương thức chuyển tiếp cổng cho cổng 8889, bạn cần phải đổi địa chỉ cổng 8888 thành 8889 và sử dụng token được sinh bởi Jupyter để mở URL trên trình duyệt web trong máy của bạn.

#### 21.3.5 Đóng Máy ảo Không Dùng đến

Do dịch vụ đám mây tính phí theo thời gian sử dụng, bạn nên đóng những máy ảo hiện không sử dụng đến. Lưu ý rằng có các giải pháp thay thế khác: “Stop” một máy ảo nghĩa là bạn có thể khởi động nó lại được. Việc này khá giống với việc tắt nguồn máy chủ thông thường của bạn. Tuy nhiên, việc dừng một máy chủ vẫn sẽ tinh một lượng nhỏ vào hóa đơn cho kích thước ổ cứng được giữ lại. Lựa chọn “Terminate” một máy ảo sẽ xoá toàn bộ dữ liệu liên quan đến nó. Dữ liệu này bao gồm cả ổ cứng, vậy nên bạn sẽ không thể khởi động nó lại được. Chỉ thực hiện thao tác này nếu bạn chắc rằng bạn sẽ không cần đến nó trong tương lai.

Nếu bạn muốn sử dụng máy ảo làm khuôn mẫu (*template*) cho nhiều máy ảo khác, hãy nhấp chuột phải vào ví dụ trong Fig. 21.3.9 và chọn “Image” → “Create” để tạo một ảnh (*image*) của máy ảo này. Sau khi hoàn thành thao tác này, chọn “Instance State” → “Terminate” để xóa máy ảo này. Lần tiếp theo bạn muốn sử dụng máy ảo này, bạn có thể thực hiện theo các bước khởi tạo và chạy một máy ảo EC2 như minh họa trong phần này để tạo một máy ảo dựa trên ảnh đã lưu. Điểm khác biệt duy nhất nằm ở bước “1. Choose AMI” như trong Fig. 21.3.4, bạn phải chọn mục “My AMIs” phía bên trái để lựa chọn ảnh bạn đã lưu. Máy ảo được tạo ra sẽ giữ lại thông tin được lưu trên ảnh đĩa cứng. Ví dụ, bạn sẽ không cần phải cài đặt lại CUDA và các môi trường thời gian chạy khác.

### 21.3.6 Tóm tắt

- Bạn có thể khởi động và dừng các máy ảo tùy theo nhu cầu mà không cần phải mua và xây dựng máy tính riêng.
- Bạn cần phải cài đặt trình điều khiển GPU phù hợp trước khi sử dụng chúng.

### 21.3.7 Bài tập

1. Dịch vụ đám mây cung cấp sự thuận tiện, nhưng nó không hề rẻ. Tìm hiểu cách khởi động máy ảo spot<sup>453</sup> để tìm hiểu cách giảm chi phí.
2. Thử nghiệm với nhiều máy chủ GPU khác nhau. Chúng nhanh đến mức nào?
3. Thử nghiệm với máy chủ đa GPU. Việc mở rộng giúp hoàn thành công việc tốt đến thế nào?

### 21.3.8 Thảo luận

- Tiếng Anh: Main Forum<sup>454</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>455</sup>

### 21.3.9 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Nguyễn Văn Quang
- Phạm Hồng Vinh
- Nguyễn Mai Hoàng Long
- Đỗ Trường Giang
- Nguyễn Văn Cường

## 21.4 Sử dụng Google Colab

Chúng tôi đã giới thiệu cách chạy cuốn sách này trên AWS trong Section 21.2 và Section 21.3. Có một lựa chọn khác là chạy cuốn sách này (bản tiếng Anh) trên Google Colab<sup>456</sup>, nền tảng cung cấp GPU miễn phí khi bạn có một tài khoản Google.

Để chạy một phần nào đó trên Colab, bạn có thể đơn giản nhấp vào nút Colab phía bên phải tiêu đề của phần đó, ví dụ như trong Fig. 21.4.1.

<sup>453</sup> <https://aws.amazon.com/ec2/spot/>

<sup>454</sup> <https://discuss.d2l.ai/t/423>

<sup>455</sup> <https://forum.machinelearningcoban.com/c/d2l>

<sup>456</sup> <https://colab.research.google.com/>



Fig. 21.4.1: Mở một phần trong Colab

Khi bạn lần đầu thực thi một ô mã nguồn, bạn sẽ nhận được một cảnh báo như trong Fig. 21.4.2. Bạn có thể nhấn “RUN ANYWAY” để bỏ qua nó.

**Warning: This notebook was not authored ...**

This notebook is being loaded from [GitHub](#). It may request access to your data stored with Google, or read data and credentials from other sessions. Please review the source code before executing this notebook.

CANCEL RUN ANYWAY

Fig. 21.4.2: Thông tin cảnh báo cho việc chạy một phần trong Colab

Tiếp theo, Colab sẽ kết nối bạn tới một máy ảo để chạy notebook này. Cụ thể, nếu cần GPU, ví dụ như khi gọi đến hàm `d2l.try_gpu()`, ta sẽ yêu cầu Colab tự động kết nối tới một máy ảo GPU.

### 21.4.1 Tóm tắt

Bạn có thể sử dụng Google Colab để chạy từng phần của cuốn sách này với GPU.

### 21.4.2 Bài tập

Thử chỉnh sửa và chạy mã nguồn của cuốn sách này sử dụng Google Colab.

### 21.4.3 Thảo luận

- Tiếng Anh: [MXNet<sup>457</sup>](#)
- Tiếng Việt: [Diễn đàn Machine Learning Cơ Bản<sup>458</sup>](#)

<sup>457</sup> <https://discuss.d2l.ai/t/424>

<sup>458</sup> <https://forum.machinelearningcoban.com/c/d2l>

#### 21.4.4 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Quang
- Nguyễn Văn Cường

Lần cập nhật gần nhất: 03/11/2020. (Cập nhật lần cuối từ nội dung gốc: 30/06/2020)

### 21.5 Lựa chọn Máy chủ & GPU

Việc huấn luyện học sâu thông thường đòi hỏi một lượng lớn tài nguyên tính toán. Ở thời điểm hiện tại, GPU là công cụ tăng tốc phần cứng tiết kiệm chi phí nhất cho việc học sâu. Cụ thể, so với CPU, GPU rẻ hơn và thường cung cấp hiệu suất cao hơn hàng chục lần. Hơn nữa, một máy chủ có thể hỗ trợ đa GPU, tối 8 GPU với các máy chủ cao cấp. Số GPU điển hình là 4 cho một máy trạm kỹ thuật, vì vấn đề tỏa nhiệt, làm mát và lượng điện tiêu thụ sẽ tăng vọt, vượt quá khả năng một văn phòng có thể cung cấp. Để triển khai trên số lượng lớn hơn, điện toán đám mây, chẳng hạn như các máy ảo P3<sup>459</sup> và G4<sup>460</sup> của Amazon là một giải pháp thực tế hơn nhiều.

#### 21.5.1 Lựa chọn Máy chủ

Thông thường không cần mua các dòng CPU cao cấp với nhiều luồng vì phần lớn việc tính toán diễn ra trên GPU. Đồng nghĩa với việc, với Khóa trình thông dịch toàn cục (GIL) trong Python, hiệu suất đơn luồng của CPU có thể là vấn đề trong các tình huống mà chúng ta có 4-8 GPU. Điều này cho thấy rằng các CPU với số lượng nhân nhỏ hơn nhưng có xung nhịp cao hơn có thể là sự lựa chọn ít tốn kém. Chẳng hạn khi lựa chọn giữa CPU 6-nhân 4 GHz và 8-nhân 3.5 GHz, thì lựa chọn thứ nhất sẽ được ưu tiên hơn, mặc dù tốc độ kết hợp lại có thể thấp hơn. Một lưu ý quan trọng là các GPU sử dụng rất nhiều năng lượng và do đó tỏa nhiệt rất nhiều, đòi hỏi khả năng làm mát tốt và một khung máy đủ lớn để sử dụng các GPU đó. Bạn đọc nên theo sát các nguyên tắc bên dưới đây nếu có thể khi thiết kế máy trạm cho học sâu:

1. **Bộ Nguồn Cấp Điện.** GPU sử dụng một lượng điện năng đáng kể. Mỗi GPU có thể cần nguồn cấp lên đến 350W (kiểm tra *công suất đỉnh* của card đồ họa thay vì công suất trung bình, vì mã nguồn được tối ưu có thể ngắn nhiều năng lượng). Nếu nguồn điện của bạn không đáp ứng được nhu cầu, hệ thống sẽ trở nên không ổn định.
2. **Kích thước khung chứa.** GPU có kích thước lớn và các đầu nối nguồn phụ trợ thường cần thêm không gian. Thêm nữa, khung máy lớn giúp dễ làm mát hơn.
3. **Làm mát GPU.** Nếu bạn có số lượng lớn GPU, bạn có thể muốn đầu tư hệ thống tản nhiệt nước. Ngoài ra, có thể sử dụng các thiết kế *tham khảo* ngay cả khi chúng có số quạt làm mát ít hơn, vì chúng đủ mỏng để cho phép thông gió giữa các thiết bị. Nếu bạn mua một GPU có nhiều quạt, nó có thể quá dày để nhận đủ không khí khi lắp đặt nhiều GPU và bạn sẽ gặp phải tình trạng khó tản nhiệt.

<sup>459</sup> <https://aws.amazon.com/ec2/instance-types/p3/>

<sup>460</sup> <https://aws.amazon.com/blogs/aws/in-the-works-ec2-instances-g4-with-nvidia-t4-gpus/>

**4. Khe cắm PCIe.** Việc chuyển dữ liệu đến và đi từ GPU (và trao đổi giữa các GPU) đòi hỏi nhiều băng thông. Chúng tôi đề xuất khe cắm PCIe 3.0 với 16 làn. Nếu bạn lắp nhiều GPU, hãy đảm bảo là bạn đọc kỹ mô tả bo mạch chủ để chắc chắn băng thông 16x đó vẫn khả dụng khi nhiều GPU được sử dụng cùng lúc và tốc độ PCIe là 3.0 thay vì PCIe cho các khe cắm bổ sung. Một số bo mạch chủ sẽ hạ xuống băng thông 8x hoặc thậm chí 4x khi nhiều GPU được cài đặt. Điều này một phần là do số lượng làn PCIe mà CPU đó cung cấp.

Tóm lại, dưới đây là một số khuyến nghị để bạn xây dựng một máy chủ học sâu:

- **Người mới bắt đầu.** Một GPU cấp thấp với mức tiêu thụ điện năng thấp (GPU dành cho chơi game giá rẻ phù hợp cho việc sử dụng học sâu 150-200W). Nếu may mắn, máy tính hiện tại của bạn đã có sẵn một GPU như trên.
- **1 GPU.** Một CPU cấp thấp với 4 nhân sẽ là quá đủ và hầu hết các bo mạch chủ đều đáp ứng được. Hãy nhắm đến ít nhất 32 GB DRAM và đầu tư một ổ SSD để truy cập dữ liệu cục bộ. Nên sử dụng nguồn cung cấp 600W là đủ. Mua GPU có nhiều quạt.
- **2 GPU.** Một CPU cấp thấp với 4-6 nhân là đủ. Hãy nhắm đến 64 GB DRAM và đầu tư vào một ổ SSD. Bạn sẽ cần nguồn tầm 1000W cho hai GPU cao cấp. Đảm bảo rằng chúng có *hai* khe cắm PCIe 3.0 x16. Nếu có thể, hãy mua một bo mạch chủ có hai khoảng trống (khoảng cách 60mm) giữa các khe PCIe 3.0 x16 để có thêm không khí. Trong trường hợp này, hãy mua hai GPU có nhiều quạt.
- **4 GPU.** Đảm bảo rằng bạn mua một CPU có tốc độ luồng đơn tương đối nhanh (cụ thể là tần số xung nhịp cao). Bạn có thể sẽ cần một CPU có số lượng làn PCIe lớn hơn, chẳng hạn như một chip AMD Threadripper. Bạn có thể sẽ cần bo mạch chủ tương đối đắt tiền để có 4 khe cắm PCIe 3.0 x16 vì chúng có thể cần PLX để ghép kênh các làn PCIe. Hãy mua GPU có thiết kế tham khảo gốc vì nó hẹp hơn và cho phép không khí lưu thông giữa các GPU. Bạn cần nguồn điện tầm 1600-2000W và ổ cắm trong văn phòng của bạn có thể không hỗ trợ điều đó. Máy chủ này có thể sẽ *gây tiếng ồn và tỏa nhiệt* nhiều. Bạn hẳn là không muốn đặt nó dưới bàn làm việc của bạn. Khuyến nghị sử dụng 128 GB DRAM. Mua một ổ SSD (1-2 TB NVMe) để lưu trữ cục bộ và một số ổ cứng theo cấu hình RAID để lưu trữ dữ liệu của bạn.
- **8 GPU.** Bạn cần mua khung máy chủ đa GPU chuyên dụng với nhiều nguồn điện dự phòng (chẳng hạn, 2 + 1 cho 1600W với mỗi bộ nguồn). Điều này sẽ yêu cầu CPU máy chủ có khe cắm kép, 256 GB ECC DRAM, một cạc mạng nhanh (khuyến nghị 10 GBE), và bạn sẽ cần kiểm tra liệu máy chủ có hỗ trợ *hình dạng kích thước vật lý* của GPU hay không. Luồng không khí và bố trí dây có sự khác biệt đáng kể giữa GPU tiêu dùng và GPU máy chủ (cụ thể ở đây là RTX 2080 so với Tesla V100). Điều này có nghĩa là bạn có thể không lắp đặt được GPU tiêu dùng vào máy chủ do không đủ khoảng trống cho cáp nguồn hoặc thiếu dây nối phù hợp (như một trong các đồng tác giả đã đau khổ khi phát hiện ra).

### 21.5.2 Lựa chọn GPU

Hiện nay, AMD và NVIDIA là hai nhà sản xuất GPU chính. NVIDIA là tiên phong trong tham gia lĩnh vực học sâu và cung cấp hỗ trợ tốt hơn cho các framework học sâu thông qua CUDA. Do đó, phần lớn người mua chọn GPU của NVIDIA.

NVIDIA cung cấp hai loại GPU, nhắm tới người dùng cá nhân (ví dụ như dòng GTX và RTX) và người dùng doanh nghiệp (qua dòng Tesla của họ). Hai loại GPU cung cấp khả năng tính toán tương đương nhau. Tuy nhiên, GPU dành cho người dùng doanh nghiệp thường sử dụng tản nhiệt cường ép (thụ động), nhiều bộ nhớ, và bộ nhớ ECC (sửa sai - error correcting). Những GPU này phù hợp hơn cho trung tâm dữ liệu và thường có giá cao hơn 10 lần so với GPU tiêu dùng.

Nếu bạn là một công ty lớn với 100+ máy chủ, bạn nên cân nhắc dòng NVIDIA Tesla hoặc thay thế bằng cách sử dụng máy chủ GPU trên đám mây. Với các phòng nghiên cứu hay một công ty tầm trung với 10+ máy chủ, dòng NVIDIA RTX có lẽ sẽ có hiệu quả chi phí tốt nhất. Bạn có thể mua máy chủ cấu hình sẵn với khung chứa Supermicro hay Asus, có thể chứa hiệu quả 4-8 GPU.

Nhà cung cấp GPU thường ra mắt thế hệ mới mỗi 1-2 năm, ví dụ như dòng GTX 1000 (Pascal) ra mắt vào 2017 và dòng RTX 2000 (Turing) ra mắt vào 2019. Mỗi dòng gồm có nhiều mẫu khác nhau cung cấp mức hiệu năng khác nhau. Hiệu năng GPU chủ yếu là sự kết hợp của ba thông số sau:

1. **Khả năng tính toán.** Thông thường ta quan tâm đến khả năng tính toán dấu phẩy động 32-bit (*32-bit floating-point*). Huấn luyện mô hình sử dụng dấu phẩy động 16-bit (*FP16 - 16-bit floating point*) cũng đang dần phổ biến. Nếu chỉ quan tâm đến tác vụ dự đoán, bạn cũng có thể sử dụng số nguyên 8-bit (*8-bit integer*). Thế hệ mới nhất của GPU Turing còn cung cấp chế độ tăng tốc 4-bit (*4-bit acceleration*). Không may là hiện nay, các thuật toán huấn luyện với số thực độ chính xác thấp vẫn chưa được phổ biến.
2. **Kích thước bộ nhớ.** Khi các mô hình của bạn trở nên lớn hơn hay khi tăng kích thước batch khi huấn luyện, bạn sẽ cần nhiều bộ nhớ GPU hơn. Hãy kiểm tra HBM2 (Bộ nhớ Băng thông cao - *High Bandwidth Memory*) và GDDR6 (DDR Đồ họa - *Graphics DDR*). HBM2 nhanh hơn nhưng đắt hơn nhiều.
3. **Băng thông bộ nhớ.** Bạn chỉ có thể tận dụng tối đa khả năng tính toán nếu bạn có đủ băng thông bộ nhớ. Hãy chọn bus bộ nhớ rộng nếu sử dụng GDDR6.

Với phần lớn người dùng, tập trung vào khả năng tính toán là đủ. Chú ý rằng các GPU khác nhau cung cấp các cách tăng tốc khác nhau, ví dụ như TensorCores của NVIDIA tăng tốc một tập con các toán tử lên tới gấp 5 lần. Vậy nên hãy đảm bảo rằng thư viện của bạn hỗ trợ việc này. Bộ nhớ GPU thì không nên ít hơn 4 GB (8 GB thì hơn). Hãy cố gắng tránh sử dụng GPU để hiện thị giao diện đồ họa người dùng (GUI), thay vào đó nếu cần hãy sử dụng card đồ họa tích hợp sẵn trong máy. Nếu bắt buộc phải dùng GPU để hiển thị GUI, hãy thêm vào 2 GB RAM cho an toàn.

Fig. 21.5.1 so sánh khả năng tính toán dấu phẩy động 32-bit và giá của các mẫu khác nhau của các dòng GTX 900, GTX 1000 và RTX 2000. Đây là bảng giá đề xuất có thể được tìm thấy trên Wikipedia.

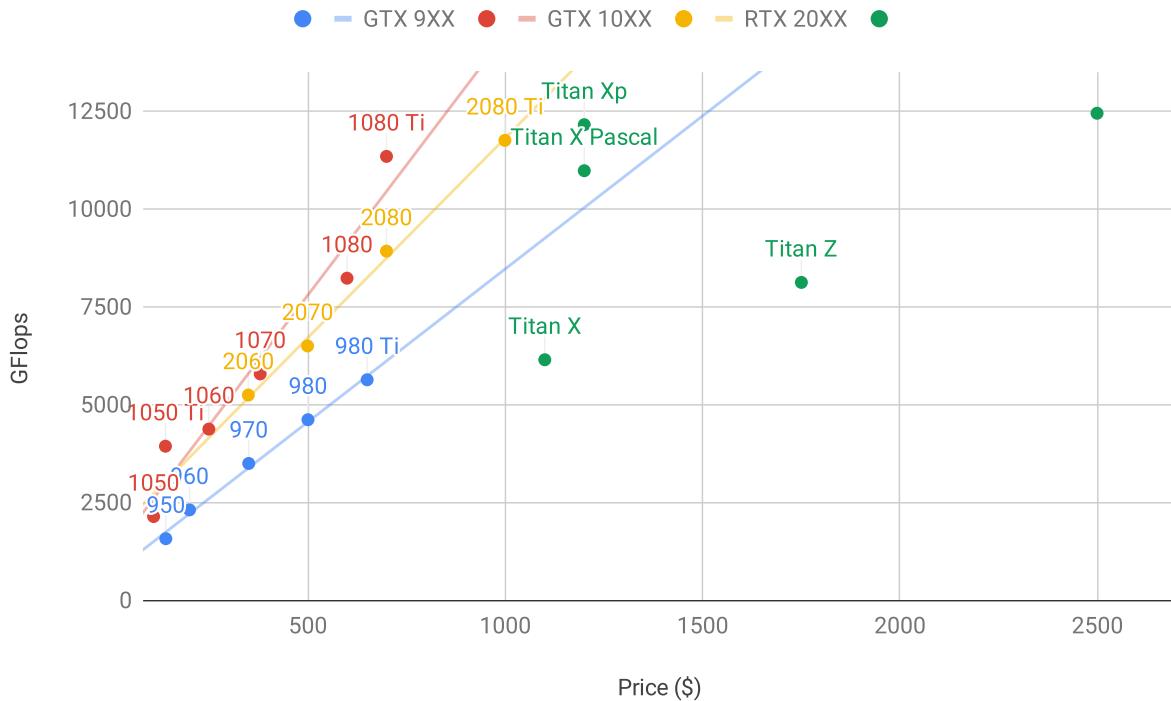


Fig. 21.5.1: So sánh khả năng tính toán dấu phẩy động và giá.

Bạn có thể thấy một số điểm sau:

- Trong cùng một dòng, giá và hiệu năng gần như tỷ lệ với nhau. Mẫu Titan yêu cầu một khoản tiền đáng kể để đổi lấy lợi ích của lượng lớn bộ nhớ GPU. Tuy nhiên, những mẫu mới hơn cung cấp hiệu quả chi phí tốt hơn, như có thể thấy qua so sánh giữa 980 Ti và 1080 Ti. Giá đắt như không cải thiện nhiều đối với dòng RTX 2000. Tuy nhiên, việc này là do chúng cung cấp hiệu năng hoàn toàn vượt trội đối với các giá trị có độ chính xác thấp (FP16, INT8 và INT4).
- tỷ lệ hiệu năng trên giá của dòng GTX 1000 lớn hơn khoảng 2 lần so với dòng 900.
- Với dòng RTX 2000, giá là một hàm *affine* của hiệu năng.



Fig. 21.5.2: Khả năng tính toán dấu phẩy động và năng lượng tiêu hao.

Fig. 21.5.2 chỉ ra lượng năng lượng tiêu hao chủ yếu tỷ lệ tuyến tính với khối lượng tính toán. Thứ hai, các thế hệ sau có hiệu quả tốt hơn. Đồ thị của dòng RTX 2000 có vẻ như mâu thuẫn với điều này. Tuy nhiên, đây là hệ quả của TensorCore yêu cầu năng lượng rất lớn.

### 21.5.3 Tóm tắt

- Chú ý nguồn, luồng bus PCIe, tốc độ CPU đơn luồng và tản nhiệt khi xây dựng máy chủ.
- Bạn nên mua thế hệ GPU mới nhất nếu có thể.
- Sử dụng đám mây để triển khai các dự án lớn.
- Máy chủ chạy nhiều ứng dụng có thể sẽ không tương thích với tất cả các GPU. Kiểm tra các thông số cơ học và tản nhiệt trước khi mua.
- Sử dụng FP16 hoặc độ chính xác thấp hơn để có được hiệu năng tốt hơn.

### 21.5.4 Thảo luận

- Tiếng Anh: [Main Forum<sup>461</sup>](https://discuss.d2l.ai/t/425)
- Tiếng Việt: [Diễn đàn Machine Learning Cơ Bản<sup>462</sup>](https://forum.machinelearningcoban.com/c/d2l)

<sup>461</sup> <https://discuss.d2l.ai/t/425>

<sup>462</sup> <https://forum.machinelearningcoban.com/c/d2l>

### 21.5.5 Những người thực hiện

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Lê Khắc Hồng Phúc
- Phạm Hồng Vinh
- Phạm Minh Đức
- Nguyễn Văn Quang
- Nguyễn Mai Hoàng Long

## 21.6 Đóng góp cho Quyển sách

Những đóng góp từ [cộng đồng](#)<sup>463</sup> giúp chúng tôi cải thiện cuốn sách này. Nếu bạn tìm thấy có lỗi đánh máy, đường dẫn hết hạn, hay phần nào đó mà bạn nghĩ chúng tôi thiếu trích dẫn, phần mã nguồn không được thanh thoát hay có chỗ giải thích không rõ ràng, xin vui lòng đóng góp và giúp chúng tôi hỗ trợ những độc giả khác. Trong khi với các quyển sách bình thường, khoảng cách giữa những lần xuất bản (đó là giữa các lần sửa lỗi đánh máy) có thể tính bằng năm, thì giờ đây chúng tôi chỉ cần vài giờ đến vài ngày để đưa một cải thiện vào bản dịch này. Để có thể đóng góp trực tiếp, bạn cần đăng một [pull request](#)<sup>464</sup> lên GitHub của bản dịch này. Khi pull request của bạn được nhóm dịch gộp (*merge*) vào repo, bạn sẽ trở thành một người đóng góp (*contributor*). Một cách khác để bạn có thể đóng góp đơn giản hơn là tạo một [issues](#)<sup>465</sup> mới và báo cáo vấn đề bạn tìm thấy, tạo một [thảo luận](#)<sup>466</sup> mới để cùng trao đổi nhiều vấn đề hơn với nhóm dịch thuật.

### 21.6.1 Thay đổi nhỏ trong Văn bản

Đóng góp phổ biến nhất là chỉnh sửa một câu hoặc chữa lỗi đánh máy. Chúng tôi khuyến nghị bạn nên tìm tệp gốc trên [github](#)<sup>467</sup> và trực tiếp chỉnh sửa tệp. Ví dụ, bạn có thể tìm tệp thông qua nút [Find file](#)<sup>468</sup> (Fig. 21.6.1) để định vị tệp gốc, tệp ở đây là một tệp markdown. Sau đó bạn nhấn nút “Edit this file” ở góc trên bên phải để thay đổi tệp markdown này.

<sup>463</sup> <https://github.com/aivivn/d2l-vn/graphs/contributors>

<sup>464</sup> <https://github.com/aivivn/d2l-vn/pulls>

<sup>465</sup> <https://github.com/aivivn/d2l-vn/issues>

<sup>466</sup> <https://github.com/aivivn/d2l-vn/discussions>

<sup>467</sup> <https://github.com/aivivn/d2l-vn>

<sup>468</sup> <https://github.com/aivivn/d2l-vn/find/master>

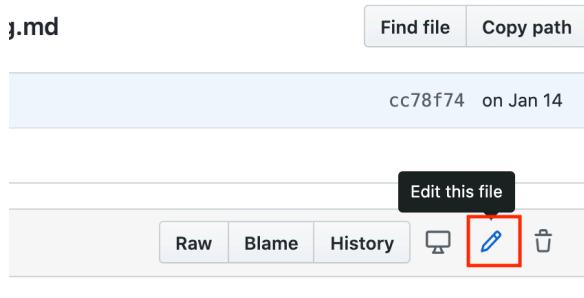


Fig. 21.6.1: Chỉnh sửa tệp trên GitHub.

Sau khi hoàn thành, điền mô tả thay đổi của bạn vào ô “Propose file change” ở cuối trang và sau đó nhấn nút “Propose file change”. Trang sẽ được chuyển hướng đến một trang mới để bạn có thể kiểm tra lại thay đổi của bạn (Fig. 21.6.7). Nếu mọi thứ đều ổn, bạn có thể đăng một pull request bằng cách nhấn vào nút “Create pull request”.

## 21.6.2 Đề xuất một Thay đổi Lớn

Nếu bạn có dự định cập nhật một phần lớn văn bản hay mã nguồn thì bạn cần phải biết thêm một chút về định dạng mà cuốn sách này sử dụng. Tệp nguồn được dựa trên [định dạng markdown](#)<sup>469</sup> với một tập các phần mở rộng thông qua gói [d2lbook](#)<sup>470</sup> ví dụ như tham chiếu đến phương trình, hình ảnh, chương mục và trích dẫn. Bạn có thể sử dụng bất kỳ trình biên tập Markdown nào để mở các tệp này và thực hiện chỉnh sửa.

Nếu bạn muốn thay đổi mã nguồn, chúng tôi khuyến nghị các bạn sử dụng Jupyter để mở các tệp Markdown này như mô tả trong [Section 21.1](#). Nhờ đó bạn có thể chạy và kiểm tra thay đổi của bạn. Xin vui lòng xóa toàn bộ các ô kết quả trước khi đăng thay đổi của bạn, hệ thống CI của chúng tôi sẽ thực thi phần bạn cập nhật để sinh kết quả.

Một số phần có thể hỗ trợ lập trình đa framework, bạn có thể sử dụng d2lbook để kích hoạt một framework cụ thể, khi đó phần lập trình các framework khác trở thành khối mã Markdown và sẽ không được thực thi khi bạn chạy “Run All” trong Jupyter. Nói cách khác, đầu tiên cài đặt d2lbook bằng cách chạy

```
pip install git+https://github.com/d2l-ai/d2l-book
```

Sau đó trong thư mục gốc d2l-en, bạn có thể kích hoạt một đoạn mã 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 đăng thay đổi của bạn lên, xin vui lòng xóa toàn bộ các ô kết quả và kích hoạt tất cả bằng

```
d2lbook activate all chapter_multilayer-perceptrons/mlp-scratch.md
```

<sup>469</sup> <https://daringfireball.net/projects/markdown/syntax>

<sup>470</sup> <http://book.d2l.ai/user/markdown.html>

Nếu bạn thêm một khối mã nguồn mới không có trong cách lập trình mặc định, tức MXNet, xin vui lòng sử dụng `#@tab` để đánh dấu khối này tại dòng đầu tiên. Ví dụ, `#@tab pytorch` cho khối mã PyTorch, `#@tab tensorflow` cho khối mã TensorFlow, hoặc `#@tab all` cho khối mã được dùng chung cho tất cả các cách lập trình. Bạn có thể tham khảo [d2lbook<sup>471</sup>](#) để biết thêm thông tin.

### 21.6.3 Thêm một Phần Mới hoặc một Cách lập trình cho Framework Mới

Nếu bạn muốn tạo một chương mới, ví dụ như học tăng cường, hoặc thêm một cách lập trình cho một framework mới, ví dụ như TensorFlow, trước tiên xin vui lòng liên hệ với nhóm dịch, có thể bằng email hoặc sử dụng [github issues<sup>472</sup>](#).

### 21.6.4 Đăng một Thay đổi Lớn

Chúng tôi đề nghị bạn phải sử dụng quy trình git chuẩn để đăng một thay đổi lớn. Tóm lại, quy trình này hoạt động như mô tả trong Fig. 21.6.2.

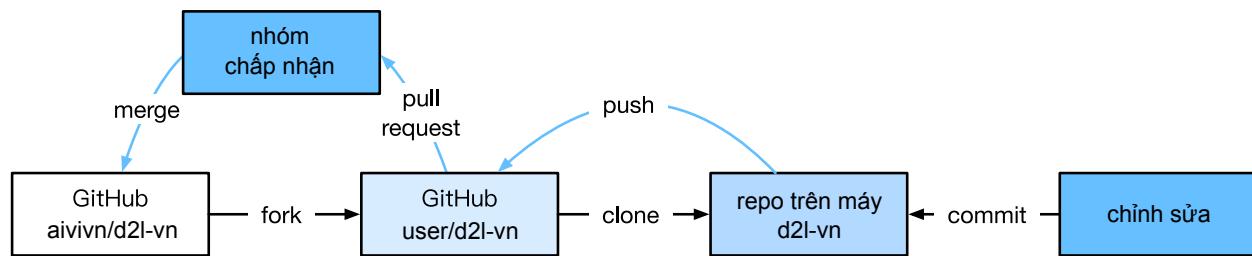


Fig. 21.6.2: Đóng góp cho quyền sách.

Chúng tôi sẽ hướng dẫn chi tiết từng bước. Nếu bạn quen thuộc với Git bạn có thể bỏ qua phần này. Để ngắn gọn chúng tôi giả sử người đóng góp có username là “astonzhang”.

#### Cài đặt Git

Cuốn sách mã nguồn mở của Git mô tả chi tiết cách cài đặt Git<sup>473</sup>. Việc này có thể được thực hiện thông qua `apt install git` trên Ubuntu Linux, bằng cách cài đặt công cụ phát triển Xcode trên macOS, hoặc bằng cách sử dụng [desktop client<sup>474</sup>](#) của GitHub. Nếu bạn không có tài khoản GitHub, bạn cần phải đăng ký một tài khoản.

<sup>471</sup> [http://book.d2l.ai/user/code\\_tabs.html](http://book.d2l.ai/user/code_tabs.html)

<sup>472</sup> <https://github.com/aivivn/d2l-vn/issues>

<sup>473</sup> <https://git-scm.com/book/en/v2>

<sup>474</sup> <https://desktop.github.com>

## Đăng nhập vào GitHub

Điền [địa chỉ<sup>475</sup>](#) của repo chứa mã nguồn bản dịch cuốn sách này vào trình duyệt web của bạn. Chọn nút Fork trong khung đỏ ở phía trên bên phải của Fig. 21.6.3, để tạo một bản sao cho repo của cuốn sách này. Nó giờ đây là *bản sao của bạn* và bạn có thể tùy ý thay đổi nó.



Fig. 21.6.3: Trang repo chứa mã nguồn.

Giờ thì repo chứa mã nguồn của cuốn sách đã được fork (tức sao chép) tới tên người dùng của bạn, ví dụ như `astonzhang/d2l-en` được chỉ ra phía trên bên trái của ảnh chụp màn hình Fig. 21.6.4.

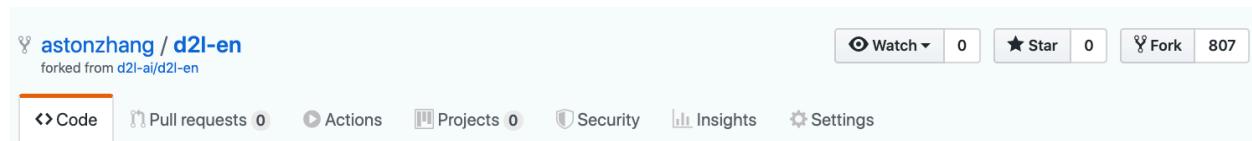


Fig. 21.6.4: Fork repo chứa mã nguồn.

## Clone Repo

Để clone một repo (tức là tạo bản sao cục bộ) ta cần phải có địa chỉ của repo đó. Nút màu xanh trong Fig. 21.6.5 hiển thị địa chỉ này. Hãy đảm bảo rằng bản sao cục bộ của bạn cập nhật gần nhất với repo chính nếu bạn quyết định giữ bản fork này lâu dài. Còn bây giờ chỉ cần đơn giản làm theo các hướng dẫn trong [Cài đặt](#) (page 11) để bắt đầu. Điểm khác biệt chính ở đây là bạn đang tải về *bản fork của riêng bạn* cho repo này.



Fig. 21.6.5: Git clone.

```
# Replace your_github_username with your GitHub username
git clone https://github.com/your_github_username/d2l-vn.git
```

<sup>475</sup> <https://github.com/aivivn/d2l-vn/>

## Chỉnh sửa Bản dịch và Đẩy lên

Giờ là lúc để chỉnh sửa bản dịch. Tốt nhất là chỉnh sửa các notebook trên Jupyter theo hướng dẫn trong [Section 21.1](#). Tạo thay đổi và kiểm tra xem chúng ổn chưa. Giả sử bạn đã điều chỉnh một lỗi đánh máy trong tệp `~/d2l-vn/chapter_appendix_tools/how-to-contribute.md`. Sau đó bạn có thể kiểm tra xem bạn đã sửa đổi những tệp nào.

Ở thời điểm này Git sẽ thông báo là tệp `chapter_appendix_tools/how-to-contribute.md` đã được sửa đổi.

```
mylaptop:d2l-vn 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 đây là những sửa đổi bạn muốn, thực thi lệnh sau:

```
git add chapter_appendix_tools/how-to-contribute.md
git commit -m 'fix typo in git documentation'
git push
```

Đoạn mã được chỉnh sửa lúc này sẽ nằm trong bản fork cá nhân của bạn cho repo này. Để yêu cầu thay đổi thêm, bạn cần phải tạo một pull request đối với repo chính thức của bản dịch.

## Pull Request

Như chỉ ra trong [Fig. 21.6.6](#), đi tới bản fork của bạn trên GitHub và chọn “New pull request”. Thao tác này sẽ mở ra một cửa sổ hiển thị những điểm khác nhau giữa bản chỉnh sửa của bạn và bản hiện trong repo chính của cuốn sách.

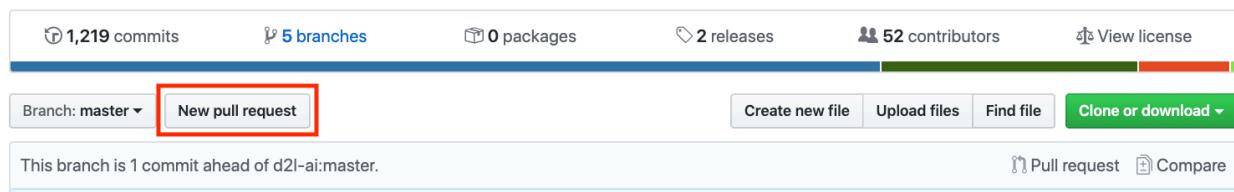


Fig. 21.6.6: Pull Request.

## Đăng Pull Request lên

Cuối cùng, đăng một pull request lên bằng cách nhấn vào nút như chỉ ra trong Fig. 21.6.7. Hãy đảm bảo mô tả các thay đổi bạn đã thực hiện trong pull request này. Việc này sẽ giúp nhóm dịch dễ dàng hơn trong việc kiểm tra và gộp vào bản dịch. Tuỳ thuộc vào các thay đổi, pull request này có thể được chấp thuận ngay lập tức, bác bỏ, hoặc khả năng cao hơn là bạn sẽ nhận được phản hồi trên các thay đổi này. Một khi bạn đã hợp nhất được chúng, chúc mừng bạn đã hoàn thành và hãy tiếp tục phát huy.

## Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).

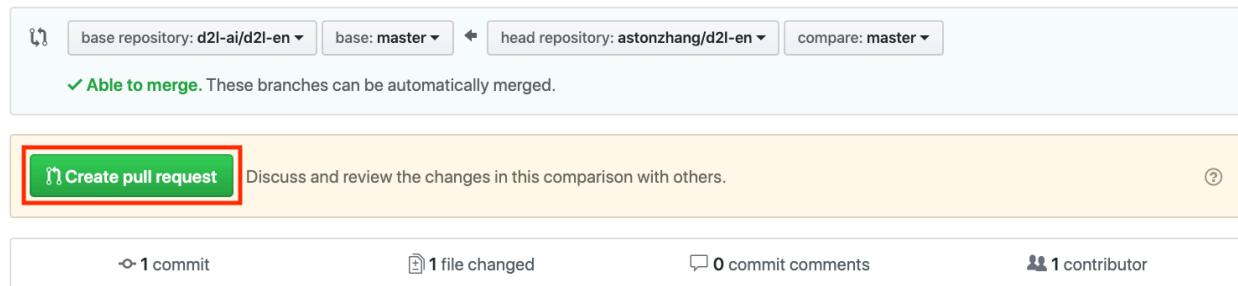


Fig. 21.6.7: Tạo Pull Request.

Pull request của bạn sẽ xuất hiện ở danh sách Pull requests trong repo chính. Chúng tôi sẽ làm mọi thứ có thể để xử lý nó nhanh chóng.

### 21.6.5 Tóm tắt

- Bạn có thể sử dụng GitHub để đóng góp cho bản dịch này.
- Bạn có thể chỉnh sửa tệp trực tiếp trên GitHub với những thay đổi nhỏ.
- Với một thay đổi lớn, xin vui lòng tạo fork cho repo này, tạo chỉnh sửa cục bộ và chỉ đóng góp một khi bạn đã sẵn sàng.
- Pull request là cách mà các đóng góp được gói lại. Cố gắng dùng đăng một pull request quá lớn do điều này khiến chúng khó hiểu và khó để hợp nhất. Tốt hơn là gửi nhiều pull request nhỏ.

### 21.6.6 Bài tập

1. Star và tạo fork của repo d2l-vn.
2. Tìm đoạn mã nào đó cần cải thiện và đăng một pull request.
3. Tìm một trích dẫn mà chúng tôi bỏ sót và đăng một pull request.
4. Thường thì trong thực hành, tốt hơn hết là khi tạo một pull request thì sử dụng một nhánh (*branch*) mới. Hãy học cách thực hiện việc này với [Git branching](#)<sup>476</sup>.

<sup>476</sup> <https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell>

### **21.6.7 Thảo luận**

- Tiếng Anh: Main Forum<sup>477</sup>
- Tiếng Việt: Diễn đàn Machine Learning Cơ Bản<sup>478</sup>

### **21.6.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Đỗ Trường Giang
- Nguyễn Văn Cường
- Nguyễn Mai Hoàng Long

## **21.7 Tài liệu API của d2l**

Mã nguồn của những thành phần dưới đây trong gói d2l và các mục mà trong đó chúng được định nghĩa và giải thích có thể được tìm thấy tại [tập tin mã nguồn](#)<sup>479</sup>.

### **21.7.1 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh

## **21.8 Những người thực hiện**

Bản dịch trong trang này được thực hiện bởi:

- Đoàn Võ Duy Thanh
- Phạm Hồng Vinh
- Nguyễn Văn Cường

---

<sup>477</sup> <https://discuss.d2l.ai/t/426>

<sup>478</sup> <https://forum.machinelearningcoban.com/c/d2l>

<sup>479</sup> <https://github.com/d2l-ai/d2l-en/tree/master/d2l>



# 22 | Bảng thuật ngữ

Các thuật ngữ cần được dịch theo chuẩn trong tập tin này.

Nếu một từ chưa có trong bảng thuật ngữ hay ở góc độ của bạn cho rằng một từ nào đó không nên được dịch ra tiếng Việt, bạn có thể mở một chủ đề thảo luận tại thẻ [Discussions<sup>481</sup>](#) của dự án.

*Chú ý giữ thứ tự theo bảng chữ cái để tiện tra cứu.*

[A](#) (page 909) [B](#) (page 910) [C](#) (page 910) [D](#) (page 912) [E](#) (page 913) [F](#) (page 914) [G](#) (page 915) [H](#) (page 915) [I](#) (page 916) [J](#) (page 916) [K](#) (page 917) [L](#) (page 917) [M](#) (page 918) [N](#) (page 919) [O](#) (page 920) [P](#) (page 920) [Q](#) (page 921) [R](#) (page 922) [S](#) (page 923) [T](#) (page 924) [U](#) (page 925) [V](#) (page 925) [W](#) (page 925) [X](#) (page ??) [Y](#) (page ??) [Z](#) (page ??)

## 22.1 A

| English                               | Tiếng Việt                | Thảo luận tại   |
|---------------------------------------|---------------------------|---|
| accuracy                              | độ chính xác              |   |
| activation function                   | hàm kích hoạt             |   |
| adversarial learning                  | học đối kháng             | <a href="https://git.io/JvQxt">https://git.io/JvQxt</a> |
| agent                                 | tác nhân                  |   |
| algorithm's performance               | chất lượng thuật toán     |   |
| alternative hypothesis                | giả thuyết đối            | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| analytical solution                   | nghiệm theo công thức     |   |
| analogy                               | loại suy                  |   |
| anchor box                            | khung neo                 |   |
| approximate inference                 | suy luận gần đúng         |   |
| approximate training                  | huấn luyện gần đúng       |   |
| argument (in programming)             | đối số                    |   |
| artificial data synthesis             | tổng hợp dữ liệu nhân tạo |   |
| artificial general intelligence (AGI) | trí tuệ nhân tạo phổ quát | <a href="https://git.io/Jvoj9">https://git.io/Jvoj9</a> |
| attention mechanisms                  | cơ chế tập trung          |   |
| automatic differentiation             | tính vi phân tự động      | <a href="https://git.io/JvojU">https://git.io/JvojU</a> |
| autoregressive                        | tự hồi quy                |   |
| average pooling                       | gộp trung bình            | <a href="https://git.io/JfGi6">https://git.io/JfGi6</a> |
| avoidable bias                        | độ chêch tránh được       |   |

<sup>481</sup> <https://github.com/aivivn/d2l-vn/discussions>

## 22.2 B

| English                                | Tiếng Việt                     | Thảo luận tại   |
|--|--------------------------------|---|
| backend                                | back-end                       |   |
| background noise                       | nhiều nền                      | <a href="https://git.io/JvQxm">https://git.io/JvQxm</a> |
| backpropagation                        | làn truyền ngược               |   |
| backpropagation through time           | làn truyền ngược qua thời gian |   |
| backward pass                          | lượt truyền ngược              | <a href="https://git.io/JvohG">https://git.io/JvohG</a> |
| bag of words                           | túi từ                         |   |
| bandit                                 | máy đánh bạc                   | <a href="https://git.io/Jfe1v">https://git.io/Jfe1v</a> |
| batch                                  | batch                          | <a href="https://git.io/JvojE">https://git.io/JvojE</a> |
| batch normalization                    | chuẩn hóa theo batch           | <a href="https://git.io/Jfe1T">https://git.io/Jfe1T</a> |
| batch size                             | kích thước batch               | <a href="https://git.io/JvXdK">https://git.io/JvXdK</a> |
| beam search                            | tìm kiếm chùm                  | <a href="https://git.io/Jf9Nl">https://git.io/Jf9Nl</a> |
| benchmark                              | đánh giá xếp hạng              | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> |
| bias (model parameter)                 | hệ số điều chỉnh               | <a href="https://git.io/Jvopx">https://git.io/Jvopx</a> |
| bias (of an estimator)                 | độ chêch                       | <a href="https://git.io/JvQxO">https://git.io/JvQxO</a> |
| bias-variance tradeoff                 | đánh đổi độ chêch - phương sai | <a href="https://git.io/JvQA6">https://git.io/JvQA6</a> |
| bidirectional recurrent neural network | mạng nơ-ron hồi tiếp hai chiều | <a href="https://git.io/JJeal">https://git.io/JJeal</a> |
| big data                               | big data                       |   |
| binomial distribution                  | phân phối nhị thức             | <a href="https://git.io/JvohQ">https://git.io/JvohQ</a> |
| blackbox dev set                       | tập phát triển blackbox        | <a href="https://git.io/JvQx3">https://git.io/JvQx3</a> |
| bounding box                           | khung chứa                     | <a href="https://git.io/JvQxs">https://git.io/JvQxs</a> |
| broadcast                              | làn truyền                     | <a href="https://git.io/Jvoj3">https://git.io/Jvoj3</a> |
| bus                                    | bus                            |   |

## 22.3 C

| English                      | Tiếng Việt               | Thảo luận tại   |
|------------------------------|--------------------------|---|
| cache                        | bộ nhớ đệm               |   |
| (strictly) convex function   | hàm lồi (chặt)           | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| candidate hidden state       | trạng thái ẩn tiềm năng  |   |
| candidate memory             | ô nhớ tiềm năng          |   |
| categorical variable         | biến hạng mục            | <a href="https://git.io/JfeXx">https://git.io/JfeXx</a> |
| category                     | hạng mục                 | <a href="https://git.io/JJDKV">https://git.io/JJDKV</a> |
| causality                    | quan hệ nhân quả         |   |
| central target word          | từ đích trung tâm        | <a href="https://git.io/JUGZQ">https://git.io/JUGZQ</a> |
| central word                 | từ trung tâm             | <a href="https://git.io/JUGZQ">https://git.io/JUGZQ</a> |
| chain rule                   | quy tắc dây chuyền       | <a href="https://git.io/Jvojk">https://git.io/Jvojk</a> |
| channel (in computer vision) | kênh                     |   |
| (model) checkpoint           | checkpoint (của mô hình) |   |
| classifier                   | bộ phân loại             |   |
| closed-form solution         | biểu thức dạng đóng      | <a href="https://git.io/Jvopd">https://git.io/Jvopd</a> |
| cloud computing              | điện toán đám mây        | <a href="https://git.io/JJn3b">https://git.io/JJn3b</a> |

continues on next page

Table 22.3.1 – continued from previous page

| English                          | Tiếng Việt                | Thảo luận tại   |
|----------------------------------|---------------------------|---|
| clustering                       | phân cụm                  | <a href="https://git.io/JvojD">https://git.io/JvojD</a> |
| code (noun)                      | mã nguồn                  |   |
| code (verb)                      | viết mã                   |   |
| coefficient                      | hệ số                     |   |
| collaborative filtering          | lọc cộng tác              | <a href="https://git.io/JfjST">https://git.io/JfjST</a> |
| command line (interface)         | cửa sổ dòng lệnh          |   |
| computational graph              | đồ thị tính toán          | <a href="https://git.io/JvohQ">https://git.io/JvohQ</a> |
| computer vision                  | thị giác máy tính         |   |
| computing (in computer science)  | điện toán                 | <a href="https://git.io/JvojH">https://git.io/JvojH</a> |
| concatenate                      | nối                       |   |
| conditional distribution         | phân phối có điều kiện    | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a> |
| confidence interval              | khoảng tin cậy            |   |
| confidence level                 | mức tin cậy               |   |
| confusion matrix                 | ma trận nhầm lẫn          | <a href="https://git.io/JvQAY">https://git.io/JvQAY</a> |
| constrain                        | ràng buộc                 |   |
| content loss (in style transfer) | mất mát nội dung          | <a href="https://git.io/JJyeI">https://git.io/JJyeI</a> |
| context-independent              | độc lập ngữ cảnh          |   |
| context-sensitive                | nhạy ngữ cảnh             |   |
| convex combination               | tổ hợp lồi                |   |
| convex optimization              | tối ưu lồi                |   |
| convex set                       | tập lồi                   | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| convolutional neural network     | mạng nơ-ron tích chập     |   |
| corpus                           | kho ngữ liệu              |   |
| coreference resolution           | phân giải đồng tham chiếu |   |
| correlation coefficient          | hệ số tương quan          |   |
| cosine                           | cô-sin                    |   |
| cost function                    | hàm chi phí               | <a href="https://git.io/Jvojp">https://git.io/Jvojp</a> |
| covariate                        | hiệp biến                 | <a href="https://git.io/JvohK">https://git.io/JvohK</a> |
| crop (image)                     | cắt (ảnh)                 |   |
| cross correlation                | tương quan chéo           |   |
| cross entropy                    | entropy chéo              |   |
| cross validation                 | kiểm định chéo            |   |

## 22.4 D

| English                   | Tiếng Việt                      | Thảo luận tại   |
|---------------------------|---------------------------------|---|
| data                      | dữ liệu                         |   |
| data augmentation         | tăng cường dữ liệu              |   |
| data manipulation         | thao tác với dữ liệu            | <a href="https://git.io/Jvohh">https://git.io/Jvohh</a> |
| data mismatch             | dữ liệu không tương đồng        |   |
| data science              | khoa học dữ liệu                | <a href="https://git.io/JvojD">https://git.io/JvojD</a> |
| data scientist            | nhà khoa học dữ liệu            | <a href="https://git.io/JvojD">https://git.io/JvojD</a> |
| datapoint (data point)    | điểm dữ liệu                    |   |
| dataset (data set)        | tập dữ liệu                     |   |
| deep learning             | học sâu                         |   |
| deferred initialization   | khởi tạo trễ                    | <a href="https://git.io/Jfe1i">https://git.io/Jfe1i</a> |
| dense layer               | tầng kết nối dày đặc            |   |
| densely connected network | mạng tích chập kết nối dày đặc  | <a href="https://git.io/JfGi1">https://git.io/JfGi1</a> |
| dev set                   | tập phát triển                  |   |
| dev set performance       | chất lượng trên tập phát triển  |   |
| development set           | tập phát triển                  |   |
| differentiable            | khả vi                          | <a href="https://git.io/JvKee">https://git.io/JvKee</a> |
| dimension                 | chiều                           |   |
| dimensionality            | kích thước chiều                |   |
| distributed               | phân tán                        |   |
| distribution              | phân phối                       |   |
| domain adaptation         | thích ứng miền                  |   |
| downsample                | giảm mẫu                        | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |
| downstream task           | tác vụ xuôi dòng                | <a href="https://git.io/JUtED">https://git.io/JUtED</a> |
| dot product               | tích vô hướng (hoặc tích trong) | <a href="https://git.io/JvKem">https://git.io/JvKem</a> |
| dropout                   | dropout                         |   |

## 22.5 E

| English                             | Tiếng Việt                  | Thảo luận tại   |
|-------------------------------------|-----------------------------|---|
| early stopping                      | dừng sớm                    |   |
| edge (in computer vision)           | biên                        | <a href="https://git.io/JfGiw">https://git.io/JfGiw</a> |
| edge detector                       | bộ phát hiện biên           | <a href="https://git.io/JfGiw">https://git.io/JfGiw</a> |
| effect size                         | hệ số ảnh hưởng             | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| eigen-decomposition                 | phân rã trị riêng           | <a href="https://git.io/JUsrl">https://git.io/JUsrl</a> |
| eigenvalue                          | trị riêng                   |   |
| eigenvector                         | vector riêng                |   |
| elementwise                         | (theo) từng phần tử         | <a href="https://git.io/Jvojn">https://git.io/Jvojn</a> |
| embedding                           | embedding                   | <a href="https://git.io/JvKeY">https://git.io/JvKeY</a> |
| encoder-decoder architecture        | kiến trúc mã hóa - giải mã  |   |
| end-to-end                          | đầu-cuối                    | <a href="https://git.io/JvQxG">https://git.io/JvQxG</a> |
| epoch (in training)                 | epoch (khi huấn luyện)      | <a href="https://git.io/Jvoha">https://git.io/Jvoha</a> |
| error analysis                      | phân tích lỗi               |   |
| error rate                          | tỉ lệ lỗi                   |   |
| estimator                           | bộ ước lượng                |   |
| evaluation metric                   | phép đánh giá               |   |
| example                             | mẫu                         |   |
| expectation                         | kỳ vọng                     | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a> |
| explicit feedback                   | phản hồi trực tiếp          | <a href="https://git.io/JvKee">https://git.io/JvKee</a> |
| exploding gradient                  | bùng nổ gradient            |   |
| exponential distribution            | phân phối mũ                | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |
| expression (in math)                | biểu thức (toán học)        | <a href="https://git.io/Jvojk">https://git.io/Jvojk</a> |
| eyeball dev set                     | tập phát triển eyeball      | <a href="https://git.io/JvQx3">https://git.io/JvQx3</a> |
| exponential weighted moving average | trung bình động trọng số mũ |   |

## 22.6 F

| English                       | Tiếng Việt                                 | Thảo luận tại   |
|-------------------------------|--|---|
| F1 score                      | chỉ số F1                                  |   |
| false negative                | âm tính giả                                |   |
| false positive                | dương tính giả                             |   |
| feature                       | đặc trưng                                  |   |
| feature extraction            | trích xuất đặc trưng                       |   |
| feature map (in CNN)          | ánh xạ đặc trưng                           |   |
| feed-forward network (FNN)    | mạng truyền xuôi                           |   |
| fine-tuning                   | tinh chỉnh                                 |   |
| filter (in CNN)               | bộ lọc                                     | <a href="https://git.io/JfeI1">https://git.io/JfeI1</a> |
| fit                           | khớp                                       | <a href="https://git.io/JvKet">https://git.io/JvKet</a> |
| first principle               | định đề cơ bản                             | <a href="https://git.io/JvKet">https://git.io/JvKet</a> |
| flatten                       | trải phẳng                                 | <a href="https://git.io/JvohO">https://git.io/JvohO</a> |
| forward pass                  | lượt truyền xuôi                           | <a href="https://git.io/JvohG">https://git.io/JvohG</a> |
| forward inference             | suy luận xuôi                              |   |
| framework                     | framework                                  |   |
| frontend                      | front-end                                  |   |
| frozen weight (freeze weight) | trọng số bị đóng băng (đóng băng trọng số) |   |
| functional analysis           | giải tích hàm                              |   |
| fully-connected               | kết nối đầy đủ                             | <a href="https://git.io/JvohR">https://git.io/JvohR</a> |
| fully convolutional network   | mạng tích chập đầy đủ                      |   |

## 22.7 G

| English                        | Tiếng Việt                        | Thảo luận tại   |
|--------------------------------|-----------------------------------|---|
| Gaussian distribution          | phân phối Gauss (phân phối chuẩn) | <a href="https://git.io/JvohV">https://git.io/JvohV</a>   |
| Gaussian noise                 | nhiễu Gauss                       |   |
| gated recurrent unit           | nút hồi tiếp có cổng              |   |
| generalization error           | lỗi khái quát                     | <a href="https://git.io/Jvohm">https://git.io/Jvohm</a>   |
| generalization gap             | khoảng cách khái quát             | <a href="https://git.io/Jvoht">https://git.io/Jvoht</a>   |
| generalization loss            | mất mát khái quát                 | <a href="https://git.io/Jvoht">https://git.io/Jvoht</a>   |
| generative adversarial network | mạng đối sinh                     | <a href="https://git.io/JvojD">https://git.io/JvojD</a>   |
| generative model               | mô hình sinh                      | <a href="https://git.io/Jvojd">https://git.io/Jvojd</a>   |
| global interpreter lock        | khóa trình thông dịch toàn cục    | <a href="https://git.io/JfGiV">https://git.io/JfGiV</a>   |
| global maximum                 | giá trị lớn nhất                  | <a href="https://git.io/Jvopx">https://git.io/Jvopx</a>   |
| global minimum                 | giá trị nhỏ nhất                  | <a href="https://git.io/Jvopx">https://git.io/Jvopx</a>   |
| gradient clipping              | gọt gradient                      |   |
| gradient descent               | hạ gradient                       | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> , <a href="https://git.io/JvQxW">https://git.io/JvQxW</a> |
| graphical model                | mô hình đồ thị                    |   |
| greedy algorithm               | thuật toán tham lam               |   |
| greedy search                  | tìm kiếm tham lam                 |   |
| ground truth                   | nhãn gốc                          | <a href="https://git.io/JvQxl">https://git.io/JvQxl</a>   |
| growth rate                    | tốc độ tăng trưởng                |   |

## 22.8 H

| English                 | Tiếng Việt               | Thảo luận tại   |
|-------------------------|--------------------------|---|
| hand-engineering        | thiết kế thủ công        |   |
| helper (function)       | (hàm) hỗ trợ             |   |
| heuristic               | thực nghiệm              |   |
| hybrid(ize)             | hybrid (hóa)             | <a href="https://git.io/JJGwt">https://git.io/JJGwt</a> , <a href="https://git.io/JJGwq">https://git.io/JJGwq</a> |
| hidden state            | trạng thái ẩn            |   |
| hidden state variable   | biến trạng thái ẩn       |   |
| hidden unit             | nút ẩn                   |   |
| hidden variable         | biến ẩn                  |   |
| human-level performance | chất lượng mức con người | <a href="https://git.io/JvQx4">https://git.io/JvQx4</a> , <a href="https://git.io/JvQxB">https://git.io/JvQxB</a> |
| hyper parameter         | siêu tham số             |   |
| hyperplane              | siêu phẳng               | <a href="https://git.io/JvojD">https://git.io/JvojD</a>   |
| hypothesis test         | kiểm định giả thuyết     | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>   |

## 22.9 I

| English                                     | Tiếng Việt                   | Thảo luận tại   |
|---|------------------------------|---|
| inference (in model training and inference) | dự đoán / suy luận           | <a href="https://git.io/JJVT2">https://git.io/JJVT2</a> |
| (statistical) inference                     | suy luận (thống kê)          | <a href="https://git.io/JJVT2">https://git.io/JJVT2</a> |
| ill-conditioned                             | (có) điều kiện xấu           |   |
| image segmentation                          | phân vùng ảnh                |   |
| implement                                   | lập trình                    | <a href="https://git.io/JvohG">https://git.io/JvohG</a> |
| implementation                              | cách lập trình               | <a href="https://git.io/JvohG">https://git.io/JvohG</a> |
| implicit feedback                           | phản hồi gián tiếp           |   |
| import (module, package)                    | nhập (mô-đun, gói thư viện)  | <a href="https://git.io/JvQxK">https://git.io/JvQxK</a> |
| imputation (in data preprocessing)          | quy buộc                     | <a href="https://git.io/Jvoh9">https://git.io/Jvoh9</a> |
| independence assumption                     | giả định độc lập             | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a> |
| indicator variable                          | biến chỉ định                | <a href="https://git.io/JvQha">https://git.io/JvQha</a> |
| inductive bias                              | thiên kiến quy nạp           |   |
| initializer                                 | bộ khởi tạo                  | <a href="https://git.io/Jfe1U">https://git.io/Jfe1U</a> |
| instance segmentation                       | phân vùng thực thể           |   |
| internal covariate shift                    | dịch chuyển hiệp biến nội bộ |   |
| intersection over union (IoU)               | phần giao trên phần hợp      |   |
| iteration                                   | vòng lặp                     |   |
| iterator                                    | iterator                     | <a href="https://git.io/JvohG">https://git.io/JvohG</a> |

## 22.10 J

| English            | Tiếng Việt          | Thảo luận tại   |
|--------------------|---------------------|---|
| joint distribution | phân phối đồng thời | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a> |

## 22.11 K

| English                 | Tiếng Việt               | Thảo luận tại   |
|-------------------------|--------------------------|---|
| k-fold cross validation | kiểm định chéo gập k-lần | <a href="https://git.io/JvQxK">https://git.io/JvQxK</a>   |
| kernel                  | hạt nhân                 | <a href="https://git.io/Jfe1I">https://git.io/Jfe1I</a> , <a href="https://git.io/Jf0vK">https://git.io/Jf0vK</a> |
| key                     | khóa                     |   |

## 22.12 L

| English                            | Tiếng Việt                     | Thảo luận tại   |
|------------------------------------|--------------------------------|---|
| long short-term Memory (LSTM)      | bộ nhớ ngắn hạn dài            | <a href="https://git.io/JvKeI">https://git.io/JvKeI</a> |
| label smoothing                    | làm mượt nhãn                  |   |
| language model(ling)               | mô hình (hoá) ngôn ngữ         |   |
| masked language model              | mô hình ngôn ngữ có mặt nạ     |   |
| latent variable                    | biến tiềm ẩn                   |   |
| law of large numbers               | luật số lớn                    | <a href="https://git.io/JvohQ">https://git.io/JvohQ</a> |
| layer                              | tầng                           |   |
| leaky                              | rò rỉ                          |   |
| learning algorithm                 | thuật toán học                 |   |
| learning curve                     | đồ thị quá trình học           | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> |
| learning rate                      | tốc độ học                     |   |
| learning rate schedule             | định thời tốc độ học           |   |
| line search                        | tìm kiếm đường thẳng           |   |
| linear                             | tuyến tính                     | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| linear algebra                     | đại số tuyến tính              |   |
| linear dependence                  | phụ thuộc tuyến tính           | <a href="https://git.io/JvKet">https://git.io/JvKet</a> |
| linear discriminant analysis (LDA) | phân tích biệt thức tuyến tính | <a href="https://git.io/Jvojw">https://git.io/Jvojw</a> |
| linear form                        | dạng tuyến tính                | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| linear independence                | độc lập tuyến tính             | <a href="https://git.io/JvKet">https://git.io/JvKet</a> |
| linear programming                 | quy hoạch tuyến tính           | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| linear regression                  | hồi quy tuyến tính             |   |
| local maximum                      | cực đại                        |   |
| local minimum                      | cực tiểu                       |   |
| locality                           | tính cục bộ                    |   |
| log-likelihood function            | hàm log hợp lý                 | <a href="https://git.io/Jvopx">https://git.io/Jvopx</a> |
| logistic regression                | hồi quy logistic               |   |
| logit (softmax)                    | logit                          | <a href="https://git.io/JvohR">https://git.io/JvohR</a> |
| loss function                      | hàm mất mát                    | <a href="https://git.io/Jvojp">https://git.io/Jvojp</a> |
| loss landscape                     | cánh quan mất mát              |   |
| loss surface                       | bề mặt mất mát                 |   |

## 22.13 M

| English                           | Tiếng Việt                      | Thảo luận tại   |
|-----------------------------------|---------------------------------|---|
| machine learning                  | học máy                         |   |
| machine translation               | dịch máy                        |   |
| marginalization                   | phép biên hóa                   | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a> |
| mask (in computer vision)         | mặt nạ                          |   |
| masked multi-head attention layer | tầng tập trung đa đầu có mặt nạ |   |
| matrix factorization              | phân rã ma trận                 | <a href="https://git.io/JUsrl">https://git.io/JUsrl</a> |
| max pooling                       | gộp cực đại                     | <a href="https://git.io/JfGi6">https://git.io/JfGi6</a> |
| maximize (in optimization)        | cực đại hóa                     |   |
| maximum likelihood estimator      | bộ ước lượng hợp lý cực đại     |   |
| mean squared error (MSE)          | trung bình bình phương sai số   | <a href="https://git.io/Jvojr">https://git.io/Jvojr</a> |
| memory cell                       | ô nhớ                           |   |
| metric                            | phép đo                         |   |
| minibatch                         | minibatch                       | <a href="https://git.io/JvojE">https://git.io/JvojE</a> |
| minimize (in optimization)        | cực tiểu hóa                    |   |
| minimizer                         | nghiệm cực tiểu                 |   |
| misclassified                     | bị phân loại nhầm               |   |
| mislabeled                        | bị gán nhãn nhầm                |   |
| model                             | mô hình                         |   |
| model capacity                    | năng lực mô hình                | <a href="https://git.io/JvQA5">https://git.io/JvQA5</a> |
| model family                      | nhóm mô hình                    |   |
| module                            | mô-đun                          |   |
| moment                            | mô-men                          |   |
| momentum                          | động lượng                      |   |
| multi-armed bandit                | máy đánh bạc đa cần             | <a href="https://git.io/Jfe1v">https://git.io/Jfe1v</a> |
| multi-class classification        | phân loại đa lớp                | <a href="https://git.io/Jvoj0">https://git.io/Jvoj0</a> |
| multi-head attention layer        | tầng tập trung đa đầu           |   |
| multinomial distribution          | phân phối đa thức               | <a href="https://git.io/JvohQ">https://git.io/JvohQ</a> |
| multitask learning                | học đa nhiệm                    | <a href="https://git.io/JvohQ">https://git.io/JvohQ</a> |

## 22.14 N

| English                              | Tiếng Việt                        | Thảo luận tại   |
|--------------------------------------|-----------------------------------|---|
| named entity (recognition)           | (nhận dạng) thực thể có tên       | <a href="https://git.io/JvojG">https://git.io/JvojG</a>   |
| natural language inference (NLI)     | suy luận ngôn ngữ tự nhiên        |   |
| natural language processing (NLP)    | xử lý ngôn ngữ tự nhiên           |   |
| natural language understanding (NLU) | hiểu ngôn ngữ tự nhiên            |   |
| negative log-likelihood function     | hàm đối log hợp lý                |   |
| negative sample/example              | mẫu âm                            |   |
| network in network                   | mạng trong mạng                   | <a href="https://git.io/JfGi1">https://git.io/JfGi1</a>   |
| neural network                       | mạng nơ-ron                       | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> , <a href="https://git.io/JvQxR">https://git.io/JvQxR</a> |
| neural style transfer                | truyền tài phong cách nơ-ron      | <a href="https://git.io/JJyeI">https://git.io/JJyeI</a>   |
| node (in neural networks)            | nút                               | <a href="https://git.io/Jvohm">https://git.io/Jvohm</a>   |
| noise-injection                      | thêm nhiễu                        |   |
| non-maximum suppression              | triệt phi cực đai                 | <a href="https://git.io/JJXrQ">https://git.io/JJXrQ</a>   |
| non-squashing activation function    | hàm kích hoạt không ép            |   |
| nonstationary distribution           | phân phối không dừng              | <a href="https://git.io/Jfe1M">https://git.io/Jfe1M</a>   |
| nonparametric                        | phi tham số                       |   |
| norm                                 | chuẩn                             | <a href="https://git.io/JvKem">https://git.io/JvKem</a>   |
| normal distribution                  | phân phối chuẩn (phân phối Gauss) | <a href="https://git.io/JvohV">https://git.io/JvohV</a>   |
| normalize                            | chuẩn hóa                         |   |
| null hypothesis                      | giả thuyết gốc                    | <a href="https://git.io/Jvoj1">https://git.io/Jvoj1</a>   |
| numerical solution                   | nghiệm xấp xỉ                     |   |

## 22.15 O

| English                | Tiếng Việt         | Thảo luận tại   |
|------------------------|--------------------|---|
| object detection       | phát hiện vật thể  |   |
| object recognition     | nhận dạng vật thể  |   |
| objective function     | hàm mục tiêu       | <a href="https://git.io/Jvojp">https://git.io/Jvojp</a> |
| offline learning       | học ngoại tuyến    | <a href="https://git.io/Jvojd">https://git.io/Jvojd</a> |
| offset                 | độ dời             | <a href="https://git.io/JfwX5">https://git.io/JfwX5</a> |
| one-hot encoding       | biểu diễn one-hot  | <a href="https://git.io/JvohR">https://git.io/JvohR</a> |
| one-sided test         | kiểm định một phía | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| one-tailed test        | kiểm định một đuôi | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| optimization landscape | cánh quan tối ưu   | <a href="https://git.io/Jfwf3">https://git.io/Jfwf3</a> |
| optimizing metric      | phép đo để tối ưu  | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> |
| orthogonal             | trực giao          | <a href="https://git.io/JvKem">https://git.io/JvKem</a> |
| orthonormal            | trực chuẩn         | <a href="https://git.io/JvKem">https://git.io/JvKem</a> |
| overfit                | quá khớp           | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> |
| overflow (numerical)   | tràn (số) trên     | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |

## 22.16 P

| English                             | Tiếng Việt                   | Thảo luận tại   |
|-------------------------------------|------------------------------|---|
| p-value                             | trị số p                     | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| padding                             | đệm                          | <a href="https://git.io/Jfe1I">https://git.io/Jfe1I</a> |
| partition function                  | hàm phân hoạch               | <a href="https://git.io/JvQxE">https://git.io/JvQxE</a> |
| pattern recognition                 | nhận dạng mẫu                | <a href="https://git.io/JvKeL">https://git.io/JvKeL</a> |
| penalty                             | lượng phạt                   | <a href="https://git.io/JvQAP">https://git.io/JvQAP</a> |
| perceptron                          | perceptron                   | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |
| performance                         | chất lượng                   | <a href="https://git.io/JvQx4">https://git.io/JvQx4</a> |
| perplexity (metric in NLP)          | perplexity                   | <a href="https://git.io/Jf9KY">https://git.io/Jf9KY</a> |
| perturbation                        | nhiều                        | <a href="https://git.io/JvQA1">https://git.io/JvQA1</a> |
| pipeline                            | pipeline                     | <a href="https://git.io/JvQxG">https://git.io/JvQxG</a> |
| pixel (component of digital images) | điểm ảnh                     |   |
| pixel (unit of measurement)         | pixel (đơn vị đo)            |   |
| plateau (noun)                      | vùng nằm ngang               |   |
| plateau (verb)                      | nằm ngang                    |   |
| policy (in reinforcement learning)  | chính sách                   | <a href="https://git.io/Jvoj9">https://git.io/Jvoj9</a> |
| pooling                             | gộp                          | <a href="https://git.io/Jfe1I">https://git.io/Jfe1I</a> |
| population                          | tổng thể                     | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| position-wise feed-forward network  | mạng truyền xuôi theo vị trí |   |
| positional encoding                 | biểu diễn vị trí             |   |
| positive sample/example             | mẫu dương                    |   |
| posterior                           | hậu nghiệm                   | <a href="https://git.io/JvQA6">https://git.io/JvQA6</a> |
| precision (vs accuracy metric)      | precision                    | <a href="https://git.io/JJ9sl">https://git.io/JJ9sl</a> |
| preconditioning                     | tiền điều kiện               |   |
| premise                             | tiền đề                      | <a href="https://git.io/JUZKL">https://git.io/JUZKL</a> |

continues on next page

Table 22.16.1 – continued from previous page

|                                    |                            |   |
|------------------------------------|----------------------------|---|
| English                            | Tiếng Việt                 | Thảo luận tại   |
| pre-train                          | tiền huấn luyện            | <a href="https://git.io/JJ1HO">https://git.io/JJ1HO</a> |
| principal component analysis (PCA) | phân tích thành phần chính | <a href="https://git.io/JvojD">https://git.io/JvojD</a> |
| prior                              | tiên nghiệm                | <a href="https://git.io/JvQA6">https://git.io/JvQA6</a> |
| probability theory                 | lý thuyết xác suất         |   |
| proposed region                    | vùng đề xuất               |   |
| proxy (in statistics)              | biểu đại diện              |   |

## 22.17 Q

|                       |                       |   |
|-----------------------|-----------------------|---|
| English               | Tiếng Việt            | Thảo luận tại   |
| quadratic             | toàn phương           | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| quadratic form        | dạng toàn phương      | <a href="https://git.io/JvohV">https://git.io/JvohV</a> |
| quadratic programming | quy hoạch toàn phương |   |
| query                 | (câu) truy vấn        |   |

## 22.18 R

| English  | Tiếng Việt                                | Thảo luận tại   |
|--|---|---|
| random variable                                    | biến ngẫu nhiên                           |   |
| recall   | recall                                    |   |
| receptive field (in CNN)                           | vùng tiếp nhận                            | <a href="https://git.io/Jftwh">https://git.io/Jftwh</a> |
| recommender system                                 | hệ thống đề xuất                          | <a href="https://git.io/JUg62">https://git.io/JUg62</a> |
| recognition  | nhận dạng                                 |   |
| rectified linear unit (ReLU)                       | đơn vị tuyến tính chỉnh lưu               | <a href="https://git.io/JvohI">https://git.io/JvohI</a> |
| recurrent neural network                           | mạng nơ-ron hồi tiếp                      |   |
| region of interest                                 | vùng quan tâm                             | <a href="https://git.io/JJokG">https://git.io/JJokG</a> |
| region of rejection                                | miền bác bỏ                               | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| regressor  | bộ hồi quy                                | <a href="https://git.io/JvKee">https://git.io/JvKee</a> |
| regularization                                     | điều chuẩn                                |   |
| reinforcement learning                             | học tăng cường                            |   |
| relative loss                                      | mất mát tương đối                         | <a href="https://git.io/JvQAH">https://git.io/JvQAH</a> |
| remote   | từ xa                                     |   |
| reparameterization                                 | tái tham số hóa                           |   |
| representation learning                            | học biểu diễn                             | <a href="https://git.io/JvojG">https://git.io/JvojG</a> |
| reproduce (reproducibility)                        | tái tạo (khả năng tái tạo)                | <a href="https://git.io/JUs2X">https://git.io/JUs2X</a> |
| rescale  | (phép) tái tỉ lệ                          | <a href="https://git.io/JfeXx">https://git.io/JfeXx</a> |
| reset gate   | cổng xóa                                  |   |
| residual network                                   | mạng phần dư                              | <a href="https://git.io/JfGi1">https://git.io/JfGi1</a> |
| reward function                                    | hàm điểm thưởng                           |   |
| robust (adjective for models, algorithms, systems) | mạnh mẽ                                   | <a href="https://git.io/Jfe1e">https://git.io/Jfe1e</a> |
| robust to noise                                    | kháng nhiễu                               | <a href="https://git.io/JvQA1">https://git.io/JvQA1</a> |
| root mean squared error (RMSE)                     | căn bậc hai trung bình bình phương sai số | <a href="https://git.io/Jvojr">https://git.io/Jvojr</a> |
| running time                                       | thời gian chạy                            |   |

## 22.19 S

| English                             | Tiếng Việt                               | Thảo luận tại  |
|-------------------------------------|--|--|
| sampling with replacement           | lấy mẫu có hoàn lại                      | <a href="https://git.io/JvQxu">https://git.io/JvQxu</a>        |
| sampling without replacement        | lấy mẫu không hoàn lại                   | <a href="https://git.io/JvQxu">https://git.io/JvQxu</a>        |
| satisficing metric                  | phép đo thỏa mãn                         | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a>        |
| scale                               | (phép) biến đổi tỉ lệ                    |  |
| scalar                              | số vô hướng                              | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>        |
| scale invariant                     | bất biến quy mô                          | <a href="https://git.io/Jftwj">https://git.io/Jftwj</a>        |
| scoring function                    | hàm tính điểm                            |  |
| serialization (programming)         | chuỗi hóa                                |  |
| semantic role labeling              | dán nhãn vai trò ngữ nghĩa               |  |
| semantic segmentation               | phân vùng theo ngữ nghĩa                 |  |
| sentiment classification (analysis) | phân loại (phân tích) cảm xúc            |  |
| sequence-aware (recommender system) | (hệ thống đề xuất) có nhận thức về chuỗi | <a href="https://git.io/JUWLM">https://git.io/JUWLM</a>        |
| sequence learning                   | học chuỗi                                | <a href="https://git.io/JvQxa">https://git.io/JvQxa</a>        |
| sequence to sequence                | chuỗi sang chuỗi                         |  |
| sequential partitioning             | phân tách tuần tự                        |  |
| server                              | máy chủ                                  |  |
| sensitivity                         | độ nhạy                                  | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>        |
| shape (in linear algebra)           | kích thước                               | <a href="https://git.io/Jvojn">https://git.io/Jvojn</a>        |
| significance test                   | kiểm định ý nghĩa                        | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>        |
| skip-gram (model)                   | (mô hình) skip-gram                      |  |
| significance test                   | kiểm định ý nghĩa                        | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>        |
| slicing (an array)                  | cắt chọn (mảng)                          | <a href="https://git.io/JvohH">https://git.io/JvohH</a>        |
| source data / distribution          | dữ liệu / phân phối gốc                  | <a href="https://git.io/JvQAy">https://git.io/JvQAy</a>        |
| spam email                          | email rác                                |  |
| speech recognition                  | nhận dạng giọng nói                      |  |
| squashing function                  | hàm ép                                   | <a href="https://git.io/JvQA5">https://git.io/JvQA5</a>        |
| standard deviation                  | độ lệch chuẩn                            | <a href="https://git.io/Jvohb">https://git.io/Jvohb</a>        |
| standardize                         | chuẩn hóa                                |  |
| state-of-the-art                    | tân tiến nhất                            |  |
| stationary point                    | điểm dừng                                | <a href="https://git.io/JvohC">https://git.io/JvohC</a>        |
| statistical inference               | suy luận thống kê                        |  |
| statistical power                   | năng lực thống kê                        | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a>        |
| statistical significance            | ý nghĩa thống kê                         | <a href="https://git.io/Jvoj1">https://git.io/Jvoj1</a>        |
| statistical significant             | có ý nghĩa thống kê                      | <a href="https://git.io/Jvoj1">https://git.io/Jvoj1</a>        |
| stochastic gradient descent         | hạ gradient ngẫu nhiên                   |  |
| stop word                           | từ dừng                                  |  |
| style loss (in style transfer)      | mất mát phong cách                       | <a href="https://git.io/JJyeI">https://git.io/JJyeI</a>        |
| stride                              | sải bước                                 | <a href="https://git.io/Jfe1I">https://git.io/Jfe1I</a>        |
| subscript                           | chỉ số dưới                              | ' <a href="https://git.io/Jvoh1">https://git.io/Jvoh1</a> <h t |
| subspace estimation                 | ước lượng không gian con                 | <a href="https://git.io/JvoJD">https://git.io/JvoJD</a>        |
| subword (NLP)                       | từ con                                   | <a href="https://git.io/JUkWJ">https://git.io/JUkWJ</a>        |
| superscript                         | chỉ số trên                              | ' <a href="https://git.io/Jvoh1">https://git.io/Jvoh1</a> <h t |
| supervised learning                 | học có giám sát                          |  |
| support vector machine (SVM)        | Máy vector hỗ trợ                        |  |

Table 22.19.1 – continued from previous page

|                                   |                   |   |
|-----------------------------------|-------------------|---|
| English                           | Tiếng Việt        | Thảo luận tại   |
| surprisal (in information theory) | lượng tin         | <a href="https://git.io/Jvoh3">https://git.io/Jvoh3</a> |
| surrogate objective               | mục tiêu thay thế | <a href="https://git.io/JvQxV">https://git.io/JvQxV</a> |
| switch                            | bộ chuyển mạch    |   |
| symbolic graph                    | đồ thị biểu tượng | <a href="https://git.io/JvojU">https://git.io/JvojU</a> |

## 22.20 T

|  |   |   |
|--|---|---|
| English                                | Tiếng Việt                                | Thảo luận tại   |
| Taylor expansion                       | khai triển Taylor                         |   |
| target data / distribution             | dữ liệu / phân phối mục tiêu              | <a href="https://git.io/JvQAY">https://git.io/JvQAY</a> |
| task-specific                          | đặc thù cho tác vụ                        |   |
| task-agnostic                          | không phân biệt tác vụ                    |   |
| tensor contraction                     | phép co tensor                            | <a href="https://git.io/JvojX">https://git.io/JvojX</a> |
| test set                               | tập kiểm tra                              |   |
| test set performance                   | chất lượng trên tập kiểm tra              |   |
| test statistic                         | tiêu chuẩn kiểm định                      | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| text data                              | dữ liệu văn bản                           |   |
| text tagging                           | gán thẻ văn bản                           | <a href="https://git.io/JUmu0">https://git.io/JUmu0</a> |
| timeseries analysis                    | phân tích dữ liệu chuỗi thời gian         |   |
| timestep                               | bước thời gian                            | <a href="https://git.io/JvojQ">https://git.io/JvojQ</a> |
| token                                  | token                                     |   |
| total variation                        | biến thiên toàn phần                      | <a href="https://git.io/JJyeI">https://git.io/JJyeI</a> |
| total variation denoising              | khử nhiễu biến thiên toàn phần            |   |
| training dev set                       | tập phát triển huấn luyện                 |   |
| training set                           | tập huấn luyện                            |   |
| training set performance               | chất lượng trên tập huấn luyện            |   |
| transcribe                             | phiên thoại                               | <a href="https://git.io/JvojN">https://git.io/JvojN</a> |
| transcription                          | bản ghi thoại                             |   |
| transfer learning                      | học truyền tải                            |   |
| transformer                            | transformer                               |   |
| transition layer                       | tầng chuyển tiếp                          |   |
| translation invariant                  | bất biến tịnh tiến                        | <a href="https://git.io/Jftwj">https://git.io/Jftwj</a> |
| transposed convolution                 | tích chập chuyển vị                       | <a href="https://git.io/JJ1HU">https://git.io/JJ1HU</a> |
| true negative                          | âm tính thật                              |   |
| true positive                          | dương tính thật                           |   |
| truncated backpropagation through time | lan truyền ngược qua thời gian bị cắt xén |   |
| tune parameters                        | điều chỉnh tham số                        |   |
| two-sided test                         | kiểm định hai phía                        | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |
| two-tailed test                        | kiểm định hai đuôi                        | <a href="https://git.io/Jvoja">https://git.io/Jvoja</a> |

## 22.21 U

| English                   | Tiếng Việt                | Thảo luận tại   |
|---------------------------|---------------------------|---|
| unavoidable bias          | độ chêch không tránh được |   |
| underfit                  | dưới khớp                 | <a href="https://git.io/JvQxY">https://git.io/JvQxY</a> |
| underflow (numerical)     | tràn (số) dưới            | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |
| unit (in neural networks) | nút                       | <a href="https://git.io/Jvohm">https://git.io/Jvohm</a> |
| unsupervised learning     | học không giám sát        |   |
| upsample                  | tăng mẫu                  | <a href="https://git.io/JvohC">https://git.io/JvohC</a> |
| upstream task             | tác vụ ngược dòng         | <a href="https://git.io/JUtED">https://git.io/JUtED</a> |

## 22.22 V

| English                    | Tiếng Việt         | Thảo luận tại   |
|----------------------------|--------------------|---|
| validation set             | tập kiểm định      | <a href="https://git.io/Jvohm">https://git.io/Jvohm</a> |
| value                      | giá trị            |   |
| vanishing gradient         | tiêu biến gradient | <a href="https://git.io/JvohI">https://git.io/JvohI</a> |
| variance (of an estimator) | phương sai         | <a href="https://git.io/JvQxO">https://git.io/JvQxO</a> |
| variational                | biến phân          |   |
| vector                     | vector             |   |
| vectorization              | vector hóa         |   |

## 22.23 W

| English                                   | Tiếng Việt        | Thảo luận tại   |
|---|-------------------|---|
| warmup (in learning rate scheduling)      | khởi động         | <a href="https://git.io/JJIT5">https://git.io/JJIT5</a> |
| weight decay                              | suy giảm trọng số | <a href="https://git.io/JvQxK">https://git.io/JvQxK</a> |
| well-behaved function (analytic function) | hàm khả vi vô hạn | <a href="https://git.io/JvojL">https://git.io/JvojL</a> |
| whitening data                            | tẩy trắng dữ liệu |   |
| worker (in distributed system)            | máy thợ           |   |
| wrapper function (in programming)         | hàm wrapper       | <a href="https://git.io/Jvohm">https://git.io/Jvohm</a> |



# Bibliography

- Ahmed, A., Aly, M., Gonzalez, J., Narayananamurthy, 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*.
- 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.
- Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5, 135–146.
- 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).
- Campbell, M., Hoane Jr, A. J., & Hsu, F.-h. (2002). Deep blue. *Artificial intelligence*, 134(1-2), 57–83.
- 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*.
- 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.
- 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*.
- Edelman, B., Ostrovsky, M., & Schwarz, M. (2007). Internet advertising and the generalized second-price auction: selling billions of dollars worth of keywords. *American economic review*, 97(1), 242–259.
- 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).
- Ginibre, J. (1965). Statistical ensembles of complex, quaternion, and real matrices. *Journal of Mathematical Physics*, 6(3), 440–449.
- 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<sup>480</sup>
- 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).
- 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.

<sup>480</sup> <https://doi.org/10.23915/distill.00006>

- Gunawardana, A., & Shani, G. (2015). Evaluating recommender systems. *Recommender systems handbook* (pp. 265–308). Springer.
- Guo, H., Tang, R., Ye, Y., Li, Z., & He, X. (2017). Deepfm: a factorization-machine based neural network for ctr prediction. *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (pp. 1725–1731).
- 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. (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).
- He, X., & Chua, T.-S. (2017). Neural factorization machines for sparse predictive analytics. *Proceedings of the 40th International ACM SIGIR conference on Research and Development in Information Retrieval* (pp. 355–364).
- He, X., Liao, L., Zhang, H., Nie, L., Hu, X., & Chua, T.-S. (2017). Neural collaborative filtering. *Proceedings of the 26th international conference on world wide web* (pp. 173–182).
- 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.
- Herlocker, J. L., Konstan, J. A., Borchers, A., & Riedl, J. (1999). An algorithmic framework for performing collaborative filtering. *22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 1999* (pp. 230–237).
- Hidasi, B., Karatzoglou, A., Baltrunas, L., & Tikk, D. (2015). Session-based recommendations with recurrent neural networks. *arXiv preprint arXiv:1511.06939*.
- 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*.
- 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).
- 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).
- Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, pp. 30–37.
- 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).
- Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., ... Stoyanov, V. (2019). Roberta: a robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

- 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*.
- 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.
- McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., ... others. (2013). Ad click prediction: a view from the trenches. *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1222–1230).
- 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.
- Parikh, A. P., Täckström, O., Das, D., & Uszkoreit, J. (2016). A decomposable attention model for natural language inference. *arXiv preprint arXiv:1606.01933*.
- 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*.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 9.
- 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).
- Rendle, S. (2010). Factorization machines. *2010 IEEE International Conference on Data Mining* (pp. 995–1000).
- Rendle, S., Freudenthaler, C., Gantner, Z., & Schmidt-Thieme, L. (2009). Bpr: bayesian personalized ranking from implicit feedback. *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence* (pp. 452–461).
- Rumelhart, D. E., Hinton, G. E., Williams, R. J., & others. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3), 1.
- 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.
- Sedhain, S., Menon, A. K., Sanner, S., & Xie, L. (2015). Autorec: autoencoders meet collaborative filtering. *Proceedings of the 24th International Conference on World Wide Web* (pp. 111–112).
- Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.
- 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.
- 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., & Narayananamurthy, 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).
- 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*.
- Tang, J., & Wang, K. (2018). Personalized top-n sequential recommendation via convolutional sequence embedding. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining* (pp. 565–573).

- 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.
- Treisman, A. M., & Gelade, G. (1980). A feature-integration theory of attention. *Cognitive psychology*, 12(1), 97–136.
- Töscher, A., Jahrer, M., & Bell, R. M. (2009). The bigchaos solution to the netflix grand prize. *Netflix prize documentation*, pp. 1–52.
- 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).
- 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*.
- 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).