



BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT THÀNH PHỐ HỒ CHÍ MINH
60 NĂM XÂY DỰNG VÀ PHÁT TRIỂN

LÊ MỸ HÀ

GIÁO TRÌNH

MẠNG NƠI RƠN HỌC SÂU VÀ ÚNG DỤNG



NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT
THÀNH PHỐ HỒ CHÍ MINH



HCMUTE

TS. LÊ MỸ HÀ

GIÁO TRÌNH
MẠNG NƠI RƠN HỌC SÂU
VÀ ỨNG DỤNG

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH - 2019

Giáo trình
**MẠNG NƠI NƠI HỌC SÂU
VÀ ỨNG DỤNG**

TS. LÊ MỸ HÀ

**NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

Khu phố 6, Phường Linh Trung, Quận Thủ Đức, TP Hồ Chí Minh
Dãy C, số 10-12 Đinh Tiên Hoàng, Phường Bến Nghé,
Quận 1, TP Hồ Chí Minh
ĐT: 028 6272 6361 – 028 6272 6390
E-mail: vnuhp@vnuhcm.edu.vn

Nhà xuất bản ĐHQG-HCM và tác giả/dối tác
liên kết giữ bản quyền®

Copyright © by VNU-HCM Press and author/
co-partnership All rights reserved

TRUNG TÂM SÁCH ĐẠI HỌC

Dãy C, số 10-12 Đinh Tiên Hoàng, Phường Bến Nghé,
Quận 1, TP Hồ Chí Minh
ĐT: 028 6681 7058 - 028 6272 6390 - 028 6272 6351
Website: <https://nxvhcm.edu.vn>

Chịu trách nhiệm xuất bản
ĐỖ VĂN BIÊN

Xuất bản năm 2019

Chịu trách nhiệm nội dung
ĐỖ VĂN BIÊN

Tổ chức bản thảo và chịu trách nhiệm về tác quyền
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM
Website: <http://hcmute.edu.vn>

Biên tập
LÊ THỊ MINH HUỆ

Sửa bản in
THANH HÀ

Số lượng 300 cuốn,
Khoảng 16 x 24 cm,
ĐKKHXB số: 1989-2019/CXBIPH/
04-103/ĐHQGTPHCM,
Quyết định XB số 105/QĐ-ĐHQGTPHCM
của NXB ĐHQG-HCM
cấp ngày 02/7/2019.
In tại: Công ty TNHH In &
bao bì Hưng Phú
Đ/c: 162A/1 – KP1A – P. An Phú –
TX. Thuận An – Bình Dương
Nộp lưu chiểu: Quý III/2019

Trình bày bìa
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM
Website: <http://hcmute.edu.vn>



ISBN: 978 – 604 – 73 – 7082 – 5

Giáo trình
MẠNG NƯỚC HỌC SÂU
VÀ ỨNG DỤNG

TS. LÊ MỸ HÀ

Bản tiếng Việt ©, TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HCM, NXB ĐHQG-HCM và CÁC TÁC GIẢ.

Bản quyền tác phẩm đã được bảo hộ bởi Luật Xuất bản và Luật Sở hữu trí tuệ Việt Nam. Nghiêm cấm mọi hình thức xuất bản, sao chụp, phát tán nội dung khi chưa có sự đồng ý của Trường Đại Học Sư Phạm Kỹ Thuật TP. HCM và Tác Giả.

ĐỀ CÓ SÁCH HAY, CẦN CHUNG TAY BẢO VỆ TÁC QUYỀN!

MỤC LỤC

LỜI NÓI ĐẦU	9
Chương 1: GIỚI THIỆU VỀ MẠNG NƠ-RON VÀ HỌC SÂU	11
1.1. NGUYÊN TẮC CƠ BẢN CỦA MẠNG NƠ-RON	11
1.1.1. KHÁI NIỆM CƠ BẢN VỀ MẠNG NƠ-RON	11
1.1.2. GIỚI THIỆU VỀ MẠNG NƠ-RON	12
1.1.3. THUẬT TOÁN PERCEPTRON	21
1.1.4. LAN TRUYỀN NGƯỢC VÀ MẠNG NHIỀU LỚP	32
1.1.5. MẠNG NHIỀU LỚP VỚI KERAS	50
1.1.6. BỐN THÀNH PHẦN TRONG CÔNG THỨC MẠNG NƠ-RON	63
1.1.7. KHỞI TẠO TRỌNG SỐ	65
1.1.8. KHỞI TẠO LIÊN TỤC	66
1.1.9. PHÂN PHỐI ĐỒNG NHẤT VÀ PHÂN PHỐI CHUẨN HÓA	66
1.1.10. ĐỒNG BỘ VÀ CHUẨN HÓA	67
1.1.11. SỰ KHÁC BIỆT TRONG TRIỂN KHAI KHỞI TẠO	67
1.1.12. TÓM TẮT	67
1.2. HỌC SÂU (DEEP LEARNING) LÀ GÌ?	68
1.2.1. LỊCH SỬ NGẮN GỌN VỀ MẠNG NƠ-RON VÀ HỌC SÂU	69
1.2.2. HỌC ĐẶC TRUNG PHÂN CẤP	73
1.2.3. TÓM TẮT	76
Chương 2: GIỚI THIỆU MẠNG NƠ-RON TÍCH CHẬP	77
2.1. MẠNG NƠ-RON TÍCH CHẬP	78
2.1.1. HIẾU VỀ PHÉP TÍCH CHẬP	78
2.1.2. CÁC KHÓI XÂY DỰNG MẠNG CNN	89
2.1.3. KIẾN TRÚC VÀ MÔ HÌNH HUÂN LUYỆN CHUNG	104
2.1.4. CÁC CNN CÓ BẤT BIẾN ĐỐI VỚI PHÉP DỊCH, XOAY VÀ THU NHỎ KHÔNG?	108
2.1.5. TÓM TẮT	109

2.2. HUẤN LUYỆN MẠNG CNN ĐẦU TIÊN	110
2.2.1. CÁU HÌNH THƯ VIỆN KERAS VÀ CHUYỂN ĐỔI ẢNH SANG DẠNG MẢNG	110
2.2.2. SHALLOWNET.....	114
2.2.3. TÓM TẮT	125
2.3. LUU VÀ TẢI MÔ HÌNH.....	124
2.3.1. MÔ HÌNH NỐI TIẾP VÀO Ô CÚNG	124
2.3.2. TẢI MÔ HÌNH ĐƯỢC HUẤN LUYỆN TRƯỚC TỪ Ô CÚNG	127
2.3.3. TÓM TẮT	130
2.4. LENET: NHẬN DẠNG CHỮ SỐ VIẾT TAY	130
2.4.1. KIẾN TRÚC LENET	131
2.4.2. TRIỂN KHAI LENET.....	132
2.4.3. LENET TRÊN TẬP DỮ LIỆU MNIST	134
2.4.4. TÓM TẮT	139
2.5. MINIVGGNET: ĐI SÂU HƠN VỚI CNNS	140
2.5.1. HỌ MẠNG VGG	141
2.5.2. TRIỂN KHAI MINIVGGNET	142
2.5.3. MINIVGGNET TRÊN CIFAR-10	146
2.5.4. TÓM TẮT	150
Chương 3: CƠ SỞ TOÁN HỌC CỦA MẠNG NƠ-RON	152
3.1. HỌC THEO THÔNG SỐ.....	152
3.1.1. GIỚI THIỆU VỀ PHÂN LOẠI TUYẾN TÍNH	153
3.1.2. VAI TRÒ CỦA CÁC HÀM TỒN THẤT	160
3.1.3. TÓM TẮT	167
3.2. PHƯƠNG PHÁP TỐI ƯU HÓA VÀ CHÍNH QUY HÓA	167
3.2.1. THUẬT TOÁN SUY GIẢM ĐỘ DỐC (GRADIENT DESCENT).....	168
3.2.2. GIẢM DÀN ĐỘ DỐC NGẪU NHIÊN (SGD).....	180
3.2.3. MỎ RỘNG THÊM VỀ SGD.....	186
3.2.4. CHÍNH QUY HÓA	189
3.2.5. TÓM TẮT	196
3.3. LẬP LỊCH BIẾU TỶ LỆ HỌC	197
3.3.1. GIẢM TỶ LỆ HỌC	197

3.3.2. TÓM TẮT	206
3.4. HIỆN TƯỢNG CHUA KHỚP VÀ QUÁ KHỚP.....	207
3.4.1. THẾ NÀO LÀ CHUA KHỚP VÀ QUÁ KHỚP	207
3.4.2. GIÁM SÁT QUÁ TRÌNH HUẤN LUYỆN	212
3.4.3. TÓM TẮT	215
3.5. MÔ HÌNH KIỂM TRA	216
3.5.1. KIỂM TRA CÁC CẢI TIẾN MÔ HÌNH MẠNG NƠ-RON	216
3.5.2. KIỂM TRA MẠNG NƠ RON TỐT NHẤT	220
3.5.3. TÓM TẮT	223
3.6. TRỰC QUAN KIẾN TRÚC MẠNG	223
3.6.1. TẦM QUAN TRỌNG TRỰC QUAN KIẾN TRÚC MẠNG NƠ-RON	223
3.6.2. TÓM TẮT	228
3.7. MẠNG NƠ-RON TÍCH CHẬP VƯỢT TRỘI ĐỀ PHÂN LOẠI	228
3.7.1. CNN TIÊN TIẾN NHẤT VỚI KERAS	229
3.7.2. PHÂN LOẠI ẢNH VỚI BỘ DỮ LIỆU IMAGENET ĐƯỢC HUẤN LUYỆN TRƯỚC BẰNG CNN	233
3.7.3. TÓM TẮT	238
Chương 4: ỨNG DỤNG HỌC SÂU VÀO LĨNH VỰC THỊ GIÁC MÁY TÍNH	240
4.1. CẤU HÌNH MÔI TRƯỜNG CHO NHÀ PHÁT TRIỂN (DEVELOPTER)	240
4.1.1. THƯ VIỆN VÀ GÓI HỖ TRỢ.....	241
4.1.2. TRƯỜNG HỢP DỰA TRÊN ĐÁM MÂY (CLOUD)	242
4.1.3. TÓM TẮT	242
4.2. NGUYÊN TẮC CƠ BẢN VỀ ẢNH	242
4.2.1. PIXELS: KHÔI XÂY DỰNG ẢNH	243
4.2.2. HỆ THỐNG TỌA ĐỘ ẢNH	247
4.2.3. TỶ LỆ MỎ RỘNG VÀ TỶ LỆ KHUNG HÌNH.....	249
4.2.4. TÓM TẮT	251
4.3. KHÁI NIỆM CƠ BẢN VỀ PHÂN LOẠI ẢNH	252
4.3.1. PHÂN LOẠI ẢNH LÀ GÌ ?.....	253

4.3.2. CÁC KIỂU HỌC	257
4.3.3. PHÂN LOẠI HỌC SÂU	261
4.3.4. TÓM TẮT	266
4.4. CÁC BỘ DỮ LIỆU ĐỂ PHÂN LOẠI ẢNH	267
4.4.1. MNIST	267
4.4.2. ĐỘNG VẬT: CHÓ, MÈO VÀ GÂU TRÚC	268
4.4.3. CIFAR-10	269
4.4.4. SMILES	270
4.4.5. KAGGLE: DOGS & CATS	271
4.4.6. FLOWERS-17	272
4.4.7. CALTECH-101	273
4.4.8. TINY IMAGENET 200	273
4.4.9. ADIENCE	273
4.4.10. IMAGENET	274
4.4.11. INDOOR CVPR (NHẬN DẠNG CẢNH TRONG NHÀ)	276
4.4.12. STANFORD CARS	276
4.4.13. TÓM TẮT	276
4.5. TRÌNH PHÂN LOẠI ẢNH ĐẦU TIÊN	277
4.5.1. LÀM VIỆC VỚI BỘ DỮ LIỆU ẢNH	277
4.5.2. K-NN: TRÌNH PHÂN LOẠI ĐƠN GIẢN	286
4.5.3. TÓM TẮT	283
4.6. NHẬN DẠNG MẶT CUỜI TRÊN KHUÔN MẶT	292
4.6.1. BỘ DỮ LIỆU SMILES	293
4.6.2. HUẤN LUYỆN MẶT CUỜI BẰNG CNN	293
4.6.3. THỰC HIỆN NHẬN DẠNG MẶT CUỜI THEO THỜI GIAN THỰC BẰNG CNN	298
4.6.4. TÓM TẮT	302

Chương 5: ỨNG DỤNG MẠNG NÓ-RON HỒI QUY

VÀO LĨNH VỰC XỬ LÝ NGÔN NGỮ304

5.1. MẠNG NÓ-RON HỒI QUY VÀ MẠNG LSTM	304
5.1.1. KHÁI NIỆM VỀ MẠNG NÓ-RON HỒI QUY	304
5.1.2. KHÁI NIỆM MẠNG LSTM	305
5.1.3. MÔ HÌNH CỦA MẠNG LSTM	306
5.1.4. DỰ ĐOÁN TRÌNH TỰ	307

5.1.5. DỰ ĐOÁN CHUỖI THỜI GIAN VỚI MÔ HÌNH LSTM	309
5.2. CHUYỂN ĐỒI LỜI NÓI THÀNH VĂN BẢN	
VÀ NGƯỢC LẠI	312
5.2.1. CHUYỂN ĐỒI LỜI NÓI THÀNH VĂN BẢN	312
5.2.2. LỜI NÓI DƯỚI DẠNG DỮ LIỆU	313
5.2.3. TÍNH NĂNG LỜI NÓI ÁNH XẠ LỜI NÓI THÀNH MA TRẬN	313
5.2.4. QUANG PHÔ: ÁNH XẠ LỜI NÓI THÀNH HÌNH ẢNH..	315
5.2.5. XÂY DỰNG BỘ PHÂN LOẠI ĐỂ NHẬN DẠNG GIỌNG NÓI THÔNG QUA CÁC TÍNH NĂNG MEL-FREQUENCY CEPSTRAL COEFFICIENT (MFCC).....	315
5.2.6. XÂY DỰNG BỘ PHÂN LOẠI ĐỂ NHẬN DẠNG GIỌNG NÓI THÔNG QUA QUANG PHÔ	316
5.2.7. CÁCH TIẾP CẬN NGUỒN MỎ ĐỂ NHẬN DẠNG GIỌNG NÓI	317
5.2.8. TƯƠNG LAI CỦA CHUYỂN ĐỒI LỜI NÓI THÀNH VĂN BẢN	318
5.3. PHÁT TRIỂN ỨNG DỤNG CHATBOT.....	318
5.3.1. TẠI SAO CÓ CHATBOT?.....	319
5.3.2. THIẾT KẾ VÀ CHỨC NĂNG CỦA CHATBOT	319
5.3.3. CÁC BƯỚC XÂY DỰNG CHATBOT	319
5.3.4. THỰC TIỄN TỐT NHẤT VỀ PHÁT TRIỂN CHATBOT.....	332
TÀI LIỆU THAM KHẢO.....	334

LỜI NÓI ĐẦU

Mạng Noron hay mạng Noron học sâu (Deep Learning Network) là lĩnh vực nghiên cứu các thuật toán, chương trình máy tính để làm sao máy móc có thể học được các tri thức và ra quyết định như con người. Đây cũng là một nhánh quan trọng của lĩnh vực trí tuệ nhân tạo đang bùng nổ nhanh chóng về số lượng người nghiên cứu cũng như các công trình mới được công bố. Những năm gần đây, hầu hết các quốc gia phát triển và cũng như đang phát triển trên thế giới có sự đầu tư mạnh mẽ về nhân lực, thiết bị để phát triển nền tảng lý thuyết cũng như đưa ra các ứng dụng quan trọng của lĩnh vực trí tuệ nhân tạo vào giải quyết các vấn đề khoa học, kỹ thuật và các lĩnh vực khác của đời sống con người.

Không nằm ngoài xu thế đó, cuốn sách này ra đời nhằm mục đích giúp người học có được các định nghĩa cơ bản về mạng Noron, mạng Noron học sâu, bản chất toán học của nó. Trong cuốn sách còn giới thiệu các loại mô hình mạng Noron học sâu tiêu biểu được công bố trong các nghiên cứu trong thời gian gần đây. Một số ứng dụng cụ thể đóng vai trò rất quan trọng của mạng Noron trong các lĩnh vực cũng được giới thiệu và phân tích. Các lĩnh vực này là nền tảng của các hệ thống thông minh bao gồm thị giác máy tính và xử lý ngôn ngữ.

Cuốn sách này được trình bày một cách đơn giản và thiết thực nhất giúp cho người đọc dễ dàng nắm bắt vấn đề được đề cập. Với các thuật toán và kèm theo sau đó là chương trình mẫu, người đọc dễ dàng làm thực nghiệm trên các tập dữ liệu đã được giới thiệu để kiểm chứng lại cơ sở lý thuyết và hiểu rõ vấn đề hơn.

Để dễ dàng tiếp cận lĩnh vực được trình bày trong cuốn sách, người đọc cũng cần trang bị trước những kiến thức toán học nền tảng như toán giải tích, đại số tuyến tính, lý thuyết toán tối ưu và kỹ năng lập trình. Xuyên suốt trong quyển sách này, các ví dụ được phân tích và lập trình mẫu trên ngôn ngữ lập trình Python. Đây là một trong những ngôn ngữ có số lượng thư viện và người sử dụng lớn nhất hiện nay về lĩnh vực mạng Noron học sâu. Một số gói thư viện như Keras, MXNET cũng được giới thiệu để người đọc có thể tận dụng những hàm có sẵn trong việc lập trình thực hiện một ứng dụng nào đó.

Nội dung quyển sách chia thành các chương đi từ giới thiệu các khái niệm cơ bản cho đến các ứng dụng cụ thể. Trong đó, phần đầu quyển sách là giới thiệu mạng Noron, mạng Noron học sâu và phân tích các cấu hình

mạng học sâu tiêu biểu. Phần giữa quyển sách cung cấp cho người đọc cơ sở toán học của mạng Noron. Phần sau cùng là các ứng dụng của mạng học sâu trong thị giác máy tính và xử lý ngôn ngữ.

Sơ lược về các chương

Chương 1: Giới thiệu khái niệm cơ bản về mạng Noron. Nội dung bao gồm giới thiệu khái niệm mạng nhiều lớp, thuật toán lan truyền ngược, khởi tạo và cập nhật trọng số cũng như học các đặc trưng phân cấp trong mạng Noron học sâu.

Chương 2: Giới thiệu các cấu hình tiêu biểu của mạng Noron học sâu. Nội dung bao gồm phân tích chi tiết các loại mạng tích chập CNN, LeNet và MiniVGGNet.

Chương 3: Trình bày cơ sở toán học của lý thuyết mạng Noron. Nội dung bao gồm học theo thông số, lý thuyết tối ưu sai số huấn luyện, hiện tượng chưa khớp và quá khớp cũng như vấn đề kiểm tra mô hình mạng.

Chương 4: Trình bày ứng dụng của mạng Noron học sâu trong lĩnh vực thị giác máy tính. Nội dung bao gồm khái niệm cơ bản về hình ảnh, các tập dữ liệu chuẩn dùng để phân tích. Các bài toán phân loại ảnh và nhận dạng đối tượng trong ảnh cũng được giới thiệu trong chương này.

Chương 5: Trình bày ứng dụng của mạng Noron hồi quy trong lĩnh vực xử lý ngôn ngữ. Nội dung bao gồm giới thiệu cơ bản về mạng Noron hồi quy, chuyển lời nói thành văn bản và ngược lại. Phần cuối trong chương này là phát triển thành ứng dụng Chatbot.

Cuối quyển sách là danh mục các tài liệu tham khảo, người đọc có thể dùng để tham chiếu đến các nghiên cứu khác được trình bày một cách chi tiết hơn.

Trong quá trình biên soạn sẽ không tránh những thiếu sót, tác giả rất mong nhận được những ý kiến đóng góp từ bạn đọc thông qua hộp thư điện tử halm@hcmute.edu.vn để lần tái bản sau được tốt hơn.

Trân trọng,

Lê Mỹ Hà

Chương 1:

GIỚI THIỆU VỀ MẠNG NƠ-RON VÀ HỌC SÂU

Trong chương này sẽ trình bày các nguyên tắc cơ bản của mạng nơ-ron bắt đầu với nghiên cứu về các mạng nơ-ron nhân tạo và cách mà chúng mô phỏng lại mạng nơ-ron sinh học trong cơ thể con người. Từ đó, xem xét thuật toán Perceptron cổ điển và vai trò của nó trong lịch sử mạng nơ-ron.

Trên cơ sở mạng Perceptron chúng ta nghiên cứu về thuật toán lan truyền ngược là nền tảng huấn luyện nơ-ron hiện đại. Nếu không có lan truyền ngược thì không thể huấn luyện mạng một cách hiệu quả. Tiếp theo là triển khai lan truyền ngược bằng Python, để đảm bảo hiểu rõ thuật toán quan trọng này.

Tất nhiên, các thư viện mạng nơ-ron hiện đại như Keras đã có sẵn thuật toán lan truyền ngược (với độ tối ưu hóa cao). Việc thực hiện lan truyền ngược bằng tay mỗi khi muốn huấn luyện một mạng nơ-ron giống như việc lập trình một danh sách liên kết hoặc cấu trúc dữ liệu dạng bảng nhỏ trước khi làm việc với một vấn đề lập trình tổng quát - không chỉ không thực tế, mà còn lãng phí thời gian và nguồn lực. Để hợp lý hóa quy trình, sẽ trình bày cách tạo ra mạng nơ-ron tiêu chuẩn bằng cách sử dụng thư viện Keras.

Ngoài, chương này sẽ thảo luận về bốn thành phần mà người đọc sẽ cần khi xây dựng một mạng nơ-ron bất kỳ.

Phần còn lại của chương này sẽ xem xét lịch sử ngắn gọn của học sâu, xem xét về những gì tạo ra một mạng nơ-ron “sâu” và khám phá khái niệm “học phân cấp” và cách thức hoạt động của nó để làm cho học sâu trở thành một trong những thành công lớn trong lĩnh vực máy học và thị giác máy tính.

1.1. NGUYÊN TẮC CƠ BẢN CỦA MẠNG NƠ-RON

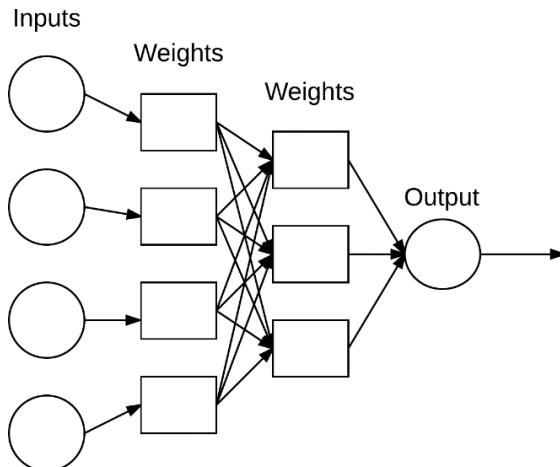
1.1.1. Khái niệm cơ bản về mạng nơ-ron

Trước khi có thể làm việc với mạng nơ-ron tích chập, trước tiên cần hiểu những nguyên tắc cơ bản của một mạng nơ-ron. Trong phần này, hãy xem xét:

- Mạng nơ-ron nhân tạo và sự liên hệ của chúng với sinh học.
- Thuật toán Perceptron cổ điển.

- Thuật toán lan truyền ngược và cách nó được sử dụng để huấn luyện các mạng nơ-ron nhiều lớp một cách hiệu quả.
- Cách huấn luyện mạng nơ-ron bằng thư viện Keras.

Khi hoàn thành chương này, người học sẽ có một sự hiểu biết sâu sắc về các mạng nơ-ron và có cơ sở để tìm hiểu sâu hơn mạng nơ-ron tích chập.



Hình 1.1: Một kiến trúc mạng nơ-ron đơn giản. Đầu vào được đưa vào mạng. Mỗi kết nối mang một tín hiệu thông qua hai lớp ẩn trong mạng. Một hàm cuối cùng tính toán lớp nhãn đầu ra.

1.1.2. Giới thiệu về mạng nơ-ron

Mạng nơ-ron là các khối xây dựng nền hệ thống học sâu. Để hiểu về học sâu, cần bắt đầu bằng việc xem xét các kiến thức cơ bản về mạng nơ-ron, bao gồm kiến trúc, loại nút và thuật toán cho việc huấn luyện mạng.

Trong phần này, hãy bắt đầu với một cái nhìn tổng quan từ lớp cao về mạng nơ-ron và ý tưởng đằng sau - bao gồm cả mối liên hệ giữa chúng với sinh học trong não người. Từ đó, xem xét về loại các kiến trúc mạng nơ-ron phổ biến nhất. Thảo luận ngắn gọn về khái niệm huấn luyện mạng nơ-ron và sau đó xem xét mối liên hệ giữa các thuật toán sử dụng để huấn luyện mạng nơ-ron như thế nào.

1.1.2.1. Mạng nơ-ron là gì ?

Nhiều nhiệm vụ liên quan đến trí thông minh nhân tạo, nhận dạng mẫu và phát hiện đối tượng là cực kỳ khó trong tự động hóa, nhưng dường như được thực hiện dễ dàng và rất tự nhiên bởi động vật và trẻ nhỏ. Ví dụ, làm thế nào để con chó của một gia đình nhận ra chủ của nó

mà không phải là một người hoàn toàn xa lạ ? Làm thế nào để một em bé học cách nhận ra sự khác biệt giữa xe buýt đưa đón của trường học và xe buýt công cộng? Và làm thế nào để bộ não của chúng ta tiềm thức thực hiện các nhiệm vụ nhận dạng mẫu phức tạp mỗi ngày mà không cần chúng ta nhận ra?

Câu trả lời nằm trong chính cơ thể. Mỗi chúng ta đều có một mạng nơ-ron sinh học ngoài đời thực được kết nối với hệ thống nơ-ron thần kinh, mạng này được tạo thành từ một số lượng lớn các tế bào nơ-ron liên kết với nhau (tế bào nơ-ron).

Từ «nơ-ron» là tế bào nơ-ron, và mạng có nghĩa là một cấu trúc giống như dạng đồ thị; Vì vậy, một mạng nơ-ron nhân tạo được hiểu là một hệ thống tính toán có gắng bắt chước (hoặc ít nhất là được lấy ý tưởng từ đó) các kết nối nơ-ron trong hệ thống nơ-ron. Mạng nơ-ron nhân tạo cũng được gọi là mạng nơ-ron, hay hệ thống nơ-ron nhân tạo. Người ta thường viết tắt mạng nơ-ron nhân tạo là ANN hay NN - sử dụng cả hai từ viết tắt trong suốt phần còn lại cuốn sách.

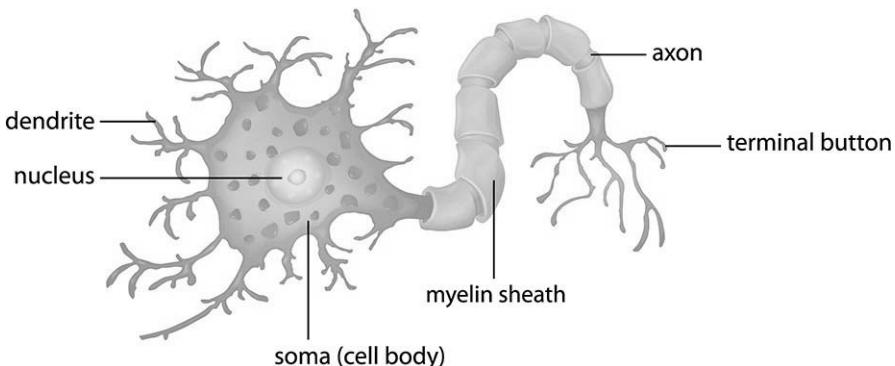
Để một hệ thống được coi là NN, nó phải chứa cấu trúc đồ thị có hướng, có nhãn trong đó mỗi nút trong đồ thị thực hiện một số tính toán đơn giản. Từ lý thuyết đồ thị, biết rằng một đồ thị có hướng bao gồm một tập hợp các nút (tức là các đỉnh) và một tập hợp các kết nối (tức là, các cạnh) liên kết các cặp nút với nhau. Trong Hình 1.1 có thể thấy một ví dụ về đồ thị NN như vậy.

Mỗi nút thực hiện một tính toán đơn giản. Mỗi kết nối sau đó mang một tín hiệu (nghĩa là đầu ra tính toán) từ nút này sang nút khác, được gắn nhãn bằng trọng số cho biết mức độ tín hiệu được khuếch đại hoặc giảm đi. Một số kết nối có trọng số lớn, dương giúp khuếch đại tín hiệu, cho thấy tín hiệu rất quan trọng khi thực hiện phân loại. Những số khác có trọng số âm, làm giảm cường độ tín hiệu, do đó có thể rằng đầu ra của nút ít quan trọng hơn trong phép phân loại cuối cùng. Gọi một hệ thống như vậy là mạng nơ-ron nhân tạo nếu nó bao gồm một cấu trúc đồ thị (như trong Hình 1.1) với các trọng số kết nối có thể điều chỉnh bằng thuật toán huấn luyện.

1.1.2.2. Liên hệ sinh học

Bộ não người gồm khoảng 10 tỷ tế bào nơ-ron, mỗi tế bào được kết nối với khoảng 10.000 tế bào nơ-ron khác. Thân tế bào nơ-ron được gọi là soma, trong đó đầu vào (đuôi gai) và đầu ra (sợi trực) kết nối soma với soma khác (Hình 1.2).

Human Neuron Anatomy



Hình 1.2: Cấu trúc một nơron sinh học. Các tế bào nơ-ron được kết nối với các tế bào nơ-ron khác thông qua đuôi gai và enuron

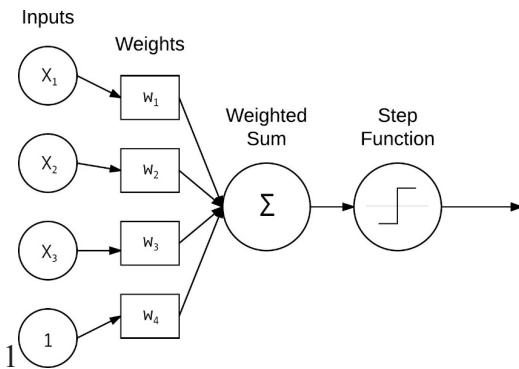
Mỗi tế bào nơ-ron nhận tín hiệu điện hóa từ các tế bào nơ-ron khác tại đuôi gai chúng. Nếu các đầu vào điện hóa này đủ mạnh để kích hoạt tế bào nơ-ron, thì tế bào nơ-ron được kích hoạt sẽ truyền tín hiệu dọc theo sợi trực nó, dọc theo sợi nhánh các tế bào nơ-ron khác. Những tế bào nơ-ron kèm theo cũng có thể kích hoạt, do đó quá trình truyền tín hiệu đi cứ tiếp diễn.

Điểm mấu chốt ở đây là việc kích hoạt nơ-ron là một biểu thức nhị phân - nơ-ron hoặc kích hoạt hoặc không. Không có các cấp độ khác nhau tại các điểm khác nhau. Nói một cách đơn giản, một tế bào nơ-ron chỉ phát tín hiệu nếu tổng tín hiệu nhận được tại soma vượt quá một ngưỡng nhất định.

Tuy nhiên, hãy nhớ rằng ANN được lấy ý tưởng từ những gì ta biết về bộ não và cách thức hoạt động nó. Mục tiêu học sâu không phải là bắt chước cách bộ não hoạt động, mà là lấy những phần hiểu biết về mạng này để tạo ra một mạng tương tự. Và cuối cùng, chúng ta không tìm hiểu về khoa học thần kinh và các chức năng sâu hơn của bộ não để có thể mô hình chính xác cách thức hoạt động của bộ não mà là lấy cảm hứng và đi tiếp.

1.1.2.3. Mô hình mạng nơ-ron nhân tạo

Hãy bắt đầu bằng cách xem xét một NN cơ bản thực hiện việc tính tổng các trọng số đầu vào trong Hình 1.3. Các giá trị x_1 , x_2 và x_3 là các đầu vào cho NN và tương ứng với một hàng đơn (tức là, điểm dữ liệu) từ ma trận thiết kế. Giá trị không đổi 1 là độ lệch được giả định là được nhúng vào ma trận thiết kế - có thể coi các đầu vào này là các vectơ đặc trưng đầu vào của NN.



Hình 1.3: Một NN đơn giản lấy tổng trọng số đầu vào x và trọng số w. Tổng trọng số này sau đó được chuyển qua hàm kích hoạt để xác định xem té bào nơ-ron có kích hoạt không.

Trong thực tế, các đầu vào này có thể là các vectơ được sử dụng để định lượng nội dung ảnh theo một cách có hệ thống, được xác định trước (ví dụ : biểu đồ màu, biểu đồ các lớp định hướng [1], mẫu nhị phân cục bộ [2], v.v.). Trong phương diện học sâu, những đầu vào này là cường độ điểm ảnh thô.

Mỗi x được kết nối với một nơ-ron thông qua một vectơ trọng số W bao gồm w_1, w_2, \dots, w_n , có nghĩa là với mỗi đầu vào x, cũng có trọng số liên quan w.

Cuối cùng, nút đầu ra ở bên phải Hình 1.3 lấy tổng trọng số, áp dụng hàm kích hoạt f (được sử dụng để xác định xem nơ-ron có kích hoạt hay không) và đưa ra một giá trị. Biểu diễn đầu ra một cách toán học, thường gấp ba dạng sau :

$$f(w_1x_1 + w_2x_2 + \dots + w_nx_n) \quad (1.1)$$

$$f\left(\sum_{i=1}^n w_i x_i\right) \quad (1.2)$$

Hoặc đơn giản f(net), trong đó

$$net = \sum_{i=1}^n w_i x_i \quad (1.3)$$

Bất kể giá trị đầu ra được biểu diễn như thế nào cũng được tính đơn giản là lấy tổng đầu vào các trọng số và sau đó áp dụng hàm kích hoạt f.

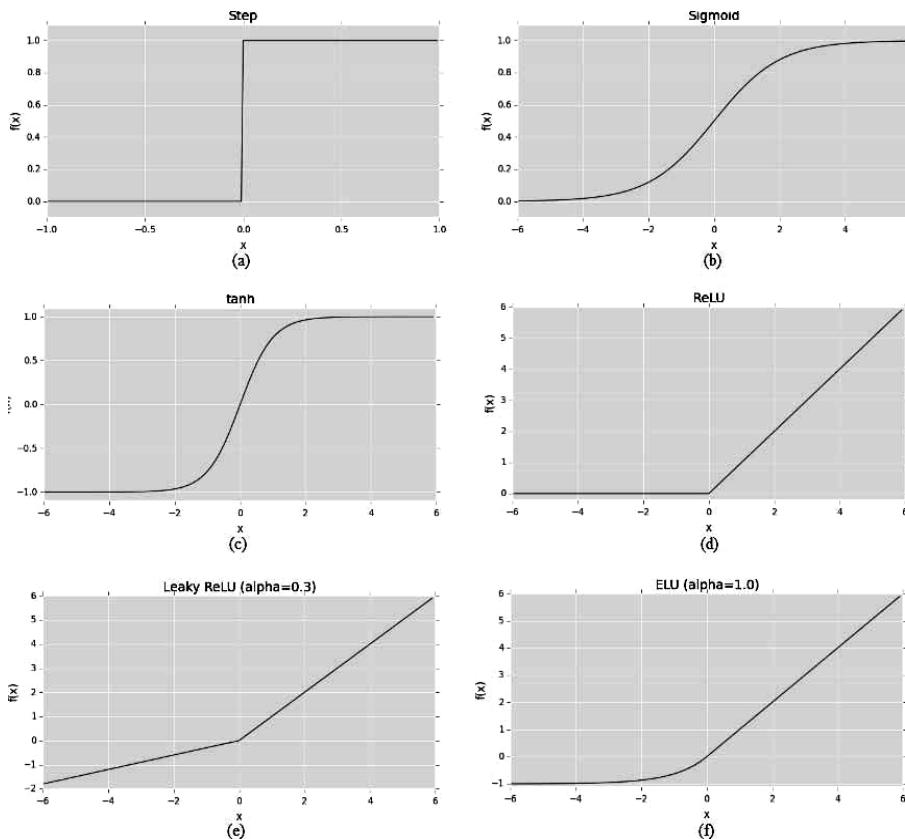
1.1.2.4. Hàm kích hoạt

Hàm kích hoạt đơn giản nhất là hàm bước, được sử dụng bởi thuật toán Perceptron (trình bày trong phần tiếp theo).

$$f(net) = \begin{cases} 1, & net > 0 \\ 0, & \text{otherwise} \end{cases} \quad (1.4)$$

Như có thể thấy từ phương trình trên, đây là một hàm ngưỡng đơn giản. Nếu có tổng trọng số $\sum_{i=1}^n w_i x_i > 0$, trả về 1, ngoài ra thì trả về 0.

Vẽ các giá trị đầu vào đọc theo trục x và đầu ra f (net) đọc theo trục y có thể thấy hàm kích hoạt này hoạt động như thế nào và tại sao (Hình 1.4, trên cùng bên trái). Đầu ra f luôn bằng 0 khi net nhỏ hơn hoặc bằng 0. Nếu net lớn hơn 0, thì f hãy trả về 1. Do đó, hàm này trông giống như một bậc thang.



Hình 1.4: Các loại hàm tác động trong mạng No-ron.

- Hàm bước
- Hàm kích hoạt Sigmoid.
- Tiếp tuyến Hyper-bolic.
- Hàm kích hoạt ReLU (hàm kích hoạt được sử dụng nhiều nhất cho các mạng no-ron sâu).
- Hàm Leaky ReLU (biến thể ReLU cho phép các giá trị âm).
- ELU (một biến thể khác ELU thường có thể hoạt động tốt hơn Leaky ReLU).

Tuy rất trực quan và dễ sử dụng nhưng hàm bước không có độ khác biệt. Điều này có thể dẫn đến các vấn đề khi áp dụng hàm suy giảm độ dốc (gradient descent) và huấn luyện mạng.

Thay vào đó, một hàm kích hoạt phẳng biến hơn được sử dụng trong xuyên suốt tài liệu NN là hàm sigmoid (Hình 1.4, trên cùng bên phải), theo phương trình:

$$t = \sum w_i x_i, s(t) = 1 / (1 + e^{-t}) \quad (1.5)$$

Hàm sigmoid là một lựa chọn tốt hơn cho việc huấn luyện mạng so với hàm bước đơn giản vì nó:

1. Liên tục và khác biệt ở mọi nơi.
2. Đối xứng quanh trục y.
3. Tiệm cận tiếp cận các giá trị bao hòa của nó.

Ưu điểm chính ở đây là sự trơn tru của hàm sigmoid giúp dễ dàng đưa ra các thuật toán. Tuy nhiên, có hai vấn đề lớn với hàm sigmoid :

1. Đầu ra sigmoid không bằng 0.
2. Các nơ-ron bao hòa về cơ bản sẽ làm gradient sai lệch, vì delta gradient sẽ cực kỳ nhỏ.

Tiếp tuyến hyperbolic, hoặc tanh (có hình dạng tương tự sigmoid) cũng được sử dụng nhiều làm hàm kích hoạt cho đến cuối những năm 1990 (Hình 1.4, giữa bên trái) : Phương trình tanh như sau :

$$f(z) = \tanh(z) = (e^z - e^{-z}) / (e^z + e^{-z}) \quad (1.6)$$

Hàm tanh không có tâm, nhưng độ dốc vẫn bị mất đi khi các tế bào nơ-ron trở nên bao hòa.

Hiện nay có nhiều lựa chọn tốt hơn cho các hàm kích hoạt hơn hàm sigmoid và tanh. Cụ thể, Hahnloser và cộng sự trong bài báo năm 2000, lựa chọn kỹ thuật số và khuếch đại tương tự cùng tồn tại trong một mạch bán dẫn lấy ý tưởng từ bộ não [3], đã giới thiệu đơn vị tuyến tính chỉnh lưu (ReLU), được định nghĩa là :

$$f(x) = \max(0, x) \quad (1.7)$$

ReLU cũng được gọi là hàm dốc do cách chúng được biểu diễn (Hình 1.4, giữa bên phải). Lưu ý là hàm bằng 0 đối với đầu vào âm nhưng sau đó tăng tuyến tính cho giá trị dương. Hàm ReLU không bao hòa và cũng cực kỳ hiệu quả về mặt tính toán.

Theo kinh nghiệm, hàm kích hoạt ReLU có xu hướng vượt trội hơn cả hàm sigmoid và tanh trong gần như tất cả các ứng dụng. Kết hợp nghiên cứu của Hahnloser và Seung trong bài báo tiếp theo của họ năm 2003, Permitted and Forbidden Sets in Symmetric Threshold-Dòngar Networks [4], người ta thấy rằng hàm kích hoạt ReLU có động lực sinh học mạnh hơn các họ hàm kích hoạt trước đó, bao gồm cả về các minh chứng toán học.

Kể từ năm 2015, ReLU là hàm kích hoạt phổ biến nhất được sử dụng trong học sâu [5]. Tuy nhiên, một vấn đề phát sinh khi có giá trị bằng 0 - độ dốc không thể được thực hiện.

Một biến thể ReLUs, được gọi là Leaky ReLUs [6] cho phép độ dốc nhỏ, khác 0 khi hàm không hoạt động :

$$f(\text{net}) = \begin{cases} \text{net}, & \text{net} >= 0 \\ \alpha \times 0, & \text{otherwise} \end{cases} \quad (1.8)$$

Vẽ đồ thị hàm này trong Hình 1.4 (phía dưới bên trái), có thể thấy rằng hàm này thực sự được phép mang một giá trị âm, không giống như các ReLU truyền thống mà kẹp đầu ra hàm ở mức 0.

ReLUs tham số, hoặc PReLU viết tắt [7], xây dựng trên ReLUs Leaky và cho phép tham số α được học trên cơ sở kích hoạt bằng cách kích hoạt, nghĩa là mỗi nút trong mạng có thể học một hệ số rò rỉ khác nhau tách biệt với các nút khác.

Cuối cùng, Exponential Dòngar Units (ELU) được giới thiệu bởi Clevert et al. trong bài viết năm 2015, học sâu nhanh và chính xác theo hàm mũ đơn vị tuyến tính (ELU) [8]:

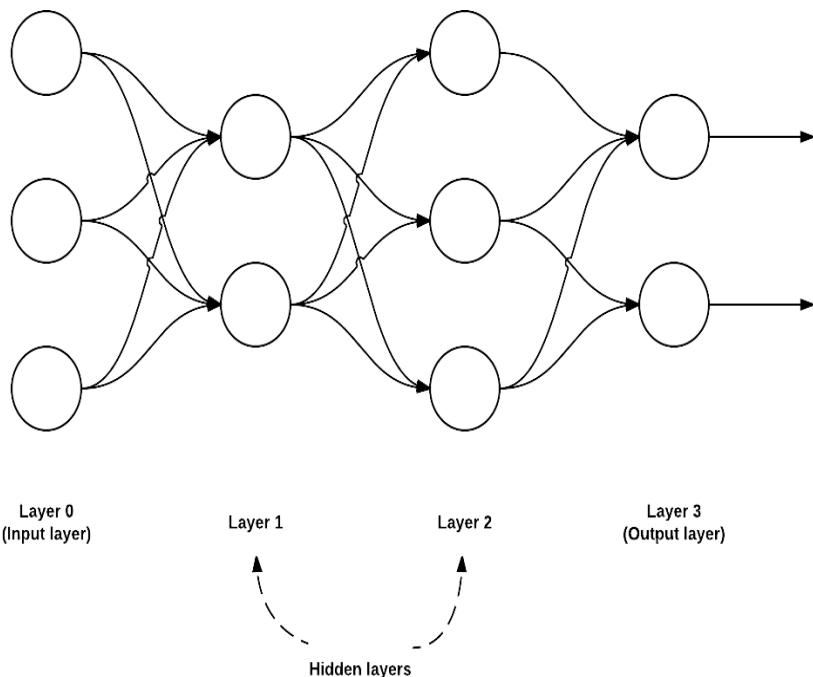
$$f(\text{net}) = \begin{cases} \text{net}, & \text{net} >= 0 \\ \alpha \times (\exp(\text{net}) - 1), & \text{otherwise} \end{cases} \quad (1.9)$$

Giá trị α là hằng số và được đặt khi cấu trúc mạng được khởi tạo - điều này không giống với PReLU nơi mà α được học. Một giá trị điển hình cho α là $\alpha = 1.0$. Hình 1.4 (dưới cùng bên phải) biểu diễn hàm kích hoạt ELU.

ELU thường có độ chính xác phân loại cao hơn ReLU. ELU hiếm khi hoạt động kém hơn hàm ReLU tiêu chuẩn.

1.1.2.5. Kiến trúc mạng lan truyền thẳng

Trong khi có rất nhiều, rất nhiều kiến trúc mạng nơ-ron khác nhau, thì kiến trúc phổ biến nhất là mạng lan truyền thẳng, như được trình bày trong Hình 1.5.



Hình 1.5: Một ví dụ về mạng nơ-ron lan truyền thẳng có 3 nút đầu vào, một lớp ẩn có 2, nút một lớp ẩn thứ hai có 3 nút và lớp đầu ra cuối cùng có 2 nút.

Trong kiểu kiến trúc này, kết nối giữa các nút chỉ được phép đi từ các nút trong lớp i đến các nút trong lớp $i + 1$ (do đó, có thuật ngữ này, lan truyền thẳng). Không có kết nối ngược hoặc liên lớp cho phép. Khi các mạng lan truyền thẳng bao gồm các kết nối phản hồi (kết nối đầu ra phản hồi vào đầu vào), chúng được gọi là mạng nơ-ron hồi quy,

Cuốn sách này tập trung vào các mạng nơ-ron lan truyền thẳng vì chúng là nền tảng học sâu hiện đại áp dụng cho thị giác máy tính. Tìm hiểu trong chương 2, mạng nơ-ron tích chập là một trường hợp đặc biệt của mạng nơ-ron truyền thẳng.

Để mô tả một mạng lan truyền thẳng, thường sử dụng một chuỗi các số nguyên để mô tả nhanh chóng và chính xác số lượng nút trong mỗi lớp. Ví dụ, mạng trong Hình 1.5 ở trên là mạng lan truyền thẳng 3-2-3-2:

Lớp 0 chứa 3 đầu vào, giá trị x_i . Đây có thể là cường độ điểm ảnh thô ảnh hoặc vectơ đặc trưng được trích xuất từ ảnh.

Lớp 1 và 2 là các lớp ẩn tương ứng chứa 2 và 3 nút.

Lớp 3 là lớp đầu ra hoặc lớp có thể nhìn thấy - đó là nơi phân loại đầu

ra tổng thể từ mạng. Lớp đầu ra thường có nhiều nút như tên nhãn của lớp; một nút biểu diễn cho một đầu ra. Ví dụ: nếu xây dựng NN để phân loại các chữ số viết tay, lớp đầu ra bao gồm 10 nút, mỗi nút cho mỗi chữ số 0-9.

1.1.2.6. Huấn luyện mạng no-ron

Huấn luyện no-ron là phương pháp sửa đổi các trọng số và kết nối giữa các nút trong mạng. Về mặt sinh học, định nghĩa việc huấn luyện theo nguyên tắc Hebb như sau:

Khi một sợi trực tê bào A đủ gần để kích thích tê bào B và lặp lại hoặc liên tục diễn ra trong quá trình kích hoạt nó, thì quá trình tăng trưởng hoặc thay đổi trao đổi chất diễn ra trong một hoặc cả hai tê bào như tê bào A sẽ kích hoạt B tăng lên - Donald Hebb [9]

Về mặt ANN, nguyên tắc này mang ý nghĩa là cần phải tăng cường độ kết nối giữa các nút có đầu ra tương tự khi được biểu diễn bởi cùng một đầu vào, gọi đây là học tương quan bởi vì sức mạnh kết nối giữa các nơ-ron cuối cùng đại diện cho mối tương quan giữa các đầu ra.

1.1.2.7. Mạng nơ-ron được sử dụng để làm gì?

Mạng nơ-ron có thể được sử dụng trong cả nhiệm vụ học có giám sát, không giám sát và bán giám sát, cung cấp kiến trúc phù hợp để sử dụng. Một đánh giá đầy đủ về NN nằm ngoài phạm vi cuốn sách này (vui lòng xem ấn phẩm của Schmidhuber [10] để tham khảo thêm về các mạng nhân tạo sâu, Mehrotra [11] nghiên cứu phương pháp cổ điển); Tuy nhiên, các ứng dụng phổ biến NN là phân loại, hồi quy, phân cụm, lượng tử hóa vectơ, liên kết mẫu và xấp xỉ hàm.

Trong thực tế, đối với hầu hết mọi khía cạnh học máy, NN đã được áp dụng dưới hình thức này hay hình thức khác. Trong nội dung cuốn sách này, sử dụng NN để phân loại ảnh và thị giác máy tính.

1.1.2.8. Tóm tắt một số vấn đề cơ bản về mạng nơ-ron

Phần này sẽ xem xét các khái niệm cơ bản về mạng nơ-ron nhân tạo (ANN, hoặc đơn giản là NN). Bắt đầu bằng cách kiểm tra động lực sinh học đằng sau ANN, sau đó tìm hiểu để có thể xác định phương trình toán học của một hàm để mô phỏng kích hoạt của một tê bào nơ-ron (tức là, hàm kích hoạt).

Dựa trên mô hình một nơ-ron, có thể định nghĩa kiến trúc mạng bao gồm (ở mức tối thiểu), một lớp đầu vào và một lớp đầu ra. Một số kiến trúc mạng có thể bao gồm nhiều lớp ẩn giữa các lớp đầu vào và đầu ra. Cuối

cùng, mỗi lớp có thể có một hoặc nhiều nút. Các nút trong lớp đầu vào không chứa hàm kích hoạt (chúng là kiểu mà ở đó, cường độ pixel riêng lẻ của ảnh được nhập vào); Tuy nhiên, các nút trong cả hai lớp đầu ra và ẩn đều chứa hàm kích hoạt.

Ngoài ra, cũng xem xét ba hàm kích hoạt phổ biến: sigmoid, tanh và ReLU (và các biến thể của nó).

Theo truyền thống, các hàm sigmoid và tanh đã được sử dụng để huấn luyện các mạng; Tuy nhiên, kể từ bài báo của Hahnloser et al. từ năm 2000 [3], hàm ReLU đã được sử dụng thường xuyên hơn.

Trong năm 2015, ReLU cho đến nay là hàm kích hoạt phổ biến nhất được sử dụng trong các kiến trúc học sâu [7]. Dựa trên sự thành công của hàm ReLU. Hàm Leaky ReLUs, một biến thể của ReLU đã tìm cách cải thiện hiệu suất mạng bằng cách cho phép hàm đảm nhận giá trị âm. Nhóm hàm Leaky ReLU bao gồm biến thể Leaky ReLU tiêu chuẩn, PReLU và ELU.

Cuối cùng, điều quan trọng cần lưu ý là mặc dù đang tập trung vào việc học trong phân loại ảnh, tuy nhiên mạng nơ-ron cũng được sử dụng tại một số mô hình trong hầu hết các lĩnh vực học máy.

Trong phần tiếp theo, thảo luận về thuật toán Perceptron cổ điển, một trong những ANN đầu tiên được tạo ra.

1.1.3. Thuật toán Perceptron

Được giới thiệu lần đầu tiên bởi Rosenblatt vào năm 1958, Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain” [12] được coi là thuật toán ANN lâu đời nhất và đơn giản nhất. Sau án phẩm này, các kỹ thuật dựa trên Perceptron nở rộ trong cộng đồng mạng nơ-ron. Bài báo này đã có đóng góp rất lớn cho sự phổ biến và sự hữu ích của mạng nơ-ron ngày nay.

Sau đó - năm 1969, một “thời đại AI đóng băng” đã xuất hiện trên cộng đồng máy học gần như đóng băng của mạng nơ-ron. Minsky và Papert đã xuất bản, Perceptrons: an introduction to computational geometry” [13], một cuốn sách làm định hình nghiên cứu về các mạng nơ-ron trong gần một thập kỷ, mặc dù có nhiều sự tranh cãi về cuốn sách này [14], nhưng các tác giả đã chứng minh thành công rằng một lớp Perceptron duy nhất không thể tách các điểm dữ liệu phi tuyến. Vì hầu hết các bộ dữ liệu trong thực tế đều có dạng phân tách phi tuyến, điều này dường

như mang Perceptron cùng và các nghiên cứu khác của mạng nơ-ron có thể đi đến ngõ cụt.

Với án phẩm của Minsky và Papert thì những hứa hẹn bị phá vỡ của ngành công nghiệp mạng nơ-ron bị phá vỡ, và sự quan tâm đến các mạng nơ-ron giảm đáng kể. Sự bắt đầu khám phá các mạng sâu hơn (đôi khi được gọi là các tri giác đa lớp) cùng với thuật toán backpropagation (Werbos [15] và Rumelhart [16]) đã khiến sự đóng băng trong nghiên cứu AI đã kết thúc và nghiên cứu mạng nơ-ron bắt đầu nóng trở lại.

Như những gì đã trình bày, Perceptron vẫn là một thuật toán rất quan trọng để hiểu vì nó tạo tiền đề cho các mạng nhiều lớp tiên tiến hơn. Đầu tiên, cần nghiên cứu về kiến trúc Perceptron và giải thích việc tạo ra một bộ huấn luyện (được gọi là quy tắc delta) được sử dụng để huấn luyện Perceptron, cũng xem xét các tiêu chí kết thúc huấn luyện mạng (tức là khi Perceptron nên ngừng huấn luyện). Sau đó, hãy triển khai thuật toán Perceptron trong Python thuận túy, sử dụng nó để nghiên cứu và kiểm tra xem mạng có thể học các bộ dữ liệu phân tách phi tuyến hay không?

1.1.3.1. Bộ dữ liệu AND, OR và XOR

Trước khi nghiên cứu về Perceptron, trước tiên, hãy thảo luận về các Toán tử bit, bao gồm AND, OR và XOR (ex-OR). Cần tham gia một khóa học khoa học máy tính ở cấp độ giới thiệu trước khi làm quen với các hàm bitwise.

Toán tử bit và bộ dữ liệu bitwise có thể cho bởi hai bit đầu vào và tạo ra bit đầu ra sau khi áp dụng toán tử. Cho hai bit đầu vào, mỗi bit có khả năng nhận giá trị 0 hoặc 1, có bốn kết hợp có thể có hai bit này - Bảng 1.1 cung cấp các giá trị đầu vào và đầu ra có thể có cho AND, OR và XOR:

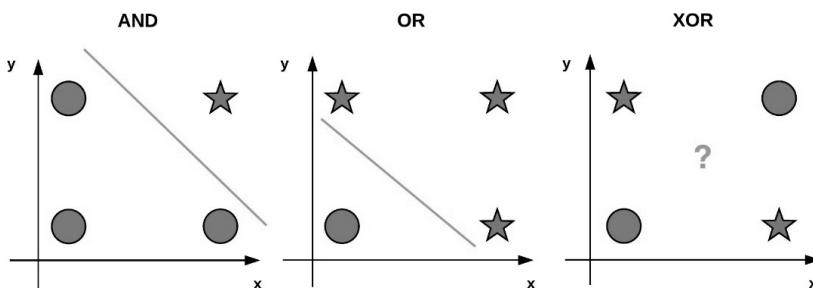
Như có thể thấy ở bên trái, AND logic là True khi và chỉ khi cả hai giá trị đầu vào là 1. Nếu một trong hai giá trị đầu vào là 0, AND trả về 0. Do đó, chỉ có một kết hợp, $x_0 = 1$ và $x_1 = 1$ khi đầu ra AND là True.

Ở giữa, có toán tử OR là True khi chỉ một trong các giá trị đầu vào là 1. Do đó, có ba kết hợp có thể có hai bit x_0 và x_1 tạo ra giá trị $y = 1$. Cuối cùng, bên phải biểu diễn Toán tử XOR là True khi và chỉ khi một nếu đầu vào là 1 nhưng không phải cả hai. Trong khi OR có ba tình huống có thể xảy ra trong đó $y = 1$, XOR chỉ có hai.

Bảng 1.1: Trái: Tập dữ liệu bitwise AND. Cho hai đầu vào, đầu ra chỉ là 1 nếu cả hai đầu vào là 1. Giữa: Tập dữ liệu bitwise OR. Cho hai đầu vào, đầu ra là 1 nếu một trong hai đầu vào là 1. Phải: Tập dữ liệu XOR (e (X) clusive OR). Cho hai đầu vào, đầu ra 1 khi và chỉ khi một đầu vào là 1, nhưng không phải cả hai.

x_0	x_1	$x_0 \& x_1$	x_0	x_1	$x_0 x_1$	x_0	x_1	$x_0 \wedge x_1$
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

Thường sử dụng các bộ dữ liệu bitwise đơn giản này để kiểm tra và gỡ lỗi các thuật toán học máy. Nếu vẽ đồ thị và trực quan hóa các giá trị AND, OR và XOR (với các vòng tròn màu đỏ là đầu ra 0 và các sao màu xanh là đầu ra) trong Hình 1.6, ta sẽ có một mẫu thú vị:



Hình 1.6: Cả hai bộ dữ liệu bit AND và OR đều có thể phân tách tuyến tính, nghĩa là có thể vẽ một dòng duy nhất (màu xanh lá cây) ngăn cách hai lớp. Tuy nhiên, đối với XOR, không thể vẽ một dòng phân tách hai lớp - do đó đây là một bộ dữ liệu phân tách phi tuyến.

Cả AND và OR đều có thể phân tách tuyến tính - có thể vẽ rõ ràng một đường phân tách các lớp 0 và 1 - điều này không đúng với XOR. Dễ dàng để nhận ra rằng không thể vẽ một đường thẳng ngăn cách rõ ràng hai lớp trong bài toán XOR. Do đó, XOR là một ví dụ về bộ dữ liệu có thể phân tách phi tuyến.

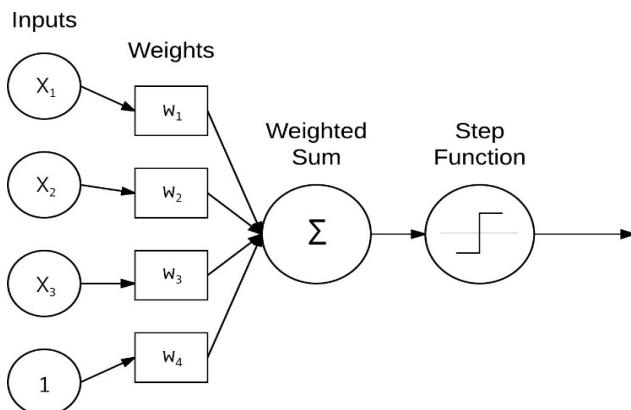
Lý tưởng nhất, chúng ta muốn các thuật toán học máy có thể tách các lớp phi tuyến vì hầu hết các bộ dữ liệu gặp phải trong thế giới thực là phi tuyến. Do đó, khi xây dựng, gỡ lỗi và đánh giá một thuật toán học máy nhất định, có thể sử dụng các giá trị bit x_0 và x_1 làm ma trận thiết kế và sau đó thử dự đoán các giá trị y tương ứng.

Không giống như quy trình tiêu chuẩn về việc chia dữ liệu thành các phần tách biệt giữa huấn luyện và kiểm tra, khi sử dụng bộ dữ liệu bitwise, chỉ cần huấn luyện và đánh giá mạng trên cùng một bộ dữ liệu. Mục tiêu ở đây chỉ đơn giản là xác định xem liệu thuật toán học có thể học được các mẫu trong dữ liệu hay không. Như đã tìm ra, thuật toán Perceptron có thể phân loại chính xác các hàm AND và OR nhưng không thể phân loại dữ liệu XOR.

1.1.3.2. Kiến trúc Perceptron

Rosenblatt [12] đã định nghĩa Perceptron là một hệ thống học bằng cách sử dụng các ví dụ được gắn nhãn (nghĩa là học có giám sát) các vectơ đặc trưng (hoặc cường độ pixel thô), ánh xạ các đầu vào này vào lớp nhãn đầu ra tương ứng của chúng.

Ở dạng đơn giản nhất, Perceptron chứa N nút đầu vào, một nút cho mỗi ngõ vào ma trận thiết kế, tiếp theo chỉ có một lớp đơn lẻ trong mạng (Hình 1.7).



Hình 1.7: Kiến trúc mạng Perceptron.

Có sự tồn tại các kết nối và trọng số tương ứng của chúng w_1, w_2, \dots, w_i từ x_i đầu vào đến nút đầu ra duy nhất trong mạng. Nút này lấy tổng số đầu vào có trọng số và áp dụng hàm bước để xác định lớp nhãn đầu ra. Perceptron xuất ra 0 hoặc 1 - 0 cho lớp 1 và 1 cho lớp 2; do đó, ở dạng ban đầu, Perceptron chỉ đơn giản là một phân loại nhị phân có hai lớp.

1.1.3.3. Quy trình huấn luyện Perceptron và quy tắc Delta

Huấn luyện một Perceptron là một hoạt động khá đơn giản. Mục tiêu là có được một tập các trọng số w phân loại chính xác từng trường hợp trong tập huấn luyện. Để huấn luyện Perceptron, lặp lại việc đưa dữ liệu huấn luyện vào nhiều lần. Mỗi lần nhìn thấy toàn bộ dữ liệu huấn luyện là một chu kỳ (epoch) đã trôi qua. Thường mất nhiều epoch cho đến khi một

vector trọng số có thể được học để được phân tách tuyến tính hai lớp dữ liệu.

Chương trình giả ngẫu nhiên cho thuật toán huấn luyện Perceptron có thể được xác định như sau:

Quá trình huấn luyện thực tế diễn ra trong các bước 2b và 2c (trình bày phần dưới đây). Đầu tiên, truyền vector đặc trưng x_j qua mạng, nhân chập với trọng số với trọng số w và thu được đầu ra y_j . Giá trị này sau đó được chuyển qua hàm bước trả về 1 nếu $x > 0$ và 0 ở các trường hợp còn lại.

Bây giờ cần cập nhật vector trọng số để bước tiếp theo có được bộ phân loại chính xác. Trong lần cập nhật này vector trọng số được xử lý theo quy tắc delta trong Bước 2c.

Biểu thức $(d_j - y_j)$ xác định xem phân loại đầu ra có đúng hay không. Nếu phân loại là chính xác, sau đó đạo hàm này sẽ bằng không. Mặt khác, đạo hàm này bằng dương hay âm sẽ thu được hướng mà trọng số được cập nhật (cuối cùng tiến gần hơn với phân loại chính xác). Sau đó, nhân $(d_j - y_j)$ với x_j , ta sẽ đến gần hơn với kết quả phân loại chính xác.

Các bước huấn luyện:

1. Khởi tạo vector trọng số với các giá trị ngẫu nhiên nhỏ

2. Cho đến khi Perceptron hội tụ

(a) Lặp lại từng vector đặc trưng x_j và nhãn thực của lớp d_i trong tập huấn luyện D

(b) Truyền x vào mạng, tính giá trị đầu ra : $y_j = f(w(t) \cdot x_j)$

(c) Cập nhật các trọng số w : $w_i(t+1) = w_i(t) + (d_j - y_j) x_j$, i cho tất cả các tính năng $0 \leq i \leq n$

Giá trị α là tốc độ học và kiểm soát mức độ lớn (hoặc nhỏ) mà một bước nhảy thực hiện. Giá trị này cần phải chọn chính xác. Giá trị α lớn sẽ khiến bước nhảy đi đúng hướng ; tuy nhiên, bước này có thể quá lớn và có thể vượt qua điểm tối ưu cục bộ / toàn cục.

Ngược lại, một giá trị α nhỏ cho phép thực hiện các bước nhảy nhỏ theo đúng hướng, đảm bảo không vượt qua mức tối thiểu địa phương / toàn phần ; Tuy nhiên, những bước nhảy nhỏ này sẽ khiến cần một khoảng thời gian lớn để việc huấn luyện hội tụ.

Cuối cùng, thêm vào vector trọng số trước thời điểm t , $w_j(t)$ để hoàn thành quá trình này.

1.1.3.4. Kết thúc huấn luyện Perceptron

Quá trình huấn luyện Perceptron được phép tiến hành cho đến khi tất cả các mẫu huấn luyện được phân loại chính xác hoặc đạt đến số lượng chu kỳ đặt trước. Quá trình huấn luyện sẽ kết thúc nếu α đủ nhỏ và dữ liệu huấn luyện có thể phân tách tuyến tính.

Điều gì sẽ xảy ra nếu dữ liệu không thể phân tách tuyến tính hoặc chọn α không chính xác? Huấn luyện sẽ thực hiện cho đến vô tận? Trong trường hợp này, ta thường đặt trước một số chu kỳ để vòng lặp dừng lại nếu số lượng phân loại sai không thay đổi trong một số lượng lớn các chu kỳ (chỉ ra rằng dữ liệu không thể phân tách tuyến tính). Để biết thêm chi tiết về thuật toán Perceptron, vui lòng tham khảo bài giảng của Andrew Ng, Stanford [17] hoặc các chương giới thiệu của Mehrota et al. [18].

1.1.3.5. Triển khai Perceptron trong Python

Chúng ta đã nghiên cứu về thuật toán Perceptron, tiếp theo sẽ thực hiện thuật toán thực tế trong Python. Tạo một tệp có tên perceptron.py trong gói pyimagesearch.nn - tệp này sẽ lưu trữ triển khai Perceptron thực tế:

```
--- pyimagesearch
|   |--- __init__.py
|   |--- nn
|   |   |--- __init__.py
|   |   |--- perceptron.py
```

Sau khi đã tạo tập tin, hãy mở nó lên và chèn đoạn chương trình sau:

```
1 # import the necessary packages
2 import numpy as np
3
4 class Perceptron:
5     def __init__(self, N, alpha=0.1):
6         # initialize the weight matrix and store the learning rate
7         self.W = np.random.randn(N + 1) / np.sqrt(N)
8         self.alpha = alpha
```

Dòng 5 định nghĩa hàm tạo cho lớp Perceptron, chấp nhận một tham số bắt buộc duy nhất sau là tham số tùy chọn thứ hai:

1. N: Số lượng cột trong vector đặc trưng đầu vào. Trong trường hợp sử dụng các bộ dữ liệu bitwise, hãy đặt N bằng 2 vì có hai đầu vào.

2. alpha: Tốc độ học cho thuật toán Perceptron. Hãy đặt giá trị này thành 0,01 theo mặc định. Các lựa chọn phổ biến về tốc độ học thường nằm trong khoảng $\alpha = 0,1, 0,01, 0,001$.

Dòng 7 tập tin là ma trận trọng số W với các giá trị ngẫu nhiên được lấy mẫu từ phân phối chuẩn hóa (Gaussian) H_y với giá trị trung bình và phương sai đơn vị bằng 0. Ma trận trọng số có các mục nhập N + 1, một mục nhập cho mỗi đầu vào N trong vecto đặc trưng, cộng với một mục nhập cho độ lệch. Chia W cho căn bậc hai số lượng đầu vào, đây là một kỹ thuật phổ biến thường được sử dụng để chia tỷ lệ ma trận trọng số, dẫn đến sự hội tụ nhanh hơn. Các kỹ thuật khởi tạo trọng số sẽ được đề cập ở các phần sau trong chương này.

Tiếp theo, hãy để xác định hàm bước:

```
10 def step(self, x):
11     # apply the step function
12     return 1 if x > 0 else 0
```

Hàm này bắt chước hành vi của hàm bước trong Hình 1.4 - nếu x dương, trả về 1, nếu không như vậy, trả về 0.

Để thực sự huấn luyện Perceptron, cần tìm hiểu một hàm có tên là fit. Nếu đã nghiên cứu về học máy, Python và thư viện scikit-learn thì sẽ biết rằng nó phổ biến để đặt tên cho hàm quy trình huấn luyện phù hợp, như fit với một mô hình cho dữ liệu.

```
14 def fit(self, X, y, epochs=10):
15     # insert a column of 1's as the last entry in the feature
16     # matrix -- this little trick allows us to treat the bias
17     # as a trainable parameter within the weight matrix
18     X = np.c_[X, np.ones((X.shape[0]))]
```

Phương thức phù hợp yêu cầu hai tham số theo sau là một tùy chọn duy nhất:

Giá trị X là dữ liệu huấn luyện thực tế. Biến y là lớp nhãn đầu ra mục tiêu (nghĩa là, dự đoán cái gì). Cuối cùng là nhập vào số lượng chu kỳ mà Perceptron huấn luyện.

Dòng 18 áp dụng thủ thuật bias (Mục 8.3) bằng cách chèn một cột dữ liệu vào dữ liệu huấn luyện, cho phép coi bias là một tham số có thể huấn luyện trực tiếp bên trong ma trận trọng số.

Tiếp theo, hãy rà soát lại quy trình huấn luyện thực tế:

```

20      # loop over the desired number of epochs
21      for epoch in np.arange(0, epochs):
22          # loop over each individual data point
23          for (x, target) in zip(X, y):
24              # take the dot product between the input features
25              # and the weight matrix, then pass this value
26              # through the step function to obtain the prediction
27              p = self.step(np.dot(x, self.W))
28
29          # only perform a weight update if our prediction
30          # does not match the target
31          if p != target:
32              # determine the error
33              error = p - target
34
35          # update the weight matrix
36          self.W += -self.alpha * error * x

```

Tại Dòng 21, bắt đầu lặp với số lượng chu kỳ mong muốn. Đối với mỗi epoch, cũng lặp qua từng điểm dữ liệu riêng lẻ x và lớp nhãn đầu ra mong muốn (Dòng 23).

Dòng 27 lấy nhân chập giữa các hàm đầu vào x và ma trận trọng số W , sau đó chuyển đầu ra qua hàm bước để có được dự đoán Perceptron.

Áp dụng quy trình huấn luyện tương tự được nêu chi tiết trong các bước huấn luyện ở trên, chỉ thực hiện cập nhật trọng số nếu dự đoán không phù hợp với dữ liệu mong muốn (Dòng 31). Nếu xảy ra lỗi, xác định lỗi (Dòng 33) bằng cách tính dấu (dương hoặc âm) thông qua thao tác tính chênh lệch.

Cập nhật ma trận trọng số được xử lý tại Dòng 36 trong đó thực hiện một bước để phân loại chính xác, nhân bước này với tốc độ học alpha. Qua một loạt các chu kỳ, Perceptron có thể học các mẫu trong dữ liệu cơ bản và thay đổi các giá trị ma trận trọng số sao cho phân loại chính xác với các mẫu đầu vào x .

Hàm cuối cùng cần tìm hiểu là predict, được sử dụng để dự đoán các lớp nhãn cho một tập hợp dữ liệu đầu vào nhất định:

```

38      def predict(self, X, addBias=True):
39          # ensure our input is a matrix
40          X = np.atleast_2d(X)
41
42          # check to see if the bias column should be added
43          if addBias:
44              # insert a column of 1's as the last entry in the feature
45              # matrix (bias)
46              X = np.c_[X, np.ones((X.shape[0]))]
47
48          # take the dot product between the input features and the
49          # weight matrix, then pass the value through the step
50          # function
51          return self.step(np.dot(X, self.W))

```

Phương pháp dự đoán yêu cầu một bộ dữ liệu đầu vào X cần được phân loại. Kiểm tra tại Dòng 43 được thực hiện để xem có cần thêm cột bias hay không.

Lấy dự đoán đầu ra cho X giống như quy trình huấn luyện - chỉ cần lấy nhân chập giữa các hàm đầu vào X và ma trận trọng số W, sau đó chuyển giá trị qua hàm bước. Đầu ra hàm bước được trả về hàm gọi.

Bây giờ, chúng ta đã triển khai lớp Perceptron, hãy để có gắng áp dụng nó vào các bộ dữ liệu bitwise và xem mạng nơ-ron thực hiện như thế nào.

1.1.3.6. Đánh giá bộ dữ liệu bit với Perceptron

Để bắt đầu, hãy nhanh chóng tạo một tệp có tên perceptron_or.py có gắng khớp mô hình Perceptron với tập dữ liệu bitwise OR:

```
1 # import the necessary packages
2 from pyimagesearch.nn import Perceptron
3 import numpy as np
4
5 # construct the OR dataset
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np.array([0, 1, 1, 1])
8
9 # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
```

Dòng 2 và 3 nhập các gói Python cần thiết. Hãy sử dụng ứng dụng Perceptron từ Phần 1.1.2 ở trên. Dòng 6 và 7 xác định tập dữ liệu OR dựa vào Bảng 1.1.

Dòng 11 và 12 huấn luyện Perceptron với tốc độ học là $\alpha = 0,1$ với tổng số 20 epoch.

Sau đó, có thể đánh giá Perceptron trên dữ liệu để xác thực rằng trên thực tế, nó đã huấn luyện về hàm OR:

```
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19     # make a prediction on the data point and display the result
20     # to our console
21     pred = p.predict(x)
22     print("[INFO] data={}, ground-truth={}, pred={}".format(
23         x, target[0], pred))
```

Tại Dòng 18, lặp qua từng điểm dữ liệu trong bộ dữ liệu OR. Đối với mỗi điểm dữ liệu này, chuyển qua mạng và có được dự đoán (Dòng 21).

Cuối cùng, Dòng 22 và 23 hiển thị điểm dữ liệu đầu vào, nhãn thật, cũng như nhãn dự đoán cho bảng điều khiển.

Để xem thuật toán Perceptron có thể huấn luyện hàm OR hay không, chỉ cần thực hiện lệnh sau:

```
$ python perceptron_or.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=1, pred=1
[INFO] data=[1 0], ground-truth=1, pred=1
[INFO] data=[1 1], ground-truth=1, pred=1
```

Chắc chắn, mạng nơ-ron có thể dự đoán chính xác rằng hoạt động OR cho $x_0 = 0$ và $x_1 = 0$ là 0 - tất cả các kết hợp khác là 1.

Bây giờ, hãy để di chuyển sang hàm AND - tạo một tệp mới có tên perceptron_and.py và chèn chương trình sau đây:

```
1 # import the necessary packages
2 from pyimagesearch.nn import Perceptron
3 import numpy as np
4
5 # construct the AND dataset
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np.array([0, 0, 0, 1])
8
9 # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
13
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19     # make a prediction on the data point and display the result
20     # to our console
21     pred = p.predict(x)
22     print("[INFO] data={}, ground-truth={}, pred={}".format(
23         x, target[0], pred))
```

Lưu ý ở đây rằng các dòng chương trình duy nhất đã thay đổi là Dòng 6 và 7 là dữ liệu AND đã thay thế dữ liệu OR.

Thực hiện lệnh sau, có thể đánh giá Perceptron trên hàm AND:

```
$ python perceptron_and.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=0
[INFO] data=[0 1], ground-truth=0, pred=0
[INFO] data=[1 0], ground-truth=0, pred=0
[INFO] data=[1 1], ground-truth=1, pred=1
```

Một lần nữa, Perceptron đã có thể mô hình hóa hàm chính xác. Hàm AND chỉ đúng khi cả $x_0 = 1$ và $x_1 = 1$ - đối với tất cả các kết hợp khác, bitwise bằng 0.

Cuối cùng, hãy phân tích xem hàm XOR có thể phân tách phi tuyến bên trong `perceptron_xor.py`:

```
1 # import the necessary packages
2 from pyimagesearch.nn import Perceptron
3 import numpy as np
4
5 # construct the XOR dataset
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np.array([0, 1, 1, 0])
8
9 # define our perceptron and train it
10 print("[INFO] training perceptron...")
11 p = Perceptron(X.shape[1], alpha=0.1)
12 p.fit(X, y, epochs=20)
13
14 # now that our perceptron is trained we can evaluate it
15 print("[INFO] testing perceptron...")
16
17 # now that our network is trained, loop over the data points
18 for (x, target) in zip(X, y):
19     # make a prediction on the data point and display the result
20     # to our console
21     pred = p.predict(x)
22     print("[INFO] data={}, ground-truth={}, pred={}".format(
23         x, target[0], pred))
```

Một lần nữa, các dòng chương trình duy nhất cần thay đổi là Dòng 6 và 7 nơi xác định dữ liệu XOR. Toán tử XOR là đúng khi và chỉ khi một (không phải cả hai) \times L là 1.

Thực hiện lệnh sau và có thể thấy rằng Perceptron không thể tìm hiểu mối quan hệ phi tuyến tính này:

```
$ python perceptron_xor.py
[INFO] training perceptron...
[INFO] testing perceptron...
[INFO] data=[0 0], ground-truth=0, pred=1
[INFO] data=[0 1], ground-truth=1, pred=1
[INFO] data=[1 0], ground-truth=1, pred=0
[INFO] data=[1 1], ground-truth=0, pred=0
```

Cho dù có chạy thử nghiệm này bao nhiêu lần với các mức học khác nhau hoặc các sơ đồ khởi tạo trọng số khác nhau, không bao giờ có thể mô hình chính xác hàm XOR với một lớp Perceptron. Thay vào đó, những gì cần là nhiều lớp hơn - và cùng với đó, bắt đầu học sâu.

1.1.4. Lan truyền ngược và mạng nhiều lớp

Lan truyền ngược được cho là thuật toán quan trọng nhất trong lịch sử mạng nơ-ron - nếu không có lan truyền ngược, không thể huấn luyện mạng học sâu đến độ sâu mà ta thấy như hiện nay. Lan truyền ngược có thể được coi là nền tảng mạng nơ-ron hiện đại và học sâu.

Lan truyền ngược đã được giới thiệu vào những năm 1970,, nhưng chỉ khi bài báo Learning representations by back-propagating errors, của Hinton và Williams được xuất bản năm 1986 [16], người ta mới có thể nghĩ ra thuật toán nhanh hơn, tinh tế hơn để huấn luyện mạng sâu.

Đã có khá nhiều nghiên cứu kinh điển về lan truyền ngược:

1. Phân tích của Andrew Ng, về việc lan truyền ngược trong khóa học “Máy học” Coursera [17].
2. Trình bày từ cơ sở toán học trong nghiên cứu - How the backpropagation algorithm works - Michael Nielsen [19].
3. Khai phá và phân tích lan truyền ngược: Khóa Stanford cs231n [20].
4. Matt Mazur với các ví dụ cụ thể (với số lượng ví dụ thực tế) cho thấy cách hoạt động lan truyền ngược [21].

Như có thể thấy, các hướng dẫn về thuật toán lan truyền ngược rất phong phú, tuy nhiên, thay vì tập hợp lại và nhắc lại những gì đã được người khác trình bày, cuốn sách sẽ có một cách tiếp cận khác: *Xây dựng một cách trực quan, dễ thực hiện theo thuật toán lan truyền ngược sử dụng ngôn ngữ Python.*

Cụ thể, chúng ta sẽ xây dựng một mạng nơ-ron thực tế và huấn luyện nó bằng thuật toán lan truyền ngược. Khi hoàn thành phần này, người học sẽ hiểu cách hoạt động của lan truyền ngược - và có lẽ quan trọng hơn, sẽ hiểu rõ về cách mà thuật toán này được sử dụng để huấn luyện các mạng nơ-ron từ đầu.

1.1.4.1. Lan truyền ngược

Thuật toán Lan truyền ngược bao gồm hai giai đoạn:

1. Truyền thẳng từ ngõ vào qua mạng và dự đoán đầu ra thu được (còn được gọi là giai đoạn lan truyền).

Bảng 1.2: Trái: Tập dữ liệu XOR bitwise (bao gồm nhãn các lớp). Phải: Ma trận thiết kế bộ dữ liệu XOR với một cột bias được chèn (không bao gồm nhãn các lớp cho ngắn gọn).

x_0	x_1	y	x_0	x_1	x_2
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	0	1	1	1

2. Lan truyền ngược trong đó tính toán độ dốc hàm tổn thất ở lớp cuối cùng (lớp dự đoán) mạng và sử dụng độ dốc này để áp dụng đệ quy chuỗi để cập nhật trọng số trong mạng (còn được gọi là trọng số giai đoạn cập nhật).

Bắt đầu bằng cách xem xét từng giai đoạn ở mức độ cấp cao. Thực hiện thuật toán lan truyền ngược bằng Python. Khi đã triển khai lan truyền ngược, mục đích đưa ra là dự đoán bằng cách sử dụng mạng (đây chỉ đơn giản là giai đoạn chuyển tiếp) chỉ với một điều chỉnh nhỏ (về đoạn chương trình) để giúp dự đoán hiệu quả hơn.

Cuối cùng, trình bày cách huấn luyện một mạng nơ-ron tùy chỉnh bằng cách sử dụng lan truyền ngược và Python trên cả hai bộ dữ liệu:

1. Bộ dữ liệu XOR
2. Bộ dữ liệu MNIST

1.1.4.2. Truyền thẳng

Mục đích việc chuyển tiếp là để lan truyền ngược đầu vào thông qua mạng bằng cách áp dụng một loạt các phép nhân chập và kích hoạt cho đến khi đến lớp đầu ra mạng (dự đoán). Để hình dung quá trình này, trước tiên, hãy xem xét bộ dữ liệu XOR (Bảng 1.2, bên trái).

Ở đây có thể thấy rằng mỗi mục X trong ma trận thiết kế (bên trái) là 2 chiều - mỗi điểm dữ liệu được biểu thị bằng hai số. Ví dụ: điểm dữ liệu đầu tiên được biểu thị bằng vectơ đặc trưng $(0, 0)$, điểm dữ liệu thứ hai bằng $(0, 1)$, v.v. Sau đó, có các giá trị đầu ra y là cột bên phải. Giá trị đầu ra mục tiêu là nhãn của các lớp. Đưa ra một dữ liệu đầu vào từ ma trận thiết kế, mục tiêu là dự đoán chính xác giá trị đầu ra mong muốn.

Như đã tìm hiểu trong mục 1.1.3, để có được độ chính xác phân loại hoàn hảo cho vấn đề này, sẽ cần một mạng nơ-ron với ít nhất một lớp ẩn

duy nhất, vì vậy hãy tiếp tục và bắt đầu với kiến trúc 2-2-1 (Hình 1.9, trên cùng). Tuy nhiên, cần bao gồm cả hệ số bias. Có hai cách để đưa hệ số bias b vào mạng (chương 3). Có thể:

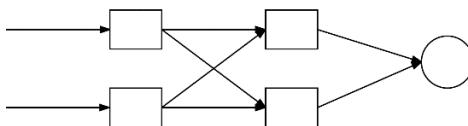
1. Sử dụng một biến riêng.

2. Xử lý sai lệch như một tham số có thể huấn luyện trong ma trận trọng số bằng cách chèn một cột 1 chiều vào các vectơ đặc trưng.

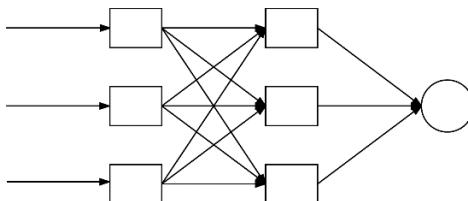
Việc chèn một cột 1 chiều vào vectơ đặc trưng được thực hiện theo chương trình có sẵn, nhưng để đảm bảo hiểu được điểm này, hãy cập nhật ma trận thiết kế XOR để thấy điều này diễn ra (Bảng 1.2, bên phải). Như có thể thấy, một cột 1 chiều đã được thêm vào các vectơ đặc trưng. Trong thực tế, có thể chèn cột này vào bất cứ nơi nào thích, nhưng thường đặt nó là (1) cột đầu tiên trong vectơ đặc trưng hoặc (2) mục cuối cùng trong vectơ đặc trưng.

Vì đã thay đổi kích thước vectơ đặc trưng đầu vào (thường được thực hiện bên trong phần triển khai mạng nơ-ron để không cần phải sửa đổi ma trận thiết kế) - làm thay đổi kiến trúc mạng (nhận thức) từ 2 - 2 - 1 thành 3 - 3 - 1 (Hình 1.8, phía dưới). Đầu vào kiến trúc mạng này là 2 - 2 - 1, nhưng khi thực hiện là 3 - 3 - 1 do bổ sung hệ số bias được nhúng trong ma trận trọng số

2-2-1



3-3-1

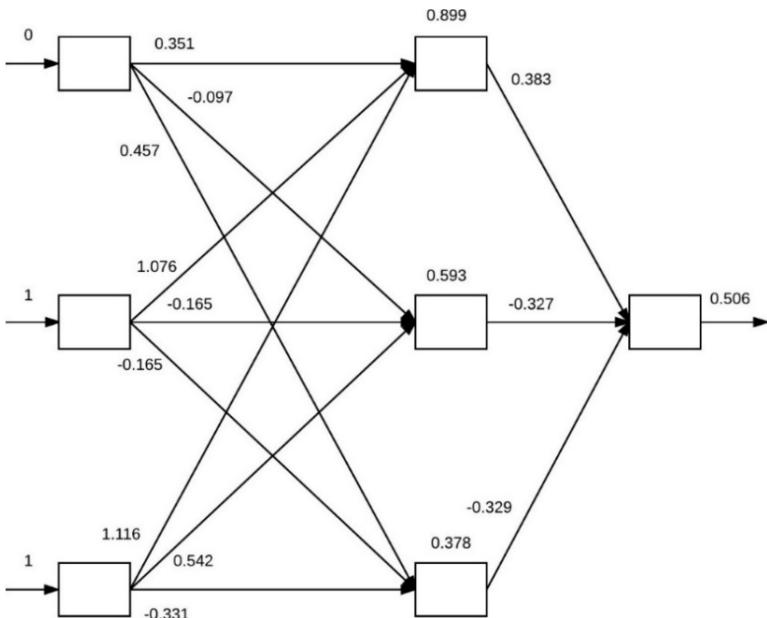


Hình 1.8: Trên cùng: Để xây dựng một mạng nơ-ron để phân loại chính xác bộ dữ liệu XOR, cần một mạng có hai nút đầu vào, hai nút ẩn và một nút đầu ra. Điều này dẫn đến một kiến trúc 2-2-1. Dưới cùng: Đại diện kiến trúc cục bộ thực tế là 3-3-1 do thủ thuật bias. Trong phần lớn các triển khai mạng nơ-ron, việc điều chỉnh ma trận trọng số này xảy ra cục bộ.

Nên nhớ rằng lớp đầu vào và tất cả các lớp ẩn đều cần một hệ số bias; tuy nhiên, lớp đầu ra cuối cùng không yêu cầu bias. Lợi ích của việc áp dụng thủ thuật bias là không cần phải theo dõi rõ ràng thông số bias nữa - giờ đây nó là một tham số có thể huấn luyện trong ma trận trọng số, do đó giúp việc huấn luyện hiệu quả hơn và dễ thực hiện hơn.

Tiếp theo, khởi tạo các trọng số trong mạng, như trong Hình 1.9. Lưu ý cách mỗi mũi tên trong ma trận trọng số có một giá trị được liên kết với nhau - đây là giá trị trọng số hiện tại cho một nút nhất định và biểu thị số lượng trong đó một đầu vào đã cho được khuếch đại hoặc giảm đi. Giá trị trọng số này sau đó hãy được cập nhật trong giai đoạn lan truyền ngược.

Ở phía xa bên trái Hình 1.9, trình bày vectơ đặc trưng $(0, 1, 1)$ (và giá trị đầu ra có giá trị 1 cho mạng). Ở đây có thể thấy rằng 0, 1 và 1 đã được gán cho ba nút đầu vào trong mạng. Để truyền các giá trị qua mạng và có được phân loại cuối cùng, cần lấy nhân chập giữa các giá trị đầu vào và trọng số, tiếp theo là áp dụng hàm kích hoạt (trong trường hợp này là hàm sigmoid).



Hình 1.9: Một ví dụ về đường truyền lan truyền thẳng. Vectơ đầu vào $[0, 1, 1]$ được truyền cho mạng. Nhân chập các đầu vào và trọng số được đưa đến hàm kích hoạt sigmoid để thu được các giá trị trong lớp ẩn (lần lượt là $0,899, 0,593$ và $0,378$). Cuối cùng, nhân chập với hàm kích hoạt sigmoid được tính cho lớp cuối cùng, thu được giá trị $0,506$. Áp dụng hàm bước cho $0,506$ mang lại 1, đây thực sự là lớp nhãn mục tiêu.

Tính toán các đầu vào cho ba nút trong các lớp ẩn:

$$1. ((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$$

$$2. \sigma ((0 \cdot 0,097) + (1 \cdot 0,125) + (1 \cdot 0,542)) = 0,593$$

$$3. \sigma ((0 \cdot 0,457) + (1 \cdot 0,165) + (1 \cdot 0,331)) = 0,378$$

Nhìn vào các giá trị nút các lớp ẩn (Hình 1.9, giữa), có thể thấy các nút đã được cập nhật phần tính toán.

Bây giờ có đầu vào của các nút lớp ẩn. Để tính toán dự đoán đầu ra, một lần nữa cần tính toán nhân chập với hàm kích hoạt sigmoid:

$$\sigma ((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

Do đó, đầu ra của mạng là 0,506. Có thể áp dụng một hàm bước để xác định xem đầu ra này có phải là phân loại chính xác hay không:

$$f(\text{net}) = \begin{cases} 1, & \text{net} > 0 \\ 0, & \text{net} \leq 0 \end{cases} \quad (1.10)$$

Áp dụng hàm bước với net = 0,506 thấy rằng mạng dự đoán 1 thực tế là lớp nhãn đúng. Tuy nhiên, mạng không tự tin lầm vào lớp nhãn này - giá trị dự đoán 0,506 rất gần với ngưỡng bước. Lý tưởng nhất là dự đoán này hãy gần hơn với 0,98 0,99, Nghĩa là mạng đã thực sự học được mô hình cơ bản trong bộ dữ liệu. Để mạng thực sự có thể học hỏi, cần áp dụng lan truyền ngược.

1.1.4.3. Lan truyền ngược

Để áp dụng thuật toán lan truyền ngược, hàm kích hoạt phải có thể tính đạo hàm một lần đối với trọng số đã cho $w_{i,j}$, loss (E), đầu ra nút o_j và đầu ra mạng j .

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{i,j}} \quad (1.11)$$

Vì ý nghĩa đằng sau thuật toán lan truyền ngược đã được giải thích cụ thể trong các ấn phẩm (Andrew Ng [17], Michael Nielsen [19], Matt Mazur [21]), nên sẽ bỏ qua việc tạo ra bản cập nhật quy tắc chuỗi lan truyền ngược thay vì giải thích nó thông qua đoạn chương trình trong phần sau.

Để tìm hiểu sâu hơn, xin vui lòng xem các tài liệu tham khảo ở trên để biết thêm thông tin về quy tắc chuỗi và vai trò nó trong thuật toán lan truyền ngược. Bằng cách giải thích quy trình này trong đoạn chương trình, mục tiêu là giúp người đọc hiểu được việc lan truyền ngược thông qua ý nghĩa triển khai, trực quan hơn.

1.1.4.4. Triển khai lan truyền ngược với Python

Bắt đầu thực hiện lan truyền ngược: Mở một tập tin mới, đặt tên cho nó là neuralnetwork.py và để bắt đầu làm việc:

```
1 # import the necessary packages
2 import numpy as np
3
4 class NeuralNetwork:
5     def __init__(self, layers, alpha=0.1):
6         # initialize the list of weights matrices, then store the
7         # network architecture and learning rate
8         self.W = []
9         self.layers = layers
10        self.alpha = alpha
```

Tại Dòng 2, nhập gói duy nhất mà cần để triển khai lan truyền ngược - thư viện xử lý số NumPy.

Dòng 5 sau đó định nghĩa hàm tạo cho lớp NeuralNetwork. Hàm tạo yêu cầu một đối số duy nhất, theo sau là một đối số tùy chọn thứ hai:

- Các lớp: Một danh sách các số nguyên biểu thị kiến trúc thực tế mạng lan truyền thẳng. Ví dụ: giá trị [2, 2, 1] hãy nghĩa là lớp đầu vào đầu tiên có hai nút, lớp ẩn có hai nút và lớp đầu ra cuối cùng có một nút.
- Alpha: chỉ định tốc độ học của mạng nơ-ron. Giá trị này được áp dụng trong giai đoạn cập nhật trọng số.

Dòng 8 khởi tạo danh sách các trọng số cho mỗi lớp, W. Sau đó lưu trữ các lớp và alpha tại Dòng 9 và 10.

Danh sách các trọng số W trống, vì vậy trước khi bắt đầu hãy khởi tạo nó

```
12     # start looping from the index of the first layer but
13     # stop before we reach the last two layers
14     for i in np.arange(0, len(layers) - 2):
15         # randomly initialize a weight matrix connecting the
16         # number of nodes in each respective layer together,
17         # adding an extra node for the bias
18         w = np.random.randn(layers[i] + 1, layers[i + 1] + 1)
19         self.W.append(w / np.sqrt(layers[i]))
```

Tại Dòng 14, bắt đầu vòng lặp qua số lớp trong mạng (tức là độ dài của các lớp (len(layers))), nhưng dừng lại trước hai lớp cuối (có thể tìm hiểu chính xác lý do tại sao sau này trong phần giải thích cách tạo hàm này ở phần sau).

Mỗi lớp trong mạng được khởi tạo ngẫu nhiên bằng cách xây dựng

ma trận trọng số $M \times N$, bằng cách lấy mẫu các giá trị từ một phân phối chuẩn, chuẩn hóa (normalize) (Dòng 18) Ma trận có kích thước là $M \times N$ và mục tiêu kết nối mọi nút trong lớp hiện tại với mọi nút trong lớp tiếp theo.

Ví dụ, hãy giả sử có lớp $[i] = 2$ và lớp $[i + 1] = 2$. Do đó, ma trận trọng số của sẽ là 2×2 để kết nối tất cả các bộ nút giữa các lớp. Tuy nhiên, chúng ta cần cẩn thận ở đây, vì chúng ta đang quên một thành phần quan trọng - hệ số bias. Để tính toán độ lệch, cần thêm một vào lớp $[i]$ và lớp $[i + 1]$ – để thay đổi ma trận trọng số thành hình dạng 3×3 cho các nút $2 + 1$ ở lớp hiện tại và $2 + 1$ cho các nút ở lớp tiếp theo. Chia tỷ lệ w bằng cách chia cho căn bậc hai của số nút trong lớp hiện tại, do đó chuẩn hóa (normalize) phương sai của mỗi đầu ra nơ ron nơ-ron [57] (Dòng 19).

Khối chương trình cuối cùng: hàm xử lý đặc biệt trong đó các kết nối đầu vào cần một hệ số bias, nhưng đầu ra thì không:

```
21     # the last two layers are a special case where the input
22     # connections need a bias term but the output does not
23     w = np.random.randn(layers[-2] + 1, layers[-1])
24     self.W.append(w / np.sqrt(layers[-2]))
```

Một lần nữa, các giá trị trọng số này được lấy mẫu ngẫu nhiên và sau đó được chuẩn hóa.

Hàm tiếp theo được định nghĩa là một phương thức ma thuật Python “magic method” named `repr` – hàm này rất hữu ích để gõ lỗi:

```
26     def __repr__(self):
27         # construct and return a string that represents the network
28         # architecture
29         return "NeuralNetwork: {}".format(
30             "-".join(str(l) for l in self.layers))
```

Trường hợp định dạng một chuỗi cho đối tượng `NeuralNetwork` bằng cách nối giá trị số nguyên các nút trong mỗi lớp. Nhập giá trị lớp là $(2, 2, 1)$, đầu ra của hàm này là:

```
1 >>> from pyimagesearch.nn import NeuralNetwork
2 >>> nn = NeuralNetwork([2, 2, 1])
3 >>> print(nn)
4 NeuralNetwork: 2-2-1
```

Tiếp theo, có thể định nghĩa hàm kích hoạt sigmoid:

```
32     def sigmoid(self, x):
33         # compute and return the sigmoid activation value for a
34         # given input value
35         return 1.0 / (1 + np.exp(-x))
```

và đạo hàm sigmoid sử dụng trong quá trình truyền ngược:

```
37     def sigmoid_deriv(self, x):
38         # compute the derivative of the sigmoid function ASSUMING
39         # that 'x' has already been passed through the 'sigmoid'
40         # function
41         return x * (1 - x)
```

Lưu ý rằng bất cứ khi nào thực hiện lan truyền ngược, hãy luôn chọn một hàm kích hoạt khả vi. Có thể lấy từ thư viện scikit-learn và xác định một hàm có tên là fit để huấn luyện NeuralNetwork:

```
43     def fit(self, X, y, epochs=1000, displayUpdate=100):
44         # insert a column of 1's as the last entry in the feature
45         # matrix -- this little trick allows us to treat the bias
46         # as a trainable parameter within the weight matrix
47         X = np.c_[X, np.ones((X.shape[0]))]
48
49         # loop over the desired number of epochs
50         for epoch in np.arange(0, epochs):
51             # loop over each individual data point and train
52             # our network on it
53             for (x, target) in zip(X, y):
54                 self.fit_partial(x, target)
55
56             # check to see if we should display a training update
57             if epoch == 0 or (epoch + 1) % displayUpdate == 0:
58                 loss = self.calculate_loss(X, y)
59                 print("[INFO] epoch={}, loss={:.7f}".format(
60                     epoch + 1, loss))
```

Phương thức phù hợp yêu cầu hai tham số, theo sau là hai tham số tùy chọn. Đầu tiên, X là dữ liệu huấn luyện. Thứ hai, y là lớp nhãn tương ứng cho mỗi mục trong X. Tiếp theo, chỉ định các epoch là số vòng lặp huấn luyện mạng. Tham số displayUpdate chỉ đơn giản kiểm soát số lượng N epoch in trong tiến trình huấn luyện đến thiết bị đầu cuối.

Tại Dòng 47, thực hiện thủ thuật bias bằng cách chèn một cột 1 là mục nhập cuối cùng trong ma trận đặc trưng, X. Từ đó, bắt đầu lặp lại số lượng chu kỳ tại Dòng 50. Đối với mỗi chu kỳ, hãy lặp qua từng cột các điểm dữ liệu riêng lẻ trong tập huấn luyện, đưa ra dự đoán về điểm dữ liệu, tính toán giai đoạn lan truyền ngược và sau đó cập nhật ma trận trọng số (Dòng 53 và 54). Các dòng 57-60 chỉ cần kiểm tra có hiển thị bản cập nhật huấn luyện ra màn hình hay không?

Máu chốt của thuật toán lan truyền ngược nằm bên trong hàm fit_partial dưới đây:

Hàm fit_partial yêu cầu hai tham số:

```
62     def fit_partial(self, x, y):
63         # construct our list of output activations for each layer
64         # as our data point flows through the network; the first
65         # activation is a special case -- it's just the input
66         # feature vector itself
67         A = [np.atleast_2d(x)]
```

- x: một điểm dữ liệu riêng lẻ từ ma trận thiết kế.
- y: lớp nhãn tương ứng.

Sau đó, khởi tạo một danh sách A- tại Dòng 67. Danh sách này chịu trách nhiệm lưu trữ các kích hoạt ngõ ra cho mỗi lớp khi điểm dữ liệu x chuyển tiếp qua mạng. Khởi tạo danh sách này với x - điểm dữ liệu đầu vào.

Từ đây, có thể bắt đầu giai đoạn lan truyền thẳng:

```
69     # FEEDFORWARD:
70     # loop over the layers in the network
71     for layer in np.arange(0, len(self.W)):
72         # feedforward the activation at the current layer by
73         # taking the dot product between the activation and
74         # the weight matrix -- this is called the "net input"
75         # to the current layer
76         net = A[layer].dot(self.W[layer])
77
78         # computing the "net output" is simply applying our
79         # nonlinear activation function to the net input
80         out = self.sigmoid(net)
81
82         # once we have the net output, add it to our list of
83         # activations
84         A.append(out)
```

Bắt đầu lặp qua mọi lớp trong mạng tại Dòng 71. Đầu vào net đưa đến lớp này được tính bằng cách nhân chập giữa hàm tách động và ma trận trọng số (Dòng 76). Đầu ra net lớp hiện tại sau đó được tính bằng cách truyền đầu vào net qua hàm kích hoạt sigmoid phi tuyến. Khi có đầu ra net, thêm nó vào danh sách kích hoạt (Dòng 84).

Đoạn chương trình này là toàn bộ quá trình lan truyền thẳng được mô tả trong Phần 1.1.3 ở trên - chỉ đơn giản là lặp qua từng lớp trong mạng, lấy nhân chập giữa kích hoạt và trọng số, chuyển giá trị qua một hàm kích hoạt phi tuyến và tiếp tục đến lớp tiếp theo. Do đó, mục cuối cùng trong A là đầu ra lớp cuối cùng trong mạng (nghĩa là dự đoán).

Bây giờ, đường truyền thẳng đã hoàn thành, có thể chuyển sang đường truyền ngược phức tạp hơn một chút

```
96     # BACKPROPAGATION
97     # the first phase of backpropagation is to compute the
98     # difference between our *prediction* (the final output
99     # activation in the activations list) and the true target
100    # value
101
102    error = A[-1] - y
103
104    # from here, we need to apply the chain rule and build our
105    # list of deltas 'D'; the first entry in the deltas is
106    # simply the error of the output layer times the derivative
107    # of our activation function for the output value
108    D = [error * self.sigmoid_deriv(A[-1])]
```

Giai đoạn đầu tiên của đường lan truyền ngược là tính toán lỗi, hoặc đơn giản là sự khác biệt giữa nhãn dự đoán và nhãn thật (Dòng 91). Vì mục cuối cùng trong danh sách kích hoạt A chứa ngõ ra của mạng, có thể truy cập dự đoán đầu ra thông qua A [-1]. Giá trị y là đầu ra đích cho điểm dữ liệu đầu vào x.

Khi sử dụng ngôn ngữ lập trình Python, việc chỉ định giá trị chỉ mục -1 cho biết mong muốn truy cập mục cuối cùng trong danh sách.

Tiếp theo, cần bắt đầu áp dụng quy tắc chuỗi để xây dựng danh sách deltas, D. Các deltas được sử dụng để cập nhật ma trận trọng số, được chia tỷ lệ theo tỷ lệ alpha học. Mục nhập đầu tiên trong danh sách deltas là lỗi lớp đầu ra nhân với đạo hàm sigmoid cho giá trị đầu ra (Dòng 97).

Với delta cho lớp cuối cùng trong mạng, bây giờ có thể làm việc ngược bằng cách sử dụng vòng lặp for:

```
99     # once you understand the chain rule it becomes super easy
100    # to implement with a 'for' loop -- simply loop over the
101    # layers in reverse order (ignoring the last two since we
102    # already have taken them into account)
103    for layer in np.arange(len(A) - 2, 0, -1):
104        # the delta for the current layer is equal to the delta
105        # of the *previous layer* dotted with the weight matrix
106        # of the current layer, followed by multiplying the delta
107        # by the derivative of the nonlinear activation function
108        # for the activations of the current layer
109        delta = D[-1].dot(self.W[layer].T)
110        delta = delta * self.sigmoid_deriv(A[layer])
111        D.append(delta)
```

Tại Dòng 103, bắt đầu lặp qua từng lớp trong mạng (bỏ qua hai lớp trước vì chúng đã được tính tại Dòng 97) theo thứ tự ngược lại khi cần làm việc với lan truyền ngược để tính toán các cập nhật delta cho mỗi lớp. Delta cho lớp hiện tại bằng với delta của lớp trước, D [-1].dot với ma trận trọng số của lớp hiện tại (Dòng 21). Để kết thúc tính toán của delta,

nhân nó bằng cách truyền hàm kích hoạt cho lớp với đạo hàm của sigmoid (Dòng 110). Sau đó, cập nhật danh sách deltas D với delta vừa tính toán (Dòng 111).

Nhìn vào khối chương trình này, chúng ta có thể thấy rằng bước lan truyền ngược được lặp đi lặp lại - chúng ta chỉ đơn giản là lấy delta từ lớp trước, nhân chấm nó với trọng số của lớp hiện tại (lấy tích chập), sau đó nhân với đạo hàm của hàm kích hoạt. Quá trình này được lặp lại cho đến khi chúng ta đạt được lớp đầu tiên trong mạng.

Với danh sách delta D đã thu được, có thể chuyển sang giai đoạn cập nhật trọng số: Nhìn vào khối chương trình này, có thể thấy rằng bước lan truyền ngược là lặp đi lặp lại - chỉ đơn giản là lấy delta từ lớp trước, dùng hàm dot với trọng số lớp hiện tại, sau đó nhân với đạo hàm kích hoạt. Quá trình này được lặp lại cho đến khi đạt được lớp đầu tiên trong mạng.

Với danh sách delta D, có thể chuyển sang giai đoạn cập nhật trọng số:

```
113      # since we looped over our layers in reverse order we need to
114      # reverse the deltas
115      D = D[::-1]
116
117      # WEIGHT UPDATE PHASE
118      # loop over the layers
119      for layer in np.arange(0, len(self.W)):
120          # update our weights by taking the dot product of the layer
121          # activations with their respective deltas, then multiplying
122          # this value by some small learning rate and adding to our
123          # weight matrix -- this is where the actual "learning" takes
124          # place
125          self.W[layer] += -self.alpha * A[layer].T.dot(D[layer])
```

Hãy nhớ rằng trong bước lan truyền ngược đã lặp qua các lớp theo thứ tự ngược lại. Để thực hiện giai đoạn cập nhật trọng số, đơn giản là đảo ngược thứ tự các mục trong D để có thể lặp qua từng lớp từ 0 đến N; D là tổng số lớp trong mạng (Dòng 115).

Cập nhật ma trận trọng số thực tế (tức là, nơi diễn ra quá trình huấn luyện thực tế) được thực hiện tại Dòng 125, đó là thuật toán suy giảm độ dốc (gradient descent). Lấy nhân chập hàm kích hoạt với lớp hiện tại, A [lớp] với các delta của lớp hiện tại, D [lớp] và nhân chúng theo tốc độ học, alpha. Giá trị này được thêm vào ma trận trọng số cho lớp hiện tại, W [lớp].

Lặp lại quá trình này cho tất cả các lớp trong mạng. Sau khi thực hiện giai đoạn cập nhật trọng số, lan truyền ngược chính thức được thực hiện. Khi mạng được huấn luyện trên một tập dữ liệu nhất định, mục tiêu là đưa ra dự đoán về bộ dữ liệu thử nghiệm, có thể được thực hiện thông qua phương pháp dự đoán dưới đây:

```

127 def predict(self, X, addBias=True):
128     # initialize the output prediction as the input features -- this
129     # value will be (forward) propagated through the network to
130     # obtain the final prediction
131     p = np.atleast_2d(X)
132
133     # check to see if the bias column should be added
134     if addBias:
135         # insert a column of 1's as the last entry in the feature
136         # matrix (bias)
137         p = np.c_[p, np.ones((p.shape[0]))]
138
139     # loop over our layers in the network
140     for layer in np.arange(0, len(self.W)):
141         # computing the output prediction is as simple as taking
142         # the dot product between the current activation value 'p'
143         # and the weight matrix associated with the current layer,
144         # then passing this value through a nonlinear activation
145         # function
146         p = self.sigmoid(np.dot(p, self.W[layer]))
147
148     # return the predicted value
149     return p

```

Hàm dự đoán chỉ đơn giản là một đường truyền thẳng cuối. Hàm này chấp nhận một tham số bắt buộc sau là tham số tùy chọn thứ hai:

- X: các điểm dữ liệu mà sẽ dự đoán lớp nhãn.
- addBias: một boolean cho biết liệu chúng ta có cần thêm một cột từ 1 đến X để thực hiện thủ thuật bias hay không.

Dòng 131 khởi tạo p, dự đoán đầu ra là điểm dữ liệu đầu vào X. Giá trị p này sẽ được truyền qua mọi lớp trong mạng, lan truyền cho đến khi đạt được dự đoán đầu ra cuối cùng. Trên các dòng 134-137, thực hiện kiểm tra xem liệu có nên nhúng hệ số bias vào các điểm dữ liệu hay không. Nếu có, chèn một cột 1 chiều làm cột cuối cùng trong ma trận (chính xác như đã làm trong phương pháp ở trên).

Từ đó, thực hiện lan truyền thẳng bằng cách lặp qua tất cả các lớp trong mạng của tại Dòng 140. Các điểm dữ liệu p được cập nhật bằng cách lấy nhân chập giữa hàm kích hoạt hiện tại p và ma trận trọng số cho lớp hiện tại, sau đó truyền đến đầu ra thông qua hàm kích hoạt sigmoid của (Dòng 146).

Lặp lại qua tất cả các lớp trong mạng đến lớp cuối cùng sẽ cho dự đoán lớp nhãn cuối cùng. Trả lại giá trị dự đoán cho hàm được gọi tại Dòng 149.

Hàm cuối cùng được xác định bên trong lớp NeuralNetwork sẽ được sử dụng để tính toán tổn thất trong toàn bộ tập huấn luyện

```

151     def calculate_loss(self, X, targets):
152         # make predictions for the input data points then compute
153         # the loss
154         targets = np.atleast_2d(targets)
155         predictions = self.predict(X, addBias=False)
156         loss = 0.5 * np.sum((predictions - targets) ** 2)
157
158         # return the loss
159     return loss

```

Hàm `notify_loss` yêu cầu truyền vào các điểm dữ liệu X cùng với các nhãn, giá trị thực tế của chúng, đưa ra dự đoán về X trên Dòng 155 và sau đó tính toán tổng bình phương sai số tại Dòng 156. Tần suất sau đó được trả về hàm gọi tại dòng 159. Khi mạng huấn luyện, sẽ thấy tần suất này giảm.

1.1.4.5. Lan truyền ngược với Python - VÍ DỤ 1: BITWISE XOR

Bây giờ, sau khi đã triển khai lớp `NeuralNetwork`, hãy tiếp tục huấn luyện mạng trên bộ dữ liệu XOR bitwise. Như đã biết với Perceptron, bộ dữ liệu không thể phân biệt tuyến tính - mục tiêu là huấn luyện một mạng nơ-ron có thể mô hình hóa hàm phi tuyến này.

Hãy tiếp tục và mở một tệp mới, đặt tên là `nn_xor.py` và chèn chương trình sau đây:

```

1 # import the necessary packages
2 from pyimagesearch.nn import NeuralNetwork
3 import numpy as np
4
5 # construct the XOR dataset
6 X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
7 y = np.array([[0], [1], [1], [0]])

```

Dòng 2 và 3 nhập các gói Python cần thiết. Lưu ý cách nhập lớp `NeuralNetwork` để thực thi. Dòng 6 và 7 sau đó xây dựng bộ dữ liệu XOR, như được mô tả trong Bảng 1.1 trong chương này.

Bây giờ có thể định nghĩa kiến trúc mạng và huấn luyện:

```

9 # define our 2-2-1 neural network and train it
10 nn = NeuralNetwork([2, 2, 1], alpha=0.5)
11 nn.fit(X, y, epochs=20000)

```

Tại Dòng 10, khởi tạo `NeuralNetwork` để có kiến trúc 2 - 2 - 1, nghĩa là ở đó

1. Một lớp đầu vào có hai nút (tức là hai đầu vào).
2. Một lớp ẩn duy nhất với hai nút.

3. Một lớp đầu ra với một nút.

Dòng 11 huấn luyện mạng với tổng số 20.000 chu kỳ.

Khi mạng được huấn luyện, hãy lặp lại các bộ dữ liệu XOR, cho phép mạng dự đoán đầu ra cho từng dữ liệu tương ứng và hiển thị dự đoán trên màn hình:

```
13 # now that our network is trained, loop over the XOR data points
14 for (x, target) in zip(X, y):
15     # make a prediction on the data point and display the result
16     # to our console
17     pred = nn.predict(x)[0][0]
18     step = 1 if pred > 0.5 else 0
19     print("[INFO] data={}, ground-truth={}, pred={:.4f}, step={}".format(
20         x, target[0], pred, step))
```

Dòng 18 áp dụng hàm bước cho đầu ra sigmoid. Nếu dự đoán là > 0.5 , trả về một, ngược lại sẽ trả về số không. Áp dụng hàm bước này cho phép tạo nhãn nhị phân lớp đầu ra, giống như hàm XOR.

Để huấn luyện mạng nơ-ron bằng cách sử dụng lan truyền ngược với Python, chỉ cần thực hiện lệnh sau:

```
$ python nn_xor.py
[INFO] epoch=1, loss=0.5092796
[INFO] epoch=100, loss=0.4923591
[INFO] epoch=200, loss=0.4677865
...
[INFO] epoch=19800, loss=0.0002478
[INFO] epoch=19900, loss=0.0002465
[INFO] epoch=20000, loss=0.0002452
```

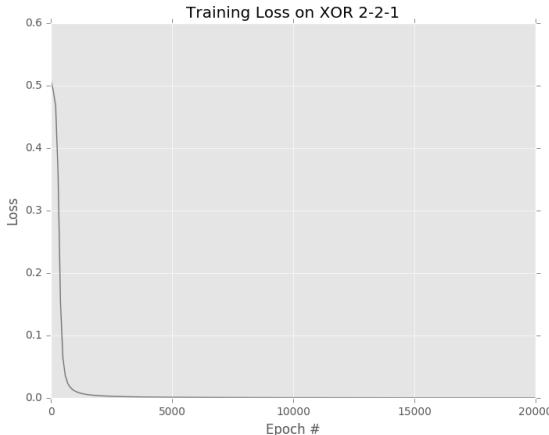
Một đồ thị hàm tổn thất bình phương được biểu diễn bên dưới (Hình 1.10). Như có thể thấy, tổn thất từ từ giảm xuống xấp xỉ 0 trong suốt quá trình huấn luyện. Hơn nữa, nhìn vào bốn dòng cuối cùng có thể được dự đoán:

```
[INFO] data=[0 0], ground-truth=0, pred=0.0054, step=0
[INFO] data=[0 1], ground-truth=1, pred=0.9894, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.9876, step=1
[INFO] data=[1 1], ground-truth=0, pred=0.0140, step=0
```

Đối với mỗi điểm dữ liệu, mạng nơ-ron có thể học chính xác mẫu XOR, chứng minh rằng mạng nơ-ron nhiều lớp có khả năng học các hàm phi tuyến.

Để chứng minh rằng ít nhất một lớp ẩn được yêu cầu để huấn luyện hàm XOR, hãy xem lại

Dòng 10 trong đó xác định kiến trúc 2 - 2 - 1:



Hình 1.10: Hàm tổn thất theo thời gian cho mạng nơ-ron 2 - 2 - 1.

```
10 # define our 2-2-1 neural network and train it
11 nn = NeuralNetwork([2, 2, 1], alpha=0.5)
12 nn.fit(X, y, epochs=20000)
```

Và thay đổi nó thành một kiến trúc 2-1:

```
10 define our 2-1 neural network and train it
11 nn = NeuralNetwork([2, 1], alpha=0.5)
12 nn.fit(X, y, epochs=20000)
```

Từ đó, huấn luyện lại mạng:

```
$ python nn_xor.py
...
[INFO] data=[0 0], ground-truth=0, pred=0.5161, step=1
[INFO] data=[0 1], ground-truth=1, pred=0.5000, step=1
[INFO] data=[1 0], ground-truth=1, pred=0.4839, step=0
[INFO] data=[1 1], ground-truth=0, pred=0.4678, step=0
```

Cho dù tốc độ học hoặc khởi tạo trọng số là bao nhiêu đi chăng nữa, sẽ không thể ước chừng hàm XOR. Thực tế này là lý do tại sao các mạng nhiều lớp có các hàm kích hoạt phi tuyến được huấn luyện thông qua lan truyền ngược rất quan trọng - chúng cho phép huấn luyện các mẫu trong các bộ dữ liệu có thể tách rời phi tuyến.

1.1.4.6. Lan truyền ngược với Python - VÍ DỤ 2: MẪU MNIST

Ví dụ hai, thực tế hơn, hãy kiểm tra một tập hợp con của bộ dữ liệu MNIST (Hình 1.11) để nhận dạng chữ số viết tay. Tập hợp con bộ dữ liệu MNIST này đã được tích hợp vào thư viện scikit-learn và bao gồm 1.797 chữ số mẫu, mỗi chữ số là ảnh xám kích thước 8×8 (ảnh gốc là 28×28).

Khi được làm phẳng, những ảnh này được thể hiện bằng $8 \times 8 = 64$ chiều vecto.



Hình 1.11: Một mẫu bộ dữ liệu MNIST. Mục tiêu bộ dữ liệu này là phân loại chính xác các chữ số viết tay, 0 - 9.

Hãy cùng đi trước và huấn luyện triển khai NeuralNetwork trên tập hợp con MNIST này ngay bây giờ. Mở một tệp mới, đặt tên là nn_mnist.py và hãy làm việc:

```
1 # import the necessary packages
2 from pyimagesearch.nn import NeuralNetwork
3 from sklearn.preprocessing import LabelBinarizer
4 from sklearn.model_selection import train_test_split
5 from sklearn.metrics import classification_report
6 from sklearn import datasets
```

Bắt đầu tại Dòng 2-6 bằng cách nhập các gói Python cần thiết

Từ đó, tải tập dữ liệu MNIST về ổ cứng bằng cách sử dụng các hàm trợ giúp scikit-learn:

```
8 # load the MNIST dataset and apply min/max scaling to scale the
9 # pixel intensity values to the range [0, 1] (each image is
10 # represented by an 8 x 8 = 64-dim feature vector)
11 print("[INFO] loading MNIST (sample) dataset...")
12 digits = datasets.load_digits()
13 data = digits.data.astype("float")
14 data = (data - data.min()) / (data.max() - data.min())
15 print("[INFO] samples: {}, dim: {}".format(data.shape[0],
16     data.shape[1]))
```

Thực hiện chuẩn hóa tối thiểu / tối đa bằng cách chia tỷ lệ từng chữ số vào phạm vi $[0, 1]$ (Dòng 14).

Tiếp theo, hãy xây dựng một bộ phân tách dữ liệu huấn luyện và thử nghiệm, sử dụng 75% dữ liệu để huấn luyện và 25% để đánh giá:

```
18 # construct the training and testing splits
19 (trainX, testX, trainY, testY) = train_test_split(data,
20     digits.target, test_size=0.25)
21
22 # convert the labels from integers to vectors
23 trainY = LabelBinarizer().fit_transform(trainY)
24 testY = LabelBinarizer().fit_transform(testY)
```

Hãy mã hóa các số nguyên lõp nhän dưới dạng vector, một quá trình gọi là mã hóa sẽ được đánh giá chi tiết trong chương này.

Từ đó, sẵn sàng huấn luyện mạng:

```
26 # train the network  
27 print("[INFO] training network...")  
28 nn = NeuralNetwork([trainX.shape[1], 32, 16, 10])  
29 print("[INFO] {}".format(nn))  
30 nn.fit(trainX, trainY, epochs=1000)
```

Ở đây có thể thấy rằng đang huấn luyện một NeuralNetwork với kiến trúc 64, 32, 16, 10. Lớp đầu ra có mười nút do thực tế là có mười lớp đầu ra có thể cho các chữ số 0-9.

Sau đó cho phép mạng huấn luyện 1.000 chu kỳ. Khi mạng đã được huấn luyện, có thể đánh giá nó trên bộ dữ liệu thử nghiệm:

```
32 # evaluate the network  
33 print("[INFO] evaluating network...")  
34 predictions = nn.predict(testX)  
35 predictions = predictions.argmax(axis=1)  
36 print(classification_report(testY.argmax(axis=1), predictions))
```

Dòng 34 tính toán các dự đoán đầu ra cho mọi điểm dữ liệu trong testX. Mảng dự đoán có hình dạng (450, 10) vì có 450 điểm dữ liệu trong tập kiểm tra, mỗi điểm có mươi xác suất nhän có thể.

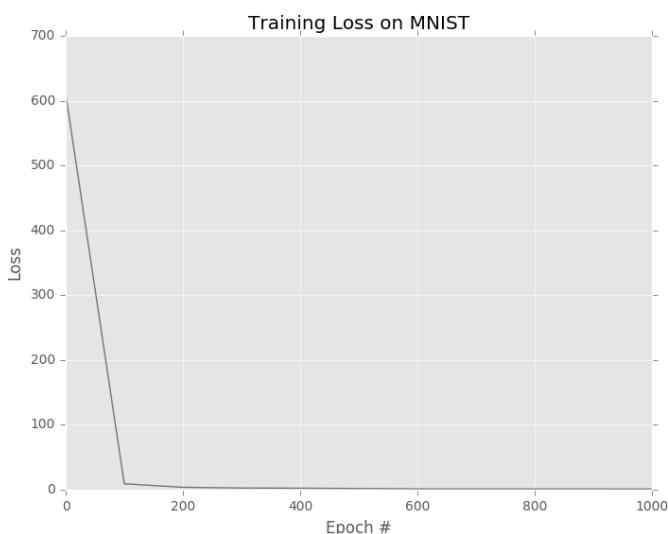
Để tìm nhän có xác suất lớn nhất cho mỗi điểm dữ liệu, sử dụng hàm argmax tại Dòng 35 - hàm này trả về chỉ mục nhän có xác suất dự đoán cao nhất. Sau đó, hiển thị báo cáo phân loại được định dạng trên màn hình tại Dòng 36.

Biểu diễn đồ thị bình phương sai số (Hình 1.12). Lưu ý rằng tồn thât bắt đầu rất cao, nhưng nhanh chóng giảm xuống trong quá trình huấn luyện. Báo cáo phân loại chứng minh rằng đạt được độ chính xác phân loại $\approx 98\%$ trên bộ dữ liệu thử nghiệm; Tuy nhiên, có khó khăn khi phân loại chữ số 4 và 5 (độ chính xác tương ứng 95% và 94%). Phần sau trong cuốn sách này, chúng ta sẽ học cách huấn luyện mạng nơ-ron tích hợp trên bộ dữ liệu đầy đủ MNIST và cải thiện độ chính xác hơn nữa.

Để huấn luyện triển khai NeuralNetwork tùy chỉnh trên bộ dữ liệu MNIST, chỉ cần thực hiện lệnh sau:

```
$python nn_mnist.py
[INFO] loading MNIST (sample) dataset...
[INFO] samples: 1797, dim: 64
[INFO] training network...
[INFO] NeuralNetwork: 64-32-16-10
[INFO] epoch=1, loss=604.5868589
[INFO] epoch=100, loss=9.1163376
[INFO] epoch=200, loss=3.7157723
[INFO] epoch=300, loss=2.6078803
[INFO] epoch=400, loss=2.3823153
[INFO] epoch=500, loss=1.8420944
[INFO] epoch=600, loss=1.3214138
[INFO] epoch=700, loss=1.2095033
[INFO] epoch=800, loss=1.1663942
[INFO] epoch=900, loss=1.1394731
[INFO] epoch=1000, loss=1.1203779
[INFO] evaluating network...
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	45
1	0.98	1.00	0.99	51
2	0.98	1.00	0.99	47
3	0.98	0.93	0.95	43
4	0.95	1.00	0.97	39
5	0.94	0.97	0.96	35
6	1.00	1.00	1.00	53
7	1.00	1.00	1.00	49
8	0.97	0.95	0.96	41
9	1.00	0.96	0.98	47
avg / total	0.98	0.98	0.98	450



Hình 1.12: Biểu diễn tổn thất huấn luyện trên tập dữ liệu MNIST bằng cách sử dụng mạng nơ ron 64 32 16 10.

1.1.4.7. Tóm tắt về lan truyền ngược

Trong phần này, chúng ta đã được học cách triển khai thuật toán lan truyền ngược từ đầu bằng Python. Lan truyền ngược là một họ khái quát các thuật toán gradient gốc được sử dụng đặc biệt để huấn luyện các mạng lan truyền thẳng nhiều lớp.

Thuật toán lan truyền ngược bao gồm hai giai đoạn:

1. Lan truyền thẳng – truyền các đầu vào mạng để có được các phân loại đầu ra.

2. Lan truyền ngược (nghĩa là giai đoạn cập nhật trọng số) trong đó tính toán độ dốc hàm tổn thất và sử dụng thông tin này để tạo vòng lặp áp dụng quy tắc chuỗi để cập nhật trọng số trong mạng.

Bất kể là chúng ta đang làm việc với các mạng nơ-ron đơn giản cho bộ tạo dữ liệu đơn giản hay mạng nơ-ron tích chập phức tạp, thuật toán lan truyền ngược vẫn được sử dụng để huấn luyện các mô hình này. Điều này được thực hiện bằng cách đảm bảo rằng các hàm kích hoạt bên trong mạng là khả vi, cho phép áp dụng quy tắc chuỗi. Hơn nữa, bất kỳ lớp nào khác trong mạng yêu cầu cập nhật trọng số/tham số, cũng phải phù hợp với lan truyền ngược.

Triển khai thuật toán lan truyền ngược bằng ngôn ngữ lập trình Python và đã tạo ra một mạng NeuralNetwork nhiều lớp. Quá trình này sau đó đã huấn luyện trên bộ dữ liệu XOR để chứng minh rằng mạng nơ-ron có khả năng học các hàm phi tuyến bằng cách áp dụng thuật toán lan truyền ngược với ít nhất một lớp ẩn. Sau đó, đã áp dụng cùng một triển khai lan truyền ngược + Python cho một tập hợp con bộ của dữ liệu MNIST để chứng minh rằng thuật toán cũng có thể được sử dụng để làm việc với dữ liệu ảnh.

Trong thực tế, lan truyền ngược không chỉ khó thực hiện (do lỗi trong việc tạo độ dốc), mà còn khó thực hiện hiệu quả nếu không có thư viện tối ưu hóa đặc biệt, đó là lý do tại sao thường sử dụng các thư viện như Keras, TensorFlow và mxnet vì trong chúng đã sử dụng các chiến lược tối ưu hóa trong lan truyền ngược.

1.1.5. Mạng nhiều lớp với Keras

Bây giờ, sau khi đã triển khai các mạng nơ-ron trong Python thuận tay, hãy chuyển sang phương thức triển khai được ưa thích - sử dụng thư viện mạng nơ-ron chuyên dụng (được tối ưu hóa cao) như Keras.

Hai phần tiếp theo sẽ thảo luận về cách triển khai các mạng

nhiều lớp, và áp dụng chúng cho các bộ dữ liệu MNIST và CIFAR-10. Những kết quả này tuy không phải là quá hiện đại, nhưng phục vụ hai mục đích:

- Giải thích cách có thể triển khai các mạng nơ-ron đơn giản bằng thư viện Keras.
- Hiểu được cơ sở sử dụng các mạng nơ-ron tiêu chuẩn mà sau này sẽ so sánh với mạng nơ-ron truyền thống (lưu ý rằng CNNs sẽ vượt trội hơn hẳn các phương pháp trước đây).

1.1.5.1. MNIST

Mục 1.1.3 ở trên, chỉ sử dụng một tập con của bộ dữ liệu MNIST vì hai lý do:

Để giải thích cách triển khai mạng nơ-ron lan truyền trong Python. Để tạo điều kiện thu thập kết quả nhanh hơn - với điều kiện là việc triển khai Python thuận túy theo định nghĩa là không được tối ưu hóa, nên sẽ mất nhiều thời gian hơn để chạy.

Do đó, chỉ sử dụng một tập con của bộ dữ liệu. Trong phần này, hãy sử dụng toàn bộ dữ liệu MNIST, bao gồm 70.000 điểm dữ liệu (7.000 ví dụ cho mỗi chữ số). Mỗi điểm dữ liệu được biểu thị bằng một vectơ 784-chiều, tương ứng với ảnh (kích thước 28 đã được làm phẳng) 28×28 trong bộ dữ liệu MNIST. Mục tiêu là huấn luyện một mạng nơ-ron (sử dụng Keras) để có được độ chính xác $> 90\%$ trên bộ dữ liệu này.

Sử dụng Keras để xây dựng kiến trúc mạng dễ dàng hơn phiên bản Python thuận túy. Trong thực tế, kiến trúc mạng thực tế chỉ tốn bốn dòng chương trình - phần còn lại chương trình trong ví dụ này chỉ đơn giản là tải dữ liệu về ổ cứng, chuyển đổi lớp nhãn và sau đó hiển thị kết quả.

Để bắt đầu, hãy mở một tệp mới, đặt tên là `keras_mnist.py` và chèn chương trình sau đây:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import classification_report
5 from keras.models import Sequential
6 from keras.layers.core import Dense
7 from keras.optimizers import SGD
8 from sklearn import datasets
9 import matplotlib.pyplot as plt
10 import numpy as np
11 import argparse
```

Dòng 2-11 nhập các gói Python cần thiết. LabelBinarizer được sử dụng để mã hóa một nhãn số nguyên dưới dạng nhãn vector. Mã hóa biến đổi các nhãn phân loại từ một số nguyên thành một vecto. Nhiều thuật toán học máy (bao gồm cả mạng nơ-ron) sẽ dễ dàng hơn khi biểu diễn nhãn này. Trong phần sau, chúng ta sẽ thảo luận về mã hóa chi tiết hơn và thực hiện một số ví dụ (bao gồm cả việc sử dụng LabelBinarizer) trong phần sau.

Train_test_split tại Dòng 3 sẽ được sử dụng để tạo các phân tách dữ liệu huấn luyện và dữ liệu thử nghiệm từ bộ dữ liệu MNIST. Hàm class_Vport hãy cung cấp cho một báo cáo được định dạng cụ thể hiển thị tổng độ chính xác của mô hình và về độ chính xác phân loại cho mỗi chữ số.

Các dòng 5-7 nhập các gói cần thiết để tạo một mạng nơ-ron lan truyền dữ liệu đơn giản với Keras. Lớp Sequential chỉ ra rằng mạng được cấp dữ liệu và các lớp sẽ được thêm vào lớp một cách tuần tự, lớp này nằm trên lớp kia. Lớp Dense tại Dòng 6 việc thực hiện các lớp fully-connected (FC-fully connected). Để mạng thực sự được huấn luyện cần áp dụng thuật toán SGD (Dòng 7) để tối ưu hóa các tham số mạng. Cuối cùng, để có quyền truy cập vào bộ dữ liệu MNIST một cách đầy đủ, cần nhập trình trợ giúp bộ dữ liệu từ scikit-learn trên Dòng 8.

Hãy chuyển sang phân tích cú pháp đối số tại dòng lệnh:

```
13 # construct the argument parse and parse the arguments
14 ap = argparse.ArgumentParser()
15 ap.add_argument("-o", "--output", required=True,
16     help="path to the output loss/accuracy plot")
17 args = vars(ap.parse_args())
```

Chỉ cần một tham số duy nhất ở đây, --output, là đường dẫn đến nơi mà đồ thị hàm tổn thất và độ chính xác theo thời gian sẽ được lưu vào ổ cứng. Tiếp theo, hãy tải dữ liệu đầy đủ bộ dữ liệu MNIST:

```
19 # grab the MNIST dataset (if this is your first time running this
20 # script, the download may take a minute -- the 55MB MNIST dataset
21 # will be downloaded)
22 print("[INFO] loading MNIST (full) dataset...")
23 dataset = datasets.fetch_mldata("MNIST Original")
24
25 # scale the raw pixel intensities to the range [0, 1.0], then
26 # construct the training and testing splits
27 data = dataset.data.astype("float") / 255.0
28 (trainX, testX, trainY, testY) = train_test_split(data,
29     dataset.target, test_size=0.25)
```

Dòng 23 tải tập dữ liệu MNIST về ổ cứng. Nếu chưa bao giờ chạy hàm này trước đó, thì bộ dữ liệu MNIST cần được tải xuống và lưu trữ cục bộ vào máy – dung lượng tải xuống là 55 MB và có thể mất một hoặc hai

phút để hoàn tất tải xuống, tùy thuộc vào kết nối internet. Khi tập dữ liệu đã được tải xuống, nó cần được lưu vào bộ nhớ cache và hãy không phải tải xuống nữa.

Sau đó, thực hiện chuẩn hóa dữ liệu tại Dòng 27 bằng cách chia tỷ lệ cường độ pixel thành phạm vi $[0, 1]$. Tạo bộ phân tách dữ liệu huấn luyện và dữ liệu thử nghiệm, sử dụng 75% dữ liệu cho huấn luyện và 25% cho thử nghiệm tại Dòng 28 và 29.

Với các dữ liệu phân tách huấn luyện và dữ liệu thử nghiệm, giờ đây có thể mã hóa nhãn:

```
31 # convert the labels from integers to vectors
32 lb = LabelBinarizer()
33 trainY = lb.fit_transform(trainY)
34 testY = lb.transform(testY)
```

Mỗi điểm dữ liệu trong bộ dữ liệu MNIST có nhãn là số nguyên trong phạm vi $[0, 9]$, một điểm cho một chữ số có thể có trong mươi số của bộ dữ liệu MNIST. Một nhãn có giá trị 0 chỉ ra rằng ảnh tương ứng chứa một chữ số 0. Tương tự, nhãn có giá trị 8 chỉ ra rằng ảnh tương ứng chứa số 8.

Tuy nhiên, trước tiên cần chuyển đổi các nhãn số nguyên này thành nhãn vector, trong đó chỉ mục trong vectơ cho nhãn được đặt thành 1 và còn lại được đặt thành 0..

Ví dụ, hãy xem xét nhãn số 3 và mã hóa nhị phân/mã hóa nó. Nhãn số 3 giờ trở thành:

```
[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
```

Lưu ý cách số chỉ cho chữ số ba được đặt thành 1t - tất cả các mục khác trong vectơ được đặt thành 0. Độc giả có thể tự hỏi tại sao mục thứ tư và không phải là mục thứ ba trong vector được cập nhật? Hãy nhớ lại rằng mục đầu tiên trong nhãn thực sự là chữ số 0. Do đó, mục nhập cho chữ số ba thực sự là chỉ số thứ tư trong danh sách.

Đây là ví dụ thứ hai, lần này với nhãn chữ số 1 nhị phân:

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
```

Mục nhập thứ hai trong vectơ gán thành một (vì mục nhập đầu tiên tương ứng với nhãn 0), trong khi tất cả các mục nhập khác được gán thành 0. Danh sách dưới đây bao gồm các đại diện mã hóa cho mỗi chữ số, 0 - 9:

```
0: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

Mã hóa này có vẻ đơn giản, nhưng nhiều thuật toán học máy (bao gồm cả mạng nơ-ron), sẽ hiệu quả hơn với biểu diễn nhãn kiểu này. May mắn thay, hầu hết các gói phần mềm học máy đều cung cấp một phương thức / hàm để thực hiện mã hóa, loại bỏ phần lớn công việc này.

Các dòng 32-34 chỉ đơn giản thực hiện quy trình mã hóa này cho các nhãn số nguyên đầu vào dưới dạng nhãn vector cho cả tập huấn luyện và tập kiểm tra.

Tiếp theo, hãy định nghĩa kiến trúc mạng:

```
36 # define the 784-256-128-10 architecture using Keras
37 model = Sequential()
38 model.add(Dense(256, input_shape=(784,), activation="sigmoid"))
39 model.add(Dense(128, activation="sigmoid"))
40 model.add(Dense(10, activation="softmax"))
```

Như có thể thấy, mạng là một kiến trúc nối tiếp, được khởi tạo bởi lớp Sequential tại Dòng 37 - kiến trúc này mang ý nghĩa là các lớp sẽ được xếp chồng lên nhau với đầu ra lớp trước được đưa vào lớp kế tiếp.

Dòng 38 định nghĩa lớp đầu tiên fully-connected trong mạng. Input_shape được đặt gán 784, là số chiều từng điểm dữ liệu MNIST. Sau đó học 256 trọng số trong lớp này và áp dụng hàm kích hoạt sigmoid. Lớp tiếp theo (Dòng 39) học 128 trọng số. Cuối cùng, Dòng 40 áp dụng một lớp fully-connected (FC) khác, lần này chỉ học 10 trọng số, tương ứng với mươi (0-9) lớp đầu ra. Thay vì sử dụng hàm kích hoạt sigmoid, hãy sử dụng hàm kích hoạt softmax để có được xác suất lớp chuẩn hóa cho mỗi dự đoán.

Huấn luyện mạng:

```
42 # train the model using SGD
43 print("[INFO] training network...")
44 sgd = SGD(0.01)
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46                 metrics=["accuracy"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48                 epochs=100, batch_size=128)
```

Tại Dòng 44, khởi tạo trình tối ưu hóa SGD với tốc độ học là 0,01 (mà thường có thể viết là 1e-2). Sử dụng hàm tổn thất entropy chéo làm tham số tổn thất (Dòng 45 và 46). Việc sử dụng hàm tổn thất entropy chéo cũng là lý do tại sao phải chuyển đổi nhãn số nguyên thành nhãn vector.

Một hàm gọi đến `model.fit` trên Dòng 47 và 48 khởi động việc huấn luyện mạng nơ-ron. Truyền dữ liệu huấn luyện và nhãn huấn luyện như hai đối số đầu tiên cho phương thức.

Xác nhận `_data` sau đó có thể được truyền đến, đó là phần thử nghiệm. Trong hầu hết các trường hợp, chẳng hạn như khi điều chỉnh siêu thông số hoặc quyết định kiến trúc mô hình, sẽ phải có bộ xác thực đúng chứ không phải dữ liệu thử nghiệm. Trong trường hợp này, chỉ trình bày cách huấn luyện một mạng nơ-ron từ đầu bằng cách sử dụng Keras nên các hướng dẫn sẽ đơn giản. Các chương sau trong cuốn sách này, sẽ trình bày rõ ràng hơn các nội dung nâng cao hơn trong phần thực hành. Tuy nhiên, hiện tại chúng ta chỉ cần tập trung vào chương trình và nắm bắt cách mà mạng được huấn luyện.

Hãy cho phép mạng huấn luyện tổng cộng 100 chu kỳ bằng cách sử dụng kích thước khoảng 128 điểm dữ liệu cùng một lúc. Phương thức trả về một dạng từ điển, `H`, sử dụng để biểu hàm tổn thất/ độ chính xác của mạng theo thời gian trong một vài khối chương trình.

Khi mạng đã hoàn thành huấn luyện, sẽ cần đánh giá trên dữ liệu thử nghiệm để có được các phân loại cuối cùng:

```
50 # evaluate the network
51 print("[INFO] evaluating network...")
52 predictions = model.predict(testX, batch_size=128)
53 print(classification_report(testY.argmax(axis=1),
54     predictions.argmax(axis=1),
55     target_names=[str(x) for x in lb.classes_]))
```

Một hàm gọi đến mô hình phương thức `predict` trả về xác suất lớp nhãn cho mọi điểm dữ liệu test X (Dòng 52). Do đó, nếu kiểm tra dự đoán mảng NumPy sẽ có dạng (`X, 10`) vì có 17.500 điểm dữ liệu trong tập kiểm tra và mười lớp nhãn có thể (các chữ số 0-9).

Mỗi kênh truyền trong một hàng nhất định là một xác suất. Để xác định lớp có xác suất lớn nhất, chỉ cần gọi `argmax` (trục = 1) như làm tại Dòng 53, sẽ truyền đến chỉ số lớp nhãn với xác suất lớn nhất và phân loại đầu ra cuối cùng. Phân loại đầu ra cuối cùng theo mạng được lập bảng và sau đó báo cáo phân loại cuối cùng được hiển thị cho bảng điều khiển tại Dòng 53-55.

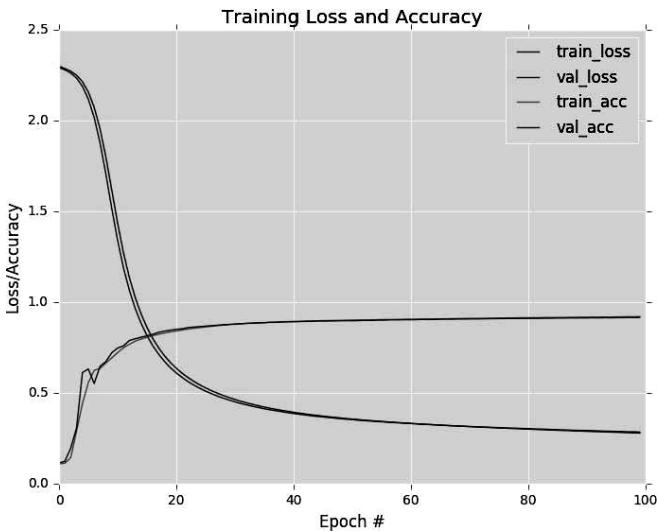
Khối chương trình cuối xử lý đồ thị tồn thât của mạng huấn luyện, độ chính xác huấn luyện, tồn thât xác thực (loss validation) và độ chính xác xác thực (accuracy validation) theo thời gian:

```
57 # plot the training loss and accuracy
58 plt.style.use("ggplot")
59 plt.figure()
60 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
61 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
62 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
63 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
64 plt.title("Training Loss and Accuracy")
65 plt.xlabel("Epoch #")
66 plt.ylabel("Loss/Accuracy")
67 plt.legend()
68 plt.savefig(args["output"])
```

Biểu đồ này sau đó được lưu vào ô cứng dựa trên đối số dòng lệnh --output.

Để huấn luyện mạng các lớp kết nối đầy đủ (FC) trên MNIST, chỉ cần thực hiện lệnh sau:

```
$ python keras_mnist.py --output output/keras_mnist.png
[INFO] loading MNIST (full) dataset...
[INFO] training network...
Train on 52500 samples, validate on 17500 samples
Epoch 1/100
  1s - loss: 2.2997 - acc: 0.1088 - val_loss: 2.2918 - val_acc: 0.1145
Epoch 2/100
  1s - loss: 2.2866 - acc: 0.1133 - val_loss: 2.2796 - val_acc: 0.1233
Epoch 3/100
  1s - loss: 2.2721 - acc: 0.1437 - val_loss: 2.2620 - val_acc: 0.1962
...
Epoch 98/100
  1s - loss: 0.2811 - acc: 0.9199 - val_loss: 0.2857 - val_acc: 0.9153
Epoch 99/100
  1s - loss: 0.2802 - acc: 0.9201 - val_loss: 0.2862 - val_acc: 0.9148
Epoch 100/100
  1s - loss: 0.2792 - acc: 0.9204 - val_loss: 0.2844 - val_acc: 0.9160
[INFO] evaluating network...
              precision      recall   f1-score   support
  0.0          0.94      0.96      0.95      1726
  1.0          0.95      0.97      0.96      2004
  2.0          0.91      0.89      0.90      1747
  3.0          0.91      0.88      0.89      1828
  4.0          0.91      0.93      0.92      1686
  5.0          0.89      0.86      0.88      1581
  6.0          0.92      0.96      0.94      1700
  7.0          0.92      0.94      0.93      1814
  8.0          0.88      0.88      0.88      1679
  9.0          0.90      0.88      0.89      1735
avg / total       0.92      0.92      0.92     17500
```



Hình 1.13: Huấn luyện mạng nơ-ron 784, 256, 128, 10 với Keras trên bộ dữ liệu đầy đủ MNIST. Lưu ý cách các đường cong tập huấn luyện và tập xác thực gần giống nhau nghĩa là không có hiện tượng quá khớp xảy ra.

Kết quả đã đạt được độ chính xác 92%. Hơn nữa, các đường cong huấn luyện và xác thực khớp với nhau gần như giống hệt nhau (Hình 1.13), cho thấy không có vấn đề hiện tượng quá khớp hoặc các vấn đề của quá trình huấn luyện.

Trên thực tế, nếu chưa làm quen với bộ dữ liệu MNIST, có thể cho rằng độ chính xác 92% là tuyệt vời (suy nghĩ của 20 năm trước). Tuy nhiên, Chương 3 sẽ cho thấy khi sử dụng mạng nơ-ron tích chập, có thể dễ dàng đạt được độ chính xác $> 98\%$. Các phương pháp hiện đại thậm chí có thể đưa độ chính xác lên 99%.

Mặc dù trên bề mặt có vẻ như mạng (fully-connected) đang hoạt động tốt, nhưng thực sự có thể làm tốt hơn nhiều. Trong phần tiếp theo, các mạng fully-connected (FC) được áp dụng cho các bộ dữ liệu khó khăn hơn trong một số trường hợp và hoàn toàn có thể làm tốt hơn là các bộ dự đoán ngẫu nhiên.

1.1.5.2. CIFAR-10

Khi nói đến thị giác máy tính và học máy, bộ dữ liệu MNIST là định nghĩa cổ điển của bộ dữ liệu tiêu chuẩn, một bộ quá đơn giản để có được kết quả chính xác cao và không đại diện cho ảnh trong thế giới thực.

Đối với một bộ dữ liệu chuẩn khó hơn, thường sử dụng CIFAR-10, một bộ sưu tập 60.000, 32×32 ảnh RGB, đó nghĩa là mỗi ảnh trong bộ

dữ liệu được đại diện bởi $32 \times 32 \times 3 = 3072$ số nguyên. Như tên của nó, CIFAR-10 bao gồm 10 lớp, bao gồm máy bay, ô tô, chim, mèo, hươu, chó, éch, ngựa, tàu, và xe tải. Một mẫu bộ dữ liệu CIFAR-10 cho mỗi lớp có thể được quan sát trong Hình 1.14.



Hình 1.14: *Ảnh ví dụ từ bộ dữ liệu mười lớp CIFAR-10.*

Mỗi lớp được biểu diễn đồng đều với 6.000 ảnh mỗi lớp. Khi huấn luyện và đánh giá mô hình học máy trên CIFAR-10, sử dụng cách phân chia dữ liệu được xác định trước bởi các tác giả và sử dụng 50.000 ảnh để huấn luyện và 10.000 ảnh để thử nghiệm.

CIFAR-10 có độ khó hơn đáng kể so với bộ dữ liệu MNIST. Thách thức đến từ sự khác biệt lớn trong cách các vật thể xuất hiện. Ví dụ: không còn có thể giả sử rằng một ảnh chứa pixel màu xanh lá cây tại một vị trí (x, y) nhất định là một con éch. Pixel này có thể là nền một khu rừng chứa một con nai. Hoặc nó có thể là màu một chiếc xe hơi hoặc xe tải màu xanh lá cây.

Các giả định này trái ngược hoàn toàn với bộ dữ liệu MNIST, nơi mạng có thể tìm hiểu các giả định liên quan đến phân bố không gian cường độ pixel. Ví dụ, sự phân tán không gian các pixel phía trước 1 khác biệt đáng kể so với 0 hoặc 5. Loại phuơng sai này trong diện mạo đối tượng làm cho việc áp dụng một loạt các lớp fully-connected khó khăn hơn nhiều. Chúng ta sẽ tìm hiểu trong phần còn lại của chương này, các mạng lớp tiêu chuẩn FC (fully-connected) không phù hợp với loại phân loại ảnh này.

Hãy bắt đầu mở một tệp mới, đặt tên là keras_cifar10.py và chèn đoạn chương trình sau đây:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from sklearn.metrics import classification_report
4 from keras.models import Sequential
5 from keras.layers.core import Dense
6 from keras.optimizers import SGD
7 from keras.datasets import cifar10
8 import matplotlib.pyplot as plt
9 import numpy as np
10 import argparse
```

Các dòng 2-10 nhập các gói Python cần thiết để xây dựng mạng fully-connected, tương tự như với MNIST. Sự khác biệt là hàm thực thi đặc biệt tại Dòng 7 - vì CIFAR-10 là một bộ dữ liệu phổ biến mà các nhà nghiên cứu đánh giá là các thuật toán học máy và học sâu, nên các thư viện học sâu thường cung cấp các hàm hỗ trợ đơn giản để tự động tải bộ dữ liệu này về ổ cứng.

Tiếp theo, có thể phân tích các đối số trong dòng lệnh:

```
16 # construct the argument parse and parse the arguments
17 ap = argparse.ArgumentParser()
18 ap.add_argument("-o", "--output", required=True,
19     help="path to the output loss/accuracy plot")
20 args = vars(ap.parse_args())
```

Đối số dòng chương trình duy nhất cần là --output, đường dẫn đến biểu đồ tồn thât/độ chính xác đầu ra.

Tiếp theo tải bộ dữ liệu CIFAR-10:

```
18 # load the training and testing data, scale it into the range [0, 1],
19 # then reshape the design matrix
20 print("[INFO] loading CIFAR-10 data...")
21 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
22 trainX = trainX.astype("float") / 255.0
23 testX = testX.astype("float") / 255.0
24 trainX = trainX.reshape((trainX.shape[0], 3072))
25 testX = testX.reshape((testX.shape[0], 3072))
```

Một hàm gọi đến cifar10.load_data tại Dòng 21 sẽ tự động tải bộ dữ liệu CIFAR-10 về ổ cứng, được phân đoạn trước thành tập huấn luyện và tập thử nghiệm. Nếu đây là lần đầu tiên gọi cifar10.load_data, thì hàm này sẽ tìm nạp và tải xuống tập dữ liệu. Dung lượng tập tin này là 170 MB, vì vậy hãy kiên nhẫn khi nó tải xuống và chưa được lưu trữ. Khi tệp được tải xuống một lần, nó sẽ được lưu trữ cục bộ trên máy và sẽ không phải tải xuống lại.

Các dòng 22 và 23 chuyển đổi kiểu dữ liệu CIFAR-10 từ các số nguyên 8 bit không dấu sang số thực, tiếp theo là chia tỷ lệ dữ liệu thành phạm vi [0, 1]. Dòng 24 và 25 chịu trách nhiệm định hình lại ma trận thiết kế cho dữ liệu huấn luyện và kiểm tra. Hãy nhớ lại rằng mỗi ảnh trong bộ dữ liệu CIFAR-10 được thể hiện bằng ảnh $32 \times 32 \times 3$.

Ví dụ, trainX có hình dạng (50000, 32, 32, 3) và testX có hình dạng (10000, 32, 32, 3). Nếu làm phẳng ảnh này thành một danh sách các giá trị floating point, danh sách sẽ có tổng cộng $32 \times 32 \times 3 = 3,072$ tổng số mục trong đó.

Để làm phẳng từng ảnh trong các bộ huấn luyện và thử nghiệm, chỉ cần sử dụng hàm .reshape NumPy. Sau khi hàm này thực thi, trainX hiện có hình dạng (50000, 3072) trong khi testX có hình dạng (10000, 3072).

Bây giờ, bộ dữ liệu CIFAR-10 đã được tải về ổ cứng, hãy một lần nữa nhị phân các số nguyên lớp nhãn thành các vector, sau đó khởi tạo một danh sách các tên thực lớp nhãn:

```
27 # convert the labels from integers to vectors
28 lb = LabelBinarizer()
29 trainY = lb.fit_transform(trainY)
30 testY = lb.transform(testY)
31
32 # initialize the label names for the CIFAR-10 dataset
33 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
34 "dog", "fog", "horse", "ship", "truck"]
```

Bây giờ, định nghĩa kiến trúc mạng:

```
36 # define the 3072-1024-512-10 architecture using Keras
37 model = Sequential()
38 model.add(Dense(1024, input_shape=(3072,), activation="relu"))
39 model.add(Dense(512, activation="relu"))
40 model.add(Dense(10, activation="softmax"))
```

Dòng 37 khởi tạo lớp Sequential. Sau đó, thêm lớp Dense đầu tiên có input_shape là 3072, một nút cho mỗi trọng số 3.072 giá trị pixel được làm phẳng trong ma trận thiết kế - lớp này sau đó chịu trách nhiệm học 1.024 trọng số, hãy đổi hàm kích hoạt sigmoid thành ReLU với hy vọng cải thiện hiệu suất mạng.

Lớp fully-connected tiếp theo (Dòng 39) học 512 trọng số, trong khi lớp cuối cùng (Dòng 40) học các trọng số tương ứng với 10 phân loại đầu

ra có thể, cùng với phân loại softmax để có được xác suất đầu ra cho mỗi lớp cuối cùng.

Bây giờ kiến trúc mạng đã được định nghĩa, có thể huấn luyện:

```
42 # train the model using SGD
43 print("[INFO] training network...")
44 sgd = SGD(0.01)
45 model.compile(loss="categorical_crossentropy", optimizer=sgd,
46     metrics=["accuracy"])
47 H = model.fit(trainX, trainY, validation_data=(testX, testY),
48     epochs=100, batch_size=32)
```

Sử dụng trình tối ưu hóa SGD để huấn luyện mạng với tốc độ học là 0,01, một lựa chọn ban đầu khá chuẩn. Mạng sẽ được huấn luyện cho tổng số 100 epoch sử dụng các đợt 32.

Khi mạng đã được huấn luyện, có thể đánh giá nó bằng cách sử dụng classification_report để có được bảng đánh giá chi tiết hơn về hiệu suất mô hình:

```
50 # evaluate the network
51 print("[INFO] evaluating network...")
52 predictions = model.predict(testX, batch_size=32)
53 print(classification_report(testY.argmax(axis=1),
54     predictions.argmax(axis=1), target_names=labelNames))
```

Và cuối cùng, biểu diễn đồ thị tồn thắt / độ chính xác theo thời gian:

```
56 # plot the training loss and accuracy
57 plt.style.use("ggplot")
58 plt.figure()
59 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
60 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
61 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
62 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
63 plt.title("Training Loss and Accuracy")
64 plt.xlabel("Epoch #")
65 plt.ylabel("Loss/Accuracy")
66 plt.legend()
67 plt.savefig(args["output"])
```

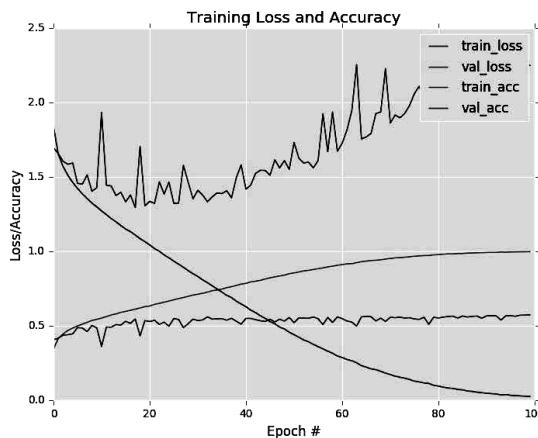
Để huấn luyện mạng trên CIFAR-10, hãy mở khung terminal và thực hiện lệnh sau:

```

$python keras cifar10.py --output output/keras cifar10.png
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
7s - loss: 1.8409 - acc: 0.3428 - val_loss: 1.6965 - val_acc: 0.4070
Epoch 2/100
7s - loss: 1.6537 - acc: 0.4160 - val_loss: 1.6561 - val_acc: 0.4163
Epoch 3/100
7s - loss: 1.5701 - acc: 0.4449 - val_loss: 1.6049 - val_acc: 0.4376
...
Epoch 98/100
7s - loss: 0.0292 - acc: 0.9969 - val_loss: 2.2477 - val_acc: 0.5712
Epoch 99/100
7s - loss: 0.0272 - acc: 0.9972 - val_loss: 2.2514 - val_acc: 0.5717
Epoch 100/100
7s - loss: 0.0252 - acc: 0.9976 - val_loss: 2.2492 - val_acc: 0.5739
[INFO] evaluating network...
      precision    recall   f1-score   support
airplane       0.63     0.66     0.64      1000
automobile     0.69     0.65     0.67      1000
bird          0.48     0.43     0.45      1000
cat           0.40     0.38     0.39      1000
deer          0.52     0.51     0.51      1000
dog           0.48     0.47     0.48      1000
frog          0.64     0.63     0.64      1000
horse         0.63     0.62     0.63      1000
ship          0.64     0.74     0.69      1000
truck         0.59     0.65     0.62      1000
avg / total    0.57     0.57     0.57     10000

```

Nhìn vào đầu ra, có thể thấy rằng mạng thu được độ chính xác 57%. Xem xét đồ thị tổn thất và độ chính xác theo thời gian (Hình 1.15), có thể thấy rằng mạng bị hiện tượng quá khớp từ chu kỳ 10. Tổn thất ban đầu bắt đầu giảm, giảm một chút, sau đó là phân kỳ, và không bao giờ giảm nữa.



Hình 1.15: Sử dụng một mạng nơ ron lan truyền ngược tiêu chuẩn dẫn đến việc hiện tượng quá khớp đáng kể trong bộ dữ liệu CIFAR-10 dày đặc thíc hơn (chú ý cách giảm tổn thất huấn luyện trong khi tổn thất xác thực tăng đáng kể). Để thành công với bộ dữ liệu CIFAR-10, hãy cần một kỹ thuật mạnh mẽ - mạng nơ-ron tích chập.

Việc giảm tổn thất huấn luyện trong khi tổn thất xác thực tăng là dấu hiệu của hiện tượng quá khớp.

Có thể xem xét tối ưu hóa siêu tham số để cải thiện chất lượng, đặc biệt là thử nghiệm với các giá trị tham số tốc độ học khác nhau và thay đổi cả độ sâu và số lượng nút trong mạng. Nhưng sẽ rất khó khăn và kết quả cải thiện không quá đáng kể. Thực tế các mạng tiếp liệu cơ bản với các lớp fully-connected không phù hợp với các bộ dữ liệu ảnh khó. Để làm được điều này, cần một cách tiếp cận tiên tiến hơn: mạng nơ-ron tích chập. Toàn bộ phần còn lại cuốn sách này sẽ trình bày về CNN. Khi hoàn thành phần khởi đầu, sẽ có thể đạt được độ chính xác hơn 79% trên CIFAR-10. Nếu chọn học sâu chuyên sâu hơn, học viên sẽ được học cách làm tăng độ chính xác lên hơn 93%, đưa vào các mạng các kết quả tốt nhất (state-of-the-art) [110].

1.1.6. Bốn thành phần trong công thức mạng nơ-ron

Có thể đã nhận thấy rằng, một số mẫu trong các chương trình Python ví dụ khi huấn luyện các mạng nơ-ron có bốn thành phần chính cần để kết hợp mạng nơ-ron và thuật toán học sâu, đó là: bộ dữ liệu, mô hình / kiến trúc, hàm tổn thất và phương pháp tối ưu hóa. Hãy xem xét từng thành phần dưới đây.

1.1.6.1. Bộ dữ liệu

Bộ dữ liệu là thành phần đầu tiên trong việc huấn luyện một mạng nơ-ron - chính dữ liệu cùng với vấn đề cần cố gắng giải quyết sẽ xác định mục tiêu cuối cùng. Ví dụ, có đang sử dụng các mạng nơ-ron để thực hiện phân tích hồi quy để dự đoán giá trị các ngôi nhà ở một vùng ngoại ô cụ thể trong 20 năm không? Là mục tiêu để thực hiện học không giám sát, chẳng hạn như giảm chiều? Hay đang cố gắng thực hiện phân loại?

Cuốn sách này trình bày về phân loại ảnh, tuy nhiên, sự kết hợp tập dữ liệu và vấn đề đang cố gắng giải quyết ảnh hưởng đến sự lựa chọn trong hàm tổn thất, kiến trúc mạng và phương pháp tối ưu hóa được sử dụng để huấn luyện mô hình. Thông thường, có ít sự lựa chọn trong tập dữ liệu (trừ khi làm việc trong một dự án tùy thích - được cung cấp một bộ dữ liệu với một số kỳ vọng về kết quả từ dự án). Sau khi đã lựa chọn tập dữ liệu, hãy huấn luyện một mô hình học máy trên bộ dữ liệu để thực hiện tốt nhiệm vụ đã cho.

1.1.6.2. Hàm tổn thất

Dựa vào dữ liệu và mục tiêu mong muốn, cần xác định hàm tổn thất phù hợp với vấn đề đang cố gắng giải quyết. Trong gần như tất cả các vấn

để phân loại ảnh bằng cách học sâu, hãy sử dụng tốn thất entropy chéo. Đối với số lớp > 2 , gọi đây là phân loại entropy chéo. Đối với bài toán hai lớp, gọi là entropy tốn thất nhị phân.

1.1.6.3. Mô hình / kiến trúc

Kiến trúc mạng có thể được coi là sự lựa chọn thực tế đầu tiên cần phải có. Tập dữ liệu có khả năng được chọn (hoặc ít nhất là đã quyết định rằng muốn làm việc với một tập dữ liệu đã cho). Và nếu đang thực hiện phân loại, rất có thể sẽ sử dụng entropy chéo làm hàm tốn thất.

Tuy nhiên, kiến trúc mạng có thể thay đổi đáng kể, đặc biệt là sau khi chọn phương pháp tối ưu hóa nào để huấn luyện mạng. Cần dành thời gian để khám phá tập dữ liệu và xem xét:

1. Có bao nhiêu nút dữ liệu.
2. Số lớp.
3. Sự tương tự / không giống nhau các lớp.
4. Phương sai trong lớp.

Bắt đầu phát triển “cảm giác” về một kiến trúc mạng được sử dụng. Thực tế học sâu bao gồm một phần là của khoa học, một phần của nghệ thuật. Phần sau của cuốn sách này được dành riêng để giúp phát triển cả hai kỹ năng này.

Hãy nhớ rằng số lượng lớp và nút trong kiến trúc mạng (cùng với bất kỳ loại mạng chính quy nào) có thể thay đổi khi thực hiện các thử nghiệm. Càng thu thập được nhiều kết quả, càng được trang bị tốt hơn để đưa ra quyết định sáng suốt về những kỹ thuật sẽ làm tiếp theo.

1.1.6.4. Phương pháp tối ưu hóa

Thành phần cuối cùng là xác định một phương pháp tối ưu hóa. Phương pháp Stochastic Gradient Descent (mục 8.2) được sử dụng khá thường xuyên. Các phương pháp tối ưu hóa khác, bao gồm RMSprop [22], Adagrad [23], Adadelta [24] và Adam [25], là những phương pháp tối ưu hóa nâng cao hơn, sẽ được đề cập trong phần thực hành.

Tuy nhiên, SGD vẫn là phần cốt thiết trong học sâu- hầu hết các mạng nơ-ron được huấn luyện thông qua SGD, bao gồm các mạng có độ chính xác cao trên các bộ dữ liệu ảnh khó, ví dụ như ImageNet.

Khi huấn luyện mạng học sâu, đặc biệt là khi bắt đầu học, SGD nên là lựa chọn tối ưu. Sau đó, cần đặt tốc độ học phù hợp và giá trị cường độ chính quy hóa, tổng số epochs mà mạng cần được huấn luyện, có thể sử

dụng động lượng (cho trước giá trị) và tăng tốc Nesterov. Cần dành thời gian để thử nghiệm với SGD để có thể thành thạo với việc điều chỉnh các tham số.

Làm quen với một thuật toán tối ưu hóa nhất định cũng tương tự như tự lái xe ô tô - tự lái chiếc xe tốt hơn những người khác vì đã dành nhiều thời gian để lái nó; hiểu chiếc xe và sự phức tạp của nó. Thông thường, một trình tối ưu hóa nhất định được chọn để huấn luyện một mạng trên bộ dữ liệu không phải vì bản thân trình tối ưu hóa tốt hơn, mà bởi vì trình điều khiển (tức là người thực hành nghiên cứu sâu) đã quen thuộc hơn với trình tối ưu hóa và hiểu được nghệ thuật điều chỉnh các thông số.

Hãy nhớ rằng việc có được một mạng nơ-ron hoạt động hợp lý trên ngay cả một tập dữ liệu vừa/nhỏ có thể mất từ 10 đến 100 giây thử nghiệm ngay cả đối với người dùng học sâu nâng cao - đừng nản lòng khi mạng hoạt động không tốt ngay từ đầu. Để trở nên thành thạo với học sâu sẽ đòi hỏi một sự đầu tư thời gian và nhiều thử nghiệm - nhưng nó sẽ có giá trị khi nắm vững cách các thành phần này kết hợp với nhau.

1.1.7. Khởi tạo trọng số

Trước khi kết thúc chương này, sẽ thảo luận ngắn gọn về khái niệm khởi tạo trọng số, hay đơn giản hơn là cách khởi tạo ma trận trọng số và vectơ sai lệch b.

Phần này không phải là một kỹ thuật khởi tạo toàn diện, tuy nhiên, sử dụng các phương pháp phổ biến từ tài liệu mạng nơ-ron và quy tắc chung. Để minh họa cách mà phương pháp khởi tạo trọng số này hoạt động, đoạn chương trình giả ngẫu nhiên cơ bản giống như Python / NumPy là phù hợp.

1.1.8. Khởi tạo liên tục

Khi áp dụng chuẩn hóa (normalize) không đổi, tất cả các trọng số trong mạng nơ-ron được khởi tạo với giá trị không đổi, C. Thông thường C bằng 0 hoặc 1.

Để hình dung điều này trong chương trình giả ngẫu nhiên, hãy xem xét một lớp tùy ý mạng nơ-ron có 64 đầu vào và 32 đầu ra (không bao gồm bias để giải thích một cách dễ dàng). Để khởi tạo các trọng số này thông qua NumPy và không khởi tạo (mặc định được sử dụng bởi thư viện Caffe, một framework học sâu phổ biến), hãy thực hiện:

```
>>> W = np.zeros((64, 32))
```

Tương tự, một khởi tạo có thể được thực hiện thông qua:

```
>>> W = np.ones((64, 32))
```

Có thể áp dụng khởi tạo liên tục bằng cách sử dụng tùy ý C bằng cách sử dụng:

```
>>> W = np.ones((64, 32)) * C
```

Mặc dù khởi tạo liên tục rất dễ nắm bắt và dễ hiểu, nhưng vẫn đề khi sử dụng phương pháp này là gần như không thể phá vỡ tính đối xứng của hàm kích hoạt [26]. Do đó, nó hiếm khi được sử dụng như một công cụ khởi tạo trọng số mạng nơ-ron.

1.1.9. Phân phối đồng nhất và phân phối chuẩn hóa

Một phân phối đồng nhất rút ra một giá trị ngẫu nhiên từ khoảng [dưới, trên] trong đó mọi giá trị trong phạm vi này có xác suất được rút ra bằng nhau.

Một lần nữa, hãy giả sử rằng đối với một lớp nhất định trong mạng nơ-ron, có 64 đầu vào và 32 đầu ra. Sau đó, khởi tạo các trọng số trong phạm vi thấp hơn = -0,05 và trên = 0,05. Áp dụng đoạn chương trình Python + NumPy sau đây sẽ cho phép đạt được chuẩn hóa mong muốn:

```
>>> W = np.random.uniform(low=-0.05, high=0.05, size=(64, 32))
```

Thực thi chương trình ở trên NumPy bằng cách tạo ngẫu nhiên $64 \times 32 = 2048$ trong phạm vi [-0,05 0,05] trong đó mỗi giá trị trong phạm vi này có xác suất bằng nhau.

Sau đó, phân phối chuẩn hóa nơi xác định mật độ xác suất cho phân phối Gaussian là:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (1.12)$$

Các tham số quan trọng nhất ở đây là Patrick (giá trị trung bình) và (độ lệch chuẩn). Bình phương độ lệch chuẩn, σ^2 , được gọi là phuong sai.

Khi sử dụng thư viện Keras, lớp RandomNormal rút ra các giá trị ngẫu nhiên từ một phân phối chuẩn hóa với $\mu = 0$ và $\sigma = 0,05$. Có thể mô phỏng hành vi này bằng NumPy bên dưới:

```
>>> W = np.random.uniform(low=-0.05, high=0.05, size=(64, 32))
```

Cả hai phân phối đồng đều và bình thường có thể được sử dụng để

khởi tạo các trọng số trong mạng nơ-ron; tuy nhiên, thường áp đặt các phương pháp phỏng đoán khác nhau để tạo ra các sơ đồ khởi tạo tốt hơn (như sẽ thảo luận trong các phần còn lại).

1.1.10. Đồng bộ và chuẩn hóa

Nếu đã từng sử dụng các framework Torch7 hoặc PyTorch, có thể nhận thấy rằng phương thức tạo trọng số mặc định “truyền hiệu quả”, được bắt nguồn từ nghiên cứu của LeCun et al. [27].

Ở đây, các tác giả định nghĩa một tham số Fin (được gọi là “fan in”, hoặc số lượng lớp đầu vào) cùng với Fout (“fan out”, hoặc số lượng đầu ra từ lớp). Sử dụng các giá trị này, có thể áp dụng khởi tạo đồng nhất bằng phương pháp sau:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(3 / float(F_in))
>>> W = np.random.uniform(low=-limit, high=limit, size=(F_in, F_out))
```

Cũng có thể sử dụng một phân phối chuẩn hóa. Thư viện Keras sử dụng phân phối chuẩn chẵn trên và dưới khi xây dựng cùng với giá trị trung bình bằng 0 như sau:

```
>>> F_in = 64
>>> F_out = 32
>>> limit = np.sqrt(1 / float(F_in))
>>> W = np.random.normal(0.0, limit, size=(F_in, F_out))
```

1.1.11. Sự khác biệt trong triển khai khởi tạo

Các giá trị giới hạn thực tế có thể khác nhau đối với đồng nhất LeCun/phân phối chuẩn hóa, đồng nhất Xavier/phân phối chuẩn hóa và đồng nhất He et al./phân phối chuẩn hóa. Ví dụ: khi sử dụng đồng nhất Xavier trong Caffe, giới hạn = -np.sqrt (3/ F_in) [26]; tuy nhiên, khởi tạo Xavier mặc định cho Keras sử dụng np.sqrt (6 / (F_in + Fout)) [28]. Không có phương pháp nào là đúng hơn so với các phương pháp khác, nhưng nên đọc tài liệu trong thư viện học sâu Keras tương ứng.

1.1.12. Tóm tắt

Trong phần 1.1 này, đã xem xét các nguyên tắc cơ bản của mạng nơ-ron. Cụ thể, tập trung vào lịch sử các mạng nơ-ron và mối quan hệ với sinh học.

Từ đó, chuyển sang các mạng nơ-ron nhân tạo như thuật toán Perceptron. Mặc dù mang tính chất quan trọng từ các quan điểm lịch sử

nhưng thuật toán Perceptron có một lỗ hổng lớn đó là chỉ phân loại chính xác các điểm phân tách phi tuyến. Để làm việc với các bộ dữ liệu khó khăn hơn, cần cả hai (1) hàm kích hoạt phi tuyến và (2) mạng nhiều lớp.

Để huấn luyện mạng nhiều lớp, phải sử dụng thuật toán lan truyền ngược. Sau đó, đã triển khai lan truyền ngược bằng tay và chứng minh rằng lúc nào thì sử dụng chúng để huấn luyện các mạng nhiều lớp kết hợp với các hàm kích hoạt phi tuyến để mô hình hóa các bộ dữ liệu phân tách phi tuyến (như XOR)

Tất nhiên, thực hiện lan truyền ngược bằng tay là một quá trình khó khăn và dễ xảy ra lỗi - do đó thường dựa vào các thư viện hiện có như Keras, Theano, TensorFlow, v.v. Điều này cho phép tập trung vào kiến trúc thực tế hơn là thuật toán cơ bản được sử dụng để huấn luyện mạng.

Cuối cùng, đã xem xét bốn thành phần chính khi làm việc với bất kỳ mạng nơ-ron nào, bao gồm tập dữ liệu, hàm tổn thất, mô hình/kiến trúc và phương pháp tối ưu hóa.

Thật không may, như một số kết quả đã chứng minh (trên bộ CIFAR-10) các mạng nơ-ron tiêu chuẩn đã không đạt được độ chính xác phân loại cao khi làm việc với các bộ dữ liệu ảnh đầy thách thức thể hiện các biến thể trong dịch thuật, xoay, góc quan sát, v.v., vì vậy sẽ phải cần có một loại mạng nơ-ron đặc biệt gọi là Mạng nơ-ron tích chập (CNN), được trình bày trong chương tiếp theo.

1.2. HỌC SÂU (DEEP LEARNING) LÀ GÌ?

“Các phương pháp học sâu là các phương pháp học đại diện với nhiều cấp độ diễn tả, thu được bằng cách điều chỉnh các mô-đun đơn giản nhưng phi tuyến mà mỗi mô-đun biến đổi đại diện ở một cấp độ (bắt đầu bằng đầu vào thô) thành đại diện ở mức cao hơn, hơi trừu tượng hơn. [...] Khía cạnh quan trọng của học sâu là các lớp này không phải thiết kế bởi các con người: chúng được học từ dữ liệu sử dụng một quy trình học có mục đích chung” - Yann LeCun, Yoshua Bengio, và Geoffrey Hinton, Nature năm 2015. [5]

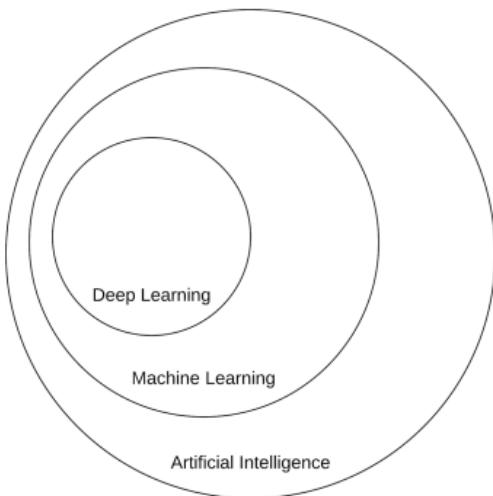
Học sâu là một lĩnh vực của học máy, là một trường con của lĩnh vực trí tuệ nhân tạo (AI). Để mô tả đồ họa về mối quan hệ này, có thể tham khảo Hình 1.16.

Mục tiêu trọng tâm của AI là cung cấp một tập hợp các thuật toán và kỹ thuật có thể được sử dụng để giải quyết vấn đề mà con người thực hiện bằng trực giác và gần như là tự động (là một thách thức lớn cho máy

tính). Một ví dụ tuyệt vời về AI là diễn giải và hiểu nội dung một bức ảnh - nhiệm vụ này là điều mà con người có thể làm một cách đơn giản, nhưng nó đã được chứng minh là cực kỳ khó khăn để thực hiện với máy móc.

Trong khi AI là một hiện thân lớn, đa dạng liên quan đến lý luận máy tự động (suy luận, kế hoạch, phương pháp giải quyết vấn đề bằng cách đánh giá kinh nghiệm và tìm giải pháp qua kiểm tra, v.v.), trường con học máy có xu hướng đặc biệt quan tâm đến nhận dạng mẫu và học từ dữ liệu.

Mạng nơ-ron nhân tạo (ANN) là một lớp các thuật toán học máy (được học từ dữ liệu và chuyên về nhận dạng mẫu) được tham khảo từ cấu trúc và chức năng não. Học sâu thuộc về họ thuật toán ANN. Trong hầu hết các trường hợp, cả hai thuật ngữ có thể được sử dụng thay thế cho nhau. Thực tế, lĩnh vực học sâu đã tồn tại hơn 60 năm với những tên gọi và phiên bản khác nhau (dựa trên xu hướng nghiên cứu), phần cứng và bộ dữ liệu có sẵn, và các lựa chọn phổ biến của các nhà nghiên cứu nổi tiếng tại thời điểm đó.



Hình 1.16: Sơ đồ Venn mô tả học sâu như là một trường con của học máy, mà học máy là một trường con của trí tuệ nhân tạo (Ảnh được tham khảo từ Hình 1.4 Goodfellow và cộng sự).

1.2.1. Lịch sử ngắn gọn về mạng nơ-ron và học sâu

Lịch sử mạng nơ-ron và học sâu trải qua một khoảng thời gian dài và không rõ ràng. Thật ngạc nhiên khi biết rằng học sâu đã tồn tại từ những năm 1940 với nhiều tên gọi khác nhau và nhiều phiên bản, bao gồm điều khiển học, kết nối học và quen thuộc nhất là mạng nơ-ron nhân tạo (ANN).

Mặc dù được tham khảo từ bộ não con người và cách mà các nơ-ron tương tác với nhau, ANN không phải là mô hình thực tế của bộ não. Thay vào đó, chúng là nguồn cảm hứng, cho phép mô phỏng mô hình cơ bản bộ não và cách làm thế nào để có thể bắt chước một số hành vi thông qua mạng nơ-ron nhân tạo.

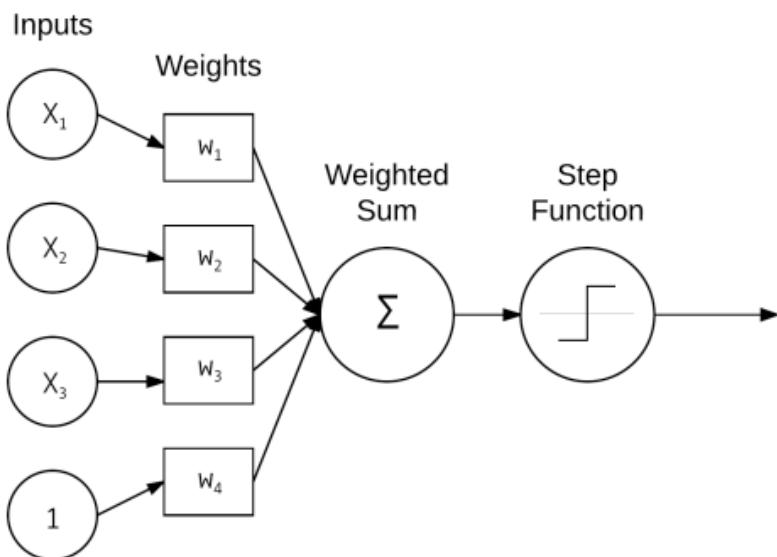
Mô hình mạng nơ-ron đầu tiên đến từ McCulloch và Pitts năm 1943 [29]. Mạng này là một bộ phân loại nhị phân, có khả năng nhận ra hai loại khác nhau dựa trên một số đầu vào. Vấn đề là các trọng số được sử dụng để xác định nhãn cho đầu vào đã cho cần phải được điều chỉnh thủ công bởi con người - loại mô hình này rõ ràng không thể mở rộng tốt nếu con người phải can thiệp.

Sau đó, vào những năm 1950, thuật toán Perceptron được đưa ra bởi Rosenblatt [12,30] - mô hình này có thể tự động học các trọng số cần thiết để phân loại đầu vào (không yêu cầu sự can thiệp của con người). Một ví dụ về kiến trúc Perceptron có thể được nhìn thấy trong Hình 1.17. Trong thực tế, quy trình huấn luyện tự động đã hình thành nên cơ sở Stochastic Gradient Descent (SGD) vẫn còn được sử dụng để huấn luyện mạng nơ-ron rất sâu ngày nay.

Trong khoảng thời gian này, các kỹ thuật dựa trên Perceptron nổi lên trong cộng đồng sử dụng mạng nơ-ron. Tuy nhiên, một nghiên cứu vào năm 1969 của Minsky và Papert [13] đã khiến cho việc nghiên cứu mạng nơ-ron bị đình trệ trong gần một thập kỷ. Công trình của họ đã chứng minh rằng một Perceptron với hàm kích hoạt tuyến tính (bất kể độ sâu) chỉ là một bộ phân loại tuyến tính, không thể giải quyết các vấn đề phi tuyến. Ví dụ chính xác về một vấn đề phi tuyến là bộ dữ liệu XOR trong Hình 1.18, để thuyết phục rằng không thể thử một đường đơn có thể tách các ngôi sao màu xanh từ các vòng tròn màu đỏ.

Hơn nữa, tác giả lập luận rằng (tại thời điểm đó) không có tài nguyên tính toán cần thiết để xây dựng các mạng nơ-ron lớn và sâu (nhìn chung, điều đó hoàn toàn chính xác).

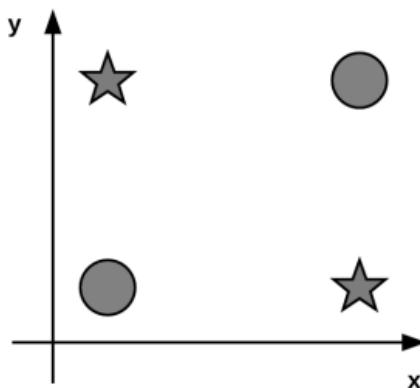
May mắn thay, thuật toán lan truyền ngược và nghiên cứu Werbos (1974) [15], Rumelhart (1986) [16] và LeCun (1998) [27] đã có thể đưa mạng nơ-ron về vị trí của nó. Nghiên cứu của họ trong thuật toán lan truyền ngược cho phép mạng nơ-ron nhiều lớp lan truyền thuận có thể được huấn luyện (Hình 1.19)



Hình 1.17: Ví dụ về kiến trúc mạng Perceptron đơn giản có số đầu vào, tính toán tổng trọng số và áp dụng hàm bước để có được kết quả dự đoán cuối cùng. Sẽ nói rõ hơn về mạng Perceptron chi tiết tại Mục 1

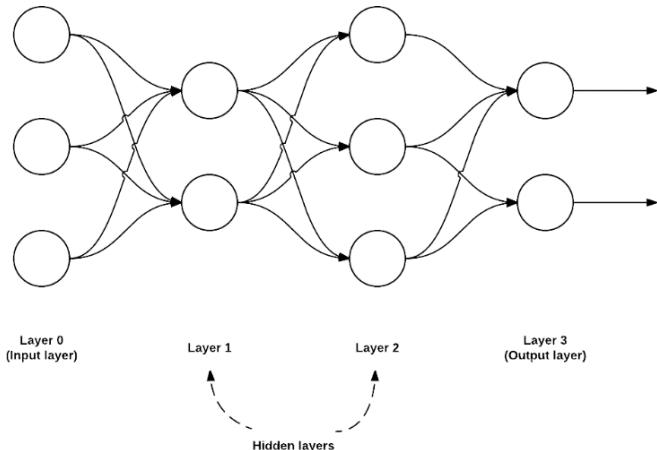
Kết hợp với các hàm kích hoạt phi tuyến, giờ đây các nhà nghiên cứu có thể tìm hiểu các hàm phi tuyến và giải quyết vấn đề XOR, mở ra hướng đi cho một lĩnh vực nghiên cứu hoàn toàn mới trong mạng nơ-ron. Nghiên cứu sâu hơn đã chứng minh rằng mạng nơ-ron là các xấp xỉ phô quát [31], có khả năng xấp xỉ bất kỳ hàm liên tục nào.

XOR Dataset (Nonlinearly Separable)



Hình 1.18: Tập dữ liệu XOR là một mô hình ví dụ ván để tách rời phi tuyến mà Perceptron không thể giải quyết (không thể vẽ đường đơn phân cách các ngôi sao màu xanh với các vòng tròn màu đỏ)

Thuật toán lan truyền ngược là nền tảng các mạng nơ-ron hiện đại cho phép để huấn luyện một cách hiệu quả các mạng nơ-ron và dạy chúng học hỏi từ những sai lầm. Tuy nhiên, trong thời điểm này, do (1) phần cứng máy tính chậm (so với máy móc hiện đại) và (2) sự thiếu hụt lớn các bộ huấn luyện đã được dán nhãn, các nhà nghiên cứu không thể huấn luyện các mạng nơ-ron có nhiều hơn hai lớp ẩn - đơn giản là do không thể tính toán được.



Hình 1.19: Một kiến trúc mạng nhiều lớp, kiến trúc mạng lan truyền thuận với một lớp đầu vào (3 nút), hai lớp ẩn (2 nút ở lớp đầu tiên và 3 nút ở lớp thứ hai) và một lớp đầu ra (2 nút).

Ngày nay, phiên bản mới nhất của mạng nơ-ron như đã biết là học sâu. Điều làm cho học sâu vượt ngoài các phiên bản trước đó của nó là phần cứng chuyên dụng nhanh hơn với nhiều tập dữ liệu huấn luyện sẵn có hơn. Hiện tại có thể huấn luyện các mạng với nhiều lớp ẩn hơn, có khả năng học theo cấp bậc, trong đó các khái niệm đơn giản được học ở các lớp thấp hơn và các mô hình trừu tượng hơn ở các lớp cao hơn của mạng.

Một ví dụ điển hình của việc học sâu qua các cấp bậc học là Convolutional Neural Network (LeCun 1988) [32] áp dụng để nhận dạng ký tự viết tay tự động tìm hiểu các mẫu phân biệt từ ảnh bằng cách xếp chồng liên tiếp các lớp liên tiếp lên nhau. Các bộ lọc ở các cấp thấp hơn của mạng nơ-ron đại diện cho các cạnh và góc, trong khi các lớp cấp cao hơn sử dụng các cạnh và góc để tìm hiểu các khái niệm trừu tượng để phân biệt các lớp ảnh.

Trong nhiều ứng dụng, CNN hiện nay được coi là trình phân loại ảnh mạnh nhất và hiện chịu trách nhiệm thúc đẩy các công nghệ tiên tiến trong

lĩnh vực thị giác máy tính để làm đòn bẩy cho quá trình học máy. Để tìm hiểu kỹ hơn về lịch sử mạng nơ-ron và học sâu, xem thêm tại [33] cũng như [34]

1.2.2. Học đặc trưng phân cấp

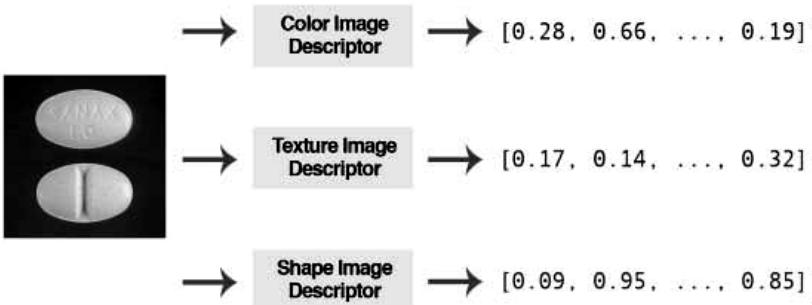
Các thuật toán học máy (nói chung) được chia thành ba nhóm là học giám sát, học không giám sát và học bán giám sát.

Trong trường hợp được giám sát, thuật toán học máy được cung cấp cả bộ dữ liệu đầu vào và mục tiêu đầu ra. Thuật toán sau đó cố gắng tìm hiểu các mẫu có thể được sử dụng để tự động ánh xạ dữ liệu đầu vào và chỉ ra mục tiêu đầu ra chính xác. Học có giám sát tương tự như có một giáo viên theo dõi người học làm một bài kiểm tra. Với kiến thức được dạy, người học cố gắng hết sức để đánh dấu câu trả lời đúng trong bài thi; tuy nhiên, nếu không chính xác, giáo viên sẽ hướng dẫn một cách làm tốt hơn, có độ chính xác hơn vào lần tới.

Trong trường hợp không được giám sát, các thuật toán học máy cố gắng tự động tìm ra các đặc trưng phân biệt mà không có bất kỳ gợi ý nào về đầu vào. Trong trường hợp này, người học cố gắng để nhóm tất cả các câu hỏi và câu trả lời tương tự lại với nhau, mặc dù người đọc không biết đâu là câu trả lời đúng và giáo viên không có mặt để hướng dẫn cho họ câu trả lời đúng. Học không giám sát rõ ràng là một vấn đề khó khăn hơn so với học có giám sát - bằng cách biết câu trả lời (tức là, mục tiêu đầu ra), có thể dễ dàng xác định các mẫu phân biệt có thể ánh xạ dữ liệu đầu vào sang phân loại mục tiêu chính xác

Trong bối cảnh học máy được áp dụng để phân loại ảnh, mục tiêu của thuật toán học máy là lấy các bộ ảnh này và xác định các mẫu có thể được sử dụng để phân biệt các lớp hoặc đối tượng ảnh khác nhau.

Trước đây, việc sử dụng các đặc trưng được thiết kế bằng tay để định lượng nội dung hình ảnh, hiếm khi sử dụng giá trị pixel thô làm đầu vào cho mô hình học máy như hiện nay, và phổ biến như là học sâu. Đối với mỗi ảnh trong tập dữ liệu đã thực hiện trích xuất đặc trưng hoặc quá trình lấy ảnh đầu vào, định lượng nó theo một số thuật toán (được gọi là trình trích xuất đặc trưng hoặc mô tả ảnh) và trả về một vectơ (tức là dãy các con số) nhằm mục đích định lượng nội dung của một hình ảnh. Hình 1.20 dưới đây mô tả quá trình định lượng ảnh có chứa các viên thuốc thông qua một loạt các mô tả ảnh có phần nền màu đen, kết cấu và mô tả kích thước hình ảnh



Hình 1.20: Định lượng nội dung ảnh có chứa thuốc theo toa thông qua một loạt các mô tả ảnh có nền màu đen, kết cấu và hình dạng hình ảnh

Các đặc trưng được thiết kế bằng tay đã mã hóa kết cấu (Mẫu nhị phân cục bộ [2], Haralick texture [35]), hình dạng (Hu Moments [36], Zernike Moments [37]) và màu sắc (dài màu sắc, lược đồ màu, biểu đồ tương quan màu [38]).

Các phương pháp khác như các bộ phát hiện điểm mấu chốt (FAST [39], Harris [40], DoG [41]) và các mô tả bất biến cục bộ (SIFT [41], SURF [42], BRIEF [43], ORB [44], v.v.) mô tả các khu vực nổi bật (nghĩa là, khu vực quan tâm nhất) của một ảnh.



Hình 1.21: HOG + phát hiện đối tượng tuyến tính SVM được áp dụng để phát hiện vị trí các biển báo dừng trong ảnh

Các phương pháp khác như Histogram of Oriented Gradients (HOG) [45] đã được chứng minh là rất tốt để phát hiện vật thể trong ảnh khi mà góc nhìn ảnh không thay đổi đáng kể so với những gì phân loại đã được huấn luyện. Một ví dụ về việc sử dụng HOG + phương pháp phát hiện tuyến tính SVM được thể hiện trong Hình 1.21- nơi phát hiện sự hiện diện các biển báo dừng trong ảnh

Trong một khoảng thời gian, nghiên cứu phát hiện đối tượng trong ảnh được thực hiện bởi thuật toán HOG và các biến thể của nó, bao gồm các phương pháp tính toán như Deformable Parts Model [46] và SVM Exemplar [47]

Trong trường hợp này, một thuật toán được xác định bằng tay để định lượng và mã hóa khía cạnh cụ thể của một hình ảnh (tức là hình dạng, kết cấu, màu sắc, v.v.). Đưa ra một ảnh đầu vào là tập hợp các pixel, sẽ áp dụng thuật toán xác định bằng tay cho các pixel và đổi lại nhận được một vectơ đặc trưng định lượng nội dung ảnh - bản thân các pixel ảnh không phục vụ mục đích nào khác ngoài mục đích là trích xuất đặc trưng. Các vectơ đặc trưng thực hiện trích xuất các đặc trưng là những gì cần được quan tâm khi chúng đóng vai trò là đầu vào cho các mô hình học máy

Học sâu, và cụ thể là mạng nơ-ron tích chập có cách tiếp cận khác. Thay vì xác định một tập hợp các quy tắc và thuật toán bằng tay để trích xuất các đặc trưng từ một ảnh thì các đặc trưng này sẽ được tự động học từ quá trình huấn luyện mạng.

Một lần nữa, quay trở lại mục tiêu của học máy: máy tính sẽ có thể học hỏi từ kinh nghiệm có sẵn về vấn đề mà nó đang cố gắng giải quyết.

Sử dụng học sâu, cố gắng hiểu vấn đề theo cách phân cấp các lớp đặc trưng. Mỗi lớp được xây dựng phía trên những lớp trước đó. Các đặc trưng trong các lớp cấp thấp trong mạng mã hóa một số biểu diễn cơ bản của vấn đề, trong khi các lớp cấp cao hơn sử dụng các lớp cơ bản này để tạo thành các tầng trừu tượng. Cách học phân cấp này cho phép loại bỏ hoàn toàn quá trình trích xuất đặc trưng được thực hiện theo cách thủ công và coi CNN là người học từ đầu đến cuối.

Cho một ảnh cung cấp các giá trị cường độ pixel làm đầu vào cho mạng nơ-ron tích chập (CNN). Một loạt các lớp ẩn được sử dụng để trích xuất các đặc trưng từ ảnh đầu vào. Các lớp ẩn này sẽ tự xây dựng trong một hệ thống phân cấp. Lúc đầu, chỉ có các vùng giống như cạnh được phát hiện trong các lớp cấp thấp của mạng. Các vùng giống như cạnh được sử dụng để xác định các góc (nơi các cạnh giao nhau) và các đường biên (phác thảo các đối tượng). Kết hợp các góc và đường biên có thể đưa ra khái niệm “phần vật thể” trong các lớp tiếp theo.

Một lần nữa, các loại đặc trưng qua các bộ lọc sẽ tự động được học (không có sự can thiệp nào trong quá trình học). Cuối cùng, lớp đầu ra được sử dụng để phân loại ảnh và thu được lớp nhãn đầu ra (bị ảnh hưởng trực tiếp hoặc gián tiếp bởi mọi nút khác trong mạng).

Có thể xem quá trình này là học phân cấp: mỗi lớp trong mạng sử dụng đầu ra của các lớp trước như là các khối để xây dựng các đặc trưng trừu tượng hơn. Những lớp được học một cách tự động - không có đặc trưng nào được trích xuất thủ công mà diễn ra trong mạng no-ron. Một trong những lợi ích chính của việc học sâu và mạng no-ron tích chập là nó cho phép bỏ qua bước trích đặc trưng và thay vào đó tập trung vào quá trình huấn luyện mạng để tìm hiểu các bộ lọc này. Tuy nhiên, huấn luyện một mạng để có được độ chính xác hợp lý trên một tập dữ liệu ảnh luôn là một nhiệm vụ không phải dễ dàng.

1.2.3. Tóm tắt

Phần 1.2 này giải quyết câu hỏi khá phức tạp là “Học sâu là gì?”

Như đã tìm hiểu, học sâu đã có từ những năm 1940, với những tên gọi khác nhau và hóa thân dựa trên các trường phái tư tưởng và xu hướng nghiên cứu phổ biến tại một thời điểm nhất định. Vẫn đề cốt lõi ở đây, học sâu thuộc về họ của mạng nơ ron nhân tạo (ANNs), một bộ các thuật toán học các mẫu được tham khảo từ cấu trúc và chức năng của một bộ não.

Các chuyên gia không có sự đồng thuận về cấu trúc của một mạng no-ron, tuy nhiên:

1. Thuật toán học sâu học theo kiểu phân cấp và do đó xếp chồng nhiều lớp lên nhau để tìm ra các đặc trưng càng trừu tượng hơn.
2. Một mạng nên có > 2 lớp để được coi là sâu (Deep)
3. Một mạng có > 10 lớp được coi là rất sâu (mặc dù con số này sẽ thay đổi là các kiến trúc như ResNet đã được huấn luyện thành công với hơn 100 lớp).

Nếu cảm thấy hơi bối rối hoặc thậm chí choáng ngợp sau khi đọc chương này, đừng lo lắng - mục đích ở đây chỉ đơn giản là cung cấp một cái nhìn tổng quan về học sâu.

Phần 1.2 cũng giới thiệu một số khái niệm và thuật ngữ có thể không quen thuộc, bao gồm pixel, cạnh và góc - chương tiếp theo sẽ đề cập đến các loại ảnh cơ bản và cung cấp cho người đọc một nền tảng cụ thể để dễ tiếp cận hơn. Sau đó, bắt đầu chuyển sang các nguyên tắc cơ bản của mạng no-ron, cho phép hiểu rõ về học sâu và mạng no-ron tích chập trong cuốn sách này. Trong khi chương này chủ yếu trình bày các khái niệm trừu tượng thì các chương khác trong cuốn sách này sẽ nghiêng về thực hành, cho phép người đọc thành thạo việc lập trình các vấn đề về học sâu và thị giác máy tính.

Chương 2: GIỚI THIỆU MẠNG NƠ-RON TÍCH CHẬP

Trong các mạng nơ-ron truyền thống (như các mạng nơ-ron đã xem xét ở chương 1), mỗi nơ-ron đầu vào được kết nối với mọi nơ-ron đầu ra ở các lớp tiếp theo - gọi đây là lớp (FC) - kết nối đầy đủ. Tuy nhiên, trong CNN, không sử dụng các lớp FC cho đến (các) lớp cuối cùng trong mạng. Do đó, có thể định nghĩa một CNN là một mạng nơ-ron chuyên vị trong một lớp tích chập chuyên dụng thay thế cho lớp được kết nối đầy đủ cho ít nhất một trong các lớp trong mạng [33].

Hàm tác động phi tuyến (như ReLU) sau đó được dùng cho đầu ra các kết cấu này và quá trình tích chập => tiếp tục kích hoạt (cùng với các loại lớp khác để giúp giảm chiều rộng và chiều cao dữ liệu đầu vào và giúp giảm hiện tượng quá khớp) cho đến khi tìm đến cuối mạng và áp dụng một hoặc hai lớp FC, nơi có thể có được các phân loại đầu ra cuối cùng.

Mỗi lớp trong CNN áp dụng một bộ lọc khác nhau, thường là hàng trăm hoặc hàng nghìn bộ lọc và tích chập các kết quả, đưa đầu ra vào lớp tiếp theo trong mạng. Trong quá trình huấn luyện, một CNN tự động tìm hiểu các giá trị cho các bộ lọc này.

Trong phân loại ảnh, CNN có thể học cách:

Phát hiện các cạnh từ dữ liệu pixel thô trong lớp đầu tiên.

Sử dụng các cạnh này để phát hiện các hình dạng

Sử dụng các hình dạng này để phát hiện các tính năng cấp cao hơn như cấu trúc khuôn mặt, các bộ phận ô tô, v.v. trong các lớp cao nhất mạng.

Lớp cuối cùng trong CNN sử dụng các tính năng cấp cao hơn này để đưa ra dự đoán liên quan đến nội dung ảnh. Trong thực tế, CNN cho hai lợi ích chính: tính bất biến cục bộ và tính tổng hợp. Khái niệm bất biến cục bộ cho phép phân loại một ảnh có chứa một đối tượng cụ thể bất kể vị trí ảnh xuất hiện ở đâu. Có được tính bất biến cục bộ này thông qua việc sử dụng gộp các lớp (được thảo luận sau trong chương này) để xác định các vùng khói lượng đầu vào với độ phản hồi cao đối với một bộ lọc cụ thể.

Lợi ích thứ hai là thành phần. Mỗi bộ lọc tạo thành một bản vá cục bộ các tính năng cấp thấp hơn, đại diện cho cấp cao hơn, tương tự như cách có thể soạn một tập hợp các hàm toán học xây dựng trên đầu ra các hàm trước: $f(g(x(h(x)))$) - Thành phần này cho phép mạng tìm hiểu các tính năng

phong phú hơn sâu hơn trong mạng. Ví dụ: mạng có thể xây dựng các cạnh từ pixel, hình dạng từ các cạnh và sau đó các đối tượng phức tạp từ hình dạng - tất cả đều diễn ra tự nhiên trong quá trình huấn luyện. Khái niệm xây dựng các tính năng cấp cao hơn từ các tính năng cấp thấp hơn chính xác là lý do tại sao CNN rất mạnh trong thị giác máy tính.

Trong phần còn lại chương này, chúng ta sẽ xem xét các phép tích chập là gì và vai trò chúng trong học sâu. Sau đó, chuyển sang các khối xây dựng CNN: các lớp và các loại lớp khác nhau được sử dụng để xây dựng các CNN riêng. Phần kết thúc sẽ xem xét các mô hình phổ biến được cấu tạo bằng việc sắp xếp các khối lại thành kiến trúc mạng CNN. Những mạng này hoạt động tốt với nhiệm vụ phân loại các tập ảnh có tính chất đa dạng.

Chương này giúp người học có (1) hiểu biết sâu sắc về mạng nơ-ron tích chập và quá trình đưa ra ý tưởng để xây dựng nó và (2) một số công thức về CNN có thể sử dụng để xây dựng các kiến trúc mạng cho riêng mình. Trong phần tiếp theo, sử dụng các nguyên tắc cơ bản và các công thức này để huấn luyện CNN

2.1. MẠNG NO-RON TÍCH CHẬP

2.1.1. Hiểu về phép tích chập

Trong phần này, giải quyết một số câu hỏi, bao gồm:

- Tích chập ảnh là gì?
- Tích chập để làm gì?
- Tại sao phải sử dụng phép tích chập?
- Làm thế nào để áp dụng tích chập trong ảnh?
- Phép tích chập đóng vai trò gì trong học sâu?

Từ “tích chập” có vẻ như là một thuật ngữ phức tạp. Nếu có bất kỳ kinh nghiệm nào trước đây về thị giác máy tính, xử lý ảnh hoặc OpenCV trước đó, thì đã áp dụng các phép tích chập, cho dù có nhận ra hay không!

Bạn đã bao giờ áp dụng làm mờ hoặc làm mịn cho một ảnh? Đúng, đó là một tích chập. Đã từng phát hiện biên? Đây cũng là phép tích chập. Nhưng bản thân thuật ngữ này có xu hướng khiến mọi người sợ hãi - thực tế, trên bề mặt, từ này thậm chí còn có ý nghĩa tiêu cực.

Về mặt học sâu, tích chập (ảnh) là một phép nhân các phần tử của hai ma trận và sau đó tính tổng các thành phần.

Tích chập ảnh được thực hiện như sau:

1. Thực hiện hai ma trận (cả hai đều có cùng kích thước).
2. Nhân chúng lại, từng phần tử với nhau
3. Cộng các phần tử sau khi nhân với nhau

Phần sau sẽ tìm hiểu thêm về các cấu trúc, đặc trưng và cách chúng được sử dụng trong các CNN trong phần còn lại phần này.

2.1.1.1. Tích chập so với tương quan chéo

Một người đọc có nền tảng trước về thị giác máy tính và xử lý ảnh có thể đã xác định mô tả về tích chập ở trên là một hoạt động tương quan chéo thay thế. Sử dụng tương quan chéo thay vì tích chập thực sự là do thiết kế. Chuyển đổi (được biểu thị bởi toán tử \triangleright) trên ảnh đầu vào hai chiều I và mặt nạ đặc trưng K hai chiều được định nghĩa là:

$$S(i, j) = (I \triangleright K)(i, j) = \sum \sum K(i - m, j - n)I(m, n) \quad (2.1)$$

Tuy nhiên, gần như tất cả các thư viện máy học và học sâu đều sử dụng hàm tương quan chéo đơn giản hóa:

$$S(i, j) = (I \triangleright K)(i, j) = \sum \sum K(i + m, j + n)I(m, n) \quad (2.2)$$

Tất cả phép toán học này là một thay đổi dấu hiệu trong cách truy cập vào tọa độ ảnh I (tức là, không thể lật các mặt nạ đặc trưng so với đầu vào khi áp dụng tương quan chéo).

Nhiều thư viện học sâu sử dụng hoạt động tương quan chéo đơn giản hóa và gọi nó là tích chập. Đối với độc giả muốn tìm hiểu sâu hơn về các phương pháp toán học đằng sau tích chập và tương quan chéo, vui lòng tham khảo Chương 3 Computer Vision: Algorithms and Applications by Szelski [48].

2.1.1.2. Ma trận lớn và ma trận nhỏ tương tự

Một ảnh là một ma trận đa chiều. Ảnh có chiều rộng (# cột) và chiều cao (# hàng), giống như ma trận. Điểm khác với các ma trận truyền thống mà chúng ta đã được học, là ảnh cũng có chiều sâu - số lượng kênh trong ảnh.

Đối với ảnh RGB tiêu chuẩn, có độ sâu 3 - một kênh cho mỗi kênh Đỏ, Xanh lục và Xanh lam tương ứng. Như vậy, có thể coi một ảnh là ma trận lớn và ma trận mặt nạ đặc trưng hoặc ma trận chập như một ma trận nhỏ được sử dụng để làm mờ, làm sắc nét, phát hiện cạnh và các hàm xử lý khác. Về cơ bản, mặt nạ đặc trưng nhỏ này nằm trên đỉnh ảnh lớn và trượt từ trái sang phải và từ trên xuống dưới, áp dụng một phép toán (nghĩa là một phép chập) ở mỗi (x, y) - phối hợp ảnh gốc.

Mặt nạ đặc trưng thường được xác định thủ công để có được các hàm xử lý ảnh khác nhau. Trên thực tế, chúng ta có thể đã quen với việc làm mờ (làm mịn trung bình, làm mịn Gaussian, làm mịn trung bình, v.v.), phát hiện cạnh (Laplacian, Sobel, Scharr, Prewitt, v.v.) và làm sắc nét - tất cả các thao tác này đều là hình thức thủ công, mặt nạ đặc trưng được xác định được thiết kế đặc biệt để thực hiện một chức năng cụ thể.

Vì vậy, điều đó đặt ra câu hỏi: có cách nào để tự động tìm hiểu các loại bộ lọc này không? Có thể sử dụng các bộ lọc để phân loại ảnh và phát hiện đối tượng? Nhưng trước hết, cần phải tìm hiểu về mặt nạ đặc trưng và phép toán tích chập.

2.1.1.3. *Mặt nạ đặc trưng*

Ảnh được xem như một ma trận lớn và một mặt nạ đặc trưng như một ma trận nhỏ, được mô tả trong Hình 2.1. Như hình minh họa, đang trượt mặt nạ đặc trưng (vùng màu đỏ) từ trái sang phải và từ trên xuống dưới dọc theo ảnh gốc. Tại mỗi (x, y) phối hợp ảnh gốc, dừng lại và kiểm tra vùng lân cận các pixel nằm ở trung tâm mặt nạ đặc trưng ảnh. Sau đó, lấy vùng lân cận pixel này, tích chập chúng với mặt nạ đặc trưng và thu được một giá trị đầu ra duy nhất. Giá trị đầu ra được lưu trữ trong ảnh đầu ra ở cùng (x, y) - phối hợp làm trung tâm mặt nạ đặc trưng.

Nếu điều này nghe có vẻ khó hiểu, hãy xem xét một ví dụ trong phần tiếp theo. Nhưng trước khi đi sâu vào một ví dụ, xem xét một mặt nạ đặc trưng là như thế nào (Hình 2.1):

131	162	232	84	91	207
104	-1	09	+11	237	109
243	-2	02	+2	135 → 26	
185	-15	00	+1	61	225
157	124	25	14	102	108
5	155	16	218	232	249

Hình 2.1: Một mặt nạ đặc trưng có thể được hiển thị dưới dạng một ma trận nhỏ trượt qua, từ trái sang phải và từ trên xuống dưới, một ảnh lớn hơn. Tại mỗi pixel trong ảnh đầu vào, vùng lân cận ảnh được tích hợp với mặt nạ đặc trưng và đầu ra được lưu trữ.

Ở trên, đã định nghĩa một mặt nạ đặc trưng vuông 3×3 (có đoán được mặt nạ đặc trưng này được sử dụng để làm gì không?). Mặt nạ đặc trưng có thể có kích thước hình chữ nhật tùy ý $M \times N$, với điều kiện cả M và N đều là số nguyên lẻ.

Hầu hết các mặt nạ đặc trưng áp dụng cho học sâu và CNN là ma trận $N \times N$ vuông, cho phép tận dụng các thư viện đại số tuyến tính tối ưu hóa hoạt động hiệu quả nhất trên ma trận vuông.

Sử dụng kích thước mặt nạ đặc trưng lẻ để đảm bảo có một số nguyên (x, y) hợp lệ ở giữa ảnh (Hình 2.2). Ở bên trái, có một ma trận 3×3 . Tâm ma trận nằm ở $x = 1, y = 1$ trong đó góc trên cùng bên trái ma trận được sử dụng làm gốc và tọa độ được lập chỉ mục bằng không. Nhưng bên phải, có một ma trận 2×2 . Tâm ma trận này nằm ở $x = 0,5, y = 0,5$.

Nhưng như đã biết, khi áp dụng phép nội suy, thì sẽ không có vị trí pixel $(0,5, 0,5)$ – tọa độ pixel phải là số nguyên. Đây là lý do tại sao phải sử dụng các kích thước mặt nạ đặc trưng lẻ: để luôn đảm bảo có một (x, y) hợp lệ ở trung tâm mặt nạ đặc trưng.

131	162	232
104	93	139
243	26	252

131	162
?	93

Hình 2.2: Trái : pixel trung tâm mặt nạ đặc trưng 3×3 nằm ở tọa độ $(1, 1)$ (được tô sáng màu đỏ). Phải : Tọa độ trung tâm mặt nạ đặc trưng có kích thước 2×2 có giá trị là gì ?

2.1.1.4. Một ví dụ tính toán bằng tay phép tích chập

Bây giờ đã thảo luận về các vấn đề cơ bản mặt nạ đặc trưng, để thảo luận về hoạt động tích chập thực tế và xem một ví dụ về nó thực sự được áp dụng để giúp củng cố kiến thức. Trong xử lý ảnh, tích chập yêu cầu ba thành phần:

1. Một ảnh đầu vào.
2. Một ma trận mặt nạ đặc trưng mà áp dụng cho ảnh đầu vào.
3. Một ảnh đầu ra để lưu trữ đầu ra ảnh được tích hợp với mặt nạ đặc trưng. Convolution (nghĩa là tương quan chéo) thực sự rất dễ dàng. Tất cả những gì cần làm là :

- Chọn một (x, y) – tích chập từ ảnh gốc.
- Đặt trung tâm mặt nạ đặc trưng tại vị trí này (x, y) .
- Thực hiện phép nhân từng phần tử vùng ảnh đầu vào và mặt nạ đặc trưng, sau đó tổng hợp các giá trị các phép toán nhân này thành một giá trị duy nhất. Tổng các phép nhân này được gọi là đầu ra mặt nạ đặc trưng.
- Sử dụng cùng (x, y) - phối hợp từ Bước 1, nhưng lần này, lưu trữ đầu ra mặt nạ đặc trưng ở cùng vị trí (x, y) như ảnh đầu ra.

Dưới đây có thể tìm thấy một ví dụ về tích chập (ký hiệu toán học là toán tử \star) vùng 3×3 ảnh với mặt nạ đặc trưng 3×3 được sử dụng để làm mờ :

$$O_{i,j} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 93 & 139 & 101 \\ 26 & 252 & 196 \\ 135 & 230 & 18 \end{bmatrix} = \begin{bmatrix} 1/9 \times 93 & 1/9 \times 139 & 1/9 \times 101 \\ 1/9 \times 26 & 1/9 \times 252 & 1/9 \times 196 \\ 1/9 \times 135 & 1/9 \times 230 & 1/9 \times 18 \end{bmatrix}$$

Therefore,

$$O_{i,j} = \sum \begin{bmatrix} 10.3 & 15.4 & 11.2 \\ 2.8 & 28.0 & 21.7 \\ 15.0 & 25.5 & 2.0 \end{bmatrix} \approx 132.$$

Sau khi áp dụng tích chập này, đặt pixel nằm ở tọa độ (i, j) ảnh đầu ra O thành $O_{i,j} = 132$.

Tích chập chỉ đơn giản là tổng phép nhân ma trận từng phần tử giữa mặt nạ đặc trưng và vùng lân cận mà mặt nạ đặc trưng trượt trên ảnh đầu vào.

2.1.1.5. Thực hiện tích chập với Python

Để giúp hiểu thêm về khái niệm cấu trúc, cùng xem một số chương trình thực tế để hiểu cách thức mặt nạ đặc trưng và tích chập được thực hiện. Chương trình này không chỉ giúp hiểu cách áp dụng các cấu trúc cho ảnh, mà còn cho phép hiểu những gì diễn ra trong thời gian ngắn khi huấn luyện CNN.

Mở một tập tin mới, đặt tên cho nó là convolutions.py và để cho phép hoạt động:

```

1 # import the necessary packages
2 from skimage.exposure import rescale_intensity
3 import numpy as np
4 import argparse
5 import cv2

```

Bắt đầu trên Dòng 2-5 bằng cách nhập các gói Python cần thiết, sử dụng NumPy và OpenCV cho các hàm xử lý mảng số và thị giác máy tính tiêu chuẩn cùng với thư viện ảnh scikit để giúp thực hiện hàm tích chập tùy chỉnh.

Tiếp theo, có thể bắt đầu xác định phương thức xác thực này:

```
7 def convolve(image, K):
8     # grab the spatial dimensions of the image and kernel
9     (iH, iW) = image.shape[:2]
10    (kH, kW) = K.shape[:2]
11
12    # allocate memory for the output image, taking care to "pad"
13    # the borders of the input image so the spatial size (i.e.,
14    # width and height) are not reduced
15    pad = (kW - 1) // 2
16    image = cv2.copyMakeBorder(image, pad, pad, pad, pad,
17        cv2.BORDER_REPLICATE)
18    output = np.zeros((iH, iW), dtype="float")
```

Hàm tích chập yêu cầu hai tham số: ảnh (xám) mà muốn đối chiếu với mặt nạ đặc trưng. Với cả ảnh và mặt nạ đặc trưng (mà giả sử là mảng NumPy), sau đó xác định kích thước không gian (tức là, chiều rộng và chiều cao) mỗi (Dòng 10 và 11).

Trước khi tiếp tục, điều quan trọng là phải hiểu quá trình trượt ma trận một ma trận tích chập trên một ảnh, áp dụng tích chập và sau đó lưu trữ đầu ra, điều này thực sự làm giảm kích thước không gian ảnh đầu vào. Tại sao lại thế này?

Nhớ lại việc tính toán diễn ra xung quanh trung tâm (x, y) - phối hợp ảnh đầu vào mặt nạ đặc trưng hiện đang được định vị. Định vị này ngụ ý rằng không có thứ gì như pixel trung tâm cho các pixel nằm dọc theo đường viền ảnh (vì các góc mặt nạ đặc trưng bị treo lơ lửng trên ảnh trong đó các giá trị không được xác định), được mô tả trong Hình 2.3.

Việc giảm kích thước không gian chỉ đơn giản là một tác dụng phụ việc áp dụng các cấu trúc cho ảnh. Đôi khi hiệu ứng này là mong muốn, và lần khác thì không, nó chỉ đơn giản phụ thuộc vào ứng dụng. Tuy nhiên, trong hầu hết các trường hợp, mong muốn ảnh đầu ra có cùng kích thước với ảnh đầu vào. Để đảm bảo kích thước giống nhau, áp dụng phần đệm (dòng 15-18). Ở đây chỉ đơn giản là sao chép các pixel dọc theo đường viền ảnh, sao cho ảnh đầu ra khớp kích thước ảnh đầu vào.

Các phương pháp đệm khác tồn tại, bao gồm đệm không (lấp đầy các đường viền bằng 0 - rất phổ biến khi xây dựng mạng nơ-ron tích chập) và quấn quanh (trong đó các pixel viền được xác định bằng cách kiểm tra

phía đối diện ảnh). Trong hầu hết các trường hợp, thấy rằng hoặc sao chép hoặc đệm không.

Bây giờ đã sẵn sàng để áp dụng tích chập thực tế cho ảnh:

```
20     # loop over the input image, "sliding" the kernel across
21     # each (x, y)-coordinate from left-to-right and top-to-bottom
22     for y in np.arange(pad, iH + pad):
23         for x in np.arange(pad, iW + pad):
24             # extract the ROI of the image by extracting the
25             # *center* region of the current (x, y)-coordinates
26             # dimensions
27             roi = image[y - pad:y + pad + 1, x - pad:x + pad + 1]
28
29             # perform the actual convolution by taking the
30             # element-wise multiplication between the ROI and
31             # the kernel, then summing the matrix
32             k = (roi * K).sum()
33
34             # store the convolved value in the output (x, y)-
35             # coordinate of the output image
36             output[y - pad, x - pad] = k
```

-1	0	+1				
-2	101	+2	232	84	91	207
-1	104	+1	139	101	237	109
	243	26	252	196	135	126
	185	135	230	48	61	225
	157	124	25	14	102	108
	5	155	116	218	232	249

Hình 2.3: Nếu đã có găng áp dụng tích chập tại pixel nằm ở $(0, 0)$, thì mặt nạ đặc trưng 3×3 treo lồng ra khỏi rìa ảnh. Lưu ý cách không có giá trị pixel ảnh đầu vào cho hàng đầu tiên và cột đầu tiên mặt nạ đặc trưng. Do đó, luôn luôn (1) bắt đầu tích chập ở vị trí hợp lệ đầu tiên hoặc (2) áp dụng phần đệm bằng 0 (được đề cập ở phần sau chương này).

Các dòng 22 và 23 lặp trên ảnh, trượt mặt nạ đặc trưng từ trái sang phải và từ trên xuống dưới, mỗi pixel một pixel. Dòng 27 trích xuất vùng quan tâm (ROI) từ ảnh bằng cách sử dụng phân đoạn mảng NumPy. Các ROI được tập trung xung quanh các tọa độ hiện tại (x, y) ảnh. ROI cũng có cùng kích thước với mặt nạ đặc trưng điều này rất quan trọng cho bước tiếp theo.

Quá trình chuyển đổi được thực hiện trên Dòng 32 bằng cách lấy phép nhân phần tử giữa các ROI và mặt nạ đặc trưng, tiếp theo là tổng hợp các mục trong ma trận. Giá trị đầu ra k sau đó được lưu trữ trong mảng đầu ra ở cùng (x, y) phối hợp (liên quan đến ảnh đầu vào).

Bây giờ có thể kết thúc phương thức tích chập:

```
38     # rescale the output image to be in the range [0, 255]
39     output = rescale_intensity(output, in_range=(0, 255))
40     output = (output * 255).astype("uint8")
41
42     # return the output image
43     return output
```

Khi làm việc với ảnh, thường xử lý các giá trị pixel nằm trong phạm vi [0, 255]. Tuy nhiên, khi áp dụng kết quả, có thể dễ dàng thu được các giá trị nằm ngoài phạm vi này. Để đưa ảnh đầu ra trở lại phạm vi [0, 255], áp dụng hàm rescale_int mật độ ảnh scikit (Dòng 39).

Chuyển đổi ảnh trở lại kiểu dữ liệu số nguyên 8 bit không dấu trên Dòng 40 (trước đây, ảnh đầu ra là loại dấu phẩy động để xử lý các giá trị pixel nằm ngoài phạm vi [0, 255]). Cuối cùng, ảnh đầu ra được trả về hàm gọi trên dòng 43.

Bây giờ, đã xác định hàm xác thực, chuyển sang phần trình điều khiển tập lệnh. Phần này chương trình xử lý phân tích các đối số dòng lệnh, xác định một loạt các mặt nạ đặc trưng áp dụng cho ảnh và sau đó hiển thị kết quả đầu ra:

```
45     # construct the argument parse and parse the arguments
46     ap = argparse.ArgumentParser()
47     ap.add_argument("-i", "--image", required=True,
48                     help="path to the input image")
49     args = vars(ap.parse_args())
```

Đoạn chương trình chỉ yêu cầu một đối số dòng lệnh, --image, là đường dẫn đến ảnh đầu vào. Sau đó có thể định nghĩa hai mặt nạ đặc trưng được sử dụng để làm mờ và làm mịn ảnh:

```
51     # construct average blurring kernels used to smooth an image
52     smallBlur = np.ones((7, 7), dtype="float") * (1.0 / (7 * 7))
53     largeBlur = np.ones((21, 21), dtype="float") * (1.0 / (21 * 21))
```

Để thuyết phục bản thân rằng mặt nạ đặc trưng này đang thực hiện làm mờ, chú ý rằng mỗi mục trong mặt nạ đặc trưng là trung bình $1 / S$, trong đó S là tổng số mục trong ma trận. Do đó, mặt nạ đặc trưng này nhân mỗi pixel đầu vào với một phần nhỏ và lấy tổng - đây chính xác là định nghĩa mức trung bình.

Sau đó có một mặt nạ đặc trưng chịu trách nhiệm làm sắc nét một ảnh:

```
55 # construct a sharpening filter
56 sharpen = np.array(
57     [0, -1, 0],
58     [-1, 5, -1],
59     [0, -1, 0]), dtype="int")
```

Sau đó, mặt nạ đặc trưng Laplacian được sử dụng để phát hiện các vùng giống như cạnh:

```
61 # construct the Laplacian kernel used to detect edge-like
62 # regions of an image
63 laplacian = np.array(
64     [0, 1, 0],
65     [1, -4, 1],
66     [0, 1, 0]), dtype="int")
```

Các mặt nạ đặc trưng Sobel có thể được sử dụng để phát hiện các vùng giống như cạnh dọc theo cả trục x và y, tương ứng:

```
68 # construct the Sobel x-axis kernel
69 sobelX = np.array(
70     [-1, 0, 1],
71     [-2, 0, 2],
72     [-1, 0, 1]), dtype="int")
73
74 # construct the Sobel y-axis kernel
75 sobelY = np.array(
76     [-1, -2, -1],
77     [0, 0, 0],
78     [1, 2, 1]), dtype="int")
```

Và cuối cùng, xác định mặt nạ đặc trưng nổi lên:

```
80 # construct an emboss kernel
81 emboss = np.array(
82     [-2, -1, 0],
83     [-1, 1, 1],
84     [0, 1, 2]), dtype="int")
```

Việc giải thích làm thế nào mỗi mặt nạ đặc trưng này được tạo ra nằm ngoài phạm vi cuốn sách này, vì vậy, hiện tại chỉ cần hiểu rằng đây là những mặt nạ đặc trưng được chế tạo thủ công để thực hiện một thao tác nhất định.

Để xử lý triệt để cách thức mặt nạ đặc trưng được xây dựng và chứng minh toán học để thực hiện một thao tác xử lý ảnh nhất định, vui lòng tham khảo Szeliksi (Chương 3) [48] hoặc từ Setosa.io [49].

Với tất cả các mặt nạ đặc trưng này, có thể gộp chúng lại với nhau

thành một bộ các bộ dữ liệu được gọi là ngân hàng mặt nạ đặc trưng:

```
86 # construct the kernel bank, a list of kernels we're going to apply
87 # using both our custom 'convolve' function and OpenCV's 'filter2D'
88 # function
89 kernelBank = (
90     ("small_blur", smallBlur),
91     ("large_blur", largeBlur),
92     ("sharpen", sharpen),
93     ("laplacian", laplacian),
94     ("sobel_x", sobelX),
95     ("sobel_y", sobelY),
96     ("emboss", emboss))
```

Xây dựng danh sách các mặt nạ đặc trưng này cho phép sử dụng để lặp lại chúng và trực quan hóa đầu ra chúng một cách hiệu quả, vì chương trình dưới đây thể hiện:

```
98 # load the input image and convert it to grayscale
99 image = cv2.imread(args["image"])
100 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
101
102 # loop over the kernels
103 for (kernelName, K) in kernelBank:
104     # apply the kernel to the grayscale image using both our custom
105     # 'convolve' function and OpenCV's 'filter2D' function
106     print("[INFO] applying {} kernel".format(kernelName))
107     convolveOutput = convolve(gray, K)
108     opencvOutput = cv2.filter2D(gray, -1, K)
109
110     # show the output images
111     cv2.imshow("Original", gray)
112     cv2.imshow("{} - convolve".format(kernelName), convolveOutput)
113     cv2.imshow("{} - opencv".format(kernelName), opencvOutput)
114     cv2.waitKey(0)
115     cv2.destroyAllWindows()
```

Các dòng 99 và 100 tải ảnh từ ổ cứng và chuyển đổi nó sang thang độ xám. Các toán tử chuyển đổi có thể và được áp dụng cho RGB hoặc các khối lượng đa kênh khác, nhưng để đơn giản, chỉ áp dụng các bộ lọc cho các ảnh thang độ xám.

Bắt đầu lặp qua bộ mặt nạ đặc trưng trong ngân hàng mặt nạ trên Dòng 103 và sau đó áp dụng mặt nạ đặc trưng hiện tại cho ảnh màu xám trên Dòng 104 bằng cách gọi phương thức xác thực hàm được xác định trước đó trong tập lệnh.

Như một kiểm tra có nhiều hay không, hàm cv2.filter2D cũng áp dụng mặt nạ đặc trưng cho ảnh xám. Hàm cv2.filter2D là OpenCV, phiên bản tối ưu hóa hơn nhiều hàm tích chập.

Cuối cùng, dòng 111-115 hiển thị ảnh đầu ra màn hình cho từng loại mặt nạ đặc trưng.

2.1.1.6. Kết quả phép tích chập

Để chạy tập lệnh (và trực quan hóa đầu ra các hoạt động tích chập khác nhau), chỉ cần đưa ra lệnh sau:

```
$ python convolutions.py --image jemma.png
```

Sau đó, thấy kết quả việc áp dụng mặt nạ đặc trưng smallBlur cho ảnh đầu vào trong Hình 2.4. Ở bên trái, có ảnh ban đầu. Sau đó, ở trung tâm, có kết quả từ hàm chập. Và bên phải, kết quả từ cv2.filter2D. Kiểm tra trực quan nhanh cho thấy rằng đầu ra khớp với cv2.filter2D, có nghĩa là hàm xác thực đang hoạt động đúng. Hơn nữa, ảnh bây giờ xuất hiện mờ và mịn.

Áp dụng một độ mờ lớn hơn, kết quả có thể được nhìn thấy trong Hình 2.5 (trên cùng bên trái). Lần này bỏ qua kết quả cv2.filter2D để tiết kiệm dung lượng. So sánh các kết quả từ Hình 2.5 đến Hình 2.4, chú ý làm thế nào khi kích thước mặt nạ đặc trưng trung bình tăng lên, lượng mờ trong ảnh đầu ra cũng tăng theo.

Ngoài ra, có thể làm sắc nét ảnh (Hình 2.5, top-mid) và phát hiện các vùng giống như cạnh thông qua toán tử Laplacian (trên cùng bên phải).

Mặt nạ đặc trưng sobel X được sử dụng để tìm các cạnh dọc trong ảnh (Hình 2.5, dưới cùng bên trái), trong khi mặt nạ đặc trưng sobel Y cho thấy các cạnh ngang (giữa dưới). Cuối cùng, có thể thấy kết quả mặt nạ đặc trưng nổi lên ở phía dưới bên trái.



Hình 2.4: Trái: Ảnh đầu vào ban đầu. Trung tâm: Áp dụng độ mờ trung bình 7×7 bằng hàm xác thực tùy chỉnh. Phải: Áp dụng hiệu ứng làm mờ 7×7 tương tự bằng cách sử dụng OpenCV cv2.filter2D - chú ý cách đầu ra hai hàm giống hệt nhau, ngũ ý rằng phương thức xác thực được thực hiện chính xác.

2.1.1.7. Vai trò tích chập trong học sâu

Khi nắm vững phần này, phải tự xác định thủ công từng mặt nạ đặc trưng cho từng hoạt động xử lý ảnh khác nhau chẳng hạn như làm mịn, làm sắc nét và phát hiện cạnh. Điều đó rất tốt và tốt, nhưng nếu có cách nào để tìm hiểu các bộ lọc này thì sao?

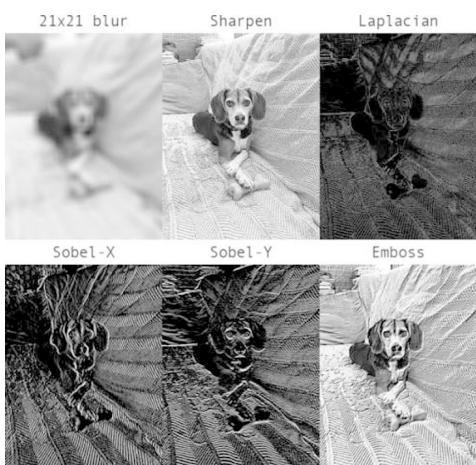
Có thể định nghĩa một thuật toán học máy có thể nhìn vào ảnh đầu vào và cuối cùng tìm hiểu các loại toán tử này không? Trên thực tế, có những loại thuật toán này là trọng tâm chính của cuốn sách này: Mạng nơ-ron tích chập (CNN).

Bằng cách áp dụng các bộ lọc tích chập, hàm tác động phi tuyến, gộp chung và truyền ngược. Sau đó sử dụng các cạnh và cấu trúc làm khôi xây dựng, cuối cùng phát hiện các vật thể cấp cao (ví dụ, khuôn mặt, mèo, chó, cốc, v.v.) trong các lớp sâu hơn mạng.

Quá trình sử dụng các lớp cấp thấp hơn này để tìm hiểu các tính năng cấp cao chính xác là tính tổng hợp các CNN đã đề cập trước đó. Nhưng chính xác làm thế nào để CNN làm điều này? Câu trả lời là bằng cách xếp chồng một tập hợp các lớp cụ thể một cách có chủ đích. Trong phần tiếp theo thảo luận về các loại lớp này, sau đó là kiểm tra các mẫu xếp chồng lớp phổ biến được sử dụng rộng rãi trong số nhiều tác vụ phân loại ảnh.

2.1.2. Các khôi xây dựng mạng CNN

Như đã học từ Chương 2, các mạng nơ-ron chấp nhận một vecto ảnh / tính năng đầu vào (một nút đầu vào cho mỗi mục nhập) và biến đổi nó thông qua một loạt các lớp ẩn, thường sử dụng các hàm tác động phi tuyến. Mỗi lớp ẩn cũng được tạo thành từ một tập hợp các nơ-ron, trong đó mỗi nơ-ron được kết nối đầy đủ với tất cả các nơ-ron ở lớp trước. Lớp cuối cùng mạng nơ-ron cũng được kết nối đầy đủ và thể hiện các phân loại đầu ra cuối cùng mạng.



Hình 2.5: Trên cùng bên trái: Áp dụng độ mờ trung bình 21×21 . Lưu ý cách ảnh này mờ hơn trong Hình 2.4. Top-mid: Sử dụng mặt nạ đặc trưng mài để tăng cường chi tiết. Trên cùng bên phải: Phát hiện các hoạt động giống như cạnh thông qua toán tử Laplacian. Dưới cùng bên trái: Tính toán các cạnh dọc bằng mặt nạ đặc trưng Sobel-X. Dưới cùng giữa: Tìm các cạnh ngang bằng mặt nạ đặc trưng Sobel-Y. Dưới cùng bên phải: Áp dụng một mặt nạ đặc trưng dập nổi.

Tuy nhiên, như kết quả phần 2.1.4 chứng minh, các mạng nơ-ron hoạt động trực tiếp trên cường độ pixel thô:

1. Không tỷ lệ tốt khi kích thước ảnh tăng lên.

2. Không cung cấp độ chính xác như mong muốn (nghĩa là, mạng lan truyền thẳng tiêu chuẩn trên CIFAR-10 chỉ thu được độ chính xác 15%).

Để chứng minh làm thế nào các mạng nơ-ron tiêu chuẩn không mở rộng tốt khi kích thước ảnh tăng lên, lại một lần nữa xem xét bộ dữ liệu CIFAR-10. Mỗi ảnh trong CIFAR-10 là 32×32 với kênh Đỏ, Xanh lục và Xanh lam, mang lại tổng cộng $32 \times 32 \times 3 = 3,072$ tổng số đầu vào cho mạng.

Tổng cộng có 3,072 đầu vào, con số này dường như không nhiều, nhưng xem xét nếu đang sử dụng 250 ảnh 250 pixel - tổng số đầu vào và trọng số nhảy lên $250 \times 250 \times 3 = 187500$ - và con số này chỉ dành cho lớp đầu vào! Chắc chắn, muốn thêm nhiều lớp ẩn với số lượng nút khác nhau trên mỗi lớp - các tham số này có thể nhanh chóng tăng lên và do hiệu suất kém của các mạng nơ-ron tiêu chuẩn về cường độ pixel thô, sự phình to này hầu như không đáng.

Thay vào đó, có thể sử dụng mạng nơ-ron tích chập (CNN) tận dụng cấu trúc ảnh đầu vào và xác định kiến trúc mạng theo cách hợp lý hơn. Không giống như mạng nơ-ron tiêu chuẩn, các lớp CNN được sắp xếp theo một khối 3D theo ba chiều: chiều rộng, chiều cao và chiều sâu (trong đó độ sâu đề cập đến chiều thứ ba âm lượng, chẳng hạn như số lượng kênh trong ảnh hoặc số các bộ lọc trong một lớp).

Để làm cho ví dụ này cụ thể hơn, một lần nữa xem xét bộ dữ liệu CIFAR-10: khối lượng đầu vào có kích thước $32 \times 32 \times 3$ (tương ứng chiều rộng, chiều cao và chiều sâu). Các nơ-ron ở các lớp tiếp theo chỉ được kết nối với một vùng nhỏ lớp trước nó (chứ không phải là cấu trúc được kết nối đầy đủ mạng nơ-ron tiêu chuẩn) - gọi kết nối cục bộ này cho phép lưu một lượng lớn các tham số trong mạng. Cuối cùng, lớp đầu ra là một khối lượng $1 \times 1 \times N$ đại diện cho ảnh được chắt lọc vào một vectơ duy nhất theo lớp điểm số. Trong trường hợp CIFAR-10, đưa ra mười lớp, $N = 10$, mang lại khối lượng $1 \times 1 \times 10$.

2.1.2.1. Các loại lớp

Có nhiều loại lớp được sử dụng để xây dựng mạng nơ-ron tích chập, nhưng những loại có thể gộp bao gồm:

- Lớp tích chập (CONV)

- Lớp kích hoạt (ACT hoặc RELU, trong đó sử dụng cùng hàm tác động thực tế)

- Lớp tập hợp (POOL)
- Lớp Kết nối đầy đủ (FC)
- Bỏ chuẩn hóa hàng loạt (BN)
- Bỏ học (DO)

Xếp chồng một loạt các lớp này theo cách cụ thể sẽ mang lại một CNN, thường sử dụng các sơ đồ đơn giản để mô tả một CNN: INPUT => CONV => RELU => FC => SOFTMAX

Ở đây, xác định một CNN đơn giản chấp nhận đầu vào, áp dụng lớp tích chập, sau đó là lớp kích hoạt, sau đó là lớp được kết nối đầy đủ và cuối cùng là phân loại softmax để có được xác suất phân loại đầu ra. Lớp kích hoạt SOFTMAX thường được bỏ qua khỏi sơ đồ mạng vì được giả định là nó kết nối trực tiếp với FC cuối cùng.

Trong số các loại lớp này, CONV và FC, (và ở mức độ thấp hơn, BN) là các lớp duy nhất chứa các tham số được học trong quá trình huấn luyện. Bản thân các lớp kích hoạt và bỏ học không được coi là các lớp thực sự, nhưng thường được bao gồm trong sơ đồ mạng để làm cho kiến trúc mạng rõ ràng. Các lớp gộp (POOL), có tầm quan trọng tương đương như CONV và FC, cũng được đưa vào sơ đồ mạng như chúng có một tác động đáng kể đến kích thước không gian ảnh khi nó di chuyển qua CNN.

CONV, POOL, RELU và FC là quan trọng nhất khi xác định kiến trúc mạng thực tế. Điều đó không nói rằng các lớp khác không quan trọng, nhưng đặt ngược lại với bộ bốn lớp quan trọng này khi chúng xác định chính kiến trúc thực tế.

Các hàm tác động thực tế được coi là một phần kiến trúc, Khi xác định kiến trúc CNN, thường bỏ qua các lớp kích hoạt từ bảng / sơ đồ để tiết kiệm không gian; tuy nhiên, các lớp kích hoạt được mặc định là một phần kiến trúc.

Trong phần còn lại, xem xét chi tiết từng loại lớp này và thảo luận về các tham số liên quan đến từng lớp (và cách đặt chúng). Phần sau chương này, thảo luận chi tiết hơn về cách xếp các lớp này đúng cách để xây dựng các kiến trúc CNN riêng.

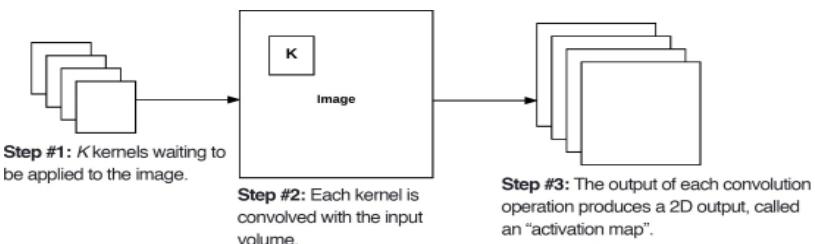
2.1.2.2. Lớp tích chập

Lớp CONV là khái niệm cốt lõi của một CNN. Các

tham số lớp CONV bao gồm một tập hợp các bộ lọc K có thể học được (tức là, mặt nạ đặc trưng), trong đó mỗi bộ lọc có chiều rộng và chiều cao và gần như luôn luôn vuông. Các bộ lọc này nhỏ (về kích thước không gian của chúng) nhưng mở rộng trong toàn bộ chiều sâu.

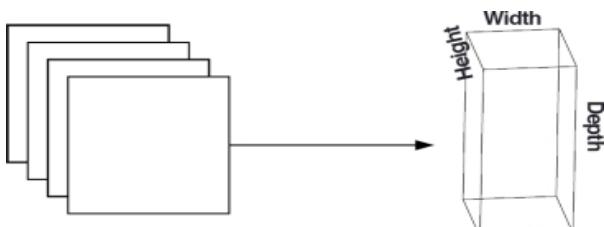
Đối với các đầu vào cho CNN, độ sâu là số kênh trong ảnh (nghĩa là độ sâu là 3 khi làm việc với ảnh RGB, một cho mỗi kênh). Đối với khối lượng sâu hơn trong mạng, độ sâu là số lượng bộ lọc được áp dụng trong lớp trước.

Để làm cho khái niệm này rõ ràng hơn, xem xét việc chuyển tiếp CNN, nơi tích chập từng bộ lọc K qua chiều rộng và chiều cao của dữ liệu đầu vào, giống như đã làm trong Phần thực thi tích chập với Python ở trên. Đơn giản hơn, có thể nghĩ về mỗi mặt nạ đặc trưng K trượt qua vùng đầu vào, tính toán một phép nhân phần tử, tính tổng, và sau đó lưu trữ giá trị đầu ra trong bản đồ kích hoạt 2 chiều, như trong Hình 2.6.



Hình 2.6: Trái: Tại mỗi lớp chập trong CNN, có các mặt nạ đặc trưng K được áp dụng cho khối lượng đầu vào. Giữa: Mỗi mặt nạ đặc trưng K được tích hợp với số lượng đầu vào. Phải: Mỗi mặt nạ đặc trưng tạo ra một đầu ra 2D, được gọi là bản đồ kích hoạt.

Sau khi áp dụng tất cả các bộ lọc K cho khối lượng đầu vào, giờ có các bản đồ kích hoạt $K \times 2$ chiều. Sau đó, xếp các bản đồ kích hoạt K dọc theo chiều sâu của mảng để tạo thành khối lượng đầu ra cuối cùng (Hình 2.7).



Hình 2.7: Sau khi có được các bản đồ kích hoạt K , chúng được xếp chồng lên nhau để tạo thành khối lượng đầu ra cho lớp tiếp theo trong mạng.

Do đó, mọi lối vào trong số lượng đầu ra là một nơ-ron mà chỉ nhìn vào một vùng nhỏ của đầu vào. Theo cách này, mạng học các bộ lọc kích hoạt khi thấy một loại tính năng cụ thể tại một vị trí không gian nhất định trong số lượng đầu vào. Ở các lớp thấp hơn của mạng, các bộ lọc có thể kích hoạt khi chúng nhìn thấy các vùng giống như cạnh hoặc góc.

Sau đó, ở các lớp sâu hơn của mạng, các bộ lọc có thể kích hoạt khi có các tính năng cấp cao, chẳng hạn như các bộ phận khuôn mặt, chân của con chó, mui xe, v.v. Khái niệm kích hoạt này quay trở lại với sự tương tự mạng nơ-ron trong Chương 2 - những tế bào nơ-ron này đang trở nên kích thích và kích hoạt hệ thống khi thấy một mô hình cụ thể trong một ảnh đầu vào.

Khái niệm tích chập một bộ lọc nhỏ với số lượng đầu vào (r) lớn có ý nghĩa đặc biệt trong mạng nơ-ron tích chập - đặc biệt là sự kết nối cục bộ và lĩnh vực tiếp nhận của một tế bào nơ-ron. Khi làm việc với ảnh, nó thường không thực tế khi kết nối các nơ-ron trong số lượng hiện tại với tất cả các nơ-ron ở số lượng trước - đơn giản là có quá nhiều kết nối và quá nhiều trọng số, khiến cho không thể huấn luyện các mạng sâu trên ảnh với kích thước không gian lớn. Thay vào đó, khi sử dụng CNN, chọn kết nối mỗi nơ-ron chỉ với một vùng cục bộ của số lượng đầu vào - gọi kích thước của vùng cục bộ này là trường tiếp nhận (hay đơn giản là biến F) của nơ-ron.

Để làm rõ điểm này, quay lại tập dữ liệu CIFAR-10 với khối lượng đầu vào là kích thước đầu vào là $32 \times 32 \times 3$. Mỗi ảnh có chiều rộng 32 pixel, chiều cao 32 pixel và độ sâu 3 (một cho mỗi kênh RGB). Nếu trường tiếp nhận có kích thước 3×3 , thì mỗi nơ-ron trong lớp CONV kết nối với vùng 3×3 cục bộ ảnh với tổng số $3 \times 3 \times 3 = 27$ trọng số (nhớ rằng độ sâu các bộ lọc là 3 vì chúng mở rộng qua toàn bộ độ sâu ảnh đầu vào, trong trường hợp này là 3 kênh).

Bây giờ, giả sử rằng không gian số lượng đầu vào đã giảm xuống kích thước nhỏ hơn, nhưng độ sâu hiện lớn hơn, do sử dụng nhiều bộ lọc sâu hơn trong mạng, do đó kích thước số lượng hiện là $16 \times 16 \times 94$. Một lần nữa, nếu giả sử trường tiếp nhận có kích thước 3×3 , thì mỗi nơ-ron trong lớp CONV có tổng cộng $3 \times 3 \times 94 = 846$ kết nối với số lượng đầu vào.

Nói một cách đơn giản, trường tiếp nhận F là kích thước bộ lọc, mang lại mặt nạ đặc trưng F được tích chập với số lượng đầu vào.

Tại thời điểm này, đã giải thích sự kết nối các nơ-ron trong số lượng đầu vào, nhưng không phải là sự sắp xếp hoặc kích thước số lượng đầu ra. Có ba tham số kiểm soát kích thước số lượng đầu ra: độ sâu, bước trượt và kích thước đệm phần tử hông, mỗi tham số xem xét bên dưới.

Độ sâu

Độ sâu số lượng đầu ra kiểm soát số lượng tế bào nơ-ron (tức là, bộ lọc) trong lớp CONV kết nối với một vùng cục bộ số lượng đầu vào. Mỗi bộ lọc tạo ra một bản đồ kích hoạt sẽ kích hoạt trên mạng với sự hiện diện các cạnh hoặc đốm màu hoặc định hướng.

Đối với một lớp CONV nhất định, độ sâu bản đồ kích hoạt sẽ là K hoặc đơn giản là số lượng bộ lọc đang học trong lớp hiện tại. Tập hợp các bộ lọc nhìn vào vị trí tương tự (x, y) đầu vào được gọi là cột độ sâu.

Bước trượt

Xem xét Hình 2.1 trước đó trong phần này, nơi đã mô tả một hoạt động tích chập khi trượt một ma trận nhỏ trên một ma trận lớn, dừng lại ở mỗi tọa độ, tính toán một phép nhân và tổng phần tử, sau đó lưu trữ đầu ra.

Trong Mục thực thi tích chập bằng Python ở trên, chỉ thực hiện một bước một pixel mỗi đinh. Trong CNN, có thể áp dụng nguyên tắc tương tự - cho mỗi bước, tạo một cột độ sâu mới xung quanh khu vực địa phương ảnh nơi tích chập từng bộ lọc K với vùng và lưu trữ đầu ra trong một khối 3D. Khi tạo các lớp CONV, thường sử dụng một bước trượt kích thước $S = 1$ hoặc $S = 2$.

Những bước tiến nhỏ hơn dẫn đến các lĩnh vực tiếp nhận chồng chéo và khối lượng đầu ra lớn hơn. Ngược lại, những bước tiến lớn hơn dẫn đến các trường tiếp nhận ít chồng chéo hơn và khối lượng đầu ra nhỏ hơn. Để làm cho khái niệm bước trượt cụ thể hơn, xem xét Bảng 2.1 trong đó có 5×5 ảnh đầu vào (trái) cùng với mặt nạ đặc trưng 3×3 Laplacian (phải).

Sử dụng $S = 1$, mặt nạ đặc trưng trượt từ trái sang phải và từ trên xuống dưới, mỗi pixel một pixel, tạo ra đầu ra sau (Hình 2.2, bên trái). Tuy nhiên, nếu áp dụng thao tác tương tự, lần này với bước trượt là $S = 2$, chúng ta bỏ qua hai pixel cùng một lúc (hai pixel đọc theo trục x và hai pixel đọc theo trục y), tạo ra số lượng đầu ra nhỏ hơn (phải).

Bảng 2.1: Ảnh 5×5 đầu vào (bên trái) tích chập với mặt nạ đặc trưng Laplacian (đúng)

95	242	186	152	39				
39	14	220	153	180	0	1	0	
5	247	212	54	46	1	-4	1	
46	77	133	110	74	0	1	0	
156	35	74	93	116				

Bảng 2.2: Trái: Đầu ra tích chập với bước trượt 1×1 . Phải: Đầu ra tích chập với 2×2 bước trượt. Lưu ý cách một bước tiến lớn hơn có thể làm giảm kích thước không gian đầu vào.

692	-315	-6		
-680	-194	305	692	-6
153	-59	-86	153	-86

Vì vậy, có thể thấy làm thế nào các lớp chập có thể được sử dụng để giảm kích thước không gian các khối đầu vào chỉ bằng cách thay đổi bước tiến mặt nạ đặc trưng. Như đã thấy, các lớp chập và các lớp gộp là các phuong thức chính để giảm kích thước đầu vào không gian. Phần các lớp gộp cũng cung cấp một ví dụ trực quan hơn về cách các kích cỡ bước trượt khác nhau ảnh hưởng đến kích thước đầu ra.

Zero-Padding

Như đã biết từ Phần thực thi tích chập với Python, cần phải thay đổi đường viền ảnh để giữ lại kích thước ảnh ban đầu khi áp dụng tích chập - điều này cũng đúng với các bộ lọc bên trong CNN. Sử dụng không đệm, có thể đệm pad đầu vào dọc theo đường viền sao cho kích thước số lượng đầu ra phù hợp với kích thước số lượng đầu vào. Lượng đệm áp dụng được kiểm soát bởi tham số P.

Kỹ thuật này đặc biệt quan trọng khi bắt đầu xem xét các kiến trúc CNN sâu, áp dụng nhiều bộ lọc CONV lén nhau. Để trực quan hóa phần đệm bằng 0, một lần nữa tham khảo Bảng 2.1 trong đó đã áp dụng mặt nạ đặc trưng 3×3 Laplacian cho ảnh đầu vào 5×5 với bước trượt S = 1. Có thể thấy trong Bảng 2.3 (bên trái) cách số lượng đầu ra nhỏ hơn (3×3) so với số lượng đầu vào (5×5) do tính chất hoạt động tích chập. Thay vào đó, nếu đặt P = 1, có thể đệm số lượng đầu vào bằng 0 (giữa) để tạo số lượng 7×7

và sau đó áp dụng thao tác tích chập, dẫn đến kích thước số lượng đầu ra phù hợp với kích thước số lượng đầu vào ban đầu là 5×5 (phải).

Nếu không có phần đệm bằng 0, kích thước không gian số lượng đầu vào giảm quá nhanh và có thể huân luyện các mạng sâu (vì khối lượng đầu vào quá nhỏ để học bất kỳ mẫu hữu ích nào).

Đặt tất cả các tham số này lại với nhau, có thể tính kích thước số lượng đầu ra là một hàm kích thước số lượng đầu vào (W , giả sử ảnh đầu vào là hình vuông, gần như luôn luôn), kích thước trường tiếp nhận F , bước trượt S , và số lượng không đệm P . Để xây dựng một lớp CONV hợp lệ, cần đảm bảo phương trình sau là một số nguyên:

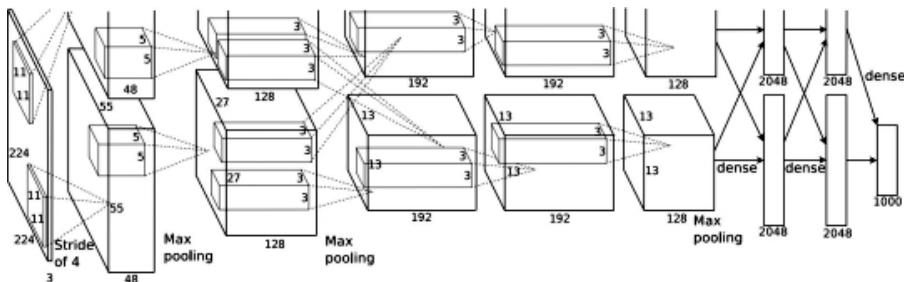
$$((W - F + 2P)/S) + 1 \quad (2.3)$$

Nếu nó không phải là một số nguyên, thì các bước được đặt không chính xác và các nơ-ron không thể được xếp theo thứ tự sao cho chúng khớp với số lượng đầu vào theo cách đối xứng.

Bảng 2.3: Bên trái: Kết quả của phép tích chập ma trận có kích thước 3×3 với 1 ma trận 5×5 (kích thước ma trận gốc đã giảm). Phải: Áp dụng đệm các giá trị 0 cho đầu vào ban đầu với $P = 1$ làm tăng kích thước ma trận lên 7×7 . Dưới cùng: Sau khi áp dụng tích chập 3×3 cho đầu vào đệm, kích thước ma trận đầu ra bằng với kích thước ma trận đầu vào ban đầu là 5×5 , do đó đệm các giá trị 0 giúp bảo tồn kích thước ma trận.

			0	0	0	0	0	0	0
			0	95	242	186	152	39	0
692	-315	-6	0	39	14	220	153	180	0
-680	-194	305	0	5	247	212	54	46	0
153	-59	-86	0	46	77	133	110	74	0
			0	156	35	74	93	116	0
			0	0	0	0	0	0	0
-99	-673	-130	-230	176					
-42	692	-315	-6	-482					
312	-680	-194	305	124					
54	153	-59	-86	-24					
-543	167	-35	-72	-297					

Ví dụ, xem xét lớp đầu tiên của kiến trúc AlexNet, mạng đã chiến thắng trong cuộc thi phân loại ImageNet 2012, đóng góp lớn cho sự phát triển của học sâu hiện nay và được áp dụng cho phân loại ảnh. Trong bài báo của họ, Krizhevsky et al. [50] ghi lại kiến trúc CNN họ theo Hình 2.8.



Hình 2.8: Sơ đồ kiến trúc AlexNet ban đầu được cung cấp bởi Krizhevsky et al. [94]. Lưu ý cách ảnh đầu vào được ghi thành $224 \times 224 \times 3$.

Lưu ý cách lớp đầu tiên tuyên bố rằng kích thước ảnh đầu vào là 224×224 pixel. Tuy nhiên, điều này có thể chính xác nếu áp dụng phương trình ở trên bằng cách sử dụng 11×11 bộ lọc, một bước bốn và không có phần đệm: $((224 - 11 + 2(0)) / 4) + 1 = 54,25$ (11,7)

Tóm lại, lớp CONV theo cách tương tự như Karpathy [51]:

Chấp nhận một khối lượng đầu vào có kích thước Winput Hinput Dinput (kích thước đầu vào thường là hình vuông, do đó, nó phẳng biến để xem Winput = Hinput).

Yêu cầu bốn tham số:

1. Số bộ lọc K (điều khiển độ sâu số lượng đầu ra).
2. Kích thước trường tiếp nhận F (kích thước mặt nạ K được sử dụng để tích chập và gần như luôn vuông, mang lại mặt nạ đặc trưng $F \times F$).
3. Bước trượt S.
4. Số lượng đệm không P.

Sẽ xem xét các cài đặt chung cho các tham số này trong Phần mẫu lớp bên dưới.

2.1.2.3. Lớp kích hoạt

Sau mỗi lớp CONV trong CNN, áp dụng hàm tác động phi tuyến, như ReLU, ELU hoặc bất kỳ biến thể Leaky ReLU nào khác được đề cập trong Chương 2, thường biểu thị các lớp kích hoạt là RELU trong sơ đồ

mang vì hầu hết kích hoạt ReLU thường được sử dụng, cũng có thể chỉ đơn giản là nêu ACT - trong cả hai trường hợp, đang làm rõ rằng một hàm tác động đang được áp dụng bên trong kiến trúc mạng.

Các lớp kích hoạt không phải là các lớp kỹ thuật, các lớp (do thực tế là không có thông số / trọng số nào được học bên trong lớp kích hoạt) và đôi khi được bỏ qua khỏi các sơ đồ kiến trúc mạng vì nó giả định rằng một kích hoạt ngay lập tức theo một phép toán tích chập.

Trong trường hợp này, đề cập đến hàm tác động nào đang sử dụng sau mỗi lớp CONV ở đâu. Ví dụ, xem xét kiến trúc mạng sau: INPUT => CONV => RELU => FC.

Để làm cho sơ đồ này ngắn gọn hơn, chỉ cần loại bỏ thành phần RELU vì nó giả định rằng một kích hoạt luôn tuân theo một phép chập: INPUT => CONV => FC.

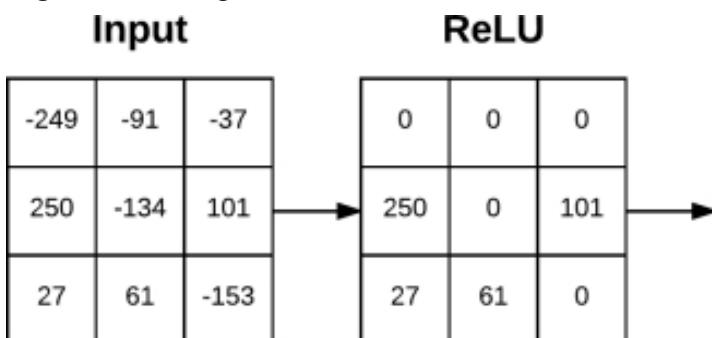
Một lớp kích hoạt chấp nhận một khối lượng đầu vào có kích thước Winput x Hinput x Dinput và sau đó áp dụng hàm tác động đã cho (Hình 2.9). Do hàm tác động được áp dụng theo từng phần tử, đầu ra lớp kích hoạt luôn giống với kích thước đầu vào

$$\text{Winput} = \text{Wout put},$$

$$\text{Hinput} = \text{Houtput}, \text{Dinput} = \text{Doutput}.$$

2.1.2.4. Lớp gộp

Có hai phương pháp để giảm kích thước một khối lượng đầu vào - các lớp CONV với bước trượt > 1 và các lớp POOL. Việc chèn các lớp POOL vào giữa là liên tiếp:



Hình 2.9: Một ví dụ về khối lượng đầu vào trải qua kích hoạt ReLU, $\max(0, x)$. Các hoạt động được thực hiện tại chỗ nên không cần tạo ra một khối lượng đầu ra riêng biệt mặc dù có thể dễ dàng hình dung được dòng chảy mạng theo cách này.

Các lớp CONV trong kiến trúc CNN:

INPUT => CONV => RELU => POOL => CONV => RELU =>
POOL => FC

Hàm chính lớp POOL là giảm dần kích thước không gian (tức là chiều rộng và chiều cao) khỏi lượng đầu vào. Làm điều này cho phép giảm số lượng tham số và tính toán trong mạng - gộp chung cũng giúp kiểm soát quá khớp.

Các lớp POOL hoạt động trên từng lớp sâu của đầu vào một cách độc lập bằng cách sử dụng hàm tối đa hoặc trung bình. Việc gộp nhóm tối đa thường được thực hiện ở giữa kiến trúc CNN để giảm kích thước không gian, trong khi nhóm trung bình thường được sử dụng làm lớp cuối cùng của mạng (ví dụ, GoogLeNet, SqueezeNet, ResNet) trong đó muốn tránh sử dụng hoàn toàn các lớp FC. Loại phổ biến nhất của lớp POOL là gộp chung tối đa, mặc dù xu hướng này đang thay đổi với sự ra đời của các kiến trúc vi mô.

Thông thường, sử dụng kích thước nhóm 2×2 , mặc dù các CNN sâu hơn sử dụng ảnh đầu vào lớn hơn (> 200 pixel) có thể sử dụng kích thước nhóm 3×3 trong kiến trúc mạng. Cũng thường đặt bước trượt là $S = 1$ hoặc $S = 2$. Hình 2.10 là một ví dụ về áp dụng nhóm tối đa với kích thước nhóm 2×2 và bước trượt $S = 1$. Lưu ý cho mỗi khối 2×2 , chỉ giữ giá trị lớn nhất, thực hiện một bước duy nhất (như cửa sổ trượt) và áp dụng lại thao tác - do đó tạo ra kích thước số lượng đầu ra là 3×3 .

Có thể giảm thêm kích thước khối lượng đầu ra bằng cách tăng bước tiến - ở đây áp dụng $S = 2$ cho cùng một đầu vào (Hình 2.10, dưới cùng). Đối với mỗi khối 2×2 trong đầu vào, chỉ giữ giá trị lớn nhất, sau đó thực hiện bước hai pixel và áp dụng lại thao tác. Việc gộp này cho phép giảm chiều rộng và chiều cao xuống hai lần, loại bỏ hiệu quả 75% kích hoạt từ lớp trước.

Tóm lại, các lớp POOL chấp nhận một khối lượng đầu vào có kích thước $W_{\text{input}} \times H_{\text{input}} \times D_{\text{input}}$. Sau đó, yêu cầu hai tham số:

- Kích thước trường tiếp nhận F
- Bước trượt S .

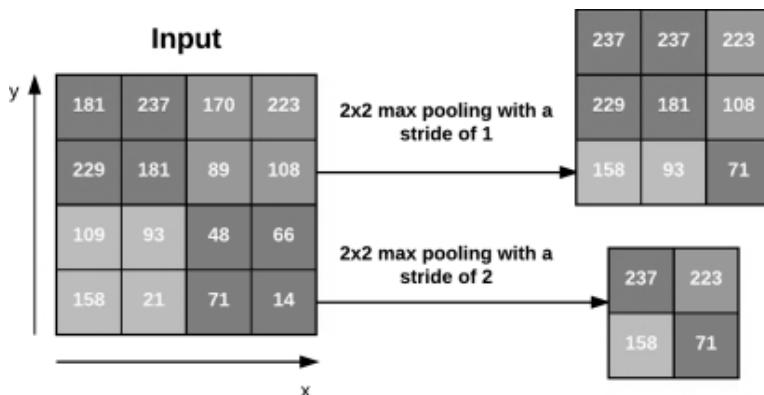
Áp dụng hoạt động POOL mang lại một khối lượng đầu ra có kích thước $W_{\text{out}} \times H_{\text{out}} \times D_{\text{out}}$,

trong đó:

- * $W_{out} = ((W_{input} - F)/S) + 1$
- * $H_{out} = ((H_{input} - F)/S) + 1$
- * $D_{output} = D_{input}$

Trong thực tế, có xu hướng thấy hai loại biến thể gộp tối đa:

* Loại 1: $F = 3$, $S = 2$ được gọi là gộp nhóm và thường được áp dụng cho khối lượng đầu vào / khối lượng đầu vào với kích thước không gian lớn.



Hình 2.10: Trái: Khối lượng 4×4 đầu vào. Phải: Áp dụng 2×2 nhóm tối đa với một bước trượt $S = 1$. Dưới cùng: Áp dụng 2×2 nhóm tối đa với $S = 2$, điều này làm giảm đáng kể kích thước không gian đầu vào.

Loại 2: $F = 2$, $S = 2$ được gọi là gộp chung. Đây là loại gộp chung nhất và được áp dụng cho ảnh có kích thước không gian nhỏ hơn.

Đối với kiến trúc mạng chấp nhận ảnh đầu vào nhỏ hơn (trong phạm vi 32×64 pixel), cũng có thể thấy $F = 2$, $S = 1$.

2.1.2.5. Lớp liên kết đầy đủ

Các nơ-ron trong các lớp FC được kết nối đầy đủ với tất cả các kích hoạt ở lớp trước, như là tiêu chuẩn cho các mạng nơ-ron tiếp theo đã thảo luận trong Chương 1. Các lớp FC luôn được đặt ở cuối mạng (nghĩa là không áp dụng một lớp CONV, sau đó là một lớp FC, tiếp theo là một lớp CONV) khác.

Thường sử dụng một hoặc hai lớp FC trước khi áp dụng trình phân loại softmax, như kiến trúc (đơn giản hóa) sau đây:

INPUT => CONV => RELU => POOL => CONV => RELU => POOL => FC => FC

Ở đây, áp dụng hai lớp được kết nối đầy đủ trước khi phân loại softmax tính toán xác suất đầu ra cuối cùng cho mỗi lớp.

2.1.2.6. Chuẩn hóa hàng loạt

Lần đầu tiên được giới thiệu bởi Ioffe và Szegedy trong bài báo năm 2015, chuẩn hóa hàng loạt: Tăng tốc huấn luyện mạng sâu bằng cách giảm sự thay đổi đồng biến nội bộ [52], gọi tắt là các lớp chuẩn hóa hàng loạt (hay gọi tắt là BN), được sử dụng để chuẩn hóa kích hoạt khối lượng đầu vào cho trước khi chuyển nó vào lớp tiếp theo trong mạng.

Nếu coi x là phần kích hoạt nhỏ, thì có thể tính \hat{x} đã chuẩn hóa thông qua phương trình sau:

$$x_i = \frac{\hat{x}_i - \mu_\beta}{\sqrt{\sigma_\beta^2 + \epsilon}} \quad (2.4)$$

Trong quá trình huấn luyện, tính toán các điểm số và $\sigma\beta$ trên mỗi đợt nhỏ, trong đó:

$$\mu_\beta = \frac{1}{M} \sum_{i=1}^m x_i \quad (2.5)$$

$$\sigma_\beta^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_\beta)^2 \quad (2.6)$$

Đặt ϵ bằng với một giá trị dương nhỏ, chẳng hạn như 1e-7 để tránh lấy căn bậc hai của 0. Áp dụng phương trình này ngụ ý rằng các kích hoạt để lại một lớp chuẩn hóa hàng loạt có giá trị trung bình và phương sai đơn vị xấp xỉ bằng 0 (nghĩa là, không tập trung).

Tại thời điểm thực nghiệm, thay thế các đợt nhỏ μ_β và σ_β bằng các mức trung bình đang chạy của chính nó và được tính toán trong quá trình huấn luyện. Điều này đảm bảo rằng có thể truyền ảnh qua mạng và vẫn có được dự đoán chính xác mà không bị sai lệch bởi μ_β và σ_β từ đợt nhỏ cuối cùng được truyền qua mạng tại thời điểm huấn luyện.

Chuẩn hóa hàng loạt đã được chứng minh là cực kỳ hiệu quả trong việc giảm số lượng chu kỳ cần thiết để huấn luyện một mạng nơ-ron. Chuẩn hóa hàng loạt cũng có thêm lợi ích là giúp ổn định việc huấn luyện, cho phép nhiều mức học và tính bền vững hơn. Tuy nhiên, việc sử dụng chuẩn hóa hàng loạt không làm giảm nhu cầu điều chỉnh các tham số này,

nhưng nó sẽ giúp dễ dàng hơn bằng cách làm cho tốc độ học và quy tắc ít biến động và dễ điều chỉnh hơn. Ngoài ra, có xu hướng nhận thấy tồn thắt cuối cùng thấp hơn và đường cong mất ổn định hơn khi sử dụng chuẩn hóa hàng loạt trong các mạng.

Hạn chế lớn nhất chuẩn hóa hàng loạt là nó thực sự có thể làm chậm thời gian treo tường để huấn luyện mạng (mặc dù cần ít chu kỳ hơn để có được độ chính xác hợp lý), khoảng 2-3 lần do tính toán thống kê theo từng đợt và chuẩn hóa.

Điều đó nói rằng, nên sử dụng chuẩn hóa hàng loạt trong gần như mọi tinh huống vì nó tạo ra sự khác biệt đáng kể. Áp dụng chuẩn hóa hàng loạt cho các kiến trúc mạng có thể giúp ngăn chặn quá khớp và cho phép có được độ chính xác phân loại cao hơn đáng kể trong ít chu kỳ hơn so với cùng một kiến trúc mạng mà không cần chuẩn hóa hàng loạt.

Vậy, các lớp chuẩn hóa hàng loạt đi đâu?

Có thể nhận thấy trong cuộc thảo luận về chuẩn hóa hàng loạt, đã bỏ qua chính xác vị trí trong kiến trúc mạng, đặt lớp chuẩn hóa hàng loạt. Trong bài báo của Ioffe và Szegedy [52], họ đã đặt chuẩn hóa hàng loạt (BN) trước khi kích hoạt:

“Thêm biến đổi BN ngay trước khi phi tuyến tính, bằng cách chuẩn hóa $x = \text{W}u + b$.”

Sử dụng sơ đồ này, một kiến trúc mạng sử dụng chuẩn hóa hàng loạt trông như thế này: NPUT => CONV => BN => RELU ...

Tuy nhiên, quan điểm về chuẩn hóa hàng loạt này không có ý nghĩa từ quan điểm thống kê. Trong bối cảnh này, một lớp BN đang chuẩn hóa việc phân phối các tính năng ra khỏi lớp CONV. Một số đặc điểm có thể âm, trong đó chúng được kẹp (tức là, được đặt lại bằng 0) bởi một hàm tác động phi tuyến tính như ReLU.

Nếu chúng ta chuẩn hóa trước khi kích hoạt, rất cần thiết bao gồm các giá trị âm bên trong chuẩn hóa. Các tính năng không tập trung sau đó được chuyển qua ReLU nơi tiêu diệt mọi kích hoạt nhỏ hơn 0 (bao gồm các tính năng có thể không âm trước khi chuẩn hóa) - lớp này hoàn toàn bãi bỏ mục đích áp dụng chuẩn hóa hàng loạt ở vị trí đầu tiên.

Thay vào đó, nếu đặt chuẩn hóa hàng loạt sau ReLU, sẽ chuẩn hóa

các tính năng có giá trị dương mà không làm sai lệch thống kê của chúng với các tính năng sẽ không được đưa vào lớp CONV tiếp theo.

Không rõ lý do tại sao Ioffe và Szegedy đề nghị đặt lớp BN trước khi kích hoạt trong bài báo của họ, nhưng các thí nghiệm tiếp theo [53] cũng như bằng chứng từ các nhà nghiên cứu sâu khác [54], xác nhận rằng việc đặt lớp chuẩn hóa hàng loạt sau khi kích hoạt phi tuyến sẽ đem lại độ chính xác cao hơn và tổn thất thấp hơn trong hầu hết các tình huống.

Đặt BN sau khi kích hoạt trong kiến trúc mạng như sau:

INPUT => CONV => RELU => BN ...

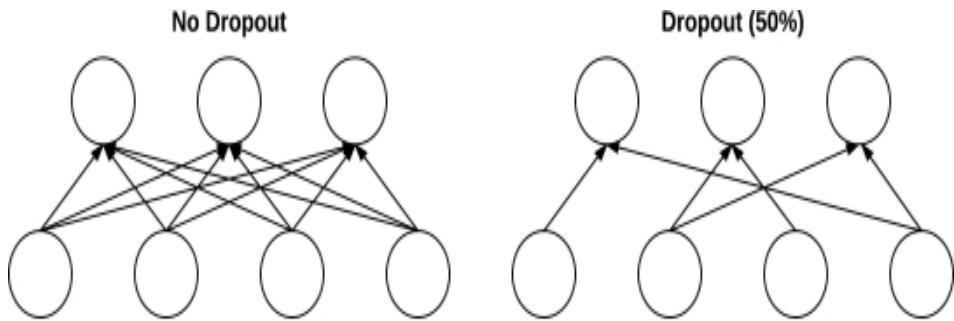
Sau khi chạy một vài trong số các thử nghiệm này, nhanh chóng nhận ra rằng BN sau khi kích hoạt hoạt động tốt hơn và có nhiều tham số quan trọng hơn để mạng điều chỉnh để có được độ chính xác phân loại cao hơn. Chúng ta sẽ thảo luận chi tiết hơn trong Mục 2.1.3.2.

2.1.2.7. Bỏ học (*DROPOUT*)

Loại lớp cuối cùng thảo luận là bỏ học. Bỏ học thực sự là một phương pháp được sử dụng nhiều nhằm mục đích giúp ngăn ngừa quá khớp và tăng độ chính xác kiểm tra, có lẽ phải trả giá bằng độ chính xác huấn luyện. Đối với mỗi đợt dữ liệu nhỏ trong tập huấn luyện các lớp bỏ học, với xác suất p , ngắt kết nối ngẫu nhiên các đầu vào từ lớp trước sang lớp tiếp theo trong kiến trúc mạng.

Hình 2.11 trực quan hóa khái niệm này, trong đó ngắt kết nối ngẫu nhiên với xác suất $p = 0,5$ kết nối giữa hai lớp FC cho một đợt dữ liệu nhỏ nhất định. Một lần nữa, chú ý cách một nửa các kết nối bị cắt đứt cho đợt nhỏ này. Sau khi tính toán lan truyền thuận và lan truyền ngược cho đợt dữ liệu nhỏ, kết nối lại các kết nối bị mất và sau đó lấy mẫu một bộ kết nối khác để bỏ học..

Lý do áp dụng bỏ học là để giảm quá khớp bằng cách thay đổi rõ ràng kiến trúc mạng trong thời gian huấn luyện. Ngắt kết nối ngẫu nhiên đảm bảo rằng không có nút nào trong mạng chịu trách nhiệm cho việc kích hoạt hệ thống điều khiển khi được trình bày với một mẫu nhất định. Thay vào đó, bỏ học đảm bảo có nhiều nút dự phòng kích hoạt khi được trình bày với các đầu vào tương tự - điều này giúp mô hình khai thác hóa.



Hình 2.11: Trái: Hai lớp mạng nơ ron được kết nối hoàn toàn không có hiện tượng bỏ học; Phải: Hai lớp giống nhau sau khi giảm 50% các kết nối.

Thông thường nhất là đặt các lớp bỏ học với $p = 0,5$ ở giữa các lớp FC, một kiến trúc trong đó lớp FC cuối cùng được coi là phân loại softmax:

... CONV => RELU => POOL => FC => DO => FC => DO => FC

Tuy nhiên, cũng có thể áp dụng bỏ học với xác suất nhỏ hơn (ví dụ: $p = 0,10, 0,25$) trong các lớp trước mạng (thông thường sau hoạt động lấy mẫu xuống, thông qua gộp chung hoặc tích chập).

2.1.3. Kiến trúc và mô hình huấn luyện chung

Như đã thấy trong suốt chương này, mạng nơ-ron tích chập được tạo thành từ bốn lớp chính: CONV, POOL, RELU và FC. Lấy các lớp này và xếp chúng lại với nhau trong một mẫu cụ thể mang lại kiến trúc CNN.

Các lớp CONV và FC (và BN) là các lớp duy nhất mạng thực sự tìm hiểu các tham số - các lớp khác chỉ chịu trách nhiệm thực hiện một thao tác nhất định. Các lớp kích hoạt (ACT) như RELU và các lớp kỹ thuật bỏ học, nhưng thường được bao gồm trong sơ đồ kiến trúc CNN để làm cho thứ tự hoạt động rõ ràng - cũng áp dụng quy ước tương tự trong phần này.

2.1.3.1. Các mẫu lớp

Cho đến nay, hình thức phổ biến nhất của kiến trúc CNN là xếp chồng một vài lớp CONV và RELU, theo sau chúng bằng thao tác POOL. Lặp lại trình tự này cho đến khi chiều rộng và chiều cao có số lượng nhỏ, tại đó áp dụng một hoặc nhiều lớp FC. Do đó, có thể rút ra kiến trúc CNN phổ biến nhất bằng cách sử dụng mẫu sau [51]:

INPUT => [[CONV => RELU]*N => POOL?]*M => [FC => RELU]*K => FC

Ở đây toán tử * ngụ ý một hay nhiều và? chỉ ra một hoạt động tùy chọn. Các lựa chọn phổ biến bao gồm:

- $0 \leq N \leq 3$
- $M >= 0$
- $0 \leq K \leq 2$

Dưới đây có thể thấy một số ví dụ về kiến trúc CNN theo mô hình này:

- INPUT => FC
- INPUT => [CONV => RELU => POOL] * 2 => FC => RELU => FC
- INPUT => [CONV => RELU => CONV => RELU => POOL]
* 3 => [FC => RELU] * 2 => FC

Dưới đây là một ví dụ về một CNN rất nồng chỉ có một lớp CONV ($N = M = K = 0$):

INPUT => CONV => RELU => FC

Một ví dụ khác về kiến trúc CNN giống như AlexNet [94] có nhiều bộ CONV => RELU => POOL, theo sau là các lớp FC:

INPUT => [CONV => RELU => POOL] * 2 => [CONV => RELU]
* 3 => POOL => [FC => RELU => DO] * 2 => SOFTMAX

Đối với các kiến trúc mạng sâu hơn, chẳng hạn như VGGNet [95], chúng sẽ xếp chồng hai (hoặc nhiều hơn) trước mỗi lớp POOL:

INPUT => [CONV => RELU] * 2 => POOL => [CONV => RELU]
* 2 => POOL => [CONV => RELU] * 3 => POOL => [CONV => RELU]
* 3 => POOL => [FC => RELU => DO] * 2 => SOFTMAX

Nói chung, áp dụng các kiến trúc mạng sâu hơn khi (1) có nhiều dữ liệu huấn luyện được gắn nhãn và (2) vấn đề phân loại là đủ thách thức. Việc xếp chồng nhiều lớp CONV trước khi áp dụng lớp POOL cho phép các lớp CONV phát triển các tính năng phức tạp hơn trước khi thao tác gộp phá hủy được thực hiện.

Như khám phá trong gói ImageNet, có nhiều kiến trúc mạng kỳ lạ hơn, lệch khỏi các mẫu này và đến lượt nó, đã tạo ra các mẫu riêng họ. Một số kiến trúc loại bỏ hoàn toàn thao tác POOL, dựa vào các lớp CONV để giảm số lượng - sau đó, ở cuối mạng, nhóm trung bình được áp dụng thay vì các lớp FC để thu được đầu vào cho các phân loại softmax.

Các kiến trúc mạng như GoogLeNet, ResNet và SqueezeNet [7, 55, 56] là những ví dụ tuyệt vời về mô hình này và chứng minh cách loại bỏ các lớp FC dẫn đến ít tham số hơn và thời gian huấn luyện nhanh hơn.

Các kiểu kiến trúc mạng này cũng có các bộ lọc xếp chồng và ghép các bộ lọc theo kích thước kênh: GoogLeNet áp dụng các bộ lọc 1×1 , 3×3 và 5×5 và sau đó ghép chúng lại với nhau theo chiều kênh để tìm hiểu các tính năng đa cấp. Những kiến trúc này được coi là kỹ thuật nâng cao.

Nếu quan tâm đến các kiến trúc nâng cao của CNN, vui lòng tham khảo gói ImageNet; mặt khác, chúng ta sẽ gắn bó với các mẫu xếp chồng lớp cơ bản cho đến khi học các nguyên tắc cơ bản học sâu.

2.1.3.2. Định luật ngón tay cái

Trong phần này, xem xét các quy tắc chung khi xây dựng mạng CNN riêng. Để bắt đầu, ảnh được trình bày cho lớp đầu vào phải là hình vuông. Sử dụng đầu vào hình vuông cho phép tận dụng các thư viện tối ưu hóa đại số tuyến tính. Kích thước lớp đầu vào phổ biến bao gồm 32×32 , 64×64 , 96×96 , 224×224 , 227×227 và 229×229 (bỏ qua số lượng kênh).

Thứ hai, lớp đầu vào cũng phải chia hết cho hai lần sau khi thao tác CONV đầu tiên được áp dụng. Có thể làm điều này bằng cách điều chỉnh kích thước bộ lọc và bước trượt. Bộ chia số theo hai quy tắc cho phép các đầu vào không gian trong mạng được lấy mẫu thuận tiện thông qua hoạt động POOL một cách hiệu quả.

Nói chung, các lớp CONV nên sử dụng các kích thước bộ lọc nhỏ hơn như 3×3 và 5×5 . Các bộ lọc 1×1 được sử dụng để học các tính năng cục bộ, nhưng chỉ trong các kiến trúc mạng nâng cao hơn. Kích thước bộ lọc lớn hơn như 7×7 và 11×11 có thể được sử dụng làm lớp CONV đầu tiên trong mạng (để giảm kích thước đầu vào ma trận, miễn là ảnh đủ lớn hơn $> 200 \times 200$ pixel); tuy nhiên, sau lớp CONV ban đầu này, kích thước bộ lọc giảm đáng kể, nếu không giảm kích thước này thì kích thước ma trận sẽ giảm quá nhanh. Cũng thường sử dụng một bước trượt $S = 1$ cho các lớp CONV, ít nhất là cho các khối lượng đầu vào lớn hơn sử dụng bước trượt $S \geq 2$ trong lớp CONV đầu tiên). Sử dụng bước trượt $S = 1$ cho phép các lớp CONV tìm hiểu các bộ lọc trong khi lớp POOL chịu trách nhiệm cho việc lấy mẫu xuống. Tuy nhiên, nhớ rằng không phải tất cả các kiến trúc mạng đều tuân theo mô hình này - một số kiến trúc bỏ qua việc gộp chung tối đa và dựa vào bước trượt CONV để giảm kích thước.

Đề xuất cá nhân thứ hai là sử dụng các lớp POOL (chứ không phải các lớp CONV) làm giảm kích thước không gian đầu vào ít nhất là cho đến khi trở nên có kinh nghiệm hơn khi xây dựng các kiến trúc CNN riêng. Khi đạt đến điểm đó, nên bắt đầu thử nghiệm sử dụng các

lớp CONV để giảm kích thước đầu vào không gian và thử xóa các lớp gộp tối đa khỏi kiến trúc.

Thông thường nhất, thấy nhóm tối đa được áp dụng trên kích thước trường tiếp nhận 2×2 và bước trượt $S = 2$. Cũng có thể thấy trường hợp trường tiếp nhận 3×3 sõm trong kiến trúc mạng để giúp giảm kích thước ảnh. Rất hiếm khi thấy các lĩnh vực tiếp nhận lớn hơn 3 vì các hoạt động này phá hủy các đầu vào của chúng.

Chuẩn hóa hàng loạt là một hoạt động tốn kém có thể làm tăng gấp đôi hoặc gấp ba thời gian cần thiết để huấn luyện CNN; tuy nhiên, nên sử dụng BN trong hầu hết các tình huống. Mặc dù BN thực sự làm chậm thời gian huấn luyện, nhưng nó cũng có xu hướng ổn định việc huấn luyện, giúp điều chỉnh các tham số khác dễ dàng hơn.

Cũng có trường hợp đặt chuẩn hóa hàng loạt sau khi kích hoạt, việc này đã trở nên phổ biến trong cộng đồng học sâu mặc dù nó đi ngược lại với bài báo gốc của Ioffe và Szegedy [53].

Chèn lớp BN vào các kiến trúc lớp phổ biến ở trên, chúng trở thành:

- INPUT => CONV => RELU => BN => FC
- INPUT => [CONV => RELU => BN => POOL] * 2 => FC => RELU => BN => FC
- INPUT => [CONV => RELU => BN => CONV => RELU => BN => POOL] * 3 => [FC => RELU => BN] * 2 => FC

Không áp dụng chuẩn hóa hàng loạt trước khi phân loại softmax vì tại thời điểm này mạng đã học được các tính năng phân loại trước đó trong kiến trúc.

Bỏ học (DO) thường được áp dụng ở giữa các lớp FC với xác suất bỏ học là 50% - nên xem xét áp dụng bỏ học trong hầu hết mọi kiến trúc xây dựng. Mặc dù không phải lúc nào cũng được thực hiện, nhưng mong muốn có các lớp bỏ học (với xác suất rất nhỏ, 10-25%) giữa các lớp POOL và CONV. Do khả năng kết nối cục bộ các lớp CONV, việc bỏ học ở đây kém hiệu quả hơn, nhưng nó thường hữu ích khi ngăn chặn hiện tượng quá khớp.

Bằng cách ghi nhớ các quy tắc này, có thể giảm bớt khó khăn khi xây dựng kiến trúc CNN vì các lớp CONV duy trì kích thước đầu vào trong khi các lớp POOL đảm nhiệm việc giảm kích thước không gian các khối, cuối cùng dẫn đến các lớp FC và các phân loại đầu ra cuối cùng.

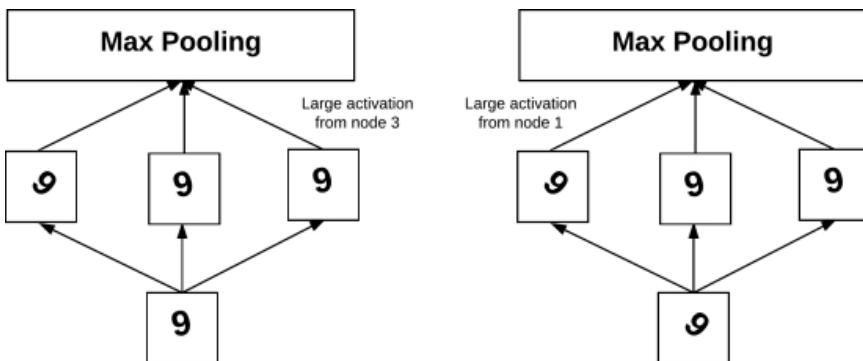
Khi thành thạo phương pháp xây dựng mạng truyền thống trực tuyến này, nên bắt đầu khám phá để loại bỏ hoàn toàn các hoạt động gộp chung và chỉ sử dụng các lớp CONV để giảm kích thước không gian, cuối cùng dẫn đến gộp chung thay vì một lớp FC.

2.1.4. Các CNN có bất biến đối với phép dịch, xoay và thu nhỏ không?

Một câu hỏi phổ biến nhận được hỏi là:

Các mạng nơ-ron tích chập có phải là bất biến đối với những thay đổi trong dịch thuật, xoay và nhân rộng? Có phải đó là lý do mà CNN là loại mạng nơ-ron phân loại ảnh mạnh mẽ như vậy?

Để trả lời câu hỏi này, trước tiên cần phân biệt giữa các bộ lọc riêng lẻ trong mạng cùng với mạng được huấn luyện cuối cùng. Các bộ lọc riêng lẻ trong CNN không phải là bất biến đối với các thay đổi về cách xoay ảnh.



Hình 2.12: Toàn bộ CNN học các bộ lọc sẽ kích hoạt khi một mẫu được trình bày theo một hướng cụ thể. Ở bên trái, chữ số 9 đã được xoay 10° . Phép quay này tương tự như nút ba đã học được chữ số 9 trông như thế nào khi xoay theo cách này. Nút này có kích hoạt cao hơn hai nút còn lại - hoạt động gộp tối đa phát hiện ra điều này. Ở bên phải có một ví dụ thứ hai, chỉ lần này số 9 đã được xoay 45° , khiến nút đầu tiên có kích hoạt cao nhất (Nguồn tham khảo: Goodfellow và cộng sự [10]).

Tuy nhiên, toàn bộ CNN có thể tìm hiểu các bộ lọc kích hoạt khi một mẫu được trình bày theo một hướng cụ thể. Ví dụ, xem xét Hình 2.12, được điều chỉnh và lấy ý tưởng từ Deep Learning Goodfellow et al. [33].

Ở đây thấy chữ số 9 (phía dưới) được trình bày cho CNN cùng với một bộ các bộ lọc mà CNN đã học (ở giữa). Vì có một bộ lọc bên trong CNN có tính năng học hỏi, một số 9 trông như thế nào, khi xoay 10° , nó kích hoạt và phát ra một kích hoạt mạnh. Kích hoạt lớn này được ghi lại trong giai đoạn gộp và cuối cùng được báo cáo là phân loại cuối cùng.

Điều tương tự cũng đúng với ví dụ thứ hai (Hình 2.12, bên trái). Ở đây, thấy một số 9 quay được 45 độ và vì có một bộ lọc trong CNN đã học được một số 9 trông như thế nào khi nó bị xoay 45 độ. Một lần nữa, bản thân các bộ lọc này không phải là bất biến xoay vòng - chỉ là CNN đã học được một số 9 trông như thế nào dưới các góc quay nhỏ tồn tại trong tập huấn luyện.

Trừ khi dữ liệu huấn luyện bao gồm các chữ số được xoay trên toàn phẳng 360 độ, CNN không thực sự là bất biến xoay vòng

Điều tương tự cũng có thể nói về tỷ lệ - bản thân các bộ lọc không phải là bất biến tỷ lệ, nhưng rất có khả năng CNN đã học được một bộ các bộ lọc kích hoạt khi các mẫu tồn tại ở các tỷ lệ khác nhau. Cũng có thể giúp CNN trở thành bất biến tỷ lệ bằng cách đưa ra ảnh ví dụ cho chúng tại thời điểm thử nghiệm theo các quy mô khác nhau, sau đó lấy trung bình các kết quả.

Bất biến là sự vượt trội của CNN, nhớ rằng bộ lọc trượt từ trái sang phải và từ trên xuống dưới qua đầu vào và kích hoạt khi đi qua một vùng, góc hoặc đốm màu cụ thể. Trong quá trình hoạt động của lớp gộp, phản hồi lớn này được tìm thấy và do đó, nhóm tất cả các vùng lân cận bằng cách kích hoạt lớn hơn. Do đó, các CNN có thể được xem như không quan tâm đến chính xác nơi một kích hoạt hoạt động, chỉ đơn giản là nó kích hoạt - và theo cách này, tự nhiên xử lý sự dịch chuyển bên trong một CNN.

2.1.5. Tóm tắt

Trong phần này đã tìm hiểu về mạng nơ-ron tích chập (CNN). Bắt đầu bằng cách thảo luận về tích chập và tương quan chéo là gì và làm thế nào các thuật ngữ được sử dụng thay thế cho nhau trong tài liệu học sâu.

Để hiểu tích chập ở mức độ cao hơn, đã triển khai nó thủ công bằng Python và OpenCV. Tuy nhiên, các hoạt động xử lý ảnh truyền thống yêu cầu xác định thủ công các mặt nạ đặc trưng và dành riêng cho một tác vụ xử lý ảnh nhất định (ví dụ: làm mịn, phát hiện cạnh, v.v.). Thay vào đó, sử dụng học sâu, có thể tìm hiểu các loại bộ lọc này được xếp chồng lên nhau để tự động khám phá các khái niệm cấp cao, gọi việc xếp chồng và tìm hiểu các tính năng cấp cao hơn dựa trên các đầu vào cấp thấp hơn là thành phần của mạng nơ-ron tích chập.

Các CNN được xây dựng bằng cách xếp chồng một chuỗi các lớp, trong đó mỗi lớp chịu trách nhiệm cho một nhiệm vụ nhất định. Các lớp CONV tìm hiểu một bộ các bộ lọc tích chập K, mỗi bộ lọc có kích thước

pixel $F \times F$. Sau đó, áp dụng các lớp kích hoạt trên đầu các lớp CONV để có được phép biến đổi phi tuyến. Các lớp POOL giúp giảm kích thước không gian số lượng đầu vào khi nó được đưa qua mạng.

Các lớp chuẩn hóa hàng loạt được sử dụng để chuẩn hóa các đầu vào thành CONV hoặc lớp kích hoạt bằng cách tính toán độ lệch trung bình và độ lệch chuẩn trong một đợt nhỏ. Sau đó, một lớp bỏ học có thể được áp dụng để ngắt kết nối các nút ngẫu nhiên từ đầu vào đã cho sang đầu ra, giúp giảm quá khớp.

Cuối cùng, kết thúc mục này bằng cách xem xét các kiến trúc CNN phổ biến có thể sử dụng để triển khai các mạng của riêng mình. Trong mục tiếp theo triển khai CNN đầu tiên trong Keras, ShallowNet, dựa trên các mẫu lớp đã đề cập ở trên. Các chương trong tương lai thảo luận về các kiến trúc mạng sâu hơn như kiến trúc LeNet bán nguyệt [32] và các biến thể kiến trúc VGGNet [57].

2.2. HUẤN LUYỆN MẠNG CNN ĐẦU TIÊN

Chúng ta đã xem xét các nguyên tắc cơ bản của mạng nơ-ron tích chập và sẵn sàng triển khai CNN đầu tiên bằng Python và Keras. Sẽ bắt đầu mục này bằng cách xem xét nhanh các cấu hình Keras mà nên ghi nhớ khi xây dựng và huấn luyện CNN

Sau đó, sẽ triển khai ShallowNet, đúng như tên gọi, là một CNN rất đơn giản với chỉ một lớp CONV. Tuy nhiên, đúng để sự đơn giản của mạng này đánh lừa - vì kết quả sẽ chứng minh, ShallowNet có khả năng đạt được độ chính xác phân loại cao hơn trên cả CIFAR-10 và bộ dữ liệu động vật so với bất kỳ phương pháp nào khác trong cuốn sách này.

2.2.1. Cấu hình thư viện KERAS và chuyển đổi ảnh sang dạng mảng

Trước khi có thể triển khai ShallowNet, trước tiên cần xem lại tệp cấu hình keras.json và cách cài đặt bên trong tệp này sẽ ảnh hưởng đến cách triển khai CNN. Ngoài ra, cũng sẽ triển khai một bộ xử lý ảnh thứ hai có tên ImageToArrayPreprocessor chấp nhận một ảnh đầu vào và sau đó chuyển đổi nó thành một mảng NumPy mà Keras có thể làm việc.

2.2.1.1. Tìm hiểu tệp cấu hình Keras.Json

Lần đầu tiên nhập thư viện Keras vào môi trường Python / thực thi tập lệnh Python nhập Keras, đồng sau Keras tạo tệp keras.json trong thư mục chính, có thể tìm thấy tệp cấu hình này trong /.keras/ keras.json. Tiếp tục mở tệp lên và xem nội dung của nó:

```
1 {  
2     "epsilon": 1e-07,  
3     "floatx": "float32",  
4     "image_data_format": "channels_last",  
5     "backend": "tensorflow"  
6 }
```

Có thể nhận thấy rằng từ điển được chương trình hóa JSON này có bốn chìa khóa và bốn giá trị tương ứng. Giá trị epsilon được sử dụng ở nhiều vị trí khác nhau trong thư viện Keras để ngăn phân chia cho các lỗi là không. Giá trị mặc định 1e-07 là phù hợp và không nên thay đổi. Sau đó có giá trị floatx xác định độ chính xác dấu phẩy động – chắc chắn khi để giá trị này ở float32.

Hai cấu hình cuối cùng, image_data_format và phụ trợ, là cực kỳ quan trọng. Theo mặc định, thư viện Keras sử dụng phụ trợ tính toán số TensorFlow, cũng có thể sử dụng hỗ trợ Theano đơn giản bằng cách thay thế tensorflow bằng theano.

Ghi nhớ những phụ trợ này khi phát triển mạng học sâu và khi triển khai chúng cho các máy khác. Keras thực hiện một công việc tuyệt vời để trừu tượng hóa phần phụ trợ, cho phép viết chương trình học sâu tương thích với phần phụ trợ (chắc chắn sẽ có nhiều phần phụ trợ hơn trong tương lai), và phần lớn, sẽ thấy rằng cả phần phụ trợ tính toán sẽ cung cấp kết quả tương tự. Nếu thấy kết quả của mình không nhất quán hoặc chương trình và đang trả về các lỗi lạ, trước tiên kiểm tra phần phụ trợ và đảm bảo rằng đã cài đặt đúng.

Cuối cùng, image_data_format có thể chấp nhận hai giá trị: channel_last hoặc channel_first. Như đã biết từ các chương trước trong cuốn sách này, ảnh được tải qua OpenCV được thể hiện theo thứ tự (hàng, cột, kênh), đây là thứ mà Keras gọi là channel_last, vì các kênh là thứ nguyên cuối cùng trong mảng.

Ngoài ra, có thể đặt image_data_format thành channel_first nơi ảnh đầu vào được thể hiện dưới dạng (kênh, hàng, cột) - chú ý số lượng kênh là thứ nguyên đầu tiên trong mảng.

Tại sao lại có 2 cài đặt? Trong cộng đồng Theano, người dùng có xu hướng sử dụng các kênh đặt hàng đầu tiên. Tuy nhiên, khi TensorFlow được phát hành, các hướng dẫn và ví dụ của họ đã sử dụng các kênh đặt hàng lần cuối. Sự khác biệt này đã gây ra một chút vấn đề khi sử dụng Keras làm chương trình tương thích với Theano vì nó có thể không tương

thích với TensorFlow tùy thuộc vào cách lập trình viên xây dựng mạng của họ. Do đó, Keras đã giới thiệu một hàm đặc biệt gọi là `img_to_array` chấp nhận ảnh đầu vào và sau đó đặt hàng các kênh chính xác dựa trên cài đặt `image_data_format`.

Nói chung, có thể cài đặt `image_data_format` vì `channel_last` và Keras sẽ đảm nhiệm việc sắp xếp thứ nguyên cho bất kể phụ trợ nào; tuy nhiên, tình huống này chỉ được gọi đến trong trường hợp đang làm việc với chương trình Keras kế thừa và nhận thấy rằng một thứ tự kênh ảnh khác được sử dụng.

2.2.1.2. Bộ xử lý ảnh thành dạng mảng

Như đã đề cập ở trên, thư viện Keras hỗ trợ hàm `img_to_array` chấp nhận ảnh đầu vào và sau đó sắp xếp chính xác các kênh dựa trên cài đặt `image_data_format`, sẽ bọc hàm này bên trong một lớp mới có tên `ImageToArrayPreprocessor`. Tạo một lớp với hàm tiền xử lý đặc biệt để thay đổi kích thước ảnh, sẽ cho phép tạo ra các chuỗi bộ xử lý trước để chuẩn bị ảnh để huấn luyện và kiểm tra một cách hiệu quả.

Để tạo bộ tiền xử lý ảnh thành mảng, tạo một tệp mới có tên là `imagetoarraypreprocessor.py` bên trong mô đun con tiền xử lý `pyimagesearch`:

```
|--- pyimagesearch
|   |--- __init__.py
|   |--- datasets
|   |   |--- __init__.py
|   |   |--- simpledatasetloader.py
|   |--- preprocessing
|   |   |--- __init__.py
|   |   |--- imagetoarraypreprocessor.py
|   |   |--- simplepreprocessor.py
```

Từ đó, mở tệp và chèn đoạn mã sau:

```
1 # import the necessary packages
2 from keras.preprocessing.image import img_to_array
3
4 class ImageToArrayPreprocessor:
5     def __init__(self, dataFormat=None):
6         # store the image data format
7         self.dataFormat = dataFormat
8
9     def preprocess(self, image):
10        # apply the Keras utility function that correctly rearranges
11        # the dimensions of the image
12        return img_to_array(image, data_format=self.dataFormat)
```

Dòng 2 nhập hàm `img_to_array` từ Keras.

Sau đó, định nghĩa hàm tạo cho lớp ImageToArrayPreprocessor trên dòng 5-7. Hàm tạo chấp nhận một tham số tùy chọn có tên dataFormat. Giá trị này mặc định là 0, điều này cho biết rằng nên sử dụng cài đặt bên trong keras.json. cũng có thể cung cấp rõ ràng một chuỗi channel_first hoặc channel_last, nhưng tốt nhất là để Keras chọn thứ tự kích thước ảnh được sử dụng dựa trên tệp cấu hình.

Cuối cùng, có hàm tiền xử lý trên dòng 9-12. Phương pháp này:

1. Chấp nhận một ảnh làm đầu vào.
2. Gọi img_to_array trên ảnh, sắp xếp các kênh dựa trên tệp cấu hình/giá trị dataFormat.
3. Trả về một mảng NumPy mới với các kênh được sắp xếp đúng.

Lợi ích của việc xác định một lớp để xử lý loại tiền xử lý ảnh này thay vì chỉ gọi img_to_array trên mỗi ảnh là giờ đây có thể xâu chuỗi các bộ tiền xử lý lại với nhau khi tải bộ dữ liệu từ đĩa.

Ví dụ: Giả sử muốn thay đổi kích thước tất cả các ảnh đầu vào thành kích thước cố định 32×32 pixel.

Để thực hiện điều này, cần khởi tạo SimpleProcessor:

```
1 sp = SimplePreprocessor(32, 32)
```

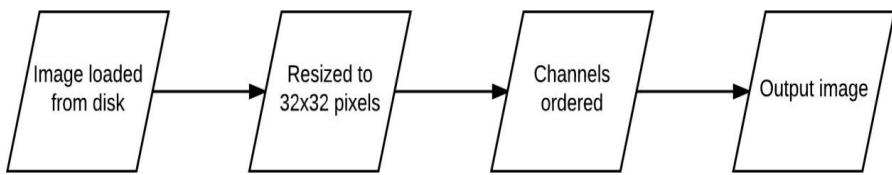
Sau khi ảnh được thay đổi kích thước, sau đó cần áp dụng thứ tự kênh chính xác - điều này có thể được thực hiện bằng ImageToArrayPreprocessor ở trên:

```
2 iap = ImageToArrayPreprocessor()
```

Bây giờ, giả sử muốn tải một tập dữ liệu ảnh từ đĩa và chuẩn bị tất cả các ảnh trong bộ dữ liệu để huấn luyện. Sử dụng SimpleDatasetLoader, nhiệm vụ trở nên rất dễ dàng:

```
3 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
4 (data, labels) = sdl.load(imagePaths, verbose=500)
```

Lưu ý cách các bộ tiền xử lý ảnh được kết nối với nhau và được áp dụng theo thứ tự liên tiếp. Đối với mỗi ảnh trong tập dữ liệu, trước tiên áp dụng SimplePreprocessor để thay đổi kích thước thành 32×32 pixel. Khi ảnh được thay đổi kích thước, ImageToArrayPreprocessor được áp dụng để xử lý việc sắp xếp các kênh ảnh. Đường dẫn xử lý ảnh này có thể được hiển thị trong Hình 2.13.



Hình 2.13: Một đường dẫn tiền xử lý ví dụ (1) tải ảnh từ đĩa, (2) thay đổi kích thước thành 32×32 pixel, (3) sắp xếp kích thước kênh và (4) xuất ảnh.

Xâu chuỗi các bộ tiền xử lý đơn giản lại với nhau theo phương pháp này, trong đó mỗi bộ tiền xử lý chịu trách nhiệm cho một công việc nhỏ, là một cách dễ dàng để xây dựng một thư viện học sâu có thể mở rộng dành riêng cho việc phân loại ảnh.

2.2.2. ShallowNet

Trong phần này, sẽ triển khai kiến trúc ShallowNet. Như tên cho thấy, kiến trúc ShallowNet chỉ chứa một vài lớp - toàn bộ kiến trúc mạng có thể được tóm tắt là: INPUT => CONV => RELU => FC.

Kiến trúc mạng đơn giản này sẽ cho phép thực hiện mạng nơ-ron tích chập bằng cách sử dụng thư viện Keras. Sau khi triển khai ShallowNet, sẽ áp dụng nó cho bộ dữ liệu động vật và CIFAR-10. Vì kết quả sẽ chứng minh, các CNN có thể vượt trội đáng kể so với các phương pháp phân loại ảnh trước đó được thảo luận trong cuốn sách này.

2.2.2.1. Thực hiện ShallowNet

Để giữ cho gói pyimagesearch gọn gàng, tạo một mô-đun con mới bên trong NN có tên conv, nơi tất cả các triển khai CNN sẽ hoạt động:

```

--- pyimagesearch
|   |--- __init__.py
|   |--- datasets
|   |--- nn
|   |   |--- __init__.py
...
|   |   |--- conv
|   |   |   |--- __init__.py
|   |   |   |--- shallownet.py
|   |--- preprocessing

```

Bên trong mô-đun con, tạo một tệp mới có tên willownet.py để lưu trữ triển khai kiến trúc ShallowNet. Từ đó, mở tệp và chèn đoạn chương trình sau:

```

1 # import the necessary packages
2 from keras.models import Sequential
3 from keras.layers.convolutional import Conv2D

```

```
4 from keras.layers.core import Activation  
5 from keras.layers.core import Flatten  
6 from keras.layers.core import Dense  
7 from keras import backend as K
```

Dòng 2-7 nhập các gói Python cần thiết. Lớp Conv2D là lớp triển khai Keras lớp tích chập. Sau đó, có lớp kích hoạt xử lý việc áp dụng hàm tác động cho đầu vào. Các lớp Flatten đưa khỏi lượng đa chiều và các flattens nó thành một mảng 1D trước khi đưa các đầu vào vào các lớp Dense (tức là, được kết nối đầy đủ).

Khi triển khai các kiến trúc mạng, thường xác định chúng trong một lớp để giữ cho chương trình có tổ chức - sẽ làm tương tự ở đây:

```
9 class ShallowNet:  
10     @staticmethod  
11     def build(width, height, depth, classes):  
12         # initialize the model along with the input shape to be  
13         # "channels last"  
14         model = Sequential()  
15         inputShape = (height, width, depth)  
16  
17         # if we are using "channels first", update the input shape  
18         if K.image_data_format() == "channels_first":  
19             inputShape = (depth, height, width)
```

Trên Dòng 9, định nghĩa lớp ShallowNet và sau đó xác định phương thức xây dựng trên Dòng 11. Mỗi CNN sẽ có phương thức xây dựng riêng - hàm này sẽ chấp nhận một số tham số, xây dựng kiến trúc mạng và sau đó trả về đến hàm gọi. Trong trường hợp này, phương thức xây dựng yêu cầu bốn tham số:

- width: chiều rộng các ảnh đầu vào sẽ được sử dụng để huấn luyện mạng (tức là số lượng cột trong ma trận).
- height: chiều cao ảnh đầu vào (nghĩa là, số lượng hàng trong ma trận).
- độ sâu: số lượng kênh trong ảnh đầu vào.
- các lớp: tổng số lớp mà mạng nên học để dự đoán. Đối với động vật, class = 3 và đối với CIFAR-10, class = 10.

Sau đó, sẽ khởi tạo inputShape cho mạng trên Dòng 15 với giả định các kênh Order đặt hàng trước. Dòng 18 và 19 thực hiện một kiểm tra để xem liệu phụ trợ Keras có được đặt thành các kênh đầu tiên hay không và nếu có, sẽ cập nhật inputShape. Dòng 15-19 áp dụng phổ biến cho hầu hết CNN xây dựng, do đó đảm bảo rằng mạng sẽ hoạt động bất kể số lượng kênh ảnh của người dùng.

Bây giờ, inputShape đã được xác định, có thể bắt đầu xây dựng kiến trúc ShallowNet:

```
21      # define the first (and only) CONV => RELU layer
22      model.add(Conv2D(32, (3, 3), padding="same",
23                      input_shape=inputShape))
24      model.add(Activation("relu"))
```

Trên Dòng 22, xác định lớp tích chập đầu tiên (và duy nhất). Lớp này sẽ có 32 bộ lọc (K) mỗi bộ lọc có kích thước 3×3 (tức là bộ lọc $F \times F$ vuông). Sẽ áp dụng cùng một phần đệm để đảm bảo kích thước đầu ra hoạt động tích chập khớp với đầu vào (sử dụng cùng một phần đệm là không cần thiết cho ví dụ này, nhưng đó là một thói quen tốt có thể bắt đầu hình thành ngay bây giờ). Sau khi tích chập, áp dụng kích hoạt ReLU trên Dòng 24.

Kết thúc việc xây dựng ShallowNet:

```
26      # softmax classifier
27      model.add(Flatten())
28      model.add(Dense(classes))
29      model.add(Activation("softmax"))
30
31      # return the constructed network architecture
32      return model
```

Để áp dụng lớp được kết nối đầy đủ, trước tiên cần làm phẳng ma trận đa chiều vào mảng 1 chiều. Hoạt động làm phẳng được xử lý bởi lệnh gọi Flatten trên Dòng 27. Sau đó, một lớp mật độ được tạo bằng cách sử dụng cùng số nút như nhãn lớp đầu ra (Dòng 28). Dòng 29 áp dụng hàm tác động softmax sẽ cung cấp cho xác suất lớp nhãn cho mỗi lớp. Kiến trúc ShallowNet được trả về hàm gọi trên Dòng 32.

Bây giờ ShallowNet đã được xác định, được sử dụng để tải một tập dữ liệu, tiền xử lý và sau đó huấn luyện mạng. Xem xét hai ví dụ tận dụng ShallowNet - động vật và CIFAR-10.

2.2.2.2. Shallownet trên tập dữ liệu động vật

Để huấn luyện ShallowNet trên bộ dữ liệu động vật, cần tạo một tệp Python riêng. Mở IDE yêu thích, tạo một tệp mới có tên willownet_animals.py, đảm bảo rằng nó ở cùng cấp thư mục với mô-đun pyimagesearch.

Từ đó, có thể lập trình:

```
1  # import the necessary packages
2  from sklearn.preprocessing import LabelBinarizer
3  from sklearn.model_selection import train_test_split
4  from sklearn.metrics import classification_report
5  from pyimagesearch.preprocessing import ImageToArrayPreprocessor
6  from pyimagesearch.preprocessing import SimplePreprocessor
7  from pyimagesearch.datasets import SimpleDatasetLoader
8  from pyimagesearch.nn.conv import ShallowNet
9  from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Dòng 2-13 nhập các gói Python cần thiết. Hầu hết các thư viện cần nhập đã thấy từ các ví dụ trước, nhưng muôn gọi sự chú ý đến Dòng 5-7 nơi nhập ImageToArrayPreprocessor, SimplePreprocessor và SimpleDatasetLoader - các lớp này sẽ tạo thành đường dẫn thực tế được sử dụng để xử lý ảnh trước khi chuyển chúng thông qua mạng. Sau đó, nhập ShallowNet trên Dòng 8 cùng với SGD trên Dòng 9 - sẽ sử dụng Stochastic Gradient Descent để huấn luyện ShallowNet.

Tiếp theo, cần phân tích các đối số dòng lệnh và lấy các đường dẫn ảnh. Chương trình chỉ yêu cầu một chuyên đổi duy nhất ở đây, --dataset, đó là đường dẫn đến thư mục chứa tập dữ liệu Động vật. Dòng 23 sau đó lấy đường dẫn tệp đến tất cả 3.000 ảnh bên trong bộ dữ liệu Động vật.

```
15 # construct the argument parser and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18     help="path to input dataset")
19 args = vars(ap.parse_args())
20
21 # grab the list of images that we'll be describing
22 print("[INFO] loading images...")
23 imagePaths = list(paths.list_images(args["dataset"]))
```

Đây là cách mà chúng ta sẽ tải và xử lý dữ liệu:

```
25 # initialize the image preprocessors
26 sp = SimplePreprocessor(32, 32)
27 iap = ImageToArrayPreprocessor()
28
29 # load the dataset from disk then scale the raw pixel intensities
30 # to the range [0, 1]
31 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
32 (data, labels) = sdl.load(imagePaths, verbose=500)
33 data = data.astype("float") / 255.0
```

Dòng 26 định nghĩa SimpleProcessor được sử dụng để thay đổi kích thước ảnh đầu vào thành 32×32 pixel. Các ImageToArrayPreprocessor sau đó được khởi tạo trên Dòng 27 để xử lý thứ tự kênh. Kết hợp các bộ tiền xử lý này với nhau trên Dòng 31 nơi khởi tạo SimpleDatasetLoader.

Xem tham số tiền xử lý hàm tạo - đang cung cấp một danh sách các bộ xử lý trước sẽ được áp dụng theo thứ tự tuần tự. Đầu tiên, một ảnh đầu vào nhất định sẽ được thay đổi kích thước thành 32×32 pixel. Sau đó, ảnh đã thay đổi kích thước sẽ được sắp xếp các kênh theo tệp cấu hình keras.json. Dòng 32 tải ảnh (áp dụng bộ tiền xử lý) và nhãn lớp. Sau đó chia tỷ lệ ảnh thành phạm vi $[0, 1]$.

Bây giờ dữ liệu và nhãn đã được tải, có thể thực hiện phân tách

thử nghiệm và huấn luyện của mình, cùng với chương trình hóa nhãn:

```
35 # partition the data into training and testing splits using 75% of
36 # the data for training and the remaining 25% for testing
37 (trainX, testX, trainY, testY) = train_test_split(data, labels,
38         test_size=0.25, random_state=42)
39
40 # convert the labels from integers to vectors
41 trainY = LabelBinarizer().fit_transform(trainY)
42 testY = LabelBinarizer().fit_transform(testY)
```

Ở đây đang sử dụng 75% dữ liệu để huấn luyện và 25% cho thử nghiệm.

Bước tiếp theo là khởi tạo ShallowNet, tiếp theo là tự huấn luyện mạng:

```
44 # initialize the optimizer and model
45 print("[INFO] compiling model...")
46 opt = SGD(lr=0.005)
47 model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
48 model.compile(loss="categorical_crossentropy", optimizer=opt,
49     metrics=["accuracy"])
50
51 # train the network
52 print("[INFO] training network...")
53 H = model.fit(trainX, trainY, validation_data=(testX, testY),
54     batch_size=32, epochs=100, verbose=1)
```

Khởi tạo hàm tối ưu hóa SGD trên dòng 46 bằng tỷ lệ học là 0,005 (sẽ thảo luận về cách điều chỉnh tỷ lệ học ở phần sau). Kiến trúc ShallowNet được khởi tạo trên Dòng 47, cung cấp chiều rộng và chiều cao 32 pixel cùng với độ sâu 3 - điều này ngũ ý rằng ảnh đầu vào là 32×32 pixel với ba kênh. Vì bộ dữ liệu động vật có ba nhãn lớp, đặt các lớp = 3.

Mô hình này sau đó được biên dịch trên dòng 48 và 49, trong đó sẽ sử dụng entropy chéo làm hàm tổn thất và SGD làm trình tối ưu hóa. Để huấn luyện thực tế mạng, thực hiện cuộc gọi đến phương thức mô hình .fit trên Dòng 53 và 54. Phương thức .fit yêu cầu vượt qua trong dữ liệu huấn luyện và thử nghiệm, cũng sẽ cung cấp dữ liệu thử nghiệm để có thể đánh giá kết quả ShallowNet sau mỗi chu kỳ. Mạng sẽ được huấn luyện trong 100 chu kỳ sử dụng kích thước đợt nhỏ là 32 (có nghĩa là 32 ảnh sẽ được hiển thị cho mạng cùng một lúc, và việc chuyển hoàn toàn về phía trước và phía sau sẽ được thực hiện để cập nhật các tham số mạng).

Sau khi huấn luyện mạng, có thể đánh giá kết quả:

```
56 # evaluate the network
57 print("[INFO] evaluating network...")
58 predictions = model.predict(testX, batch_size=32)
59 print(classification_report(testY.argmax(axis=1),
60     predictions.argmax(axis=1),
61     target_names=["cat", "dog", "panda"]))
```

Để có được các dự đoán đầu ra trên dữ liệu thử nghiệm, gọi `.predict` mô hình. Một kết quả phân loại được định dạng được hiển thị trên màn hình ở dòng 59-61.

Khởi chương trình cuối cùng hiển thị đồ thị về độ chính xác và tổn thất theo thời gian cho cả dữ liệu huấn luyện và thử nghiệm:

```
63 # plot the training loss and accuracy
64 plt.style.use("ggplot")
65 plt.figure()
66 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
67 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
68 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
69 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
70 plt.title("Training Loss and Accuracy")
71 plt.xlabel("Epoch #")
72 plt.ylabel("Loss/Accuracy")
73 plt.legend()
74 plt.show()
```

Để huấn luyện ShallowNet trên bộ dữ liệu Động vật, chỉ cần thực hiện lệnh sau:

```
$ python willownet_animals.py --dataset..../datasets/animals
```

Huấn luyện nên thực hiện nhanh vì mạng rất nồng và bộ dữ liệu ảnh tương đối nhỏ:

```
[INFO] loading images...
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] compiling model...
[INFO] training network...
Train on 2250 samples, validate on 750 samples
Epoch 1/100
0s - loss: 1.0290 - acc: 0.4560 - val_loss: 0.9602 - val_acc: 0.5160
Epoch 2/100
0s - loss: 0.9289 - acc: 0.5431 - val_loss: 1.0345 - val_acc: 0.4933
...
Epoch 100/100
0s - loss: 0.3442 - acc: 0.8707 - val_loss: 0.6890 - val_acc: 0.6947
[INFO] evaluating network...
      precision    recall   f1-score   support
      cat        0.58      0.77      0.67      239
      dog        0.75      0.40      0.52      249
      panda      0.79      0.90      0.84      262
avg / total        0.71      0.69      0.68      750
```

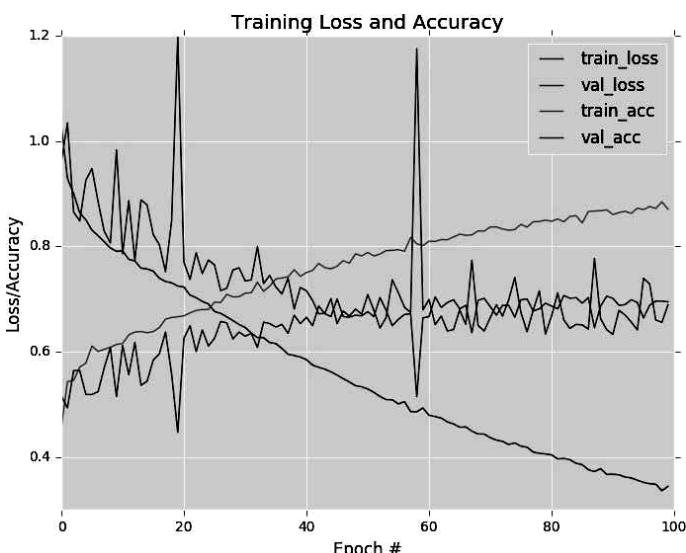
Do số lượng dữ liệu huấn luyện ít, các chu kỳ khá nhanh, chỉ mất chưa đến một giây trên cả CPU và GPU.

Như có thể thấy từ đầu ra ở trên, ShallowNet đã đạt được độ chính xác phân loại 71% trên dữ liệu thử nghiệm, đây là một sự cải tiến lớn so 59% tốt nhất trước đây bằng cách sử dụng các mạng nơ-ron đơn giản. Sử dụng các mạng huấn luyện tiên tiến hơn, cũng như kiến trúc mạnh hơn, sẽ có thể tăng độ chính xác phân loại cao hơn nữa.

Tổn thất và độ chính xác được vẽ theo thời gian được hiển thị trong Hình 2.14. Trên trục x có số chu kỳ và trên trục y có sự tổn thất và độ chính xác. Xem xét con số này, có thể thấy rằng việc học hơi biến động với những đột biến lớn trong khoảng 20 và chu kỳ 60 - kết quả này có thể là do tốc độ học quá cao, mục 3.3 sẽ giải quyết vấn đề này.

Ngoài ra, lưu ý rằng tổn thất huấn luyện và kiểm tra phân kỳ rất lớn trong chu kỳ 30, ngụ ý rằng mạng đang mô hình hóa dữ liệu huấn luyện quá chặt chẽ và quá khớp, có thể khắc phục vấn đề này bằng cách lấy thêm dữ liệu hoặc áp dụng các kỹ thuật như tăng dữ liệu.

Khoảng chu kỳ 60 độ chính xác kiểm tra bão hòa - không thể vượt qua độ chính xác phân loại 70%, trong khi đó độ chính xác huấn luyện tiếp tục tăng lên hơn 85%. Một lần nữa, thu thập thêm dữ liệu huấn luyện, áp dụng tăng dữ liệu và cẩn thận hơn khi điều chỉnh tỷ lệ học sẽ giúp cải thiện kết quả trong tương lai.



Hình 2.14: Đồ thị về sự hàm tổn thất và độ chính xác trong quá trình 100 chu kỳ cho kiến trúc ShallowNet được huấn luyện trên bộ dữ liệu động vật.

Điểm mấu chốt ở đây là mạng nơ-ron tích chập cực kỳ đơn giản có thể đạt được độ chính xác phân loại 71% trên bộ dữ liệu Động vật, trong khi đó mức tốt nhất trước đây chỉ là 59% - cải thiện hơn 12%.

2.2.2.3. ShallowNet trên tập dữ liệu CIFAR-10

Cũng áp dụng kiến trúc ShallowNet cho bộ dữ liệu CIFAR-10 để xem liệu có thể cải thiện kết quả của mình hay không. Mở một tệp mới, đặt tên là willownet_cifar10.py và chèn chương trình sau đây:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from sklearn.metrics import classification_report
4 from pyimagesearch.nn.conv import ShallowNet
5 from keras.optimizers import SGD
6 from keras.datasets import cifar10
7 import matplotlib.pyplot as plt
8 import numpy as np
9
10 # load the training and testing data, then scale it into the
11 # range [0, 1]
12 print("[INFO] loading CIFAR-10 data...")
13 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
14 trainX = trainX.astype("float") / 255.0
15 testX = testX.astype("float") / 255.0
16
17 # convert the labels from integers to vectors
18 lb = LabelBinarizer()
19
20 trainY = lb.fit_transform(trainY)
21 testY = lb.transform(testY)
22
23 # initialize the label names for the CIFAR-10 dataset
24 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"]
```

Dòng 2-8 nhập các gói thư viện Python cần thiết. Sau đó, tải bộ dữ liệu CIFAR-10 (được chia thành các tập huấn luyện và thử nghiệm), sau đó chia tỷ lệ cường độ pixel ảnh thành phạm vi [0, 1]. Vì ảnh CIFAR-10 được xử lý trước và việc đặt hàng kênh được xử lý tự động bên trong cifar10.load_data, không cần phải áp dụng bất kỳ lớp tiền xử lý tùy chỉnh nào.

Các nhãn sau đó được chương trình hóa một lần thành các vectơ trên dòng 18-20. cũng khởi tạo tên nhãn cho bộ dữ liệu CIFAR-10 trên dòng 23 và 24.

Bây giờ dữ liệu đã được chuẩn bị, có thể huấn luyện ShallowNet. Dòng 28 khởi tạo trình tối ưu hóa SGD với tỷ lệ học là 0,01. ShallowNet sau đó được xây dựng trên Dòng 29 sử dụng chiều rộng 32, chiều cao 32, độ sâu 3 (vì ảnh CIFAR-10 có 3 kênh). Đặt các lớp = 10 vì, như tên cho thấy, có mười lớp trong bộ dữ liệu CIFAR-10. Mô hình được biên soạn

trên dòng 30 và 31 sau đó được huấn luyện trên dòng 35 và 36 trong suốt 40 chu kỳ.

```
26 # initialize the optimizer and model
27 print("[INFO] compiling model...")
28 opt = SGD(lr=0.01)
29 model = ShallowNet.build(width=32, height=32, depth=3, classes=10)
30 model.compile(loss="categorical_crossentropy", optimizer=opt,
31     metrics=["accuracy"])
32
33 # train the network
34 print("[INFO] training network...")
35 H = model.fit(trainX, trainY, validation_data=(testX, testY),
36     batch_size=32, epochs=40, verbose=1)
```

Đánh giá ShallowNet được thực hiện theo cách chính xác như ví dụ trước với bộ dữ liệu Động vật:

```
38 # evaluate the network
39 print("[INFO] evaluating network...")
40 predictions = model.predict(testX, batch_size=32)
41 print(classification_report(testY.argmax(axis=1),
42     predictions.argmax(axis=1), target_names=labelNames))
```

Cũng sẽ vẽ đồ thị về sự tồn thât và độ chính xác theo thời gian để có thể biết được mạng đang hoạt động như thế nào:

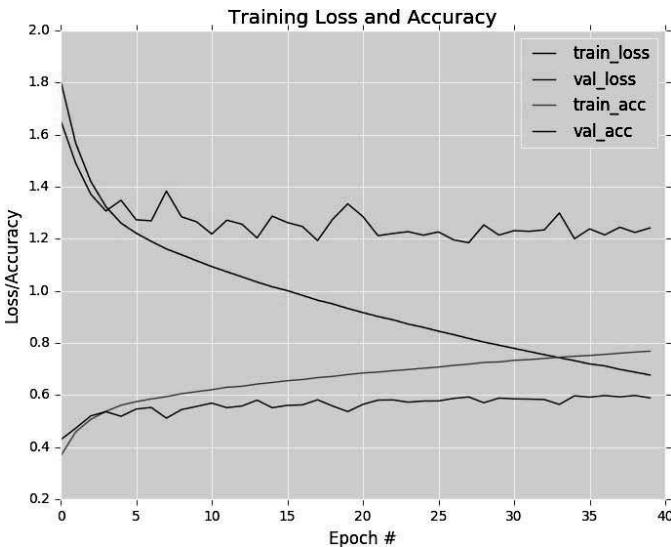
```
44 # plot the training loss and accuracy
45 plt.style.use("ggplot")
46 plt.figure()
47 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
48 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
49 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
50 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
51 plt.title("Training Loss and Accuracy")
52 plt.xlabel("Epoch #")
53 plt.ylabel("Loss/Accuracy")
54 plt.legend()
55 plt.show()
```

Để huấn luyện ShallowNet trên CIFAR-10, chỉ cần thực hiện lệnh sau:

```
$ python willownet_cifar10.py
```

Một lần nữa, chu kỳ khá nhanh do kiến trúc mạng nông và bộ dữ liệu tương đối nhỏ. Sử dụng GPU, đã thu được 5 giây trong khi CPU mất 22 giây cho mỗi chu kỳ.

Sau 40 chu kỳ ShallowNet được đánh giá và thấy rằng nó đạt được độ chính xác 60% trên bộ thử nghiệm, tăng từ độ chính xác 57% trước đó bằng cách sử dụng các mạng nơ-ron đơn giản.



Hình 2.15: Tồn thát và độ chính xác cho ShallowNet được huán luyện trên CIFAR-10. Mạng có được độ chính xác phân loại 60%; Tuy nhiên, nó là quá khóp. Độ chính xác cao hơn có thể đạt được bằng cách áp dụng chính quy hóa, sẽ đề cập sau trong cuốn sách này.

Quan trọng hơn, đồ thị tồn thát và độ chính xác trong Hình 2.15 cho biết thêm về quá trình huán luyện chứng tỏ rằng tồn thát xác nhận không tăng vọt. Sự tồn thát/độ chính xác trong quá trình huán luyện và kiểm tra bắt đầu chuyển hướng qua chu kỳ 10. Một lần nữa, điều này có thể được quy cho tỷ lệ học lớn hơn và thực tế không sử dụng các phương pháp để giúp chống lại quá khóp (tham số chính quy, bỏ học, tăng dữ liệu, v.v.).

Rất dễ bị hiện tượng quá khóp với tập dữ liệu CIFAR-10 do số lượng mẫu huán luyện có độ phân giải rất hạn chế. Khi trở nên thành thạo hơn khi xây dựng và huán luyện mạng nơ-ron tích chập tùy chỉnh, sẽ khám phá các phương pháp để tăng độ chính xác phân loại trên CIFAR-10 đồng thời giảm quá khóp.

2.2.3. Tóm tắt

Trong Mục 2.2 này, đã triển khai kiến trúc mạng nơ-ron tích chập đầu tiên, ShallowNet và huán luyện bộ dữ liệu động vật và CIFAR-10. ShallowNet thu được 71% độ chính xác phân loại trên Động vật, tăng 12% so với mức tốt nhất trước đây khi sử dụng các mạng nơ-ron đơn giản.

Khi được áp dụng cho CIFAR-10, ShallowNet đạt độ chính xác 60%, tăng so với mạng đạt độ chính xác 57% (tốt nhất trước đó) bằng cách sử dụng NN nhiều lớp đơn giản (và không có hiện tượng quá khóp đáng kể).

ShallowNet là một CNN cực kỳ đơn giản, chỉ sử dụng một lớp CONV - có thể đạt được độ chính xác cao hơn bằng cách huấn luyện các mạng sâu hơn với nhiều bộ hoạt động CONV => RELU => POOL.

2.3. LUU VÀ TAI MÔ HÌNH

Chúng ta đã học cách huấn luyện mạng nơ-ron tích chập đầu tiên bằng thư viện Keras. Tuy nhiên, có thể nhận thấy rằng mỗi lần muốn đánh giá mạng hoặc kiểm tra trên một tập hợp ảnh, trước tiên cần phải huấn luyện trước khi có thể thực hiện bất kỳ loại đánh giá nào. Yêu cầu này có thể rất phức tạp.

Chỉ làm việc với một mạng đơn giản trên một tập dữ liệu nhỏ thì có thể huấn luyện tương đối nhanh, nhưng nếu mạng sâu và cần huấn luyện trên một tập dữ liệu lớn hơn nhiều, thì phải mất nhiều giờ hoặc thậm chí vài ngày để huấn luyện? Có cần phải đầu tư thời gian và nguồn lực để huấn luyện mang mọi lúc không? Hoặc có cách nào để lưu mô hình vào ổ cứng sau khi huấn luyện xong và sau đó chỉ cần tải từ ổ cứng khi muốn phân loại ảnh mới? Quá trình lưu và tải một mô hình được huấn luyện được gọi là tuần tự hóa mô hình và là chủ đề chính của mục này.

2.3.1. Mô hình nối tiếp vào ổ cứng

Sử dụng thư viện Keras, tuần tự hóa mô hình đơn giản như gọi `model.save` trên một mô hình được huấn luyện và sau đó tải thông qua hàm `load_model`. Trong phần đầu tiên mục 2.3 này, sửa đổi tập lệnh huấn luyện ShallowNet từ phần trước để tuần tự hóa mạng sau khi được huấn luyện về bộ dữ liệu động vật. Sau đó, tạo một tập lệnh Python thứ hai giải thích cách tải mô hình tuần tự hóa từ ổ cứng.

Bắt đầu với phần huấn luyện - mở một tệp mới, đặt tên là `willownet_train.py` và chèn chương trình sau đây:

```
 1 # import the necessary packages
 2 from sklearn.preprocessing import LabelBinarizer
 3 from sklearn.model_selection import train_test_split
 4 from sklearn.metrics import classification_report
 5 from pyimagesearch.preprocessing import ImageToArrayPreprocessor
 6 from pyimagesearch.preprocessing import SimplePreprocessor
 7 from pyimagesearch.datasets import SimpleDatasetLoader
 8 from pyimagesearch.nn.conv import ShallowNet
 9 from keras.optimizers import SGD
10 from imutils import paths
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Dòng 2-13: nhập các gói thư viện Python cần thiết. Phần lớn chương

trình trong ví dụ này giống hệt với willownet_animals.py từ mục 2.2. Xem xét toàn bộ tệp, vì mục đích chính xác và chắc chắn gọi ra các thay đổi quan trọng được thực hiện để thực hiện tuần tự hóa mô hình, nhưng để đánh giá chi tiết về cách huấn luyện ShallowNet trên bộ dữ liệu động vật, vui lòng tham khảo phần thực thi ShallowNet ở mục 2.2.

Tiếp theo phân tích các dòng lệnh:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-d", "--dataset", required=True,
18     help="path to input dataset")
19 ap.add_argument("-m", "--model", required=True,
20     help="path to output model")
21 args = vars(ap.parse_args())
```

Tập lệnh trước chỉ yêu cầu một hàm duy nhất, --dataset, là đường dẫn đến bộ dữ liệu đầu vào là động vật. Tuy nhiên, như có thể thấy, đã thêm một hàm khác ở đây - --model, đó là đường dẫn đến nơi muốn lưu mạng sau khi hoàn tất huấn luyện.

Bây giờ có thể lấy các đường dẫn đến ảnh trong --dataset, khởi tạo bộ tiền xử lý và tải tập dữ liệu ảnh từ ổ cứng

```
23 # grab the list of images that we'll be describing
24 print("[INFO] loading images...")
25 imagePaths = list(paths.list_images(args["dataset"]))
26
27 # initialize the image preprocessors
28 sp = SimplePreprocessor(32, 32)
29 iap = ImageToArrayPreprocessor()
30
31 # load the dataset from disk then scale the raw pixel intensities
32 # to the range [0, 1]
33 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
34 (data, labels) = sdl.load(imagePaths, verbose=500)
35 data = data.astype("float") / 255.0
```

Bước tiếp theo là phân vùng dữ liệu thành 2 phần là huấn luyện và thử nghiệm, cùng với việc mã hóa nhãn dưới dạng vecto:

```
37 # partition the data into training and testing splits using 75% of
38 # the data for training and the remaining 25% for testing
39 (trainX, testX, trainY, testY) = train_test_split(data, labels,
40     test_size=0.25, random_state=42)
41
42 # convert the labels from integers to vectors
43 trainY = LabelBinarizer().fit_transform(trainY)
44 testY = LabelBinarizer().fit_transform(testY)
```

Huấn luyện ShallowNess được xử lý bởi chương trình bên dưới:

```
46 # initialize the optimizer and model
47 print("[INFO] compiling model...")
48 opt = SGD(lr=0.005)
49 model = ShallowNet.build(width=32, height=32, depth=3, classes=3)
50 model.compile(loss="categorical_crossentropy", optimizer=opt,
51     metrics=["accuracy"])
52
53 # train the network
54 print("[INFO] training network...")
55 H = model.fit(trainX, trainY, validation_data=(testX, testY),
56     batch_size=32, epochs=100, verbose=1)
```

Bây giờ mạng đã được huấn luyện, cần lưu vào ổ cứng. Quá trình này đơn giản như gọi `model.save` và cung cấp đường dẫn đến nơi mạng đầu ra được lưu vào ổ cứng:

```
58 # save the network to disk
59 print("[INFO] serializing network...")
60 model.save(args["model"])
```

Phương thức `.save` lấy trọng số và trạng thái trình tối ưu hóa và tuần tự hóa chúng vào ổ cứng ở định dạng HDF5. Như thấy trong phần tiếp theo, việc tải các trọng số này từ ổ cứng cũng dễ như lưu trữ chúng.

Từ đây, đánh giá mạng:

```
62 # evaluate the network
63 print("[INFO] evaluating network...")
64 predictions = model.predict(testX, batch_size=32)
65 print(classification_report(testY.argmax(axis=1),
66     predictions.argmax(axis=1),
67     target_names=["cat", "dog", "panda"]))
```

Hiển thị sự tồn thắt và độ chính xác của mô hình:

```
69 # plot the training loss and accuracy
70 plt.style.use("ggplot")
71 plt.figure()
72 plt.plot(np.arange(0, 100), H.history["loss"], label="train_loss")
73 plt.plot(np.arange(0, 100), H.history["val_loss"], label="val_loss")
74 plt.plot(np.arange(0, 100), H.history["acc"], label="train_acc")
75 plt.plot(np.arange(0, 100), H.history["val_acc"], label="val_acc")
76 plt.title("Training Loss and Accuracy")
77 plt.xlabel("Epoch #")
78 plt.ylabel("Loss/Accuracy")
79 plt.legend()
80 plt.show()
```

Để chạy tập lệnh, chỉ cần thực hiện lệnh sau:

```
$ python shallownet_train.py --dataset ../datasets/animals \
--model shallownet_weights.hdf5
```

Sau khi mạng đã hoàn thành huấn luyện, liệt kê nội dung thư mục:

```
$ ls  
shallownet_load.py shallownet_train.py shallownet_weights.hdf5
```

Và thấy một tệp có tên willownet_weights.hdf5 - tệp này là mạng được tuân tự hóa. Bước tiếp theo là lấy mạng đã lưu này và tải từ ổ cứng.

2.3.2. Tải mô hình được huấn luyện trước từ ổ cứng

Bây giờ đã huấn luyện mô hình và tuân tự hóa, cần tải mô hình từ ổ cứng. Đây là một ứng dụng thực tế việc tuân tự hóa mô hình, trình bày cách phân loại các ảnh riêng lẻ từ bộ dữ liệu động vật và sau đó hiển thị các ảnh được phân loại lên màn hình:

Mở một tệp mới, đặt tên là willownet_load.py:

```
1 # import the necessary packages  
2 from pyimagesearch.preprocessing import ImageToArrayPreprocessor  
3 from pyimagesearch.preprocessing import SimplePreprocessor  
4 from pyimagesearch.datasets import SimpleDatasetLoader  
5 from keras.models import load_model  
6 from imutils import paths  
7 import numpy as np  
8 import argparse  
9 import cv2
```

Bắt đầu bằng cách nhập các gói thư viện Python cần thiết. Các dòng 2-4 nhập các lớp sử dụng để xây dựng đường dẫn tiêu chuẩn thay đổi kích thước ảnh thành kích thước cố định, chuyển đổi thành mảng tương thích Keras và sau đó sử dụng các bộ tiền xử lý này để tải toàn bộ dữ liệu ảnh vào bộ nhớ.

Hàm thực tế được sử dụng để tải mô hình được huấn luyện từ ổ cứng là load_model trên Dòng 5. Hàm này chịu trách nhiệm chấp nhận đường dẫn đến mạng được huấn luyện (tệp HDF5), giải mã trọng số và trình tối ưu hóa bên trong tệp HDF5 và đặt trọng số bên trong tệp HDF5, kiến trúc vì vậy có thể (1) tiếp tục huấn luyện hoặc (2) sử dụng mạng để phân loại ảnh mới.

Cũng nhập các ràng buộc OpenCV trên Dòng 9 để có thể vẽ nhãn phân loại trên ảnh và hiển thị chúng trên màn hình.

Tiếp theo, phân tích một số dòng lệnh:

```
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14     help="path to input dataset")
15 ap.add_argument("-m", "--model", required=True,
16     help="path to pre-trained model")
17 args = vars(ap.parse_args())
18
19 # initialize the class labels
20 classLabels = ["cat", "dog", "panda"]
```

Giống như trong willownet_save.py, cần hai đối số dòng lệnh:

1. --bộ dữ liệu: đường dẫn đến thư mục chứa ảnh muốn phân loại (trong trường hợp này là bộ dữ liệu Động vật).

2.--model: đường dẫn đến mạng được huấn luyện nối tiếp trên ổ cứng.

Dòng 20 sau đó khởi tạo một danh sách các nhãn lớp cho bộ dữ liệu động vật.

Chương trình tiếp theo xử lý lấy mẫu ngẫu nhiên 10 đường dẫn ảnh từ bộ dữ liệu động vật để phân loại:

```
22 # grab the list of images in the dataset then randomly sample
23 # indexes into the image paths list
24 print("[INFO] sampling images...")
25 imagePaths = np.array(list(paths.list_images(args["dataset"])))
26 idxs = np.random.randint(0, len(imagePaths), size=(10,))
27 imagePaths = imagePaths[idxs]
```

Mỗi trong số mười ảnh này cần được xử lý trước, vì vậy, khởi tạo bộ xử lý trước và tải 10 ảnh từ ổ cứng:

```
29 # initialize the image preprocessors
30 sp = SimplePreprocessor(32, 32)
31 iap = ImageToArrayPreprocessor()
32
33 # load the dataset from disk then scale the raw pixel intensities
34 # to the range [0, 1]
35 sdl = SimpleDatasetLoader(preprocessors=[sp, iap])
36 (data, labels) = sdl.load(imagePaths)
37 data = data.astype("float") / 255.0
```

Lưu ý cách tiền xử lý hình ảnh giống như cách tạo ra ảnh trong quá trình huấn luyện. Không thực hiện bước này có thể dẫn đến phân loại không chính xác vì mạng được trình bày với các mẫu mà không thể nhận ra. Luôn luôn cẩn thận để đảm bảo hình ảnh kiểm tra được xử lý trước giống như hình ảnh huấn luyện.

Tiếp theo, tải mạng đã lưu từ ổ cứng:

```
39 # load the pre-trained network
40 print("[INFO] loading pre-trained network...")
41 model = load_model(args["model"])
```

Tải mạng nối tiếp đơn giản như gọi `load_model` và cung cấp đường dẫn đến mô hình tập tin HDF5 nằm trên ổ cứng. Khi mô hình được tải, có thể dự đoán trên mười ảnh:

```
43 # make predictions on the images
44 print("[INFO] predicting...")
45 preds = model.predict(data, batch_size=32).argmax(axis=1)
```

Hãy nhớ rằng phương thức mô hình `.predict` trả về một danh sách xác suất cho mỗi ảnh trong dữ liệu, một xác suất tương ứng cho mỗi nhãn lớp, tương ứng. Lấy `argmax` trên trực = 1 tìm thấy chỉ mục nhãn lớp với xác suất lớn nhất cho mỗi ảnh.

Bây giờ đã có dự đoán, trực quan kết quả:

```
47 # loop over the sample images
48 for (i, imagePath) in enumerate(imagePaths):
49     # load the example image, draw the prediction, and display it
50     # to our screen
51     image = cv2.imread(imagePath)
52     cv2.putText(image, "Label: {}".format(classLabels[preds[i]]),
53                 (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
54     cv2.imshow("Image", image)
55     cv2.waitKey(0)
```

Trên Dòng 48, bắt đầu lặp qua 10 đường dẫn ảnh được lấy mẫu ngẫu nhiên. Đối với mỗi ảnh, tải từ ổ cứng (Dòng 51) và vẽ dự đoán nhãn lớp trên chính ảnh (Dòng 52 và 53). Ảnh đầu ra sau đó được hiển thị trên màn hình trên dòng 54 và 55.

Để thử `willownet_load.py`, hãy thực hiện lệnh sau:

```
$ python shallownet_load.py --dataset ../datasets/animals \
--model shallownet_weights.hdf5
[INFO] sampling images...
[INFO] loading pre-trained network...
[INFO] predicting...
```

Dựa trên đầu ra, có thể thấy rằng ảnh đã được lấy mẫu, trọng số ShallowNet được huấn luyện trước đã được tải từ ổ cứng và ShallowNet đã đưa ra dự đoán về ảnh. Đã bao gồm một mẫu dự đoán từ ShallowNet được vẽ trên chính các ảnh trong Hình 2.16.



Hình 2.16: Một mẫu ảnh được phân loại chính xác bởi CNN ShallowNet

Hãy nhớ rằng ShallowNet đang đạt được độ chính xác phân loại 70% trên bộ dữ liệu động vật, nghĩa là gần như cứ một trong ba ảnh ví dụ được phân loại không chính xác. Hơn nữa, dựa theo hàm classification_report từ mục ShallowNet với bộ dữ liệu động vật biết rằng mạng vẫn đang cố gắng để phân biệt giữa chó và mèo. Khi tiếp tục quá trình áp dụng học sâu vào các nhiệm vụ phân loại thị giác máy tính, nên xem xét các phương pháp để giúp tăng độ chính xác phân loại.

2.3.3. Tóm tắt

Trong mục 2.3 này, đã học cách:

1. Huấn luyện một mạng.
2. Tối ưu hóa trọng số của mạng và trạng thái tối ưu hóa vào ổ cứng.
3. Tải xuống mạng được huấn luyện và phân loại ảnh.

Sau đó trong Chương 3, khám phá cách có thể lưu trọng số mô hình vào ổ cứng sau mỗi chu kỳ, cho phép kiểm tra điểm kiểm soát mạng và chọn một hoạt động tốt nhất. Tiết kiệm trọng số mô hình trong quá trình huấn luyện thực tế cũng cho phép khởi động lại huấn luyện từ một điểm cụ thể nếu mạng bắt đầu có dấu hiệu quá khóp.

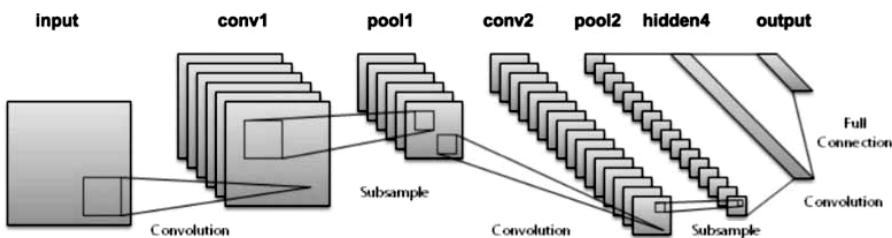
2.4. LENET: NHẬN DẠNG CHỮ SỐ VIẾT TAY

Kiến trúc LeNet là một công trình tinh túy trong cộng đồng học sâu, lần đầu tiên được giới thiệu bởi LeCun et al. trong bài báo năm 1998 họ, việc học dựa trên Gradient được áp dụng cho nhận dạng tài liệu [19]. Như tên của bài báo cho thấy, động lực thúc đẩy thực hiện LeNet chủ yếu là cho nhận dạng ký tự quang học (OCR).

Kiến trúc LeNet đơn giản và nhỏ, rất hoàn hảo cho việc giảng dạy những điều cơ bản về CNN.

Trong mục này, tìm cách tái tạo các thực nghiệm tương tự như LeCun trong bài báo năm 1998 của họ. Bắt đầu bằng cách xem xét kiến trúc LeNet và sau đó triển khai mạng bằng Keras. Cuối cùng, đánh giá LeNet trên bộ dữ liệu MNIST để nhận dạng chữ số viết tay.

2.4.1. Kiến trúc LENET



Hình 2.17: Kiến trúc LeNet bao gồm hai chuỗi CONV => TANH => Tập hợp lớp POOL theo sau là lớp được kết nối đầy đủ và đầu ra softmax.

Bây giờ người đọc đã hiểu các khái niệm xây dựng mạng nơ ron tích chập trong mục 2.2 bằng ShallowNet, đã sẵn sàng thực hiện bước tiếp theo và thảo luận về LeNet. Kiến trúc LeNet (Hình 2.17) là một mạng xuất sắc, mạng này nhỏ và dễ hiểu - nhưng đủ lớn để cung cấp kết quả thú vị.

Bảng 2.4: Tóm tắt bảng về kiến trúc LeNet. Kích thước khối lượng đầu ra được bao gồm cho mỗi lớp, cùng với kích thước bộ lọc tích hợp / kích thước nhóm khi có liên quan.

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$28 \times 28 \times 1$	
CONV	$28 \times 28 \times 20$	$5 \times 5, K = 20$
ACT	$28 \times 28 \times 20$	
POOL	$14 \times 14 \times 20$	2×2
CONV	$14 \times 14 \times 50$	$5 \times 5, K = 50$
ACT	$14 \times 14 \times 50$	
POOL	$7 \times 7 \times 50$	2×2
FC	500	
ACT	500	
FC	10	
SOFTMAX	10	

Hơn nữa, sự kết hợp LeNet + MNIST có thể dễ dàng chạy trên CPU, giúp người mới bắt đầu dễ dàng thực hiện bước đầu tiên trong việc học sâu và CNN. Theo nhiều cách, LeNet + MNIST là “Hello World”, tương đương với việc học sâu áp dụng cho phân loại ảnh. Kiến trúc LeNet bao gồm các lớp sau, sử dụng mẫu CONV => ACT => POOL từ mục 2.1.3:

INPUT => CONV => TANH => POOL => CONV => TANH => POOL
=>FC => TANH => FC

Lưu ý cách kiến trúc LeNet sử dụng hàm tác động tanh thay vì ReLU phổ biến hơn. Trở lại năm 1998, ReLU đã không được sử dụng trong các bài toán học sâu - việc sử dụng tanh hoặc sigmoid như một hàm tác động là phổ biến hơn. Khi triển khai LeNet ngày hôm nay, phổ biến để trao đổi TANH cho RELU – cần tuân theo hướng dẫn tương tự này và sử dụng ReLU làm hàm tác động sau trong mục này.

Bảng 2.4 tóm tắt các tham số cho kiến trúc LeNet. Lớp đầu vào lấy một ảnh đầu vào với 28 hàng, 28 cột và một kênh duy nhất (mức xám) cho độ sâu (nghĩa là kích thước các ảnh bên trong bộ dữ liệu MNIST). Sau đó, tìm hiểu 20 bộ lọc, mỗi bộ lọc có kích thước 5×5 . Lớp CONV được theo sau bởi kích hoạt ReLU, sau đó là nhóm tối đa với kích thước 2×2 và bước trượt 2×2 . Khối tiếp theo kiến trúc theo cùng một mẫu, lần này học 50 bộ lọc 5×5 .

LeNet phổ biến để thấy số lượng lớp CONV tăng trong các lớp sâu hơn của mạng khi kích thước đầu vào không gian thực tế giảm. Sau đó có hai lớp FC. Lớp FC đầu tiên chứa 500 nút ẩn theo sau là kích hoạt ReLU. Lớp FC cuối cùng kiểm soát số lượng nhãn lớp đầu ra (0-9; một cho mỗi 10 chữ số có thể). Cuối cùng, áp dụng kích hoạt softmax để có được xác suất cho từng lớp.

2.4.2. Triển khai LENET

Đưa ra Bảng 2.4 ở trên, đã sẵn sàng triển khai kiến trúc LeNet bằng thư viện Keras. Bắt đầu bằng cách thêm một tệp mới có tên lenet.py bên trong mô-đun con pyimagesearch.nn.conv - tệp này lưu trữ triển khai LeNet thực tế:

```
--- pyimagesearch
|   |--- __init__.py
|   |--- nn
|   |   |--- __init__.py
...
|   |   |--- conv
|   |   |   |--- __init__.py
|   |   |   |--- lenet.py
|   |   |   |--- shalownet.py
```

Từ đó, mở lenet.py và có thể bắt đầu mã hóa:

```
1 # import the necessary packages
2 from keras.models import Sequential
3 from keras.layers.convolutional import Conv2D
4 from keras.layers.convolutional import MaxPooling2D
5 from keras.layers.core import Activation
6 from keras.layers.core import Flatten
7 from keras.layers.core import Dense
8 from keras import backend as K
```

Dòng 2-8 xử lý việc nhập các gói Python cần thiết, các lần nhập này hoàn toàn giống với triển khai ShallowNet từ mục 2.2 và tạo thành tập hợp nhập bắt buộc cần thiết khi xây dựng (gắn) bất kỳ CNN nào bằng Keras.

Sau đó, xác định phương thức xây dựng LeNet bên dưới, được sử dụng để thực sự xây dựng kiến trúc mạng:

```
10 class LeNet:
11     @staticmethod
12     def build(width, height, depth, classes):
13         # initialize the model
14         model = Sequential()
15         inputShape = (height, width, depth)
16
17         # if we are using "channels first", update the input shape
18         if K.image_data_format() == "channels_first":
19             inputShape = (depth, height, width)
```

Phương thức xây dựng yêu cầu bốn tham số:

1. Chiều rộng ảnh đầu vào.
2. Chiều cao ảnh đầu vào.
3. Số lượng kênh (độ sâu) ảnh.
4. Các nhãn lớp số trong nhiệm vụ phân loại.

Lớp Sequential, khôi xây dựng các mạng tuần tự xếp chồng tuần tự một lớp lên trên lớp kia được khởi tạo trên Dòng 14. Sau đó, khởi tạo inputShape như thể sử dụng thứ tự các kênh ra lệnh. Trong trường hợp cấu hình Keras được thiết lập để sử dụng thứ tự các kênh đầu tiên, cập nhật inputShape trên dòng 18 và 19.

Tập hợp đầu tiên CONV => RELU => Các lớp POOL được xác định bên dưới:

```
21         # first set of CONV => RELU => POOL layers
22         model.add(Conv2D(20, (5, 5), padding="same",
23                         input_shape=inputShape))
24         model.add(Activation("relu"))
25         model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

Lớp CONV tim hiểu 20 bộ lọc, mỗi bộ lọc có kích thước 5×5 . Sau đó, áp dụng hàm tác động ReLU, sau đó là nhóm 2×2 với một bước 2×2 , do đó giảm 75% kích thước số lượng đầu vào.

Một bộ CONV => RELU => POOL sau đó được áp dụng, lần này học 50 bộ lọc thay vì 20:

```
27     # second set of CONV => RELU => POOL layers
28     model.add(Conv2D(50, (5, 5), padding="same"))
29     model.add(Activation("relu"))
30     model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

Số lượng đầu vào sau đó có thể được làm phẳng và một lớp được kết nối đầy đủ với 500 nút có thể được áp dụng:

```
32     # first (and only) set of FC => RELU layers
33     model.add(Flatten())
34     model.add(Dense(500))
35     model.add(Activation("relu"))
```

Tiếp theo là phân loại softmax cuối cùng:

```
37     # softmax classifier
38     model.add(Dense(classes))
39     model.add(Activation("softmax"))
40
41     # return the constructed network architecture
42     return model
```

Bây giờ đã mã hóa kiến trúc LeNet, có thể chuyển sang áp dụng vào bộ dữ liệu MNIST.

2.4.3. LENET trên tập dữ liệu MNIST

Bước tiếp theo là tạo một tập lệnh thực hiện các bước sau:

1. Tải tập dữ liệu MNIST từ ổ cứng.
2. Khởi tạo kiến trúc LeNet.
3. Huấn luyện LeNet.
4. Đánh giá hiệu suất mạng.

Để huấn luyện và đánh giá LeNet trên MNIST, hãy tạo một tệp mới có tên lenet_mnist.py và có thể bắt đầu:

```
1 # import the necessary packages
2 from pyimagesearch.nn.conv import LeNet
3
4 from keras.optimizers import SGD
5 from sklearn.preprocessing import LabelBinarizer
6 from sklearn.model_selection import train_test_split
7 from sklearn.metrics import classification_report
8 from sklearn import datasets
9 from keras import backend as K
10 import matplotlib.pyplot as plt
11 import numpy as np
```

Tại thời điểm này, việc nhập Python bắt đầu cảm thấy khá chuẩn với một mẫu đáng chú ý xuất hiện. Trong phần lớn các ví dụ trong cuốn sách này, phải nhập:

1. Một kiến trúc mạng để huấn luyện.
2. Một trình tối ưu hóa để huấn luyện mạng (trong trường hợp này là SGD).
3. Một hàm được sử dụng để xây dựng các phần tách huấn luyện và thử nghiệm một tập dữ liệu đã cho.
4. Một hàm để tính toán một báo cáo phân loại để có thể đánh giá hiệu suất bộ phân loại.

Bộ dữ liệu MNIST đã được xử lý trước nên chỉ cần tải thông qua lệnh gọi hàm sau:

```
12 # grab the MNIST dataset (if this is your first time using this
13 # dataset then the 55MB download may take a minute)
14 print("[INFO] accessing MNIST...")
15 dataset = datasets.fetch_mldata("MNIST Original")
16 data = dataset.data
```

Dòng 15 tải tập dữ liệu MNIST từ ổ cứng. Nếu đây là lần đầu tiên gọi hàm fetch_mldata bằng chuỗi “MNIST Original”, thì bộ dữ liệu MNIST cần phải được tải xuống từ kho lưu trữ dữ liệu mldata.org. Tập dữ liệu MNIST được tuần tự hóa thành một tệp 55 MB, do đó tùy thuộc vào kết nối internet, quá trình tải xuống này có thể mất từ vài giây đến vài phút.

```
18 # if we are using "channels first" ordering, then reshape the
19 # design matrix such that the matrix is:
20 # num_samples x depth x rows x columns
21 if K.image_data_format() == "channels_first":
22     data = data.reshape(data.shape[0], 1, 28, 28)
23
24 # otherwise, we are using "channels last" ordering, so the design
25 # matrix shape should be: num_samples x rows x columns x depth
26 else:
27     data = data.reshape(data.shape[0], 28, 28, 1)
```

Điều quan trọng cần lưu ý là mỗi mẫu MNIST bên trong dữ liệu được biểu thị bằng một vectơ 784-d (tức là cường độ điểm ảnh thô) một ảnh 28×28 thang độ xám. Do đó, cần định hình lại ma trận dữ liệu tùy thuộc vào việc đang sử dụng các kênh đầu tiên kênh hay kênh cuối cùng sắp xếp theo thứ tự:

- Nếu đang thực hiện thứ tự các kênh thứ tự đầu tiên (Dòng 21 và

22), thì ma trận dữ liệu được định hình lại sao cho số lượng mẫu là mục nhập đầu tiên trong ma trận, kênh đơn là mục nhập thứ hai, theo sau là số hàng và cột (tương ứng 28 và 28).

- Mặt khác, giả định rằng đang sử dụng thứ tự các kênh cuối cùng theo thứ tự, trong trường hợp ma trận được định hình lại thành số lượng mẫu trước, số hàng, số cột và cuối cùng là số kênh (Dòng 26 và 27).

Bây giờ, ma trận dữ liệu đã được định hình đúng, có thể thực hiện phân tách huấn luyện và kiểm tra, chú ý mở rộng cường độ pixel ảnh đến phạm vi [0, 1] trước tiên:

```
29 # scale the input data to the range [0, 1] and perform a train/test
30 # split
31 (trainX, testX, trainY, testY) = train_test_split(data / 255.0,
32         dataset.target.astype("int"), test_size=0.25, random_state=42)
33
34 # convert the labels from integers to vectors
35 le = LabelBinarizer()
36 trainY = le.fit_transform(trainY)
37 testY = le.transform(testY)
```

Sau khi phân tách dữ liệu, cũng mã hóa nhãn lớp dưới dạng các vectơ thay vì các giá trị số nguyên đơn. Ví dụ: nếu nhãn lớp cho một mẫu nhất định là 3, thì đầu ra mã hóa nhãn là: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

Lưu ý: tất cả các mục trong vectơ bằng 0 ngoại trừ chỉ mục thứ tư được đặt thành 1.

Quá trình thực hiện được thiết lập để huấn luyện LeNet trên MNIST:

```
39 # initialize the optimizer and model
40 print("[INFO] compiling model...")
41 opt = SGD(lr=0.01)
42 model = LeNet.build(width=28, height=28, depth=1, classes=10)
43 model.compile(loss="categorical_crossentropy", optimizer=opt,
44     metrics=["accuracy"])
45
46 # train the network
47 print("[INFO] training network...")
48 H = model.fit(trainX, trainY, validation_data=(testX, testY),
49     batch_size=128, epochs=20, verbose=1)
```

Dòng 41 khởi tạo trình tối ưu hóa SGD với tỷ lệ học là 0,01. Bản thân LeNet được khởi tạo trên Dòng 42, chỉ ra rằng tất cả các ảnh đầu vào trong tập dữ liệu có chiều rộng 28 pixel, cao 28 pixel và có độ sâu là 1. Cho rằng có mười lớp trong bộ dữ liệu MNIST (mỗi lớp có một chữ số 0-9), đặt các lớp = 10.

Dòng 43 và 44 biên dịch mô hình bằng cách sử dụng tốn thất entropy

chéo làm hàm tổn thất. Dòng 48 và 49 huấn luyện LeNet trên MNIST với tổng số 20 chu kỳ sử dụng kích thước đợt nhỏ là 128.

Cuối cùng, có thể đánh giá hiệu suất trên mạng cũng như về sự tổn thất và độ chính xác theo thời gian trong khôi mã cuối cùng bên dưới:

```
51 # evaluate the network
52 print("[INFO] evaluating network...")
53 predictions = model.predict(testX, batch_size=128)
54 print(classification_report(testY.argmax(axis=1),
55     predictions.argmax(axis=1),
56     target_names=[str(x) for x in le.classes_]))
57
58 # plot the training loss and accuracy
59 plt.style.use("ggplot")
60 plt.figure()
61 plt.plot(np.arange(0, 20), H.history["loss"], label="train_loss")
62 plt.plot(np.arange(0, 20), H.history["val_loss"], label="val_loss")
63 plt.plot(np.arange(0, 20), H.history["acc"], label="train_acc")
64 plt.plot(np.arange(0, 20), H.history["val_acc"], label="val_acc")
65 plt.title("Training Loss and Accuracy")
66 plt.xlabel("Epoch #")
67 plt.ylabel("Loss/Accuracy")
68 plt.legend()
69 plt.show()
```

Đã đề cập đến thực tế này trước đây trong thực thi ShallowNet với bộ dữ liệu động vật ở mục 2.2 khi đánh giá ShallowNet, nhưng hãy chắc chắn rằng hiểu Dòng 53 đang làm gì khi `model.predict` được gọi. Đối với mỗi mẫu trong `testX`, kích thước 1 ô 128 được xây dựng và sau đó được chuyển qua mạng để phân loại. Sau khi tất cả các điểm dữ liệu thử nghiệm đã được phân loại, biến dự đoán được trả về.

Biến dự đoán thực sự là một mảng NumPy có hình dạng (`len(testX)`, 10) ngũ ý rằng hiện có 10 xác suất được liên kết với mỗi nhãn lớp cho mỗi điểm dữ liệu trong `testX`. Lấy dự đoán `.argmax` (trục = 1) trong phân loại numport trên dòng 54-56 tìm thấy chỉ số nhãn có xác suất lớn nhất (nghĩa là, phân loại đầu ra cuối cùng). Dựa vào phân loại cuối cùng từ mạng, có thể so sánh các nhãn lớp dự đoán với các nhãn thực tế.

Để thực thi tập lệnh, chỉ cần ban hành lệnh sau:

```
$python lenet_mnist.py
```

Tập dữ liệu MNIST sau đó nên được tải xuống và/hoặc tải từ ổ cứng và huấn luyện sẽ bắt đầu:

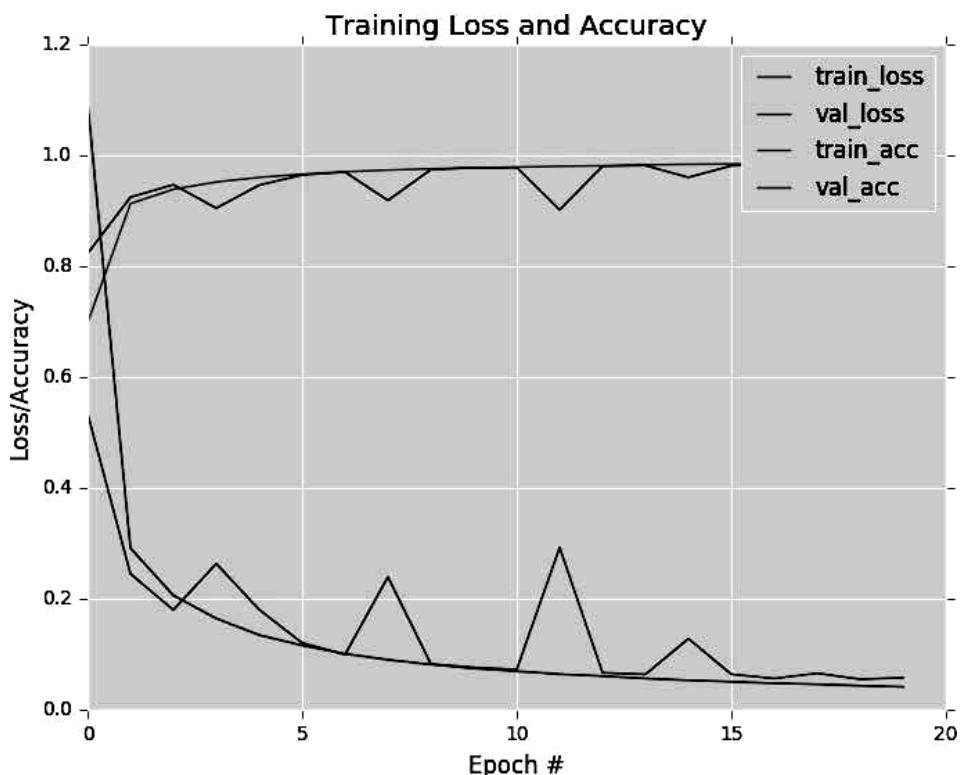
```
[INFO] accessing MNIST...
[INFO] compiling model...
[INFO] training network...
Train on 52500 samples, validate on 17500 samples
Epoch 1/20
3s - loss: 1.0970 - acc: 0.6976 - val_loss: 0.5348 - val_acc: 0.8228
...
Epoch 20/20
3s - loss: 0.0411 - acc: 0.9877 - val_loss: 0.0576 - val_acc: 0.9837
[INFO] evaluating network...

```

	precision	recall	f1-score	support
0	0.99	0.99	0.99	1677
1	0.99	0.99	0.99	1935
2	0.99	0.98	0.99	1767
3	0.99	0.97	0.98	1766
4	1.00	0.98	0.99	1691
5	0.99	0.98	0.98	1653
6	0.99	0.99	0.99	1754
7	0.98	0.99	0.99	1846
8	0.94	0.99	0.97	1702
9	0.98	0.98	0.98	1709
avg / total	0.98	0.98	0.98	17500

Sử dụng GPU Titan X, đã đạt được chu kỳ ba giây. Chỉ sử dụng CPU, số giây trên mỗi chu kỳ đã tăng lên ba mươi. Sau khi huấn luyện hoàn tất, có thể thấy rằng LeNet đang đạt được độ chính xác phân loại 98%, tăng rất nhiều so với khi sử dụng các mạng nơ-ron tiêu chuẩn trong Chương 1 mục 1.1.

Hơn nữa, nhìn vào biểu đồ tồn thết và độ chính xác theo thời gian trong Hình 2.18 chúng tôi rằng mạng đang hoạt động khá tốt. Chỉ sau 5 chu kỳ, LeNet đã đạt độ chính xác phân loại 96%. Mắt mát trên cả bộ dữ liệu huấn luyện và bộ dữ liệu kiểm tra tiếp tục giảm chỉ với một số ít các đột biến nhỏ do tỷ lệ học không đổi và không phân rã (một khái niệm đề cập sau trong Chương 3). Vào cuối chu kỳ thứ 20, đang đạt độ chính xác 98% trên bộ thử nghiệm.



Hình 2.18: Huấn luyện LeNet trên MNIST. Chỉ sau hai mươi chu kỳ, đã đạt được độ chính xác phân loại 98%.

Biểu đồ này thể hiện sự ổn định và chính xác LeNet trên MNIST được cho là biểu đồ tinh túy đang tìm kiếm: sự ổn định về huấn luyện và xác nhận và độ chính xác bắt chước lẫn nhau (gần như) không có dấu hiệu quá khớp.

Ngoài ra còn có vấn đề là bộ dữ liệu MNIST được xử lý trước rất nhiều và không đại diện cho các vấn đề phân loại ảnh gấp phải trong thế giới thực. Các nhà nghiên cứu có xu hướng sử dụng bộ dữ liệu MNIST làm chuẩn để đánh giá các thuật toán phân loại mới. Nếu các phương thức của họ không thể đạt được độ chính xác phân loại $> 95\%$, thì có một lỗi hỏng trong logic toán hoặc chính việc thực hiện.

Tuy nhiên, áp dụng LeNet cho MNIST là một cách tuyệt vời để có được kết quả đầu tiên khi áp dụng học sâu vào các vấn đề phân loại ảnh khi bắt chước bài báo của LeCun et al..

2.4.4. Tóm tắt

Trong mục 2.4 này, chúng ta đã tìm hiểu kiến trúc LeNet, được giới

thiệu bởi LeCun et al. trong bài báo năm 1998, học dựa trên Gradient được áp dụng cho nhận dạng tài liệu [32]. LeNet là một công trình tinh túy trong tài liệu học sâu - đã chứng minh một cách triệt để cách các mạng nơ-ron có thể được huấn luyện để nhận ra các vật thể trong ảnh theo cách từ đầu đến cuối (nghĩa là không phải trích xuất tính năng, mạng có thể học được mô hình từ chính ảnh).

LeNet theo tiêu chuẩn ngày hôm nay vẫn được coi là một mạng đơn giản. Chỉ với bốn lớp có thể huấn luyện (hai lớp CONV và hai lớp FC), độ sâu LeNet nhạt so với độ sâu các kiến trúc hiện đại như VGG (16 và 19 lớp) và ResNet (hơn 100 lớp)).

Trong phần tiếp theo, chúng ta thảo luận về một biến thể kiến trúc VGGNet còn được gọi là MiniVG- GNet. Biến thể kiến trúc này sử dụng các nguyên tắc chỉ dẫn chính xác giống như Simonyan và Zisserman, công việc [57], nhưng làm giảm độ sâu, cho phép huấn luyện mạng trên các bộ dữ liệu nhỏ hơn.

2.5. MINIVGGNET: ĐI SÂU HƠN VỚI CNNS

Trong mục 2.4, đã thảo luận về LeNet, một mạng nơ-ron tích chập trong tài liệu học sâu và thị giác máy tính. VGGNet, (đôi khi được gọi đơn giản là VGG), lần đầu tiên được Simonyan và Zisserman giới thiệu trong bài báo năm 2014, nặng nơ-ron tích chập học rất sâu để nhận dạng ảnh quy mô lớn [57]. Đóng góp chính của họ là chứng minh rằng kiến trúc - với các bộ lọc rất nhỏ (3×3) có thể được huấn luyện ở độ sâu ngày càng cao hơn (16-19 lớp) và có được phân loại hiện đại của ImageNet đầy thách thức.

- Rõ ràng, các kiến trúc mạng trong tài liệu học sâu đã sử dụng hỗn hợp các kích thước bộ lọc:

+ Lớp đầu tiên CNN thường bao gồm các kích thước bộ lọc nằm trong khoảng từ 7×7 [94] đến 11×11 [58]. Từ đó, kích thước bộ lọc giảm dần xuống còn 5×5 . Cuối cùng, chỉ có các lớp sâu nhất mạng sử dụng 3×3 bộ lọc.

+ VGGNet là duy nhất ở chỗ sử dụng 3×3 mặt nạ trong toàn bộ kiến trúc. Việc sử dụng các mặt nạ nhỏ này được cho là điều giúp VGGNet khai quát hóa các vấn đề phân loại bên ngoài mạng được huấn luyện ban đầu.

+ Bất cứ khi nào thấy một kiến trúc mạng bao gồm 3×3 bộ lọc, có thể yên tâm rằng được lấy cảm hứng từ VGGNet. Việc xem xét toàn bộ các biến thể lớp 16 và 19 VGGNet không nằm trong phần giới thiệu này. Để đánh giá chi tiết về VGG16 và VGG19, vui lòng tham khảo mục 2.1.

- Thay vào đó, xem xét họ mạng VGG và xác định các đặc điểm mà CNN phải thể hiện để phù hợp với họ này. Từ đó, triển khai một phiên bản nhỏ hơn VGGNet có tên MiniVGGNet có thể dễ dàng được huấn luyện trên hệ thống. Việc triển khai này cũng trình bày cách sử dụng hai lớp quan trọng đã thảo luận trong mục 2.1 - chuẩn hóa hàng loạt (BN) và bô học.

2.5.1. Họ mạng VGG

Họ mạng nơ-ron tích chập VGG có thể được đặc trưng bởi 2 thành phần chính:

1.Tất cả các lớp CONV trong mạng chỉ sử dụng bộ lọc 3×3 .

2. Xếp chồng nhiều bộ CONV => Tập hợp lớp RELU (trong đó số lớp CONV liên tiếp => Các lớp RELU thường tăng sâu hơn) trước khi áp dụng thao tác POOL.

Trong mục này, thảo luận về một biến thể kiến trúc VGGNet được gọi là MiniVGGNet do thực tế là mạng này đơn giản hơn đáng kể. Xem xét chi tiết và triển khai kiến trúc VGG ban đầu do Simonyan và Zisserman đề xuất, cùng với phần trình diễn về cách huấn luyện mạng trên tập dữ liệu ImageNet.

2.5.1.1. Kiến trúc VGGNET (MINI)

Trong cả ShallowNet và LeNet, đã áp dụng một loạt các lớp CONV => RELU => POOL. Tuy nhiên, trong VGGNet, xếp chồng nhiều lớp CONV => RELU trước khi áp dụng một lớp POOL duy nhất. Việc làm này cho phép mạng tìm hiểu các tính năng phong phú hơn từ các lớp CONV trước khi tải xuống kích thước đầu vào không gian thông qua hoạt động POOL.

Nhìn chung, MiniVGGNet bao gồm hai bộ CONV => RELU => CONV => RELU => POOL, tiếp theo là một tập hợp các lớp FC => RELU => FC => SOFTMAX. Hai lớp CONV đầu tiên học 32 bộ lọc, mỗi bộ có kích thước 3×3 . Hai lớp CONV thứ hai học 64 bộ lọc, mỗi bộ có kích

thước 3×3 . Các lớp POOL thực hiện gộp tối đa trên cửa sổ 2×2 với 2×2 bước trượt, cũng chèn các lớp chuẩn hóa hàng loạt sau khi kích hoạt cùng với các lớp bỏ (DO) sau các lớp POOL và FC.

- Bản thân kiến trúc mạng được trình bày chi tiết trong Bảng 2.5, trong đó kích thước ảnh đầu vào ban đầu được giả định là $32 \times 32 \times 3$ vì huấn luyện MiniVGGNet trên CIFAR-10 (sau đó so sánh hiệu suất với ShallowNet).

- Một lần nữa, lưu ý cách các lớp chuẩn hóa và bỏ học hàng loạt được bao gồm trong kiến trúc mạng dựa trên Định luật Ngón tay cái (Mục 2.1.3.2). Áp dụng chuẩn hóa hàng loạt giúp giảm tác động việc quá khớp và tăng độ chính xác phân loại trên CIFAR-10.

2.5.2. Triển khai MiniVGGNet

Đưa ra mô tả về MiniVGGNet trong Bảng 2.5, giờ đây có thể triển khai kiến trúc mạng bằng cách sử dụng Keras. Để bắt đầu, hãy thêm một tệp mới có tên minivggnet.py bên trong mô-đun con pyimagesearch.nn.conv - đó là nơi triển khai MiniVGGNet:

```
--- pyimagesearch
|   |--- __init__.py
|   |--- nn
|   |   |--- __init__.py
...
|   |   |--- conv
|   |   |   |--- __init__.py
|   |   |   |--- lenet.py
|   |   |   |--- minivggnet.py
|   |   |--- shallownet.py
```

Sau khi tạo tệp minivggnet.py, nhập các thư viện cần thiết:

```
1 # import the necessary packages
2 from keras.models import Sequential
3 from keras.layers.normalization import BatchNormalization
4 from keras.layers.convolutional import Conv2D
5 from keras.layers.convolutional import MaxPooling2D
6 from keras.layers.core import Activation
7 from keras.layers.core import Flatten
8 from keras.layers.core import Dropout
9 from keras.layers.core import Dense
10 from keras import backend as K
```

Bảng 2.5: Tóm tắt bảng về kiến trúc MiniVGGNet. Kích thước khối lượng đầu ra được bao gồm cho mỗi lớp, cùng với kích thước bộ lọc tích hợp / kích thước nhóm khi có liên quan. Lưu ý cách chỉ áp dụng 3×3 kết luận.

Layer Type	Output Size	Filter Size / Stride
INPUT IMAGE	$32 \times 32 \times 3$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
CONV	$32 \times 32 \times 32$	$3 \times 3, K = 32$
ACT	$32 \times 32 \times 32$	
BN	$32 \times 32 \times 32$	
POOL	$16 \times 16 \times 32$	2×2
DROPOUT	$16 \times 16 \times 32$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
CONV	$16 \times 16 \times 64$	$3 \times 3, K = 64$
ACT	$16 \times 16 \times 64$	
BN	$16 \times 16 \times 64$	
POOL	$8 \times 8 \times 64$	2×2
DROPOUT	$8 \times 8 \times 64$	
FC	512	
ACT	512	
BN	512	
DROPOUT	512	
FC	10	
SOFTMAX	10	

Dòng 2-10 nhập các lớp bắt buộc từ thư viện Keras. Hầu hết các thư viện đã thấy trước đây, nhưng cần chú ý đến BatchN normalization (Dòng 3) và Dropout (Dòng 8) - các lớp này cho phép áp dụng chuẩn hóa và bỏ học hàng loạt cho kiến trúc mạng.

Giống như việc triển khai cả ShallowNet và LeNet, định nghĩa một phương thức xây dựng có thể được gọi để xây dựng kiến trúc bằng cách sử dụng chiều rộng, chiều cao, chiều sâu và số lớp được cung cấp:

```

12 class MiniVGGNet:
13     @staticmethod
14     def build(width, height, depth, classes):
15         # initialize the model along with the input shape to be
16         # "channels last" and the channels dimension itself
17         model = Sequential()
18         inputShape = (height, width, depth)
19         chanDim = -1
20
21         # if we are using "channels first", update the input shape
22         # and channels dimension
23         if K.image_data_format() == "channels_first":
24             inputShape = (depth, height, width)
25             chanDim = 1

```

Dòng 17 khởi tạo lớp Sequential, khôi xây dựng các mạng nơ-ron tuần tự trong Keras. Sau đó, khởi tạo inputShape, giả sử đang sử dụng các kênh đặt hàng cuối cùng (Dòng 18).

Dòng 19 giới thiệu một biến mà đã thấy trước đây, chanDim, chỉ số thứ nguyên kênh. Chuẩn hóa hàng loạt hoạt động trên các kênh, vì vậy để áp dụng BN, cần biết trực tiếp nào chuẩn hóa. Đặt chanDim = -1 ngụ ý rằng chỉ mục kích thước kênh cuối cùng trong hình dạng đầu vào (nghĩa là, thứ tự các kênh cuối cùng). Tuy nhiên, nếu đang sử dụng thứ tự kênh đầu tiên (Dòng 23-25), cần cập nhật inputShape và đặt chanDim = 1, vì kích thước kênh hiện là mục nhập đầu tiên trong hình dạng đầu vào.

Khối lớp đầu tiên MiniVGGNet được xác định bên dưới:

```

27         # first CONV => RELU => CONV => RELU => POOL layer set
28         model.add(Conv2D(32, (3, 3), padding="same",
29                         input_shape=inputShape))
30         model.add(Activation("relu"))
31         model.add(BatchNormalization(axis=chanDim))
32         model.add(Conv2D(32, (3, 3), padding="same"))
33         model.add(Activation("relu"))
34         model.add(BatchNormalization(axis=chanDim))
35         model.add(MaxPooling2D(pool_size=(2, 2)))
36         model.add(Dropout(0.25))

```

- Ở đây, có thể thấy kiến trúc bao gồm (CONV => RELU => BN) * 2 => POOL => DO. Dòng 28 định nghĩa một lớp CONV với 32 bộ lọc, mỗi bộ lọc có kích thước bộ lọc 3×3 . Sau đó, áp dụng kích hoạt ReLU (Dòng 30) ngay lập tức được đưa vào lớp BatchN normalization (Dòng 31) để tập trung vào các kích hoạt.

- Tuy nhiên, thay vì áp dụng lớp POOL để giảm kích thước không gian đầu vào, áp dụng một bộ CONV => RELU => BN - điều này cho

phép mạng tìm hiểu các tính năng phong phú hơn, một cách phổ biến khi huấn luyện CNN sâu hơn.

- Trên Dòng 35, sử dụng MaxPooling2D với kích thước là 2×2 . Vì không đặt bước trượt một cách rõ ràng, Keras mặc nhiên cho rằng bước trượt bằng với kích thước gộp tối đa (là 2×2).

- Sau đó áp dụng Dropout trên Dòng 36 với xác suất $p = 0,25$, điều này ngũ ý rằng một nút từ lớp POOL ngắt kết nối ngẫu nhiên khỏi lớp tiếp theo với xác suất 25% trong quá trình huấn luyện. Áp dụng bỏ học để giúp giảm ảnh hưởng quá khớp. (Người đọc có thể đọc thêm về hiện tượng bỏ học trong mục 2.1.2.7). Sau đó thêm khỏi lớp thứ 2 vào MiniVGGNet bên dưới:

```
38      # second CONV => RELU => CONV => RELU => POOL layer set
39      model.add(Conv2D(64, (3, 3), padding="same"))
40      model.add(Activation("relu"))
41      model.add(BatchNormalization(axis=chanDim))
42      model.add(Conv2D(64, (3, 3), padding="same"))
43      model.add(Activation("relu"))
44      model.add(BatchNormalization(axis=chanDim))
45      model.add(MaxPooling2D(pool_size=(2, 2)))
46      model.add(Dropout(0.25))
```

Chương trình ở trên tuân theo cùng một mẫu chính xác như trên, tuy nhiên, hiện tại đang tìm hiểu hai bộ 64 bộ lọc trái ngược với 32 bộ lọc. Một lần nữa, thông thường là tăng số lượng bộ lọc khi kích thước đầu vào không gian giảm sâu hơn trong mạng.

Tiếp đến là tập hợp các lớp FC => RELU đầu tiên (và duy nhất):

Lớp FC có 512 nút, được sau bởi kích hoạt ReLU và BN. Cũng áp dụng bỏ học tại đây, tăng xác suất lên 50% - thông thường thấy bỏ học với $p = 0,5$ được áp dụng ở giữa các lớp FC.

Cuối cùng, áp dụng trình phân loại softmax và trả lại kiến trúc mạng cho hàm gọi:

```
48      # first (and only) set of FC => RELU layers
49      model.add(Flatten())
50      model.add(Dense(512))
51      model.add(Activation("relu"))
52      model.add(BatchNormalization())
53      model.add(Dropout(0.5))
```

Bây giờ, đã triển khai kiến trúc MiniVGGNetarch, hãy chuyển sang áp dụng cho bộ dữ liệu CIFAR-10.

2.5.3. MiniVGGNet trên CIFAR-10

Theo mô hình huấn luyện MiniVGGNet tương tự như đã làm cho LeNet trong mục 2.4, chỉ lần này với bộ dữ liệu CIFAR-10:

- Tải tập dữ liệu CIFAR-10 từ ổ cứng.
- Khởi tạo kiến trúc MiniVGGNet.
- Huấn luyện MiniVGGNet bằng cách sử dụng dữ liệu huấn luyện.
- Đánh giá hiệu suất mạng với dữ liệu thử nghiệm.

Để tạo tập lệnh điều khiển để huấn luyện MiniVGGNet, hãy mở một tệp mới, đặt tên là minivggnet_cifar10.py và chèn chương trình sau đây:

```
1 # set the matplotlib backend so figures can be saved in the background
2 import matplotlib
3 matplotlib.use("Agg")
4
5 # import the necessary packages
6 from sklearn.preprocessing import LabelBinarizer
7 from sklearn.metrics import classification_report
8 from pyimagesearch.nn.conv import MiniVGGNet
9 from keras.optimizers import SGD
10 from keras.datasets import cifar10
11 import matplotlib.pyplot as plt
12 import numpy as np
13 import argparse
```

Dòng 2 nhập thư viện matplotlib mà sau đó sử dụng để vẽ chính xác và tồn thắt theo thời gian. Cần đặt phụ trợ matplotlib thành Agg để chỉ ra phần không tương tác để tiết kiệm tài nguyên. Tùy thuộc vào phụ trợ matplotlib mặc định là gì và có đang truy cập vào máy học sâu từ xa không (through qua SSH), nếu điều đó xảy ra, matplotlib báo lỗi khi có hiển thị hình. Thay vào đó, có thể chỉ cần đặt nền là Agg và ghi quá trình huấn luyện vào ổ cứng khi hoàn thành việc huấn luyện mạng.

Các dòng 9-13 nhập phần còn lại các gói Python cần thiết, tất cả các gói đã thấy trước đây - ngoại lệ là MiniVGGNet trên Dòng 11 đã triển khai trong phần trước.

Tiếp theo, phân tích cú pháp đối số dòng lệnh:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-o", "--output", required=True,
18                 help="path to the output loss/accuracy plot")
19 args = vars(ap.parse_args())
```

Đoạn chương trình này chỉ yêu cầu một đối số dòng lệnh duy nhất, --output, đường dẫn đến biểu đồ mất và huấn luyện đầu ra.

Bây giờ có thể tải bộ dữ liệu CIFAR-10 (được chia trước thành dữ liệu huấn luyện và thử nghiệm), chia tỷ lệ các pixel thành phạm vi [0, 1], sau đó mã hóa một nhãn:

```
21 # load the training and testing data, then scale it into the
22 # range [0, 1]
23 print("[INFO] loading CIFAR-10 data...")
24 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
25 trainX = trainX.astype("float") / 255.0
26 testX = testX.astype("float") / 255.0
27
28 # convert the labels from integers to vectors
29 lb = LabelBinarizer()
30 trainY = lb.fit_transform(trainY)
31 testY = lb.transform(testY)
32
33 # initialize the label names for the CIFAR-10 dataset
34 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
35 "dog", "frog", "horse", "ship", "truck"]
```

Biên soạn mô hình và bắt đầu huấn luyện MiniVGGNet:

```
37 # initialize the optimizer and model
38 print("[INFO] compiling model...")
39 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
40 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
41 model.compile(loss="categorical_crossentropy", optimizer=opt,
42 metrics=["accuracy"])
43
44 # train the network
45 print("[INFO] training network...")
46 H = model.fit(trainX, trainY, validation_data=(testX, testY),
47 batch_size=64, epochs=40, verbose=1)
```

Sử dụng SGD làm trình tối ưu hóa với tỷ lệ học là $\alpha = 0,1$ và thời hạn động lượng là $\gamma = 0,9$. Đặt Nesterov = True chỉ ra rằng muốn áp dụng gradient tăng tốc Nesterov cho trình tối ưu hóa SGD (Mục 3.2).

Một thuật ngữ tối ưu hóa đã thấy là một thông số phân rã. Lập luận này được sử dụng để từ từ giảm tốc độ học theo thời gian. Chương tiếp theo thảo luận về Trình lập lịch biểu tỷ lệ học cho thấy phân rã tỷ lệ học rất hữu ích trong việc giảm quá khóp và đạt được độ chính xác phân loại cao hơn - tỷ lệ học càng nhỏ, cập nhật trọng số càng nhỏ. Một cài đặt chung cho phân rã là chia tỷ lệ học ban đầu cho tổng số chu kỳ - trong trường hợp này, huấn luyện mang tổng cộng 40 chu kỳ với tỷ lệ học ban đầu là 0,01, do đó phân rã = $0,01 / 40$.

Sau khi huấn luyện hoàn tất, có thể đánh giá mạng và hiển thị báo cáo phân loại được định dạng như sau:

```

49 # evaluate the network
50 print("[INFO] evaluating network...")
51 predictions = model.predict(testX, batch_size=64)
52 print(classification_report(testY.argmax(axis=1),
53     predictions.argmax(axis=1), target_names=labelNames))

```

Với lưu đồ thị tồn thât và độ chính xác:

```

55 # plot the training loss and accuracy
56 plt.style.use("ggplot")
57 plt.figure()
58 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
59 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
60 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
61 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
62 plt.title("Training Loss and Accuracy on CIFAR-10")
63 plt.xlabel("Epoch #")
64 plt.ylabel("Loss/Accuracy")
65 plt.legend()
66 plt.savefig(args["output"])

```

Khi đánh giá MiniVGGNet, đã thực hiện hai thực nghiệm:

1. Chuẩn hóa hàng loạt.
2. Không chuẩn hóa hàng loạt.

Hãy cùng đi trước và xem xét các kết quả này để so sánh hiệu suất mạng tăng lên khi áp dụng chuẩn hóa hàng loạt.

2.5.3.1. Chuẩn hóa hàng loạt

Để huấn luyện MiniVGGNet trên bộ dữ liệu CIFAR-10, chỉ cần thực hiện lệnh sau:

```

$python minivggnet cifar10.py --output output/cifar10_minivggnet with bn.png
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
23s - loss: 1.6001 - acc: 0.4691 - val loss: 1.3851 - val acc: 0.5234
Epoch 2/40
23s - loss: 1.1237 - acc: 0.6079 - val loss: 1.1925 - val acc: 0.6139
Epoch 3/40
23s - loss: 0.9680 - acc: 0.6610 - val loss: 0.8761 - val acc: 0.6909
...
Epoch 40/40
23s - loss: 0.2557 - acc: 0.9087 - val loss: 0.5634 - val acc: 0.8236
[INFO] evaluating network...
      precision    recall   f1-score   support
airplane       0.88      0.81      0.85      1000
automobile     0.93      0.89      0.91      1000
bird          0.83      0.68      0.75      1000
cat           0.69      0.65      0.67      1000
deer          0.74      0.85      0.79      1000
dog           0.72      0.77      0.74      1000
frog          0.85      0.89      0.87      1000
horse         0.85      0.87      0.86      1000
ship          0.89      0.91      0.90      1000
truck         0.88      0.91      0.90      1000
avg / total    0.83      0.82      0.82      10000

```

Trên GPU, chu kỳ khá nhanh ở giai đoạn thứ 23. Trên CPU, các chu kỳ dài hơn đáng kể, với tốc độ 171 giây.

Sau khi hoàn thành huấn luyện, có thể thấy MiniVGGNet đang đạt được độ chính xác phân loại 83% trên bộ dữ liệu CIFAR-10 với chuẩn hóa hàng loạt - kết quả này cao hơn đáng kể so với độ chính xác 60% khi áp dụng ShallowNet trong mục 3.2. Do đó, có thể thấy cách kiến trúc mạng sâu hơn có thể học các tính năng phong phú hơn, phân biệt hơn.

Nhưng vai trò bình thường hóa hàng loạt là gì? Có thực sự giúp ở đây? Để tìm hiểu, di chuyển sang phần tiếp theo.

2.5.3.2. Không có chuẩn hóa hàng loạt

Quay trở lại việc triển khai MiniVGGNet.py và nhận xét tất cả các lớp BatchNormalization, như vậy:

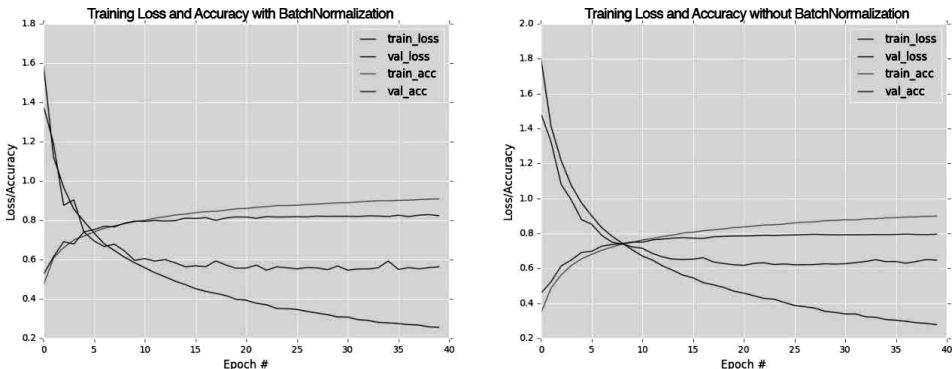
```
27      # first CONV => RELU => CONV => RELU => POOL layer set
28      model.add(Conv2D(32, (3, 3), padding="same",
29                      input_shape=inputShape))
30      model.add(Activation("relu"))
31      #model.add(BatchNormalization(axis=chanDim))
32      model.add(Conv2D(32, (3, 3), padding="same"))
33      model.add(Activation("relu"))
34      #model.add(BatchNormalization(axis=chanDim))
35      model.add(MaxPooling2D(pool_size=(2, 2)))
36      model.add(Dropout(0.25))
```

Khi đã nhận xét tất cả các lớp BatchN normalization từ mạng, hãy huấn luyện lại MiniVGGNet trên CIFAR-10:

```
$python minivggnet cifar10.py --output output/cifar10_minivggnet without bn.png
[INFO] loading CIFAR-10 data...
[INFO] kompling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
13s - loss: 1.8055 - acc: 0.3426 - val loss: 1.4872 - val acc: 0.4573
Epoch 2/40
13s - loss: 1.4133 - acc: 0.4872 - val_loss: 1.3246 - val_acc: 0.5224
Epoch 3/40
13s - loss: 1.2162 - acc: 0.5628 - val_loss: 1.0807 - val_acc: 0.6139
...
Epoch 40/40
13s - loss: 0.2780 - acc: 0.8996 - val loss: 0.6466 - val acc: 0.7955
[INFO] evaluating network...
      precision    recall   f1-score   support
airplane       0.83      0.80      0.82      1000
automobile     0.90      0.89      0.90      1000
bird          0.75      0.69      0.71      1000
cat           0.64      0.57      0.61      1000
deer          0.75      0.81      0.78      1000
dog           0.69      0.72      0.70      1000
frog          0.81      0.88      0.85      1000
horse         0.85      0.83      0.84      1000
ship          0.90      0.88      0.89      1000
truck         0.84      0.89      0.86      1000
avg / total    0.79      0.80      0.79     10000
```

Điều đầu tiên nhận thấy là mạng huấn luyện nhanh hơn mà không cần chuẩn hóa hàng loạt (13 giây so với 23, giảm 43%). Tuy nhiên, khi mạng hoàn thành huấn luyện, nhận thấy độ chính xác phân loại thấp hơn là 79%.

Khi biểu diễn MiniVGGNet với chuẩn hóa hàng loạt (trái) và không có chuẩn hóa hàng loạt (phải) song song trong Hình 2.19, có thể thấy sự bình thường hóa ảnh hưởng tích cực đến quá trình huấn luyện:



Hình 2.19: Trái: MiniVGGNet được huấn luyện về CIFAR-10 với chuẩn hóa hàng loạt. Phải: MiniVGGNet được huấn luyện về CIFAR-10 mà không cần chuẩn hóa hàng loạt. Áp dụng chuẩn hóa hàng loạt cho phép có được độ chính xác phân loại cao hơn và giảm ảnh hưởng quá khứ.

Lưu ý rằng tồn thắt cho MiniVGGNet khi không chuẩn hóa hàng loạt bắt đầu tăng qua chu kỳ 30, cho thấy mạng đang quá tải dữ liệu huấn luyện. cũng có thể thấy rõ ràng độ chính xác nhận đã bão hòa trong chu kỳ 25.

Mặt khác, việc triển khai MiniVGGNet với chuẩn hóa hàng loạt ổn định hơn. Mặc dù cả tồn thắt và độ chính xác đều bắt đầu đi ngang qua chu kỳ 35, nhưng lại quá tệ - đây là một trong nhiều lý do tại sao khuyên nên áp dụng chuẩn hóa hàng loạt cho các kiến trúc mạng riêng.

2.5.4. Tóm tắt

Trong mục 2.5, đã thảo luận về họ mạng nơ-ron tích chập VGG. Một CNN có thể được coi là VGG-net như sau:

1. Chỉ sử dụng các bộ lọc 3×3 , bắt kề độ sâu mạng.
2. Có nhiều lớp CONV \Rightarrow RELU được áp dụng trước một thao tác POOL duy nhất, đôi khi có nhiều lớp CONV \Rightarrow RELU xếp chồng lên nhau khi mạng tăng chiều sâu.

Sau đó, đã triển khai một mạng từ VGG, được đặt tên phù hợp là MiniVGGNet. Kiến trúc mạng này bao gồm hai bộ (CONV => RELU) * 2) => Các lớp POOL theo sau là một lớp FC.

=> RELU => FC => Tập hợp lớp SOFTMAX. cũng áp dụng chuẩn hóa hàng loạt sau mỗi lần kích hoạt cũng như bỏ học sau mỗi nhóm và lớp được kết nối đầy đủ. Để đánh giá MiniVGGNet, đã sử dụng bộ dữ liệu CIFAR-10.

Độ chính xác tốt nhất trước đây trên CIFAR-10 chỉ bằng 60% từ mạng ShallowNet (Mục 3.2). Tuy nhiên, bằng cách sử dụng MiniVGGNet, có thể tăng độ chính xác tới 83%.

Cuối cùng, đã kiểm tra vai trò chuẩn hóa hàng loạt trong học sâu và CNN. Với chuẩn hóa hàng loạt, MiniVGGNet đạt độ chính xác phân loại 83% - nhưng nếu không có chuẩn hóa hàng loạt, độ chính xác giảm xuống 79% (và cũng bắt đầu thấy dấu hiệu của hiện tượng quá khớp).

Vì vậy, điều đáng nói ở đây là:

1. Chuẩn hóa hàng loạt có thể dẫn đến sự hội tụ nhanh hơn, ổn định hơn với độ chính xác cao hơn.

2. Tuy nhiên, những lợi thế có được nhờ đánh đổi thời gian huấn luyện - việc chuẩn hóa hàng loạt đòi hỏi nhiều thời gian hơn để huấn luyện mạng. Bù lại mạng sẽ có độ chính xác cao hơn trong ít chu kỳ hơn.

Điều đó nói lên rằng, thời gian huấn luyện phải thêm vào thường vượt xa điểm tiêu cực và rất khuyến khích áp dụng chuẩn hóa hàng loạt cho các kiến trúc mạng.

Chương 3: CƠ SỞ TOÁN HỌC CỦA MẠNG NƠ-RON

3.1. HỌC THEO THÔNG SỐ

Chúng ta sẽ tìm hiểu về trình phân loại k-NN - một mô hình học máy đơn giản đến nỗi nó không thực hiện bất kỳ cách học thực tế nào. Chỉ đơn giản là phải lưu trữ dữ liệu huấn luyện bên trong mô hình và sau đó dự đoán được đưa ra tại thời điểm kiểm tra bằng cách so sánh các điểm dữ liệu kiểm tra với dữ liệu huấn luyện.

Ưu và nhược điểm của k-NN đã được xem xét nhiều, nhưng đối với các bộ dữ liệu quy mô lớn và học sâu, yếu điểm của k-NN là chính dữ liệu. Mặc dù huấn luyện có thể đơn giản, nhưng quá trình kiểm tra khá chậm, với nút thắt là việc tính toán khoảng cách giữa các vecto. Việc tính toán khoảng cách giữa các điểm huấn luyện và kiểm tra sẽ chia tỷ lệ tuyến tính với số điểm trong tập dữ liệu, làm cho phương thức không thực tế khi các bộ dữ liệu trở nên khá lớn. Trong khi có thể áp dụng các phương pháp lân cận gần nhất (Approximate Nearest Neighbor) như ANN [59], FLANN [60] hoặc Annoy [61] để tăng tốc tìm kiếm, thì điều đó vẫn không làm k-NN ngừng hoạt động mà không duy trì bản sao dữ liệu khi khởi tạo.

Để xem tại sao lưu trữ một bản sao chính xác dữ liệu huấn luyện trong mô hình là một vấn đề, xem xét huấn luyện mô hình k-NN và sau đó triển khai đến cơ sở khách hàng 100, 1000 hoặc thậm chí 1.000.000 người dùng. Nếu tập huấn luyện chỉ có vài megabyte, thì đây có thể không phải là vấn đề - nhưng nếu tập huấn luyện được đo bằng gigabyte đến terabyte (như trường hợp nhiều bộ dữ liệu mà áp dụng học sâu) thì sao?

Xem xét tập huấn luyện bộ dữ liệu ImageNet bao gồm hơn 1,2 triệu ảnh. Nếu đã huấn luyện mô hình k-NN trên bộ dữ liệu này và sau đó cố gắng triển khai nó cho một nhóm người dùng, sẽ cần những người dùng này tải xuống mô hình k-NN đại diện cho bản sao 1,2 triệu ảnh. Tùy thuộc vào cách nén và lưu trữ dữ liệu, mô hình này có thể đo bằng hàng trăm gigabyte đến terabyte trong chi phí lưu trữ và chi phí mạng. Điều này không chỉ gây lãng phí tài nguyên mà còn không tối ưu để xây dựng mô hình học máy.

Thay vào đó, một cách tiếp cận hấp dẫn hơn sẽ là xác định mô hình học máy có thể học các mẫu từ dữ liệu đầu vào trong thời gian huấn luyện (yêu cầu dành nhiều thời gian hơn cho quá trình huấn luyện), nhưng lợi ích

có được xác định bởi một số lượng nhỏ các tham số có thể dễ dàng được sử dụng để đại diện cho mô hình, bất kể quy mô huấn luyện. Kiểu học máy này được gọi là học tham số.

Trong mục này, sẽ tìm hiểu khái niệm học tham số và xem xét về cách thực hiện một trình phân loại tuyến tính đơn giản. Như sẽ thấy sau trong cuốn sách này, học tham số là nền tảng các thuật toán học máy hiện đại và học sâu.

3.1.1. Giới thiệu về phân loại tuyến tính

Nửa đầu mục này tập trung vào lý thuyết cơ bản và toán học xung quanh phân loại tuyến tính - và nói chung, các thuật toán phân loại tham số học các mẫu từ dữ liệu huấn luyện. Từ đó, cung cấp một triển khai phân loại tuyến tính thực tế và ví dụ trong Python để có thể thấy các loại thuật toán này hoạt động như thế nào.

3.1.1.1. Bốn thành phần học có tham số

“Tham số” chính xác có nghĩa là gì?

Nói một cách đơn giản: tham số hóa là quá trình xác định các tham số cần thiết của một mô hình nhất định. Trong nhiệm vụ học máy, tham số hóa bao gồm xác định một vấn đề theo bốn thành phần chính: dữ liệu, hàm tính điểm, hàm tổn thất và trọng số và sai lệch. Sẽ xem xét từng thành phần dưới đây.

Dữ liệu

Đây là dữ liệu đầu vào được học. Dữ liệu này bao gồm cả các điểm dữ liệu (nghĩa là, cường độ pixel thô từ ảnh, đặc trưng được trích xuất, v.v.) và lớp nhãn liên quan đến chúng. Thông thường, biểu thị dữ liệu theo ma trận thiết kế đa chiều [33].

Mỗi hàng trong ma trận thiết kế đại diện cho một điểm dữ liệu trong khi mỗi cột (có thể là một mảng nhiều chiều) ma trận tương ứng với một đặc trưng khác nhau. Ví dụ, xem xét bộ dữ liệu gồm 100 ảnh trong không gian màu RGB, mỗi ảnh có kích thước 32×32 pixel. Ma trận thiết kế cho bộ dữ liệu này sẽ là $X \subseteq R^{100 \times (32 \times 32 \times 3)}$ trong đó X_i xác định ảnh thứ i trong R . Sử dụng ký hiệu này, X_1 là ảnh đầu tiên, X_2 là ảnh thứ hai, v.v.

Cùng với ma trận thiết kế, cũng định nghĩa một vectơ y trong đó y_i cung cấp lớp nhãn cho ví dụ thứ i trong tập dữ liệu.

Hàm tính điểm (Scoring Function): Hàm tính điểm chấp nhận dữ liệu làm đầu vào và ánh xạ dữ liệu tới lớp nhãn. Chẳng hạn, với tập hợp

các ảnh đầu vào, hàm tính điểm sẽ lấy các điểm dữ liệu này, áp dụng một số hàm f (hàm tính điểm) và sau đó trả về các lớp nhãn dự đoán, tương tự như mã giả bên dưới:

INPUT_IMAGES => $F(\text{INPUT_IMAGES}) \Rightarrow \text{OUTPUT_CLASS_LABELS}$

Hàm tổn thất (Loss function)

Hàm tổn thất định lượng mức độ nhãn dự đoán đúng với nhãn thực tế. Mức độ tương đồng giữa hai bộ nhãn này càng cao, sai số càng thấp (và độ chính xác phân loại càng cao, ít nhất là trên tập huấn luyện).

Mục tiêu khi huấn luyện một mô hình học máy là để giảm thiểu hàm tổn thất, từ đó tăng độ chính xác phân loại.

Trọng số và độ lệch (Weigh and bias)

Mã trận trọng số, thường được ký hiệu là W và vectơ sai lệch b được gọi là trọng số hoặc tham số phân loại thực sự sẽ tối ưu hóa. Dựa trên điều ra hàm tính điểm và hàm tổn thất, sẽ điều chỉnh và thay đổi các giá trị trọng số và độ lệch để tăng độ chính xác phân loại.

Tùy thuộc vào loại mô hình, có thể tồn tại nhiều tham số hơn, nhưng ở cấp độ cơ bản nhất, đây là bốn khái niệm xây dựng việc học được tham số hóa có thể gặp phải. Khi xác định bốn thành phần chính này, có thể áp dụng các phương pháp tối ưu hóa cho phép tìm một tập hợp các tham số W và b để giảm thiểu hàm tổn thất đối với hàm tính điểm (trong khi tăng độ chính xác phân loại trên dữ liệu).

Tiếp theo, cùng xem xét cách các thành phần này có thể phối hợp với nhau để xây dựng bộ phân loại tuyến tính, biến đổi dữ liệu đầu vào thành dự đoán thực tế

3.1.1.2. Phân loại tuyến tính: từ ảnh đến nhãn

Trong phần này, sẽ xem xét một động lực toán học hơn phương pháp mô hình được tham số hóa để học máy.

Để bắt đầu, cần có dữ liệu. Giả sử rằng tập dữ liệu huấn luyện được ký hiệu là x_i , trong đó mỗi ảnh có lớp nhãn liên quan y_i . sẽ giả sử rằng $i = 1, \dots, N$ và $y_i = 1, \dots, K$, nghĩa là có N điểm dữ liệu có thứ nguyên D , được tách thành K loại duy nhất.

Để làm cho ý tưởng này cụ thể hơn, xem xét bộ dữ liệu động vật từ mục 5.5. Trong bộ dữ liệu này, có tổng số $N = 3000$ ảnh. Mỗi ảnh là 32×32 pixel, được biểu thị trong không gian màu RGB (nghĩa là ba kênh trên mỗi ảnh). Có thể biểu diễn mỗi ảnh là $D = 32 \times 32 \times 3 = 3,072$ giá trị riêng biệt.

Cuối cùng, biết có tổng cộng $K = 3$ lớp nhãn: một nhãn cho các lớp chó, mèo và gấu trúc tương ứng.

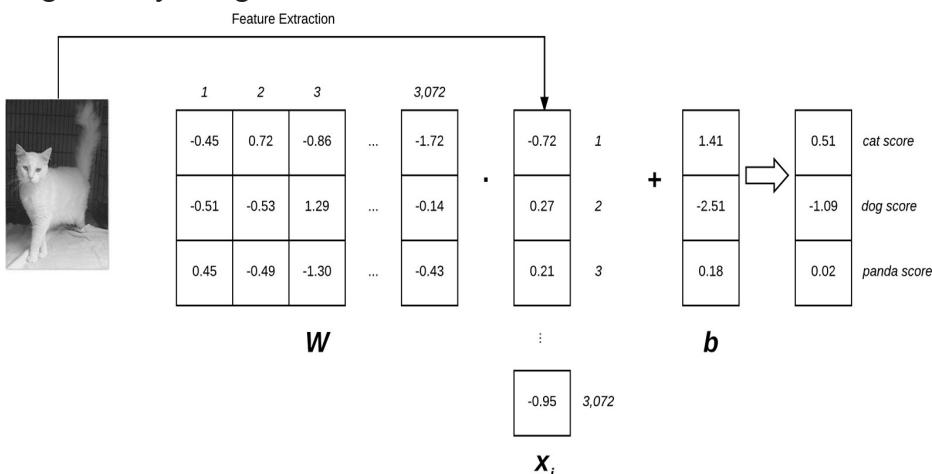
Với các biến này, bây giờ phải xác định hàm tính điểm f để ánh xạ các ảnh tới điểm lớp nhãn. Một phương pháp để thực hiện việc chấm điểm này là thông qua ánh xạ tuyến tính đơn giản:

$$f(x_i, W, b) = Wx_i + b \quad (3.1)$$

giả sử rằng mỗi x_i được biểu diễn dưới dạng một vectơ cột đơn có hình dạng $[D \times 1]$ (trong ví dụ này, sẽ làm phẳng các ảnh $32 \times 32 \times 3$ thành một danh sách 3.072 số nguyên). Ma trận trọng số W sau đó sẽ có hình dạng $[K \times D]$ (số lượng lớp nhãn theo chiều ảnh đầu vào). Cuối cùng b , vectơ sai lệch sẽ có kích thước $[K \times 1]$. Vectơ sai lệch cho phép dịch chuyển và dịch hàm tính điểm theo hướng này hay hướng khác mà không thực sự ảnh hưởng đến ma trận trọng số W . Các tham số sai lệch thường rất quan trọng cho việc huấn luyện.

Quay trở lại ví dụ về tập dữ liệu động vật, mỗi x_i được biểu thi bằng một danh sách 3.072 giá trị pixel, do đó, x_i có hình dạng $[3,072 \times 1]$. Ma trận trọng số W sẽ có dạng $[3 \times 3,072]$ và cuối cùng, vectơ sai lệch b sẽ có kích thước $[3 \times 1]$.

Hình 3.1 là một minh họa về hàm tính điểm phân loại tuyến tính f . Ở bên trái, có ảnh đầu vào ban đầu, được biểu thị dưới dạng ảnh $32 \times 32 \times 3$. Sau đó, làm phẳng ảnh này thành một danh sách cường độ 3.072 pixel bằng cách lấy mảng 3D và định hình lại nó thành danh sách 1D.



Hình 3.1: Minh họa phép nhân ma trận trọng số W và vectơ đặc trưng x , tiếp theo là bổ sung thuật ngữ sai lệch. Hình ảnh được tham khảo từ ví dụ của Karpathy trong khóa học cs231n Đại học Stanford [62].

Ma trận trọng số W chứa ba hàng (một cho mỗi lớp nhẫn) và 3.072 cột (một cho mỗi pixel trong ảnh). Sau khi nhân chập giữa W và x_i , thêm vào vectơ sai lệch b - kết quả là điểm thực tế. Điểm mang lại ba giá trị ở bên phải: điểm số tương ứng với nhẫn chó, mèo và gấu trúc.

Nhìn vào hình và phương trình trên, có thể hiểu rằng x_i và y_i đầu vào là cố định và không phải để sửa đổi. Chắc chắn, có thể thu được các x_i khác nhau bằng cách áp dụng các biến đổi khác nhau cho ảnh đầu vào - nhưng một khi biến đổi ảnh thành điểm, các giá trị này không thay đổi. Trong thực tế, các tham số duy nhất (về mặt học được tham số) là ma trận trọng số W và vectơ sai lệch b. Do đó, mục tiêu là sử dụng hàm tính điểm và hàm tổn thất để tối ưu hóa (nghĩa là sửa đổi một cách có hệ thống) các vectơ trọng số và sai lệch sao cho độ chính xác phân loại tăng lên.

Chính xác làm thế nào tối ưu hóa ma trận trọng số phụ thuộc vào hàm tổn thất, nhưng thường liên quan đến một số hình thức giảm độ dốc, sẽ xem xét các hàm tổn thất trong mục này. Các phương pháp tối ưu hóa như gradient descent (và các biến thể) sẽ được xem xét trong mục 3.2. Tuy nhiên, hiện tại, có thể hiểu đơn giản là được cung cấp hàm tính điểm, cũng sẽ xác định hàm tổn thất cho biết việc dự đoán tốt như thế nào với dữ liệu đầu vào.

3.1.1.3. *Ưu điểm việc học theo tham số và phân loại tuyến tính*

Có hai lợi thế chính để sử dụng học tham số:

1. Khi hoàn thành huấn luyện mô hình, có thể loại bỏ dữ liệu đầu vào và chỉ giữ lại ma trận trọng số W và vectơ sai lệch b. Điều này làm giảm đáng kể kích thước mô hình vì cần lưu trữ hai bộ vectơ (so với toàn bộ tập huấn luyện).

2. Phân loại dữ liệu kiểm tra mới một cách nhanh chóng. Để thực hiện phân loại, tất cả những gì cần làm là lấy phép nhân W và x_i , tiếp theo bằng cách thêm vào độ lệch b (nghĩa là áp dụng hàm tính điểm). Làm theo cách này nhanh hơn đáng kể so với việc cần so sánh từng điểm kiểm tra với mọi ví dụ huấn luyện, như trong thuật toán k-NN.

3.1.1.4. *Ứng dụng phân loại tuyến tính đơn giản với ngôn ngữ lập trình Python*

Bây giờ đã xem xét lại khái niệm huấn luyện và phân loại tuyến tính để thực hiện một chương trình phân loại tuyến tính đơn giản bằng Python.

Mục đích của ví dụ này không phải là để chứng minh cách huấn luyện một mô hình từ đầu đến cuối (sẽ đề cập đến điều đó trong mục sau),

nhưng chỉ đơn giản là chỉ ra cách khởi tạo ma trận trọng số W, vectơ sai lệch b, sau đó sử dụng các tham số này để phân loại ảnh thông qua một phép nhân chập đơn giản

Để bắt đầu ví dụ này, mục tiêu ở đây là viết một tập lệnh Python sẽ phân loại chính xác Hình 3.2 là con chó.



Hình 3.2: Ảnh đầu vào của ví dụ về một bộ phân loại tuyến tính đơn giản.

Để xem làm thế nào có thể thực hiện phân loại này, mở một tệp mới, đặt tên là `dongar_example.py`, và chèn đoạn chương trình sau:

```
1 # import the necessary packages
2 import numpy as np
3 import cv2
4
5 # initialize the class labels and set the seed of the pseudorandom
6 # number generator so we can reproduce our results
7 labels = ["dog", "cat", "panda"]
8 np.random.seed(1)
```

Dòng 2 và 3 nhập các gói Python cần thiết. Sẽ sử dụng NumPy để xử lý số và OpenCV để tải ảnh ví dụ từ ổ cứng.

Dòng 7 khởi tạo danh sách các lớp nhãn mục tiêu cho bộ dữ liệu. Dòng 8 trên động vật trong khi Dòng 8 thiết lập trình tạo số giả cho NumPy, đảm bảo rằng có thể sao chép kết quả kiểm tra này.

Tiếp theo, để khởi tạo ma trận trọng số và vectơ sai lệch

```
10 # randomly initialize our weight matrix and bias vector -- in a
11 # *real* training and classification task, these parameters would
12 # be *learned* by our model, but for the sake of this example,
13 # let's use random values
14 W = np.random.randn(3, 3072)
15 b = np.random.randn(3)
```

Dòng 14 khởi tạo ma trận trọng số W với các giá trị ngẫu nhiên từ phân chia đồng đều, được lấy mẫu trong phạm vi $[0, 1]$. Ma trận trọng số này có 3 hàng (một cho mỗi lớp nhãn) và 3072 cột (một cho mỗi pixel trong ảnh $32 \times 32 \times 3$).

Sau đó, khởi tạo vectơ sai lệch trên Dòng 15 - vectơ này cũng được điền ngẫu nhiên với các giá trị được lấy mẫu thống nhất trên phân phối $[0, 1]$. Vectơ sai lệch có 3 hàng (tương ứng với số lượng lớp nhãn) cùng với một cột.

Nếu huấn luyện trình phân loại tuyến tính này từ đầu, sẽ cần tìm hiểu các giá trị W và b thông qua quy trình tối ưu hóa. Tuy nhiên, vì chưa đạt đến giai đoạn tối ưu hóa huấn luyện một mô hình, đã khởi tạo trình tạo số giả ngẫu nhiên với giá trị 1 để đảm bảo các giá trị ngẫu nhiên cung cấp cho phân loại chính xác (đã kiểm tra các giá trị khởi tạo ngẫu nhiên trước thời hạn để xác định giá trị cho phân loại chính xác). Hiện tại, chỉ cần coi ma trận trọng số W và vectơ sai lệch b là “mảng hộp đen” được tối ưu hóa - cách các tham số này được học như thế nào trình bày trong mục tiếp theo.

Bây giờ, ma trận trọng số và vectơ sai lệch đã được khởi tạo, để tải ảnh ví dụ từ ổ cứng:

```
17 # load our example image, resize it, and then flatten it into our
18 # "feature vector" representation
19 orig = cv2.imread("beagle.png")
20 image = cv2.resize(orig, (32, 32)).flatten()
```

Dòng 19 tải ảnh từ ổ cứng thông qua cv2.imread. Sau đó, thay đổi kích thước ảnh thành 32×32 pixel (bỏ qua tỷ lệ khung hình) trên Dòng 20 - ảnh hiện được thể hiện dưới dạng một mảng NumPy $(32, 32, 3)$, làm phẳng thành một vector 3.072

Bước tiếp theo là tính toán điểm lớp nhãn đầu ra bằng cách áp dụng hàm tính điểm:

```
22 # compute the output scores by taking the dot product between the
23 # weight matrix and image pixels, followed by adding in the bias
24 scores = W.dot(image) + b
```

Dòng 24 là chính hàm tính điểm - nó đơn giản là phép nhân chấm giữa ma trận trọng số W và cường độ pixel ảnh đầu vào, tiếp theo là thêm vào độ lệch b.

Cuối cùng, khởi động cuối cùng xử lý việc viết các giá trị hàm cho mỗi lớp nhãn vào thiết bị đầu cuối, sau đó hiển thị kết quả ra màn hình.

```

26 # loop over the scores + labels and display them
27 for (label, score) in zip(labels, scores):
28     print("[INFO] {}: {:.2f}".format(label, score))
29
30 # draw the label with the highest score on the image as our
31 # prediction
32 cv2.putText(orig, "Label: {}".format(labels[np.argmax(scores)]),
33             (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.9, (0, 255, 0), 2)
34
35 # display our input image
36 cv2.imshow("Image", orig)
37 cv2.waitKey(0)

```

Để thực hiện ví dụ, chỉ cần gõ lệnh sau:

```

$ python linear_example.py
[INFO] dog: 7963.93
[INFO] cat: -2930.99
[INFO] panda: 3362.47

```

Lưu ý cách lớp Chó có giá trị đánh giá điểm lớn nhất, hàm ý rằng lớp Chó sẽ được chọn làm dự đoán phân loại. Trong thực tế, có thể thấy con chó thể hiện chính xác trên ảnh đầu vào (Hình 3.2) trong Hình 3.3.



Hình 3.3: Trong ví dụ này, bộ phân loại tuyến tính có thể gắn nhãn chính xác cho ảnh đầu vào là một con chó. Phần sau trong cuốn sách này, sẽ học cách huấn luyện các trọng số để tự động đưa ra những dự đoán này.

Trong thực tế, sẽ không bao giờ khởi tạo các giá trị W và b và cho rằng chúng sẽ cung cấp cho phân loại chính xác mà không cần thông qua một quy trình. Thay vào đó, khi huấn luyện các mô hình học máy riêng từ đầu, sẽ cần tối ưu hóa và học W và b thông qua thuật toán tối ưu hóa, chẳng hạn như giảm độ dốc.

Để cập đến tối ưu hóa và giảm độ dốc trong mục tiếp theo. Cách phân loại tuyến tính thực hiện phân loại bằng cách lấy nhân chập giữa ma

trận trọng số và điểm dữ liệu đầu vào, tiếp theo là thêm vào sự sai lệch. Do đó, toàn bộ mô hình có thể được xác định thông qua hai giá trị: ma trận trọng số và vectơ sai lệch.

3.1.2. Vai trò của các hàm tổn thất

Trong phần cuối cùng, xem xét về khái niệm học tham số. Kiểu học này cho phép lấy các bộ dữ liệu đầu vào và lớp nhãn và thực sự học một hàm ánh xạ đầu vào tới các dự đoán đầu ra bằng cách xác định một bộ tham số và tối ưu hóa chúng.

Nhưng để thực sự học hỏi việc ánh xạ từ dữ liệu đầu vào sang lớp nhãn thông qua hàm tính điểm, cần xem xét về hai khái niệm quan trọng:

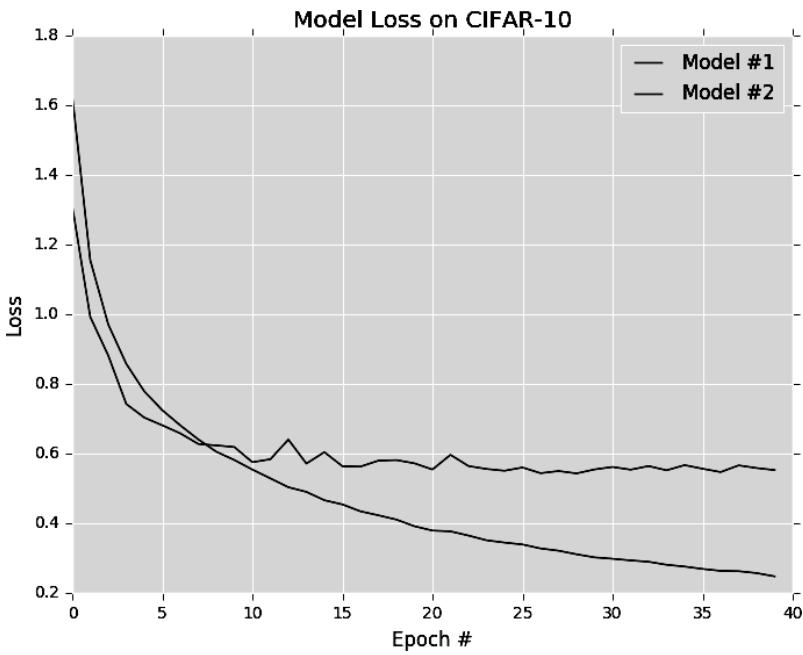
1. Hàm tổn thất
2. Phương pháp tối ưu hóa

Phần còn lại mục này dành riêng cho các hàm tổn thất phổ biến sẽ gặp khi xây dựng mạng nơ-ron và mạng học sâu. Mục 3.2 tìm hiểu các phương pháp tối ưu hóa

Một lần nữa, mục này là một đánh giá ngắn gọn về các hàm tổn thất và vai trò chúng trong học tham số. Một xem xét kỹ lưỡng về các hàm tổn thất nằm ngoài phạm vi cuốn sách này, bạn đọc có thể tham khảo trong Andrew Ng - Coursera [17], Witten et al. [63], Harrington [64] và Marsland [65] nếu muốn bổ sung các dẫn xuất chặt chẽ hơn về mặt toán học.

3.1.2.1. *Hàm tổn thất là gì?*

Ở cấp độ cơ bản nhất, một hàm tổn thất định lượng sẽ xác định mức độ tốt hay xấu của một bộ dự đoán phân loại các điểm dữ liệu đầu vào trong một tập dữ liệu. Một cách trực quan hóa các hàm tổn thất được biểu diễn theo thời gian cho hai mô hình riêng biệt được huấn luyện trên bộ dữ liệu CIFAR-10 được hiển thị trong Hình 3.4. Tổn thất càng nhỏ, công việc phân loại càng tốt trong việc mô hình hóa mối quan hệ giữa dữ liệu đầu vào và lớp nhãn đầu ra (mặc dù có một điểm có thể điều chỉnh mô hình - bằng cách mô hình hóa dữ liệu huấn luyện quá chặt chẽ, mô hình mất khả năng để khai quát hóa, một hiện tượng sẽ được xem xét chi tiết trong mục 3.4). Ngược lại, sai số càng lớn, càng cần phải thực hiện nhiều công việc để tăng độ chính xác phân loại.



Hình 3.4: Sai số huấn luyện cho hai mô hình riêng biệt được huấn luyện trên bộ dữ liệu CIFAR-10 được biểu diễn theo thời gian. Hàm tổn thất định lượng cách thức tốt hay xấu. Đây là việc mà một mô hình nhất định đang thực hiện để phân loại các điểm dữ liệu từ bộ dữ liệu. Mô hình 1 đạt được mức giảm thấp hơn đáng kể so với Mô hình 2.

Để cải thiện độ chính xác phân loại, cần điều chỉnh các tham số ma trận trọng số W hoặc vectơ sai lệch b . Cách tiến hành cập nhật các tham số này là một vấn đề tối ưu hóa, sẽ được đề cập trong mục tiếp theo. Hiện tại, chỉ cần hiểu rằng một hàm tổn thất có thể được sử dụng để định lượng mức độ hàm tính điểm đang hoạt động tốt như thế nào trong việc phân loại các điểm dữ liệu đầu vào.

Lý tưởng nhất là sai số sẽ giảm theo thời gian khi điều chỉnh các tham số mô hình. Như Hình 3.4 chứng minh, sai số Mô hình 1 bắt đầu cao hơn một chút so với Mô hình 2, nhưng sau đó giảm nhanh và tiếp tục ở mức thấp khi được huấn luyện về bộ dữ liệu CIFAR-10. Ngược lại, sai số cho Mô hình 2 giảm ban đầu nhưng nhanh chóng bị đình trệ. Trong ví dụ cụ thể này, Mô hình 1 đang đạt được sai số tổng thể thấp hơn và có khả năng là một mô hình mong muốn hơn sẽ được sử dụng để phân loại các ảnh khác từ bộ dữ liệu CIFAR-10. Chúng ta nói “có khả năng” bởi vì có thể Mô hình 1 đã quá phù hợp với dữ liệu huấn luyện, khái niệm quá khớp này và cách phát hiện ra nó sẽ được đề cập đến trong mục 3.4.

3.1.2.2. Hàm tổn thất của SVM nhiều lớp

Máy SVM nhiều lớp (như tên cho thấy) được lấy ý tưởng từ (Máy tính tuyến tính) (SVM) [66] sử dụng hàm tính điểm f để ánh xạ điểm dữ liệu tới điểm số cho từng lớp nhãn. Hàm f này là một ánh xạ học đơn giản:

$$f(x_i, W, b) = Wx_i + b \quad (3.2)$$

Sau khi đã có hàm tính điểm, cần xác định mức độ hiệu quả của hàm này hay hiệu suất (đưa ra ma trận trọng số W và vectơ sai lệch b) khi đưa ra dự đoán. Để thực hiện quyết định này, cần một hàm tổn thất.

Nhớ rằng khi tạo một mô hình học máy, có một ma trận thiết kế X, trong đó mỗi hàng trong X chứa một điểm dữ liệu mà muốn phân loại. Trong bối cảnh phân loại ảnh, mỗi hàng trong X là một ảnh và tìm cách gắn nhãn chính xác cho ảnh này. Có thể truy cập ảnh thứ i bên trong X thông qua cú pháp x_i .

Tương tự, cũng có một vectơ y chứa lớp nhãn cho mỗi X. Các giá trị y này là nhãn thực tế và hy vọng rằng hàm tính điểm sẽ dự đoán chính xác. Giống như có thể truy cập một ảnh nhất định dưới dạng x_i , có thể truy cập lớp nhãn được liên kết thông qua y_i .

Để đơn giản, viết tắt hàm tính điểm là s:

$$s = f(x_i, W) \quad (3.3)$$

Có thể đạt được số điểm dự đoán lớp thứ j thông qua điểm dữ liệu thứ i:

$$s_j = f(x_i, W)_j \quad (3.4)$$

Sử dụng cú pháp này, có thể kết hợp tất cả lại với nhau và có được hàm tổn thất cơ sở:

$$L_i \sum_{j \neq y_i} \max(0, s_i - s_{y_i} + 1) \quad (3.5)$$

Về cơ bản, hàm tổn thất đang tổng hợp trên tất cả các lớp không chính xác ($i = j$) và so sánh đầu ra hàm tính điểm được trả về cho lớp nhãn thứ j (lớp không chính xác) và lớp y_i (lớp đúng). Áp dụng thao tác tối đa để kẹp các giá trị ở mức 0, điều này rất quan trọng để đảm bảo không tính tổng các giá trị âm.

Một x_i đã cho được phân loại chính xác khi mất $L_i = 0$. Để rút ra sự tổn thất trong toàn bộ tập huấn luyện, chỉ cần lấy giá trị trung bình từng L_i :

$$L = \frac{1}{N} \sum_{i=1}^N L_i \quad (3.6)$$

Một hàm tổn thất liên quan khác có thể gấp là hàm tổn thất cơ sở bình phương:

$$L_i \sum_{j \neq y_i} \max(0, s_i - s_{y_i} + 1)^2 \quad (3.7)$$

Việc sử dụng hàm tổn thất nào hoàn toàn phụ thuộc vào tập dữ liệu. Hàm tổn thất cơ sở tiêu chuẩn được sử dụng nhiều hơn, nhưng trên một số bộ dữ liệu, biến thể bình phương có thể thu được độ chính xác tốt hơn. Nhìn chung, đây là một siêu tham số nên xem xét điều chỉnh.

Một ví dụ về sai số SVM nhiều lớp:

Bây giờ, sau khi đã xem xét các phương trình toán học đãng sau hàm tổn thất cơ sở, sẽ đưa ra một ví dụ hoạt động. Một lần nữa, lại sử dụng bộ dữ liệu Động vật trên động vật nhằm mục đích phân loại một ảnh nhất định có chứa một con mèo, chó hoặc gấu trúc. Để bắt đầu, xem Hình 3.5 trong đó đã bao gồm ba ví dụ huấn luyện từ ba lớp bộ dữ liệu động vật.

Cho một số ma trận trọng số tùy ý W và vectơ sai lệch b , điểm đầu ra $f(x, W) = Wx + b$ được hiển thị trong phần thân ma trận. Điểm càng lớn, hàm tính điểm càng có sự liên quan hơn đến dự đoán.

Bắt đầu bằng cách tính toán sự tổn thất Li cho lớp con chó:

```
1 >>> max(0, 1.33 - 4.26 + 1) + max(0, -1.01 - 4.26 + 1)
2 0
```

Lưu ý rằng phương trình ở đây bao gồm hai thuật ngữ - sự khác biệt giữa điểm số chó dự đoán và cả điểm số mèo và gấu trúc. Ngoài ra, quan sát mức độ mất cho con chó bằng 0 - điều này ngụ ý rằng con chó đã được dự đoán chính xác. Một cuộc điều tra nhanh về ảnh số 1 từ Hình 3.5 ở trên cho thấy kết quả này là đúng: điểm số con chó là con chó lớn hơn cả điểm số con mèo và con gấu trúc.

Tương tự, có thể tính toán sai số bản lề cho Ảnh số 2, ảnh này có chứa một con mèo:

```
3 >>> max(0, 3.76 - (-1.20) + 1) + max(0, -3.81 - (-1.20) + 1)
4 5.96
```



	Image #1	Image #2	Image #3
Dog	4.26	3.76	-2.37
Cat	1.33	-1.20	1.03
Panda	-1.01	-3.81	-2.27

Hình 3.5: Ở đầu hình, có ba ảnh đầu vào: một cho mỗi lớp chó, mèo và gấu trúc tương ứng. Phần thân bảng chứa các đầu ra hàm tính điểm từng lớp. Sử dụng hàm tính điểm để tính tổng sai số cho mỗi ảnh đầu vào.

Trong trường hợp này, hàm tổn thất lớn hơn 0, cho thấy dự đoán không chính xác. Nhìn vào hàm tính điểm, thấy rằng mô hình dự đoán con chó là nhãn được đề xuất với số điểm là 3,76 (vì đây là nhãn có số điểm cao nhất). Biết rằng nhãn này không chính xác - và trong mục 3.2, sẽ học cách tự động điều chỉnh các trọng số để sửa các dự đoán này.

Cuối cùng, để tính toán hàm tổn thất cơ sở cho ví dụ gấu trúc:

```
5 >>> max(0, -2.37 - (-2.27) + 1) + max(0, 1.03 - (-2.27) + 1)  
6 5.199999999999999
```

Một lần nữa, tổn thất là khác 0, vì vậy biết rằng có một dự đoán không chính xác. Nhìn vào hàm tính điểm, mô hình đã gắn nhãn không chính xác cho ảnh này là mèo trong khi nó phải là gấu trúc.

Sau đó có thể có được tổng sai số trong ba ví dụ bằng cách lấy trung bình:

```
7 >>> (0.0 + 5.96 + 5.2) / 3.0  
8 3.72
```

Do đó, với ba ví dụ huấn luyện, sai số bản lề tổng thể là 3,72 cho các tham số W và b.

Cũng lưu ý rằng sai số bằng 0 chỉ với một trong ba ảnh đầu vào, nhưng hai trong số các dự đoán là không chính xác. Trong mục tiếp theo, sẽ học cách tối ưu hóa W và b để đưa ra dự đoán tốt hơn bằng cách sử dụng hàm tổn thất để giúp lái xe và điều khiển đi đúng hướng.

3.1.2.3. Phân loại tổn thất entropy chéo và phân loại Softmax

Mặc dù hàm tổn thất cơ sở khá phổ biến, nhưng có nhiều khả năng gặp phải tình trạng tổn thất entropy chéo và phân loại Softmax trong bối cảnh học sâu và mạng nơ-ron tích chập.

Trình phân loại Softmax cung cấp cho xác suất cho mỗi lớp nhãn trong khi hàm tổn thất cơ sở cung cấp cho margin.

Dễ dàng hơn nhiều khi giải thích xác suất thay vì điểm số lè. Hơn nữa, đối với các bộ dữ liệu như ImageNet, thường xem xét độ chính xác cấp 5 mạng nơ-ron tích chập (nơi kiểm tra xem nhãn có phải thực sự nằm trong top 5 nhãn được dự đoán bởi mạng cho ảnh đầu vào cụ thể không). Kiểm tra (1) là lớp nhãn thực sự tồn tại trong top 5 nhãn dự đoán 5 và (2) là xác suất liên quan đến mỗi nhãn là một thuộc tính tốt.

Tìm hiểu về tổn thất entropy chéo

Trình phân loại Softmax là một khái quát dạng nhị phân hồi quy

logistic. Cũng giống như hàm tổn thất cơ sở hoặc hàm tổn thất cơ sở bình phương, hàm ánh xạ f được xác định sao cho nó lấy một bộ dữ liệu x_i và ánh xạ chúng tới lớp nhãn đầu ra thông qua phép nhân chấm ma trận dữ liệu và ma trận trọng số W (bỏ qua thuật ngữ sai lệch cho ngắn gọn):

$$f(x_i, W) = Wx_i \quad (3.8)$$

Tuy nhiên, không giống như hàm tổn thất cơ sở, có thể hiểu các điểm số này là xác suất không chuẩn hóa cho mỗi lớp nhãn, tương đương với việc hoán đổi hàm tổn thất cơ sở với tổn thất entropy chéo:

$$L_i = \log(esy_i / \sum esj) \quad (3.9)$$

Để bắt đầu, hàm tổn thất nên giảm:

$$L_i = -\log P(Y=y_i | X=x_i) \quad (3.10)$$

Câu lệnh xác suất có thể được hiểu là:

$$P(Y=k | X=x_i) = e^{s_{y_i}} / \sum e^s \quad (3.11)$$

Hàm tính điểm tiêu chuẩn là:

$$s = f(x_i, W) \quad (3.12)$$

Nhìn chung, điều này mang lại hàm tổn thất cuối cùng cho một điểm dữ liệu duy nhất, giống như trên:

$$L_i = \log(esy_i / \sum esj) \quad (3.13)$$

Lưu ý rằng logarit ở đây thực sự là cơ sở e (logarit tự nhiên) vì đang thực hiện nghịch đảo lũy thừa so với e trước đó. Hàm lũy thừa và chuẩn hóa thực tế thông qua tổng số mũ là hàm Softmax. Cũng giống như hàm tổn thất cơ sở và hàm tổn thất cơ sở bình phương, tính toán sai số entropy chéo trên toàn bộ tập dữ liệu được thực hiện bằng cách lấy trung bình:

$$L = \frac{1}{N} \sum_{i=1}^N L_i \quad (3.14)$$

Một lần nữa, lại có tình bối rối qua thuật ngữ chính quy từ hàm tổn thất. Quay trở lại chính quy, giải thích nó là gì, cách sử dụng và lý do tại sao nó quan trọng đối với mạng nơ-ron và học sâu nằm trong mục 3.2. Nếu các phương trình trên có vẻ đáng sợ, đừng lo lắng - sẽ giải quyết các ví dụ bằng số trong phần tiếp theo để đảm bảo hiểu cách thức hoạt động của tổn thất entropy chéo.

3.1.2.4. Một số ví dụ về hàm Softmax

Để chứng minh tổn thất entropy chéo trong hành động, xem xét Hình 3.6. Mục tiêu là phân loại xem ảnh trên có chứa chó, mèo hay gấu trúc hay không phải 3 loài trên. Rõ ràng, có thể thấy rằng ảnh là một con gấu trúc - nhưng trình phân loại Softmax nghĩ gì? Để tìm hiểu, sẽ cần phải làm việc qua từng bảng trong Hình 3.6.

Bảng đầu tiên bao gồm đầu ra hàm tính điểm f cho mỗi trong ba lớp tương ứng. Các giá trị này là xác suất đăng nhập không chuẩn hóa cho ba lớp để hàm mũ lũy thừa đầu ra hàm tính điểm (e^s , trong đó s là giá trị hàm số điểm), mang lại xác suất không chuẩn hóa (bảng thứ hai).

Bước tiếp theo là lấy mẫu số, tính tổng số mũ và chia cho tổng, từ đó mang lại xác suất thực tế liên quan đến mỗi lớp nhãn (bảng thứ ba). Lưu ý cách xác suất tổng hợp thành một. Cuối cùng, có thể lấy logarit tự nhiên âm, $\ln(p)$, trong đó p là xác suất chuẩn hóa, mang lại sai số cuối cùng (bảng thứ tư và cuối cùng).

Trong trường hợp này, trình phân loại Softmax sẽ báo cáo chính xác ảnh là gấu trúc với độ tin cậy 93,93%. Sau đó, có thể lặp lại quy trình này cho tất cả các ảnh trong tập huấn luyện, lấy mức trung bình và thu được sai số entropy chéo chung cho tập huấn luyện. Quá trình này cho phép định lượng mức độ tốt hay xấu một bộ thông số đang hoạt động trên tập huấn luyện.

Scoring Function	
Dog	-3.44
Cat	1.16
Panda	3.91



Input Image

	Scoring Function	Unnormalized Probabilities
Dog	-3.44	0.03
Cat	1.16	3.19
Panda	3.91	49.90

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities
Dog	-3.44	0.0321	0.0006
Cat	1.16	3.1899	0.0601
Panda	3.91	49.8990	0.9393

	Scoring Function	Unnormalized Probabilities	Normalized Probabilities	Negative Log Loss
Dog	-3.44	0.0321	0.0006	
Cat	1.16	3.1899	0.0601	
Panda	3.91	49.8990	0.9393	0.0626

Hình 3.6: Bảng đầu tiên: Để tính toán sai số entropy chéo, bắt đầu với đầu ra hàm tính điểm. Bảng thứ hai: lũy thừa các giá trị đầu ra từ hàm tính điểm cho xác suất không chuẩn hóa. Bảng thứ ba: Để có được xác suất thực tế, chia mỗi xác suất không chuẩn hóa riêng lẻ cho tổng tất cả các xác suất không chuẩn hóa. Bảng thứ tư: Lấy logarit tự nhiên âm xác suất cho sự thật mặt đất chính xác mang lại sai số cuối cùng cho điểm dữ liệu.

3.1.3. Tóm tắt

Trong mục này, tác giả đã xem xét bốn thành phần việc học được tham số hóa:

1. Dữ liệu
2. Hàm tính điểm
3. Hàm tổn thất
4. Trọng lượng và độ lệch

Trong bối cảnh phân loại ảnh, dữ liệu đầu vào là tập dữ liệu ảnh. Hàm tính điểm tạo ra các dự đoán cho một ảnh đầu vào nhất định. Hàm tổn thất sau đó định lượng mức độ tốt hay xấu một tập hợp dự đoán trên tập dữ liệu. Cuối cùng, ma trận trọng số và vectơ sai lệch là những gì cho phép thực sự học hỏi từ dữ liệu đầu vào - các tham số này sẽ được điều chỉnh liên tục thông qua các phương pháp tối ưu hóa nhằm cố gắng đạt được độ chính xác phân loại cao hơn.

Sau đó đã xem xét hai hàm tổn thất phổ biến: hàm tổn thất cơ sở và tổn thất entropy chéo. Mặc dù hàm tổn thất cơ sở được sử dụng trong nhiều ứng dụng học máy (như SVM), nhưng có thể chắc chắn rằng sẽ gặp tổn thất entropy chéo với tần suất nhiều hơn chủ yếu do thực tế là phân loại Softmax xác suất đầu ra thay bởi các lè. Tổn thất entropy chéo và phân loại softmax mô tả giống cơ chế hoạt động sinh học của não người với dạng xác suất. Để biết thêm thông tin về hàm tổn thất cơ sở mất và tổn thất entropy chéo, vui lòng tham khảo khóa học Stanford cs231n Đại học Stanford [62, 67].

Trong mục tiếp theo, sẽ xem xét các phương pháp tối ưu hóa được sử dụng để điều chỉnh ma trận trọng số và vectơ sai lệch. Các phương pháp tối ưu hóa cho phép các thuật toán thực sự học hỏi từ dữ liệu đầu vào bằng cách cập nhật ma trận trọng số và vectơ sai lệch dựa trên đầu ra các hàm tính điểm và tổn thất. Sử dụng các kỹ thuật này, có thể thực hiện các bước tăng dần đối với các giá trị tham số có được sai số thấp hơn và độ chính xác cao hơn. Các phương pháp tối ưu hóa là nền tảng mạng nơ-ron hiện đại và học sâu, và nếu không có chúng, sẽ không thể học các mẫu từ dữ liệu đầu vào, vì vậy chắc chắn bạn phải chú ý đến các mục sắp tới.

3.2. PHƯƠNG PHÁP TỐI UU HÓA VÀ CHÍNH QUY HÓA

Hầu như tất cả các phương pháp học sâu sẽ hiệu quả hơn với một thuật toán rất quan trọng: Stochastic Gradient Descent (SGD), - Goodfellow et al. [33]

Trong các chương trước, đã thảo luận về khái niệm học tham số và

cách học này cho phép xác định hàm tính điểm mà nó ánh xạ dữ liệu đầu vào thành các nhãn lớp đầu ra.

Hàm tính điểm này được xác định theo hai tham số quan trọng là ma trận trọng số W và vectơ sai lệch b . Hàm tính điểm nhận các tham số này làm đầu vào và trả về một dự đoán cho mỗi điểm dữ liệu đầu vào x_i .

Đã xem xét hai hàm tổn thất phổ biến: hàm tổn thất SVM nhiều lớp và hàm tổn thất entropy chéo. Các hàm tổn thất về cơ bản được sử dụng để đánh giá mức độ tốt hay kém của bộ dự đoán (tập hợp các tham số) phân loại các điểm dữ liệu đầu vào trong bộ dữ liệu.

Phản kế tiếp sẽ xem xét khía cạnh quan trọng nhất học máy (hay cụ thể với mạng nơ-ron và học sâu) đó là tối ưu hóa. Các thuật toán tối ưu hóa là các động cơ cung cấp sức mạnh cho mạng nơ-ron và cho phép chúng học các mẫu từ bộ dữ liệu. Biết rằng để nhận được một bộ phân loại có độ chính xác cao phụ thuộc vào việc tìm tập các trọng số W và b sao cho các điểm dữ liệu được phân loại chính xác.

Nhưng làm thế nào để tìm và nhận được ma trận trọng số W và vectơ sai lệch b có được độ chính xác phân loại cao? Có nên khởi tạo ngẫu nhiên chúng, đánh giá và lặp đi lặp lại nhiều lần không? Để hy vọng rằng đến một lúc nào đó tìm được một tập hợp các tham số có được sự phân loại hợp lý? Có thể làm thế, tuy nhiên do các mạng học sâu hiện đại có các tham số lên tới hàng chục triệu, có thể mất rất nhiều thời gian để thử sai trước một bộ thông số hợp lý.

Thay vì dựa vào tính ngẫu nhiên thuần túy, cần xác định một thuật toán tối ưu hóa cho phép cải thiện W và b . Mục này sẽ xem xét thuật toán phổ biến nhất được sử dụng để huấn luyện mạng nơ-ron và mô hình học sâu – Thuật toán Suy giảm độ dốc (Gradient Descent). Gradient có nhiều biến thể nhưng trong mỗi trường hợp ý tưởng là như nhau: lặp lại việc đánh giá các tham số, tính toán tổn thất, sau đó thực hiện một bước nhỏ theo hướng giảm thiểu tổn thất.

3.2.1. Thuật toán suy giảm độ dốc (GRADIENT DESCENT)

Thuật toán suy giảm độ dốc (Gradient Descent) có hai phần chính:

1. Triển khai tiêu chuẩn “vanilla”

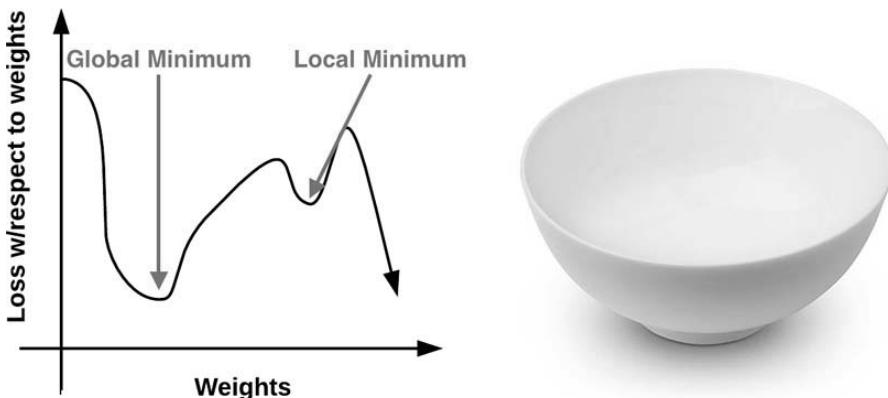
2. Phiên bản tối ưu hóa “ngẫu nhiên” – được sử dụng phổ biến hơn.

Trong phần này, xem xét việc triển khai vanilla cơ bản là nền tảng cơ sở kiến thức. Sau khi có những hiểu biết cơ bản về Gradient Descent, sẽ xem xét phiên bản ngẫu nhiên. Sau đó học về các tham

số thêm vào bao gồm động lượng và gia tốc Nesterov của thuật toán Gradient Descent.

3.2.1.1. Đồ thị biểu diễn sự tồn thắt và tối ưu hóa bề mặt

Thuật toán suy giảm độ dốc (Gradient Descent) là một thuật toán tối ưu hóa vòng lặp trên đồ thị hàm tồn thắt (còn được gọi là bề mặt tối ưu hóa). Ví dụ Gradient Descent chính xác là trực quan hóa các trọng số đọc theo trục x và hàm tồn thắt của một tập các trọng số đã cho đọc theo trục y (Hình 3.7, bên trái):



Hình 3.7: **Trái** : Hàm tồn thắt biểu diễn trên trục 2D. **Phải** : Hàm tồn thắt trong thực tế có thể được hình dung như một cái bát. Mục tiêu là áp dụng Thuật toán suy giảm độ dốc (Gradient Descent) để điều hướng đến đáy bát này (nơi có tồn thắt thấp).

Như có thể thấy, hàm tồn thắt có nhiều đỉnh và vùng trũng dựa trên các giá trị mà các tham số đảm nhận. Mỗi đỉnh là một cực đại cục bộ đại diện cho các khu vực mà giá trị hàm có sai số rất cao - cực đại cục bộ có sai số lớn nhất trên toàn bộ hàm tồn thắt là cực đại toàn phần. Tương tự, cũng có mức cực tiểu cục bộ đại diện cho nhiều vùng tồn thắt nhỏ.

Mức cực tiểu cục bộ với sai số nhỏ nhất trong toàn vùng tồn thắt là mức cực tiểu toàn phần. Về lý tưởng, mục đích là tìm cực tiểu toàn phần này, vị trí mà các tham số đảm nhận các giá trị tối ưu nhất có thể.

Câu hỏi đặt ra là: Nếu muốn đạt đến mức tối thiểu toàn phần, tại sao không trực tiếp nhảy vào nó?

Vấn đề ở đây là đồ thị hàm tồn thắt không biết trước. Thực tế ta không biết nó sẽ như thế nào. Thuật toán tối ưu hóa sẽ tìm vị trí điểm cực tiểu mà không rơi vào cực đại cục bộ mà không biết trước đồ thị của hàm tồn thắt.

3.2.1.2. Độ dốc (GRADIENT) trong GRADIENT DESCENT

Để giải thích về Thuật toán suy giảm độ dốc (Gradient Descent) trực quan hơn một chút, giả sử rằng có một robot - đặt tên cho nó là Chad (Hình 3.8, bên trái). Khi thực hiện Gradient Descent, thả ngẫu nhiên Chad ở đâu đó trên đồ thị hàm tốn thất (Hình 3.8, bên phải).



Hình 3.8: Trái : Robot Chad. Phải : Công việc Chad để điều hướng đồ thị hàm tốn thất và đi xuống đáy cái chén. Thật không may, cảm biến duy nhất mà Chad có thể sử dụng để điều khiển điều hướng là một hàm đặc biệt, được gọi là hàm tốn thất, hàm này phải hướng dẫn anh ta đến khu vực có giá trị hàm tốn thất thấp hơn.

Bây giờ Chad làm việc để điều hướng đến đáy cái chén (nơi có tốn thất tối thiểu). Có vẻ đủ dễ dàng phải không? Tất cả những gì Chad phải làm là tự định hướng sao cho anh ấy đối mặt với xuống dốc và đi lên dốc cho đến khi chạm đáy bát.

Nhưng vấn đề ở đây: Chad là một robot rất thông minh. Chad chỉ có một cảm biến - cảm biến này cho phép anh ta lấy các tham số W và b và sau đó tính toán hàm mất L . Do đó, Chad có thể tính toán vị trí tương đối trên đồ thị hàm tốn thất, nhưng Chad hoàn toàn không biết đang đi theo hướng nào nên thực hiện một bước để di chuyển bản thân đến gần đáy của bát.

Chad làm gì? Câu trả lời là áp dụng Gradient Descent. Tất cả những gì Chad cần làm là thực hiện Gradient Descent W . Có thể tính Gradient Descent W trên tất cả các chiều bằng phương trình sau: (công thức 3.15).

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h} \quad (3.15)$$

Nếu nhiều hơn 1 giá trị, Gradient Descent trở thành một vecto các

đạo hàm riêng. Vấn đề với phương trình này là:

1. Là một phép xấp xỉ với Gradient Descent.
2. Lập trình rất chậm.

Trong thực tế, có thể sử dụng phương pháp phân tích độ dốc thay thế. Phương pháp này chính xác và nhanh chóng, nhưng vô cùng khó để thực hiện do các đạo hàm và phép tính đa biến.

Trong nội dung cuốn sách chỉ đơn giản hóa khái niệm Gradient Descent là gì: cố gắng tối ưu hóa các tham số để giảm tổn thất thấp và độ chính xác phân loại cao thông qua một quá trình lặp theo hướng giảm thiểu tổn thất.

3.2.1.3. Thủ thuật bias

Trước khi chuyển sang thực hiện thuật toán Gradient Descent, thảo luận về một kỹ thuật có tên là “thủ thuật bias” (bias trick), một phương pháp kết hợp ma trận trọng số W và vecto sai lệch b thành một tham số duy nhất. Hàm tính điểm được xác định là: (công thức 3.16)

$$f(x_i, W, b) = W_{x_i} + b \quad (3.16)$$

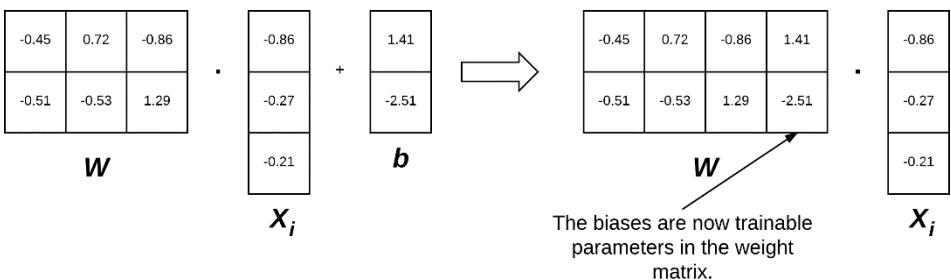
Thường rất khó khăn để theo dõi hai biến riêng biệt, cả về giải thích và triển khai, để tránh tình trạng này, có thể kết hợp W và b lại với nhau. Để kết hợp cả ma trận bias và trọng số, có thể thêm một giá trị kích thước (tức là cột) vào dữ liệu đầu vào X giữ hằng số 1 - đây là kích thước bias.

Thông thường, thêm cột mới cho từng x_i riêng lẻ làm kích thước đầu tiên hoặc cột cuối cùng. Trong thực tế, có thể chọn bất kỳ vị trí tùy ý nào để chèn một cột vào ma trận thiết kế. Như vậy cho phép viết lại hàm tính điểm thông qua một ma trận nhân: (công thức 3.17)

$$f(x_i, W) = W_{x_i} \quad (3.17)$$

Một lần nữa, được phép bỏ qua b ở đây vì nó được nhúng vào ma trận trọng số.

Bộ dữ liệu về động vật, ảnh có kích thước $32 \times 32 \times 3$ với tổng số 3.072 pixel. Mỗi x_i được biểu thị bằng một vecto $[3072 \times 1]$. Việc thêm vào một thứ nguyên có giá trị không đổi, bây giờ mở rộng vecto thành $[3073 \times 1]$. Tương tự, kết hợp ma trận trọng số và bias cũng mở rộng ma trận trọng số W thành $[3 \times 3073]$ thay vì $[3 \times 3072]$. Theo cách này, có thể coi bias là một tham số có thể học được trong ma trận trọng số mà không phải theo dõi rõ ràng trong một biến biệt lập.



Hình 3.9: Trái: Thông thường coi ma trận trọng số và vecto sai lệch là hai tham số riêng biệt. Phải: Tuy nhiên, thực sự có thể nhúng vecto sai lệch vào ma trận trọng số (từ đó biến nó thành một tham số có thể huấn luyện trực tiếp bên trong ma trận trọng số bằng cách khởi tạo ma trận trọng số bằng một cột phụ).

Để hình dung thủ thuật bias, xem Hình 3.9 (bên trái) nơi tách ma trận trọng số và độ lệch. Cho đến bây giờ, con số này mô tả là cách nghĩ về hàm tính điểm. Nhưng thay vào đó, có thể kết hợp W và b với nhau (Hình 3.9, bên phải). Áp dụng thủ thuật bias cho phép chỉ học một ma trận trọng số duy nhất. Đối với tất cả các ví dụ tiếp theo, bất cứ khi nào đề cập đến W thì cũng bao gồm vecto sai lệch b trong ma trận trọng số.

3.2.1.4. Mã giả cho Gradient Descent

Phần dưới đây là mã giả cho thuật toán suy giảm độ dốc theo gradient được lấy ý tưởng từ các slide cs231n [8]

```

1 while True:
2     Wgradient = evaluate_gradient(loss, data, W)
3     W += -alpha * Wgradient

```

Mã giả ngẫu nhiên này là những mà gì tất cả các biến thể Gradient Descent gốc được xây dựng. Bắt đầu Dòng 1 bằng cách lặp cho đến khi một số điều kiện được đáp ứng, thường là:

Một số lượng chu kỳ xác định đã trôi qua (có nghĩa là thuật toán học đã “thấy” được mỗi điểm dữ liệu huấn luyện N lần).

Tồn thắt đã trở nên đủ thấp hoặc độ chính xác huấn luyện đã đủ cao.

Loss đã không được cải thiện trong M chu kỳ tiếp liên tiếp.

Dòng 2 sau đó gọi một hàm có tên là evaluate_gradient. Hàm này yêu cầu ba tham số:

1. Loss: Một hàm được sử dụng để tính toán tồn thắt trên các tham số W và dữ liệu đầu vào hiện tại.

2. Dữ liệu: Dữ liệu huấn luyện trong đó mỗi mẫu huấn luyện được thể hiện bằng một ảnh (hoặc vecto đặc trưng).

3. W: Ma trận trọng số thực tế mà đang tối ưu hóa hơn. Mục tiêu là áp dụng giảm Gradient Descent để tìm W mang lại tổn thất tối thiểu.

Hàm evaluate_gradient trả về một vecto có K- chiều, trong đó K là số lượng kích thước trong vecto ảnh/đặc trưng. Biến Wgradient là Gradient Descent thực tế, trong đó có một mục nhập Gradient Descent cho mỗi thứ nguyên.

Sau đó, áp dụng thuật toán Gradient Descent tại Dòng 3. Nhân Wgradient với alpha (α), đó là tốc độ học. Tốc độ học kiểm soát kích thước bước.

Trong thực tế, thường dành nhiều thời gian để tìm giá trị tối ưu α - đó là thông số quan trọng nhất trong mô hình. Nếu α quá lớn, dành toàn bộ thời gian để tìm quanh đồ thị tổn thất, thực tế không bao giờ thực sự giảm dần xuống đáy lưu vực (trừ khi này ngẫu nhiên đưa đến đó bằng may mắn thuần túy). Ngược lại, nếu α quá nhỏ, thì cần nhiều lần lặp (có lẽ là rất nhiều) để đến đáy lưu vực. Việc tìm giá trị tối ưu α là một bài toán hóc búa và tốn một thời gian nhất định để cố gắng tìm giá trị tối ưu cho biến này trong mô hình và tập dữ liệu.

3.2.1.5. Thực hiện thuật toán suy giảm độ dốc (Gradient Descent) cơ bản trong Python

Bây giờ đã biết cơ bản về Gradient Descent, triển khai nó trong Python và sử dụng nó để phân loại dữ liệu. Mở một tệp mới, đặt tên là gradient_descent.py và chèn chương trình sau đây:

```
1 # import the necessary packages
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import classification_report
4 from sklearn.datasets import make_blobs
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import argparse
8
9 def sigmoid_activation(x):
10     # compute the sigmoid activation value for a given input
11     return 1.0 / (1 + np.exp(-x))
```

Dòng 2-7 nhập các gói Python cần thiết

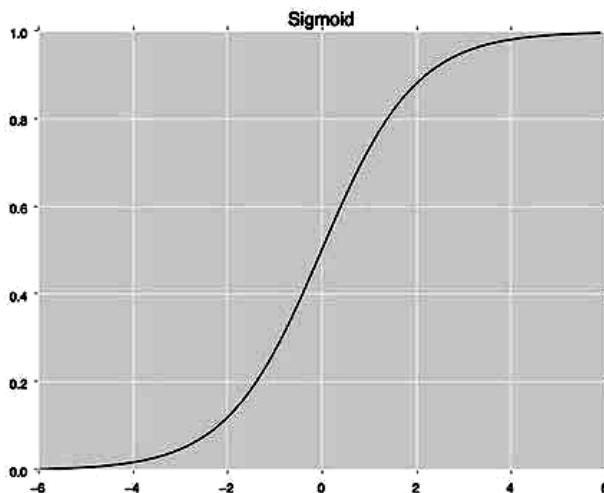
Sau đó, xác định hàm sigmoid_activation trên Dòng 9. Khi được vẽ, hàm này giống với một đường cong nối đường xiên hình chữ nhật (Hình 3.10). Gọi đó là hàm tác động bởi vì hàm này kích hoạt và kích hoạt tính

năng ON (giá trị đầu ra $> 0,5$) hoặc đường tắt OFF (giá trị đầu ra $\leq 0,5$) dựa trên đầu vào x .

Có thể xác định mối quan hệ này thông qua phương pháp dự đoán dưới đây:

```
13 def predict(X, W):
14     # take the dot product between our features and weight matrix
15     preds = sigmoid_activation(X.dot(W))

16
17     # apply a step function to threshold the outputs to binary
18     # class labels
19     preds[preds <= 0.5] = 0
20     preds[preds > 0] = 1
21
22     # return the predictions
23     return preds
```



Hình 3.10: Hàm tách động sigmoid. Hàm này tập trung tại $x = 0,5$, $y = 0,5$. Hàm bao hòa ở phía cuối.

Cho một tập hợp các điểm dữ liệu đầu vào X và có trọng số W , gọi hàm `sigmoid_activation` để có được một bộ dự đoán (Dòng 15). Sau đó, ngưỡng các dự đoán: mọi dự đoán có giá trị $\leq 0,5$ được đặt thành 0 trong khi mọi dự đoán có giá trị $> 0,5$ được đặt thành 1 (Dòng 19 và 20). Các dự đoán sau đó được trả về hàm gọi trên Dòng 23.

Mặc dù có các lựa chọn thay thế (tốt hơn) cho hàm tách động sigmoid, nhưng sigmoid là một hàm tách động phi tuyến tính có thể sử dụng để vượt ngưỡng dự đoán.

Tiếp theo, để phân tích cú pháp đối số dòng lệnh:

```

25 # construct the argument parse and parse the arguments
26 ap = argparse.ArgumentParser()
27 ap.add_argument("-e", "--epochs", type=float, default=100,
28     help="# of epochs")
29 ap.add_argument("-a", "--alpha", type=float, default=0.01,
30     help="learning rate")
31 args = vars(ap.parse_args())

```

Cho hai đối số dòng lệnh (tùy chọn) cho tập lệnh:

--epochs: số lượng chu kỳ sử dụng khi huấn luyện trình phân loại bằng cách sử dụng Gradient Descent.

--alpha: tốc độ học cho Gradient Descent giảm dần. 0,1, 0,01 và 0,001 là các giá trị tốc độ học ban đầu, nhưng một lần nữa, đây là một siêu tham số mà cần điều chỉnh cho các vấn đề phân loại riêng.

Bây giờ, các đối số dòng lệnh đã được phân tích cú pháp, để tạo một số dữ liệu để phân loại:

```

33 # generate a 2-class classification problem with 1,000 data points,
34 # where each data point is a 2D feature vector
35 (X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
36     cluster_std=1.5, random_state=1)
37 y = y.reshape((y.shape[0], 1))
38
39 # insert a column of 1's as the last entry in the feature
40 # matrix -- this little trick allows us to treat the bias
41 # as a trainable parameter within the weight matrix
42 X = np.c_[X, np.ones((X.shape[0]))]
43
44 # partition the data into training and testing splits using 50% of
45 # the data for training and the remaining 50% for testing
46 (trainX, testX, trainY, testY) = train_test_split(X, y,
47     test_size=0.5, random_state=42)

```

Trên Dòng 35, thực hiện gọi hàm `make_blobs`, tạo ra 1.000 điểm dữ liệu được phân tách thành hai lớp. Các điểm dữ liệu này là 2D, nghĩa là rằng các vecto đặc trưng có độ dài là 2. Các nhãn cho mỗi điểm dữ liệu này là 0 hoặc 1. Mục tiêu là huấn luyện một trình phân loại dự đoán chính xác nhãn của lớp cho từng điểm dữ liệu.

Dòng 42 áp dụng thủ thuật bias (chi tiết ở trên) cho phép bỏ qua việc theo dõi rõ ràng vecto sai lệch b, bằng cách chèn một cột hoàn toàn mới 1s làm mục cuối cùng trong ma trận thiết kế X. Thêm một cột chứa hàng số giá trị trên tất cả các vecto tính năng cho phép coi bias là một tham số có thể huấn luyện trong ma trận trọng số W chứ không phải là một biến hoàn toàn biệt lập.

Khi đã chèn thêm cột, phân vùng dữ liệu thành các phần tách thử

nghiệm và thử nghiệm trên Dòng 46 và 47, sử dụng 50% dữ liệu cho huấn luyện và 50% cho thử nghiệm.

Khối tiếp theo xử lý việc khởi tạo ngẫu nhiên ma trận trọng số bằng cách sử dụng phân phối đồng nhất sao cho có cùng số kích thước với các hàm ngõ vào (bao gồm cả độ lệch):

```
49 # initialize our weight matrix and list of losses
50 print("[INFO] training...")
51 W = np.random.randn(X.shape[1], 1)
52 losses = []
```

Cũng có thể thấy khởi tạo 0 và 1 trong trọng số, nhưng như phần sau trong cuốn sách này có thảo luận, khởi tạo tốt là rất quan trọng để huấn luyện một mạng nơ-ron, do đó việc khởi tạo ngẫu nhiên cùng với các phương pháp phỏng đoán đơn giản thường tốt hơn trong đa số trường hợp [68].

Dòng 52 khởi tạo một danh sách để theo dõi các sai số sau mỗi chu kỳ. Khi kết thúc tập lệnh Python, biểu thị sự tồn thât (lý tưởng giảm theo thời gian).

Tất cả các biến hiện đã được khởi tạo, vì vậy có thể chuyển sang quy trình huấn luyện và Gradient Descent thực tế:

```
54 # loop over the desired number of epochs
55 for epoch in np.arange(0, args["epochs"]):

56     # take the dot product between our features 'X' and the weight
57     # matrix 'W', then pass this value through our sigmoid activation
58     # function, thereby giving us our predictions on the dataset
59     preds = sigmoid_activation(trainX.dot(W))

60

61     # now that we have our predictions, we need to determine the
62     # 'error', which is the difference between our predictions and
63     # the true values
64     error = preds - trainY
65     loss = np.sum(error ** 2)
66     losses.append(loss)
```

Trên Dòng 55, bắt đầu lặp qua số lượng --epoch được cung cấp. Theo mặc định, cho phép quy trình huấn luyện để thấy được một trong số các điểm huấn luyện tổng cộng 100 lần (do đó, 100 epoch). Dòng 59 lấy nhân chập giữa toàn bộ tập huấn luyện và ma trận trọng số.

W: đầu ra phép nhân chập này được cung cấp thông qua hàm tác động sigmoid, mang lại dự đoán.

Dựa vào dự đoán, bước tiếp theo là xác định lỗi các dự đoán, hay đơn giản hơn là sự khác biệt giữa dự đoán và các giá trị thực (Dòng 64). Dòng

65 tính toán sai số bình phương nhỏ nhất so với dự đoán, một tồn thát đơn giản thường được sử dụng cho các vấn đề phân loại nhị phân. Mục tiêu quy trình huấn luyện này là để giảm thiểu lỗi bình phương nhỏ nhất, thêm khoản sai số này vào danh sách tồn thát trên Dòng 66, vì vậy sau đó có thể vạch ra tồn thát theo thời gian.

Bây giờ có lỗi, có thể tính toán Gradient Descent và sau đó sử dụng nó để cập nhật ma trận trọng số W:

```
68 # the gradient descent update is the dot product between our
69 # features and the error of the predictions
70 gradient = trainX.T.dot(error)
71
72 # in the update stage, all we need to do is "nudge" the weight
73 # matrix in the negative direction of the gradient (hence the
74 # term "gradient descent" by taking a small step towards a set
75 # of "more optimal" parameters
76 W += -args["alpha"] * gradient
77
78 # check to see if an update should be displayed
79 if epoch == 0 or (epoch + 1) % 5 == 0:
80     print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
81         loss))
```

Dòng 70 xử lý tính Gradient Descent, là phép nhân chập giữa các điểm dữ liệu X và lỗi.

Dòng 76 là bước quan trọng nhất trong thuật toán và là nơi diễn ra quá trình giảm Gradient Descent thực tế. Ở đây cập nhật ma trận trọng số W bằng cách thực hiện một bước theo hướng tiêu cực Gradient Descent, do đó cho phép di chuyển về phía dưới đáy lưu vực đồ thị tồn thát (do đó, sử dụng thuật ngữ Gradient Descent). Sau khi cập nhật ma trận trọng số, kiểm tra xem liệu bản cập nhật có được hiển thị cho thiết bị đầu cuối không (Dòng 79-81) và sau đó tiếp tục lặp cho đến khi số lượng epoch mong muốn được đáp ứng - do đó, Gradient Descent là thuật toán lặp.

Bộ phân loại hiện đang được huấn luyện. Bước tiếp theo là đánh giá:

```
83 # evaluate our model
84 print("[INFO] evaluating...")
85 preds = predict(testX, W)
86 print(classification_report(testY, preds))
```

Để thực sự đưa ra dự đoán bằng ma trận trọng số W, gọi phương thức dự đoán trên testX và W trên Dòng 85. Đưa ra dự đoán, hiển thị báo cáo phân loại được định dạng cho thiết bị đầu cuối trên Dòng 86.

Khối cuối cùng biểu diễn đồ thị (1) dữ liệu test để có thể hình dung được tập dữ liệu đang cố gắng phân loại và (2) hàm tồn thát theo thời gian:

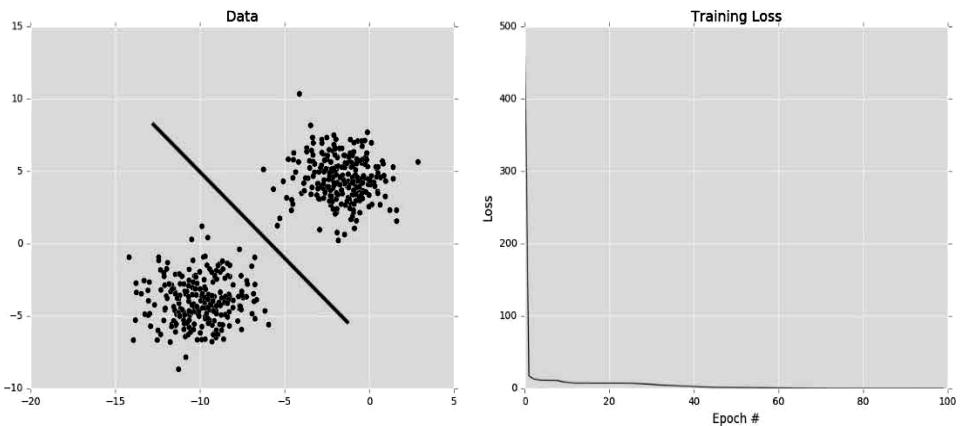
```
88 # plot the (testing) classification data
89 plt.style.use("ggplot")
90 plt.figure()
91 plt.title("Data")
92 plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY, s=30)
93
94 # construct a figure that plots the loss over time
95 plt.style.use("ggplot")
96 plt.figure()
97 plt.plot(np.arange(0, args["epochs"]), losses)
98 plt.title("Training Loss")
99 plt.xlabel("Epoch #")
100 plt.ylabel("Loss")
101 plt.show()
```

3.2.1.6. Kết quả Gradient Descent

Để thực thi tập lệnh, chỉ cần đưa ra lệnh như bên dưới

```
$ python gradient_descent.py
[INFO] training...
[INFO] epoch=1, loss=486.5895513
[INFO] epoch=5, loss=11.1087812
[INFO] epoch=10, loss=9.1312984
[INFO] epoch=15, loss=7.0049498
[INFO] epoch=20, loss=6.9914949
[INFO] epoch=25, loss=6.9382765
[INFO] epoch=30, loss=5.8285461
[INFO] epoch=35, loss=4.1750536
[INFO] epoch=40, loss=2.7319634
[INFO] epoch=45, loss=1.3891531
[INFO] epoch=50, loss=1.0787992
[INFO] epoch=55, loss=0.8927193
[INFO] epoch=60, loss=0.6001450
[INFO] epoch=65, loss=0.3200953
[INFO] epoch=70, loss=0.1651333
[INFO] epoch=75, loss=0.0941329
[INFO] epoch=80, loss=0.0602669
[INFO] epoch=85, loss=0.0424516
[INFO] epoch=90, loss=0.0321485
[INFO] epoch=95, loss=0.0256970
[INFO] epoch=100, loss=0.0213877
```

Như có thể thấy trong Hình 3.11 (bên trái), tập dữ liệu rõ ràng có thể phân tách tuyến tính (nghĩa là có thể vẽ một đường phân tách hai lớp dữ liệu). Tồn thắt cũng giảm đáng kể, bắt đầu rất cao và sau đó nhanh chóng giảm (phải). Có thể thấy sự tồn thắt giảm nhanh như thế nào bằng cách điều tra đầu ra thiết bị đầu cuối ở trên. Lưu ý rằng tồn thắt ban đầu > 400 nhưng giảm xuống ≈ 1.0 bởi epoch 50. Vào thời điểm huấn luyện kết thúc bởi epoch 100, tồn thắt đã giảm xuống một mức độ lớn đến 0,02.



Hình 3.11: Trái: Tập dữ liệu đầu vào cần phân loại thành hai bộ: đỏ và xanh. Bộ dữ liệu này rõ ràng có thể phân tách tuyến tính vì có thể vẽ một dòng duy nhất phân chia gọn gàng bộ dữ liệu thành hai lớp. Phải: Học một tập hợp các tham số để phân loại dữ liệu thông qua độ dốc góc. Tỷ lệ thất bại đầu rất cao nhưng nhanh chóng giảm xuống gần bằng 0.

Biểu đồ này xác nhận rằng ma trận trọng số đang được cập nhật theo cách cho phép trình phân loại học hỏi từ dữ liệu huấn luyện. Tuy nhiên, dựa trên phần còn lại đầu ra thiết bị cuối cùng, có vẻ như trình phân loại đã phân loại sai một số điểm dữ liệu (<5 trong số chúng):

[INFO] evaluating...				
	precision	recall	f1-score	support
0	1.00	0.99	1.00	250
1	0.99	1.00	1.00	250
avg / total	1.00	1.00	1.00	500

Lưu ý cách mà vòng lặp thứ 0 được phân loại chính xác 100% thời gian, nhưng vòng 2 phân loại chính xác chỉ 99%. Lý do cho sự khác biệt này là do gốc vanilla gradient chỉ thực hiện cập nhật trọng số một lần cho mỗi epoch - trong ví dụ này, đã huấn luyện mô hình cho 100 epoch, do đó chỉ có 100 cập nhật. Tùy thuộc vào việc khởi tạo ma trận trọng số và kích thước tốc độ huấn luyện, có thể không thể huấn luyện được một mô hình có thể phân tách các điểm (mặc dù chúng có thể phân tách tuyến tính).

Trong thực tế, các lần chạy tiếp theo tập lệnh này có thể tiết lộ rằng cả hai lớp có thể được phân loại chính xác 100% thời gian - kết quả phụ thuộc vào các giá trị ban đầu mà W đảm nhận. Để tự xác minh kết quả này, hãy chạy tập lệnh gradient_descent.py nhiều lần.

Đối với việc giảm độ dốc đơn giản, nên huấn luyện nhiều vòng lặp hơn với tốc độ học nhỏ hơn để giúp khắc phục vấn đề này. Tuy nhiên, như sẽ thấy trong phần tiếp theo, một biến thể độ dốc có tên là Stochastic Gradient Descent thực hiện cập nhật trọng số cho mỗi khoảng dữ liệu huấn luyện (có nhiều cập nhật trọng số trên mỗi epoch). Phương pháp này cho sự hội tụ nhanh hơn, ổn định hơn.

3.2.2. Giảm dần độ dốc ngẫu nhiên (SGD)

Trong phần trước, đã thảo luận về Gradient Descent (gradient vanila), thuật toán tối ưu hóa bậc nhất có thể được sử dụng để học một tập các trọng số phân loại cho việc học được tham số hóa. Tuy nhiên, việc triển khai gradient vanilla này có thể bị chậm khi chạy trên các bộ dữ liệu lớn - trên thực tế, nó thậm chí có thể được coi là lãng phí tài nguyên tính toán.

Thay vào đó, nên áp dụng Stochastic Gradient Descent (SGD), một sửa đổi đơn giản cho tính toán thuật toán độ dốc tiêu chuẩn và cập nhật ma trận trọng số W trên các khoảng dữ liệu huấn luyện nhỏ, thay vì toàn bộ tập huấn luyện. Mặc dù việc sửa đổi này dẫn đến việc cập nhật nhiều lần hơn, nó cũng cho phép thực hiện nhiều bước hơn theo độ dốc (một bước trên mỗi khoảng so với một lần trên mỗi epoch), cuối cùng dẫn đến hội tụ nhanh hơn và không ảnh hưởng tiêu cực đến tổn thất độ chính xác và phân loại.

SGD được coi là thuật toán quan trọng nhất khi huấn luyện mạng nơ-ron sâu. Mặc dù SGD đã được giới thiệu hơn 57 năm trước [69], nó vẫn là công cụ cho phép huấn luyện các mạng lớn để học các mẫu từ các điểm dữ liệu.

3.2.2.1. SGD khoảng nhỏ

Xem xét thuật toán độ dốc tiêu chuẩn, rõ ràng là phương thức này sẽ chạy rất chậm trên các tập dữ liệu lớn. Lý do cho sự chậm chạp này là mỗi lần lặp lại độ dốc yêu cầu tính toán một dự đoán cho từng điểm huấn luyện trong bộ dữ liệu huấn luyện trước khi được phép cập nhật ma trận trọng số. Đối với các bộ dữ liệu ảnh như ImageNet, có hơn 1,2 triệu ảnh huấn luyện, việc tính toán này có thể mất nhiều thời gian.

Nó cũng chỉ ra rằng các tính toán dự đoán cho mọi điểm huấn luyện trước khi thực hiện tính ma trận trọng số là lãng phí về mặt tính toán và không giúp được gì cho mô hình.

Thay vào đó, những gì nên làm là cập nhật từng đợt. Có thể cập nhật mã giả để chuyển đổi gốc vanilla để trở thành SGD bằng cách thêm một lệnh gọi hàm bổ sung:

```
1 while True:  
2     batch = next_training_batch(data, 256)  
3     Wgradient = evaluate_gradient(loss, batch, W)  
4     W += -alpha * Wgradient
```

Sự khác biệt duy nhất giữa độ dốc vanilla và SGD là việc bổ sung hàm `next_training_batch`. Thay vì tính toán độ dốc trên toàn bộ tập dữ liệu, thì lấy mẫu dữ liệu, ta có một khoảng. Đánh giá độ dốc trên khoảng đó và cập nhật ma trận trọng số W . Từ góc độ thực hiện, cố gắng chọn ngẫu nhiên các mẫu huấn luyện trước khi áp dụng SGD vì thuật toán rất nhạy với các khoảng giá trị.

Sau khi xem xét về mã giả cho SGD, nhận thấy một tham số mới: kích thước khoảng dữ liệu. Trong một triển khai SGD thuần túy, kích thước khoảng nhỏ sẽ là 1, nghĩ là sẽ lấy mẫu ngẫu nhiên một điểm dữ liệu từ tập huấn luyện, tính toán độ dốc và cập nhật các tham số. Tuy nhiên, thường sử dụng các khoảng nhỏ có kích thước > 1 . Kích thước khoảng điển hình bao gồm 32, 64, 128 và 256.

Vì vậy, tại sao phải sử dụng kích thước khoảng > 1 ? Đầu tiên, kích thước khoảng > 1 giúp giảm phuơng sai trong cập nhật tham số (<http://pyimg.co/pd5w0>), dẫn đến sự hội tụ ổn định hơn. Thứ hai, kích thước khoảng được xác định vì chúng cho phép các thư viện tối ưu hóa đại số tuyến tính bên trong hiệu quả hơn.

Nói chung, kích thước khoảng không phải là siêu tham số, [62]. Nếu sử dụng GPU để huấn luyện mạng nơ-ron, cần xác định có bao nhiêu mẫu huấn luyện sẽ truyền qua GPU và sau đó thay đổi giá trị kích thước bằng mũ hai gần nhất kích thước khoảng(ví dụ: $2^0, 2^1, 2^3\dots$) sao cho khoảng đó phù hợp với GPU. Để huấn luyện CPU, thường sử dụng một trong các kích thước khoảng được liệt kê ở trên để đảm bảo được sự hỗ trợ từ các thư viện tối ưu hóa đại số tuyến tính.

3.2.2.2. Thực hiện SGD khoảng nhỏ

Hãy thực hiện SGD và xem nó khác với độ dốc vanilla tiêu chuẩn như thế nào. Mở một tệp mới, đặt tên là `sgd.py` và chèn đoạn chương trình sau:

```

1 # import the necessary packages
2 from sklearn.model_selection import train_test_split
3 from sklearn.metrics import classification_report
4 from sklearn.datasets import make_blobs
5 import matplotlib.pyplot as plt
6 import numpy as np
7 import argparse
8
9 def sigmoid_activation(x):
10     # compute the sigmoid activation value for a given input
11     return 1.0 / (1 + np.exp(-x))

```

Các dòng 2-7 * nhập các gói Python cần thiết, giống như ví dụ gradient_descent.py trước đó trong chương này. Các dòng 9-11 xác định hàm sigmoid_activation, cũng giống với phiên bản gốc độ dốc.

Trên thực tế, phương pháp dự đoán không thay đổi:

```

13 def predict(X, W):
14     # take the dot product between our features and weight matrix
15     preds = sigmoid_activation(X.dot(W))
16
17     # apply a step function to threshold the outputs to binary
18     # class labels
19     preds[preds <= 0.5] = 0
20     preds[preds > 0] = 1
21
22     # return the predictions
23     return preds

```

Tuy nhiên, điều thay đổi là việc bổ sung hàm next_batch:

```

25 def next_batch(X, y, batchSize):
26     # loop over our dataset 'X' in mini-batches, yielding a tuple of
27     # the current batched data and labels
28     for i in np.arange(0, X.shape[0], batchSize):
29         yield (X[i:i + batchSize], y[i:i + batchSize])

```

Phương thức next_batch yêu cầu ba tham số:

1. X: tập dữ liệu huấn luyện về các vecto đặc trưng / cường độ pixel ảnh thô.

2. y: các nhãn lớp liên quan đến từng điểm dữ liệu huấn luyện.

3. batchSize: kích thước từng khoảng nhỏ sẽ được trả về.

Các dòng 28 và 29 sau đó lặp lại các ví dụ huấn luyện, thu được các tập hợp con cả X và y dưới dạng các khoảng nhỏ.

Tiếp theo, có thể phân tích các đối số dòng lệnh:

```

31 # construct the argument parse and parse the arguments
32 ap = argparse.ArgumentParser()
33 ap.add_argument("-e", "--epochs", type=float, default=100,
34     help="# of epochs")
35 ap.add_argument("-a", "--alpha", type=float, default=0.01,
36     help="learning rate")
37 ap.add_argument("-b", "--batch-size", type=int, default=32,
38     help="size of SGD mini-batches")
39 args = vars(ap.parse_args())

```

Xem xét cả chuyển đổi --epochs (số epoch) và --alpha (tốc độ học) từ ví dụ về độ dốc vanilla - nhưng cũng lưu ý rằng đang giới thiệu một tham số thứ ba: --batch-size, như tên gọi là kích thước từng khoảng nhỏ. Sẽ mặc định giá trị này là 32 điểm dữ liệu trên mỗi khoảng nhỏ.

Khối mã tiếp theo xử lý việc tạo ra vấn đề phân loại 2 lớp với 1.000 điểm dữ liệu, thêm cột vector giá trị sai lệch bias, sau đó thực hiện phân tách huấn luyện và kiểm tra:

```

41 # generate a 2-class classification problem with 1,000 data points,
42 # where each data point is a 2D feature vector
43 (X, y) = make_blobs(n_samples=1000, n_features=2, centers=2,
44     cluster_std=1.5, random_state=1)
45 y = y.reshape((y.shape[0], 1))
46
47 # insert a column of 1's as the last entry in the feature
48 # matrix -- this little trick allows us to treat the bias
49 # as a trainable parameter within the weight matrix
50 X = np.c_[X, np.ones((X.shape[0]))]
51
52 # partition the data into training and testing splits using 50% of
53 # the data for training and the remaining 50% for testing
54 (trainX, testX, trainY, testY) = train_test_split(X, y,
55     test_size=0.5, random_state=42)

```

Sau đó, sẽ khởi tạo ma trận trọng số và tồn thắt giống như trong ví dụ trước:

```

57 # initialize our weight matrix and list of losses
58 print("[INFO] training...")
59 W = np.random.randn(X.shape[1], 1)
60 losses = []

```

Sự thay đổi thực sự diễn ra tiếp theo khi lặp lại số lượng vòng lặp mong muốn, lấy mẫu các khoảng nhỏ:

```

62 # loop over the desired number of epochs
63 for epoch in np.arange(0, args["epochs"]):
64     # initialize the total loss for the epoch
65     epochLoss = []
66
67     # loop over our data in batches
68     for (batchX, batchY) in next_batch(X, y, args["batch_size"]):
69         # take the dot product between our current batch of features

```

```

70     # and the weight matrix, then pass this value through our
71     # activation function
72     preds = sigmoid_activation(batchX.dot(W))

73
74     # now that we have our predictions, we need to determine the
75     # 'error', which is the difference between our predictions
76     # and the true values
77     error = preds - batchY
78     epochLoss.append(np.sum(error ** 2))

```

Trên Dòng 63, bắt đầu lặp qua số lượng --epoch được truy xuất. Sau đó, lặp lại dữ liệu huấn luyện theo từng đợt trên Dòng 68. Đối với mỗi vòng lặp, tính toán nhân chập giữa khoảng và W, sau đó chuyển kết quả qua hàm tác động sigmoid để có được dự đoán. Tính toán sai số bình phương nhỏ nhất cho khoảng trên Dòng 77 và sử dụng giá trị này để cập nhật epochLoss trên Dòng 78.

Bây giờ đã xuất hiện lỗi, có thể tính toán cập nhật độ dốc, là nhân chập giữa các điểm dữ liệu khoảng hiện tại và lỗi trên 1 khoảng:

```

80     # the gradient descent update is the dot product between our
81     # current batch and the error on the batch
82     gradient = batchX.T.dot(error)

83
84     # in the update stage, all we need to do is "nudge" the
85     # weight matrix in the negative direction of the gradient
86     # (hence the term "gradient descent") by taking a small step
87     # towards a set of "more optimal" parameters
88     W += -args["alpha"] * gradient

```

Dòng 88 xử lý cập nhật ma trận trọng số dựa trên độ dốc, được chia tỷ lệ theo hệ số học --alpha. Lưu ý giai đoạn cập nhật trọng số diễn ra bên trong vòng lặp hàng loạt - điều này nghĩa là có nhiều cập nhật trọng số trên mỗi epoch.

Sau đó, có thể cập nhật lịch sử tổn thất bằng cách lấy trung bình trên tất cả các khoảng trong vòng lặp và sau đó hiển thị bản cập nhật cho terminal (của trình biên dịch) nếu cần:

```

90     # update our loss history by taking the average loss across all
91     # batches
92     loss = np.average(epochLoss)
93     losses.append(loss)

94
95     # check to see if an update should be displayed
96     if epoch == 0 or (epoch + 1) % 5 == 0:
97         print("[INFO] epoch={}, loss={:.7f}".format(int(epoch + 1),
98                                         loss))

```

Việc đánh giá trình phân loại được thực hiện theo cách tương tự như trong việc giảm độ dốc vanilla - chỉ cần gọi dự đoán trên dữ liệu testX bằng ma trận trọng số W đã học:

```
100 # evaluate our model  
101 print("[INFO] evaluating...")  
102 preds = predict(testX, W)  
103 print(classification_report(testY, preds))
```

Kết thúc tập lệnh, vẽ dữ liệu phân loại thử nghiệm (test) và cùng với sự tồn thắt trên mỗi epoch:

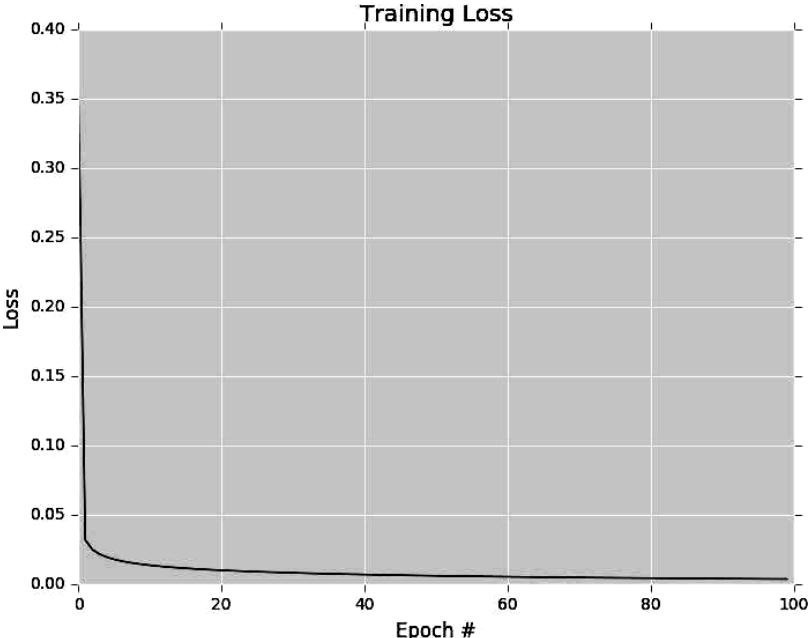
```
105 # plot the (testing) classification data  
106 plt.style.use("ggplot")  
107 plt.figure()  
108 plt.title("Data")  
109 plt.scatter(testX[:, 0], testX[:, 1], marker="o", c=testY, s=30)  
110  
111 # construct a figure that plots the loss over time  
112 plt.style.use("ggplot")  
113 plt.figure()  
114 plt.plot(np.arange(0, args["epochs"]), losses)  
115 plt.title("Training Loss")  
116 plt.xlabel("Epoch #")  
117 plt.ylabel("Loss")  
118 plt.show()
```

3.2.2.3. Kết quả SGD

Để hình dung kết quả từ việc thực nghiệm, chỉ cần thực hiện lệnh sau:

```
$ python sgd.py  
[INFO] training...  
[INFO] epoch=1, loss=0.3701232  
[INFO] epoch=5, loss=0.0195247  
[INFO] epoch=10, loss=0.0142936  
[INFO] epoch=15, loss=0.0118625  
[INFO] epoch=20, loss=0.0103219  
[INFO] epoch=25, loss=0.0092114  
[INFO] epoch=30, loss=0.0083527  
[INFO] epoch=35, loss=0.0076589  
[INFO] epoch=40, loss=0.0070813  
[INFO] epoch=45, loss=0.0065899  
[INFO] epoch=50, loss=0.0061647  
[INFO] epoch=55, loss=0.0057920  
[INFO] epoch=60, loss=0.0054620  
[INFO] epoch=65, loss=0.0051670  
[INFO] epoch=70, loss=0.0049015  
[INFO] epoch=75, loss=0.0046611  
[INFO] epoch=80, loss=0.0044421  
[INFO] epoch=85, loss=0.0042416  
[INFO] epoch=90, loss=0.0040575  
[INFO] epoch=95, loss=0.0038875  
[INFO] epoch=100, loss=0.0037303  
[INFO] evaluating...  
 precision    recall   f1-score   support  
 0          1.00      1.00      1.00      250  
 1          1.00      1.00      1.00      250  
avg / total       1.00      1.00      1.00      50
```

Sử dụng cùng một bộ dữ liệu blobby như trong Hình 3.11 (bên trái) ở trên để phân loại có thể so sánh kết quả SGD với độ dốc tiêu chuẩn vanilla. Hơn nữa, ví dụ SGD sử dụng cùng hệ số học (0,1) và cùng số lượng vòng lặp (100) như độ dốc vanilla. Tuy nhiên, cần lưu ý xem đường cong tốn thất mượt mà hơn như thế nào trong Hình 3.12.



Hình 3.12: Áp dụng Stochastic Gradient Descent cho tập dữ liệu về các điểm dữ liệu đỏ và xanh. Sử dụng SGD, đường cong mượt mà hơn nhiều. Hơn nữa, có thể đạt được một mức độ tốn thất thấp hơn vào cuối vòng lặp thứ 100 (so với tiêu chuẩn, độ dốc vanilla).

Nghiên cứu các giá trị tốn thất thực tế ở cuối vòng lặp thứ 100, sẽ nhận thấy rằng tốn thất thu được từ SGD có bậc của giá trị thấp hơn độ dốc vanilla (tương ứng 0,003 so với 0,021). Sự khác biệt này là do nhiều cập nhật trọng số trên mỗi epoch, giúp mô hình có nhiều cơ hội hơn để học từ các cập nhật được thực hiện cho ma trận trọng số. Hiệu ứng này thậm chí còn rõ rệt hơn trên các bộ dữ liệu lớn, chẳng hạn như ImageNet, nơi có hàng triệu ví dụ huấn luyện và các cập nhật nhỏ, tăng dần trong các tham số có thể dẫn đến giải pháp tốn thất thấp (nhưng không nhất thiết là tối ưu).

3.2.3. Mở rộng thêm về SGD

Có hai phần mở rộng chính mà bạn sẽ gặp trong thực tế. Đầu tiên là động lượng [70], một phương pháp được sử dụng để tăng tốc SGD, cho phép nó học nhanh hơn bằng cách tập trung vào các kích thước có điểm

dốc theo cùng một hướng. Phương pháp thứ hai là gia tốc Nesterov [71], một phần mở rộng cho động lượng tiêu chuẩn.

3.2.3.1. Động lượng

Có thể nêu ví dụ về việc đi từ trên triền đồi xuống chân đồi, khi đi xuống động lượng càng lớn thì xuống càng nhanh hơn.

Động lượng áp dụng cho SGD có tác dụng tương tự - mục tiêu là xây dựng dựa trên cập nhật trọng số tiêu chuẩn để có một tham số động lượng như vậy, từ đó cho phép mô hình có được ổn định thấp hơn (và độ chính xác cao hơn) trong ít vòng lặp hơn. Do đó, thuật ngữ động lượng nên tăng cường độ cập nhật cho các chiều có độ dốc hướng theo cùng một hướng và sau đó giảm độ ảnh hưởng của các cập nhật đối với các chiều mà độ dốc chuyển hướng [70, 72].

Quy tắc cập nhật trọng số trước đây chỉ đơn giản bao gồm việc chia tỷ lệ độ dốc theo hệ số học:

$$W = W - \alpha \nabla w f(W) \quad (3.18)$$

Bây giờ giới thiệu thuật ngữ động lượng V, được chia tỷ lệ bởi γ :

$$V = \gamma \nabla - \alpha \nabla w f(W) \quad (3.19)$$

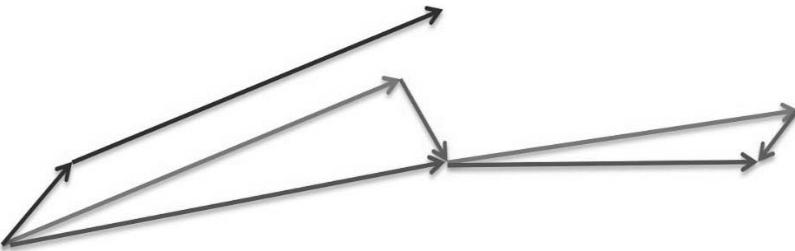
$$W = W + V \quad (3.20)$$

Tham số động lượng thường được đặt bằng 0,9; mặc dù thực tế có thể đặt giá trị bằng 0,5 cho đến khi việc học ổn định và sau đó tăng lên 0,9 - cực kỳ hiếm khi thấy động lượng $< 0,5$. Để xem xét chi tiết hơn về động lượng, vui lòng tham khảo Sutton [73] và Qian [70].

3.2.3.2. Gia tốc Nesterov

Như vấn đề triền dốc lúc trước, nếu có động lực lớn, có thể vượt quá khớp tối thiểu cục bộ và tiếp tục lao xuống. Do đó, sẽ rất thuận lợi khi biết thời điểm nên giảm tốc độ, đó là nơi mà Nesterov tăng tốc độ dốc [71].

Gia tốc Nesterov có thể được khái niệm hóa như một bản cập nhật chính xác cho động lượng cho phép có được một ý tưởng gần đúng về nơi các tham số sẽ ở sau bản cập nhật. Nhìn vào tổng quan của các đường gradient descent trên các khoảng nhỏ [22], có thể quan sát trực quan về gia tốc Nesterov (Hình 3.13).



Hình 3.13: Một mô hình đồ họa gia tốc Nesterov. Đầu tiên, thực hiện một bước nhảy lớn theo hướng gradient trước đó, sau đó đo độ dốc nơi kết thúc và thực hiện chỉnh sửa.

Sử dụng động lượng tiêu chuẩn, tính toán độ dốc (vecto nhỏ màu xanh) và sau đó thực hiện một bước nhảy lớn theo hướng độ dốc (vecto màu xanh lớn). Theo gia tốc Nesterov, trước tiên sẽ thực hiện một bước nhảy lớn theo hướng độ dốc trước đó (vecto màu nâu), đo độ dốc và sau đó thực hiện hiệu chỉnh (vecto màu đỏ) - vecto màu xanh lá cây là bản cập nhật được điều chỉnh cuối cùng bởi gia tốc Nesterov (được diễn giải bởi Ruder [72]).

Lý thuyết và xem xét chi tiết về mặt toán học của gia tốc Nesterov nằm ngoài phạm vi cuốn sách này. Đối với những người thích nghiên cứu gia tốc Nesterov chi tiết hơn, vui lòng tham khảo Ruder [72], Bengio [74] và Sutskever [75].

3.2.3.3. Một số kinh nghiệm

Động lượng là một thuật ngữ quan trọng có thể làm tăng sự hội tụ mô hình; có xu hướng không cần quan tâm về siêu tham số này, so với hệ số học và hạn chế chính quy hóa (được thảo luận trong phần tiếp theo), các nút là vị trí quan trọng nhất để điều chỉnh.

Nguyên tắc là bắt cứ khi nào sử dụng SGD, cũng áp dụng động lượng. Trong hầu hết các trường hợp, có thể đặt nó là 0,9 mặc dù Karpathy [76] đề nghị bắt đầu từ 0,5 và tăng nó lên các giá trị lớn hơn khi các vòng lặp tăng lên.

Đối với tăng tốc Nesterov, nên sử dụng trên các bộ dữ liệu nhỏ hơn, nhưng đối với các bộ dữ liệu lớn (như ImageNet), hầu như luôn không dùng nó. Mặc dù tăng tốc Nesterov có sự đảm bảo về mặt lý thuyết, nhưng tất cả các sản phẩm chính được huấn luyện trên ImageNet (ví dụ, AlexNet [50], VGGNet [57], ResNet [7], Inception [55], v.v.) đều chỉ sử dụng SGD với động lượng - không một bài báo từ nhóm mô hình này sử dụng gia tốc Nesterov.

Kinh nghiệm đã khiến thấy rằng khi huấn luyện các mạng sâu trên các bộ dữ liệu lớn, SGD sẽ dễ dàng hoạt động hơn khi sử dụng động lượng và loại bỏ gia tốc Nesterov. Các bộ dữ liệu nhỏ hơn, mặt khác, có xu hướng tốt hơn với tăng tốc Nesterov. Tuy nhiên, hãy nhớ rằng đây là ý kiến dựa trên kinh nghiệm truyền miệng và có thể thay đổi.

3.2.4. Chính quy hóa

Nhiều chiến lược được sử dụng trong học máy được thiết kế rõ ràng để giảm lỗi kiểm tra, có thể phải trả giá bằng tăng lỗi huấn luyện. Những chiến lược này được gọi chung là chính quy hóa. - - Goodfellow et al. [33]

Trong các phần trước chương này, đã thảo luận về hai hàm tổn thất quan trọng: mất SVM nhiều lớp và mất entropy chéo. Sau đó, đã thảo luận về độ dốc và cách một mạng thực tế có thể học bằng cách cập nhật các tham số trọng số một mô hình. Mặc dù hàm tổn thất cho phép xác định mức độ bô tham số đang thực hiện trên một nhiệm vụ phân loại nhất định, nhưng chính hàm tổn thất không tính đến việc ma trận trọng số trông như thế nào.

Hãy nhớ rằng đang làm việc trong một không gian có giá trị thực, do đó, có một bộ tham số vô hạn sẽ có được độ chính xác phân loại hợp lý trên tập dữ liệu (đối với một số định nghĩa về vụ hợp lý).

Làm thế nào để chọn một tập hợp các tham số giúp đảm bảo mô hình mô tả tốt? Hoặc, ít nhất, làm giảm tác hại của hiện tượng quá khớp. Câu trả lời là chính quy hóa. Thứ hai, chỉ với tốc độ học, chính quy hóa là tham số quan trọng nhất có thể điều chỉnh. Có nhiều loại kỹ thuật chính quy hóa khác nhau, chẳng hạn như chính quy hóa L1, chính quy hóa L2 (thường được gọi là giảm phân trọng số) và Mạng đòn hồi [77], được sử dụng bằng cách cập nhật hàm tổn thất của chính nó, thêm một tham số bổ sung để hạn chế dung lượng mô hình.

Cũng có các loại chính quy hóa có thể được thêm rõ ràng vào kiến trúc mạng

- Dropout là ví dụ chọn lọc việc chính quy hóa như vậy. Sau đó có các hình thức chính quy ngầm được áp dụng trong quá trình huấn luyện. Ví dụ về chính quy hóa ngầm bao gồm tăng dữ liệu và dừng sớm. Trong phần này, chủ yếu tập trung vào chính quy hóa được tham số hóa bằng cách sửa đổi các hàm tổn thất và cập nhật.

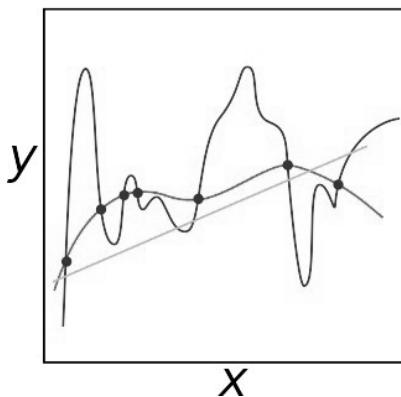
Trong phần mở đầu, sẽ xem xét về dropout và sau đó sẽ thảo luận về việc quá khớp, cũng như cách có thể sử dụng dừng sớm như một công cụ

của chính quy hóa. Các ví dụ về tăng dữ liệu được sử dụng như là chính quy hóa cũng sẽ được trình bày.

3.2.4.1. Chính quy hóa là gì và tại sao cần nó?

Chính quy hóa giúp kiểm soát hiệu suất mô hình, đảm bảo rằng các mô hình tốt hơn trong việc phân loại (chính xác) các điểm dữ liệu mà chúng không được huấn luyện, còn gọi là khả năng khai quát hóa. Nếu không áp dụng chính quy hóa, các trình phân loại có thể dễ dàng trở nên quá phức tạp và không phù hợp với dữ liệu huấn luyện, trong trường hợp đó mất khả năng khai quát hóa dữ liệu thử nghiệm (tập dữ liệu test) và các điểm dữ liệu bên ngoài bộ thử nghiệm (chẳng hạn như ảnh mới trong thế giới thật).

Tuy nhiên, chính quy hóa quá nhiều có thể là một điều không tốt. Có thể gặp rủi ro khi chưa khớp, trong trường hợp đó mô hình hoạt động kém trên dữ liệu huấn luyện và không thể mô hình hóa mối quan hệ giữa dữ liệu đầu vào và nhãn lớp đầu ra (vì hạn chế dung lượng mô hình quá nhiều). Ví dụ, hãy xem xét các điểm sau đây, cùng với các hàm khác nhau phù hợp với các điểm này (Hình 3.14).



Hình 3.14: Một ví dụ về underfitting (đường màu cam), hiện tượng quá khớp (đường màu xanh) và khai quát hóa (đường màu xanh lá cây). Mục tiêu khi xây dựng các trình phân loại học sâu là để có được các loại chức năng màu xanh lá cây này, phù hợp với dữ liệu huấn luyện nhưng tránh tình trạng thừa. Chính quy hóa có thể giúp có được loại phù hợp mong muốn.

Đường màu cam là một ví dụ về chưa khớp - không nắm bắt được mối quan hệ giữa các điểm dữ liệu. Mặt khác, đường màu xanh là một ví dụ về hiện tượng quá khớp - có quá nhiều tham số trong mô hình và đạt tất cả các điểm trong tập dữ liệu, nó cũng thay đổi rất nhiều giữa các điểm.

Nó không phải là một đáp ứng phù hợp. Sau đó, có hàm màu xanh lá cây cũng đạt được tất cả các điểm trong bộ dữ liệu nhưng thực hiện theo cách đơn giản hơn nhiều, có thể dự đoán được.

Mục tiêu của việc chính quy hóa là để có được các loại chức năng màu xanh lá cây này, phù hợp với dữ liệu huấn luyện, nhưng tránh làm hiện tượng quá khớp dữ liệu huấn luyện (màu xanh) hoặc không mô hình hóa được mối quan hệ cơ bản (màu cam). Thảo luận về cách giám sát huấn luyện và phát hiện cả việc thiếu và thừa trong mục 3.4; tuy nhiên, chính quy hóa là một khía cạnh quan trọng trong học máy và sử dụng chính quy hóa để kiểm soát tổng quát hóa mô hình. Để hiểu về chính quy hóa và tác động nó đối với hàm tổn thất và quy tắc cập nhật trọng số, hãy đến với phần tiếp theo.

3.2.4.2. Cập nhật giảm tổn thất và trọng số bao gồm chính quy hóa

Hãy bắt đầu với hàm tổn thất entropy chéo

$$L_i = -\log \left(e^{s_{y_i}} / \sum_j e^{s_j} \right) \quad (3.21)$$

Tổn thất trên toàn bộ tập huấn luyện có thể được viết là:

$$L = \frac{1}{N} \sum_{i=1}^N L_i \quad (3.22)$$

Bây giờ, cần ma trận trọng số W sao cho mọi điểm dữ liệu trong tập huấn luyện được phân loại chính xác, có nghĩa là tổn thất L = 0 cho tất cả L_i.

Độ chính xác đạt được 100% - nhưng hãy đặt nghi vấn với ma trận trọng số này - nó có đồng nhất không? Hay nói cách khác, có những lựa chọn tốt hơn về W sẽ cải thiện khả năng mô hình hóa để khai quát hóa và giảm hiện tượng quá khớp?

Nếu có W như vậy, làm sao biết? Và làm thế nào có thể kết hợp loại hạn chế này vào hàm tổn thất của mình? Câu trả lời là xác định một hạn chế chính quy, một chức năng hoạt động trên ma trận trọng số. Hạn chế chính quy thường được viết dưới dạng hàm, R(W). Phương trình 3.23 dưới đây cho thấy hạn chế chính quy hóa phổ biến nhất, chính quy L2 (còn gọi là phân rã trọng số):

$$R(W) = \sum_i \sum_j W_{i,j}^2 \quad (3.23)$$

Hàm này chính xác làm gì? Trong Python, nó chỉ đơn giản là lấy tổng bình phương trên một mảng:

```

1  penalty = 0
2
3  for i in np.arange(0, W.shape[0]):
4      for j in np.arange(0, W.shape[1]):
5          penalty += (W[i][j] ** 2)

```

Những gì đang làm ở đây là lặp qua tất cả các mục trong ma trận và lấy tổng bình phương. Tổng bình phương trong hạn chế chính quy hóa L2 không khuyến khích các trọng số lớn trong ma trận W. Tại sao lại muốn ngăn cản các giá trị trọng số lớn? Nói tóm lại, bằng cách xử phạt các trọng số lớn, có thể cải thiện khả năng khai quát hóa, và do đó giảm tình trạng hiện tượng quá khớp. Hãy nghĩ về nó theo cách này - giá trị trọng số càng lớn thì ảnh hưởng của nó đến dự đoán đầu ra càng nhiều. Kích thước với giá trị trọng số lớn hơn có thể tự điều khiển dự đoán đầu ra bộ phân loại (tất nhiên với điều kiện giá trị trọng số đủ lớn), điều này sẽ dẫn đến quá khớp.

Để giảm thiểu ảnh hưởng đến các kích thước khác nhau đối với các phân loại đầu ra, áp dụng chính quy hóa, từ đó tìm kiếm các giá trị W có tính đến tất cả các kích thước thay vì các kích thước có giá trị lớn. Trong thực tế, có thể thấy rằng chính quy hóa làm giảm độ chính xác huấn luyện một chút nhưng thực sự làm tăng độ chính xác kiểm tra (trên tập dữ liệu testing).

Một lần nữa, hàm tổn thất có dạng cơ bản giống nhau, chỉ là bây giờ thêm vào hệ số chính quy:

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W) \quad (3.24)$$

Thuật ngữ đầu tiên đã thấy trước đây - đó là tổn thất trung bình trên tất cả các mẫu trong tập huấn luyện.

Thuật ngữ thứ hai là mới - đây là hạn chế chính quy. Biến là một siêu tham số kiểm soát số lượng hoặc cường độ chính quy hóa đang áp dụng. Trong thực tế, cả tốc độ học α và thuật ngữ chính quy λ đều là các siêu tham số phải mất nhiều thời gian nhất để điều chỉnh.

Mở rộng tổn thất entropy chéo để bao gồm chính quy L2 mang lại phương trình sau:

$$L = \frac{1}{N} \sum_{i=1}^N \left[-\log \left(e^{s_{y_i}} / \sum_j e^{s_j} \right) \right] + \lambda \sum_i \sum_j W_{i,j}^2 \quad (3.25)$$

Cũng có thể mở rộng mảng nhiều lớp SVM:

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \left[\max(0, s_j - s_{y_i} + 1) \right] + \lambda \sum_i \sum_j w_{i,j}^2 \quad (3.26)$$

Bây giờ, hãy xem quy tắc cập nhật trọng số tiêu chuẩn:

$$W = W - \alpha \nabla wf(W) \quad (3.27)$$

Phương pháp này cập nhật các trọng số dựa trên nhiều độ dốc theo hệ số học α . Nếu có tính đền chính quy, quy tắc cập nhật trọng số trở thành:

$$W = W - \alpha \nabla wf(W) + \lambda R(W) \quad (3.28)$$

Ở đây, đang thêm một thuật ngữ tuyến tính âm vào độ dốc (tức là, độ dốc giảm dần), cản trở các trọng số lớn, với mục tiêu cuối cùng là giúp mô hình dễ dàng khái quát hóa hơn.

3.2.4.3. Các loại kỹ thuật chính quy

Nói chung, sẽ thấy ba loại chính quy phổ biến được áp dụng trực tiếp cho hàm tổn thất. Đầu tiên, đã xem xét trước đó, chính quy L2 (hay còn gọi là phân rã trọng số)

$$R(W) = \sum_i \sum_j W_{i,j}^2 \quad (3.29)$$

Cũng có chính quy L1 lấy giá trị tuyệt đối thay vì bình phương:

$$R(W) = \sum_i \sum_j |W_{i,j}| \quad (3.30)$$

Chuẩn hóa mạng đòn hồi [98] tìm cách kết hợp cả chuẩn hóa L1 và L2:

$$R(W) = \sum_i \sum_j \beta W_{i,j}^2 + |W_{i,j}| \quad (3.31)$$

Các loại phương thức chính quy khác tồn tại như sửa đổi trực tiếp kiến trúc mạng cùng với phương thức mà mạng được huấn luyện thực sự sẽ được xem xét trong các chương sau.

Về mặt phương pháp chính quy nên sử dụng, nên coi lựa chọn này là một siêu tham số cần tối ưu hóa và thực hiện các thử nghiệm để xác định xem có nên áp dụng chính quy hóa không, và nếu vậy thì nên áp dụng phương pháp chính quy nào, và giá trị thích hợp λ là gì. Để biết thêm chi tiết về chính quy hóa, hãy tham khảo Chương 7 Goodfellow et al. [33], phần Chính quy hóa trực tuyến từ hướng dẫn DeepLearn.net [78] và các ghi chú từ bài giảng Karpathy tựa cs231n Neural Networks II [79].

3.2.4.4. Chính quy hóa áp dụng cho phân loại ảnh

Để minh họa cho chính quy hóa trong, viết một số đoạn Python để áp dụng bộ dữ liệu động vật. Mở một tập tin mới, đặt tên cho nó là normalization.py và chèn đoạn chương sau:

```
1 # import the necessary packages
2 from sklearn.linear_model import SGDClassifier
3 from sklearn.preprocessing import LabelEncoder
4 from sklearn.model_selection import train_test_split
5 from pyimagesearch.preprocessing import SimplePreprocessor
6 from pyimagesearch.datasets import SimpleDatasetLoader
7 from imutils import paths
8 import argparse
```

Dòng 2-8 nhập các gói Python cần thiết. Đã biết tất cả các hàng thư viện này trước đây, ngoại trừ SGDClassifier. Như tên lớp này cho thấy, việc triển khai này gói gọn tất cả các khái niệm đã xem xét trong chương này, bao gồm:

- Hàm tồn thât
- Số vòng lặp
- Tốc độ học
- Quy ước chính quy

Tiếp theo, có thể phân tích các đối số dòng lệnh và lấy danh sách các ảnh từ ổ cứng:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-d", "--dataset", required=True,
13     help="path to input dataset")
14 args = vars(ap.parse_args())
15
16 # grab the list of image paths
17 print("[INFO] loading images...")
18 imagePaths = list(paths.list_images(args["dataset"]))
```

Đưa ra các đường dẫn ảnh, sẽ thay đổi kích thước chúng thành 32x32 pixel, tải chúng từ ổ cứng vào bộ nhớ và sau đó biến đổi chúng thành một mảng 3.072:

```
20 # initialize the image preprocessor, load the dataset from disk,
21 # and reshape the data matrix
22 sp = SimplePreprocessor(32, 32)
23 sdl = SimpleDatasetLoader(preprocessors=[sp])
24 (data, labels) = sdl.load(imagePaths, verbose=500)
25 data = data.reshape((data.shape[0], 3072))
```

Cũng sẽ mã hóa các nhãn dưới dạng số nguyên và thực hiện phân

tách huấn luyện- thử nghiệm (training-testing), sử dụng 75% dữ liệu cho huấn luyện và 25% còn lại để thử nghiệm:

```
27 # encode the labels as integers
28 le = LabelEncoder()
29 labels = le.fit_transform(labels)
30
31 # partition the data into training and testing splits using 75% of
32 # the data for training and the remaining 25% for testing
33 (trainX, testX, trainY, testY) = train_test_split(data, labels,
34 test_size=0.25, random_state=5)
```

Hãy để L áp dụng một vài loại phương pháp chính quy hóa khác nhau khi huấn luyện Trình phân loại SGD:

```
36 # loop over our set of regularizers
37 for r in (None, "l1", "l2"):
38     # train a SGD classifier using a softmax loss function and the
39     # specified regularization function for 10 epochs
40     print("[INFO] training model with '{} penalty'.format(r))
41     model = SGDClassifier(loss="log", penalty=r, max_iter=10,
42         learning_rate="constant", eta0=0.01, random_state=42)
43     model.fit(trainX, trainY)
44
45     # evaluate the classifier
46     acc = model.score(testX, testY)
47     print("[INFO] '{} penalty accuracy: {:.2f}%".format(r,
48             acc * 100))
```

Dòng 37 vòng trên bộ thường xuyên, bao gồm cả không chính quy. Sau đó, khởi tạo và huấn luyện SGDClassifier trên dòng 41-43.

Sử dụng tốn thất entropy chéo, với hạn chế chính quy là r và mặc định λ là 0,0001. Sử dụng SGD để huấn luyện mô hình trong 10 vòng lặp với hệ số học là $\alpha = 0,01$. Sau đó, đánh giá bộ phân loại và hiển thị kết quả chính xác cho màn hình trên dòng 46-48.

Để xem mô hình SGD được huấn luyện với các loại chính quy khác nhau, chỉ cần thực hiện lệnh sau:

```
$ python regularization.py --dataset ../datasets/animals
[INFO] loading images...
...
[INFO] training model with 'None' penalty
[INFO] 'None' penalty accuracy: 50.40%
[INFO] training model with 'l1' penalty
[INFO] 'l1' penalty accuracy: 52.53%
[INFO] training model with 'l2' penalty
[INFO] 'l2' penalty accuracy: 55.07%
```

Có thể thấy mà không cần chính quy, có được độ chính xác 50,40%. Sử dụng chính quy L1, độ chính xác tăng lên 52,53%. Chuẩn hóa L2 đạt độ chính xác cao nhất là 55,07%.

Trên thực tế, ví dụ này quá nhỏ để thể hiện tất cả các lợi thế của việc áp dụng phương pháp chính quy hóa - vì vậy, sẽ phải đợi cho đến khi bắt đầu huấn luyện mạng nơ-ron tích chập. Tuy nhiên, lợi ích mà việc chính quy hóa có thể mang lại là sự cải thiện về độ chính xác trên tập dữ liệu kiểm tra và giảm tinh trạng overfitting bằng cách chỉ cần điều chỉnh các siêu tham số một cách tức thì.

3.2.5. Tóm tắt

Trong mục này, đã giới thiệu về việc học sâu và đi sâu vào phần cốt lõi của mô hình: thuật toán suy giảm độ dốc (gradient descent) và đã nghiên cứu hai loại độ dốc gốc:

1. Tiêu chuẩn vanilla.
2. Phiên bản ngẫu nhiên được sử dụng phổ biến hơn.

Độ dốc vanilla chỉ thực hiện một cập nhật trọng số trên mỗi chu kỳ (epoch) nên rất chậm hối tụ trên các tập dữ liệu lớn. Thay vào đó, phiên bản ngẫu nhiên áp dụng nhiều cập nhật trọng số cho mỗi chu kỳ bằng cách tính toán độ dốc trên các khoảng nhỏ. Bằng cách sử dụng SGD, có thể giảm đáng kể thời gian để huấn luyện một mô hình, giảm tổn thất và có độ chính xác cao hơn. Kích thước khoảng điển hình bao gồm 32, 64, 128 và 256.

Các thuật toán Gradient Descent được điều khiển thông qua tốc độ học: đây là thông số quan trọng nhất để điều chỉnh chính xác khi huấn luyện các mô hình.

Nếu tốc độ học quá lớn, sẽ chỉ đơn giản là xoay quanh tổn thất toàn phần và thực tế không học bất kỳ mẫu nào từ dữ liệu. Mặt khác, nếu hệ số học quá nhỏ, sẽ cần một số lần lặp chính xác để đạt được ngay một tổn thất hợp lý. Để tìm ra hệ số đúng, sẽ phải tốn nhiều thời gian để điều chỉnh hệ số học.

Chúng ta đã xem xét về chính quy hóa, mà nó làm tăng độ chính xác trên tập dữ liệu kiểm tra và trả giá bằng độ chính xác trên tập huấn luyện. Chính quy hóa bao gồm một loạt các kỹ thuật. đặc biệt tập trung vào các phương pháp chính quy được áp dụng cho các hàm tổn thất và quy tắc cập nhật trọng số, bao gồm chính quy L1, chính quy L2 và Elastic Net.

Về mặt học sâu và mạng nơ-ron, sẽ thấy chính quy hóa L2 được sử dụng để phân loại ảnh – thủ thuật là điều chỉnh tham số lớn để bao gồm đúng số lượng chính quy.

Trong mục tiếp theo, sẽ thảo luận về các mạng nơ-ron, thuật toán

backpropagation và cách huấn luyện các mạng nơ-ron trên các bộ dữ liệu tùy chỉnh.

3.3. LẬP LỊCH BIỂU TỶ LỆ HỌC

Tác giả đã huấn luyện kiến trúc MiniVGGNet trên bộ dữ liệu CIFAR-10. Để giúp giảm bớt ảnh hưởng việc hiện tượng quá khớp, tác giả đã đưa ra khái niệm thêm hệ số phân rã vào tỷ lệ học khi áp dụng SGD để huấn luyện mạng.

Trong mục này, sẽ thảo luận về khái niệm lịch trình tỷ lệ học, đôi khi được gọi là tốc độ học hoặc tỷ lệ học thích nghi. Bằng cách điều chỉnh tốc độ học trên cơ sở chu kỳ, có thể giảm tổn thất, tăng độ chính xác và thậm chí trong một số trường hợp nhất định sẽ giảm tổng thời gian cần thiết để huấn luyện một mạng.

3.3.1. Giảm tỷ lệ học

Lịch trình tỷ lệ học đơn giản và chậm chạp nhất là lịch trình giảm dần tỷ lệ học theo thời gian. Để xem xét lý do tại sao lịch biểu tỷ lệ học là một phương pháp thú vị để áp dụng để giúp tăng độ chính xác mô hình, hãy xem xét công thức cập nhật trọng số tiêu chuẩn từ mục 3.2.1.5:

```
W += -args["alpha"] * gradient
```

Lưu ý rằng tốc độ học α kiểm soát “bước” mà thực hiện dọc theo gradient. Các giá trị lớn hơn α ngụ ý rằng đang thực hiện các bước lớn hơn, trong khi các giá trị nhỏ hơn α sẽ tạo ra các bước nhỏ - nếu α bằng 0, mạng không thể thực hiện bất kỳ bước nào cả (vì gradient nhân với 0 là 0).

Trong các ví dụ trước trong cuốn sách này, tỷ lệ học là không đổi và thường đặt $\alpha = \{0.1, 0.01\}$ và sau đó huấn luyện mạng cho một số chu kỳ cố định mà không thay đổi tốc độ học. Phương pháp này có thể hoạt động tốt trong một số trường hợp, nhưng nó thường có lợi để giảm tỷ lệ học theo thời gian.

Khi huấn luyện mạng, tác giả đang cố gắng tìm một số vị trí dọc theo khu vực tổn thất, nơi mạng có được độ chính xác hợp lý. Nó không phải là một cực tiểu toàn cục hoặc thậm chí là một cực tiểu cục bộ nhưng trong thực tế, chỉ cần tìm một khu vực vùng tổn thất với mức tổn thất thấp là khá tốt.

Nếu liên tục giữ tỷ lệ học cao, có thể vượt qua các khu vực tổn thất thấp này vì sẽ thực hiện các bước quá lớn để đi xuống các khu vực này. Thay vào đó, những gì có thể làm là giảm tốc độ học, do đó cho phép mạng thực hiện các bước nhỏ hơn - tốc độ giảm này cho phép mạng đi xuống các khu vực tổn thất “tối ưu hơn” và nếu không sẽ bỏ qua hoàn toàn bởi tỷ lệ học lớn hơn.

Do đó, có thể xem quá trình lập tỷ lệ học như sau:

1. Tìm một tập hợp các trọng số tối ưu ngay từ đầu trong quá trình huấn luyện với tỷ lệ học cao hơn.

2. Điều chỉnh các trọng số này sau quá trình huấn luyện để tìm các trọng số tối ưu hơn bằng cách sử dụng tỷ lệ học nhỏ hơn.

Có hai loại trình lập lịch biểu tỷ lệ học chính có thể gặp:

1. Bộ lập biểu tỷ lệ học giảm dần dựa trên số epoch (như hàm tuyến tính, đa thức hoặc hàm mũ).

2. Lập lịch biểu tỷ lệ học giảm dựa trên chu kỳ cụ thể (chẳng hạn như hàm từng phần). Tác giả sẽ xem xét cả hai loại lập lịch biểu tỷ lệ học trong chương này.

3.3.1.1. Lịch trình phân rã tiêu chuẩn trong KERAS

Thư viện Keras hỗ trợ với bộ lập lịch tốc độ học dựa trên thời gian - nó được điều khiển thông qua tham số phân rã (decay) các lớp tối ưu hóa (như SGD).

Quay trở lại mục 2.5 phía trước, hãy xem đoạn chương trình khởi tạo SGD và MiniVGGNet:

```
37 # initialize the optimizer and model
38 print("[INFO] compiling model...")
39 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
40 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
41 model.compile(loss="categorical_crossentropy", optimizer=opt,
42 metrics=["accuracy"])
```

Ở đây, khởi tạo trình tối ưu hóa SGD với tốc độ học là $\alpha = 0,01$, động lượng $\gamma = 0,9$ và cho biết rằng đang sử dụng gradient tăng tốc Nesterov. Sau đó, đặt phân rã (γ) thành tỷ lệ học chia cho tổng số chu kỳ đang huấn luyện mạng (một quy tắc chung), kết quả là $0,01/40 = 0,00025$.

Keras áp dụng lịch trình tỷ lệ học sau đây để điều chỉnh tỷ lệ học sau mỗi chu kỳ:

$$\alpha_{e+1} = \alpha_e \times 1 / (1 + \gamma * e) \quad (3.32)$$

Nếu đặt tham số phân rã thành 0 (giá trị mặc định trong các trình tối ưu hóa Keras), tác giả nhận thấy không có ảnh hưởng đến tốc độ học (ở đây tác giả đặt epoch hiện tại là $e = 1$ để chứng minh điều này):

$$\alpha_{e+1} = 0.01 \times 1 / (1 + 0.0 * 1) = 0.01 \quad (3.33)$$

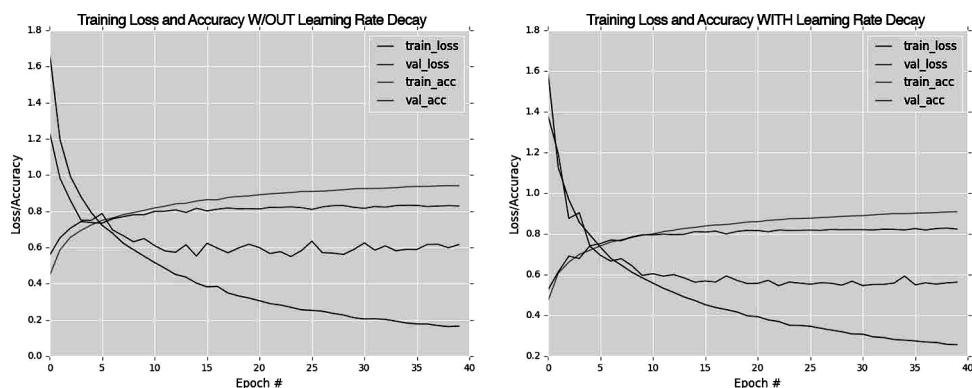
Nhưng nếu thay vì sử dụng $\text{decay} = 0,01/40$, nhận thấy rằng tốc độ học bắt đầu giảm sau mỗi chu kỳ (Bảng 3.1).

Sử dụng tham số phân rã tỷ lệ học dựa trên thời gian này, mô hình MiniVGGNet đã đạt được độ chính xác phân loại 83%, như trong mục 3.5. Tác giả khuyến khích đặt tham số phân rã decay = 0 trong trình tối ưu hóa SGD và sau đó chạy lại thực nghiệm. có thể nhận thấy rằng mạng cũng đạt được độ chính xác phân loại 83%; tuy nhiên, bằng cách xem xét các sơ đồ của hai mô hình trong Hình 3.15, sẽ nhận thấy rằng hiện tượng quá khớp đang bắt đầu xảy ra khi validation loss tăng lên qua chu kỳ 25 (bên trái).

Bảng 3.1: Bảng biểu thị mức độ giám học theo thời gian bằng cách sử dụng 40 epoch, tỷ lệ học ban đầu là $\alpha = 0,01$ và thời hạn phân rã là $0,04 / 40$

Epoch	Learning Rate (α)
1	0.01
2	0.00990
3	0.00971
...	...
38	0.00685
39	0.00678
40	0.00672

Kết quả này trái ngược với khi đặt tham số phân rã = $0,01 / 40$ (phải) và có được một kết quả học tốt hơn nhiều (và chưa kể, độ chính xác cao hơn). Bằng cách sử dụng tham số phân rã tỷ lệ học, thường không chỉ có thể cải thiện độ chính xác phân loại, mà còn giảm bớt ảnh hưởng của hiện tượng quá khớp, do đó làm tăng hiệu suất khai quật mô hình.



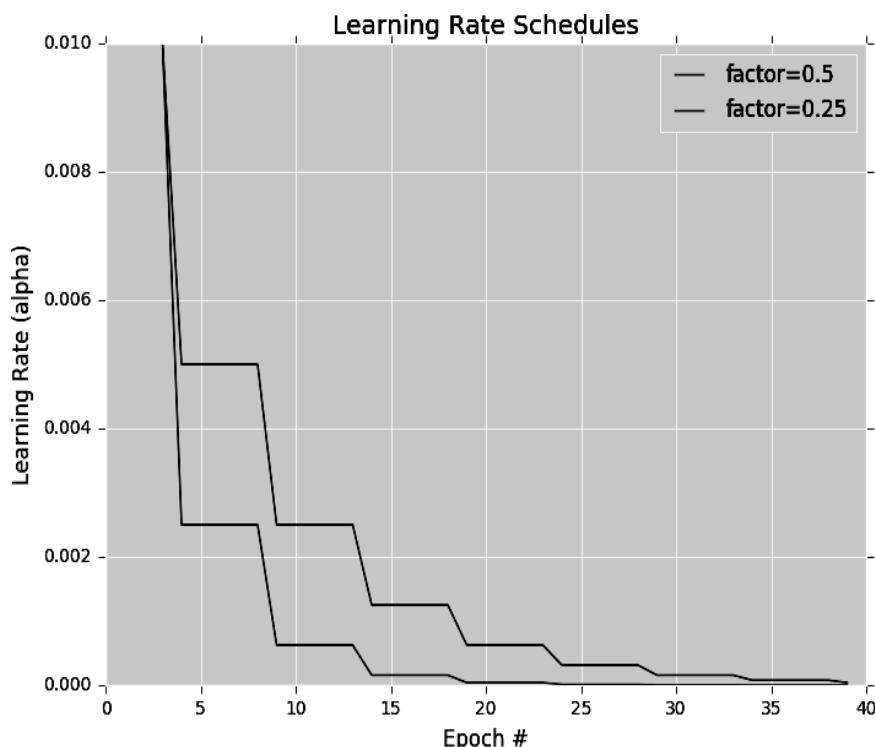
Hình 3.15: Trái: Huấn luyện MiniVGGNet trên CIFAR-10 mà không bị suy giảm tốc độ học. Lưu ý cách tốn thát bắt đầu tăng qua chu kỳ 25, cho thấy rằng quá khớp đang xảy ra. Phải: Áp dụng hệ số phân rã $0,01 / 40$. Điều này làm giảm tỷ lệ học theo thời gian, giúp giảm bớt ảnh hưởng của quá khớp.

3.3.1.2. Phân rã theo bước

Một lịch trình tỷ lệ học phổ biến khác là phân rã theo từng bước, trong đó giảm tỷ lệ học một cách có hệ thống sau các chu kỳ cụ thể trong quá trình huấn luyện. Các trình lập lịch biểu tỷ lệ phân rã bước có thể được coi là một hàm từng phần, như trong Hình 3.16. Ở đây, tốc độ học là không đổi đối với một số chu kỳ (epoch), sau đó giảm xuống và không đổi một lần nữa, sau đó lại giảm xuống, v.v.

Khi áp dụng phân rã bước cho tỷ lệ học sẽ có 2 lựa chọn:

1. Xác định một phương trình mô hình hóa việc giảm tỷ lệ học mà mong muốn đạt được.
2. Sử dụng `ctrl + c` để huấn luyện một mạng học sâu, nơi huấn luyện một số lượng chu kỳ với tốc độ học nhất định cuối cùng, hiệu suất xác thực đã bị đình trệ, sau đó `ctrl + c` để dừng tập lệnh, điều chỉnh tốc độ học và tiếp tục huấn luyện.



Hình 3.16: Một ví dụ về hai lịch biểu tỷ lệ học giảm tỷ lệ học theo kiểu từng phần. Giảm giá trị hệ số làm tăng tốc độ giảm. Trong mỗi trường hợp, tỷ lệ học tiến gần đến 0 ở chu kỳ cuối cùng.

Tác giả chủ yếu tập trung vào việc giảm từng phần dựa trên phương trình để lập kế hoạch học trong chương này. Phương pháp $\text{ctrl} + c$ tiên tiến hơn và thường được áp dụng cho các bộ dữ liệu lớn hơn bằng cách sử dụng các mạng nơ-ron sâu hơn trong đó số lượng chính xác cần thiết để có được độ chính xác hợp lý là không xác định.

Khi áp dụng phân rã bước, thường giảm tốc độ học xuống (1) một nửa hoặc (2) một thứ tự cường độ sau mỗi số chu kỳ cố định. Ví dụ: giả sử, giả sử tỷ lệ học ban đầu là $\alpha = 0,1$. Sau 10 chu kỳ, giảm tỷ lệ học xuống $\alpha = 0,05$. Sau 10 chu kỳ huấn luyện khác (tức là tổng số chu kỳ thứ 20), α lại giảm 0,5 lần nữa, sao cho $\alpha = 0,025$, v.v. Trên thực tế, đây là lịch biểu tỷ lệ học chính xác được vẽ trong Hình 3.16 ở trên (đường màu đỏ). Đường màu xanh hiển thị mức giảm mạnh hơn nhiều với hệ số 0,25.

3.3.1.3. Thực hiện tùy chỉnh tỷ lệ học trong Keras

Một cách thuận tiện, thư viện Keras cung cấp lớp LearningRateScheduler cho phép xác định hàm tỷ lệ học tùy chỉnh và sau đó tự động áp dụng trong quá trình huấn luyện. Hàm này sẽ lấy số chu kỳ (epoch) làm đối số và sau đó tính tỷ lệ học mong muốn dựa trên hàm đã xác định.

Trong ví dụ này, tác giả sẽ định nghĩa một hàm từng phần sẽ giảm tốc độ học bằng một yếu tố F nhất định sau mỗi chu kỳ D . Phương trình sẽ là

$$\alpha_{E+1} = \alpha_I \times F^{(1+E)/D} \quad (3.34)$$

Trong đó α_I là tốc độ học ban đầu, F là giá trị kiểm soát tốc độ học giảm xuống, D là mức giảm mỗi giá trị epochs và E là chu kỳ hiện tại. Yếu tố F càng lớn thì tốc độ học sẽ càng chậm. Ngược lại, hệ số F càng nhỏ thì tốc độ học sẽ giảm càng nhanh.

Được viết bằng ngôn ngữ Python, phương trình này có thể được thể hiện dưới dạng:

```
alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
```

Hãy thực hiện tỷ lệ học tùy chỉnh này và sau đó áp dụng nó cho MiniVG- GNet trên CIFAR-10. Mở một tệp mới, đặt tên nó là cifar10_lr_decay.py và lập trình:

```
1 # set the matplotlib backend so figures can be saved in the background
2 import matplotlib
3 matplotlib.use("Agg")
4
5 # import the necessary packages
6 from sklearn.preprocessing import LabelBinarizer
7 from sklearn.metrics import classification_report
8 from pyimagesearch.nn.conv import MiniVGGNet
9 from keras.callbacks import LearningRateScheduler
10 from keras.optimizers import SGD
11 from keras.datasets import cifar10
12 import matplotlib.pyplot as plt
13 import numpy as np
14 import argparse
```

Các dòng 2-14 nhập các gói Python cần thiết như trong tập lệnh minivggnet_cifar10.py ban đầu Mục 2.5. Tuy nhiên, hãy lưu ý đến Dòng 9 nơi nhập LearningRateScheduler từ thư viện Keras - lớp này sẽ cho phép xác định tỷ lệ học tùy chỉnh lập lịch biều.

Nói về tỷ lệ học tùy chỉnh:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.25
21     dropEvery = 5
22
23     # compute learning rate for the current epoch
24     alpha = initAlpha * (factor ** np.floor((1 + epoch) / dropEvery))
25
26     # return the learning rate
27     return float(alpha)
```

Dòng 16 định nghĩa hàm step_decay chấp nhận một tham số bắt buộc duy nhất - epoch thuê. Sau đó, xác định tỷ lệ học ban đầu (0,01), hệ số bỏ-drop (0,25), đặt dropEvery = 5, nhớ rằng sẽ giảm tỷ lệ học xuống 0,25 mỗi năm chu kỳ (dòng 19-21).

Tính toán tỷ lệ học mới cho chu kỳ hiện tại trên Dòng 24 bằng cách sử dụng phương trình 3.16 ở trên. Tốc độ học mới này được trả về cho hàm gọi trên Dòng 27, cho phép Keras cập nhật nội bộ tốc độ học tối ưu hóa.

Từ đây có thể tiếp tục lập trình:

```
29 # construct the argument parse and parse the arguments
30 ap = argparse.ArgumentParser()
31 ap.add_argument("-o", "--output", required=True,
32                 help="path to the output loss/accuracy plot")
33 args = vars(ap.parse_args())
34
35 # load the training and testing data, then scale it into the
36 # range [0, 1]
37 print("[INFO] loading CIFAR-10 data...")
38 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
39 trainX = trainX.astype("float") / 255.0
40 testX = testX.astype("float") / 255.0
41
42 # convert the labels from integers to vectors
43 lb = LabelBinarizer()
44 trainY = lb.fit_transform(trainY)
45 testY = lb.transform(testY)
46
47 # initialize the label names for the CIFAR-10 dataset
48 labelNames = ["airplane", "automobile", "bird", "cat", "deer",
49               "dog", "frog", "horse", "ship", "truck"]
```

Các dòng 30-33 phân tích các đối số dòng lệnh. Sau đó, tải bộ dữ liệu CIFAR-10 từ ổ cứng và chia tỷ lệ cường độ pixel thành phạm vi [0, 1] ở dòng 37-40. Các dòng 43-45 xử lý gán nhãn

Tiếp theo, huấn luyện mạng:

```
51 # define the set of callbacks to be passed to the model during
52 # training
53 callbacks = [LearningRateScheduler(step_decay)]
54
55 # initialize the optimizer and model
56 opt = SGD(lr=0.01, momentum=0.9, nesterov=True)
57 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
58 model.compile(loss="categorical_crossentropy", optimizer=opt,
59                 metrics=["accuracy"])
60
61 # train the network
62 H = model.fit(trainX, trainY, validation_data=(testX, testY),
63                batch_size=64, epochs=40, callbacks=callbacks, verbose=1)
```

Dòng 53 rất quan trọng vì nó khởi tạo danh sách các lệnh gọi lại. Tùy thuộc vào cách xác định lệnh gọi lại, Keras sẽ gọi hàm này vào

lúc bắt đầu hoặc kết thúc mỗi epoch, v.v. LearningRateScheduler sẽ gọi step_decay vào cuối mỗi epoch, cho phép cập nhật việc học trước chu kỳ tiếp theo bắt đầu.

Dòng 56 khởi tạo trình tối ưu hóa SGD với động lượng (momentum) 0,9 và gradient tăng tốc Nesterov. Tham số lr sẽ bị bỏ qua ở đây vì sẽ sử dụng hàm gọi lại LearningRateScheduler để có thể loại bỏ hoàn toàn tham số này; tuy nhiên, cần đưa nó vào và để nó khớp với initAlpha.

Dòng 57 khởi tạo MiniVGGNet, sau đó sẽ huấn luyện 40 epoch trên các dòng 62 và 63. Sau khi mạng được huấn luyện, có thể đánh giá:

```
65 # evaluate the network
66 print("[INFO] evaluating network...")
67 predictions = model.predict(testX, batch_size=64)
68 print(classification_report(testY.argmax(axis=1),
69     predictions.argmax(axis=1), target_names=labelNames))
```

Hiển thị tỷ lệ sai số và độ chính xác:

```
71 # plot the training loss and accuracy
72 plt.style.use("ggplot")
73 plt.figure()
74 plt.plot(np.arange(0, 40), H.history["loss"], label="train_loss")
75 plt.plot(np.arange(0, 40), H.history["val_loss"], label="val_loss")
76 plt.plot(np.arange(0, 40), H.history["acc"], label="train_acc")
77 plt.plot(np.arange(0, 40), H.history["val_acc"], label="val_acc")
78 plt.title("Training Loss and Accuracy on CIFAR-10")
79 plt.xlabel("Epoch #")
80 plt.ylabel("Loss/Accuracy")
81 plt.legend()
82 plt.savefig(args["output"])
```

Để đánh giá hiệu quả hệ số bỏ học (drop) đối với việc lập lịch tốc độ học và độ chính xác phân loại mạng tổng thể, sẽ đánh giá hai yếu tố giảm: 0,25 và 0,5, tương ứng. Hệ số giảm 0,25 sẽ giảm nhanh hơn đáng kể so với tỷ lệ 0,5.

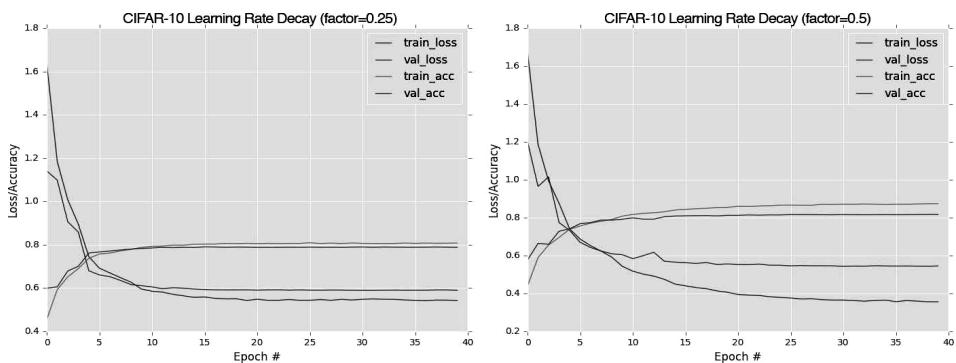
Một lần nữa, hãy chú ý rằng hệ số giảm 0,25 nhanh hơn làm giảm tốc độ học. Tác giả sẽ tiếp tục và đánh giá mức giảm tốc độ học nhanh hơn 0,25 (Dòng 20) - để thực thi tập lệnh, chỉ cần đưa ra lệnh sau:

```

$python cifar10 lr_decay.py --output output/lr_decay f0.25 plot.png
[INFO] loading CIFAR-10 data...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
34s - loss: 1.6380 - acc: 0.4550 - val_loss: 1.1413 - val_acc: 0.5993
Epoch 2/40
34s - loss: 1.1847 - acc: 0.5925 - val_loss: 1.0986 - val_acc: 0.6057
...
Epoch 40/40
34s - loss: 0.5423 - acc: 0.8081 - val_loss: 0.5899 - val_acc: 0.7885
[INFO] evaluating network...
      precision    recall   f1-score   support
airplane        0.81      0.81      0.81      1000
automobile      0.91      0.89      0.90      1000
bird           0.71      0.65      0.68      1000
cat            0.63      0.60      0.62      1000
deer           0.72      0.79      0.75      1000
dog            0.70      0.67      0.68      1000
frog           0.80      0.88      0.84      1000
horse          0.86      0.83      0.84      1000
ship           0.87      0.90      0.88      1000
truck          0.87      0.87      0.87      1000
avg / total     0.79      0.79      0.79     10000

```

Ở đây thấy rằng mạng chỉ đạt được độ chính xác phân loại 79%. Tốc độ học đang giảm khá mạnh - sau chu kỳ 15 chỉ là 0,00125, có nghĩa là mạng đang thực hiện các bước rất nhỏ dọc theo bối cảnh tồn thắt. Hành vi này có thể được nhìn thấy trong Hình 3.17 (bên trái), trong đó tồn thắt xác thực và độ chính xác về cơ bản đã bị định trệ sau 15 chu kỳ.



Hình 3.17: Trái: Biểu diễn độ chính xác / tồn thắt mạng bằng cách giảm tốc độ học nhanh hơn với hệ số 0,25. Lưu ý cách mà tồn thắt / độ chính xác bị định trệ trong 15 chu kỳ trước khi mà tỷ lệ học quá nhỏ. Phải: Độ chính xác / tồn thắt mạng với tốc độ học giảm chậm. Lần này, mạng có thể tiếp tục học 30 chu kỳ trước cho đến khi định trệ xảy ra.

Thay vào đó, nếu thay đổi hệ số giảm 0,5 bằng cách đặt hệ số (factor) = 0,5 bên trong step_decay:

```
16 def step_decay(epoch):
17     # initialize the base initial learning rate, drop factor, and
18     # epochs to drop every
19     initAlpha = 0.01
20     factor = 0.5
21     dropEvery = 5
```

Và sau đó chạy lại thực nghiệm có được độ chính xác phân loại cao hơn:

Lần này, với tốc độ học giảm chậm, có được độ chính xác 82%. Nhìn vào Hình 3.17 (bên phải), có thể thấy rằng mạng tiếp tục học qua chu kỳ 25-30 cho đến khi tổn thất trên dữ liệu xác thực qua chu kỳ 30, tốc độ học rất nhỏ ở mức 2,44e-06 và không thể thực hiện bất kỳ thay đổi đáng kể nào đối với các trọng số ảnh hưởng đến sự tổn thất / độ chính xác trên dữ liệu xác thực.

3.3.2. Tóm tắt

Mục đích của mục này là xem xét khái niệm về lập biểu tỷ lệ học và cách chúng có thể được sử dụng để tăng độ chính xác phân loại. Đã thảo luận về hai loại lịch trình tỷ lệ học chính:

1. Lịch trình dựa trên thời gian giảm dần dựa trên số chu kỳ.
2. Bộ lập lịch dựa trên tỷ lệ bỏ học dựa trên một chu kỳ cụ thể

Độ chính xác bộ lập lịch tỷ lệ học nên sử dụng (nếu hoàn toàn nên sử dụng bộ lập lịch) là một phần quá trình thực nghiệm. Thông thường, thực nghiệm đầu tiên sẽ không sử dụng bất kỳ loại phân rã hoặc lập lịch tỷ lệ học nào để có thể có được đường cong chính xác và đường cong tổn thất hoặc chính xác.

Từ đó, có thể giới thiệu lịch biểu dựa trên thời gian tiêu chuẩn do Keras cung cấp (với định luật ngón tay cái là decay = alpha_init / epochs) và chạy thực nghiệm thứ hai để đánh giá kết quả. Một vài thực nghiệm tiếp theo có thể liên quan đến việc hoán đổi lịch trình cơ sở thời gian cho một thực nghiệm dựa trên thả bằng các yếu tố thả khác nhau.

Tùy thuộc vào mức độ khó khăn bộ dữ liệu phân loại cùng với độ sâu mạng, có thể chọn phương pháp ctrl + c để huấn luyện.

Nhìn chung, hãy sẵn sàng dành một lượng thời gian đáng kể để huấn luyện mạng và đánh giá các bộ thông số và thói quen học khác nhau. Ngay cả các bộ dữ liệu và dự án đơn giản cũng có thể tổn thất từ 10 đến 100 thực nghiệm để có được mô hình có độ chính xác cao.

Tại thời điểm này trong nghiên cứu về học sâu, nên hiểu rằng huấn luyện mạng nơ-ron là một phần khoa học. Mục tiêu trong cuốn sách này là cung cấp cho người đọc kiến thức khoa học đằng sau việc huấn luyện một mạng cùng với các quy tắc thông thường sử dụng để có thể tìm hiểu các đồ thị được đưa ra sau đó - nhưng hãy nhớ rằng không có gì vượt qua được các thực nghiệm thực tế.

Càng thực hành nhiều hơn trong việc huấn luyện mạng nơ-ron, ghi lại kết quả những gì đã làm và những gì chưa làm, sẽ càng trở nên tốt hơn. Không có lỗi tắt khi nghiên cứu về vấn đề huấn luyện mạng nơ-ron, cần bỏ ra hàng giờ và trở nên thành thạo với trình tối ưu hóa SGD cùng với các tham số của chúng.

3.4. HIỆN TƯỢNG CHƯA KHỚP VÀ QUÁ KHỚP

Trong mục 3.2, đã nói sơ qua về hiện tượng chưa khớp và quá khớp. Nay sẽ đi sâu hơn và thảo luận về cả việc chưa khớp và quá khớp trong lĩnh vực học sâu. Để giúp hiểu khái niệm về cả chưa khớp và quá khớp, sẽ cung cấp một số hình ảnh và số liệu để có thể kết hợp đường cong tổn thất và độ chính xác của kết quả huấn luyện để nhận xét. Từ đó, sẽ thảo luận về cách có thể huấn luyện mạng nơ-ron theo thời gian thực với thư viện hỗ trợ là Keras. Hiện nay, phải đợi cho đến khi mạng huấn luyện hoàn thành kết thúc mới có thể hiển thị sự tổn thất và độ chính xác của quá trình huấn luyện.

Chờ cho đến khi kết thúc quá trình huấn luyện để hình dung sự tổn thất và độ chính xác có thể gây lãng phí thời gian, đặc biệt là nếu các thực nghiệm mất quá nhiều thời gian để thực thi và không có cách nào có thể hình dung được sự tổn thất và độ chính xác trong quá trình huấn luyện, có thể dành hàng giờ hoặc thậm chí nhiều ngày để huấn luyện một mạng khi không nhận ra rằng quá trình huấn luyện đã bị dừng sau vài chu kỳ đầu tiên.

Thay vào đó, sẽ có lợi hơn nhiều nếu có thể hiển thị sơ đồ huấn luyện và tổn thất sau mỗi chu kỳ và hình dung ra kết quả. Từ đó có thể đưa ra các quyết định tốt hơn, sáng suốt hơn về việc nên dừng thực nghiệm sớm hay tiếp tục huấn luyện.

3.4.1. Thế nào là chưa khớp và quá khớp

Khi huấn luyện mạng nơ-ron, cần phải *hết sức quan tâm* đến cả hiện tượng chưa khớp và quá khớp. Hiện tượng chưa khớp xảy ra khi mô hình không thể có được tổn thất đủ thấp trên *tập huấn luyện*. Trong trường hợp này, mô hình không học được các mẫu cơ bản trong dữ liệu huấn luyện.

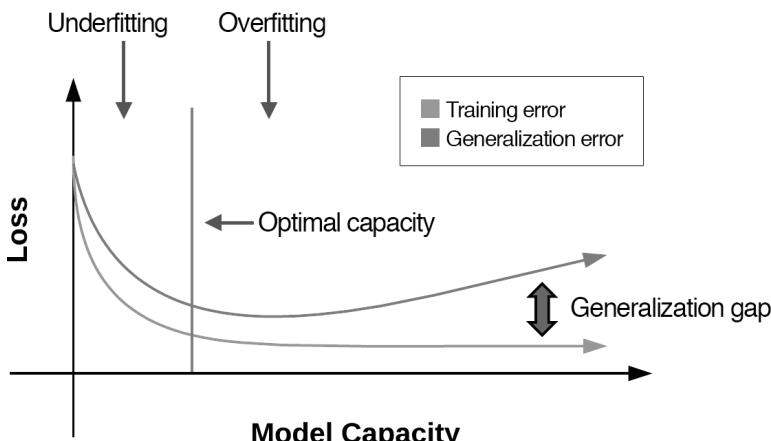
Do đó, mục tiêu khi huấn luyện một mô hình học máy là:

- Giảm tổn thất huấn luyện càng nhiều càng tốt.
- Trong khi đảm bảo khoảng cách giữa sai số huấn luyện và kiểm tra là nhỏ một cách hợp lý.

Kiểm soát xem một mô hình có hiệu suất hoạt động kém hoặc phù hợp có thể được thực hiện bằng cách điều chỉnh *hiệu suất* của mạng nơ-ron. Có thể tăng hiệu suất bằng cách thêm nhiều lớp và số lượng các nơ-ron vào mạng. Tương tự, có thể giảm hiệu suất bằng cách loại bỏ các lớp và số lượng nơ-ron và áp dụng các kỹ thuật chính quy (giảm tham số phân rã decay, tỷ lệ bỏ học, tăng thêm dữ liệu, dừng huấn luyện kịp thời, v.v.).

Hình 3.18 (tham khảo từ ví dụ tuyệt vời Hình 5.3 của Goodfellow và cộng sự, trang 112 [33]), giúp hình dung mối quan hệ giữa hiện tượng chưa khớp và quá khớp chung với với hiệu suất của mô hình:

Trên trục x, đã vẽ sơ đồ hiệu suất mạng và trên trục y là tổn thất, trong đó tổn thất thấp hơn là mong muốn. Thông thường, khi một mạng bắt đầu huấn luyện, đang ở trong khu vực chưa khớp, (Hình 3.18, bên trái). Tại thời điểm này, chỉ đơn giản là cố gắng tìm hiểu một số mẫu ban đầu trong dữ liệu cơ bản và di chuyển các trọng số ra khỏi các khởi tạo ngẫu nhiên thành các giá trị cho phép thực sự học hỏi từ dữ liệu. Lý tưởng nhất là cả tổn thất huấn luyện và tổn thất xác thực sẽ giảm cùng nhau trong giai đoạn này - sự sụt giảm đó chứng tỏ rằng mạng thực sự đang học.



Hình 3.18: Mối quan hệ giữa hiệu suất huấn luyện và tổn thất. Đường màu tím đọc tách biệt hiệu suất tối ưu từ phần chưa khớp (bên trái) và phần quá khớp (bên phải). Hiệu suất tối ưu xảy ra khi tổn thất cả hai cấp độ huấn luyện và khai quát. Khi tổn thất khai quát hóa tăng có nghĩa là hiện tượng quá khớp đang xảy ra. Lưu ý: Hình ảnh tham khảo từ Goodfellow và cộng sự, trang 112 [33].

Tuy nhiên, khi hiệu suất mô hình tăng lên (do mạng sâu hơn, nhiều số lượng nơ-ron hơn, v.v.), sẽ đạt được “hiệu suất tối ưu” của mạng. Tại thời điểm này, sự tổn thất và độ chính xác huấn luyện và xác thực bắt đầu phân kỳ, và một khoảng cách đáng chú ý bắt đầu hình thành. Mục tiêu là *hạn chế khoảng cách này*, do đó cần duy trì tính tổng quát của mô hình.

Nếu không giới hạn được khoảng cách này, sẽ kết hợp vào khu vực quá khớp (Hình 3.18, bên phải). Tại thời điểm này, tổn thất huấn luyện sẽ bị đình trệ và tiếp tục giảm trong khi tổn thất xác thực bị đình trệ và cuối cùng tăng lên. Sự tăng lên của tổn thất xác thực trong một loạt các chu kỳ liên tiếp là một chỉ số lớn về tình trạng quá khớp.

Vì vậy, làm thế nào để chống lại hiện tượng quá khớp? Cơ bản sẽ có hai kỹ thuật:

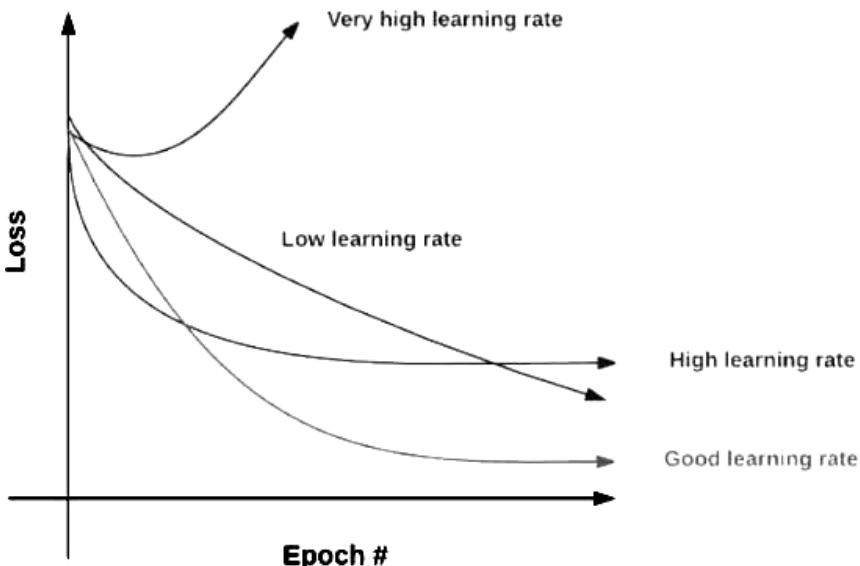
1. Giảm độ phức tạp mô hình, chọn một mạng đơn giản hơn với ít lớp hơn và số lượng nơ-ron trong mạng

2. Áp dụng các phương pháp chính quy.

Sử dụng các mạng nơ-ron nhỏ hơn có thể hoạt động cho các bộ dữ liệu nhỏ hơn, nhưng nói chung, đây không phải là giải pháp tối ưu. Thay vào đó, nên áp dụng các kỹ thuật chính quy hóa như phân rã trọng số, tỷ lệ bỏ học, tăng thêm dữ liệu, v.v. Trong thực tế, sử dụng các kỹ thuật chính quy để kiểm soát hiện tượng quá khớp thay vì điều chỉnh kích thước mạng [129] trừ khi có lý do phù hợp để tin rằng kiến trúc mạng đơn giản là *quá phức tạp* ảnh hưởng đến vấn đề.

3.4.1.1. Ảnh hưởng của tỷ lệ học

Trong phần trước, đã xem xét một ví dụ về việc quá khớp - nhưng tỷ lệ học đóng vai trò gì trong việc dẫn đến hiện tượng quá khớp? Thực sự có một cách để phát hiện nếu mô hình bị quá khớp khi đưa ra một tập hợp siêu tham số chỉ bằng cách kiểm tra đường cong tổn thất? Chỉ cần nhìn vào Hình 3.19 trong một ví dụ tham khảo từ Karpathy và các cộng sự [76].



Hình 3.19: Một đồ thị trực quan hóa ảnh hưởng đến tỷ lệ học khác nhau sẽ bị tổn thất (tham khảo từ Karpathy và cộng sự [76]). Tỷ lệ học rất cao (màu đỏ) sẽ giảm tỷ lệ bõ ban đầu nhưng tăng đáng kể ngay sau đó. Tỷ lệ học thấp (màu xanh) sẽ xấp xỉ tuyến tính trong sự tổn thất theo thời gian trong khi tỷ lệ học cao (màu tím) sẽ giảm nhanh chóng nhưng lại chững lại. Cuối cùng, tốc độ học tốt (màu xanh lá cây) sẽ giảm với tốc độ nhanh hơn tuyến tính, cho phép điều hướng vùng tổn thất.

Trên trục x, đã vẽ các chu kỳ của mạng nơ-ron cùng với tổn thất tương ứng trên trục y. Lý tưởng nhất là tổn thất huấn luyện và tổn thất xác thực sẽ trông giống như đường cong màu xanh lá cây, nơi tổn thất giảm nhanh chóng nhưng không nhanh đến mức không thể điều hướng cảnh quan tổn thất và giải quyết vào khu vực của tổn thất thấp.

Hơn nữa, khi vẽ cả tổn thất huấn luyện và xác thực cùng một lúc, có thể hiểu rõ hơn về quá trình huấn luyện. Tốt hơn là, tổn thất huấn luyện và xác thực sẽ gần như bắt chước lẫn nhau, chỉ có một khoảng cách nhỏ giữa tổn thất huấn luyện và tổn thất xác thực, cho thấy hiện tượng quá khớp là không đáng kể.

Tuy nhiên, nhiều ứng dụng trong thực tế, hành vi bắt chước là đơn giản không thực tế hoặc thậm chí là mong muôn vì có thể ngụ ý rằng sẽ tổn thất nhiều thời gian để huấn luyện cho mô hình. Do đó, chỉ cần ghi nhớ khoảng cách giữa các lần huấn luyện và tổn thất xác thực. Chừng nào khoảng cách này không tăng lên đáng kể, chúng ta biết có một mức độ chấp nhận của hiện tượng quá khớp. Tuy nhiên, nếu không duy trì được

khoảng cách này và việc huấn luyện và xác thực tổn thất phân kỳ rất nhiều, thì biết rằng có nguy cơ xảy ra hiện tượng quá khớp. Khi tổn thất xác thực bắt đầu tăng, biết rằng hiện tượng quá khớp đang xảy ra.

3.4.1.2. Chú ý đến đường cong huấn luyện

Khi huấn luyện mạng nơ-ron, hãy chú ý đến đường cong tổn thất và độ chính xác cho cả dữ liệu huấn luyện và dữ liệu xác thực. Trong vài chu kỳ đầu tiên, có vẻ như một mạng nơ-ron đang hoạt động tốt, có thể bị chưa khớp một chút - nhưng mô hình này có thể thay đổi nhanh chóng và có thể bắt đầu thấy sự khác biệt trong huấn luyện và tổn thất xác thực. Khi điều này xảy ra, đánh giá mô hình:

- Có phải đang áp dụng bất kỳ kỹ thuật chính quy?
- Có phải tỷ lệ học quá cao?
- Có phải mạng có quá sâu không?

Một lần nữa, huấn luyện mạng học sâu là một phần khoa học. Cách tốt nhất để nghiên cứu cách hiểu các đường cong này là huấn luyện càng nhiều mạng càng tốt và kiểm tra các đồ thị. Theo thời gian, sẽ phát triển ý thức về những gì hoạt động và những gì không - nhưng không hy vọng sẽ có được nó ngay khi thử lần đầu tiên.

Cuối cùng, cũng nên chấp nhận thực tế rằng việc cung cấp quá khớp cho một số bộ dữ liệu nhất định là không thể tránh khỏi. Ví dụ, rất dễ dàng để phù hợp với một mô hình cho bộ dữ liệu CIFAR-10. Nếu tổn thất tập luyện và tổn thất xác thực bắt đầu phân kỳ, đừng hoảng sợ - chỉ cần cố gắng kiểm soát khoảng cách càng nhiều càng tốt.

Cũng nhận ra rằng khi giảm tỷ lệ học trong các chu kỳ sau này (chẳng hạn như khi sử dụng công cụ lập lịch tỷ lệ học), sẽ trở nên dễ dàng hơn để phù hợp hơn.

3.4.1.3. Điều gì xảy ra khi tổn thất xác thực thấp hơn so với tổn thất huấn luyện?

Một hiện tượng khác có thể gặp là khi tổn thất xác thực thấp hơn so với tổn thất tập luyện. Sao có thể như thế được? Làm thế nào một mạng có thể hoạt động tốt hơn trên dữ liệu xác thực khi các mẫu mà nó đang cố gắng học là từ dữ liệu huấn luyện? Có nên thực hiện hiệu suất huấn luyện luôn tốt hơn so với tổn thất kiểm tra hoặc kiểm tra?

Trong thực tế, có nhiều lý do về vấn đề này. Có lẽ cách khảo sát đơn giản nhất là:

1. Dữ liệu huấn luyện đang nhìn thấy là tất cả các ví dụ “khó” để phân loại.

2. Trong khi dữ liệu xác thực bao gồm các điểm dữ liệu “dễ dàng”.

Tuy nhiên, trừ khi *có tình* lấy mẫu dữ liệu theo cách này, không chắc rằng việc phân chia thực nghiệm và huấn luyện ngẫu nhiên sẽ phân chia gọn gàng các loại điểm dữ liệu này.

Một lý do thứ hai sẽ là tăng dữ liệu. Bao gồm việc tăng dữ liệu chi tiết, nhưng ý chính là trong quá trình huấn luyện, thay đổi ngẫu nhiên các ảnh huấn luyện bằng cách áp dụng các biến đổi ngẫu nhiên cho chúng như dịch, xoay, thay đổi kích thước và cắt. Do những thay đổi này, mạng liên tục thấy các ví dụ tăng cường dữ liệu huấn luyện, là một hình thức chính quy, cho phép mạng tổng quát tốt hơn với dữ liệu xác thực trong khi có thể hoạt động kém hơn trên tập huấn luyện.

Một lý do thứ 3 có thể là tập huấn luyện không đủ mạnh, có thể muốn xem xét tăng tỷ lệ học và điều chỉnh tính chắc chắn chính quy.

3.4.2. Giám sát quá trình huấn luyện

Trong phần đầu tiên phần này, sẽ tạo ra một hàm gọi lại *TrainingMonitor* sẽ được gọi vào cuối mỗi chu kỳ khi huấn luyện một mạng với Keras. Màn hình này sẽ tuần tự hóa sự tổn thất và độ chính xác cho cả tập huấn luyện và xác thực được đặt vào ô cứng, sau đó là xây dựng một biểu đồ dữ liệu.

Áp dụng hàm gọi lại này trong quá trình huấn luyện sẽ cho phép giám sát quá trình huấn luyện và phát hiện ra tình trạng quá khớp quá sớm, cho phép dừng thực nghiệm và tiếp tục cố gắng điều chỉnh các tham số.

3.4.2.1. Quá trình huấn luyện

Lớp *TrainingMonitor* sẽ chạy trong mô đun con *pyimagesearch*.
callbacks:

```
|--- pyimagesearch
|   |--- __init__.py
|   |--- callbacks
|   |   |--- __init__.py
|   |   |--- trainingmonitor.py
|   |--- datasets
|   |--- nn
|   |--- preprocessing
|   |--- utils
```

Tạo tập tin *trainingmonitor.py* và để bắt đầu:

```
1 # import the necessary packages
2 from keras.callbacks import BaseLogger
3 import matplotlib.pyplot as plt
4 import numpy as np
5 import json
6 import os
7
8 class TrainingMonitor(BaseLogger):
9     def __init__(self, figPath, jsonPath=None, startAt=0):
10         # store the output path for the figure, the path to the JSON
11         # serialized file, and the starting epoch
12         super(TrainingMonitor, self).__init__()
13         self.figPath = figPath
14         self.jsonPath = jsonPath
15         self.startAt = startAt
```

Dòng 1-6 nhập các gói Python cần thiết. Để tạo một lớp ghi lại sự tồn thât và độ chính xác vào ổ cứng, sẽ cần phải mở rộng lớp BaseLogger của Keras (Dòng 2).

Hàm tạo cho lớp *TrainingMonitor* được định nghĩa trên Dòng 9. Hàm tạo yêu cầu một đối số, theo sau là hai đối số tùy chọn:

- *figPath*: đường dẫn đến đồ thị đầu ra mà có thể sử dụng để hình dung sự tồn thât và độ chính xác theo thời gian.
- *jsonPath*: một đường dẫn tùy chọn được sử dụng để tuân tự hóa các giá trị tồn thât và độ chính xác dưới dạng tệp JSON. Đường dẫn này hữu ích nếu muốn sử dụng lịch sử huấn luyện để tạo các khung hình ảnh tùy chỉnh.
- *startAt*: đây là chu kỳ bắt đầu mà việc huấn luyện được nối lại khi sử dụng huấn luyện *ctrl + c*.

Tiếp theo, định nghĩa hàm gọi lại *on_train_begin*, như tên cho thấy, được gọi một lần khi quá trình huấn luyện bắt đầu:

```
17     def on_train_begin(self, logs={}):
18         # initialize the history dictionary
19         self.H = {}
20
21         # if the JSON history path exists, load the training history
22         if self.jsonPath is not None:
23             if os.path.exists(self.jsonPath):
24                 self.H = json.loads(open(self.jsonPath).read())
25
26             # check to see if a starting epoch was supplied
27             if self.startAt > 0:
28                 # loop over the entries in the history log and
29                 # trim any entries that are past the starting
30                 # epoch
31                 for k in self.H.keys():
32                     self.H[k] = self.H[k][:self.startAt]
```

Trên dòng 19, định nghĩa H, được sử dụng để thể hiện “lịch sử” của tồn thât, xem cách mà dictionary được cập nhật trong hàm on_epoch_end trong đoạn chương trình tiếp theo.

Dòng 22 kiểm tra xem đường dẫn JSON có được cung cấp không. Nếu vậy, sẽ kiểm tra xem tệp JSON này có tồn tại không. Với điều kiện là tệp JSON tồn tại, tải nội dung của nó và cập nhật lịch sử dictionary H cho đến khi bắt đầu chu kỳ (vì đó là nơi sẽ tiếp tục huấn luyện từ đó).

Bây giờ có thể chuyển sang hàm quan trọng nhất, on_epoch_end, được gọi khi một chu kỳ huấn luyện hoàn thành:

```
34     def on_epoch_end(self, epoch, logs={}):
35         # loop over the logs and update the loss, accuracy, etc.
36         # for the entire training process
37         for (k, v) in logs.items():
38             l = self.H.get(k, [])
39             l.append(v)
40             self.H[k] = l
```

Phương thức on_epoch_end được tự động cung cấp cho các tham số từ Keras. Đầu tiên là một số nguyên biểu thị số chu kỳ. Thứ hai là một dictionary, logs, trong đó có sự tồn thât về huấn luyện và xác thực + độ chính xác cho chu kỳ hiện tại, lặp lại từng mục trong logs và sau đó cập nhật lịch sử dictionary (dòng 37-40).

Sau khi chương trình này thực thi, dictionary H hiện có bốn phần:

1. train_loss
2. train_acc
3. val_loss
4. val_acc

Duy trì một danh sách các giá trị cho mỗi khóa này. Mỗi danh sách được cập nhật vào cuối mỗi chu kỳ, do đó cho phép vẽ đường cong chính xác và tồn thât được cập nhật ngay khi chu kỳ hoàn thành.

Trong trường hợp jsonPath được cung cấp, sẽ tuần tự hóa lịch sử H vào ô cứng:

```
42         # check to see if the training history should be serialized
43         # to file
44         if self.jsonPath is not None:
45             f = open(self.jsonPath, "w")
46
47             f.write(json.dumps(self.H))
48             f.close()
```

Cuối cùng cũng có thể xây dựng đồ thị thực tế:

```
49      # ensure at least two epochs have passed before plotting
50      # (epoch starts at zero)
51      if len(self.H["loss"]) > 1:
52          # plot the training loss and accuracy
53          N = np.arange(0, len(self.H["loss"]))
54          plt.style.use("ggplot")
55          plt.figure()
56          plt.plot(N, self.H["loss"], label="train_loss")
57          plt.plot(N, self.H["val_loss"], label="val_loss")
58          plt.plot(N, self.H["acc"], label="train_acc")
59          plt.plot(N, self.H["val_acc"], label="val_acc")
60          plt.title("Training Loss and Accuracy [Epoch {}]".format(
61              len(self.H["loss"])))
62          plt.xlabel("Epoch #")
63          plt.ylabel("Loss/Accuracy")
64          plt.legend()
65
66          # save the figure
67          plt.savefig(self.figPath)
68          plt.close()
```

3.4.3. Tóm tắt

Trong mục 3.4 này, đã xem xét hiện tượng chưa khớp và quá khớp. Chưa khớp xảy ra khi mô hình không thể có được tổn thất đủ thấp trên tập huấn luyện. Trong khi đó, quá khớp xảy ra khi khoảng cách giữa tổn thất huấn luyện và tổn thất xác thực quá lớn, cho thấy mạng đang mô hình hóa các mẫu cơ bản trong dữ liệu huấn luyện quá mạnh.

Chưa khớp tương đối dễ dàng để hạn chế: chỉ cần thêm nhiều lớp hoặc số lượng nơ-ron vào mạng.

Khi quá khớp xảy ra, nên xem xét:

1. Giảm hiệu suất mạng bằng cách loại bỏ các lớp / nơ ron (không được khuyến khích trừ khi có dữ liệu nhỏ).

2. Áp dụng các kỹ thuật chính quy hóa mạnh mẽ hơn.

Trong gần như tất cả các trường hợp, trước tiên nên thử áp dụng phương pháp chính quy hơn là giảm kích thước mạng - ngoại lệ là nếu đang cố gắng huấn luyện một mạng sâu rộng trên một tập dữ liệu nhỏ.

Sau khi hiểu được mối quan hệ giữa năng suất của mô hình và cả 2 hiện tượng chưa khớp và quá khớp, chúng ta đã học cách giám sát quá trình huấn luyện và phát hiện ra tình trạng quá khớp khi nó xảy ra – quá trình này cho phép dừng các mạng huấn luyện sớm thay vì lãng phí thời gian để

cho mạng phù hợp. Cuối cùng, kết thúc mục này bằng cách xem xét một số ví dụ điển hình về tình trạng quá khớp.

3.5. MÔ HÌNH KIỂM TRA

Trong mục 2.3, đã thảo luận về cách lưu và tuần tự các mô hình vào ô cứng sau khi huấn luyện hoàn tất. Và trong chương trước, đã học cách phát hiện ra hiện tượng chua khớp và quá khớp khi chúng đang diễn ra, cho phép loại bỏ các thực nghiệm hoạt động không tốt.

Tuy nhiên, có thể tự hỏi liệu nó có thể kết hợp cả hai quá trình này không. Có thể tuần tự hóa các mô hình bất cứ khi nào tổn thất hoặc độ chính xác cải thiện? Hoặc có thể chỉ tuần tự hóa mô hình tốt nhất (nghĩa là mô hình có tổn thất thấp nhất hoặc độ chính xác cao nhất) trong quá trình huấn luyện? Và may mắn thay, chúng ta không phải xây dựng một hàm gọi lại tùy chỉnh - hàm này được đưa ngay vào Keras.

3.5.1. Kiểm tra các cải tiến mô hình mạng nơ-ron

Một ứng dụng tốt để kiểm tra là tuần tự hóa mạng vào ô cứng mỗi khi có sự cải thiện trong quá trình huấn luyện. Xác định một cải tiến cải thiện là một sự giảm tổn thất hoặc tăng độ chính xác - sẽ thiết lập tham số này trong hàm gọi lại Keras.

Trong ví dụ này, sẽ huấn luyện kiến trúc MiniVGGNet trên bộ dữ liệu CIFAR-10 và sau đó tuần tự hóa trọng số mạng vào ô cứng mỗi khi cải thiện hiệu suất mô hình. Để bắt đầu, hãy mở một tệp mới, đặt tên là cifar10_checkpoint_improvements.py và chèn chương trình sau đây

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from pyimagesearch.nn.conv import MiniVGGNet
4 from keras.callbacks import ModelCheckpoint
5 from keras.optimizers import SGD
6 from keras.datasets import cifar10
7 import argparse
8 import os
```

Dòng 2-8 nhập các gói thư viện Python cần thiết. Hãy lưu ý về lớp ModelCheckpoint được nhập trên Dòng 4 - lớp này sẽ cho phép kiểm tra điểm và tuần tự hóa các mạng vào ô cứng bất cứ khi nào tìm thấy sự cải thiện gia tăng về hiệu suất mô hình.

Tiếp theo, phân tích các dòng lệnh:

```
10 # construct the argument parse and parse the arguments
11 ap = argparse.ArgumentParser()
12 ap.add_argument("-w", "--weights", required=True,
13                 help="path to weights directory")
14 args = vars(ap.parse_args())
```

Đối số dòng lệnh duy nhất cần là - trọng số, đường dẫn đến thư mục đầu ra sẽ lưu trữ các mô hình nối tiếp trong quá trình huấn luyện. Sau đó, thực hiện quy trình chuẩn là tải tập dữ liệu CIFAR-10 từ ổ cứng, tăng cường độ pixel trong phạm vi [0, 1], sau đó lập trình gán nhãn:

```
16 # load the training and testing data, then scale it into the
17 # range [0, 1]
18 print("[INFO] loading CIFAR-10 data...")
19 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
20 trainX = trainX.astype("float") / 255.0
21 testX = testX.astype("float") / 255.0
22
23 # convert the labels from integers to vectors
24 lb = LabelBinarizer()
25 trainY = lb.fit_transform(trainY)
26 testY = lb.transform(testY)
```

Với dữ liệu, hiện đã sẵn sàng để khởi tạo trình tối ưu hóa SGD cùng với kiến trúc MiniVGGNet:

```
28 # initialize the optimizer and model
29 print("[INFO] compiling model...")
30 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
31 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
32 model.compile(loss="categorical_crossentropy", optimizer=opt,
33                 metrics=["accuracy"])
```

Sẽ sử dụng trình tối ưu hóa SGD với tốc độ học ban đầu là $\alpha = 0,01$ và sau đó từ từ phân rã nó trong quá trình 40 epoch, cũng sẽ áp dụng động lượng $\gamma = 0,9$ và chỉ ra rằng tốc độ Nesterov cũng nên được sử dụng.

Kiến trúc MiniVGGNet được khởi tạo để chấp nhận ảnh đầu vào có chiều rộng 32 pixel, chiều cao 32 pixel và độ sâu 3 (số kênh). Đặt các lớp = 10 vì bộ dữ liệu CIFAR-10 có mười lớp nhãn có thể.

Bước quan trọng để kiểm tra điểm mạng có thể được tìm thấy trong khối chương trình bên dưới:

```
35 # construct the callback to save only the *best* model to disk
36 # based on the validation loss
37 fname = os.path.sep.join([args["weights"],
38                         "weights-{epoch:03d}-{val_loss:.4f}.hdf5"])
```

```
39 checkpoint = ModelCheckpoint(fname, monitor="val_loss", mode="min",
40     save_best_only=True, verbose=1)
41 callbacks = [checkpoint]
```

Trên dòng 37 và 38, xây dựng một chuỗi mẫu tên tệp (fname) đặc biệt mà Keras sử dụng khi ghi mô hình vào ổ cứng. Biến đầu tiên trong mẫu, {epoch: 03d}, là số epoch, được viết thành ba chữ số.

Biến thứ hai là số liệu muôn theo dõi để cải thiện, {val_loss: .4f}, chính sự ổn định cho xác thực được đặt trên chu kỳ hiện tại. Tất nhiên, nếu muốn theo dõi độ chính xác xác thực, có thể thay thế val_loss bằng val_acc. Thay vào đó, nếu muốn theo dõi ổn định và độ chính xác huấn luyện, biến sẽ trở thành train_loss và train_acc.

Khi mẫu tên tệp đầu ra được xác định, sẽ khởi tạo lớp ModelCheckpoint trên dòng 39 và 40. Tham số đầu tiên cho ModelCheckpoint là chuỗi đại diện cho mẫu tên tệp. Sau đó chuyển qua những gì muôn theo dõi. Trong trường hợp này, muôn theo dõi ổn định xác thực (val_loss).

Tham số mode kiểm soát liệu hàm ModelCheckpoint nên tìm kiếm các giá trị giảm thiểu số liệu hoặc tối đa hóa nó hay không. Vì chúng ta đang làm việc với ổn định, thấp hơn là tốt hơn, vì vậy đặt mode = "min". Thay vào đó, nếu làm việc với val_acc, sẽ đặt mode = "max" (vì độ chính xác cao hơn sẽ tốt hơn).

Đặt save_best_only = True đảm bảo rằng mô hình tốt nhất mới nhất (theo số liệu được theo dõi) sẽ không bị ghi đè. Cuối cùng, cài đặt biến verbose = 1 chỉ đơn giản là thông báo kết quả cho người dùng khi một mô hình đang được tuân tự hóa vào ổ cứng trong quá trình huấn luyện.

Dòng 41 sau đó xây dựng một danh sách các hàm gọi lại - hàm gọi lại duy nhất cần là *checkpoint*.

Bước cuối cùng là chỉ cần huấn luyện mạng và cho phép *checkpoint* kiểm soát phần còn lại:

```
43 # train the network
44 print("[INFO] training network...")
45 H = model.fit(trainX, trainY, validation_data=(testX, testY),
46     batch_size=64, epochs=40, callbacks=callbacks, verbose=2)
```

Để thực thi tập lệnh, chỉ cần mở terminal và gõ lệnh sau:

```
$ python cifar10_checkpoint_improvements.py --weights weights/improvements
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
171s - loss: 1.6700 - acc: 0.4375 - val_loss: 1.2697 - val_acc: 0.5425
Epoch 2/40
Epoch 00001: val_loss improved from 1.26973 to 0.98481, saving model to test/
weights-001-0.9848.hdf5
...
Epoch 40/40
Epoch 00039: val_loss did not improve
315s - loss: 0.2594 - acc: 0.9075 - val_loss: 0.5707 - val_acc: 0.8190
```

Như có thể thấy kết quả là Hình 3.20, mỗi khi tần suất xác thực giảm, lưu một mô hình tuần tự mới vào ổ cứng.

```
adrianrosebrock — adrian@annalee: ~/pyimagesearch/dlbook/checkpointing_models
315s - loss: 0.3161 - acc: 0.8867 - val_loss: 0.5704 - val_acc: 0.8175
Epoch 31/40
Epoch 00030: val_loss did not improve
318s - loss: 0.3060 - acc: 0.8899 - val_loss: 0.5649 - val_acc: 0.8188
Epoch 32/40
Epoch 00031: val_loss did not improve
316s - loss: 0.3062 - acc: 0.8907 - val_loss: 0.5640 - val_acc: 0.8207
Epoch 33/40
Epoch 00032: val_loss did not improve
316s - loss: 0.2948 - acc: 0.8947 - val_loss: 0.5738 - val_acc: 0.8158
Epoch 34/40
Epoch 00033: val_loss improved from 0.56279 to 0.55456, saving model to test/weights-033-0.5546.hdf5
316s - loss: 0.2949 - acc: 0.8940 - val_loss: 0.5546 - val_acc: 0.8218
Epoch 35/40
Epoch 00034: val_loss did not improve
314s - loss: 0.2854 - acc: 0.8969 - val_loss: 0.5704 - val_acc: 0.8187
Epoch 36/40
Epoch 00035: val_loss did not improve
316s - loss: 0.2836 - acc: 0.8987 - val_loss: 0.5650 - val_acc: 0.8209
Epoch 37/40
Epoch 00036: val_loss did not improve
315s - loss: 0.2697 - acc: 0.9046 - val_loss: 0.5669 - val_acc: 0.8204
Epoch 38/40
Epoch 00037: val_loss did not improve
316s - loss: 0.2680 - acc: 0.9044 - val_loss: 0.5711 - val_acc: 0.8219
Epoch 39/40
Epoch 00038: val_loss did not improve
316s - loss: 0.2634 - acc: 0.9041 - val_loss: 0.6066 - val_acc: 0.8096
Epoch 40/40
Epoch 00039: val_loss did not improve
315s - loss: 0.2594 - acc: 0.9075 - val_loss: 0.5707 - val_acc: 0.8190
```

Hình 3.20: Kiểm tra các mô hình riêng lẻ mỗi khi hiệu suất mô hình được cải thiện, kết quả là tệp nhiều trọng số sau khi huấn luyện hoàn tất.

Vào cuối quá trình huấn luyện, có 18 tệp riêng biệt, mỗi tệp cho mỗi cải tiến giá tăng:

```
$ find . -printf "%f\n" | sort
./
weights-000-1. 2697.hdf5
weights-001-0. 9848.hdf5
weights-003-0. 8176.hdf5
weights-004-0. 7987.hdf5
weights-005-0. 7722.hdf5
weights-006-0. 6925.hdf5
weights-007-0. 6846.hdf5
weights-008-0. 6771.hdf5
weights-009-0. 6212.hdf5
weights-012-0. 6121.hdf5
weights-013-0. 6101.hdf5
weights-014-0. 5899.hdf5
weights-015-0. 5811.hdf5
weights-017-0. 5774.hdf5
weights-019-0. 5740.hdf5
weights-022-0. 5724.hdf5
weights-024-0. 5628.hdf5
weights-033-0. 5546.hdf5
```

Như có thể thấy, mỗi tên tệp có ba thành phần. Đầu tiên là một chuỗi tĩnh, trọng số. Sau đó có số chu kỳ. Thành phần cuối cùng tên tệp là số liệu đang đo lường để cải thiện, trong trường hợp này là tần thắt xác thực.

Tần thắt xác thực tốt nhất đã thu được trên epoch 33 với giá trị 0,5546. Sau đó, có thể lấy mô hình này và tải nó từ ổ cứng (mục 2.3) và tiếp tục đánh giá nó hoặc áp dụng nó cho các hình ảnh tùy chỉnh (sẽ được trình bày trong mục tiếp theo).

Hãy nhớ rằng kết quả sẽ không khớp vì các mạng là ngẫu nhiên và được khởi tạo với các biến ngẫu nhiên. Tùy thuộc vào các giá trị ban đầu, có thể có các điểm kiểm tra mô hình khác nhau đáng kể, nhưng khi kết thúc quá trình huấn luyện, các mạng sẽ có được độ chính xác tương tự (\pm một vài điểm phần trăm).

3.5.2. Kiểm tra mạng nơ-ron tốt nhất

Có lẽ nhược điểm lớn nhất với các cải tiến gia tăng điểm kiểm tra là kết thúc với một loạt các tệp bổ sung mà chúng ta quan tâm, điều này đặc biệt đúng nếu tần thắt xác thực di chuyển lên và xuống qua các chu kỳ huấn luyện - mỗi cải tiến gia tăng này sẽ được chụp và nối tiếp vào ổ cứng. Trong trường hợp này, tốt nhất là chỉ lưu một mô hình và ghi đè lên nó mỗi khi số liệu cải thiện trong quá trình huấn luyện.

May mắn thay, thực hiện hành động này cũng đơn giản như cập nhật lớp ModelCheckpoint để chấp nhận một chuỗi đơn giản (nghĩa là, một đường dẫn tệp không có bất kỳ biến mấu nào). Sau đó, bất cứ khi nào số liệu cải thiện, tập tin đó chỉ đơn giản là ghi đè. Để hiểu được quy trình này,

nhanh chóng tạo một tệp Python thứ hai có tên cifar10_checkpoint_best.py và xem xét sự khác biệt.

Đầu tiên, cần nhập các gói Python cần thiết:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelBinarizer
3 from pyimagesearch.nn.conv import MiniVGGNet
4 from keras.callbacks import ModelCheckpoint
5 from keras.optimizers import SGD
6 from keras.datasets import cifar10
7 import argparse
```

Sau đó phân tích các đối số dòng lệnh:

```
9 # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-w", "--weights", required=True,
12     help="path to best model weights file")
13 args = vars(ap.parse_args())
```

Tên đối số dòng lệnh là giống nhau (--weights), nhưng mô tả chuyển đổi bây giờ khác nhau: đường dẫn đến mô hình trọng số tốt nhất tập tin. Do đó, đối số dòng lệnh này sẽ là một chuỗi đơn giản cho một đường dẫn đầu ra - sẽ không có sự hấp dẫn nào được áp dụng cho chuỗi này.

Từ đó, có thể tải bộ dữ liệu CIFAR-10 và chuẩn bị cho việc huấn luyện:

```
15 # load the training and testing data, then scale it into the
16 # range [0, 1]
17 print("[INFO] loading CIFAR-10 data...")
18 ((trainX, trainY), (testX, testY)) = cifar10.load_data()
19 trainX = trainX.astype("float") / 255.0
20 testX = testX.astype("float") / 255.0
21
22 # convert the labels from integers to vectors
23 lb = LabelBinarizer()
24 trainY = lb.fit_transform(trainY)
25 testY = lb.transform(testY)
```

Cũng như khởi tạo trình tối ưu hóa SGD và kiến trúc MiniVGGNet:

```
27 # initialize the optimizer and model
28 print("[INFO] compiling model...")
29 opt = SGD(lr=0.01, decay=0.01 / 40, momentum=0.9, nesterov=True)
30 model = MiniVGGNet.build(width=32, height=32, depth=3, classes=10)
31 model.compile(loss="categorical_crossentropy", optimizer=opt,
32     metrics=["accuracy"])
```

Bây giờ đã sẵn sàng để cập nhật chương trình ModelCheckpoint:

```
34 # construct the callback to save only the *best* model to disk
35 # based on the validation loss
36 checkpoint = ModelCheckpoint(args["weights"], monitor="val_loss",
37     save_best_only=True, verbose=1)
38 callbacks = [checkpoint]
```

Lưu ý cách chuỗi mã frame biến tồn thắt - tất cả những gì đang làm là cung cấp giá trị của trọng số ModelCheckpoint. Vì không có giá trị mẫu để điền vào, Keras sẽ chỉ ghi đè lên tệp trọng số được tuân tự hóa hiện có bất cứ khi nào số liệu giám sát cải thiện (trong trường hợp này là tồn thắt xác thực).

Cuối cùng, huấn luyện trên mạng trong khôi chương trình bên dưới:

```
40 # train the network
41 print("[INFO] training network...")
42 H = model.fit(trainX, trainY, validation_data=(testX, testY),
43     batch_size=64, epochs=40, callbacks=callbacks, verbose=2)
```

Để thực thi tập lệnh, hãy đưa ra lệnh sau:

```
$python cifar10_checkpoint best.py --weights test best/cifar10_best_weights.hdf5
[INFO] loading CIFAR-10 data...
[INFO] compiling model...
[INFO] training network...
Train on 50000 samples, validate on 10000 samples
Epoch 1/40
Epoch 00000: val_loss improved from inf to 1.26677, saving model to
    test_best/cifar10_best_weights.hdf5
305s - loss: 1.6657 - acc: 0.4441 - val_loss: 1.2668 - val_acc: 0.5584 Epoch 2/40
Epoch 00001: val_loss improved from 1.26677 to 1.21923, saving model to |
    test_best/cifar10_best_weights.hdf5
309s - loss: 1.1996 - acc: 0.5828 - val_loss: 1.2192 - val_acc: 0.5798
...
Epoch 40/40
Epoch 00039: val_loss did not improve
173s - loss: 0.2615 - acc: 0.9079 - val_loss: 0.5511 - val_acc: 0.8250
```

Ở đây có thể thấy rằng ghi đè lên tệp cifar10_best_weights.hdf5 với mạng được cập nhật chỉ khi tồn thắt xác thực giảm. Điều này có hai lợi ích chính:

1. Chỉ có một tệp được tuân tự hóa vào cuối quá trình huấn luyện - chu kỳ mô hình thu được tồn thắt thấp nhất.
2. Không nắn bát được các cải tiến gia tăng, nơi tồn thắt dao động lên xuống. Thay vào đó, chỉ lưu và ghi đè lên mô hình tốt nhất hiện có nếu số liệu có sai số thấp hơn tất cả các chu kỳ trước đó.

Để xác thực điều này, hãy xem thư mục trọng số/tốt nhất nơi có thể thấy chỉ có một tệp đầu ra:

```
$ls -l weights/best/  
total 17024  
-rw-rw-r-- 1 adrian adrian 17431968 Apr 28 09:47 cifar10_best_weights.hdf5
```

Sau đó, có thể lấy MiniVGGNet được tuân tự hóa này và đánh giá thêm về dữ liệu thực nghiệm hoặc áp dụng nó cho ảnh (được đề cập trong Mục 3.5).

3.5.3. Tóm tắt

Trong mục 3.5, đã xem xét cách giám sát một số liệu nhất định (ví dụ: tổn thất xác thực, độ chính xác xác thực, v.v.) trong quá trình huấn luyện và sau đó lưu các mạng hiệu suất cao vào ổ cứng. Có hai phương pháp để thực hiện điều này bên trong Keras:

1. *Tăng cường cải thiện* điểm kiểm tra.
2. Điểm kiểm tra *chỉ duy nhất* mô hình tốt nhất được tìm thấy trong quá trình.

3.6. TRỰC QUAN KIẾN TRÚC MẠNG

Một khái niệm chưa thảo luận là trực quan kiến trúc mạng, quá trình xây dựng đồ thị các nút và các kết nối liên quan trong mạng và lưu đồ thị vào đĩa dưới dạng ảnh (ví dụ: .PNG, .JPG, v.v.). Các nút trong đồ thị biểu thị các lớp, trong khi các kết nối giữa các nút biểu thị luồng dữ liệu qua mạng.

Các đồ thị này thường bao gồm các thành phần sau cho mỗi lớp:

1. Kích thước số lượng đầu vào.
2. Kích thước số lượng đầu ra.
3. Tùy ý tên của lớp nơ-ron

Thường sử dụng trực quan hóa kiến trúc mạng khi (1) gỡ lỗi kiến trúc mạng tùy chỉnh và (2) phát hành, trong đó trực quan hóa kiến trúc để hiểu hơn bao gồm nguồn chương trình thực tế hoặc cố gắng xây dựng bảng để truyền tải thông tin tương tự. Trong phần còn lại chương này, sẽ tìm hiểu cách xây dựng các đồ thị trực quan hóa kiến trúc mạng bằng cách sử dụng Keras, sau đó nối tiếp đồ thị vào đĩa như một ảnh thực.

3.6.1. Tầm quan trọng trực quan hóa kiến trúc mạng nơ-ron

Trực quan hóa kiến trúc một mô hình là một công cụ gỡ lỗi quan trọng, đặc biệt nếu:

1. Hiện thực hóa một kiến trúc trong một nghiên cứu, nhưng không quen thuộc với nó.

2. Thực hiện kiến trúc mạng tùy chỉnh

Nói tóm lại, trực quan hóa mạng xác nhận các giả định rằng chương trình đang xây dựng chính xác mô hình mà dự định xây dựng. Bằng cách kiểm tra ảnh ngõ ra đồ thị, có thể thấy nếu có lỗ hổng trong logic. Các lỗ hổng phổ biến nhất bao gồm:

1. Các lớp đặt hàng không chính xác trong mạng.
2. Giả sử kích thước số lượng đầu ra (không chính xác) sau lớp CONV hoặc POOL.

Bất cứ khi nào thực hiện kiến trúc mạng, nên trực quan hóa kiến trúc mạng sau mỗi khối của lớp CONV và POOL, điều này sẽ cho phép xác thực các giả định.

Trong phần còn lại mục 3.6, sẽ giúp trực quan các kiến trúc mạng để tránh các loại tình huống có vấn đề này.

3.6.1.1. Cài đặt Graphviz và Pydot

Để xây dựng một đồ thị của mạng và lưu nó vào đĩa bằng Keras, cần cài đặt điều kiện tiên quyết là graphviz:

Trên Ubuntu, chỉ cần gõ lệnh như sau:

```
$ sudo apt-get install graphviz
```

Trong khi trên macOS, có thể cài đặt graphviz qua Homebrew:

```
$ brew install graphviz
```

Khi thư viện graphviz được cài đặt, cần cài đặt hai gói thư viện của Python:

```
$ pip install graphviz==0.5.2
$ pip install pydot-ng==1.0.0
```

Các hướng dẫn ở trên được bao gồm trong mục 4.1 khi định cấu hình môi trường phát triển cho lĩnh vực học sâu. Nếu đang cố gắng để cài đặt các thư viện này, vui lòng xem tài liệu bổ sung liên quan ở cuối mục 4.1.

3.6.1.2. Trực quan hóa mạng với KERAS

Trực quan kiến trúc mạng với Keras cực kỳ đơn giản. Để xem mức độ dễ dàng nó, hãy mở một tệp mới, đặt tên là visualize_architecture.py và chèn đoạn chương trình sau:

```
1 # import the necessary packages
2 from pyimagesearch.nn.conv import LeNet
3 from keras.utils import plot_model
4
5 # initialize LeNet and then write the network architecture
6 # visualization graph to disk
7 model = LeNet.build(28, 28, 1, 10)
8 plot_model(model, to_file="lenet.png", show_shapes=True)
```

Dòng 2 gọi thư viện triển khai LeNet (mục 2.4) - đây là kiến trúc mạng mà sẽ trực quan hóa. Dòng 3 nhập hàm plot_model từ Keras. Như tên hàm này cho thấy, plot_model chịu trách nhiệm xây dựng một đồ thị dựa trên các lớp bên trong mô hình đầu vào và sau đó ghi đồ thị vào đĩa một ảnh.

Trên Dòng 7, khởi tạo kiến trúc LeNet như thể sẽ áp dụng nó cho bộ dữ liệu MNIST để phân loại chữ số. Các tham số bao gồm chiều rộng số lượng đầu vào (28 pixel), chiều cao (28 pixel), độ sâu (1 kênh) và tổng số nhãn lớp (10).

Cuối cùng, dòng 8 vẽ mô hình và lưu nó vào đĩa dưới tên lenet.png.

Để thực thi tập lệnh, chỉ cần mở terminal và gõ lệnh sau:

```
$ python visualize_architecture.py
```

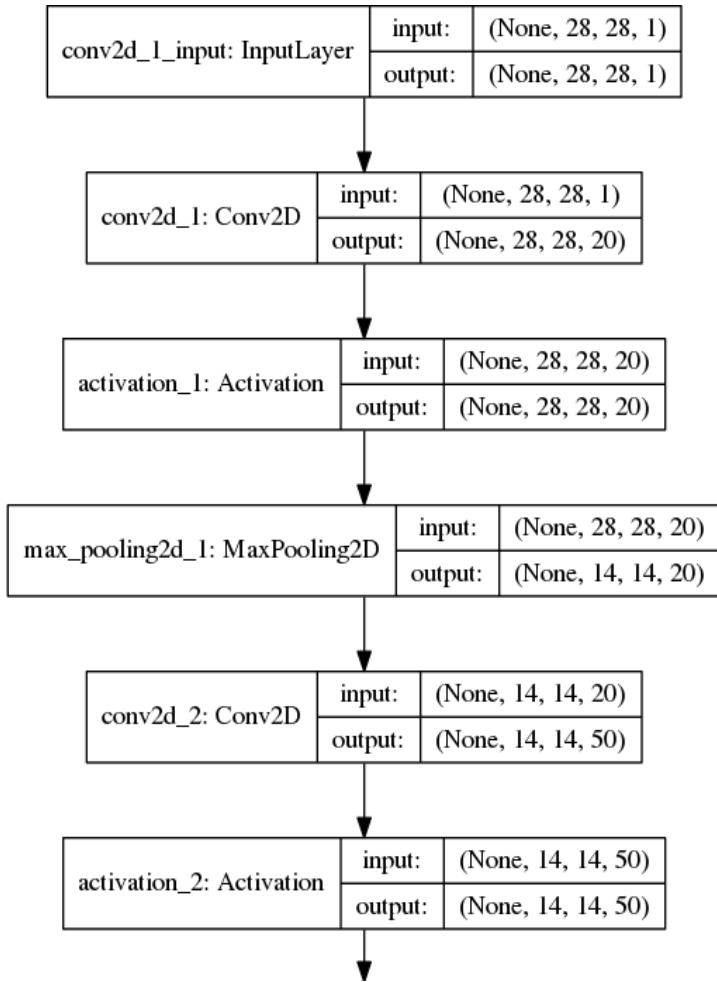
Khi lệnh thực hiện xong, hãy kiểm tra thư mục làm việc hiện tại:

```
$ ls
lenet.png  visualize_architecture.py
```

Xét kiến trúc LeNet, thấy lớp đầu tiên là InputLayer chấp nhận ảnh đầu vào có kích thước là $28 \times 28 \times 1$. Các kích thước không gian cho đầu vào và đầu ra lớp giống như một vị trí giữ chỗ cho dữ liệu đầu vào.

Kết quả hình dạng dữ liệu là $(0, 28 \times 28 \times 1)$. 0 là kích thước thực sự mỗi đợt. Khi trực quan hóa kiến trúc mạng, Keras không biết kích thước mỗi đợt dự định nên để lại giá trị là 0. Khi huấn luyện giá trị này sẽ thay đổi thành 32, 64, 128, v.v. hoặc bất kỳ kích thước mỗi đợt cho là phù hợp.

Tiếp theo, dữ liệu đưa đến lớp CONV đầu tiên, nơi học 20 đặc trưng đầu vào có kích thước $28 \times 28 \times 1$. Đầu ra lớp CONV đầu tiên này là $28 \times 28 \times 20$. Giữ lại kích thước không gian ban đầu do không đệm, nhưng bằng cách học 20 đặc trưng, đã thay đổi kích thước số lượng.

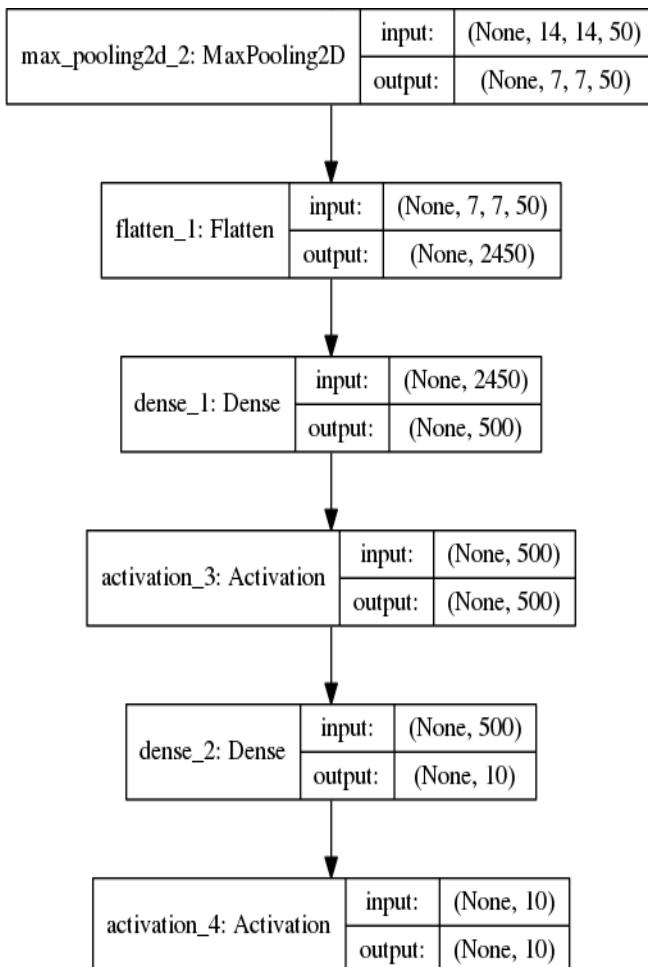


Hình 3.21: Phần I mô tả đồ họa kiến trúc mạng LeNet được tạo bởi Keras. Mỗi nút trong đồ thị biểu thị một chức năng lớp cụ thể (nghĩa là, tích chập, gộp, kích hoạt, làm phẳng, kết nối đầy đủ, v.v.). Mũi tên biểu thị luồng dữ liệu qua mạng. Mỗi nút cũng bao gồm kích thước đầu vào số lượng và kích thước đầu ra sau một thao tác nhất định.

Một lớp kích hoạt theo lớp CONV, theo định nghĩa không thê thay đổi kích thước số lượng đầu vào. Tuy nhiên, thao tác POOL có thể giảm kích thước số lượng - ở đây số lượng đầu vào giảm từ $28 \times 28 \times 20$ xuống còn $14 \times 14 \times 20$.

Lớp CONV thứ hai chấp nhận kích thước $14 \times 14 \times 20$ làm đầu vào, nhưng sau đó học 50 đặc trưng, thay đổi kích thước đầu ra thành $14 \times 14 \times 50$ (một lần nữa, phần đệm bằng 0 được tận dụng để đảm bảo tích chập không làm giảm chiều rộng và chiều cao đầu vào). Một hàm kích hoạt được áp

dụng trước phép toán POOL. POOL một lần nữa giảm một nửa chiều rộng và chiều cao từ $14 \times 14 \times 50$ xuống còn $7 \times 7 \times 50$.



Hình 3.22: Phần II trực quan hóa kiến trúc LeNet, bao gồm các lớp được kết nối đầy đủ và phân loại softmax. Trong trường hợp này, giả sử việc khởi tạo LeNet sẽ được sử dụng với bộ dữ liệu MNIST để có 10 nút đầu ra trong lớp softmax cuối cùng.

Tại thời điểm này, đã sẵn sàng để áp dụng các lớp FC. Để thực hiện điều này, đầu vào $7 \times 7 \times 50$ được làm phẳng thành một danh sách 2450 giá trị (được tính $7 \times 7 \times 50 = 2450$). Bây giờ đã làm phẳng đầu ra phần tích chập trong mạng, có thể áp dụng lớp FC chấp nhận 2450 giá trị đầu vào và tìm hiểu 500 nút. Một kích hoạt sau, tiếp theo là một lớp FC khác, lần này giảm 500 xuống còn 10 (tổng số nhãn lớp cho bộ dữ liệu MNIST).

Cuối cùng, một trình phân loại softmax được áp dụng cho mỗi trong số 10 nút đầu vào, cung cấp cho xác suất lớp cuối cùng.

3.6.2. Tóm tắt

Tương tự như việc có thể biểu thị kiến trúc LeNet bằng chương trình, cũng có thể trực quan chính mô hình đó giống như một hình ảnh. Khi bắt đầu hành trình học sâu, rất khuyến khích sử dụng chương trình này để trực quan hóa bất kỳ mạng nào đang làm việc, đặc biệt nếu người đọc không quen thuộc với chúng. Chương trình giúp người đọc hiểu về luồng dữ liệu qua mạng và cách thay đổi kích thước số lượng dựa trên các lớp CONV, POOL và FC; sẽ giúp hiểu rõ hơn về kiến trúc thay vì chỉ dựa vào chương trình.

3.7. MẠNG NƠ-RON TÍCH CHẬP VƯỢT TRỘI ĐỂ PHÂN LOẠI

Cho đến nay, đã học được cách huấn luyện mạng nơ ron tích chập tùy chỉnh riêng từ đầu. Hầu hết các CNN này đều ở phía đơn giản hơn (và trên các bộ dữ liệu nhỏ hơn) để họ có thể dễ dàng huấn luyện về CPU mà không cần phải sử dụng GPU đắt tiền hơn, cho phép nắm vững kiến thức cơ bản về mạng nơ-ron và học sâu.

Tuy nhiên, vì đã làm việc với các mạng đơn giản hơn và các bộ dữ liệu nhỏ hơn, đã có thể tận dụng sức mạnh phân loại đầy đủ mà việc học sâu mang lại. Thư viện Keras hỗ trợ với 5 kiểu mạng CNN đã được huấn luyện trước về bộ dữ liệu ImageNet:

- VGG16
- VGG19
- ResNet50
- Inception V3
- Xception

Như đã thảo luận trong mục 4.4, mục tiêu thử thách nhận dạng ảnh quy mô lớn ImageNet (ILSVRC) [80] là huấn luyện một mô hình có thể phân loại chính xác một ảnh đầu vào thành 1.000 loại đối tượng riêng biệt. 1.000 loại này đại diện cho các lớp đối tượng thường gặp trong cuộc sống hàng ngày, chẳng hạn như các loài chó, mèo, các đối tượng khác nhau trong gia đình, các loại phương tiện, và nhiều hơn nữa.

Điều này ngũ ý rằng nếu tận dụng các mạng CNN được huấn luyện trước trên bộ dữ liệu ImageNet, có thể nhận ra tất cả 1.000 danh mục đối tượng này - không cần huấn luyện!

Trong mục này, sẽ xem xét các mô hình ImageNet tiên tiến được huấn luyện trước trong thư viện Keras. Sau đó sẽ trình bày cách có thể viết một tập lệnh Python để sử dụng các mạng này để phân loại các ảnh tùy chỉnh mà không phải huấn luyện các mô hình này từ đầu.

3.7.1. CNN tiên tiến nhất với KERAS

Tại thời điểm này, người đọc có thể tự hỏi:

“Tôi không có một GPU đắt tiền. Làm cách nào tôi có thể sử dụng các mạng học sâu lớn đã được huấn luyện trước trên các bộ dữ liệu lớn hơn nhiều so với những gì chúng tôi đã làm việc trong cuốn sách này?”

Để trả lời câu hỏi đó, hãy xem xét mục 3.1 về học tập theo thông số. Hãy nhớ lại rằng điểm học tập tham số là hai lần:

1. Xác định mô hình học máy có thể học các mẫu từ dữ liệu đầu vào trong thời gian huấn luyện (yêu cầu dành nhiều thời gian hơn cho quá trình huấn luyện), nhưng quá trình thử nghiệm sẽ nhanh hơn nhiều.
2. Có được một mô hình có thể được xác định bằng cách sử dụng một số lượng nhỏ các tham số có thể dễ dàng đại diện cho mạng, bất kể quy mô huấn luyện.

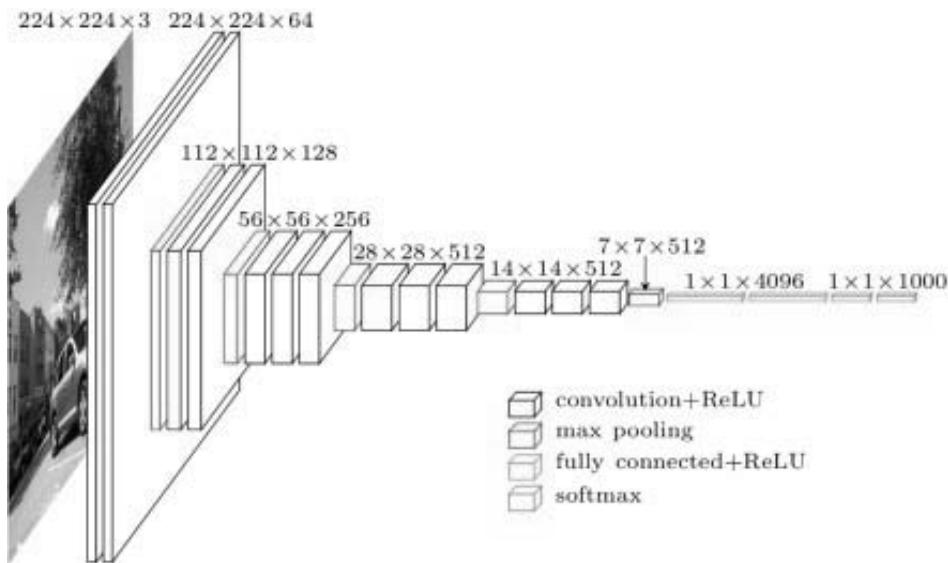
Do đó, kích thước mô hình thực tế là một hàm các tham số, không phải là lượng dữ liệu huấn luyện. Có thể huấn luyện một CNN rất sâu (như VGG hoặc ResNet) trên bộ dữ liệu 1 triệu ảnh hoặc bộ dữ liệu 100 ảnh - nhưng kích thước mô hình đầu ra kết quả sẽ giống nhau vì kích thước mô hình được xác định bởi kiến trúc được chọn.

Thứ hai, mạng nơ-ron tái trước phần lớn công việc. Dành phần lớn thời gian để thực sự huấn luyện CNN, cho dù lý do là vì độ sâu kiến trúc, số lượng dữ liệu huấn luyện hoặc số lượng thử nghiệm phải chạy để điều chỉnh siêu tham số.

Phần cứng được tối ưu hóa như GPU cho phép tăng tốc quá trình huấn luyện vì cần thực hiện cả chuyển tiếp và vượt qua trong thuật toán truyền ngược - như đã biết, quá trình này là cách mạng học thực sự. Tuy nhiên, một khi mạng được huấn luyện, chỉ cần thực hiện chuyển tiếp để phân loại một ảnh đầu vào nhất định. Lan truyền ngược thì chắc chắn nhanh hơn, cho phép phân loại ảnh đầu vào bằng cách sử dụng các mạng nơ-ron sâu trên CPU.

Trong hầu hết các trường hợp, các kiến trúc mạng được trình bày trong chương này sẽ có thể đạt được hiệu suất thời gian thực trên CPU (vì sẽ cần một GPU) - nhưng điều đó ổn; vẫn có thể sử dụng các mạng được huấn luyện trước này trong các ứng dụng riêng.

3.7.1.1. VGG16 và VGG19



Hình 3.23: Một ảnh của kiến trúc VGG. Ảnh có kích thước $224 \times 224 \times 3$ được nhập vào mạng. Các bộ lọc tích chập có kích thước là 3×3 sau đó được áp dụng với nhiều cấu trúc xếp chồng lên nhau trước các hoạt động gộp tối đa sâu hơn trong kiến trúc

Kiến trúc mạng VGG (Hình 3.23) được Simonyan và Zisserman giới thiệu trong bài báo năm 2014, *Very Deep Convolutional Networks for Large Scale Image Recognition* [57].

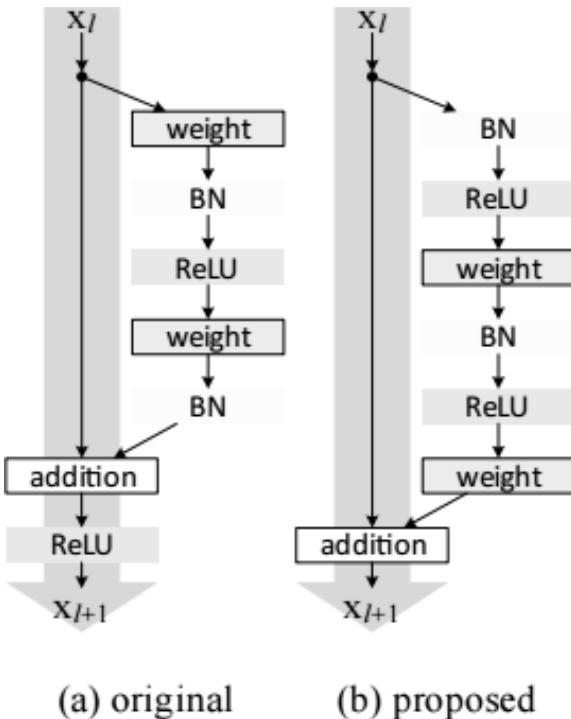
Như đã thảo luận trong mục 2.5, mạng VGG được đặc trưng bởi chỉ sử dụng 3×3 lớp chập xếp chồng lên nhau theo chiều sâu tăng dần. Giảm kích thước số lượng được xử lý bằng cách gộp tối đa. Hai lớp được kết nối đầy đủ, mỗi lớp có 4.096 nút, sau đó được phân loại bằng softmax.

Trong năm 2014, mạng lớp 16 và 19 được coi là rất sâu, mặc dù hiện tại có kiến trúc ResNet có thể được huấn luyện thành công ở độ sâu 50-200 cho ImageNet và hơn 1.000 cho CIFAR-10. Thật không may, có 2 nhược điểm lớn với VGG:

1. Rất khó để huấn luyện (may mắn thay, chỉ kiểm tra ảnh đầu vào trong chương này).

2. Bản thân trọng số mạng khá lớn. Do độ sâu và số lượng nút được kết nối đầy đủ, các tệp trọng số được tuân tự hóa cho VGG16 là 533MB trong khi VGG19 là 574 MB.

3.7.1.2. RESNET



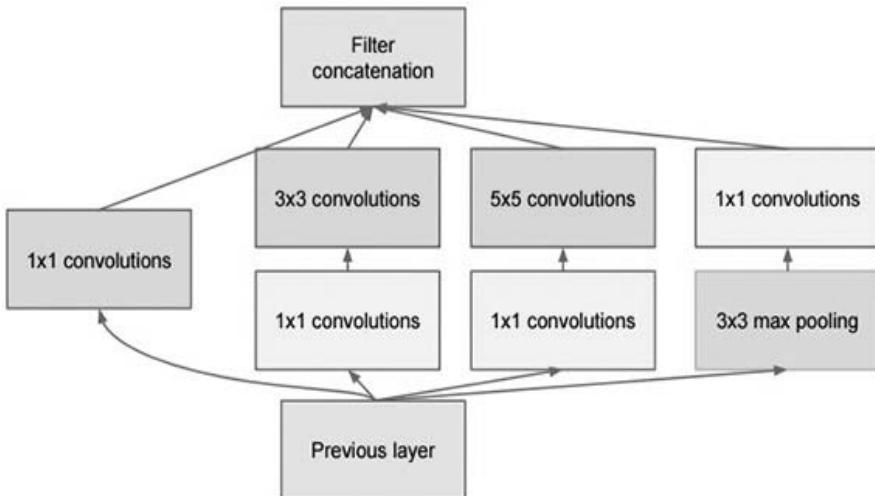
Hình 3.24: Trái: Mô-đun còn lại ban đầu. Phải: Mô-đun còn lại được cập nhật bằng cách tiền kích hoạt. Hình ảnh tham khảo từ He et al., 2016 [130].

Lần đầu tiên được giới thiệu bởi He et al. trong bài báo năm 2015 họ, *Deep Residual Learning for Image Recognition* [7], kiến trúc ResNet đã trở thành một công việc tinh túy trong tài liệu học tập sâu, chứng minh rằng các mạng cực kỳ sâu có thể được huấn luyện bằng cách sử dụng SGD tiêu chuẩn (và hàm khởi tạo hợp lý) thông qua việc sử dụng các mô hình còn lại.

Độ chính xác cao hơn có thể đạt được bằng cách cập nhật mô hình còn lại để sử dụng ánh xạ nhận dạng (Hình 3.24), như được thể hiện trong nghiên cứu tiếp theo năm 2016 của họ, *Identity Mappings in Deep Residual Networks* [81].

Điều đó nói rằng, hãy nhớ rằng việc triển khai ResNet50 (như trong 50 lớp trọng số) trong thư viện Keras dựa trên bài báo năm 2015 trước đây. Mặc dù ResNet sâu hơn nhiều so với VGG16 và VGG19, kích thước mô hình thực sự nhỏ hơn đáng kể do sử dụng nhóm trung bình toàn cục thay vì các lớp được kết nối đầy đủ, giúp giảm kích thước mô hình xuống 102MB cho ResNet50.

3.7.1.3. Inception V3



Hình 3.25: Mô hình khởi động ban đầu được sử dụng trong GoogLeNet. Mô hình Inception hoạt động như một trình trích xuất tính năng đa cấp của thành phố bằng cách tính toán 1×1 , 3×3 và 5×5 kết cấu trong cùng một mô hình của mạng. Hình từ Szegedy và các cộng sự, 2014 [55].

Mô hình Inception được giới thiệu bởi Szegedy et al. bài báo năm 2014 của họ, Đi sâu hơn với Convolutions [55]. Mục tiêu của mô hình Inception (Hình 3.25) là hoạt động như bộ trích xuất tính năng đa cấp, bằng cách tính toán các kết cấu 1×1 , 3×3 và 5×5 trong cùng một mô hình của mạng - đầu ra của các bộ lọc này là sau đó xếp chồng dọc theo kích thước kênh trước khi được đưa vào lớp tiếp theo trong mạng.

Sự xuất hiện ban đầu kiến trúc này được gọi là GoogLeNet, nhưng sau đó được đặt tên là Inception vN trong đó N đề cập đến số phiên bản do Google đưa ra. Kiến trúc Inception V3 được bao gồm trong nhân của Keras xuất phát từ sau bởi nghiên cứu Szegedy và cộng sự, Rethinking the Inception Architecture for Computer Vision (2015) [82], trong đó đề xuất cập nhật cho mô hình khởi động để tăng thêm độ chính xác phân loại ImageNet. Trọng số Inception V3 nhỏ hơn cả VGG và ResNet, với 96 MB.

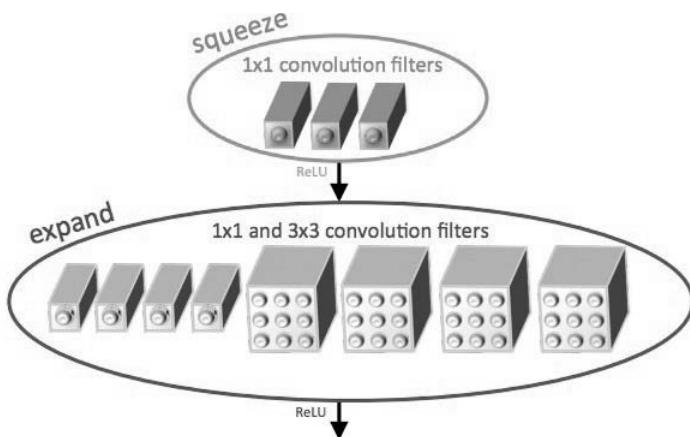
3.7.1.4. XCEPTION

Xception đã được đề xuất bởi không ai khác ngoài chính François Chollet, người sáng tạo và bảo trì chính thư viện Keras, trong bài viết năm 2016, Xception: Deep Learning with Depthwise Tách rời [83]. Xception là một phần mở rộng cho kiến trúc Inception, thay thế các mô hình Inception tiêu chuẩn bằng các cấu trúc phân tách theo chiều sâu. Trọng số Xception

là nhỏ nhất trong số các mạng được huấn luyện trước có trong thư viện Keras, có trọng số 91MB.

3.7.1.5. Có thể tối giản mạng hơn không?

Không được bao gồm trong thư viện Keras, kiến trúc SqueezeNet [84] thường được sử dụng khi SqueezeNet chiếm dung lượng rất nhỏ, chỉ 4,9 MB và là thường được sử dụng khi các mạng cần được huấn luyện và sau đó được triển khai qua mạng và/hoặc tới các thiết bị bị hạn chế tài nguyên.



Hình 3.26: Mô-đun “fire” trong SqueezeNet, bao gồm “squeeze” và “expand”. Hình tham khảo từ Iandola et al, 2016 [127].

3.7.2. Phân loại ảnh với bộ dữ liệu IMAGENET được huấn luyện trước bằng CNN

Hãy cùng học cách phân loại ảnh với mạng nơ-ron tích chập được huấn luyện trước bằng thư viện Keras.

Chỉ cần mở một tệp mới, đặt tên là `fantenet_pretrained.py` và chèn chương trình sau đây:

```
1 # import the necessary packages
2 from keras.applications import ResNet50
3 from keras.applications import InceptionV3
4 from keras.applications import Xception # TensorFlow ONLY
5 from keras.applications import VGG16
6 from keras.applications import VGG19
7 from keras.applications import imagenet_utils
8 from keras.applications.inception_v3 import preprocess_input
9 from keras.preprocessing.image import img_to_array
10 from keras.preprocessing.image import load_img
11 import numpy as np
12 import argparse
13 import cv2
```

Dòng 2-13 nhập các gói Python cần thiết. Như có thể thấy, hầu hết các gói là một phần thư viện Keras. Đặc biệt, **Dòng 2-6** xử lý việc nhập các cài đặt Keras ResNet50, Inception V3, Xception, VGG16 và VGG19, tương ứng. Xin lưu ý rằng mạng Xception chỉ tương thích với backend của TensorFlow.

Dòng 7 cung cấp cho quyền truy cập vào mô-đun con `fantenet_utils`, một bộ các hàm tiện lợi tiện dụng sẽ giúp xử lý trước ảnh đầu vào và giải chương trình các phân loại đầu ra dễ dàng hơn.

Phần còn lại nhập các hàm trợ giúp khác, tiếp theo là NumPy cho các hoạt động số và cv2 cho các ràng buộc OpenCV.

Tiếp theo, hãy để phân tích cú pháp đối số dòng lệnh:

```
15 # construct the argument parse and parse the arguments
16 ap = argparse.ArgumentParser()
17 ap.add_argument("-i", "--image", required=True,
18     help="path to the input image")
19 ap.add_argument("-model", "--model", type=str, default="vgg16",
20     help="name of pre-trained network to use")
21 args = vars(ap.parse_args())
```

Chỉ yêu cầu một đối số dòng lệnh, `--image`, là đường dẫn đến ảnh đầu vào mà muốn phân loại. cũng sẽ chấp nhận một đối số dòng lệnh tùy chọn, `--model`, một chuỗi xác định CNN được huấn luyện trước mà muốn sử dụng - giá trị này mặc định là `vgg16` cho kiến trúc VGG16.

Vì chấp nhận tên một mạng được huấn luyện trước thông qua một đối số dòng lệnh, cần xác định một dictionary Python ánh xạ các tên mô hình (chuỗi) vào các lớp Keras thực tế của chúng:

```
23 # define a dictionary that maps model names to their classes
24 # inside Keras
25 MODELS = {
26     "vgg16": VGG16,
27     "vgg19": VGG19,
28     "inception": InceptionV3,
29     "xception": Xception, # TensorFlow ONLY
30     "resnet": ResNet50
31 }
32
33 # ensure a valid model name was supplied via command line argument
34 if args["model"] not in MODELS.keys():
35     raise AssertionError("The --model command line argument should "
36         "be a key in the 'MODELS' dictionary")
```

Dòng 25-31 định nghĩa dictionary MODELS để ánh xạ chuỗi tên mô hình đến lớp tương ứng. Nếu không tìm thấy tên mô hình bên trong MODELS, sẽ nâng `AssertionError` (Dòng 34-36).

Như đã biết, CNN lấy một ảnh làm đầu vào và sau đó trả về một tập hợp các xác suất tương ứng với các nhãn lớp làm đầu ra. Các kích thước ảnh đầu vào điển hình cho một CNN được huấn luyện trên ImageNet là 224×224 , 227×227 , 256×256 và 299×299 ; tuy nhiên, cũng có thể thấy các kích thước khác VGG16, VGG19, và ResNet tất cả chấp nhận 224×224 ảnh đầu vào trong khi Inception V3 và Xception yêu cầu đầu vào 299×299 pixel, như được thể hiện bằng khái chương trình sau:

```
38 # initialize the input image shape (224x224 pixels) along with
39 # the pre-processing function (this might need to be changed
40 # based on which model we use to classify our image)
41 inputShape = (224, 224)

42 preprocess = imagenet_utils.preprocess_input
43
44 # if we are using the InceptionV3 or Xception networks, then we
45 # need to set the input shape to (299x299) [rather than (224x224)]
46 # and use a different image processing function
47 if args["model"] in ("inception", "xception"):
48     inputShape = (299, 299)
49     preprocess = preprocess_input
```

Ở đây, khởi tạo inputShape là 224×224 pixel. Cũng khởi tạo hàm xử lý để trở thành tiền xử lý tiêu chuẩn từ Keras. Tuy nhiên, nếu đang sử dụng Inception hoặc Xception, cần đặt inputShape thành 299×299 pixel, sau đó cập nhật tiền xử lý để sử dụng một hàm tiền xử lý riêng biệt thực hiện một kiểu chia tỷ lệ khác.

Bước tiếp theo là tải trọng số kiến trúc mạng được huấn luyện trước từ đĩa và khởi tạo mô hình:

```
51 # load our the network weights from disk (NOTE: if this is the
52 # first time you are running this script for a given network, the
53 # weights will need to be downloaded first -- depending on which
54 # network you are using, the weights can be 90-575MB, so be
55 # patient; the weights will be cached and subsequent runs of this
56 # script will be *much* faster)
57 print("[INFO] loading {}".format(args["model"]))
58 Network = MODELS[args["model"]]
59 model = Network(weights="imagenet")
```

Dòng 58 sử dụng dictionary MODELS cùng với đối số dòng lệnh `--model` để lấy đúng lớp mạng. CNN sau đó được khởi tạo trên dòng 59 bằng cách sử dụng trọng số ImageNet được huấn luyện trước.

Một lần nữa, hãy nhớ rằng trọng số VGG16 và VGG19 là 500MB. Trọng số ResNet là 100MB, trong khi trọng số Inception và Xception nằm trong khoảng 90-100MB. Nếu đây là lần đầu tiên chạy tập lệnh này cho một kiến trúc mạng nhất định, các trọng số này sẽ được (tự động) tải xuống

và lưu vào bộ nhớ cache vào đĩa cục bộ. Tùy thuộc vào tốc độ internet, điều này có thể tốn thời gian một lúc.

Tuy nhiên, một khi các trọng số được tải xuống, chúng sẽ không cần phải tải xuống nữa, cho phép các lần chạy tiếp theo fantenet_pretrained.py chạy nhanh hơn nhiều.

Mạng hiện đã được tải và sẵn sàng phân loại ảnh - chỉ cần chuẩn bị ảnh để phân loại bằng cách xử lý trước:

```
61 # load the input image using the Keras helper utility while ensuring
62 # the image is resized to 'inputShape', the required input dimensions
63 # for the ImageNet pre-trained network
64 print("[INFO] loading and pre-processing image...")
65 image = load_img(args["image"], target_size=inputShape)
66 image = img_to_array(image)
67
68 # our input image is now represented as a NumPy array of shape
69 # (inputShape[0], inputShape[1], 3) however we need to expand the
70 # dimension by making the shape (1, inputShape[0], inputShape[1], 3)
71 # so we can pass it through the network
72 image = np.expand_dims(image, axis=0)
73
74 # pre-process the image using the appropriate function based on the
75 # model that has been loaded (i.e., mean subtraction, scaling, etc.)
76 image = preprocess(image)
```

Dòng 65 tải ảnh đầu vào từ đĩa bằng cách sử dụng inputShape được cung cấp để thay đổi kích thước chiều rộng và chiều cao ảnh. Giả sử đang sử dụng thứ tự các kênh thứ tự trước, ảnh đầu vào hiện được thể hiện dưới dạng một mảng NumPy với hình dạng (inputShape[0], inputShape[1], 3).

Tuy nhiên, huấn luyện hoặc phân loại ảnh theo mỗi đợt với CNN, vì vậy cần thêm một chiều bổ sung cho mảng thông qua hàm np.Exand_dims trên Dòng 72. Sau khi gọi np.Exand_dims, ảnh sẽ có hình dạng (1, inputShape[0], inputShape[1], 3). Việc quên thêm kích thước phụ này sẽ dẫn đến lỗi khi gọi phương pháp .predict mô hình.

Cuối cùng, Dòng 76 gọi hàm xử lý trước thích hợp để thực hiện phép trù trung bình và / hoặc chia tỷ lệ.

Bây giờ đã sẵn sàng để truyền ảnh qua mạng và có được các phân loại đầu ra:

```
78 # classify the image
79 print("[INFO] classifying image with '{}'...".format(args["model"]))
80 preds = model.predict(image)
81 P = imagenet_utils.decode_predictions(preds)
82
83 # loop over the predictions and display the rank-5 predictions +
84 # probabilities to our terminal
85 for (i, (imagenetID, label, prob)) in enumerate(P[0]):
86     print("{}: {:.2f}%".format(i + 1, label, prob * 100))
```

Một hàm gọi đến .predict trên Dòng 80 trả về các dự đoán từ CNN. Đưa ra các dự đoán này, chuyển chúng vào hàm tiện ích ImageNet, .decode_predictions, để cung cấp cho danh sách ID nhãn lớp ImageNet, nhãn có thể đọc được con người và xác suất được liên kết với mỗi nhãn lớp. Dự đoán top 5 (tức là các nhãn có xác suất lớn nhất) sau đó được in đến thiết bị đầu cuối trên Dòng 85 và 86.

Khởi chương trình cuối cùng sẽ xử lý tải ảnh từ đĩa thông qua OpenCV, vẽ dự đoán số 1 trên ảnh và cuối cùng hiển thị nó lên màn hình :

```
88 # load the image via OpenCV, draw the top prediction on the image,
89 # and display the image to our screen
90 orig = cv2.imread(args["image"])
91 (imagenetID, label, prob) = P[0][0]
92 cv2.putText(orig, "Label: {}".format(label), (10, 30),
93             cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
94 cv2.imshow("Classification", orig)
95 cv2.waitKey(0)
```

Để xem các mạng ImageNet được huấn luyện trước, hãy để chuyển sang phần tiếp theo.

3.7.2.1. Kết quả phân loại

Để phân loại ảnh bằng cách sử dụng mạng được huấn luyện trước và Keras, chỉ cần sử dụng tập lệnh `imagenet_pretrained.py` và sau đó cung cấp (1) đường dẫn đến ảnh đầu vào mà muốn phân loại và (2) tên kiến trúc mạng mà muốn sử dụng. Đã bao gồm các lệnh ví dụ cho từng mạng được huấn luyện sẵn có sẵn trong Keras bên dưới:

```
$ python imagenet_pretrained.py \
    --image example_images/example_01.jpg --model vgg16
$ python imagenet_pretrained.py \
    --image example_images/example_02.jpg --model vgg19
$ python imagenet_pretrained.py \
    --image example_images/example_03.jpg --model inception
$ python imagenet_pretrained.py \
    --image example_images/example_04.jpg --model xception
$ python imagenet_pretrained.py \
    --image example_images/example_05.jpg --model resnet
```

Hình 3.27 bên dưới hiển thị kết quả được tạo cho các ảnh đầu vào khác nhau. Trong mỗi trường hợp, nhãn được dự đoán bởi kiến trúc mạng đã cho phản ánh chính xác nội dung của hình ảnh.



Hình 3.27: Kết quả việc áp dụng các mạng ImageNet được huấn luyện trước cho các ảnh đầu vào. Trong mỗi ví dụ, mạng được huấn luyện trước trả về kết quả phân loại chính xác.

3.7.3. Tóm tắt

Trong mục 3.7 này, đã xem xét 5 loại mạng nơ-ron tích chập được huấn luyện trước về bộ dữ liệu ImageNet bên trong thư viện Keras:

1. VGG16
2. VGG19
3. ResNet50
4. Inception V3
5. Xception

Sau đó đã học cách sử dụng từng kiến trúc mạng này để phân loại ảnh đầu vào. Với bộ dữ liệu ImageNet bao gồm 1000 danh mục đối tượng phổ biến mà có thể gặp trong cuộc sống hàng ngày, các mô hình này tạo ra các trình phân loại tuyệt vời.

Chương 4:

ỨNG DỤNG HỌC SÂU VÀO LĨNH VỰC THỊ GIÁC MÁY TÍNH

4.1. CẤU HÌNH MÔI TRƯỜNG CHO NHÀ PHÁT TRIỂN (DEVELOPTER)

Khi nói đến việc học một công nghệ mới (đặc biệt là học sâu), cấu hình môi trường phát triển có xu hướng là một nửa trận chiến. Giữa các hệ điều hành khác nhau, các phiên bản phụ thuộc khác nhau và các thư viện thực tế tự cấu hình môi trường phát triển học sâu riêng.

Những vấn đề này được kết hợp thêm bởi tốc độ mà các thư viện học sâu được cập nhật và phát hành - các đặc trưng mới thúc đẩy sự đổi mới, nhưng cũng phá vỡ các phiên bản trước. Cụ thể, bộ công cụ CUDA là một ví dụ tuyệt vời: trung bình có 2-3 phiên bản mới CUDA mỗi năm.

Với mỗi bản phát hành mới mang lại sự tối ưu hóa, đặc trưng mới và khả năng huấn luyện mạng nơ-ron nhanh hơn. Nhưng mỗi bản phát hành phức tạp hơn nữa tương thích ngược. Chu kỳ phát hành nhanh này ngụ ý rằng việc học sâu không chỉ phụ thuộc vào cách định cấu hình môi trường phát triển mà còn cả khi định cấu hình.

Nên sử dụng mục này để giúp làm quen với các thư viện học sâu khác nhau sẽ sử dụng trong cuốn sách này, sau đó làm theo hướng dẫn trên các trang liên kết đến các thư viện từ cuốn sách này.

4.1.1. Thư viện và gói hỗ trợ

Để học sâu thành công, cần có bộ công cụ và gói phù hợp. Phần này mô tả chi tiết ngôn ngữ lập trình cùng với các thư viện chính sẽ sử dụng để nghiên cứu sâu về thị giác máy tính.

4.1.1.1. Python

Sẽ sử dụng ngôn ngữ lập trình Python cho tất cả các ví dụ bên “Mạng nơ-ron học sâu và ứng dụng”. Python là một ngôn ngữ dễ học và là cách tốt nhất để làm việc với các thuật toán học sâu. Cú pháp đơn giản, trực quan cho phép tập trung vào việc học những điều cơ bản học sâu, thay vì mất hàng giờ để sửa các lỗi biên dịch ngớ ngẩn như trong các ngôn ngữ khác.

4.1.1.2. Keras

Để xây dựng và huấn luyện mạng học sâu, chủ yếu sử dụng thư viện

Keras. Keras hỗ trợ cả TensorFlow và Theano, giúp việc xây dựng và huấn luyện mạng trở nên siêu dễ dàng. Vui lòng tham khảo Mục 5.2 để biết thêm thông tin về khả năng tương thích TensorFlow và Theano với Keras.

4.1.1.3. MXNET

Cũng sẽ sử dụng mxnet, một thư viện học sâu chuyên về học đa máy, phân tán. Khả năng song song huấn luyện trên nhiều GPU/thiết bị là rất quan trọng khi huấn luyện kiến trúc mạng nơ-ron sâu trên các bộ dữ liệu ảnh lớn (như ImageNet).

4.1.1.4. OpenCV, Scikit-Image, Scikit-Learn và một số chương trình khác

Vì cuốn sách này tập trung vào việc áp dụng học sâu vào thị giác máy tính, sẽ tận dụng một vài thư viện bổ sung. Không cần phải là chuyên gia trong các thư viện này hoặc có kinh nghiệm trước đó để thành công khi sử dụng cuốn sách này, nhưng khuyên nên tự làm quen với những điều cơ bản OpenCV nếu có thể.

Mục tiêu chính OpenCV là xử lý ảnh thời gian thực. Thư viện này đã có từ năm 1999, nhưng đến tận phiên bản 2.0 năm 2009, đã thấy sự hỗ trợ Python đáng kinh ngạc bao gồm việc thể hiện ảnh dưới dạng mảng NumPy.

Thư viện OpenCV được viết bằng C / C ++, nhưng các ràng buộc Python được cung cấp khi chạy cài đặt. OpenCV đưa ra tiêu chuẩn thực tế khi xử lý ảnh, vì vậy sẽ sử dụng nó khi tải ảnh từ ổ cứng, hiển thị chúng lên màn hình và thực hiện các hoạt động xử lý ảnh cơ bản.

Để bổ sung cho OpenCV, cũng sẽ sử dụng một chút ảnh scikit [85] (scikit-image.org), một bộ các thuật toán để xử lý ảnh.

Scikit-learn [86] (scikit-learn.org), là một thư viện Python nguồn mở để học máy, kiểm tra chéo và trực quan hóa - thư viện này bổ sung tốt cho Keras, đặc biệt là khi nói đến việc tạo ra các phân tách huấn luyện / kiểm tra/xác nhận và xác nhận tính chính xác các mô hình học sâu.

4.1.2. Trường hợp dựa trên đám mây (CLOUD)

Một nhược điểm lớn của Ubuntu VM là theo định nghĩa máy ảo, VM không được phép truy cập vào các thành phần vật lý máy chủ (chẳng hạn như GPU). Khi huấn luyện các mạng học sâu lớn hơn, có GPU là vô cùng có lợi.

Đối với những người muốn có quyền truy cập vào GPU khi huấn luyện mạng nơ-ron của mình, có thể sử dụng một trong hai cách sau:

1. Định cấu hình phiên bản Amazon EC2 có hỗ trợ GPU.
2. Đăng ký tài khoản FloydHub và định cấu hình phiên bản GPU trong đám mây.

Điều quan trọng cần lưu ý là mỗi tùy chọn này đều tính phí dựa trên số giờ (EC2) hoặc giây (FloydHub) mà phiên bản được khởi động. Nếu quyết định sử dụng GPU trong tuyến đường đám mây, chắc chắn so sánh giá cả và ý thức về chi tiêu - không có gì tệ hơn là nhận được một hóa đơn lớn, bất ngờ cho việc sử dụng đám mây.

Nếu chọn sử dụng một cá thể dựa trên đám mây thì sẽ khuyến khích sử dụng Sơ đồ máy được cấu hình sẵn (AMI). AMI đi kèm với tất cả các thư viện học sâu sẽ cần trong cuốn sách này được cấu hình sẵn và cài đặt sẵn.

Để tìm hiểu thêm về AMI, vui lòng tham khảo Deep Learning for Computer Vision với trang web đồng hành Python.

4.1.3. Tóm tắt

Khi nói đến việc cấu hình môi trường phát triển học sâu, có một số tùy chọn. Nếu muốn làm việc từ máy cục bộ, thì điều đó hoàn toàn hợp lý, nhưng trước tiên sẽ cần phải biên dịch và cài đặt một số phụ thuộc. Nếu đang dự định sử dụng GPU tương thích CUDA trên máy cục bộ, một vài bước cài đặt bổ sung cũng sẽ được yêu cầu.

Nếu muốn sử dụng GPU nhưng không được gắn vào hệ thống, xem xét sử dụng các phiên bản dựa trên đám mây như Amazon EC2 hoặc FloydHub. Mặc dù các dịch vụ này phải chịu phí hàng giờ cho việc sử dụng, nhưng chúng có thể giúp tiết kiệm tiền khi so sánh với việc mua GPU trả trước.

Cuối cùng, xin lưu ý rằng nếu có kế hoạch thực hiện bất kỳ nghiên cứu hoặc phát triển học nghiêm túc nào, cân nhắc sử dụng môi trường Linux như Ubuntu. Mặc dù công việc học sâu hoàn toàn có thể được thực hiện trên Windows (không được khuyến nghị) hoặc macOS (hoàn toàn chấp nhận được nếu mới bắt đầu), gần như tất cả các môi trường cấp sản xuất để học sâu tận dụng các hệ điều hành dựa trên Linux - ghi nhớ thực tế này khi đang cấu hình môi trường phát triển học sâu riêng.

4.2. NGUYÊN TẮC CƠ BẢN VỀ ẢNH

Trước khi có thể bắt đầu xây dựng các trình phân loại ảnh, trước tiên người đọc cần hiểu rõ về hình ảnh.

Mục này sẽ tìm hiểu về khái niệm pixel là gì, cách chúng được sử dụng để tạo thành ảnh và cách truy cập các pixel được biểu thị dưới dạng mảng NumPy (gần như tất cả các thư viện xử lý ảnh trong Python đều hiển thị dưới dạng mảng NumPy, bao gồm OpenCV và scikit-image).

Cuối cùng mục này sẽ kết thúc với một cuộc thảo luận về tỷ lệ khung hình của một ảnh và mối quan hệ khi chuẩn bị bộ dữ liệu ảnh để huấn luyện một mạng nơ-ron.

4.2.1. PIXELS: khối xây dựng ảnh

Pixel là các khối xây dựng một hình ảnh thô. Mỗi ảnh bao gồm một tập hợp các pixel. Không có bất kỳ khái niệm nào để hiểu chi tiết về cấu tạo của hình ảnh như là pixel.

Thông thường, một pixel được coi là màu sắc hay cường độ của ánh sáng xuất hiện ở một vị trí nhất định trong ảnh. Nếu nghĩ về một ảnh như một miếng lưới thì mỗi ô vuông sẽ chứa một giá trị pixel. Ví dụ, xem Hình 4.1.

Ảnh trong Hình 4.1 ở trên có độ phân giải 1000×750 , nghĩa là chiều rộng tương ứng 1000 pixel và chiều cao 750 pixel. Có thể hiểu một ảnh như một ma trận (đa chiều). Trong trường hợp này, ma trận có 1000 cột (chiều rộng) với 750 hàng (chiều cao). Nhìn chung, có tổng cộng $1000 \times 750 = 750,000$ pixel trong ảnh.

Hầu hết các pixel được thể hiện theo hai cách:

1. Grayscale / kênh đơn
2. Màu

Trong ảnh xám, mỗi pixel là một giá trị vô hướng trong khoảng từ 0 đến 255, trong đó số 0 tương ứng với màu đen và 255 là màu trắng. Các giá trị trong khoảng từ 0 đến 255 là các sắc độ màu xám khác nhau, trong đó các giá trị gần 0 hơn sẽ tối hơn và các giá trị gần với 255 sẽ sáng hơn. Gradient của ảnh xám trong Hình 4.2 cho thấy các pixel tối hơn ở phía bên trái và các pixel sáng dần về phía bên phải. Pixel màu thường được biểu diễn trong không gian màu RGB (ảnh màu còn được biểu diễn trong các không gian màu khác, nhưng nằm ngoài nội dung cuốn sách này và không liên quan đến việc khái niệm về học sâu).

Các pixel trong không gian màu RGB không còn là giá trị vô hướng

như trong ảnh xám, thay vào đó, các pixel được biểu thị bằng một danh sách gồm ba giá trị: một giá trị cho thành phần Red, một cho Green và một cho Blue. Để xác định màu trong mô hình màu RGB, tất cả những gì cần làm là xác định lượng Đỏ, Xanh lục và Xanh lam có trong một pixel.



Hình 4.1: Kích thước ảnh là chiều rộng 1000 pixel và chiều cao 750 pixel, với tổng số là $1000 \times 750 = 750000$ pixel

0

255

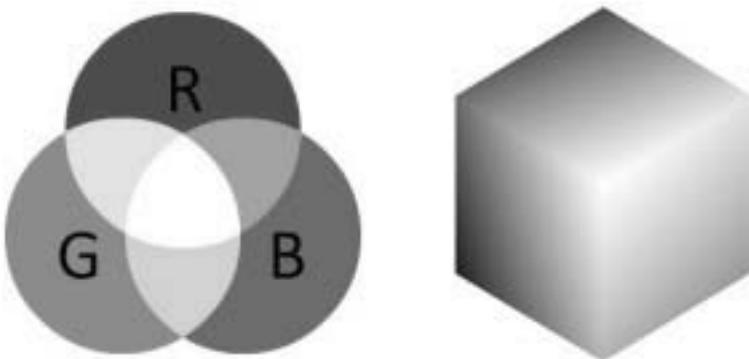
Hình 4.2: Gradient của ảnh biểu thị các giá trị pixel chuyển từ đen (0) sang trắng (255).

Mỗi kênh Đỏ, Xanh lục và Xanh lam có thể có các giá trị được xác định trong phạm vi $[0, 255]$ cho tổng số 256, trong đó 0 biểu thị không có đại diện và 255 biểu thị đầy đủ. Do giá trị pixel chỉ cần nằm trong phạm vi $[0, 255]$, thường sử dụng các số nguyên không dấu 8 bit để biểu thị cường độ.

Như đã thấy khi xây dựng mạng nơ-ron đầu tiên, sẽ thường xử lý ảnh bằng cách thực hiện phép trừ hoặc chia tỷ lệ trung bình, điều này sẽ yêu cầu chuyển đổi ảnh thành kiểu dữ liệu dấu phẩy động. Lưu ý điều này vì các loại dữ liệu được sử dụng bởi các thư viện tải ảnh từ ổ cứng (như OpenCV) thường sẽ cần phải được chuyển đổi trước khi áp dụng thực toán học trực tiếp cho ảnh.

Với ba giá trị Đỏ, Xanh lục và Xanh lam, có thể kết hợp chúng thành một tuple RGB ở dạng (đỏ, lục, lam). Bộ dữ liệu này đại diện cho một màu nhất định trong không gian màu RGB. Không gian màu RGB là một ví dụ

về không gian màu *phụ gia*: càng nhiều màu được thêm vào, pixel càng trở nên sáng hơn và gần với màu trắng hơn. Có thể hình dung không gian màu RGB trong Hình 4.3 (bên trái). Như có thể thấy, thêm màu đỏ và màu xanh lá cây để có được màu vàng. Thêm màu đỏ và màu xanh mang lại màu hồng. Và thêm cả ba màu đỏ, xanh lá cây và xanh dương với nhau, tạo ra màu trắng.



Hình 4.3: Trái: Không gian màu RGB là phụ gia. Càng kết hợp nhiều màu đỏ, xanh lá cây và xanh dương càng tiến gần đến màu trắng. Phải: Khối RGB

Để làm cho ví dụ này trở nên cụ thể hơn, xem xét màu sắc Trắng - sẽ thay đổi giá trị từng ô là màu đỏ, xanh lá cây và xanh dương, như thế này: (255, 255, 255). Sau đó, để tạo màu đen, sẽ thay đổi giá trị từng ô thành (0, 0, 0), vì màu đen thì các giá trị pixel đều là 0. Để tạo màu đỏ, sẽ thay đổi giá trị ô của kênh màu đỏ thành 255 và hai ô còn lại của kênh xanh lục và xanh lam là giá trị số 0, như thế này (255, 0, 0).

Không gian màu RGB cũng thường được hiển thị dưới dạng khối lập phương (Hình 4.3, bên phải). Vì màu RGB được định nghĩa là bộ ba giá trị, mỗi giá trị trong phạm vi [0, 255] do đó có thể nghĩ về khối có chứa $256 \times 256 \times 256 = 16,777,216$ màu, tùy thuộc vào giá trị cường độ Đỏ, Xanh lục và Xanh lam được đặt vào mỗi ô

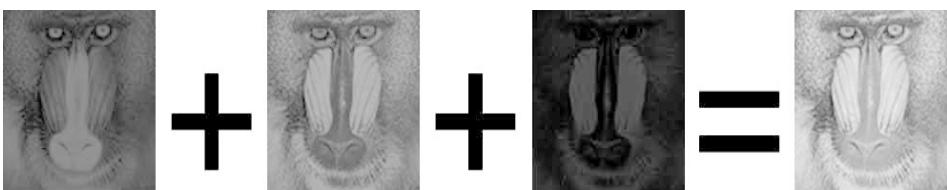
Ví dụ, xem xét cách thức mà màu đỏ, xanh lục và xanh lam cần để tạo ra một màu duy nhất (Hình 4.4, trên cùng). Ở đây, đặt R = 252, G = 198, B = 188 để tạo tông màu tương tự như màu da của người da trắng (có lẽ hữu ích khi xây dựng ứng dụng để phát hiện màu da trong ảnh). Như có thể thấy, giá trị Màu đỏ được thể hiện rất nhiều với ô gần như bằng giá trị tối đa. Màu xanh lá cây và màu xanh được thể hiện gần bằng nhau. Kết hợp các màu này theo cách *phụ gia*, có được tông màu tương tự như màu da của người da trắng.



Hình 4.4: Trên cùng: Một ví dụ về việc thêm các thành phần màu Đỏ, Xanh lục và Xanh lam khác nhau để tạo ra “tông màu da trắng”, có lẽ hữu ích trong chương trình phát hiện da. Dưới cùng: Tạo màu xanh lam bằng cách trộn nhiều lượng Đỏ, Xanh lục và Xanh lam.

4.2.1.1. Hình thành một ảnh từ các kênh

Như đã biết, một ảnh RGB được biểu thị bằng ba giá trị, một giá trị tương ứng cho mỗi thành phần Đỏ, Xanh lục và Xanh lam. Có thể hiểu một ảnh RGB bao gồm 3 ma trận độc lập có chiều rộng W và chiều cao H, như trong Hình 4.5. Có thể kết hợp ba ma trận này để thu được một mảng nhiều chiều với hình dạng $W \times H \times D$ trong đó D là chiều sâu hoặc số kênh (đối với không gian màu RGB, D = 3):



Hình 4.5: Biểu diễn một ảnh trong không gian màu RGB, trong đó mỗi kênh là một ma trận độc lập, khi kết hợp lại sẽ tạo thành ảnh cuối cùng.

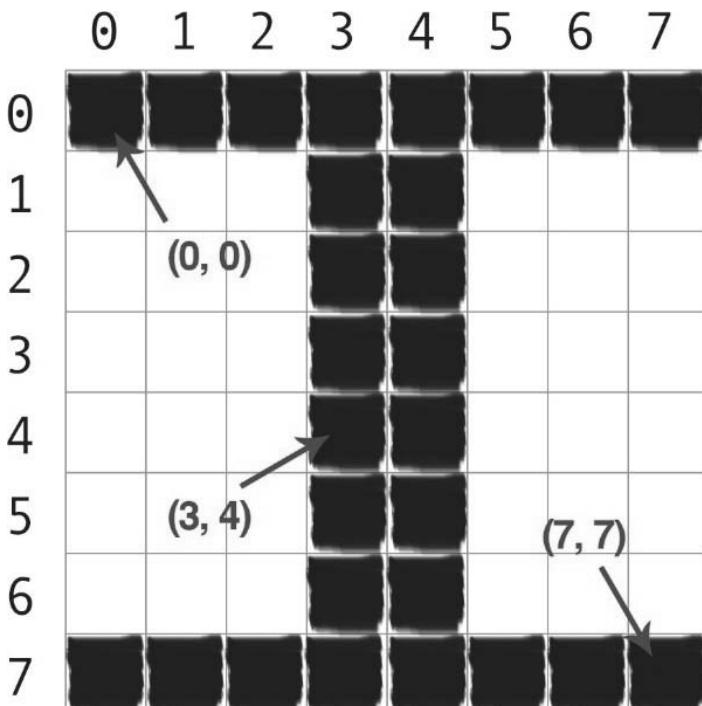
Biết rằng chiều sâu ảnh rất khác so với chiều sâu mạng nơ-ron, vấn đề này sẽ trở nên rõ ràng khi bắt đầu huấn luyện mạng nơ-ron tích chập. Tuy nhiên, hiện tại chỉ đơn giản cần hiểu rằng phần lớn các ảnh sẽ:

- Được biểu thị trong không gian màu RGB theo ba kênh, mỗi kênh có giá trị trong phạm vi [0, 255]. Một pixel đã cho trong ảnh RGB là danh sách gồm ba số nguyên: một giá trị cho Đỏ, thứ hai cho Xanh lục và giá trị cuối cùng cho Xanh lam.
- Được định nghĩa theo chương trình là mảng đa chiều 3D NumPy với chiều rộng, chiều cao và chiều sâu.

4.2.2. Hệ thống tọa độ ảnh

Như đã đề cập trong Hình 4.1 trước đó, một ảnh được biểu diễn dưới dạng một miếng lưới các pixel. Để làm cho vấn đề này rõ ràng hơn, tưởng tượng miếng lưới như một mảnh giấy vẽ đồ thị. Sử dụng đồ thị này, điểm gốc $(0, 0)$ tương ứng với góc trên bên trái ảnh. Khi di chuyển xuống và sang phải, cả hai giá trị x và y đều tăng.

Hình 4.6 cung cấp một cách nhìn trực quan về biểu diễn này trên biểu đồ. Ở đây có chữ cái trên một mảnh giấy vẽ. Thấy rằng đây là một miếng lưới có kích thước 8×8 với tổng số 64 pixel. Điều quan trọng cần lưu ý là đang đếm từ 0 chứ không phải 1. Ngôn ngữ Python được lập chỉ mục từ số 0, có nghĩa là luôn bắt đầu đếm từ số 0. Lưu ý điều này vì sẽ tránh được nhiều sự nhầm lẫn sau này (đặc biệt là nếu lập trình từ môi trường MATLAB).



Hình 4.6: Chữ cái I đặt trên một mảnh giấy vẽ đồ thị. Các pixel được truy cập bởi các tọa độ (x, y) , trong đó đi x cột sang bên phải và hàng y xuống

Ví dụ về lập chỉ mục bằng 0, xem xét các cột pixel 4 ở bên phải và 5 hàng xuống được lập chỉ mục theo điểm $(3, 4)$, một lần nữa nhớ rằng đang đếm từ 0 thay vì 1.

4.2.2.1. Ảnh dưới dạng mảng NUMPY



Hình 4.7: Tải một ảnh có tên example.png từ ổ cứng và hiển thị nó lên màn hình dùng thư viện hỗ trợ OpenCV

Các thư viện xử lý ảnh như OpenCV và scikit-image miêu tả cho ảnh RGB dưới dạng mảng NumPy đa chiều với hình dạng (chiều cao, chiều rộng, chiều sâu). Những người lần đầu tiên sử dụng thư viện xử lý ảnh thường bị nhầm lẫn bởi biểu diễn này - tại sao chiều cao lại đi trước chiều rộng vì thông thường chiều rộng trước sau đó là chiều cao? Câu trả lời là do ký hiệu ma trận.

Khi xác định kích thước ma trận, luôn viết nó dưới dạng hàng \times cột. Số lượng hàng trong một ảnh là chiều cao của ảnh trong khi số lượng cột là chiều rộng ảnh. Độ sâu vẫn sẽ vẫn là số lượng kênh màu.

Do đó, mặc dù có thể hơi khó hiểu khi thấy hình dạng mảng NumPy được biểu thị là (chiều cao, chiều rộng, chiều sâu), mô tả này thực sự có ý nghĩa trực quan khi xem xét cách xây dựng và chú thích ma trận.

Ví dụ, thư viện OpenCV và hàm cv2.imread được sử dụng để tải ảnh từ ổ cứng và hiển thị kích thước của nó:

```
1 import cv2
2 image = cv2.imread("example.png")
3 print(image.shape)
4 cv2.imshow("Image", image)
5 cv2.waitKey(0)
```

Ở đây tải một ảnh có tên example.png từ ổ cứng và hiển thị nó lên màn hình, như ảnh chụp màn hình từ Hình 4.7. Kết quả hiển thị như sau:

```
$ python load_display.py  
(248, 300, 3)
```

Ảnh này có chiều rộng là 300 pixel (số lượng cột), chiều cao là 248 pixel (số lượng hàng) và độ sâu 3 (số lượng kênh). Để truy cập một giá trị pixel riêng lẻ từ ảnh, sử dụng lập chỉ mục mảng NumPy đơn giản:

```
1 (b, g, r) = image[20, 100] # accesses pixel at x=100, y=20  
2 (b, g, r) = image[75, 25] # accesses pixel at x=25, y=75  
3 (b, g, r) = image[90, 85] # accesses pixel at x=85, y=90
```

Một lần nữa, lưu ý cách giá trị y được truyền vào trước giá trị x - cú pháp này ban đầu có thể cảm thấy không thoải mái, nhưng nó phù hợp với cách truy cập các giá trị trong ma trận: đầu tiên, chỉ định số hàng sau đó là số cột. Từ đó, được cung cấp một bộ dữ liệu mô tả cho các thành phần Đỏ, Xanh lục và Xanh lam của hình ảnh.

4.2.2.2. Thứ tự RGB và BGR

Điều quan trọng cần lưu ý là OpenCV lưu trữ các kênh RGB theo thứ tự ngược lại. Mặc dù thường nghĩ về các màu Đỏ, Xanh lục và Xanh lam, OpenCV thực sự lưu trữ các giá trị pixel theo thứ tự Xanh lam, Xanh lục, Đỏ.

Tại sao OpenCV làm điều này? Câu trả lời đơn giản là lý do lịch sử. Các nhà phát triển ban đầu của thư viện OpenCV đã chọn định dạng màu BGR vì thứ tự BGR rất phổ biến giữa các nhà sản xuất máy ảnh và các nhà phát triển phần mềm khác tại thời điểm đó [87].

Nói một cách đơn giản, thứ tự BGR này được thực hiện vì lý do lịch sử và bắt buộc phải lựa chọn. Nó là một điểm cần lưu ý khi lập trình với thư viện OpenCV.

4.2.3. Tỷ lệ mở rộng và tỷ lệ khung hình

Chia tỷ lệ (hoặc đơn giản là thay đổi kích thước hình ảnh) là quá trình tăng hoặc giảm kích thước ảnh về chiều rộng và chiều cao. Khi thay đổi kích thước ảnh, điều quan trọng là phải ghi nhớ tỷ lệ khung hình, đó là tỷ lệ giữa chiều rộng và chiều cao ảnh. Bỏ qua tỷ lệ khung hình có thể dẫn đến ảnh trông như bị nén và biến dạng, như trong Hình 4.8.

Ở bên trái là ảnh gốc. Phía trên và phía dưới là hai ảnh đã bị biến dạng do không giữ tỷ lệ khung hình. Kết quả cuối cùng là những ảnh này bị bóp méo và bị cắt xén. Để tránh khỏi vấn đề này, chỉ cần chia tỷ lệ chiều rộng và chiều cao ảnh bằng số lượng nhau khi thay đổi kích thước ảnh.

Từ quan điểm thẩm mỹ nghiêm ngặt, hầu như luôn muốn đảm bảo tỷ lệ khung ảnh được duy trì khi thay đổi kích thước ảnh - **nhung hướng dẫn này không phải là trường hợp cho học sâu**. Hầu hết các mạng nơ-ron và mạng nơ-ron tích chập được áp dụng cho nhiệm vụ phân loại ảnh đều giả sử đầu vào có kích thước cố định, nghĩa là kích thước tất cả các ảnh truyền qua mạng nơ-ron phải giống nhau. Các lựa chọn phổ biến cho kích thước ảnh chiều rộng và chiều cao được nhập vào mạng nơ-ron tích chập bao gồm 32×32 , 64×64 , 224×224 , 227×227 , 256×256 và 299×299 .

Giả sử rằng đang thiết kế một mạng sẽ cần phân loại 224 ảnh, tuy nhiên, tập dữ liệu bao gồm các ảnh là 312×234 , 800×600 và 770×300 , trong số các kích thước ảnh khác - phải xử lý những ảnh này như thế nào? Có phải chỉ đơn giản bỏ qua tỷ lệ khung hình và xử lý biến dạng (Hình 4.9, dưới cùng bên trái)? Hay nghĩ ra một sơ đồ khác để thay đổi kích thước ảnh, chẳng hạn như thay đổi kích thước ảnh dọc theo kích thước ngắn nhất nó và sau đó lấy phần cắt trung tâm (Hình 4.9) ?

Như có thể thấy ở phía dưới bên trái (tỷ lệ khung ảnh đã bị bỏ qua) dẫn đến một ảnh trông bị méo mó và bị nén. Sau đó, ở góc dưới bên phải, thấy rằng tỷ lệ khung ảnh đã được duy trì, nhưng phải cắt xén một phần ảnh. Điều này có thể đặc biệt bất lợi cho hệ thống phân loại ảnh nếu vô tình cắt một phần hoặc tất cả các đối tượng muốn xác định.

Phương pháp nào là tốt nhất? Đối với một số bộ dữ liệu, có thể chỉ cần bỏ qua tỷ lệ khung hình, bóp méo và nén ảnh trước khi đưa chúng vào mạng để huấn luyện. Trên các bộ dữ liệu khác, nó có lợi khi xử lý trước chúng bằng cách thay đổi kích thước dọc theo chiều ngắn nhất và sau đó cắt xén trung tâm.

312×234 ; Hệ số tỉ lệ = 1.33



236×86 ; Hệ số tỉ lệ = 2.75



100×118 ; Hệ số tỉ lệ = 0.84



Hình 4.8: **Trái :**Ảnh gốc. **Trên và dưới :**Kết quả là ảnh bị biến dạng sau khi thay đổi kích thước mà không giữ tỷ lệ khung hình (nghĩa là tỷ lệ chiều rộng so với chiều cao ảnh).



Hình 4.9: **Trên :**Ảnh đầu vào ban đầu. **Dưới bên trái :**Thay đổi kích thước ảnh thành 224×224 pixel bằng cách bỏ qua tỷ lệ khung hình. **Dưới bên phải :**Thay đổi kích thước ảnh 224×224 pixel bằng cách thay đổi kích thước đầu tiên dọc theo kích thước ngắn nhất và sau đó lấy trung tâm cắt.

4.2.4. Tóm tắt

Mục này đã xem xét các nguyên lý xây dựng khái niệm cơ bản một ảnh - pixel. Ảnh xám được mô tả bằng một vecto vô hướng duy nhất, cường độ / độ sáng của từng pixel. Không gian màu phổ biến nhất là không gian màu RGB trong đó mỗi pixel trong ảnh được biểu thị bằng 3 tuple : một pixel cho mỗi thành phần Đỏ, Xanh lục và Xanh lam tương ứng.

Các thư viện xử lý ảnh và thị giác máy tính trong ngôn ngữ lập trình Python tận dụng thư viện xử lý số NumPy và mô tả ảnh dưới dạng mảng NumPy đa chiều. Các mảng này có hình dạng (chiều cao, chiều rộng, chiều sâu).

Chiều cao được chỉ định đầu tiên vì chiều cao là số lượng hàng trong ma trận. Chiều rộng tiếp theo, vì nó là số cột trong ma trận. Cuối cùng, độ sâu là số lượng kênh trong ảnh. Trong không gian màu RGB, độ sâu được cố định bằng 3.

Cuối cùng, kết thúc mục này bằng cách xem xét tỷ lệ khung ảnh và vai trò của nó khi thay đổi kích thước ảnh làm đầu vào cho mạng nơ-ron và mạng nơ-ron tích chập.

4.3. KHÁI NIỆM CƠ BẢN VỀ PHÂN LOẠI ẢNH

“Một bức tranh đáng giá ngàn lời nói” - thành ngữ tiếng Anh.

Câu thành ngữ này rất phổ biến. Nó đơn giản có nghĩa là một ý tưởng phức tạp có thể được truyền đạt trong một ảnh duy nhất. Cho dù kiểm tra biểu đồ đầu tư danh mục đầu tư chứng khoán, xem xét sự lan truyền của một trò chơi bóng đá sắp tới, hoặc chỉ đơn giản là tham gia vào nghệ thuật và nét vẽ một bậc thầy hội họa, liên tục tiếp thu nội dung trực quan, diễn giải ý nghĩa và lưu trữ kiến thức để sử dụng sau.

Tuy nhiên, đối với máy tính, việc diễn giải nội dung ảnh ở một khía cạnh khác - tất cả các máy tính đều nhìn thấy ảnh là một ma trận lớn các con số. Nó không có ý tưởng liên quan đến những suy nghĩ, kiến thức hoặc ý nghĩa mà ảnh đang cố gắng truyền tải.

Để hiểu nội dung ảnh, phải áp dụng phân loại ảnh, đó là nhiệm vụ sử dụng thuật toán thị giác máy tính và máy học để trích xuất ý nghĩa từ ảnh. Hành động này có thể đơn giản như gán nhãn cho nội dung ảnh hoặc nâng cao như diễn giải nội dung ảnh và trả về câu có thể đọc được con người.

Phân loại ảnh là một lĩnh vực nghiên cứu rất lớn, bao gồm rất nhiều kỹ thuật - và với sự phổ biến học sâu, nó đang tiếp tục phát triển.

Phân loại ảnh và hiểu nội dung ảnh hiện đang và sẽ là lĩnh vực phụ trợ phổ biến nhất thị giác máy tính trong mười năm tới. Trong tương lai, sẽ thấy rằng các công ty như Google, Microsoft, Yahoo và các công ty khác sẽ nhanh chóng xây các công ty khởi nghiệp hiểu nội dung ảnh thành công. Người đọc sẽ thấy ngày càng nhiều ứng dụng tiêu dùng trên điện thoại thông minh có thể hiểu và giải thích nội dung ảnh. Ngay cả các cuộc chiến tranh có thể sẽ được trang bị máy bay không người lái được điều khiển tự động bằng các thuật toán thị giác máy tính.

Trong mục này, cung cấp một cái nhìn tổng quan cấp cao về phân loại ảnh là gì, cùng với nhiều thách thức mà thuật toán phân loại ảnh phải đối mặt cũng như xem xét ba loại học khác nhau liên quan đến phân loại ảnh và học máy.

Cuối cùng, sẽ kết thúc mục này bằng cách xem xét về bốn bước huấn luyện một mạng học sâu để phân loại ảnh và so sánh bốn bước này với trích xuất đặc trưng thủ công truyền thống.

4.3.1. Phân loại ảnh là gì ?

Phân loại ảnh là nhiệm vụ gán nhãn cho ảnh từ một bộ danh mục được xác định trước.

Thực tế, điều này có nghĩa là phân tích một ảnh đầu vào và trả về một nhãn phân loại ảnh. Nhãn luôn từ một tập hợp các danh mục có thể được xác định trước.

Ví dụ: Giả sử rằng tập hợp các danh mục có thể có bao gồm :

Chủng loại = {mèo, chó, gấu trúc}

Ảnh (Hình 4.10) cho mô hình phân loại :



Hình 4.10: Mục tiêu mô hình phân loại ảnh là lấy ảnh đầu vào và gán nhãn dựa trên tập hợp các danh mục được xác định trước.

Mục tiêu ở đây là lấy ảnh đầu vào này và gán nhãn cho nó từ các danh mục - trong trường hợp này là con chó.

Hệ thống phân loại cũng có thể gán nhiều nhãn cho ảnh thông qua xác suất, chẳng hạn như con chó: 95%; mèo: 4%; gấu trúc: 1%.

Chính thức hơn, với ảnh đầu vào là $W \times H$ pixel với ba kênh, Đỏ, Xanh lục và Xanh lam, mục tiêu là lấy ảnh pixel $W \times H \times 3 = N$ và tìm ra cách phân loại chính xác nội dung ảnh.

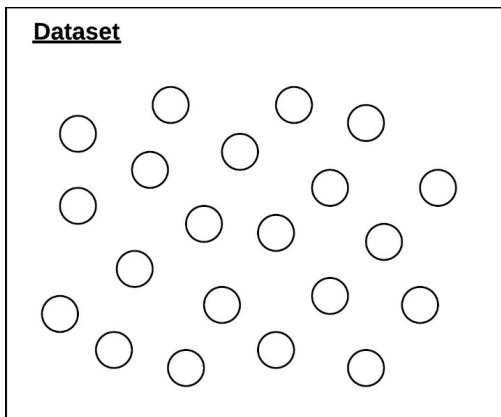
4.3.1.1. Lưu ý về thuật ngữ

Khi thực hiện học máy và học sâu, có một bộ dữ liệu mà chúng ta đang cố gắng rút ra thông tin từ nó. Mỗi ví dụ/mục trong tập dữ liệu (cho dù đó là dữ liệu ảnh, dữ liệu văn bản, dữ liệu âm thanh, v.v.) là một điểm dữ liệu. Do đó, một bộ dữ liệu là một tập hợp các điểm dữ liệu (Hình 4.11).

Mục tiêu là áp dụng thuật toán học máy và học sâu để khám phá các mẫu cơ bản trong bộ dữ liệu, cho phép phân loại chính xác các điểm dữ liệu mà thuật toán chưa gặp phải. Cần dành thời gian để làm quen với thuật ngữ này :

1. Trong bối cảnh phân loại ảnh, bộ dữ liệu là một bộ sưu tập các ảnh.
2. Do đó, mỗi ảnh là một điểm dữ liệu.

Sử dụng thuật ngữ ảnh và điểm dữ liệu thay thế cho nhau trong suốt phần còn lại cuốn sách này, vì vậy ghi nhớ điều này ngay bây giờ.



Hình 4.11: Tập dữ liệu (hình chữ nhật bên ngoài) là tập hợp các điểm dữ liệu (hình tròn).

4.3.1.2. Khoảng cách ngữ nghĩa

Nhìn vào hai bức ảnh (trên cùng) trong Hình 4.12. Nó khá là dễ để nói sự khác biệt giữa hai bức ảnh - rõ ràng có một con mèo ở bên trái và một con chó ở bên phải. Nhưng tất cả những gì máy tính nhìn thấy là hai ma trận pixel lớn (phía dưới).

Biết rằng tất cả các máy tính nhìn thấy là một ma trận pixel lớn, đi đến vấn đề khoảng cách ngữ nghĩa. Khoảng cách ngữ nghĩa là sự khác biệt giữa cách con người cảm nhận nội dung ảnh so với cách ảnh có thể được thể hiện theo cách mà máy tính có thể hiểu được quy trình.

Một lần nữa, kiểm tra trực quan nhanh hai bức ảnh trên có thể cho thấy sự khác biệt giữa hai loài động vật. Nhưng trong thực tế, máy tính không có ý tưởng về những con vật trong ảnh để bắt đầu. Để làm rõ điểm này, xem Hình 4.13, có một bức ảnh về một bãi biển yên tĩnh.

Có thể mô tả ảnh như sau :

Không gian : Bầu trời ở trên cùng ảnh và cát / đại dương ở dưới cùng.

Màu sắc : Bầu trời có màu xanh thẫm, nước biển có màu xanh nhạt hơn bầu trời, trong khi cát tan.

Kết cấu : Bầu trời có hoa văn tương đối đồng đều, trong khi cát rất thô.

Làm thế nào để mã hóa tất cả các thông tin này theo cách mà một máy tính có thể hiểu nó? Câu trả lời là áp dụng trích xuất đặc trưng để định lượng nội dung ảnh. Trích xuất đặc trưng là quá trình lấy ảnh đầu vào, áp dụng thuật toán và thu được một vectơ đặc trưng (nghĩa là danh sách các số) định lượng ảnh.

Để thực hiện quá trình này, có thể xem xét áp dụng các đặc trưng được thiết kế bằng tay như HOG, LBP hoặc các phương pháp tiếp cận truyền thống khác để định lượng ảnh. Một phương pháp khác được thực hiện trong cuốn sách này là áp dụng học sâu để tự động tìm hiểu một tập hợp các đặc trưng có thể được sử dụng để định lượng và cuối cùng gắn nhãn nội dung ảnh.

Tuy nhiên, nó không đơn giản bởi vì một khi bắt đầu kiểm tra ảnh trong thế giới thực, phải đối mặt với nhiều thử thách.



*	151	121	1	93	165	204	14	214	28	235	*	29	142	142	75	22	109	111	28	6	5
	62	67	17	234	27	1	221	37	189	141		137	168	41	206	100	70	219	127	114	191
	20	168	155	113	178	228	25	130	139	221		205	154	226	14	89	86	242	67	203	15
	236	136	158	230	18	5	165	17	30	155		247	47	128	123	253	229	181	251	232	28
	174	148	93	70	95	106	151	10	160	214		68	75	24	99	93	63	215	222	102	180
	103	126	58	16	138	136	98	202	42	233		206	246	85	183	215	3	62	64	77	216
	235	103	52	37	94	104	173	86	223	113		126	80	165	149	196	75	186	60	179	193
	212	15	179	139	48	232	194	46	174	37		44	253	164	253	14	216	175	30	46	254
	119	81	241	172	95	170	29	210	22	194		137	23	33	203	241	21	144	63	244	188
	129	19	33	253	229	5	152	233	52	44		32	214	142	121	249	109	99	232	183	71
	88	200	194	185	148	200	223	190	164	102		45	36	152	27	190	137	61	1	237	247
	113	16	220	215	143	104	247	29	97	203		1	14	241	70	2	30	151	67	169	205
	9	210	102	246	75	9	158	184	184	129		32	80	102	32	99	169	91	166	73	214
	124	52	76	148	249	107	65	216	187	181		186	219	9	283	209	240	40	249	119	122
	6	251	52	208	46	65	185	38	77	240		177	252	38	203	119	0	217	139	139	157
	150	194	28	206	148	197	208	28	74	93		154	145	49	251	150	185	235	23	230	156
	33	183	248	153	168	205	146	100	254	218		157	168	223	60	247	118	5	180	16	206
	130	53	128	212	61	226	201	110	140	183		102	208	195	246	140	138	54	191	139	79
	165	246	22	102	151	213	40	138	8	93		17	233	85	169	166	24	49	40	160	97
	152	251	101	230	23	162	70	238	75	24		84	242	247	144	203	3	19	24	198	88
	187	105	152	83	167	98	125	180	136	121		67	67	185	98	123	106	168	105	127	153
	139	197	55	269	28	124	208	288	104	40		37	113	214	252	203	80	146	211	7	16
	123	19	144	223	62	253	202	188	47	242		142	241	66	86	214	133	146	253	189	200
	220	144	31	16	136	123	227	62	183	163		67	215	174	111	189	54	144	56	59	163

Hình 4.12: Trên : Bộ não có thể thấy rõ sự khác biệt giữa ảnh có chứa một con mèo và ảnh có chứa một con chó. Dưới : Tuy nhiên, tất cả các máy tính «nhìn thấy» là một ma trận lớn về số. Sự khác biệt giữa cách cảm nhận một ảnh và cách ảnh được thể hiện (một ma trận số) được gọi là khoảng cách ngữ nghĩa.

4.3.1.3. Thách thức

Nếu khoảng cách ngữ nghĩa không đủ một vấn đề, cũng phải xử lý các yếu tố biến đổi [33] trong cách ảnh hoặc đổi tượng xuất hiện.

Để bắt đầu, chúng ta có điểm nhìn thay đổi (viewpoint variation). Trong đó một đối tượng có thể được định hướng/xoay theo nhiều chiều liên quan đến cách chụp và đối tượng chụp.

Ngoài ra, cũng phải tính đến sự thay đổi tỷ lệ. Phương pháp phân loại ảnh phải được chấp nhận đối với các loại biến thể tỷ lệ này.

Một trong những biến thể khó nhất để giải thích là biến dạng.

Phân loại ảnh cũng sẽ có thể xử lý các phần, trong đó các phần lớn đổi tượng muôn phân loại được ẩn khỏi chế độ xem trong ảnh (Hình 4.16 ở phía dưới). Ở bên trái, phải có một ảnh một con chó. Và ở bên phải, có một bức ảnh cùng một con chó, nhưng chú ý cách con chó đang nghỉ ngơi bên dưới vỏ bọc, bị che khuất khỏi thị giác. Con chó vẫn rõ ràng trong cả hai ảnh - nó chỉ nhìn thấy rõ hơn ở một ảnh so với ảnh kia. Các thuật toán phân loại ảnh vẫn có thể phát hiện và dán nhãn cho sự hiện diện con chó trong cả hai ảnh.

Tiếp tục, cũng phải tính đến sự lộn xộn của hậu cảnh. Những ảnh này có nhiễu rất nhiều, và có rất nhiều thứ đang diễn ra trong đó. Chỉ cần quan tâm đến một đối tượng cụ thể trong ảnh, tuy nhiên, còn có tất cả các nhiễu khác. Nếu nó không dễ để thực hiện đối với chúng ta, tưởng tượng nó khó như thế nào đối với một máy tính không có hiểu biết về ngữ nghĩa ảnh.

Cuối cùng, có một biến thể trong lớp. Ví dụ điển hình về sự biến đổi trong lớp trong thị giác máy tính đang hiển thị sự đa dạng hóa của những chiếc ghế. Từ những chiếc ghế thoải mái dùng để ngồi đọc một cuốn sách, cho đến những chiếc ghế xếp trên bàn bếp cho những buổi họp mặt gia đình, đến những chiếc ghế trang trí nghệ thuật cực kỳ hiện đại được tìm thấy trong những ngôi nhà bề thế, thì một chiếc ghế vẫn là một chiếc ghế - và thuật toán phân loại ảnh phải có thể phân loại tất cả các biến thể một cách chính xác.

Có bắt đầu cảm thấy một chút choáng ngợp với sự phức tạp của việc xây dựng một bộ phân loại ảnh? Thật không may, nó chỉ trở nên tồi tệ hơn - nó không đủ để hệ thống phân loại ảnh mạnh mẽ với các biến thể này một cách độc lập, nhưng hệ thống cũng phải xử lý nhiều biến thể kết hợp với nhau.

Vậy làm thế nào để chiếm số lượng biến thể đáng kinh ngạc như vậy

trong các đối tượng hoặc hình ảnh? Nói chung, cỗ găng đóng khung vấn đề tốt nhất có thể. Đưa ra các giả định liên quan đến nội dung ảnh và các biến thể muôn thể hiện. Ngoài ra, cũng xem xét phạm vi dự án - mục tiêu cuối cùng là gì? Và đang cỗ găng xây dựng cái gì?

Thị giác máy tính phân loại ảnh và hệ thống học sâu được triển khai trong thế giới thực, đưa ra các giả định và cân nhắc cẩn thận trước khi một chương trình được viết.

Nếu tiếp cận quá rộng, như muôn phân loại và phát hiện từng đối tượng trong bếp, (nơi có thể có hàng trăm đối tượng có thể) thì hệ thống phân loại khó có thể hoạt động tốt trừ khi có nhiều năm kinh nghiệm xây dựng phân loại ảnh - và thậm chí sau đó, không có gì đảm bảo cho sự thành công của dự án.

Nhưng nếu đóng khung vấn đề và thu hẹp phạm vi, chẳng hạn như muôn nhận ra bếp và tủ lạnh, thì hệ thống phải có nhiều khả năng hoạt động và chính xác, đặc biệt nếu đây là lần đầu tiên làm việc với phân loại ảnh và học sâu.

Điểm quan trọng ở đây là luôn xem xét phạm vi phân loại ảnh. Mặc dù học sâu và mạng nơ-ron kết hợp đã chứng minh sức mạnh đáng kể và khả năng phân loại dưới nhiều thách thức khác nhau, vẫn nên giữ phạm vi dự án chặt chẽ và được xác định rõ nhất có thể.

Biết rằng ImageNet [80], bộ dữ liệu chuẩn thực tế cho các thuật toán phân loại ảnh, bao gồm 1.000 đối tượng gấp trong cuộc sống hàng ngày - và bộ dữ liệu này vẫn được các nhà nghiên cứu tích cực sử dụng để cỗ găng thúc đẩy kết quả của học sâu.

Học sâu không phải là phép thuật. Thay vào đó, học sâu mạnh mẽ và hữu ích khi sử dụng đúng cách, nhưng nguy hiểm nếu được sử dụng mà không có sự cân nhắc thích hợp. Trong suốt phần còn lại cuốn sách này, sẽ hướng dẫn về hành trình học hỏi sâu và giúp chỉ ra khi nào nên tiếp cận với các công cụ quyền lực này và khi nào nên tham khảo một cách tiếp cận đơn giản hơn (hoặc đề cập đến nếu vấn đề không hợp lý để giải quyết cho ảnh phân loại).

4.3.2. Các kiểu học

Có ba loại học có thể gặp phải trong quá trình học máy và học sâu : học có giám sát, học không giám sát và học bán giám sát. Cuốn sách này tập trung chủ yếu vào việc học có giám sát trong bối cảnh học sâu. Tuy nhiên, mô tả cả ba loại học này được trình bày dưới đây.

4.3.2.1. Học có giám sát

Các thuật toán học có giám sát phổ biến bao gồm hồi quy logistic, vectơ hỗ trợ máy học (SVM) [66, 88], Random Forest [89] và mạng nơ-ron nhân tạo.

Trong bối cảnh phân loại ảnh, giả sử tập dữ liệu ảnh bao gồm các ảnh cùng với lớp nhãn tương ứng với chúng mà có thể sử dụng để dạy cho trình phân loại máy học mỗi loại trông như thế nào. Nếu trình phân loại đưa ra dự đoán không chính xác, thì có thể áp dụng các phương pháp để sửa lỗi sai nó.

Sự khác biệt giữa học có giám sát, không giám sát và bán giám sát có thể được hiểu rõ nhất bằng cách xem ví dụ trong Bảng 4.1. Cột đầu tiên trong bảng là nhãn được liên kết với một ảnh cụ thể. Sáu cột còn lại tương ứng với vectơ đặc trưng cho từng điểm dữ liệu - ở đây, đã chọn định lượng nội dung ảnh bằng cách tính độ lệch trung bình và độ lệch chuẩn cho từng kênh màu RGB, tương ứng.

Thuật toán học có giám sát sẽ đưa ra dự đoán về từng vectơ đặc trưng này và nếu nó đưa ra dự đoán không chính xác, sẽ cố gắng sửa nó bằng cách cho nó biết nhãn chính xác thực sự là gì. Quá trình này sau đó sẽ tiếp tục cho đến khi đạt được tiêu chí dừng mong muốn, chẳng hạn như độ chính xác, số lần lặp quá trình học hoặc đơn giản là một lượng thời gian treo tường tùy ý.

Bảng 4.1: Bảng dữ liệu chứa cả lớp nhãn (chó hoặc mèo) và các vectơ đặc trưng cho từng điểm dữ liệu (độ lệch trung bình và độ lệch chuẩn từng kênh màu Đỏ, Xanh lục và Xanh lam). Đây là một ví dụ về một nhiệm vụ phân loại được giám sát.

Label	R_μ	G_μ	B_μ	R_σ	G_σ	B_σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
Cat	120.23	121.59	181.43	145.58	69.13	116.91
Cat	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
Dog	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

4.3.2.2. Học không giám sát

Ngược lại với học có giám sát, học không giám sát (đôi khi được gọi

là học tự học) không có nhãn liên quan đến dữ liệu đầu vào và do đó không thể sửa mô hình nếu dự đoán không chính xác.

Quay trở lại ví dụ về bảng tính, chuyển đổi một vấn đề học có giám sát sang một vấn đề học không giám sát cũng đơn giản như việc loại bỏ cột nhãn (Bảng 4.2).

Hầu hết các thuật toán học không giám sát đều thành công nhất khi có thể tìm hiểu cấu trúc cơ bản của bộ dữ liệu và sau đó, áp dụng các đặc trưng đã học cho một vấn đề học có giám sát khi có quá ít dữ liệu được dán nhãn.

Các thuật toán học máy cổ điển cho việc học không giám sát bao gồm Principle Component Analysis (PCA) và phân cụm k-mean. Cụ thể đối với các mạng nơ-ron, tự mã hóa (self-autoencoder), Self-Organizing-Maps (SOM) và lý thuyết cộng hưởng thích nghi (Adaptive Resonance Theory) được áp dụng cho việc học không giám sát. Học không giám sát là một lĩnh vực nghiên cứu cực kỳ hấp dẫn và vẫn chưa được giải quyết. Vì vậy, cuốn sách này không tập trung vào việc học không giám sát.

Bảng 4.2: Các thuật toán học không giám sát cố gắng tìm hiểu các mẫu cơ bản trong bộ dữ liệu mà không có lớp nhãn. Trong ví dụ này, đã loại bỏ cột lớp nhãn, do đó biến nhiệm vụ này thành vấn đề học không giám sát.

R μ	G μ	B μ	R σ	G σ	B σ
57.61	41.36	123.44	158.33	149.86	93.33
120.23	121.59	181.43	145.58	69.13	116.91
124.15	193.35	65.77	23.63	193.74	162.70
100.28	163.82	104.81	19.62	117.07	21.11
177.43	22.31	149.49	197.41	18.99	187.78
149.73	87.17	187.97	50.27	87.15	36.65

Bảng 4.3: Khi thực hiện học bán giám sát, chỉ có nhãn cho một tập hợp con các vectơ ảnh / đặc trưng và phải cố gắng gắn nhãn các điểm dữ liệu khác để sử dụng chúng làm dữ liệu huấn luyện bổ sung.

Label	R μ	G μ	B μ	R σ	G σ	B σ
Cat	57.61	41.36	123.44	158.33	149.86	93.33
?	120.23	121.59	181.43	145.58	69.13	116.91
?	124.15	193.35	65.77	23.63	193.74	162.70
Dog	100.28	163.82	104.81	19.62	117.07	21.11
?	177.43	22.31	149.49	197.41	18.99	187.78
Dog	149.73	87.17	187.97	50.27	87.15	36.65

4.3.2.3. Học bán giám sát

Vậy, điều gì xảy ra nếu chỉ có một số nhãn được liên kết với dữ liệu và không có nhãn cho nhãn kia? Có cách nào để có thể áp dụng một số kết hợp giữa học có giám sát và không giám sát mà vẫn có thể phân loại từng điểm dữ liệu không? Hóa ra câu trả lời là có - chỉ cần áp dụng phương pháp học bán giám sát.

Quay trở lại ví dụ về bảng tính, nói rằng chỉ có nhãn cho một phần nhỏ dữ liệu đầu vào (Bảng 4.3). Thuật toán học bán giám sát sẽ lấy các phần dữ liệu đã biết, phân tích chúng và cố gắng gắn nhãn cho từng điểm dữ liệu chưa được gắn nhãn để sử dụng làm dữ liệu huấn luyện bổ sung. Quá trình này có thể lặp lại cho nhiều lần lặp lại khi thuật toán bán giám sát tìm hiểu cấu trúc bộ dữ liệu, dữ liệu để đưa ra dự đoán chính xác hơn và tạo ra dữ liệu huấn luyện đáng tin cậy hơn.

Học bán giám sát đặc biệt hữu ích trong thị giác máy tính, nơi thường tồn thời gian, tẻ nhạt và tốn kém (ít nhất là theo giờ làm việc) để dán nhãn cho từng ảnh trong tập huấn luyện. Trong trường hợp đơn giản là không có thời gian hoặc tài nguyên để gắn nhãn cho từng ảnh riêng lẻ, chỉ có thể gắn nhãn một phần nhỏ dữ liệu và sử dụng phương pháp học bán giám sát để gắn nhãn và phân loại các ảnh còn lại.

Các thuật toán học bán giám sát thường phải đối mặt với các bộ dữ liệu đầu vào có ít nhất một nhãn để xác định chính xác phân loại. Thông thường, thuật toán huấn luyện được giám sát càng được dán nhãn chính xác, dự đoán càng chính xác (điều này đặc biệt đúng với các thuật toán học sâu).

Khi số lượng dữ liệu huấn luyện giảm, độ chính xác chắc chắn bị ảnh hưởng. Học bán giám sát có mối quan hệ giữa độ chính xác và lượng dữ liệu, cố gắng giữ độ chính xác phân loại trong giới hạn cho phép trong khi giảm đáng kể lượng dữ liệu huấn luyện cần thiết để xây dựng mô hình - kết quả cuối cùng là phân loại chính xác (nhưng thông thường không phải là chính xác như một bộ phân loại có giám sát) với ít dữ liệu nỗ lực và huấn luyện hơn. Các lựa chọn phổ biến cho học bán giám sát bao gồm truyền bá nhãn [90], lan truyền nhãn [91], mạng hình thang [92] và đồng học / đồng huấn luyện [93].

Một lần nữa, chủ yếu tập trung vào việc học có giám sát trong cuốn sách này, vì cả học không giám sát và bán giám sát trong bối cảnh học sâu cho thị giác máy tính vẫn là những chủ đề nghiên cứu rất tích cực mà không có hướng dẫn rõ ràng về phương pháp sử dụng.

4.3.3. Phân loại học sâu

Dựa trên hai phần trước về phân loại ảnh và các loại thuật toán học, có thể bắt đầu cảm thấy hơi bị hấp dẫn với các điều khoản mới, các cân nhắc và những gì có vẻ là một biến thể không thể vượt qua trong việc xây dựng một trình phân loại ảnh, nhưng sự thật là việc xây dựng một trình phân loại ảnh khá đơn giản, một khi hiểu được quy trình.

Trong phần này, sẽ xem xét một sự thay đổi quan trọng trong suy nghĩ nhưng cần thực hiện khi làm việc với máy học. Từ đó, sẽ xem xét bốn bước xây dựng bộ phân loại ảnh dựa trên học sâu cũng như so sánh và đối chiếu việc học máy dựa trên đặc trưng truyền thống so với học sâu từ đầu đến cuối.

4.3.3.1. Sự thay đổi trong tư duy

Trước khi gấp phải bắt cứ điều gì phức tạp, bắt đầu với một thứ mà rất quen thuộc: chuỗi Fibonacci.

Chuỗi Fibonacci là một chuỗi các số trong đó số tiếp theo chuỗi được tìm thấy bằng cách tính tổng hai số nguyên trước nó. Ví dụ, với dãy 0, 1, 1, số tiếp theo được tìm thấy bằng cách thêm $1 + 1 = 2$. Tương tự, cho 0, 1, 1, 2, số nguyên tiếp theo trong chuỗi là $1 + 2 = 3$.

Theo mô hình đó, số ít các số đầu tiên trong chuỗi như sau:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Tất nhiên, cũng có thể định nghĩa mẫu này trong một hàm Python (cực kỳ không tối ưu hóa) bằng cách sử dụng đệ quy:

```
1 >>> def fib(n):
2     ...
3         if n == 0:
4             ...
5                 return 0
6         elif n == 1:
7             ...
8                 return 1
9         else:
10             ...
11                 return fib(n-1) + fib(n-2)
12
13 >>>
```

Sử dụng mã này, có thể tính số thứ n trong chuỗi bằng cách cung cấp giá trị n cho hàm sợi. Ví dụ: để tính toán số thứ 7 trong chuỗi Fibonacci:

```
9 >>> fib(7)
10 13
```

Và số thứ 13:

```
11 >>> fib(13)
12 233
```

Và cuối cùng là số thứ 35:

```
13 >>> fib(35)
14 9227465
```

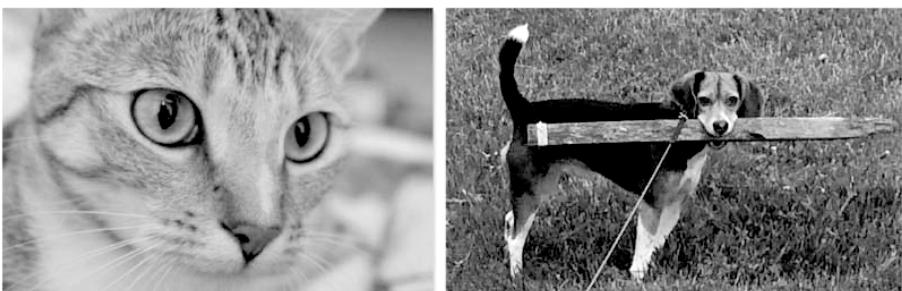
Như có thể thấy, chuỗi Fibonacci rất đơn giản và là một ví dụ về một nhóm các hàm :

1. Chấp nhận một đầu vào, trả về một đầu ra.
2. Quá trình được xác định rõ.
3. Đầu ra dễ dàng kiểm chứng tính chính xác.
4. Cho mượn chính nó để mã sửa lỗi và bộ kiểm tra.

Tính toán một chuỗi Fibonacci, lấy dữ liệu từ cơ sở dữ liệu hoặc tính toán độ lệch trung bình và độ lệch chuẩn từ danh sách các số, các hàm này đều được xác định rõ và dễ dàng xác minh tính chính xác.

Thật không may, đây không phải là trường hợp để học sâu và phân loại ảnh.

Nhớ lại từ mục 3.1.2, trong đó các ảnh một con mèo và một con chó, như Hình 4.13. Thủ tướng tượng đang cố viết một hàm thủ tục không chỉ có thể nói sự khác biệt giữa hai bức ảnh này mà bất kỳ bức ảnh nào về một con mèo và một con chó. Làm thế nào để hoàn thành nhiệm vụ này? Có kiểm tra các giá trị pixel riêng lẻ ở các mức phối hợp (x, y) khác nhau không? Viết hàng trăm câu lệnh if/other? Và làm thế nào sẽ duy trì và xác minh tính đúng đắn hệ thống dựa trên quy tắc lớn? Câu trả lời ngắn gọn là: không.



Hình 4.13: Làm thế nào có thể viết về một phần mềm để nhận ra sự khác biệt giữa chó và mèo trong ảnh? Kiểm tra các giá trị pixel riêng lẻ? Có một cách tiếp cận dựa trên quy tắc? thử viết (và duy trì) hàng trăm câu lệnh if/else?

Không giống như mã hóa một thuật toán để tính toán chuỗi Fibonacci hoặc sắp xếp danh sách các số, nó không trực quan hoặc rõ ràng làm thế nào để tạo ra một thuật toán để phân biệt sự khác nhau giữa ảnh mèo và chó. Do đó, thay vì cố gắng xây dựng một hệ thống dựa trên quy tắc để mô tả xem mỗi thể loại trông như thế nào, thì thay vào đó, có thể thực hiện một cách tiếp cận dựa trên dữ liệu bằng cách cung cấp các ví dụ về mỗi loại trông như thế nào và sau đó dạy thuật toán nhận ra sự khác biệt giữa các danh mục sử dụng các ví dụ này.

Những ví dụ này là tập dữ liệu huấn luyện về ảnh được dán nhãn, trong đó mỗi điểm dữ liệu trong tập dữ liệu huấn luyện bao gồm:

1. Một hình ảnh
2. Nhãn hoặc danh mục hình ảnh (tức là chó, mèo, gấu trúc, v.v.)

Một lần nữa, điều quan trọng là mỗi ảnh này đều có nhãn liên quan đến chúng bởi vì thuật toán học có giám sát sẽ cần phải xem các nhãn này để dạy chính cách nhận biết từng loại. ghi nhớ điều này, để đi trước và thực hiện bốn bước để xây dựng một mô hình học sâu.

BUỚC 1: TẬP HỢP DỮ LIỆU

Thành phần đầu tiên để xây dựng một mạng học sâu là thu thập dữ liệu ban đầu. Cần bản thân ảnh cũng như các nhãn liên kết với mỗi ảnh. Các nhãn này phải đến từ một tập hợp hữu hạn các danh mục, chẳng hạn như thể loại = chó, mèo, gấu trúc.

Hơn nữa, số lượng ảnh cho mỗi danh mục phải gần như thống nhất (nghĩa là, cùng một số ví dụ cho mỗi danh mục). Nếu có số lượng ảnh con mèo gấp đôi so với ảnh con chó và gấp năm lần số lượng ảnh gấu trúc so với ảnh con mèo, thì bộ phân loại sẽ trở nên sai lệch khi đưa vào các danh mục được đại diện nhiều này.

Mất cân bằng lớp là một vấn đề phổ biến trong học máy và tồn tại một số cách để khắc phục nó, nhưng cần ghi nhớ phương pháp tốt nhất để tránh các vấn đề học do mất cân bằng lớp chỉ đơn giản là tránh mất cân bằng lớp hoàn toàn.

BUỚC 2: PHÂN CHIA DỮ LIỆU

Bây giờ có tập dữ liệu ban đầu, cần chia nó thành hai phần:

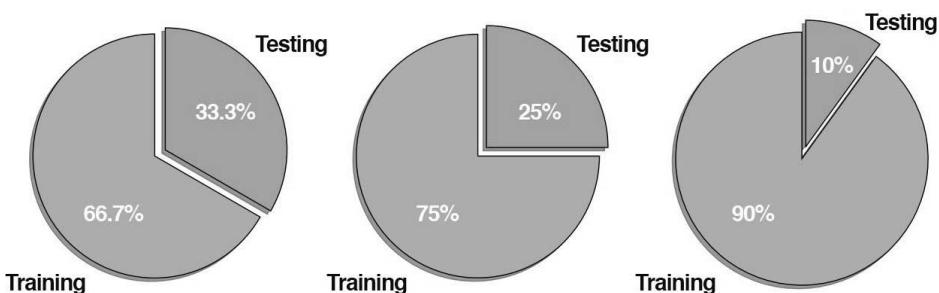
1. Một bộ huấn luyện
2. Một bộ kiểm tra

Một bộ huấn luyện được sử dụng bởi trình phân loại để tìm hiểu về

việc mỗi loại trông như thế nào bằng cách đưa ra dự đoán về dữ liệu đầu vào và sau đó tự sửa khi dự đoán sai. Sau khi trình phân loại đã được huấn luyện, có thể đánh giá hiệu suất trên bộ kiểm tra.

Điều cực kỳ quan trọng là tập huấn luyện và tập kiểm tra độc lập với nhau và không trùng nhau. Nếu sử dụng bộ tự kiểm tra như một phần dữ liệu huấn luyện, thì trình phân loại có một lợi thế không công bằng vì nó đã thấy các ví dụ kiểm tra trước đó và đã học được từ họ. Thay vào đó, phải giữ bộ kiểm tra này tách biệt hoàn toàn với quy trình huấn luyện và chỉ sử dụng nó để đánh giá mạng.

Kích thước phân chia phổ biến cho các bộ huấn luyện và kiểm tra bao gồm 66,6% / 33,3%, 75%/25% và 90%/10%, tương ứng. (Hình 4.14):



Hình 4.14: Ví dụ về phân chia dữ liệu kiểm tra và huấn luyện phổ biến

Những phân chia dữ liệu này có ý nghĩa, nhưng nếu có các tham số để điều chỉnh thì sao? Mạng nơ-ron có một số nút và đòn bẩy (ví dụ: tốc độ học, phân rã, chính quy, v.v.) cần được điều chỉnh và quay số để đạt được hiệu suất tối ưu, gọi các loại siêu tham số này, và điều quan trọng là chúng phải được đặt đúng.

Trong thực tế, cần kiểm tra một loạt các siêu tham số này và xác định tập hợp các tham số hoạt động tốt nhất. Có thể bị cám dỗ bởi việc sử dụng dữ liệu kiểm tra để điều chỉnh các giá trị này, nhưng một lần nữa, đây là một điều không nên! Bộ kiểm tra chỉ được sử dụng để đánh giá hiệu suất mạng.

Thay vào đó, nên tạo phân chia dữ liệu thứ ba được gọi là bộ xác thực. Tập hợp dữ liệu này (thông thường) xuất phát từ dữ liệu huấn luyện và được sử dụng làm dữ liệu kiểm tra giả mạo, vì vậy có thể điều chỉnh. Chỉ sau khi xác định các giá trị siêu tham số bằng cách sử dụng bộ xác thực, mới chuyển sang thu thập kết quả chính xác cuối cùng trong dữ liệu kiểm tra.

Thường phân bổ khoảng 10-20% dữ liệu huấn luyện để xác nhận.

Nếu việc chia dữ liệu thành các khối có vẻ phức tạp, thì thực tế là không. Như sẽ thấy trong mục tiếp theo, nó khá đơn giản và có thể được thực hiện chỉ với một dòng chương trình duy nhất nhờ vào thư viện scikit-learn.

BUỚC 3: HUẤN LUYỆN MẠNG

Với tập hợp ảnh huấn luyện, bây giờ có thể huấn luyện mạng. Mục tiêu ở đây là để mạng học cách nhận biết từng loại trong dữ liệu được dán nhãn. Khi mô hình mắc lỗi, nó học được từ sai lầm này và tự cải thiện.

Vì vậy, làm thế nào để thực sự học làm việc? Nói chung, áp dụng một hình thức gradient descent, như đã xem xét trong mục 3.2. Phần còn lại cuốn sách này được dành riêng để trình bày cách huấn luyện mạng nơ-ron từ đầu, vì vậy sẽ trì hoãn một cuộc xem xét chi tiết về quá trình huấn luyện cho đến lúc đó.

BUỚC 4: ĐÁNH GIÁ

Cuối cùng, cần đánh giá mạng được huấn luyện. Đôi với mỗi ảnh trong bộ kiểm tra, trình bày chúng cho mạng và yêu cầu nó dự đoán nhãn ảnh là gì. Sau đó, lập bảng dự đoán mô hình cho một ảnh trong bộ kiểm tra. Cuối cùng, những dự đoán mô hình này được so sánh với các nhãn thực tế từ bộ kiểm tra. Các nhãn sự thật thể hiện thể loại ảnh thực sự là gì. Từ đó, có thể tính toán số lượng dự đoán mà trình phân loại có được và tính toán các báo cáo tổng hợp như độ chính xác, được sử dụng để định lượng toàn bộ hiệu suất mạng.

4.3.3.2. So sánh học dựa trên học đặc trưng so với học sâu để phân loại ảnh

Theo cách tiếp cận truyền thống, dựa trên đặc trưng để phân loại ảnh, thực sự có một bước được chèn giữa bước 2 và bước 3 - bước này là trích xuất đặc trưng. Trong giai đoạn này, áp dụng các thuật toán được thiết kế bằng tay như HOG [45], LBPs [2], v.v. để định lượng nội dung ảnh dựa trên một thành phần cụ thể ảnh mà muốn mã hóa (ví dụ: hình dạng, màu sắc, kết cấu). Với các đặc trưng này, sau đó tiến hành huấn luyện trình phân loại và đánh giá nó.

Khi xây dựng mạng nơ-ron tích chập, thực sự có thể bỏ qua bước trích xuất đặc trưng. Lý do cho điều này là bởi vì CNN là mô hình đầu cuối. Trình bày dữ liệu đầu vào thô (pixel) cho mạng. Mạng sau đó học các bộ lọc bên trong các lớp ẩn nó có thể được sử dụng để phân biệt giữa các lớp đối tượng. Đầu ra mạng sau đó là phân phối xác suất trên lớp nhãn.

Một trong những khía cạnh thú vị việc sử dụng CNN là không còn phải lo lắng về các đặc trưng được thiết kế bằng tay - có thể để mạng tự tìm

hiểu các đặc trưng thay thế. Tuy nhiên, sự đánh đổi này không phải trả giá. Huấn luyện CNN có thể là một quá trình không tầm thường, vì vậy chuẩn bị dành thời gian đáng kể để làm quen với trải nghiệm và chạy nhiều kiểm tra để xác định những gì hoạt động và không hoạt động.

4.3.3. Điều gì xảy ra khi dự đoán không chính xác?

Chắc chắn, sẽ huấn luyện một mạng học sâu trên tập huấn luyện, đánh giá nó trên tập kiểm tra (nhận thấy rằng nó đạt được độ chính xác cao), và sau đó áp dụng nó cho các ảnh nằm ngoài cả tập huấn luyện và kiểm tra - chỉ để thấy rằng mạng hoạt động kém.

Vấn đề này được gọi là khái quát hóa, khả năng mạng tổng quát hóa và dự đoán chính xác lớp nhãn ảnh không tồn tại như một phần dữ liệu kiểm tra hoặc huấn luyện nó.

Khả năng tổng quát hóa mạng là khía cạnh quan trọng nhất nghiên cứu học sâu - nếu có thể huấn luyện các mạng có thể khái quát hóa các bộ dữ liệu bên ngoài mà không cần huấn luyện lại hoặc tinh chỉnh, sẽ có những bước tiến lớn trong học máy, cho phép các mạng được sử dụng lại trong một loạt các lĩnh vực. Khả năng tổng quát hóa một mạng sẽ được xem xét nhiều lần trong cuốn sách này, nhưng tác giả muốn đưa ra chủ đề ngay bây giờ vì chắc chắn sẽ gặp phải các vấn đề khái quát hóa, đặc biệt là khi học được các bài học sâu.

Thay vì trở nên thất vọng với mô hình không phân loại chính xác một ảnh, xem xét tập hợp các yếu tố biến thể được đề cập ở trên. Tập dữ liệu huấn luyện có phản ánh chính xác các ví dụ về các yếu tố biến đổi này không? Nếu không, sẽ cần thu thập thêm dữ liệu huấn luyện (và đọc phần còn lại cuốn sách này để tìm hiểu các kỹ thuật khác để chống lại sự khái quát hóa).

4.3.4. Tóm tắt

Trong mục này, đã tìm hiểu phân loại ảnh là gì và tại sao nó là một nhiệm vụ đầy thách thức để máy tính thực hiện tốt (mặc dù con người thực hiện nó bằng trực giác mà dường như không cần nỗ lực). Sau đó, đã xem xét về ba loại chính học máy, học có giám sát, học không giám sát, học bán giám sát - cuốn sách này chủ yếu tập trung vào học có giám sát, trong đó có cả các ví dụ huấn luyện và lớp nhãn liên quan đến chúng. Học bán giám sát và học không giám sát là cả hai lĩnh vực nghiên cứu mở cho học sâu (và trong học máy nói chung).

Cuối cùng, đã xem xét bốn bước trong phân loại học sâu. Các bước này bao gồm thu thập dữ liệu, chia dữ liệu thành các bước huấn

luyện, kiểm tra và xác thực, huấn luyện mạng và cuối cùng là đánh giá mô hình.

Không giống như các cách tiếp cận dựa trên đặc trưng truyền thống yêu cầu sử dụng các thuật toán thủ công để trích xuất các đặc trưng từ một ảnh, các mô hình phân loại ảnh, chẳng hạn như mạng nơ-ron tích chập, là các phân loại đầu cuối để tìm hiểu các đặc trưng có thể được sử dụng để phân biệt lớp ảnh.

4.4. CÁC BỘ DỮ LIỆU ĐỂ PHÂN LOẠI ẢNH

Tại thời điểm này, đã quen với các nguyên tắc cơ bản phân loại ảnh nhưng trước khi đi sâu vào bất kỳ mã nào xem xét cách thực sự lấy dữ liệu và xây dựng trình phân loại ảnh, trước tiên xem xét các bộ dữ liệu mà sẽ thấy trong “Mạng nơ-ron học sâu và ứng dụng”.

Một số trong các bộ dữ liệu này về cơ bản là đã được giải quyết, cho phép có được các bộ phân loại có độ chính xác cực cao (độ chính xác $> 95\%$) với ít chi phí. Các bộ dữ liệu khác đại diện cho các loại kỹ thuật thị giác máy tính và các vấn đề học sâu vẫn là các chủ đề nghiên cứu mở trong ngày nay và còn lâu mới được giải quyết. Cuối cùng, một vài trong số các bộ dữ liệu là một phần các cuộc thi và thử thách phân loại ảnh.

Điều quan trọng là phải xem xét các bộ dữ liệu này ngay bây giờ để hiểu rõ về các thách thức mà có thể mong đợi khi làm việc với chúng trong các phần sau.

4.4.1. MNIST

Công cụ MNIST (NIST là viết tắt Viện Tiêu chuẩn và Công nghệ Quốc gia trong khi đó, M là viết tắt cái tên vì dữ liệu đã được xử lý trước để giảm bớt kỳ gánh nặng nào đối với việc xử lý thị giác máy tính và chỉ tập trung vào nhiệm vụ nhận dạng chữ số) một trong những bộ dữ liệu được nghiên cứu kỹ lưỡng nhất trong thị giác máy tính và tài liệu học máy.



Hình 4.15: Một mẫu bộ dữ liệu MNIST. Mục tiêu bộ dữ liệu này là phân loại chính xác chữ số viết tay, 0-9.

Mục tiêu của bộ dữ liệu này là phân loại chính xác các chữ số viết tay 0-9. Trong nhiều trường hợp, bộ dữ liệu này là một điểm chuẩn, một tiêu chuẩn mà các thuật toán học máy được xếp hạng. Trên thực tế, NIST được nghiên cứu rất kỹ đến nỗi Geoffrey Hinton đã mô tả bộ dữ liệu này như là vụ ăn cắp máy học [33] (một loài Drosophila là một loài ruồi giấm), so sánh cách các nhà nghiên cứu sinh học vừa chớm nở sử dụng những con ruồi giấm này vì chúng dễ dàng nuôi cây masse, có một thời gian thế hệ ngắn, và đột biến dễ dàng thu được.

Đồng quan điểm đó, bộ dữ liệu MNIST là bộ dữ liệu đơn giản dành cho những người thực hành học sâu sớm để có được hương vị đầu tiên họ về việc huấn luyện một mạng nơ-ron mà không cần quá nhiều nỗ lực (rất dễ dàng để có được độ chính xác phân loại > 97%) - huấn luyện mô hình mạng nơ-ron trên MNIST rất giống với Hello, World tương đương với học máy.

Bản thân MNIST bao gồm 60.000 ảnh huấn luyện và 10.000 ảnh kiểm tra. Mỗi vector đặc trưng là 784-dim, tương ứng với cường độ điểm ảnh thang độ xám 28×28 ảnh. Các cường độ pixel thang độ xám này là các số nguyên không dấu, nằm trong phạm vi $[0, 255]$. Tất cả các chữ số được đặt trên nền đen với nền trước là màu trắng và màu xám. Với các cường độ pixel thô này, mục tiêu là huấn luyện một mạng nơ-ron để phân loại chính xác các chữ số.

4.4.2. Động vật: chó, mèo và gấu trúc

Mục đích bộ dữ liệu này là phân loại chính xác một ảnh có chứa một con chó, mèo hoặc gấu trúc. Chỉ chứa 3.000 ảnh, bộ dữ liệu động vật có nghĩa là một bộ dữ liệu khác giới thiệu về trò chơi mà có thể nhanh chóng huấn luyện một mô hình học sâu trên CPU hoặc GPU và có được độ chính xác hợp lý.

Trong mục 1.1, sẽ sử dụng bộ dữ liệu này để trình bày cách sử dụng các pixel ảnh như một vector đặc trưng không đổi sang mô hình máy học chất lượng cao ngoài việc sử dụng mạng nơ-ron tích chập (CNN).

Ảnh cho bộ dữ liệu này được thu thập bằng cách lấy mẫu ảnh Chó và Mèo cùng với bộ dữ liệu ImageNet cho các ví dụ về gấu trúc.



Hình 4.16: Một mẫu bộ dữ liệu động vật 3 lớp bao gồm 1.000 ảnh cho mỗi con chó, mèo và gấu trúc tương ứng với tổng số 3.000 ảnh.

4.4.3. CIFAR-10



Hình 4.17: Cũng giống như MNIST, CIFAR-10 được coi là một bộ dữ liệu tiêu chuẩn khác cho phân loại ảnh trong thị giác máy tính và tài liệu học máy. CIFAR-10 bao gồm 60.000 mẫu ảnh.

Ảnh $32 \times 32 \times 3$ (RGB) dẫn đến kích thước vectơ đặc trưng là 3072.

Đúng như tên gọi, CIFAR-10 bao gồm 10 lớp, bao gồm máy bay, ô tô, chim, mèo, hươu, chó, éch, ngựa, tàu và xe tải.

Mặc dù nó rất dễ dàng để huấn luyện một mô hình đạt được độ chính xác phân loại $> 97\%$ trên MNIST, nhưng thực chất khó hơn để có được một mô hình như vậy cho CIFAR-10 (và nó là một người anh lón hơn, CIFAR-100) [94].

Thách thức đến từ sự khác biệt lớn trong cách các vật thể xuất hiện. Ví dụ: không còn có thể giả sử rằng một ảnh chứa pixel màu xanh lá cây tại một vị trí (x, y) nhất định là một con éch. Pixel này có thể là một phần nền một khu rừng chứa một con nai. Hoặc, pixel có thể chỉ đơn giản là màu một chiếc xe tải màu xanh lá cây.

Các giả định này trái ngược hoàn toàn với bộ dữ liệu MNIST nơi mạng có thể tìm hiểu các giả định liên quan đến phân bố không gian cường độ pixel. Ví dụ: phân bố không gian các pixel phía trước số 1 khác biệt đáng kể so với 0 hoặc 5.

Mặc dù là một bộ dữ liệu nhỏ, CIFAR-10 vẫn thường xuyên được sử dụng để đánh giá các kiến trúc CNN mới.

4.4.4. SMILES

Như tên cho thấy, bộ dữ liệu SMILES [95] bao gồm ảnh khuôn mặt đang cười hoặc không cười. Tổng cộng, có 13.165 ảnh thang độ xám trong bộ dữ liệu, với mỗi ảnh có kích thước 64×64 .

Ảnh trong bộ dữ liệu này được cắt xén chặt quanh khuôn mặt cho phép nghĩ ra các thuật toán học máy chỉ tập trung vào nhiệm vụ nhận diện nụ cười. Phân tách tiền xử lý thị giác máy tính từ học máy (đặc biệt là các bộ dữ liệu điểm chuẩn) là xu hướng phổ biến sẽ thấy khi xem xét các bộ dữ liệu điểm chuẩn phổ biến. Trong một số trường hợp, nó không công bằng khi cho rằng một nhà nghiên cứu về máy học có đủ khả năng tiếp xúc với thị giác máy tính để xử lý trước một tập dữ liệu ảnh trước khi áp dụng thuật toán học máy chính họ.

Điều đó nói rằng, xu hướng này đang thay đổi nhanh chóng, và bất kỳ ai quan tâm đến việc áp dụng học máy vào các vấn đề về thị giác máy tính được cho là có ít nhất một nền tảng cơ bản trong thị giác máy tính. Xu hướng này sẽ tiếp tục trong tương lai, vì vậy nếu có kế hoạch học sâu về thị giác máy tính ở bất kỳ chiều sâu nào, chắc chắn nghiên cứu thêm về thị giác máy tính, ngay cả khi nó chỉ là nguyên tắc cơ bản.



Hình 4.18: Phía trên: Ví dụ về khuôn mặt “mỉm cười”. Phía dưới: Mẫu khuôn mặt “không cười”. Sau này chúng tôi sẽ huấn luyện một mạng neural tích chập để nhận ra giữa khuôn mặt tươi cười và không cười trong các luồng video thời gian thực.

4.4.5. KAGGLE: DOGS & CATS

Thử thách Chó và Mèo là một phần cuộc thi Kaggle để đưa ra thuật toán học có thể phân loại chính xác một ảnh là chứa một con chó hoặc một con mèo. Tổng cộng có 25.000 ảnh được cung cấp để huấn luyện thuật toán với các độ phân giải ảnh khác nhau. Một mẫu bộ dữ liệu có thể được nhìn thấy trong Hình 4.19.

Cách quyết định xử lý trước ảnh có thể dẫn đến các mức hiệu suất khác nhau, một lần nữa chứng minh rằng một nền tảng về thị giác máy tính và xử lý ảnh cơ bản sẽ là một chặng đường dài khi nghiên cứu học sâu.



Hình 4.19: Các mẫu từ cuộc thi Chó và Mèo của Kaggle. Mục tiêu thử thách 2 lớp này là xác định chính xác một ảnh đầu vào nhất định có chứa “con chó” hoặc “con mèo”.

4.4.6. FLOWERS-17

Bộ dữ liệu Flower-17 là một bộ dữ liệu danh mục 17 với 80 ảnh mỗi lớp được quản lý bởi Nilsback et al. [96]. Mục tiêu bộ dữ liệu này là dự đoán chính xác các loài hoa cho một ảnh đầu vào nhất định. Một mẫu bộ dữ liệu Flower-17 có thể được nhìn thấy trong Hình 4.20.

Flower-17 có thể được coi là một bộ dữ liệu đầy thách thức do sự thay đổi đáng kể về tỷ lệ, góc nhìn, sự lộn xộn nền, điều kiện ánh sáng khác nhau và sự thay đổi trong lớp. Hơn nữa, chỉ với 80 ảnh cho mỗi lớp, việc các mô hình học sâu tìm hiểu một đại diện cho mỗi lớp sẽ trở nên khó khăn. Theo nguyên tắc chung, khuyên nên có 1.000-5.000 ảnh ví dụ cho mỗi lớp khi huấn luyện một mạng nơ-ron sâu [33].

Sẽ nghiên cứu bộ dữ liệu Flower-17 bên trong phần thực hành và khám phá các phương pháp để cải thiện phân loại bằng các phương pháp học chuyên như khai thác đặc trưng và tinh chỉnh.



Hình 4.20: Một mẫu gồm năm lớp (trong tổng số mươi bảy) trong bộ dữ liệu Flower-17 trong đó mỗi lớp đại diện cho một loài hoa cụ thể.

4.4.7. CALTECH-101

Được giới thiệu bởi Fei-Fei et al. [97] năm 2004, bộ dữ liệu CALTECH-101 là bộ dữ liệu điểm chuẩn phổ biến để phát hiện đối tượng. Thường được sử dụng để phát hiện đối tượng (tức là, dự đoán các tọa độ (x, y) hộp giới hạn cho một đối tượng cụ thể trong ảnh), cũng có thể sử dụng CALTECH-101 để nghiên cứu thuật toán học sâu.

Bộ dữ liệu 8,677 ảnh bao gồm 101 danh mục trải rộng trên nhiều đối tượng khác nhau, bao gồm voi, xe đạp, bóng đá và thậm chí cả bộ não con người, chỉ để nêu tên một số. Bộ dữ liệu CALTECH-101 thể hiện sự mất cân bằng lớp nặng (có nghĩa là có nhiều ảnh ví dụ cho một số danh mục hơn các loại khác), khiến cho việc nghiên cứu từ góc độ mất cân bằng lớp trở nên thú vị.

Các cách tiếp cận trước đây để phân loại ảnh thành CALTECH-101 thu được độ chính xác trong khoảng 35-65% [98, 99, 100]. Tuy nhiên, như thể hiện trong Gói thực hành, dễ dàng tận dụng việc học sâu để phân loại ảnh để đạt được độ chính xác phân loại hơn 99%.

4.4.8. TINY IMAGENET 200

Mạng nơ-ron tích chập cho lớp nhận dạng trực quan [20] đã đưa ra một thách thức phân loại ảnh cho các sinh viên tương tự như thách thức ImageNet, nhưng phạm vi nhỏ hơn. Có tổng cộng 200 lớp ảnh trong bộ dữ liệu này với 500 ảnh để huấn luyện, 50 ảnh để xác nhận và 50 ảnh để kiểm tra cho mỗi lớp. Mỗi ảnh đã được xử lý trước và được cắt thành $64 \times 64 \times 3$ pixel giúp sinh viên dễ dàng tập trung vào các kỹ thuật học sâu hơn là các hàm tiền xử lý thị giác máy tính.

Tuy nhiên, các bước tiền xử lý được áp dụng bởi Karpathy và Johnson thực sự làm cho vấn đề khó khăn hơn một chút vì một số thông tin quan trọng, phân biệt đối xử bị cắt bỏ trong nhiệm vụ tiền xử lý. Điều đó nói rằng, sê trình bày cách huấn luyện kiến trúc VGGNet, GoogLenet và ResNet trên bộ dữ liệu này và khẳng định vị trí hàng đầu trên bảng xếp hạng.

4.4.9. Adience

Bộ dữ liệu Adience, được xây dựng bởi Eidering et al. 2014 [101], được sử dụng để tạo điều kiện thuận lợi cho việc nghiên cứu về tuổi và giới tính. Tổng cộng có 26.580 ảnh được bao gồm trong bộ dữ liệu với độ tuổi từ 0-60. Mục tiêu của bộ dữ liệu này là dự đoán chính xác cả độ tuổi và giới tính đối tượng trong ảnh. Sẽ xem xét thêm về bộ dữ liệu Adience

(và xây dựng hệ thống nhận dạng giới tính và độ tuổi chính) bên trong Gói ImageNet. có thể xem một mẫu bộ dữ liệu Adience trong Hình 4.21.



Hình 4.21: Một mẫu bộ dữ liệu Adience để nhận biết độ tuổi và giới tính.
Độ tuổi dao động từ 0-60 +.

4.4.10. IMAGENET

4.4.10.1. ImageNet là gì?

ImageNet thực sự là một dự án nhằm ghi nhãn và phân loại ảnh thành gần 22.000 danh mục dựa trên một tập hợp các từ và cụm từ xác định.

Tại thời điểm viết bài này, có hơn 14 triệu ảnh trong dự án ImageNet. Để tổ chức một lượng dữ liệu khổng lồ như vậy, ImageNet tuân theo hệ thống phân cấp WordNet [102]. Mỗi từ hoặc cụm từ có ý nghĩa bên trong WordNet được gọi là một cụm từ đồng nghĩa bộ phận hay bộ từ đồng nghĩa với tên tiếng Anh. Trong bộ dữ liệu ImageNet, ảnh được sắp xếp theo các bộ này, với mục tiêu là có hơn 1.000 ảnh cho mỗi bộ.

4.4.10.2. Nhận dạng ảnh quy mô lớn ImageNet (ILSVRC)

Trong bối cảnh thị giác máy tính và học sâu, bất cứ khi nào nghe mọi người nói về ImageNet, rất có thể họ đang đề cập đến thử thách nhận dạng ảnh quy mô lớn ImageNet [80], hoặc đơn giản là ILSVRC.

Mục tiêu theo dõi phân loại ảnh trong thử thách này là huấn luyện một mô hình có thể phân loại ảnh thành 1.000 loại riêng biệt bằng cách sử

dụng khoảng 1,2 triệu ảnh để huấn luyện, 50.000 để xác thực và 100.000 để kiểm tra. 1000 loại ảnh này đại diện cho các lớp đối tượng mà gặp trong cuộc sống hàng ngày, chẳng hạn như các loài chó, mèo, các đối tượng khác nhau trong gia đình, các loại phuơng tiện, và nhiều hơn nữa.

Khi nói đến phân loại ảnh, thách thức ImageNet là tiêu chuẩn thực tế cho các thuật toán phân loại thị giác máy tính - và bảng xếp hạng cho thử thách này đã bị chi phối bởi mạng nơ-ron tích chập và các kỹ thuật học sâu từ năm 2012.

Bên trong bộ ImageNet, sẽ trình bày cách huấn luyện các kiến trúc mạng (AlexNet, SqueezeNet, VGGNet, GoogLeNet, ResNet) cho kết quả cao trong bộ dữ liệu phổ biến này.

4.4.10.3. Kaggle: thử thách nhận dạng cảm xúc của khuôn mặt

Một thử thách khác được đặt ra bởi Kaggle, mục tiêu thử thách nhận dạng biểu hiện khuôn mặt (FER) là xác định chính xác cảm xúc mà một người đang trải qua chỉ đơn giản là từ ảnh khuôn mặt họ. Tổng cộng có 35.888 ảnh được cung cấp trong thử thách FER với mục tiêu gắn nhãn biểu hiện khuôn mặt nhất định thành bảy loại khác nhau:

1. Giận
2. Ghê tởm
3. Sợ hãi
4. Hạnh phúc
5. Buồn
6. Bất ngờ
7. Bình thường



Hình 4.22: Ảnh ghép các ví dụ ImageNet được đặt bởi Đại học Stanford. Bộ dữ liệu này rất lớn với hơn 1,2 triệu ảnh và 1.000 danh mục đối tượng có thẻ. ImageNet được coi là tiêu chuẩn thực tế cho các thuật toán phân loại ảnh chuẩn.

4.4.11. Indoor CVPR (nhận dạng cảnh trong nhà)

Bộ dữ liệu nhận dạng cảnh trong nhà [103], như tên cho thấy, bao gồm một số cảnh trong nhà, bao gồm cửa hàng, nhà ở, không gian giải trí, khu vực làm việc và không gian công cộng. Mục tiêu bộ dữ liệu này là huấn luyện chính xác một mô hình có thể nhận ra từng khu vực.

4.4.12. Stanford Cars

Một bộ dữ liệu khác được tổng hợp bởi Stanford, Cars Dataset [104] bao gồm 16.185 ảnh 196 lớp xe. Có thể cắt lát dữ liệu này theo bất kỳ cách nào muốn dựa trên kiểu dáng xe, kiểu xe hoặc thậm chí năm sản xuất. Mặc dù có khá ít ảnh trên mỗi lớp (với sự mất cân bằng lớp), sẽ trình bày cách sử dụng mạng nơ-ron tích chập để đạt được độ chính xác phân loại > 95% khi dán nhãn cho kiểu dáng và mẫu xe.

4.4.13. Tóm tắt

Trong mục này, đã xem xét các bộ dữ liệu sẽ gặp trong phần còn lại của “Mạng nơ-ron học sâu và ứng dụng”. Một số bộ dữ liệu được coi là bộ dữ liệu kiểm tra, một bộ ảnh nhỏ có thể sử dụng để tìm hiểu về mạng nơ-ron và học sâu. Một số bộ dữ liệu khác rất nổi tiếng (lịch sử phát triển) và được xem là tiêu chuẩn để đánh giá kiến trúc mạng mới. Cuối cùng, các bộ dữ liệu như ImageNet là chủ đề nghiên cứu mở và được sử dụng để nâng cao tính tiên tiến của học sâu.



Hình 4.23: Một mẫu biểu cảm khuôn mặt bên trong Kaggle: Thủ thách nhận dạng biểu hiện khuôn mặt. sẽ huấn luyện một CNN để nhận biết và xác định từng cảm xúc này. CNN này cũng sẽ có thể chạy trong thời gian thực trên CPU, cho phép nhận ra cảm xúc trong các luồng video.



Hình 4.24: Bộ dữ liệu Stanford Cars Dataset bao gồm 16.185 ảnh với 196 loại xe và kiểu mẫu. Sẽ tìm hiểu cách đạt được độ chính xác phân loại > 95% trên tập dữ liệu này bên trong ImageNet Bundle.

4.5. TRÌNH PHÂN LOẠI ẢNH ĐẦU TIÊN

Trong những mục trước đó, đã dành một khoảng thời gian hợp lý để xem xét về các nguyên tắc cơ bản ảnh, các kiểu học và thậm chí là bốn bước mà có thể thực hiện khi xây dựng các trình phân loại ảnh. Nhưng vẫn chưa xây dựng một bộ phân loại ảnh thực tế riêng.

Điều đó sẽ thay đổi trong mục này. Sẽ bắt đầu bằng cách xây dựng một vài tiện ích trợ giúp để tạo điều kiện cho quá trình tiền xử lý và tải ảnh từ ổ cứng. Từ đó, sẽ xem xét về trình phân loại k-Nearest Neighbor (k-NN), lần đầu tiên tiếp xúc với việc sử dụng máy học để phân loại ảnh. Trên thực tế, thuật toán này đơn giản đến nỗi nó không thực hiện bất kỳ chương trình học thực tế nào - tuy nhiên đây vẫn là một thuật toán quan trọng để có thể đánh giá cách các mạng nơ-ron học từ dữ liệu trong các phần sau.

Cuối cùng, sẽ áp dụng thuật toán k-NN để phân loại các loài động vật khác nhau trong ảnh.

4.5.1. Làm việc với bộ dữ liệu ảnh

Khi làm việc với bộ dữ liệu ảnh, trước tiên phải xem xét tổng kích thước tập dữ liệu theo byte. Dữ liệu có đủ lớn để phù hợp với RAM có sẵn trên máy không? Có thể tải tập dữ liệu như thẻ đang tải một ma trận hoặc mảng lớn không?

Điều đó nói rằng, phải luôn nhận thức được kích thước tập dữ liệu trước khi bắt đầu làm việc với các thuật toán phân loại ảnh. Như sẽ thấy trong suốt phần còn lại mục này, dành thời gian để sắp xếp, tiền xử lý và tải tập dữ liệu là một khía cạnh quan trọng việc xây dựng trình phân loại ảnh.

4.5.1.1. Giới thiệu bộ dữ liệu động vật

Bộ dữ liệu động vật trực tuyến là một bộ dữ liệu ví dụ đơn giản mà

đã tổng hợp để trình bày cách huấn luyện các trình phân loại ảnh bằng cách sử dụng các kỹ thuật học máy đơn giản cũng như các thuật toán học sâu nâng cao. Ảnh bên trong bộ dữ liệu động vật thuộc về ba lớp riêng biệt: chó, mèo và gấu trúc, với 1.000 ảnh ví dụ cho mỗi lớp. Ảnh chó và mèo được lấy mẫu từ thử thách Chó và Mèo của Kaggle, trong khi ảnh gấu trúc được lấy mẫu từ bộ dữ liệu ImageNet [80].

Chỉ chứa 3.000 ảnh, bộ dữ liệu động vật có thể dễ dàng phù hợp với bộ nhớ chính của máy, điều này sẽ giúp việc huấn luyện các mô hình của chúng tôi nhanh hơn rất nhiều, mà không yêu cầu viết bất kỳ đoạn chương trình nào để quản lý bộ dữ liệu không thể phù hợp với bộ nhớ. Trên hết, một mô hình học sâu có thể nhanh chóng được huấn luyện về bộ dữ liệu này trên CPU hoặc GPU. Bất kể thiết lập phần cứng của người đọc là gì, người đọc có thể sử dụng bộ dữ liệu này để tìm hiểu những điều cơ bản về học máy và học sâu.

Mục tiêu trong phần này là tận dụng trình phân loại k-NN để có gắng nhận ra từng loài này trong một ảnh chỉ sử dụng cường độ điểm ảnh thô (nghĩa là không có quá trình trích xuất đặc trưng nào đang diễn ra). Như sẽ thấy, cường độ điểm ảnh thô không thích hợp tốt với thuật toán k-NN. Tuy nhiên, đây là một kiểm tra điểm chuẩn quan trọng để chạy vì vậy có thể đánh giá cao lý do tại sao mạng nơ-ron tích chập có thể đạt được độ chính xác cao như vậy về cường độ pixel thô trong khi thuật toán học máy truyền thống không thực hiện được.



Hình 4.25: Một mẫu bộ dữ liệu động vật 3 lớp bao gồm 1.000 ảnh cho mỗi con chó, mèo và gấu trúc tương ứng với tổng số 3.000 ảnh.

4.5.1.2. Bộ công cụ của học sâu

Chúng ta sẽ xây dựng bộ công cụ học sâu tùy chỉnh trong toàn bộ cuốn sách này. Bắt đầu với các hàm và class hỗ trợ cơ bản để xử lý ảnh và tải các bộ dữ liệu nhỏ, cuối cùng xây dựng để triển khai Mạng nơ-ron tích chập hiện đại.

Trên thực tế, đây là bộ công cụ chính xác được sử dụng khi thực hiện các thí nghiệm học sâu riêng. Bộ công cụ này sẽ được xây dựng từng mảnh, cho phép xem các thành phần riêng lẻ tạo nên gói, cuối cùng trở thành một thư viện chính thức có thể được sử dụng để nhanh chóng xây dựng và huấn luyện các mạng học sâu tùy chỉnh riêng.

Đầu tiên phải xác định cấu trúc dự án bộ công cụ:

```
|--- pyimagesearch
```

Như có thể thấy, có một mô-đun duy nhất có tên pyimagesearch. Tất cả các mã mà phát triển sẽ tồn tại bên trong mô-đun pyimagesearch. Đối với mục này, sẽ cần xác định hai mô hình con:

```
|--- pyimagesearch
|   |--- __init__.py
|   |--- datasets
|   |   |--- __init__.py
|   |   |--- simpledatasetloader.py
|   |--- preprocessing
|   |   |--- __init__.py
|   |   |--- simpleprocessor.py
```

Các mô hình con bộ của dữ liệu sẽ bắt đầu thực hiện một lớp có tên SimpleDatasetLoader.

Sẽ sử dụng lớp này để tải các bộ dữ liệu ảnh nhỏ từ ổ cứng (có thể vừa với bộ nhớ chính), tùy ý xử lý trước từng ảnh trong bộ dữ liệu theo một số hàm và sau đó trả về:

1. Ảnh (tức là, cường độ pixel thô)
2. Lớp nhãn được liên kết với mỗi ảnh

Sau đó có mô hình con tiền xử lý. Như sẽ thấy trong các phần sau, có một số phương pháp tiền xử lý mà có thể áp dụng cho tập dữ liệu ảnh để tăng độ chính xác phân loại, bao gồm phép trừ trung bình, lấy mẫu và ngẫu nhiên hoặc đơn giản thay đổi kích thước ảnh thành kích thước cố định. Trong trường hợp này, lớp SimplePreProcessor sẽ thực hiện việc sau - tải một ảnh từ ổ cứng và thay đổi kích thước nó thành một kích thước

cố định, bỏ qua tỷ lệ khung hình. Trong hai phần tiếp theo, sẽ triển khai SimplePreProcessor và SimpleDatasetLoader bằng tay.

4.5.1.3. Bộ xử lý ảnh cơ bản

Các thuật toán học máy như k-NN, SVM và thậm chí mạng nơ-ron tích chập yêu cầu tất cả các ảnh trong bộ dữ liệu phải có kích thước vectơ đặc trưng cố định. Trong trường hợp ảnh, yêu cầu này ngụ ý rằng ảnh phải được xử lý trước và thu nhỏ để có chiều rộng và chiều cao giống hệt nhau.

Có một số cách để thực hiện thay đổi kích thước và tỷ lệ này, từ các phương pháp nâng cao hơn, tôn trọng tỷ lệ khung ảnh gốc đến ảnh được chia tỷ lệ thành các phương pháp đơn giản bỏ qua tỷ lệ khung hình và chỉ đơn giản là ép chiều rộng và chiều cao theo kích thước yêu cầu. Chính xác phương pháp nào nên sử dụng thực sự phụ thuộc vào mức độ phức tạp các yếu tố biến đổi - trong một số trường hợp, bỏ qua tỷ lệ khung hình hoạt động tốt; trong các trường hợp khác, có thể duy trì tỷ lệ khung hình.

Trong mục này, sẽ bắt đầu với giải pháp cơ bản: xây dựng bộ tiền xử lý ảnh thay đổi kích thước ảnh, bỏ qua tỷ lệ khung hình. Mở SimplepreProcessor.py và sau đó chèn đoạn mã sau:

```
1 # import the necessary packages
2 import cv2
3
4 class SimplePreprocessor:
5     def __init__(self, width, height, inter=cv2.INTER_AREA):
6         # store the target image width, height, and interpolation
7         # method used when resizing
8         self.width = width
9         self.height = height
10        self.inter = inter
11
12    def preprocess(self, image):
13        # resize the image to a fixed size, ignoring the aspect
14        # ratio
15        return cv2.resize(image, (self.width, self.height),
16                           interpolation=self.inter)
```

Dòng 2 nhập gói yêu cầu duy nhất, các ràng buộc OpenCV. Sau đó, định nghĩa hàm tạo cho lớp SimpleProcessor trên dòng 5. Hàm tạo yêu cầu hai đối số, theo sau là một đối số tùy chọn thứ ba, mỗi đối số chi tiết bên dưới:

- Width: chiều rộng mục tiêu ảnh đầu vào sau khi thay đổi kích thước.
- Height: chiều cao mục tiêu ảnh đầu vào sau khi thay đổi kích thước.
- inter: một tham số tùy chọn được sử dụng để kiểm soát thuật toán nội

suy nào được sử dụng khi thay đổi kích thước.

Hàm tiền xử lý được xác định trên Dòng 12 yêu cầu một đối số duy nhất - ảnh đầu vào mà chúng ta muốn tiền xử lý.

Dòng 15 và 16 xử lý trước ảnh bằng cách thay đổi kích thước của nó thành kích thước chiều rộng và chiều cao cố định mà sau đó quay lại hàm gọi.

Một lần nữa, bộ tiền xử lý này theo định nghĩa rất cơ bản - tất cả những gì đang làm là chấp nhận một ảnh đầu vào, thay đổi kích thước của nó thành một kích thước cố định và sau đó trả lại cho nó. Tuy nhiên, khi kết hợp với trình tải dữ liệu ảnh trong phần tiếp theo, bộ xử lý trước này sẽ cho phép tải nhanh và xử lý trước một tập dữ liệu từ ổ cứng, cho phép nhanh chóng di chuyển qua phân loại ảnh và chuyển sang các khía cạnh quan trọng hơn.

4.5.1.4. Xây dựng trình tải ảnh

Bây giờ, Bộ xử lý đơn giản đã được xác định, để chuyển sang SimpleDatasetLoader:

```
1 # import the necessary packages
2 import numpy as np
3 import cv2
4 import os
5
6 class SimpleDatasetLoader:
7     def __init__(self, preprocessors=None):
8         # store the image preprocessor
9         self.preprocessors = preprocessors
10
11     # if the preprocessors are None, initialize them as an
12     # empty list
13     if self.preprocessors is None:
14         self.preprocessors = []
```

Các dòng 2-4 nhập các gói Python cần thiết NumPy để xử lý số, cv2 cho các ràng buộc OpenCV và os để có thể trích xuất tên các thư mục con trong đường dẫn ảnh.

Dòng 7 định nghĩa hàm tạo cho SimpleDatasetLoader trong đó có thể tùy ý chuyển vào danh sách các bộ xử lý ảnh (như SimpleProcessor) có thể được áp dụng tuần tự cho một ảnh đầu vào nhất định.

Việc chỉ định các bộ tiền xử lý này là một danh sách thay vì một giá trị duy nhất là điều quan trọng - sẽ có lúc cần thay đổi kích thước ảnh thành một kích thước cố định, sau đó thực hiện một số tỷ lệ (như phép trừ trung bình), sau đó chuyển đổi mảng ảnh đến một định dạng phù hợp với Keras.

Mỗi bộ tiền xử lý này có thể được thực hiện độc lập, cho phép áp dụng chúng một cách tuần tự cho một ảnh một cách hiệu quả.

Sau đó có thể chuyển sang phương thức tải, có thể là SimpleDatasetLoader:

```
16     def load(self, imagePaths, verbose=-1):
17         # initialize the list of features and labels
18         data = []
19         labels = []
20
21         # loop over the input images
22         for (i, imagePath) in enumerate(imagePaths):
23             # load the image and extract the class label assuming
24             # that our path has the following format:
25             # /path/to/dataset/{class}/{image}.jpg
26             image = cv2.imread(imagePath)
27             label = imagePath.split(os.path.sep)[-2]
```

Phương thức tải yêu cầu một tham số duy nhất - imagePaths, đây là danh sách chỉ định đường dẫn tệp đến ảnh trong tập dữ liệu nằm trên ổ cứng, cũng có thể cung cấp một giá trị cho verbose. Mức độ chi tiết này có thể được sử dụng để xuất ra các bản cập nhật lên bàn điều khiển, cho phép theo dõi số lượng ảnh mà SimpleDatasetLoader đã xử lý.

Dòng 18 và 19 khởi tạo danh sách dữ liệu (tức là, chính ảnh) cùng với nhãn, danh sách lớp nhãn cho ảnh.

Dòng 22 bắt đầu lặp qua từng ảnh đầu vào. Đối với mỗi ảnh này, tải nó từ ổ cứng (dòng 26) và trích xuất lớp nhãn dựa trên đường dẫn tệp (dòng 27), đưa ra giả định rằng các bộ dữ liệu được tổ chức trên ổ cứng theo cấu trúc thư mục sau:

/dataset_name/class/image.jpg

Tập dữ liệu có thể có bất cứ tên gọi nào, trong trường hợp này là động vật. Lớp nên là tên lớp nhãn. Ví dụ, lớp gồm các loại chó, mèo hoặc gấu trúc. Cuối cùng, image.jpg là tên ảnh thực tế.

Dựa trên cấu trúc thư mục phân cấp này, có thể giữ các bộ dữ liệu gọn gàng và có tổ chức. Do đó, an toàn khi giả định rằng tất cả các ảnh bên trong thư mục con chó là ví dụ về chó. Tương tự, giả định rằng tất cả các ảnh trong thư mục gấu trúc đều chứa các ví dụ về gấu trúc.

Gần như mọi tập dữ liệu được xem xét bên trong “mạng nơ-ron học sâu và ứng dụng” sẽ tuân theo cấu trúc thiết kế thư mục phân cấp này - rất khuyến khích cũng làm như vậy cho các dự án riêng.

Bây giờ ảnh được tải từ ổ cứng, có thể xử lý trước (nếu cần):

```

29             # check to see if our preprocessors are not None
30             if self.preprocessors is not None:
31
31                 # loop over the preprocessors and apply each to
32                 # the image
33                 for p in self.preprocessors:
34                     image = p.preprocess(image)
35
36             # treat our processed image as a "feature vector"
37             # by updating the data list followed by the labels
38             data.append(image)
39             labels.append(label)

```

Dòng 30 kiểm tra nhanh để đảm bảo rằng các bộ tiền xử lý là không có. Nếu kiểm tra vượt qua, lặp lại từng bộ tiền xử lý trên Dòng 33 và áp dụng tuần tự chúng cho ảnh trên Dòng 34 - hành động này cho phép tạo thành một chuỗi các bộ tiền xử lý có thể được áp dụng cho mọi ảnh trong bộ dữ liệu.

Khi ảnh đã được xử lý trước, sẽ cập nhật danh sách dữ liệu và nhãn tương ứng (Dòng 39 và 39).

Khối mã cuối cùng chỉ đơn giản là xử lý các bản cập nhật in cho bảng điều khiển và sau đó trả lại 2 tuple dữ liệu và nhãn cho hàm gọi:

```

41             # show an update every 'verbose' images
42             if verbose > 0 and i > 0 and (i + 1) % verbose == 0:
43                 print("[INFO] processed {}/{}".
43                      format(i + 1,
44                             len(imagePaths)))
45
46             # return a tuple of the data and labels
47             return (np.array(data), np.array(labels))

```

Nhu có thể thấy, trình tải dữ liệu rất đơn giản theo thiết kế; tuy nhiên, nó cho phép áp dụng bất kỳ số lượng bộ xử lý ảnh nào cho mọi ảnh trong bộ dữ liệu một cách dễ dàng. Nhắc nhở duy nhất của trình tải dữ liệu này là giả định rằng tất cả các ảnh trong bộ dữ liệu có thể vừa với bộ nhớ chính cùng một lúc. Đối với các bộ dữ liệu quá lớn để phù hợp với hệ thống RAM, sẽ cần thiết kế một bộ tải dữ liệu phức tạp hơn.

Bây giờ đã hiểu làm thế nào để (1) xử lý trước một ảnh và (2) tải một bộ ảnh từ ổ cứng, bây giờ có thể chuyển sang giai đoạn phân loại ảnh.

4.5.2. K-NN: trình phân loại đơn giản

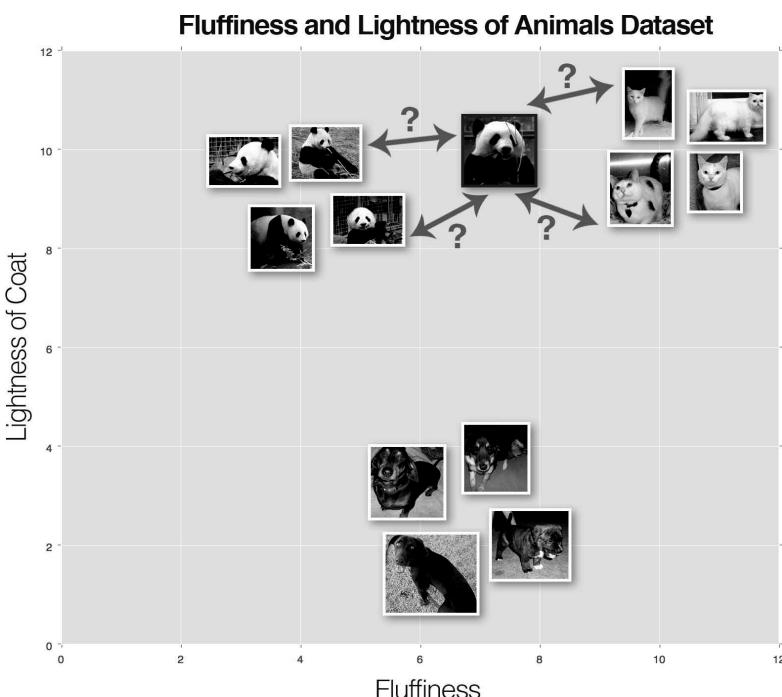
Trình phân loại k-Nearest Neighbor cho đến nay là thuật toán phân loại ảnh và máy học đơn giản nhất. Trên thực tế, nó rất đơn giản đến nỗi nó không thực sự học được bất cứ điều gì. Thay vào đó, thuật toán này trực tiếp dựa vào khoảng cách giữa các vectơ đặc trưng (trong trường hợp cường độ điểm ảnh RGB ảnh thô).

Nói một cách đơn giản, thuật toán k-NN phân loại các điểm dữ liệu chưa biết bằng cách tìm lớp phỏ biến nhất trong số các ví dụ k gần nhất. Mỗi điểm dữ liệu trong k điểm dữ liệu gần nhất sẽ bỏ phiếu và loại nào có số phiếu bầu cao nhất sẽ thắng như Hình 4.26 minh họa.

Để thuật toán k-NN hoạt động, nó đưa ra giả định chính rằng các ảnh có nội dung ảnh tương tự nằm gần nhau trong một không gian n chiều. Ở đây, có thể thấy ba loại ảnh, được ký hiệu là chó, mèo và gấu trúc, tương ứng. Trong ví dụ này, đã vẽ “độ bông” lớp lông động vật dọc theo trục x và “độ sáng” lớp lông dọc theo trục y. Mỗi điểm dữ liệu động vật được nhóm tương đối gần nhau trong không gian n chiều. Điều này ngụ ý rằng khoảng cách giữa hai ảnh con mèo nhỏ hơn nhiều so với khoảng cách giữa con mèo và con chó.

Tuy nhiên, để áp dụng trình phân loại k-NN, trước tiên cần chọn một số liệu khoảng cách hoặc hàm tương tự. Một lựa chọn phỏ biến bao gồm khoảng cách Euclide (thường được gọi là khoảng cách L2):

$$d(p, q) = \sqrt{\sum_{i=1}^N (q_i - p_i)^2} \quad (4.1)$$



Hình 4.26: Cho tập dữ liệu về chó, mèo, gấu trúc, làm thế nào có thể phân loại ảnh được phác thảo bằng màu đỏ?

Tuy nhiên, để áp dụng trình phân loại k-NN, trước tiên chúng ta cần chọn một số liệu khoảng cách hoặc hàm tương tự. Một lựa chọn phổ biến bao gồm khoảng cách Euclide (thường được gọi là khoảng cách L2):

$$d(p, q) = \sum_{i=1}^N |q_i - p_i| \quad (4.2)$$

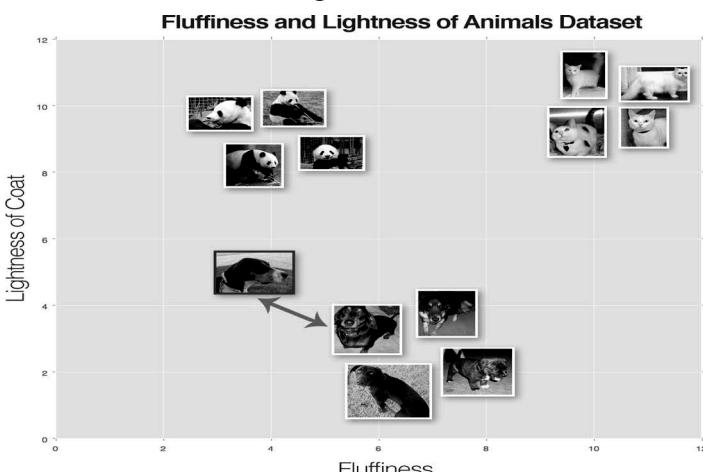
Trong thực tế, có thể sử dụng bất kỳ hàm số liệu / độ tương tự nào phù hợp nhất với dữ liệu (và cung cấp cho kết quả phân loại tốt nhất). Tuy nhiên, trong phần còn lại bài học này, sẽ sử dụng thước đo khoảng cách phổ biến nhất: khoảng cách Euclide.

4.5.2.1. Một ví dụ của k-NN

Tại thời điểm này, hiểu các nguyên tắc thuật toán k-NN. Biết rằng nó dựa vào khoảng cách giữa các vectơ/ảnh đặc trưng để phân loại. Và biết rằng nó đòi hỏi một hàm khoảng cách/tương tự để tính các khoảng cách này.

Nhưng làm thế nào để thực sự phân loại? Để trả lời câu hỏi này, nhìn vào Hình 4.27. Ở đây có một bộ dữ liệu ba loại động vật - chó, mèo và gấu trúc - và đã vẽ chúng theo độ mịn và độ bông bộ lông chúng.

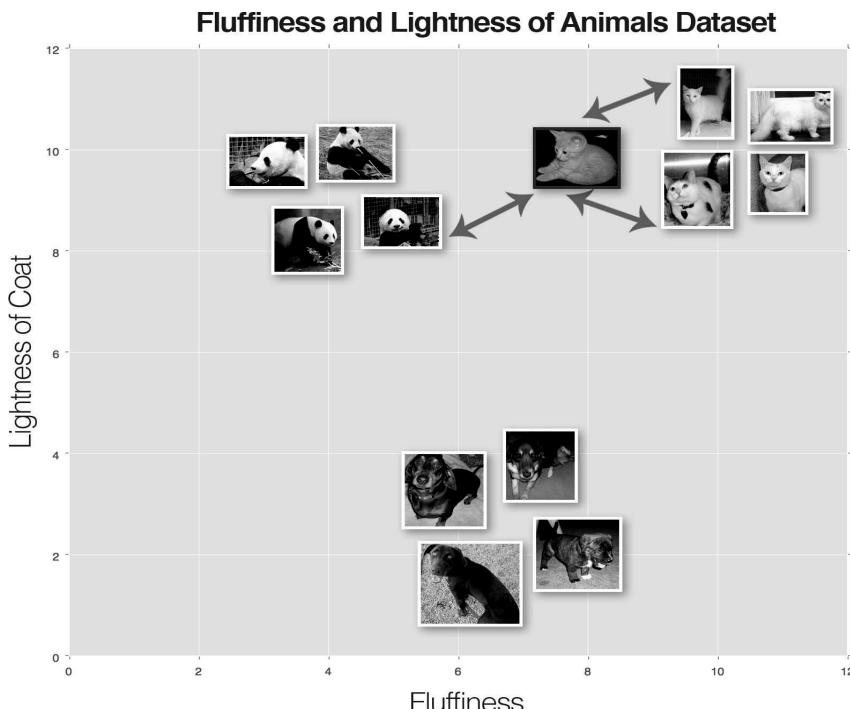
Ngoài ra, đã chèn một “động vật không xác định” mà đang cố gắng phân loại chỉ bằng một đặc trưng lân cận duy nhất (tức là, $k = 1$). Trong trường hợp này, động vật gần nhất với ảnh đầu vào là một điểm dữ liệu chó; do đó ảnh đầu vào nên được phân loại là con chó.



Hình 4.27: Trong ví dụ này, đã chèn một ảnh không xác định (được tô màu đỏ) vào tập dữ liệu và sau đó sử dụng khoảng cách giữa động vật chưa biết và tập dữ liệu động vật để phân loại.

Để thử một con vật khác không xác định khác, lần này sử dụng $k = 3$ (Hình 4.28). Đã tìm thấy hai con mèo và một con gấu trúc trong ba kết quả hàng đầu. Vì danh mục mèo có số phiếu bầu lớn nhất, sẽ phân loại ảnh đầu vào là mèo.

Có thể tiếp tục thực hiện quy trình này cho các giá trị khác nhau k , nhưng cho dù k lớn hay nhỏ, nguyên tắc vẫn giữ nguyên - loại có số phiếu bầu lớn nhất trong k điểm huấn luyện gần nhất sẽ thắng và được sử dụng làm nhãn cho điểm dữ liệu đầu vào.



Hình 4.28: Phân loại một con vật khác, lần này sử dụng $k = 3$ thay vì $k = 1$. Vì có hai ảnh con mèo gần với ảnh đầu vào hơn ảnh gấu trúc đơn lẻ, dán nhãn ảnh đầu vào này là một con mèo.

4.5.2.2. Siêu tham số của k-NN

Có hai siêu tham số rõ ràng cần quan tâm khi chạy thuật toán k-NN. Tham số đầu tiên là giá trị k . Giá trị tối ưu k là gì? Nếu nó quá nhỏ (chẳng hạn như $k = 1$), thì đạt được hiệu quả nhưng dễ bị nhiễu và các điểm dữ liệu ngoại lệ. Tuy nhiên, nếu k quá lớn, thì có nguy cơ làm trộn trù quá khớp kết quả phân loại và tăng độ lệch (b).

Tham số thứ hai nên xem xét là số liệu khoảng cách thực tế. Khoảng cách Euclide là sự lựa chọn tốt nhất? Khoảng cách Manhattan thì sao?

Trong phần tiếp theo, sẽ huấn luyện bộ phân loại k-NN trên bộ dữ liệu Động vật và đánh giá mô hình trên bộ kiểm tra. Khuyến khích thực nghiệm xung quanh các giá trị k khác nhau cùng với các khoảng cách số liệu khác nhau, lưu ý cách mà hiệu suất thay đổi.

4.5.2.3. Thực hiện k-NN

Mục tiêu của mục này là huấn luyện một bộ phân loại k-NN về cường độ pixel thô trên bộ dữ liệu Động vật và sử dụng nó để phân loại các ảnh động vật chưa biết. Sử dụng bốn bước để huấn luyện các trình phân loại từ mục 3.3.2:

Bước 1 - Tập hợp bộ dữ liệu Động vật bao gồm 3.000 ảnh với 1.000 ảnh cho mỗi con chó, mèo và gấu trúc, tương ứng. Mỗi ảnh được thể hiện trong không gian màu RGB, sẽ xử lý trước mỗi ảnh bằng cách thay đổi kích thước thành 32×32 pixel. Có tám đến ba kênh RGB, kích thước ảnh đã thay đổi ngụ ý rằng mỗi ảnh trong bộ dữ liệu được biểu thị bằng $32 \times 32 \times 3 = 3,072$ số nguyên.

Bước 2 - Tách bộ dữ liệu: Đối với ví dụ đơn giản này, sẽ sử dụng hai phần tách dữ liệu. Một phần để huấn luyện, và phần còn lại để kiểm tra, sẽ bỏ qua bộ xác thực để điều chỉnh siêu tham số và để lại điều này như một bài tập cho người đọc.

Bước 3 - Huấn luyện Trình phân loại: Trình phân loại k-NN sẽ được huấn luyện về cường độ điểm ảnh thô trong tập huấn luyện.

Bước 4 - Đánh giá: Sau khi phân loại k-NN được huấn luyện, có thể đánh giá hiệu suất trên bộ kiểm tra.

Đầu tiên, mở một tệp mới, đặt tên là knn.py và chèn đoạn chương trình sau:

```
1 # import the necessary packages
2 from sklearn.neighbors import KNeighborsClassifier
3 from sklearn.preprocessing import LabelEncoder
4 from sklearn.model_selection import train_test_split
5 from sklearn.metrics import classification_report
6 from pyimagesearch.preprocessing import SimplePreprocessor
7 from pyimagesearch.datasets import SimpleDatasetLoader
8 from imutils import paths
9 import argparse
```

Dòng 2-9 nhập các gói Python cần thiết. Điều cần lưu ý là dòng 2: KNeighborsClassifier thực hiện thuật toán k-NN, được hỗ trợ bởi thư viện scikit-learn.

Dòng 3: LabelEncoder chương trình, một tiện ích trợ giúp để chuyển đổi các nhãn được biểu diễn dưới dạng chuỗi thành số nguyên trong đó có một số

nguyên duy nhất cho mỗi lớp nhãn (một cách phổ biến khi áp dụng học máy).

Dòng 4: Sẽ nhập hàm train_test_split, đây là một hàm tiện ích được sử dụng để giúp tạo các phân tách huấn luyện và kiểm tra.

Dòng 5: Hàm class_Vport là một hàm tiện ích khác được sử dụng để giúp đánh giá hiệu suất trình phân loại và in bảng kết quả đã được định dạng lên bảng điều khiển.

Cũng có thể thấy các triển khai về SimplePreProcessor và SimpleDatasetLoader được nhập tương ứng trên dòng 6 và dòng 7.

Tiếp theo, để phân tích cú pháp đối số dòng lệnh:

```
11 # construct the argument parse and parse the arguments
12 ap = argparse.ArgumentParser()
13 ap.add_argument("-d", "--dataset", required=True,
14     help="path to input dataset")
15 ap.add_argument("-k", "--neighbors", type=int, default=1,
16     help="# of nearest neighbors for classification")
17 ap.add_argument("-j", "--jobs", type=int, default=-1,
18     help="# of jobs for k-NN distance (-1 uses all available cores)")
19 args = vars(ap.parse_args())
```

Yêu cầu một đối số dòng lệnh, theo sau là hai đối số tùy chọn, từng được xem xét bên dưới:

- --dataset: đường dẫn đến nơi tập dữ liệu ảnh đầu vào nằm trên ổ cứng.

- - Lân cận: tùy chọn, số lượng lân cận k sẽ áp dụng khi sử dụng thuật toán k-NN.

- --jobs: tùy chọn, số lượng công việc đồng thời sẽ chạy khi tính toán khoảng cách giữa điểm dữ liệu đầu vào và tập huấn luyện. Giá trị -1 sẽ sử dụng tất cả các lõi có sẵn trên bộ xử lý.

Bây giờ, đối số dòng lệnh được phân tích cú pháp, có thể lấy đường dẫn tệp ảnh trong tập dữ liệu, sau đó tải và xử lý trước chúng (Bước 1 trong phân loại):

```
21 # grab the list of images that we'll be describing
22 print("[INFO] loading images...")
23 imagePaths = list(paths.list_images(args["dataset"]))
24
25 # initialize the image preprocessor, load the dataset from disk,
26 # and reshape the data matrix
27 sp = SimplePreprocessor(32, 32)
28 sdl = SimpleDatasetLoader(preprocessors=[sp])
29 (data, labels) = sdl.load(imagePaths, verbose=500)
30 data = data.reshape((data.shape[0], 3072))
31
32 # show some information on memory consumption of the images
33 print("[INFO] features matrix: {:.1f}MB".format(
34     data nbytes / (1024 * 1000.0)))
```

Dòng 23 lấy đường dẫn tệp đến tất cả các ảnh trong tập dữ liệu. Sau đó, khởi tạo Bộ xử lý SimplePre được sử dụng để thay đổi kích thước mỗi ảnh thành 32×32 pixel trên Dòng 27.

SimpleDatasetLoader được khởi tạo trên dòng 28, cung cấp SimplePreProcessor khởi tạo làm đối số (ngụ ý rằng sp sẽ được áp dụng cho mọi ảnh trong tập dữ liệu).

Một hàm gọi đến .load trên Dòng 29 tải tập dữ liệu ảnh thực tế từ ổ cứng. Phương pháp này trả về 2 tuple dữ liệu (mỗi ảnh được thay đổi kích thước thành 32×32 pixel) cùng với các nhãn cho mỗi ảnh. Sau khi tải ảnh từ ổ cứng, mảng NumPy dữ liệu có .shape là $(3000, 32, 32, 3)$, cho biết có 3.000 ảnh trong bộ dữ liệu, mỗi pixel 32×32 pixel với 3 kênh.

Tuy nhiên, để áp dụng thuật toán k-NN, cần phải làm phẳng các ảnh từ hình đại diện 3D sang một danh sách cường độ điểm ảnh duy nhất. Để thực hiện điều này, Dòng 30 gọi phương thức .reshape trên mảng NumPy dữ liệu, làm phẳng các ảnh $32 \times 32 \times 3$ thành một mảng có hình dạng $(3000, 3072)$. Dữ liệu ảnh thực tế đã thay đổi hoàn toàn - các ảnh được biểu diễn đơn giản dưới dạng danh sách 3.000 mục nhập, mỗi mục 3.072-dim ($32 \times 32 \times 3 = 3,072$).

Để chứng minh cần bao nhiêu bộ nhớ để lưu trữ 3.000 ảnh này trong bộ nhớ, Dòng 33 và 34 tính toán số byte mà mảng tiêu thụ và sau đó chuyển đổi số thành megabyte.

Tiếp theo, cùng nhau xây dựng các phần tách kiểm tra và huấn luyện (Bước 2):

```
36 # encode the labels as integers
37 le = LabelEncoder()
38 labels = le.fit_transform(labels)
39
40 # partition the data into training and testing splits using 75% of
41 # the data for training and the remaining 25% for testing
42 (trainX, testX, trainY, testY) = train_test_split(data, labels,
43     test_size=0.25, random_state=42)
```

Dòng 37 và 38 chuyển đổi nhãn (được biểu thị dưới dạng chuỗi) thành số nguyên nơi có một số nguyên duy nhất cho mỗi lớp. Chuyển đổi này cho phép ánh xạ lớp mèo thành số nguyên 0, lớp chó thành số nguyên 1 và lớp gấu trúc thành số nguyên 2. Nhiều thuật toán học máy cho rằng các lớp nhãn được mã hóa dưới dạng số nguyên, vì vậy điều quan trọng là phải có thói quen thực hiện bước này.

Việc tính toán phân tách huấn luyện và kiểm tra được xử lý bởi hàm train_test_split trên Dòng 42 và 43. Ở đây phân vùng dữ liệu và nhãn thành

hai bộ duy nhất: 75% dữ liệu cho huấn luyện và 25% cho kiểm tra.

Người ta thường sử dụng biến X để chỉ một tập dữ liệu có chứa các điểm dữ liệu sẽ sử dụng để huấn luyện và kiểm tra trong khi y để cập đến các lớp nhãn (sẽ tìm hiểu thêm về điều này trong Chương 3 về học được tham số hóa). Do đó, sử dụng các biến trainX và testX để tham khảo các ví dụ huấn luyện và kiểm tra tương ứng. Các biến trainY và testY là nhãn huấn luyện và kiểm tra, ký hiệu này phổ biến trong suốt cuốn sách này và trong các cuốn sách, khóa học và hướng dẫn học máy khác.

Cuối cùng, có thể tạo trình phân loại k-NN và đánh giá nó (Bước 3 và 4 trong phân loại ảnh):

```
45 # train and evaluate a k-NN classifier on the raw pixel intensities
46 print("[INFO] evaluating k-NN classifier...")
47 model = KNeighborsClassifier(n_neighbors=args["neighbors"],
48     n_jobs=args["jobs"])
49 model.fit(trainX, trainY)
50 print(classification_report(testY, model.predict(testX),
51     target_names=le.classes_))
```

Dòng 47 và 48 khởi tạo lớp KNeighorClassifier. Một cuộc gọi đến phương thức .fit trên Dòng 49, huấn luyện bộ phân loại, mặc dù không có chương trình học thực tế nào diễn ra ở đây - mô hình k-NN chỉ đơn giản là lưu trữ dữ liệu trainX và trainY để có thể dự đoán kiểm tra được thiết lập bằng cách tính khoảng cách giữa dữ liệu đầu vào và dữ liệu trainX.

Các dòng 50 và 51 đánh giá trình phân loại bằng cách sử dụng hàm phân loại. Ở đây, cần cung cấp lớp nhãn testY, lớp nhãn dự đoán từ mô hình và tùy chọn tên lớp nhãn (ví dụ: con chó, con mèo, con gấu trúc).

4.5.2.4. Kết quả của k-NN

Để chạy trình phân loại k-NN, thực hiện lệnh sau

```
$ python knn.py --dataset ../datasets/animals
```

Sau đó, sẽ thấy đầu ra sau tương tự như sau:

```
[INFO] loading images...
[INFO] processed 500/3000
[INFO] processed 1000/3000
[INFO] processed 1500/3000
[INFO] processed 2000/3000
[INFO] processed 2500/3000
[INFO] processed 3000/3000
[INFO] features matrix: 9.0MB|
[INFO] evaluating k-NN classifier...
```

	precision	recall	f1-score	support
cats	0.39	0.49	0.43	239
dogs	0.36	0.47	0.41	249
panda	0.79	0.36	0.50	262
avg / total	0.52	0.44	0.45	750

Lưu ý cách ma trận đặc trưng chỉ tiêu thụ 9 MB bộ nhớ cho 3.000 ảnh, mỗi kích thước $32 \times 32 \times 3$ - bộ dữ liệu này có thể dễ dàng được lưu trữ trong bộ nhớ trên các máy hiện đại mà không gặp vấn đề gì. Đánh giá trình phân loại, thấy rằng đã thu được độ chính xác 52% - độ chính xác này không phải là xấu đối với một trình phân loại không thực hiện bất kỳ trò chơi học thực sự nào, cho rằng xác suất đoán ngẫu nhiên câu trả lời đúng là 1/3.

Tuy nhiên, thật thú vị khi kiểm tra tính chính xác từng lớp nhau. Lớp gấu trúc đã được phân loại chính xác 79% thời gian, có khả năng là do thực tế là gấu trúc chủ yếu là đen và trắng và do đó những ảnh này nằm gần nhau hơn trong không gian 3,072 mờ.

Chó và mèo có được độ chính xác phân loại thấp hơn đáng kể ở mức tương ứng 39% và 36%. Những kết quả này có thể được quy cho thực tế là chó và mèo có thể có màu lông rất giống nhau và màu lông chúng không thể được sử dụng để phân biệt giữa chúng. Nhiều nền (như cỏ ở sân sau, màu chiếc ghế mà một con vật đang nghỉ ngơi, v.v.) cũng có thể gây nhầm lẫn cho thuật toán k-NN vì nó không thể học được bất kỳ mô hình nào phân biệt các loài này. Sự nhầm lẫn này là một trong những nhược điểm chính của thuật toán k-NN: trong khi nó đơn giản, nó cũng không thể học được từ dữ liệu.

Mục 3.1 đã xem xét về khái niệm học theo tham số, trong đó thực sự có thể học các mẫu từ chính các ảnh thay vì giả định các ảnh có nội dung tương tự sẽ nhóm lại với nhau trong một không gian n chiều.

4.5.2.5. Ưu và nhược điểm của k-NN

Một ưu điểm chính thuật toán k-NN là nó rất đơn giản để thực hiện và hiểu. Hơn nữa, bộ phân loại hoàn toàn không mất thời gian để huấn luyện, vì tất cả những gì cần làm là lưu trữ các điểm dữ liệu cho mục đích tính toán khoảng cách sau này giữa chúng và có được phân loại cuối cùng.

Tuy nhiên, phải đánh đổi cho sự đơn giản này tại thời điểm phân loại. Việc phân loại một điểm kiểm tra mới yêu cầu so sánh với mọi điểm dữ liệu trong dữ liệu huấn luyện, theo thang điểm O (N), khiến cho việc làm việc với các bộ dữ liệu lớn hơn bị cấm tính toán.

Có thể giải quyết chi phí thời gian này bằng cách sử dụng thuật toán xấp xỉ lân cận gần nhất (ANN) (như kd-tree [105], FLANN [60], dự đoán ngẫu nhiên [106, 107, 108], v.v.); tuy nhiên, bằng cách sử dụng các thuật toán này, yêu cầu trao đổi độ phức tạp không gian/thời gian để “làm chính xác” thuật toán láng giềng gần nhất vì thực hiện xấp xỉ. Điều đó nói rằng,

trong nhiều trường hợp, rất đáng để nỗ lực và mất một chút chính xác để sử dụng thuật toán k-NN. Hành vi này trái ngược với hầu hết các thuật toán học máy (và tất cả các mạng nơ-ron), nơi dành một lượng lớn thời gian để huấn luyện mô hình để có được độ chính xác cao, và đến lượt nó, có các phân loại rất nhanh trong thời gian kiểm tra.

Cuối cùng, thuật toán k-NN phù hợp hơn với các không gian đặc trưng chiều thấp (mà ảnh không có). Khoảng cách trong không gian đặc trưng chiều cao thường không trực quan, có thể đọc thêm trong bài viết của Pedro Domingo [109].

Nó cũng rất quan trọng để lưu ý rằng thuật toán k-NN thực sự không học hỏi bất cứ điều gì - thuật toán không thể làm cho nó thông minh hơn nếu nó mắc lỗi; Nó chỉ đơn giản dựa vào khoảng cách trong không gian n chiều để phân loại.

Với những nhược điểm này, tại sao phải nghiên cứu thuật toán k-NN? Lý do là thuật toán rất đơn giản. Nó rất dễ hiểu. Và quan trọng nhất, nó là cơ sở để so sánh các mạng nơ-ron và mạng nơ-ron tích.

4.5.3. Tóm tắt

Trong mục này, đã học cách xây dựng bộ xử lý ảnh đơn giản và tải bộ dữ liệu ảnh vào bộ nhớ. Sau đó, đã xem xét về phân loại k-Nearest Neighbor hoặc k-NN.

Thuật toán k-NN phân loại các điểm dữ liệu chưa biết bằng cách so sánh điểm dữ liệu chưa biết với từng điểm dữ liệu trong tập huấn luyện. Việc so sánh được thực hiện bằng cách sử dụng hàm khoảng cách hoặc số liệu tương tự. Sau đó, từ các ví dụ tương tự k nhất trong tập huấn luyện, tích lũy số lượng phiếu bầu người dùng cho mỗi nhãn. Hạng mục có số phiếu bầu cao nhất, sẽ được chọn làm bảng phân loại tổng thể.

Mặc dù đơn giản và trực quan, thuật toán k-NN có một số nhược điểm. Đầu tiên là không học được bất cứ điều gì - nếu thuật toán mắc lỗi, không có cách nào để cải tiến chính xác và cải thiện chính cho các phân loại sau này. Thứ hai, khi không có cấu trúc dữ liệu riêng biệt, quy mô tuyến tính của thuật toán k-NN với số lượng điểm dữ liệu, khiến nó không chỉ thực sự khó sử dụng khi kích thước dữ liệu lớn mà còn đặt ra dấu hỏi về cách sử dụng [109].

Hiện tại đã tìm hiểu cơ sở về phân loại ảnh bằng thuật toán k-NN, có thể nghiên cứu về các tham số, nền tảng mà tất cả các mạng nơ-ron và học sâu được xây dựng trên đó. Sử dụng phương pháp học theo tham số, thực sự có thể học từ dữ liệu đầu vào và tìm hiểu các mẫu cơ bản. Quá trình này

sẽ cho phép xây dựng các bộ phân loại ảnh có độ chính xác cao, tăng hiệu suất k-NN.

4.6. NHẬN DẠNG MẶT CUỜI TRÊN KHUÔN MẶT

4.6.1. Bộ dữ liệu SMILES

Bộ dữ liệu SMILES bao gồm các ảnh khuôn mặt đang cười hoặc không cười [95]. Tổng cộng, có 13165 ảnh xám trong bộ dữ liệu, với mỗi ảnh có kích thước là 64×64 pixel.

Như Hình 4.29, ảnh trong bộ dữ liệu này được cắt xén quanh khuôn mặt, điều này sẽ giúp quá trình huấn luyện trở nên dễ dàng hơn vì sẽ có thể học được các mô hình mặt cười trực tiếp từ các ảnh đầu vào, giống như đã thực hiện tương tự trong các mục trước trong cuốn sách này.



Hình 4.29: Hàng trên: Ví dụ về khuôn mặt “mỉm cười”. Hàng dưới: khuôn mặt “không cười”. Trong mục này, sẽ huấn luyện một mạng neural tích chập để nhận ra giữa khuôn mặt cười và không cười trong các video thời gian thực.

Tuy nhiên, cắt xén gần đặt ra một vấn đề trong quá trình kiểm tra- vì ảnh đầu vào sẽ không chỉ chứa khuôn mặt mà cả nền ảnh, trước tiên cần phải định vị khuôn mặt trong ảnh và trích xuất ROI khuôn mặt trước khi có thể đưa vào mạng để phát hiện. May mắn thay, khi sử dụng các phương pháp thị giác máy tính truyền thống như Haar cascades, thì đây là một nhiệm vụ dễ dàng

Vấn đề thứ hai cần xử lý trong bộ dữ liệu SMILES là sự mất cân bằng lớp. Mặc dù có 13165 ảnh trong bộ dữ liệu, nhưng 9475 trong số các ví dụ này không cười, trong khi chỉ có 3690 thuộc về lớp mỉm cười. Cho rằng có hơn 2,5 lần số lượng ảnh “không mỉm cười” với các ví dụ về mặt mỉm cười, cần cẩn thận khi thiết kế quy trình huấn luyện.

Như đã thấy sau trong mục này, có thể chống lại sự mất cân bằng lớp bằng cách tính toán một trọng số cho mỗi lớp trong thời gian huấn luyện.

4.6.2. Huấn luyện mặt cười bằng CNN

Bước đầu tiên trong việc xây dựng ứng dụng phát hiện mặt cười là huấn luyện một mạng CNN trên bộ dữ liệu SMILES để phân biệt giữa một khuôn mặt đang cười và không cười. Để hoàn thành ứng dụng này, hãy tạo ra một tệp mới có tên train_model.py. Từ đó, chèn đoạn chương trình sau:

```
1 # import the necessary packages
2 from sklearn.preprocessing import LabelEncoder
3 from sklearn.model_selection import train_test_split
4 from sklearn.metrics import classification_report
5 from keras.preprocessing.image import img_to_array
6 from keras.utils import np_utils
7 from pyimagesearch.nn.conv import LeNet
8 from imutils import paths
9 import matplotlib.pyplot as plt
10 import numpy as np
11 import argparse
12 import imutils
13 import cv2
14 import os
```

Dòng 2-14 nhập các gói Python cần thiết đã sử dụng tất cả các gói thư viện trước đây, dòng 7 nhập mạng LeNet (mục 3.4) - đây là kiến trúc sẽ sử dụng khi tạo ứng dụng phát hiện mặt cười.

Tiếp theo, hãy phân tích cú pháp đối số dòng lệnh:

```
16 # construct the argument parse and parse the arguments
17 ap = argparse.ArgumentParser()
18 ap.add_argument("-d", "--dataset", required=True,
19     help="path to input dataset of faces")
20 ap.add_argument("-m", "--model", required=True,
21     help="path to output model")
22 args = vars(ap.parse_args())
23
24 # initialize the list of data and labels
25 data = []
26 labels = []
```

Đoạn chương trình sẽ yêu cầu hai đối số dòng lệnh, mỗi đối số được nêu chi tiết bên dưới:

1. --dataset: đường dẫn đến thư mục SMILES nằm trên ổ cứng.

2. --model: đường dẫn đến trọng số LeNet được tuân tự hóa sẽ được lưu sau khi huấn luyện. Bây giờ đã sẵn sàng để tải tập dữ liệu SMILES từ ổ cứng và lưu trữ trong bộ nhớ:

```

28 # loop over the input images
29 for imagePath in sorted(list(paths.list_images(args["dataset"]))):
30     # load the image, pre-process it, and store it in the data list
31     image = cv2.imread(imagePath)
32     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
33     image = imutils.resize(image, width=28)
34     image = img_to_array(image)
35     data.append(image)
36
37     # extract the class label from the image path and update the
38     # labels list
39     label = imagePath.split(os.path.sep)[-3]
40     label = "smiling" if label == "positives" else "not_smiling"
41     labels.append(label)

```

Trên Dòng 29, lặp lại tất cả các ảnh trong thư mục đầu vào --dataset. Đối với mỗi ảnh này:

1. Tải dữ liệu từ ổ cứng (Dòng 31).
2. Chuyển đổi ảnh sang mức xám (Dòng 32).
3. Thay đổi kích thước để có kích thước đầu vào cố định là 28x28 pixel (Dòng 33).
4. Chuyển đổi ảnh thành một mảng tương thích với Keras và thứ tự kênh (Dòng 34).
5. Thêm ảnh vào danh sách dữ liệu mà LeNet sẽ được huấn luyện.

Các dòng 39-41 xử lý trích xuất nhãn lớp từ imagePath và cập nhật danh sách nhãn. Bộ dữ liệu SMILES lưu trữ các khuôn mặt cười nằm trong thư mục con SMILES/positives/positives7, trong khi không phải là khuôn mặt cười nằm trong thư mục con SMILES/negatives/negatives7.

Do đó, đưa ra đường dẫn đến một ảnh:

[SMILES/positives/positives7/10007.jpg](#)

Bây giờ dữ liệu và nhãn đã được tạo, có thể chia tỷ lệ cường độ pixel thành phạm vi [0, 1] và sau đó áp dụng chương trình hóa cho nhãn:

```

43 # scale the raw pixel intensities to the range [0, 1]
44 data = np.array(data, dtype="float") / 255.0
45 labels = np.array(labels)
46
47 # convert the labels from integers to vectors
48 le = LabelEncoder().fit(labels)
49 labels = np_utils.to_categorical(le.transform(labels), 2)

```

Khỏi chương trình tiếp theo xử lý vấn đề mất cân bằng dữ liệu bằng cách tính toán các trọng số lớp:

```
51 # account for skew in the labeled data
52 classTotals = labels.sum(axis=0)
53 classWeight = classTotals.max() / classTotals
```

Dòng 52 tính tổng số ví dụ cho mỗi lớp. Trong trường hợp này, classTotals sẽ là một mảng: [9485, 3690] cho lần lượt không phải là mặt cười và mặt cười.

Sau đó, chia tỷ lệ các tổng này trên Dòng 53 để thu được lớp trọng số được sử dụng để xử lý sự mất cân bằng lớp, có được mảng: [1, 2.56]. Trọng số này có nghĩa rằng mạng sẽ coi mọi trường hợp mặt cười là 2.56 trường hợp không phải mặt cười và giúp chống lại sự mất cân bằng lớp bằng cách khuếch đại mức giảm theo trường hợp bằng một trọng số lớn hơn khi nhìn thấy các ví dụ mặt cười

Bây giờ đã tính toán trọng số lớp, có thể chuyển sang phân vùng dữ liệu thành 2 phần huấn luyện và kiểm tra, sử dụng 80% dữ liệu cho huấn luyện và 20% cho kiểm tra:

```
55 # partition the data into training and testing splits using 80% of
56 # the data for training and the remaining 20% for testing
57 (trainX, testX, trainY, testY) = train_test_split(data,
58         labels, test_size=0.2, stratify=labels, random_state=42)
```

Cuối cùng, đã sẵn sàng để huấn luyện mạng LeNet:

```
60 # initialize the model
61 print("[INFO] compiling model...")
62 model = LeNet.build(width=28, height=28, depth=1, classes=2)
63 model.compile(loss="binary_crossentropy", optimizer="adam",
64     metrics=["accuracy"])
65
66 # train the network
67 print("[INFO] training network...")
68 H = model.fit(trainX, trainY, validation_data=(testX, testY),
69     class_weight=classWeight, batch_size=64, epochs=15, verbose=1)
```

Dòng 62 khởi tạo kiến trúc LeNet sẽ chấp nhận ảnh đơn 28×28 . Cho rằng chỉ có hai lớp (mim cười so với không cười), đặt các lớp = 2. Cũng sẽ sử dụng binary_crossentropy thay vì categorical_crossentropy làm hàm tổn thất. Một lần nữa, entropy chéo phân loại chỉ được sử dụng khi số lượng lớp nhiều hơn hai.

Cho đến thời điểm này, đã sử dụng trình tối ưu hóa SGD để huấn luyện mạng. Ở đây, sẽ sử dụng Adam (Dòng 63) [110], bao gồm các trình tối ưu hóa nâng cao hơn (bao gồm Adam, RMSprop, Adadelta); tuy nhiên, vì hiệu quả của ví dụ này, chỉ cần hiểu rằng Adam có thể hội tụ nhanh hơn SGD trong một số trường hợp.

Một lần nữa, trình tối ưu hóa và các tham số liên quan thường được coi là siêu tham số mà cần điều chỉnh khi huấn luyện mạng. Khi đặt ví dụ, thấy rằng thuật toán Adam thực hiện tốt hơn đáng kể so với SGD.

Các dòng 68 và 69 huấn luyện LeNet cho tổng cộng 15 chu kỳ sử dụng lớp trọng số được cung cấp để chống lại sự mất cân bằng lớp. Khi mạng được huấn luyện, có thể đánh giá và nối tiếp các trọng số vào ô cứng:

```
71 # evaluate the network
72 print("[INFO] evaluating network...")
73 predictions = model.predict(testX, batch_size=64)
74 print(classification_report(testY.argmax(axis=1),
75     predictions.argmax(axis=1), target_names=le.classes_))
76
77 # save the model to disk
78 print("[INFO] serializing network...")
79 model.save(args["model"])
```

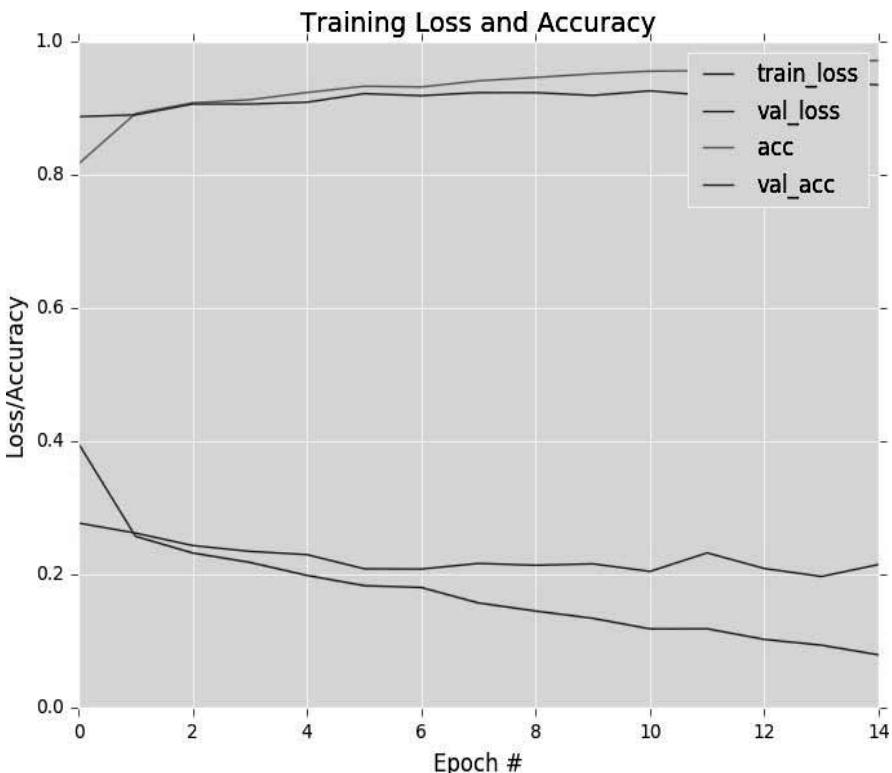
Xây dựng một đường cong huấn luyện cho mạng để có thể hình dung hiệu suất:

```
81 # plot the training + testing loss and accuracy
82 plt.style.use("ggplot")
83 plt.figure()
84 plt.plot(np.arange(0, 15), H.history["loss"], label="train_loss")
85 plt.plot(np.arange(0, 15), H.history["val_loss"], label="val_loss")
86 plt.plot(np.arange(0, 15), H.history["acc"], label="acc")
87 plt.plot(np.arange(0, 15), H.history["val_acc"], label="val_acc")
88 plt.title("Training Loss and Accuracy")
89 plt.xlabel("Epoch #")
90 plt.ylabel("Loss/Accuracy")
91 plt.legend()
92 plt.show()
```

Để huấn luyện mô hình phát hiện mặt cười, hãy thực hiện lệnh sau:

```
$python train_model.py --dataset ./datasets/SMILEsmileD \
--model output/lenet.hdf5
[INFO] compiling model...
[INFO] training network...
Train on 10532 samples, validate on 2633 samples
Epoch 1/15
8s - loss: 0.3970 - acc: 0.8161 - val_loss: 0.2771 - val_acc: 0.8872
Epoch 2/15
8s - loss: 0.2572 - acc: 0.8919 - val_loss: 0.2620 - val_acc: 0.8899
Epoch 3/15
7s - loss: 0.2322 - acc: 0.9079 - val_loss: 0.2433 - val_acc: 0.9062
...
Epoch 15/15
8s - loss: 0.0791 - acc: 0.9716 - val_loss: 0.2148 - val_acc: 0.9351
[INFO] evaluating network...
      precision    recall   f1-score   support
not_smiling       0.95      0.97      0.96     1890
    smiling        0.91      0.86      0.88      743
avg / total       0.93      0.94      0.93     2633
```

Sau 15 chu kỳ, có thể thấy rằng mạng đang đạt được độ chính xác phân loại 93%.



Hình 4.30: Sơ đồ đường cong huấn luyện cho kiến trúc LeNet được huấn luyện trên bộ dữ liệu SMILES. Sau 15 chu kỳ, đang đạt được độ chính xác phân loại $\approx 93\%$ trên bộ thử nghiệm .

Sau 6 chu kỳ tồn thất xác thực bắt đầu đình trệ - huấn luyện thêm qua chu kỳ 15 sẽ dẫn đến quá khứp. Nếu muốn, sẽ cải thiện độ chính xác mô hình nhận dạng mặt cười bằng cách sử dụng thêm dữ liệu huấn luyện, bằng cách:

1. Thu thập dữ liệu huấn luyện bổ sung.
2. Áp dụng các kỹ thuật tăng số dữ liệu như dịch ngẫu nhiên, xoay và dịch chuyển tập huấn luyện hiện có.

4.6.3. Thực hiện nhận dạng mặt cười theo thời gian thực bằng CNN

Bây giờ đã huấn luyện mô hình, bước tiếp theo là xây dựng tập lệnh Python để truy cập tệp webcam/video và áp dụng tính năng phát hiện mặt cười cho mỗi khung hình. Để thực hiện bước này, hãy mở một tệp mới, đặt tên là detect_smile.py:

```
1 # import the necessary packages
2 from keras.preprocessing.image import img_to_array
3 from keras.models import load_model
4 import numpy as np
5 import argparse
6 import imutils
7 import cv2
```

Dòng 2-7 nhập các gói Python cần thiết. Hàm `img_to_array` sẽ được sử dụng để chuyển đổi từng khung hình riêng lẻ từ luồng video sang một mảng được sắp xếp kênh chính xác. Hàm `load_model` sẽ được sử dụng để tải trọng số mô hình LeNet được huấn luyện từ ổ cứng.

Tập lệnh `detectsmile.py` yêu cầu hai đối số dòng lệnh theo sau là một tùy chọn thứ ba:

```
9 # construct the argument parse and parse the arguments
10 ap = argparse.ArgumentParser()
11 ap.add_argument("-c", "--cascade", required=True,
12     help="path to where the face cascade resides")
13 ap.add_argument("-m", "--model", required=True,
14     help="path to pre-trained smile detector CNN")
15 ap.add_argument("-v", "--video",
16     help="path to the (optional) video file")
17 args = vars(ap.parse_args())
```

Đối số đầu tiên, `--cascade` là đường dẫn đến thuật toán Haar cascades được sử dụng để phát hiện khuôn mặt trong ảnh. Paul Viola và Michael Jones (2001) mô tả chi tiết về Haar cascades trong nghiên cứu của họ, Phát hiện đối tượng nhanh bằng cách sử dụng một loạt các tính năng đơn giản Boosted [111]. Nghiên cứu này đã trở thành một trong những bài báo được trích dẫn nhiều nhất trong tài liệu về thị giác máy tính.

Thuật toán Haar cascades có khả năng phát hiện các đối tượng trong ảnh, bất kể vị trí và tỷ lệ nào. Có lẽ hấp dẫn nhất (và có liên quan đến ứng dụng), mô hình có thể chạy trong thời gian thực trên phần cứng hiện đại. Trên thực tế, động lực đằng sau công việc của Viola và Jones là tạo ra một mô hình tìm khuôn mặt.

Do đánh giá chi tiết về phát hiện đối tượng bằng các phương pháp thị giác máy tính truyền thống nằm ngoài phạm vi cuốn sách này, nên xem lại thuật toán Haar cascades, cùng với phương pháp HOG + SVM tuyển tính để phát hiện đối tượng.

Đối số dòng chung thứ hai, `--model`, chỉ định đường dẫn đến trọng số LeNet được nối tiếp trên ổ cứng. Tập lệnh sẽ mặc định để đọc các khung

hình ảnh từ webcam tích hợp hoặc camera giao tiếp bằng cổng USB; tuy nhiên, thay vào đó, nếu muốn đọc các khung hình ảnh từ một tệp, có thể chỉ định tệp qua lựa chọn --video.

Trước khi có thể phát hiện mặt cười, trước tiên cần thực hiện một số khởi tạo:

```
19 # load the face detector cascade and smile detector CNN
20 detector = cv2.CascadeClassifier(args["cascade"])
21 model = load_model(args["model"])
22
23 # if a video path was not supplied, grab the reference to the webcam
24 if not args.get("video", False):
25     camera = cv2.VideoCapture(0)
26
27 # otherwise, load the video
28 else:
29     camera = cv2.VideoCapture(args["video"])
```

Các dòng 20 và 21 tải bộ phát hiện khuôn mặt bằng thuật toán Haar cascades và mô hình LeNet được huấn luyện trước đó. Nếu đường dẫn video không được cung cấp, sẽ thay đổi thông số để webcam được hoạt động (Dòng 24 và 25). Mặt khác, tệp video trên ổ cứng sẽ được mở ở dòng 28 và 29.

Xét đoạn chương trình xử lý chính của ứng dụng:

```
31 # keep looping
32 while True:
33     # grab the current frame
34     (grabbed, frame) = camera.read()
35
36     # if we are viewing a video and we did not grab a frame, then we
37     # have reached the end of the video
38     if args.get("video") and not grabbed:
39         break
40
41     # resize the frame, convert it to grayscale, and then clone the
42     # original frame so we can draw on it later in the program
43     frame = imutils.resize(frame, width=300)
44     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
45     frameClone = frame.copy()
```

Dòng 32 bắt đầu một vòng lặp sẽ tiếp tục cho đến khi (1) dừng tập lệnh hoặc (2) đến cuối tệp video (với điều kiện - đường dẫn video được áp dụng).

Dòng 34 lấy khung hình tiếp theo từ video. Nếu không thể lấy được khung hình ảnh, thì đã đạt đến cuối tệp video. Mặt khác, xử lý trước khung để phát hiện khuôn mặt bằng cách thay đổi kích thước

khung hình để có chiều rộng 300 pixel (Dòng 43) và chuyển đổi thành ảnh xám (Dòng 44).

Phương thức `.detectMultiScale` xử lý việc phát hiện vùng giới hạn (x, y) các mặt trong khung:

```
47     # detect faces in the input frame, then clone the frame so that
48     # we can draw on it
49     rects = detector.detectMultiScale(gray, scaleFactor=1.1,
50             minNeighbors=5, minSize=(30, 30),
51             flags=cv2.CASCADE_SCALE_IMAGE)
```

Ở đây chuyển qua ảnh xám và chỉ ra rằng đối với một khu vực nhất định được coi là một khuôn mặt, phải có chiều rộng tối thiểu 30×30 pixel.

Phương thức `.detectMultiScale` trả về danh sách 4 tuple tạo thành hình chữ nhật giới hạn khuôn mặt trong khung. Hai giá trị đầu tiên trong danh sách này là các tọa độ bắt đầu (x, y). Hai giá trị thứ hai trong danh sách `rects` lần lượt là chiều rộng và chiều cao khung giới hạn lặp qua từng bộ hộp giới hạn dưới đây. Đối với mỗi khung giới hạn, sử dụng phép cắt mảng NumPy để trích xuất ROI mặt (Dòng 58). Khi có vùng ROI, xử lý trước và chuẩn bị phân loại thông qua LeNet bằng cách thay đổi kích thước, chia tỷ lệ, chuyển đổi thành một mảng tương thích với Keras và đệm ảnh với một chiều bổ sung (Dòng 59-62).

```
53     # loop over the face bounding boxes
54     for (fX, fY, fW, fH) in rects:
55         # extract the ROI of the face from the grayscale image,
56         # resize it to a fixed 28x28 pixels, and then prepare the
57         # ROI for classification via the CNN
58         roi = gray[fY:fY + fH, fX:fX + fW]
59         roi = cv2.resize(roi, (28, 28))
60         roi = roi.astype("float") / 255.0
61         roi = img_to_array(roi)
62         roi = np.expand_dims(roi, axis=0)
```

Một khi vùng ROI được xử lý trước, có thể được chuyển qua mạng LeNet để phân loại:

```
64     # determine the probabilities of both "smiling" and "not
65     # smiling", then set the label accordingly
66     (notSmiling, smiling) = model.predict(roi)[0]
67     label = "Smiling" if smiling > notSmiling else "Not Smiling"
```

Một hàm gọi đến `.predict` trên Dòng 66 trả về xác suất không cười và cười. Dòng 67 gán nhãn tùy thuộc vào tỷ lệ nào lớn hơn.

Một khi có nhãn, có thể vẽ, cùng với khung giới hạn tương ứng

```
69     # display the label and bounding box rectangle on the output
70     # frame
71     cv2.putText(frameClone, label, (fX, fY - 10),
72                 cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 0, 255), 2)
73     cv2.rectangle(frameClone, (fX, fY), (fX + fW, fY + fH),
74                   (0, 0, 255), 2)
```

Xử lý đoạn chương trình cuối cùng hiển thị khung hình ảnh:

```
76     # show our detected faces along with smiling/not smiling labels
77     cv2.imshow("Face", frameClone)

78

79     # if the 'q' key is pressed, stop the loop
80     if cv2.waitKey(1) & 0xFF == ord("q"):
81         break

82

83     # cleanup the camera and close any open windows
84     camera.release()
85     cv2.destroyAllWindows()
```

Nếu phím q được nhấn, thoát khỏi tập lệnh.

Để chạy detect_smile.py bằng webcam, hãy thực hiện lệnh sau:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \
--model output/lenet.hdf5
```

Thay vào đó, nếu muốn sử dụng tệp video, sẽ cập nhật lệnh để sử dụng chế độ --video:

```
$ python detect_smile.py --cascade haarcascade_frontalface_default.xml \
--model output/lenet.hdf5 --video path/to/your/video.mov
```

Hãy chú ý cách LeNet dự đoán chính xác về mặt cười và hay không mim cười dựa trên biểu hiện của khuôn mặt.

4.6.4. Tóm tắt

Trong mục này, đã học cách xây dựng một ứng dụng thị giác máy tính và ứng dụng học sâu để thực hiện phát hiện mặt cười. Để thực hiện điều này, trước tiên hãy huấn luyện kiến trúc LeNet trên bộ dữ liệu SMILES.

Do sự mất cân bằng lớp trong bộ dữ liệu SMILES, đã phát hiện ra cách tính trọng số của mỗi lớp được sử dụng để giúp giảm thiểu vấn đề.

Sau khi được huấn luyện, đánh giá LeNet trên bộ dữ liệu kiểm tra và nhận thấy mô hình thu được độ chính xác phân loại là 93%. Độ chính xác phân loại cao hơn có thể đạt được bằng cách thu thập thêm

dữ liệu huấn luyện hoặc áp dụng tăng dữ liệu cho bộ dữ liệu huấn luyện hiện có.

Sau đó, đã tạo một tập lệnh Python để đọc các khung hình từ webcam hoặc video, phát hiện khuôn mặt và sau đó áp dụng mạng được huấn luyện trước. Để phát hiện khuôn mặt, đã sử dụng thuật toán Haar cascades trong thư viện OpenCV. Khi một khuôn mặt được phát hiện, trích xuất từ khung hình và sau đó chuyển qua LeNet để xác định xem người đó có cười hay không cười. Nhìn chung, hệ thống phát hiện mặt cười có thể dễ dàng chạy trong thời gian thực trên CPU bằng phần cứng hiện đại.

Chương 5: ỨNG DỤNG MẠNG NO-RON HỒI QUY VÀO LĨNH VỰC XỬ LÝ NGÔN NGỮ

Chương này sẽ thảo luận về các khái niệm của mạng nơ-ron hồi quy (RNNs) và biến thể của nó là mạng bộ nhớ dài ngắn (LSTM). LSTM chủ yếu được sử dụng để dự đoán chuỗi trình tự. Hãy tìm hiểu về các loại dự đoán chuỗi trình tự và sau đó tìm hiểu cách thực hiện dự báo chuỗi thời gian với sự hỗ trợ của mô hình LSTM.

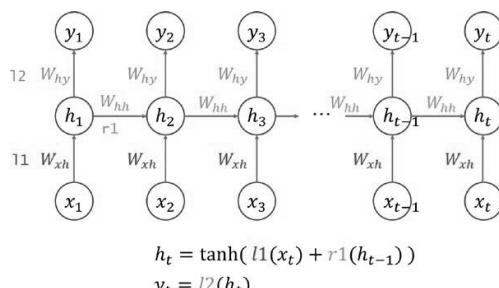
Ngoài ra sẽ ứng dụng mạng RNN và LSTM vào lĩnh vực xử lý ngôn ngữ dưới dạng lời nói hoặc văn bản.

5.1. MẠNG NO-RON HỒI QUY VÀ MẠNG LSTM

5.1.1. Khái niệm về mạng nơ-ron hồi quy

Mạng nơ-ron hồi quy là một loại mạng thần kinh nhân tạo phù hợp nhất để nhận dạng các mẫu trong chuỗi dữ liệu, chẳng hạn như văn bản, video, lời nói, ngôn ngữ, bộ gen và dữ liệu chuỗi thời gian. RNN là một thuật toán cực kỳ mạnh mẽ có thể phân loại, phân cụm và đưa ra dự đoán về dữ liệu, đặc biệt là chuỗi thời gian và văn bản.

RNN có thể được xem như một mạng MLP (multilayered perceptron) với việc bổ sung các vòng lặp vào kiến trúc. Trong Hình 5.1, có thể thấy rằng có một lớp đầu vào (với các nút như x_1, x_2, \dots, x_t), một lớp ẩn (với các nút như h_1, h_2, \dots, h_t) và một lớp đầu ra (với các nút như y_1, y_2, \dots, y_t). Điều này tương tự với kiến trúc MLP. Sự khác biệt là các nút của các lớp ẩn được liên kết với nhau. Trong vanilla (cơ bản) RNN / LSTM, các nút được kết nối theo một hướng. Điều này có nghĩa là h_2 phụ thuộc vào h_1 (và x_2) và h_3 phụ thuộc vào h_2 (và x_3). Nút trong lớp ẩn được quyết định bởi nút trước đó trong lớp ẩn.



Hình 5.1: Mạng nơ-ron hồi quy

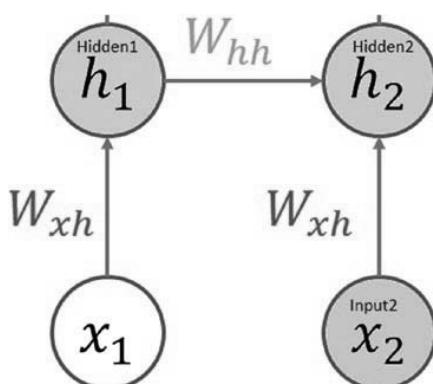
Kiểu kiến trúc này đảm bảo rằng đầu ra tại $t = n$ phụ thuộc vào các đầu vào tại $t = n$, $t = n-1$ và $t = 1$. Nói cách khác, đầu ra phụ thuộc vào chuỗi dữ liệu thay vì một đoạn dữ liệu duy nhất (Hình 5.2).

$(Input1) \rightarrow Output1$
$(Input2, Input1) \rightarrow Output2$
$(Input3, Input2, Input1) \rightarrow Output3$
$(Input4, Input3, Input2, Input1) \rightarrow Output4$

Hình 5.2: Lưu đồ

Hình 5.3 cho thấy cách các nút của lớp ẩn được kết nối với các nút của lớp đầu vào.

$(Input1) \rightarrow Hidden1$
$(Input2, Hidden1) \rightarrow Hidden2$
$(Input3, Hidden2) \rightarrow Hidden3$
$(Input4, Hidden3) \rightarrow Hidden4$



Hình 5.3: Các kết nối

Trong một RNN, nếu các chuỗi khá dài, độ dốc-gradient (rất cần thiết để điều chỉnh trọng số và độ lệch) được tính toán trong quá trình huấn luyện (backpropagation). Chúng có thể biến mất (nhân với nhiều giá trị nhỏ hơn 1) hoặc loại ra (nhân với nhiều giá trị lớn hơn 1), làm cho mô hình huấn luyện rất chậm.

5.1.2. Khái niệm mạng LSTM

Bộ nhớ dài ngắn là một kiến trúc biến thể của mạng RNN, giải quyết vấn đề biến mất và loại bỏ các gradient và giải quyết vấn đề huấn luyện qua

các chuỗi dài và giữ lại bộ nhớ. Tất cả các mạng RNN có các vòng phản hồi trong lớp lặp lại. Các vòng phản hồi giúp lưu giữ thông tin trong bộ nhớ theo thời gian. Nhưng, có thể khó huấn luyện mạng RNN tiêu chuẩn để giải quyết các vấn đề đòi hỏi phải học trong thời gian dài hạn. Độ dốc của hàm mất phân rã theo cấp số nhân theo thời gian (một hiện tượng được gọi là vấn đề độ dốc biến mất), rất khó để huấn luyện các RNN điển hình. Đó là lý do tại sao một RNN được sửa đổi theo cách bao gồm một ô nhớ có thể duy trì thông tin trong bộ nhớ trong thời gian dài. RNN sửa đổi tốt hơn được gọi là LSTM. Trong LSTM, một bộ công được sử dụng để kiểm soát khi thông tin đi vào bộ nhớ, giải quyết vấn đề độ dốc biến mất hoặc loại bỏ.

Các kết nối hiện tại thêm trạng thái hoặc bộ nhớ vào mạng, cho phép tìm hiểu và khai thác bản chất quan sát theo thứ tự trong các chuỗi đầu vào. Bộ nhớ trong có nghĩa là đầu ra của mạng có điều kiện trong bối cảnh gần đây trong chuỗi đầu vào, không phải là những gì vừa được trình bày dưới dạng đầu vào của mạng.

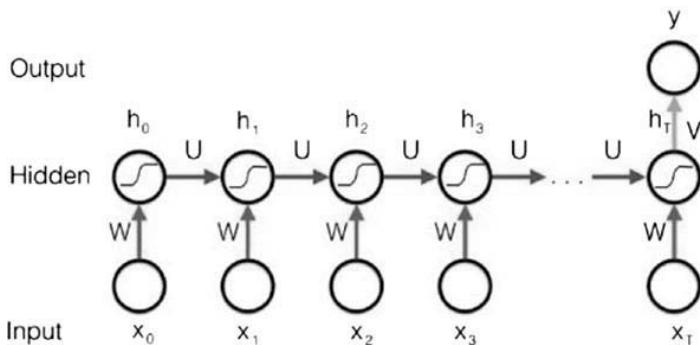
5.1.3. Mô hình của mạng LSTM

LSTM có thể có một trong các chế độ sau:

- Mô hình một đầu vào-một đầu ra
- Mô hình một đầu vào-nhiều đầu ra
- Mô hình nhiều đầu vào-một đầu ra
- Mô hình nhiều đầu vào-nhiều đầu ra

Ngoài các chế độ này, các mô hình nhiều-nhiều được đóng bộ hóa cũng đang được sử dụng, đặc biệt là để phân loại video.

Hình 5.4 cho thấy một LSTM nhiều-một. Điều này ngụ ý rằng nhiều đầu vào tạo ra một đầu ra trong mô hình này.



Hình 5.4: LSTM dạng nhiều đầu vào-một đầu ra

5.1.4. Dự đoán trình tự

LSTM phù hợp nhất cho dữ liệu chuỗi. LSTM có thể dự đoán, phân loại và tạo dữ liệu chuỗi. Một chuỗi có nghĩa là một thứ tự quan sát, thay vì một tập hợp các quan sát. Một ví dụ về chuỗi là một chuỗi thử nghiệm trong đó các dấu thời gian và giá trị theo thứ tự (theo thời gian) của chuỗi. Một ví dụ khác là một video, có thể được coi là một chuỗi các hình ảnh hoặc một chuỗi các đoạn âm thanh.

Dự đoán dựa trên chuỗi dữ liệu được gọi là dự đoán trình tự. Dự đoán trình tự được chia thành bốn loại.

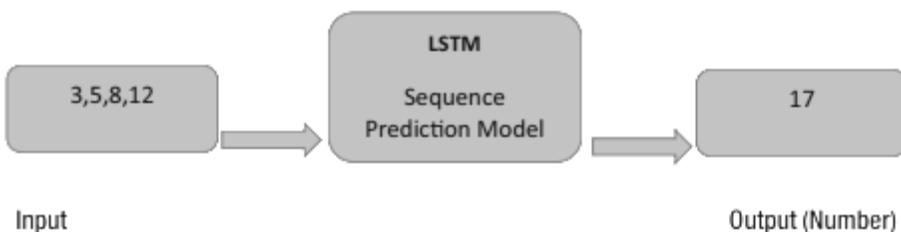
- Dự đoán số thứ tự
- Phân loại trình tự
- Tạo chuỗi
- Dự đoán theo trình tự

5.1.4.1. Dự đoán số thứ tự

Dự đoán số thứ tự là dự đoán giá trị tiếp theo cho một chuỗi nhất định. Các trường hợp sử dụng là dự báo thị trường chứng khoán và dự báo thời tiết. Dưới đây là một ví dụ:

Trình tự đầu vào: 3,5,8,12

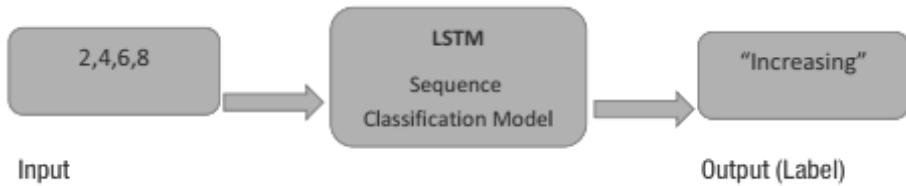
Đầu ra: 17



5.1.4.2. Phân loại trình tự

Phân loại trình tự dự đoán nhãn lớp cho một chuỗi nhất định. Nó có thể sử dụng để phát hiện gian lận (sử dụng chuỗi giao dịch làm đầu vào để phân loại/dự đoán liệu tài khoản có bị hack hay không) và phân loại sinh viên dựa trên kết quả kiểm tra (trình tự các điểm thi trong sáu tháng qua theo thời gian). Dưới đây là một ví dụ:

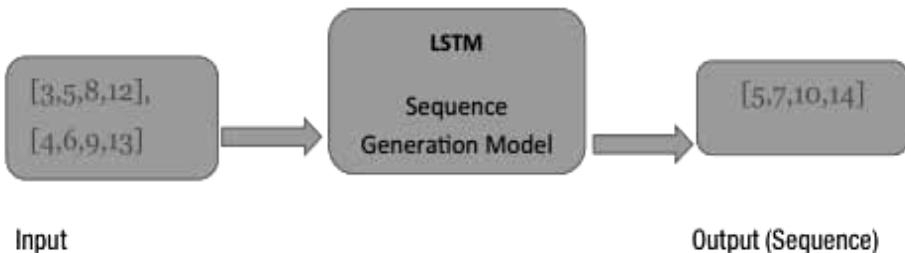
- Trình tự đầu vào: 2,4,6,8
- Đầu ra: Tăng



5.1.4.3. Tạo chuỗi

Tạo chuỗi là khi tạo một chuỗi đầu ra mới có các thuộc tính giống như các chuỗi đầu vào trong tập văn bản đầu vào. Các trường hợp sử dụng là tạo văn bản (đưa ra 100 dòng blog, tạo dòng tiếp theo của blog) và tạo nhạc (lấy ví dụ âm nhạc, tạo ra bản nhạc mới). Dưới đây là một ví dụ:

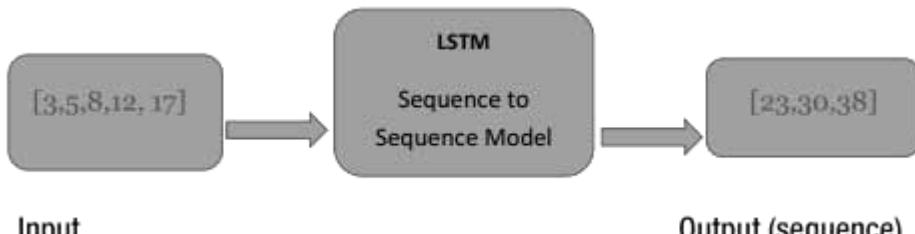
- Trình tự nhập: [3,5,8,12], [4,6,9,13]
- Đầu ra: [5,7,10,14]



5.1.4.4. Dự đoán trình tự (sequence-to-sequence prediction)

Dự đoán trình tự là khi dự đoán trình tự tiếp theo cho một chuỗi nhất định. Các trường hợp sử dụng được ghi lại bằng tài liệu và dự báo chuỗi thời gian nhiều giai đoạn (dự đoán một chuỗi số). Dưới đây là một ví dụ:

- Trình tự nhập: [3, 5, 8, 12, 17]
- Đầu ra: [23,30,38]



Như đã đề cập, LSTM được sử dụng để dự báo chuỗi thời gian trong các lĩnh vực.

Hãy để đi qua một mô hình LSTM. Giả sử rằng tệp CSV được cung cấp trong đó cột đầu tiên là thời gian và cột thứ hai là một giá trị. Nó có thể đại diện cho dữ liệu cảm biến (IoT).

Với dữ liệu chuỗi thời gian, phải dự đoán các giá trị cho tương lai.

5.1.5. Dự đoán chuỗi thời gian với mô hình LSTM

Dưới đây là ví dụ đầy đủ về dự báo chuỗi thời gian với LSTM:

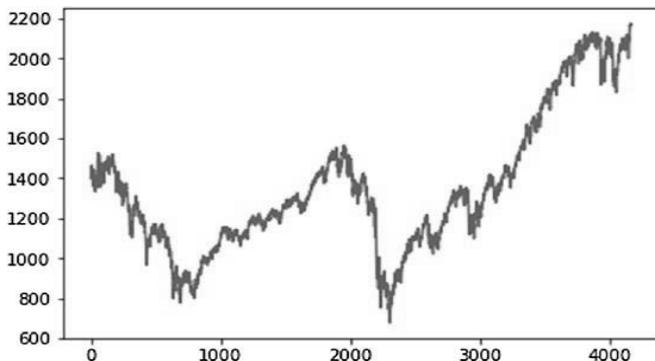
```
# Simple LSTM for a time series data
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
import pylab

# convert an array of values into a timeseries data
def create_timeseries(series, ts_lag=1):
    dataX = []
    dataY = []
    n_rows = len(series)-ts_lag
    for i in range(n_rows-1):
        a = series[i:(i+ts_lag), 0]
        dataX.append(a)
        dataY.append(series[i + ts_lag, 0])

    X, Y = np.array(dataX), np.array(dataY)
    return X, Y

# fix random seed for reproducibility
np.random.seed(230)
# load dataset
dataframe = read_csv('sp500.csv', usecols=[0])
plt.plot(dataframe)
plt.show()
```

Hình 5.5 cho thấy một biểu đồ của dữ liệu



Hình 5.5: Sơ đồ dữ liệu

Đoạn chương trình sau:

```
# Changing datatype to float32 type
series = datafram.values.astype('float32')

# Normalize the dataset
scaler = StandardScaler()
series = scaler.fit_transform(series)

# split the datasets into train and test sets
train_size = int(len(series) * 0.75)
test_size = len(series) - train_size
train, test = series[0:train_size,:], series[train_size:len(series),:]

# reshape the train and test dataset into X=t and Y=t+1
ts_lag = 1
trainX, trainY = create_timeseries(train, ts_lag)
testX, testY = create_timeseries(test, ts_lag)

# reshape input data to be [samples, time steps, features]
trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.
shape[1]))
testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))

# Define the LSTM model
model = Sequential()
model.add(LSTM(10, input_shape=(1, ts_lag)))
model.add(Dense(1))
model.compile(loss='mean_squared_logarithmic_error',
optimizer='adagrad')

# fit the model
model.fit(trainX, trainY, epochs=500, batch_size=30)
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
```

```

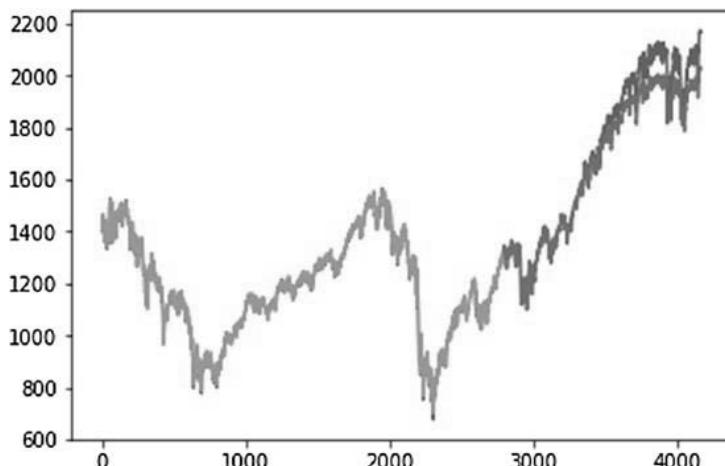
# rescale predicted values
trainPredict = scaler.inverse_transform(trainPredict)
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])

# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0],
trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0],
testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))

# plot baseline and predictions
pylab.plot(trainPredictPlot)
pylab.plot(testPredictPlot)
pylab.show()

```

Trong Hình 5.6, người đọc có thể thấy biểu đồ của chuỗi thời gian thực tế so với dự đoán. Phần màu cam là dữ liệu huấn luyện, phần màu xanh lam là dữ liệu thử nghiệm và phần màu xanh lá cây là đầu ra dự đoán.



Hình 5.6: Đồ thị của chuỗi thời gian thực tế so với dự đoán

Cho đến nay, đã học được các khái niệm về RNN, LSTM và dự báo chuỗi thời gian với mô hình LSTM.

LSTM đã được sử dụng trong phân loại văn bản. Sử dụng LSTM để xây dựng các trình phân loại văn bản. Đầu tiên, một văn bản được chuyển

đổi thành số bằng cách sử dụng word như word2vec hoặc glove. Sau đó, phân loại trình tự được thực hiện thông qua LSTM.

Cách tiếp cận này cung cấp độ chính xác cao hơn nhiều so với một tập từ điển hình hoặc tf-idf thông thường sau là các phân loại ML (machine learning) như SVM (support vector machine), Random Forest. Trong chương tiếp theo, có thể thấy LSTM có thể được sử dụng cho các phân loại như thế nào.

5.2. CHUYỂN ĐỔI LỜI NÓI THÀNH VĂN BẢN VÀ NGUỒN LẠI

Trong mục này, sẽ tìm hiểu về tầm quan trọng của việc chuyển đổi lời nói thành văn bản và chuyển văn bản thành giọng nói, ngoài ra còn tìm hiểu về các chức năng và thành phần cần thiết để thực hiện loại chuyển đổi này.

Cụ thể, bao gồm những vấn đề sau đây:

- Tại sao muốn chuyển đổi lời nói thành văn bản
- Lời nói dưới dạng dữ liệu
- Tính năng lời nói ánh xạ lời nói thành ma trận
- Các biểu đồ phổ, ánh xạ lời nói thành hình ảnh
- Xây dựng bộ phân loại để nhận dạng giọng nói thông qua các tính năng Mel-frequency cepstral coefficient (MFCC)
- Xây dựng bộ phân loại để nhận dạng giọng nói thông qua quang phổ
- Cách tiếp cận nguồn mở để nhận dạng giọng nói
- Nguồn cung cấp dịch vụ nhận thức phổ biến
- Tương lai của chuyển đổi lời nói thành văn bản

5.2.1. Chuyển đổi lời nói thành văn bản

Chuyển đổi lời nói thành văn bản, theo thuật ngữ layman, có nghĩa là một ứng dụng nhận ra các từ được nói bởi một người và chuyển đổi giọng nói thành văn bản. Có rất nhiều lý do để sử dụng chuyển đổi lời nói thành văn bản.

- Người mù hoặc người bị tật nguyền có thể điều khiển các thiết bị khác nhau chỉ bằng giọng nói.
- Có thể ghi âm các cuộc họp và các sự kiện khác bằng cách chuyển đổi cuộc hội thoại bằng giọng nói thành bản sao văn bản.

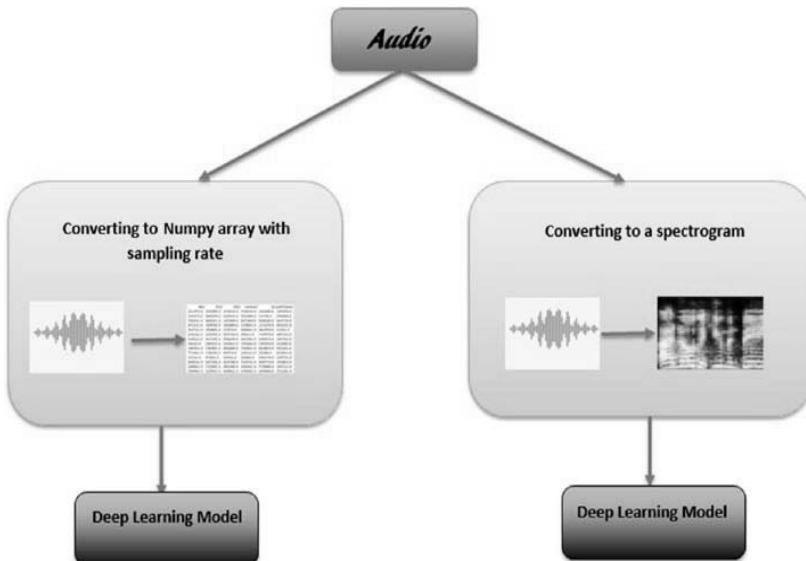
- Có thể chuyển đổi âm thanh trong các tệp video và âm thanh để có phụ đề của các từ được nói.
- Có thể dịch từ sang ngôn ngữ khác bằng cách nói vào một thiết bị bằng một ngôn ngữ và chuyển đổi văn bản thành lời nói bằng ngôn ngữ khác.

5.2.2. Lời nói dưới dạng dữ liệu

Bước đầu tiên để thực hiện bất kỳ hệ thống nhận dạng giọng nói tự động là để có được các tính năng. Nói cách khác, xác định các thành phần của sóng âm thanh hữu ích để nhận dạng nội dung ngôn ngữ và xóa tất cả các tính năng vô dụng khác chỉ là tiếng ồn.

Lời nói của mỗi người được lọc theo hình dạng của thanh quản cũng như là cả lưỡi và răng. Những âm thanh phát ra phụ thuộc vào hình dạng này. Để xác định âm vị được đưa ra chính xác, cần xác định hình dạng này một cách chính xác. Có thể nói rằng hình dạng của thanh quản biểu hiện thành một đường bao của phổ công suất thời gian ngắn. Công việc của MFCC là thể hiện chính xác đường bao này.

Lời nói cũng có thể được biểu diễn dưới dạng dữ liệu bằng cách chuyển đổi nó thành biểu đồ phổ (Hình 5.7).



Hình 5.7: *Lời nói được xem như dữ liệu*

5.2.3. Tính năng lời nói ánh xạ lời nói thành ma trận

MFCC được sử dụng rộng rãi trong nhận dạng giọng nói tự động.

Thang đo Mel liên quan đến tần số cảm nhận, hoặc cao độ của âm cơ bản với tần số đo thực tế của nó.

Có thể chuyển đổi âm thanh theo thang tần số sang thang đo Mel bằng công thức sau:

$$M(f) = 1125 \ln(1+f/700) \quad (5.1)$$

Để chuyển đổi nó trở lại tần số, sử dụng công thức sau:

```
M - 1 (m) = 700 \exp(m / 1125) - 1  
def mfcc(signal,samplerate=16000,winlen=0.025,winstep=0.01,  
        numcep=13, nfilt=26,nfft=512,lowfreq=0,highfreq=None,n:  
        preemph=0.97, ceplifter=22,appendEnergy=True)
```

Đây là các tham số được sử dụng:

- signal: đây là tín hiệu mà người đọc cần tính toán các tính năng MFCC, là một mảng của $N * 1$ (đọc tệp WAV).
- samplerate: đây là tỷ lệ lấy mẫu tín hiệu
- winlen: đây là chiều dài cửa sổ phân tích tính bằng giây. Theo mặc định, là 0,025 giây.
- winstep: đây là bước cửa sổ kế tiếp. Theo mặc định, là 0,01 giây.
- numcep: đây là số lượng cepstrum (cepstrum là kết quả của việc lấy biến đổi Fourier ngược của logarit của phổ ước tính của tín hiệu) mà hàm sẽ trả về. Theo mặc định, nó là 13.
- nfilt: đây là số lượng bộ lọc trong ngân hàng bộ lọc. Theo mặc định, nó là 26.
- nfft: đây là kích thước của biến đổi Fourier nhanh (FFT). Theo mặc định, nó là 512.
- lowfreq: đây là cạnh dải thấp nhất, tính bằng hertz. Theo mặc định, nó là 0.
- highfreq: đây là cạnh dải cao nhất, tính bằng hertz. Theo mặc định, đó là tỷ lệ mẫu chia cho 2.
- preemph: điều này áp dụng bộ lọc ưu tiên trước với sự ưu tiên là hệ số. 0 có nghĩa là không có bộ lọc. Theo mặc định, nó là 0,97.
- ceplifter: điều này áp dụng một bộ nâng cho các hệ số cepstral cuối cùng. 0 có nghĩa là không nâng lên. Theo mặc định, nó là 22.

- appendEnergy: hệ số cepstral zeroth được thay thế bằng logarit của tổng năng lượng khung nếu nó được đặt là true.

Hàm này trả về một mảng Numpy chứa các tính năng. Mỗi hàng chứa một vector tính năng.

5.2.4. Quang phổ: ánh xạ lời nói thành hình ảnh

Một quang phổ là một đại diện hình ảnh hoặc điện tử. Ý tưởng là chuyển đổi một tệp âm thanh thành hình ảnh và chuyển hình ảnh thành các mô hình học sâu như CNN và LSTM để phân tích và phân loại.

Biểu đồ phổ được tính là một chuỗi các FFT của các phân đoạn dữ liệu được. Một định dạng phổ biến là một biểu đồ với hai kích thước hình học; một trực biểu thị thời gian và một trực khác biểu thị tần số. Kích thước thứ ba sử dụng màu sắc hoặc kích thước của điểm để biểu thị biên độ của một tần số cụ thể tại một thời điểm cụ thể. Các biểu đồ phổ thường được tạo theo một trong hai cách. Chúng có thể được xấp xỉ như một ngân hàng bộ lọc kết quả từ một loạt các bộ lọc thông dài. Hoặc, trong Python, có một chức năng trực tiếp ánh xạ âm thanh thành phổ.

5.2.5. Xây dựng bộ phân loại để nhận dạng giọng nói thông qua các tính năng MEL-FREQUENCY CEPSTRAL COEFFICIENT (MFCC)

Để xây dựng một trình phân loại để nhận dạng giọng nói, cần cài đặt gói Python python_speech_features.

Người đọc có thể sử dụng lệnh pip cài đặt python_speech_features để cài đặt gói này

Hàm mfcc tạo ma trận tính năng cho tệp âm thanh. Để xây dựng một bộ phân loại nhận ra giọng nói của những người khác nhau, cần thu thập dữ liệu giọng nói ở định dạng WAV. Sau đó, chuyển đổi tất cả các tệp âm thanh thành ma trận bằng hàm mfcc. Chương trình để trích xuất các tính năng từ tệp WAV được hiển thị ở đây:

```
from python_speech_features import mfcc
from python_speech_features import delta
from python_speech_features import logfbank
import scipy.io.wavfile as wav

(samplerate,signal) = wav.read("audio.wav")
mfccfeatures = mfcc(signal,samplerate)
dmfccfeature = delta(mfccfeatures, 2)
fbankfeature = logfbank(signal,samplerate)

print(fbankfeature)
```

Nếu chạy chương trình ở trên, người đọc sẽ nhận được đầu ra ở dạng sau:

```
[[ 7.66608682  7.04137131  7.30715423 ...,  9.43362359  9.11932984  
  9.93454603]  
 [ 4.9474559   4.97057377  6.90352236 ...,  8.6771281   8.86454547  
  9.7975147 ]  
 [ 7.4795622   6.63821063  5.98854983 ...,  8.78622734  8.805521  
  9.83712966]  
 ...,  
 [ 7.8886269   6.57456605  6.47895433 ...,  8.62870034  8.79965464  
  9.67997298]  
 [ 5.73028657  4.87985847  6.64977329 ...,  8.64089442  8.62887745  
  9.90470194]  
 [ 8.8449656   6.67098127  7.09752316 ...,  8.84914694  8.97807983  
  9.45123015]]
```

Ở đây, mỗi hàng đại diện cho một vector tính năng.

Thu thập càng nhiều bản ghi âm giọng nói của một người càng tốt và nối thêm ma trận tính năng của từng tệp âm thanh trong ma trận này.

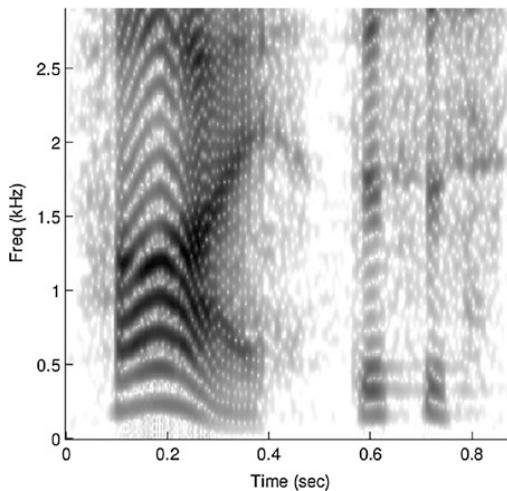
Điều này sẽ hoạt động như tập dữ liệu huấn luyện của người đọc.

Lặp lại các bước tương tự với tất cả các lớp khác.

Sau khi bộ dữ liệu được chuẩn bị, người đọc có thể điều chỉnh dữ liệu này vào bất kỳ mô hình học sâu nào (được sử dụng để phân loại) để phân loại giọng nói của những người khác nhau.

5.2.6. Xây dựng bộ phân loại để nhận dạng giọng nói thông qua quang phổ

Sử dụng phương pháp phổ kế chuyển đổi tất cả các tệp âm thanh thành hình ảnh (Hình 5.8), vì vậy tất cả những gì phải làm là chuyển đổi tất cả các tệp âm thanh trong dữ liệu huấn luyện thành hình ảnh và đưa các hình ảnh đó vào mô hình học sâu giống như làm một CNN.



Hình 5.8: Biểu đồ phổ mẫu

Đây là chương trình Python để chuyển đổi tệp âm thanh thành biểu đồ phổ:

```
import matplotlib.pyplot as plt
from scipy import signal
from scipy.io import wavfile

sample_rate, samples = wavfile.read('monoAudioFile.wav')
frequencies, times, spectrogram = signal.spectrogram(samples, sample_rate)

plt.imshow(spectrogram)
plt.ylabel('Freq(kHz)')
plt.xlabel('Time (sec)')
plt.show()
```

5.2.7. Cách tiếp cận nguồn mở để nhận dạng giọng nói

Có các gói nguồn mở có sẵn cho Python thực hiện chuyển đổi lời nói thành văn bản và chuyển văn bản thành giọng nói.

Sau đây là một số API chuyển đổi giọng nói thành văn bản nguồn mở:

- PocketSphinx
- Google Speech
- Google Cloud Speech
- Wit.ai
- Houndify
- IBM Speech to Text API
- Microsoft Bing Speech

Đã sử dụng tất cả những thứ này, có thể nói rằng chúng hoạt động khá tốt; giọng Mỹ đặc biệt rõ ràng.

Nếu quan tâm đến việc đánh giá độ chính xác của chuyển đổi, cần một số liệu: tỷ lệ lỗi từ (WER).

5.2.8. Tương lai của chuyển đổi lời nói thành văn bản

Công nghệ nhận dạng giọng nói đã và đang đạt được tiến bộ lớn. Hàng năm, nó chính xác hơn khoảng 10 đến 15 phần trăm so với năm trước. Trong tương lai, nó sẽ cung cấp giao diện tương tác nhất cho máy tính.

Có rất nhiều ứng dụng sẽ sớm được chứng kiến trên thị trường, bao gồm sách tương tác, điều khiển robot và giao diện xe tự lái. Dữ liệu lời nói cung cấp một số khả năng mới thú vị bởi vì đó là tương lai của ngành công nghiệp. Trí thông minh lời nói cho phép mọi người nhẫn tin, nhận hoặc đưa ra mệnh lệnh, đưa ra khiếu nại và thực hiện bất kỳ công việc nào mà họ đã sử dụng để nhập thủ công. Cung cấp trải nghiệm khách hàng tuyệt vời và có lẽ đó là lý do tại sao tất cả các bộ phận và doanh nghiệp đổi mới với khách hàng có xu hướng sử dụng các ứng dụng lời nói rất nhiều. Có thể thấy một tương lai tuyệt vời cho các nhà phát triển ứng dụng về lời nói.

5.3. PHÁT TRIỂN ỨNG DỤNG CHATBOT

Các hệ thống trí tuệ nhân tạo đóng vai trò giao diện cho tương tác giữa người và máy thông qua văn bản hoặc giọng nói được gọi là chatbot.

Các tương tác với chatbot có thể đơn giản hoặc phức tạp. Một ví dụ về sự tương tác đơn giản có thể hỏi về cập nhật tin tức mới nhất. Các tương tác có thể trở nên phức tạp hơn khi chúng liên quan đến việc khắc phục sự cố với điện thoại Android. Thuật ngữ chatbot đã trở nên phổ biến rộng rãi trong những năm qua và đã phát triển thành nền tảng ưa thích nhất cho sự tương tác của người dùng. Một hình thức nâng cao của một chatbot, giúp tự động hóa các nhiệm vụ do người dùng thực hiện.

Chương này về chatbot sẽ đóng vai trò là một hướng dẫn bao gồm tất cả những gì, làm thế nào, ở đâu, khi nào và tại sao có chatbot!

Cụ thể, sẽ bao gồm những điều sau đây:

- Tại sao muốn sử dụng chatbot
- Các thiết kế và chức năng của chatbot
- Các bước để xây dựng một chatbot
- Phát triển chatbot bằng API
- Hoạt động tốt nhất của chatbot

5.3.1. Tại sao có Chatbots?

Điều quan trọng đối với một chatbot là hiểu được thông tin nào mà người dùng đang tìm kiếm, được gọi là mục đích. Giả sử người dùng muốn biết nhà hàng chay gần nhất; người dùng có thể hỏi câu hỏi đó theo nhiều cách có thể. Một chatbot (cụ thể là trình phân loại nội dung bên trong chatbot) phải có thể hiểu được ý định vì người dùng muốn có câu trả lời đúng. Trong thực tế, để đưa ra câu trả lời đúng, chatbot phải có khả năng hiểu bối cảnh, ý định, thực thể và tình cảm. Chatbot thậm chí phải tính đến bất cứ điều gì được thảo luận trong phiên. Ví dụ, người dùng có thể hỏi câu hỏi giá gà ở tiệm KFC là bao nhiêu? Mặc dù người dùng đã hỏi giá, công cụ trò chuyện có thể hiểu nhầm và cho rằng người dùng đang tìm nhà hàng. Vì vậy, để đáp lại, chatbot có thể cung cấp tên của nhà hàng.

5.3.2. Thiết kế và chức năng của Chatbots

Một chatbot kích hoạt các cuộc trò chuyện thông minh với con người thông qua ứng dụng AI.

Giao diện thông qua đó cuộc trò chuyện diễn ra được tạo điều kiện thông qua văn bản nói hoặc văn bản. Facebook Messenger, Slack và Telegram sử dụng các nền tảng nhắn tin chatbot, phục vụ nhiều mục đích, bao gồm đặt hàng sản phẩm trực tuyến, đầu tư và quản lý tài chính, v.v. Một khía cạnh quan trọng của chatbot là chúng biến khả năng trò chuyện theo ngữ cảnh. Các chatbot trò chuyện với người dùng theo cách tương tự như cách con người trò chuyện trong cuộc sống hàng ngày. Mặc dù các chatbot có thể trò chuyện theo ngữ cảnh, nhưng chúng vẫn còn một chặng đường dài để giao tiếp theo ngữ cảnh với mọi thứ.

Nhưng giao diện trò chuyện đang sử dụng ngôn ngữ để kết nối máy với con người, giúp mọi người hoàn thành công việc một cách thuận tiện bằng cách cung cấp thông tin theo cách thức theo ngữ cảnh.

Hơn nữa, chatbot đang xác định lại cách thức kinh doanh đang tiến hành. Từ việc tiếp cận với người tiêu dùng để chào đón họ đến hệ sinh thái của doanh nghiệp để cung cấp thông tin cho người tiêu dùng về các sản phẩm khác nhau và các tính năng của họ, chatbot đang giúp tất cả. Chatbot đang nổi lên như một cách thuận tiện nhất để giao dịch với người tiêu dùng một cách kịp thời và thỏa đáng.

5.3.3. Các bước xây dựng Chatbot

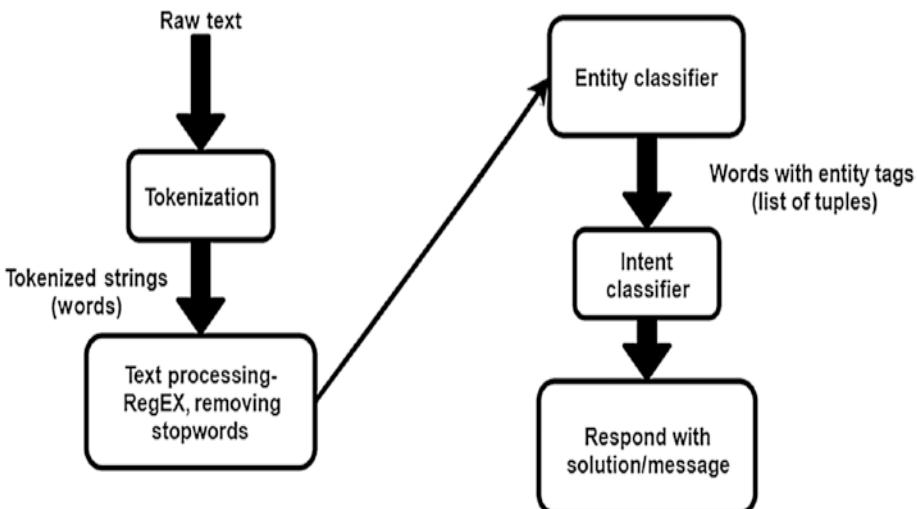
Một chatbot được xây dựng để giao tiếp với người dùng và mang lại cho họ cảm giác rằng họ đang giao tiếp với con người chứ không phải một

cái máy. Nhưng khi người dùng đưa ra đầu vào, thông thường họ sẽ không cung cấp đầu vào theo cách thích hợp.

Nói cách khác, họ có thể nhập các dấu chấm câu không cần thiết hoặc có thể có nhiều cách khác nhau để hỏi cùng một câu hỏi.

Ví dụ: đối với các nhà hàng ở gần tôi? Một người dùng có thể nhập vào các nhà hàng bên cạnh tôi?

Do đó, cần xử lý trước dữ liệu để công cụ chatbot có thể dễ dàng hiểu được. Hình 5.9 cho thấy quá trình, được trình bày chi tiết trong các phần sau.



Hình 5.9: Lưu đồ để hiển thị cách công cụ chatbot xử lý chuỗi đầu vào và trả lời hợp lệ.

5.3.3.1. Tiền xử lý văn bản và tin nhắn

Tiền xử lý văn bản và tin nhắn bao gồm một số bước

Mã thông báo

Cắt các câu thành các từ đơn (được gọi là mã thông báo) được gọi là mã thông báo. Trong Python, nói chung, một chuỗi được token hóa và được lưu trữ trong một danh sách.

Ví dụ, câu “Artificial intelligence is all about applying mathematics”. Trở thành như sau:

[“Artifcial”, “intelligence”, “is”, “all”, “about”, “applying”, “mathematics”]

Đây là chương trình ví dụ:

```
from nltk.tokenize import TreebankWordTokenizer
l = "Artificial intelligence is all about applying mathematics"
token = TreebankWordTokenizer(). tokenize(l)
print(token)
```

Xóa dấu chấm câu

Cũng có thể loại bỏ các dấu chấm câu không cần thiết trong câu.

Ví dụ: “Tôi có thể nhận được danh sách các nhà hàng, nơi giao hàng tận nhà”.

“Tôi có thể nhận danh sách các nhà hàng giao hàng tận nhà”.

Đây là chương trình ví dụ:

```
from nltk.tokenize import TreebankWordTokenizer
from nltk.corpus import stopwords
l = "Artificial intelligence is all about applying
mathematics!"
token = TreebankWordTokenizer(). tokenize(l)
output = []
output = [k for k in token if k.isalpha()]
print(output)
```

Xóa từ dừng

Các từ dừng là những từ có trong một câu mà không có nhiều khác biệt nếu bị loại bỏ. Mặc dù định dạng của câu thay đổi, điều này giúp ích rất nhiều trong việc hiểu ngôn ngữ tự nhiên (NLU).

Ví dụ: câu “Thông minh nhân tạo có thể thay đổi lối sống của người dân” trở thành như sau sau khi loại bỏ các từ dừng:

“Trí thông minh nhân tạo thay đổi lối sống con người.”

Đây là chương trình ví dụ:

```
from nltk.tokenize import TreebankWordTokenizer
from nltk.corpus import stopwords
l = "Artificial intelligence is all about applying mathematics"
token = TreebankWordTokenizer(). tokenize(l)
stop_words = set(stopwords.words('english'))
output= []
for k in token:
    if k not in stop_words:
        output.append(k)
print(output)
```

Những từ được coi là từ dùng có thể thay đổi. Có một số bộ từ dùng được xác định trước được cung cấp bởi bộ công cụ ngôn ngữ tự nhiên (NLTK), Google và hơn thế nữa.

5.3.3.2. Công nhận thực thể được đặt tên

Công nhận thực thể được đặt tên (NER), còn được gọi là nhận dạng thực thể, là nhiệm vụ phân loại các thực thể trong văn bản thành các lớp được xác định trước như tên của một quốc gia, tên của một người, v.v. Cũng có thể định nghĩa các lớp riêng

Ví dụ: áp dụng NER cho câu “trận đấu đánh cầu hôm nay Ấn Độ vs Úc là tuyệt vời”. Hãy cho kết quả như sau:

[Hôm nay] Thời gian [Ấn Độ] Quốc gia vs [Úc] Quốc gia [cricket]
Trận đấu trò chơi thật tuyệt vời.

Để chạy chương trình cho NER, cần tải xuống và nhập các gói cần thiết, như được đề cập trong đoạn chương trình sau.

Sử dụng Stanford NER

Để chạy chương trình, tải xuống english.all.3 class.distsim.crf.ser.gz và các tệp stanford-ner.jar.

```
from nltk.tag import StanfordNERTagger
from nltk.tokenize import word_tokenize

StanfordNERTagger("stanford-ner/classifiers/english.all.3class.
distsim.crf.ser.gz",
"stanford-ner/stanford-ner.jar")

text = "Ron was the founder of Ron Institute at New York"
text = word_tokenize(text)
ner_tags = ner_tagger.tag(text)

print(ner_tags)
```

Tải xuống tệp ner_model.dat của MITIE để chạy chương trình

```

from mitie.mitie import *
from nltk.tokenize import word_tokenize

print("loading NER model...")
ner = named_entity_extractor("mitie/MITIE-models/english/
ner_model.dat".encode("utf8"))

text = "Ron was the founder of Ron Institute at New york".
encode("utf-8")
text = word_tokenize(text)

ner_tags = ner.extract_entities(text)
print("\nEntities found:", ner_tags)

for e in ner_tags:
    range = e[0]
    tag = e[1]
    entity_text = " ".join(text[i].decode() for i in range)
    print( str(tag) + " : " + entity_text)

```

Tải xuống tệp Total_word_feature_extractor.dat của MITIE (<https://github.com/mit-nlp/MITIE>) để chạy chương trình.

```

from mitie.mitie import *

sample = ner_training_instance([b"Ron", b"was", b"the", b"founder",
b"of", b"Ron", b"Institute", b"at", b"New", b"York", b"."])

sample.add_entity(range(0, 1), "person".encode("utf-8"))
sample.add_entity(range(5, 7), "organization".encode("utf-8"))
sample.add_entity(range(8, 10), "Location".encode("utf-8"))

```

```

trainer = ner_trainer("mitie/MITIE-models/english/total_word_
feature_extractor.dat".encode("utf-8"))

trainer.add(sample)

ner = trainer.train()

tokens = [b"John", b"was", b"the", b"founder", b"of", b"John",
b"University", b"."]
entities = ner.extract_entities(tokens)
print ("\nEntities found:", entities)
for e in entities:
    range = e[0]
    tag = e[1]
    entity_text = " ".join(str(tokens[i]) for i in range)
    print ("    " + str(tag) + ": " + entity_text)

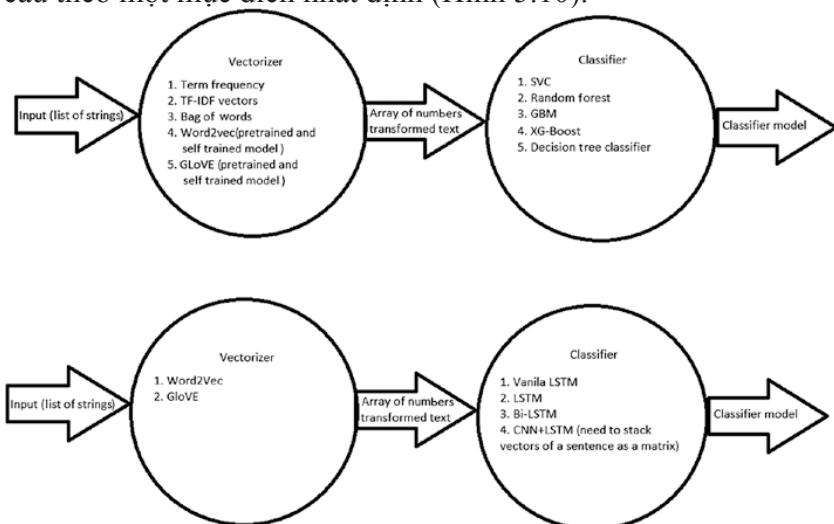
```

5.3.3.3. Phân loại nội dung

Phân loại nội dung là bước trong NLU nơi có gắng hiểu người dùng muốn gì. Dưới đây là hai ví dụ về đầu vào cho một chatbot để tìm địa điểm gần đó:

- Tôi cần mua đồ tạp hóa. Giá trị: Mục đích là tìm cửa hàng tạp hóa gần đó.
- Tôi muốn ăn chay. Ý định là tìm kiếm những nhà hàng gần đó, những quán chay lý tưởng.

Về cơ bản, cần hiểu những gì người dùng đang tìm kiếm và phân loại yêu cầu theo một mục đích nhất định (Hình 5.10).



Hình 5.10: Sơ đồ chung của phân loại nội dung, từ câu đến vector đến mô hình

Để làm điều này, cần huấn luyện một mô hình để phân loại các yêu cầu thành ý định bằng thuật toán, đi từ câu đến vectơ đến mô hình.

Nhúng từ ngữ

Nhúng từ ngữ là kỹ thuật chuyển đổi văn bản thành số. Thật khó để áp dụng bất kỳ thuật toán trong văn bản. Do đó, phải chuyển đổi nó thành số.

Sau đây là các loại kỹ thuật nhúng từ khác nhau.

a) Đếm Vector

Giả sử có ba tài liệu (D1, D2 và D3) và có N từ duy nhất trong nhóm tài liệu. Tạo một ma trận ($D \times N$), được gọi là C, được gọi là vectơ đếm. Mỗi mục nhập của ma trận là tần số của từ duy nhất trong tài liệu đó.

Hãy xem điều này bằng cách sử dụng một ví dụ.

- D1: Pooja is very lazy.
- D2: But she is intelligent.
- D3: She hardly comes to class.

Ở đây, $D = 3$ và $N = 12$.

Những từ đặc biệt như là hardly, lazy, But, to, Pooja, she, intelligent, comes, very, class và is.

Do đó, vectơ đếm, C, sẽ như sau:

	Hardly	laziest	But	to	Pooja	she	intelligent	comes	very	class	is
D1	0	1	0	0	1	0	0	0	1	0	1
D2	0	0	1	0	0	1	1	0	0	0	1
D3	1	0	0	1	0	1	0	1	0	1	0

b) Term Frequency-Inverse Document Frequency (TF-IDF)

Đối với kỹ thuật này, người đọc cung cấp cho mỗi từ trong câu một số tùy thuộc vào số lần từ đó xảy ra trong câu đó và cũng tùy thuộc vào tài liệu. Các từ xuất hiện nhiều lần trong một câu và không quá nhiều lần trong một tài liệu sẽ có giá trị cao.

Ví dụ, hãy xem xét một tập hợp các câu:

- I am a boy

- I am a girl
- Where do you live

TF-IDF biến đổi bộ tính năng cho các câu trước đó, như hiển thị ở đây:

Am	Boy	Girl	Where	do	you	live
1. 0.60	0.80	0	0	0	0	0
2. 0.60	0	0.80	0	0	0	0
3.0	0	0	0.5	0.5	0.5	0.5

Có thể nhập gói TFIDF và sử dụng để tạo bảng này.

Bây giờ hãy xem một số chương trình mẫu. Có thể sử dụng trình phân loại vectơ hỗ trợ trên các tính năng được chuyển đổi TF-IDF của chuỗi yêu cầu.

```
#import required packages
import pandas as pd
from random import sample
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, accuracy_score
# read csv file
data = pd.read_csv("intent1.csv")
print(data.sample(6))
```

Trước khi tiếp tục với chương trình, ở đây, một ví dụ về tập dữ liệu:

Description (Message)	intent_label (Target)
good non-veg restaurant near me	0
i am looking for a hospital	1
good hospital for heart operation	1
international school for kids	2
non-veg restaurant around me	0
school for small Kids	2

Trong ví dụ này, đây là các giá trị sử dụng:

- 0 có nghĩa là tìm kiếm một nhà hàng.
- 1 có nghĩa là tìm kiếm một bệnh viện.
- 2 có nghĩa là tìm kiếm một trường học.

Bây giờ hãy để cùng làm việc với bộ dữ liệu.

```
# split dataset into train and test.  
X_train, X_test, Y_train, Y_test = train_test_split(data  
["Description"], data["intent_label"], test_size=3)  
print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)  
  
# vectorize the input using tfidf values.  
tfidf = TfidfVectorizer()  
tfidf = tfidf.fit(X_train)  
X_train = tfidf.transform(X_train)  
X_test = tfidf.transform(X_test)  
  
# label encoding for different categories of intents  
le = LabelEncoder().fit(Y_train)  
Y_train = le.transform(Y_train)  
Y_test = le.transform(Y_test)  
  
# other models like GBM, Random Forest may also be used  
model = SVC()  
model = model.fit(X_train, Y_train)  
p = model.predict(X_test)  
# calculate the f1_score. average="micro" since we want to  
calculate score for multiclass.  
# Each instance(rather than class(search for macro average))  
contribute equally towards the scoring.  
print("f1_score:", f1_score(Y_test, p, average="micro"))  
print("accuracy_score:", accuracy_score(Y_test, p))
```

5.3.3.4. WORD2VEC

Có nhiều phương pháp khác nhau để nhận vecto từ cho một câu, nhưng lý thuyết chính đằng sau tất cả các kỹ thuật là đưa ra các từ tương tự một biểu diễn vecto tương tự. Vì vậy, những từ như người đàn ông và chàng trai và cô gái sẽ có vecto tương tự. Độ dài của mỗi vecto có thể được

đặt. Ví dụ về các kỹ thuật Word2vec bao gồm GloVe và CBOW (n-gram có hoặc không bỏ qua gram).

Có thể sử dụng Word2vec bằng cách huấn luyện nó cho tập dữ liệu (nếu có đủ dữ liệu cho sự có) hoặc có thể sử dụng dữ liệu được huấn luyện trước. Word2vec có sẵn trên Internet. Các mô hình tiền xử lý đã được huấn luyện trên các tài liệu không lò như dữ liệu Wikipedia, tweet, v.v. và chúng hầu như luôn luôn tốt cho tất cả vấn đề.

Một ví dụ về một số kỹ thuật mà có thể sử dụng để huấn luyện trình phân loại nội dung là sử dụng 1D-CNN trên các vectơ từ của các từ trong câu, được thêm vào danh sách cho mỗi câu.

```
# import required packages
from gensim.models import Word2Vec
import pandas as pd
import numpy as np
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils.np_utils import to_categorical
from keras.layers import Dense, Input, Flatten
from keras.layers import Conv1D, MaxPooling1D, Embedding, Dropout
from keras.models import Model

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, accuracy_score
# read data
data = pd.read_csv("intent1.csv")

# split data into test and train
X_train, X_test, Y_train, Y_test = train_test_split(data
["Description"], data["intent_label"], test_size=6)

# label encoding for different categories of intents
le = LabelEncoder().fit(Y_train)
Y_train = le.transform(Y_train)
Y_test = le.transform(Y_test)

# get word_vectors for words in training set
X_train = [sent for sent in X_train]
X_test = [sent for sent in X_test]
```

```
# get word_vectors for words in training set
X_train = [sent for sent in X_train]
X_test = [sent for sent in X_test]
# by default genism.Word2Vec uses CBOW, to train word vecs.
We can also use skipgram with it
# by setting the "sg" attribute to number of skips we want.
# CBOW and Skip gram for the sentence "Hi Ron how was your
day?" becomes:
# Continuos bag of words: 3-grams {"Hi Ron how", "Ron how was",
"how was your" ...}
# Skip-gram 1-skip 3-grams: {"Hi Ron how", "Hi Ron was", "Hi
how was", "Ron how
# your", ...}
# See how: "Hi Ron was" skips over "how".
# Skip-gram 2-skip 3-grams: {"Hi Ron how", "Hi Ron was", "Hi
Ron your", "Hi was
# your", ...}
# See how: "Hi Ron your" skips over "how was".
# Those are the general meaning of CBOW and skip gram.
word_vecs = Word2Vec(X_train)
print("Word vectors trained")

# prune each sentence to maximum of 20 words.
max_sent_len = 20

# tokenize input strings
tokenizer = Tokenizer()
tokenizer.fit_on_texts(X_train)
sequences = tokenizer.texts_to_sequences(X_train)
sequences_test = tokenizer.texts_to_sequences(X_test)
word_index = tokenizer.word_index
vocab_size = len(word_index)
```

```
# sentences with less than 20 words, will be padded with zeroes  
to make it of length 20  
# sentences with more than 20 words, will be pruned to 20.  
x = pad_sequences(sequences, maxlen=max_sent_len)  
X_test = pad_sequences(sequences_test, maxlen=max_sent_len)  
  
# 100 is the size of wordvec.  
embedding_matrix = np.zeros((vocab_size + 1, 100))  
  
# make matrix of each word with its word_vectors for the CNN model.  
# so each row of a matrix will represent one word. There will  
be a row for each word in  
  
# the training set  
for word, i in word_index.items():  
    try:  
        embedding_vector = word_vecs[word]  
    except:  
        embedding_vector = None  
    if embedding_vector is not None:  
        embedding_matrix[i] = embedding_vector  
print("Embeddings done")  
vocab_size = len(embedding_matrix)  
  
# CNN model requires multiclass labels to be converted into one  
hot encoding.  
# i.e. each column represents a label, and will be marked one  
for corresponding label.  
y = to_categorical(np.asarray(Y_train))  
  
embedding_layer = Embedding(vocab_size,  
                             100,  
                             weights=[embedding_matrix],  
                             input_length=max_sent_len,  
                             trainable=True)  
sequence_input = Input(shape=(max_sent_len,), dtype='int32')
```

```

# stack each word of a sentence in a matrix. So each matrix
represents a sentence.
# Each row in a matrix is a word(Word Vector) of a sentence.
embedded_sequences = embedding_layer(sequence_input)

# build the Convolutional model.
l_cov1 = Conv1D(128, 4, activation='relu')(embedded_sequences)
l_pool1 = MaxPooling1D(4)(l_cov1)
l_flat = Flatten()(l_pool1)

hidden = Dense(100, activation='relu')(l_flat)
preds = Dense(len(y[0]), activation='softmax')(hidden)
model = Model(sequence_input, preds)
model.compile(loss='binary_crossentropy',optimizer='Adam')

print("model fitting - simplified convolutional neural
network")
model.summary()

# train the model
model.fit(x, y, epochs=10, batch_size=128)

#get scores and predictions.
p = model.predict(X_test)
p = [np.argmax(i) for i in p]
score_cnn = f1_score(Y_test, p, average="micro")
print("accuracy_score:",accuracy_score(Y_test, p))
print("f1_score:", score_cnn)

```

Sự phù hợp mô hình là một mạng nơ ron tích chập đơn giản hóa, như được hiển thị ở đây:

Layer (Type)	Output Shape	Param #
input_20 (inputlayer)	(none, 20)	0
embedding_20 (embedding)	(none, 20, 100)	2800
conv1d_19 (Conv1D)	(none, 17, 128)	51328
max_pooling1d_19 (Maxpooling)	(none, 4, 128)	0
flatten_19 (Flatten)	(none, 512)	0
dense_35 (Dense)	(none, 100)	51300
dense_36 (Dense)	(none, 3)	303

Dưới đây là số lượng tham số:

- Tổng thông số: 105,731
- Thông số có thẻ huấn luyện: 105,731
- Tham số không thẻ kiểm soát: 0

5.3.3.5. Xây dựng phản hồi

Phản hồi là một phần quan trọng khác của chatbot. Dựa trên cách chatbot trả lời, người dùng có thể bị thu hút bởi nó. Bất cứ khi nào một chatbot được tạo ra, điều cần lưu ý là người dùng của nó. Cần biết ai sẽ sử dụng nó và cho mục đích gì sẽ được sử dụng. Ví dụ: một chatbot cho một trang web nhà hàng sẽ chỉ được hỏi về nhà hàng và thực phẩm. Vì vậy, biết ít nhiều những câu hỏi sẽ được hỏi. Do đó, đối với mỗi mục đích, lưu trữ nhiều câu trả lời có thể được sử dụng sau khi xác định mục đích, vì vậy người dùng sẽ không nhận được cùng một câu trả lời nhiều lần. Cũng có thể có một ý định cho bất kỳ câu hỏi ngoài ngữ cảnh nào; ý định đó có thể có nhiều câu trả lời và chọn ngẫu nhiên, chatbot có thể trả lời.

Ví dụ: nếu mục đích là của Hello, thì có thể có nhiều câu trả lời như là Hello, Hello! How are you? Hi, can I help you?".

Chatbot có thể chọn bất kỳ một câu trả lời ngẫu nhiên.

Trong chương mẫu sau, đang lấy đầu vào từ người dùng, nhưng trong chatbot gốc, mục đích được xác định bởi chính chatbot dựa trên bất kỳ câu hỏi nào của người dùng.

```
import random
intent = input()
output = ["Hello! How are you","Hello! How are you doing","Hii!
How can I help you","Hey! There","Hiiii","Hello! How can I
assist you?","Hey! What's up?"]
if(intent == "Hii"):
    print(random.choice(output))
```

5.3.4. Thực tiễn tốt nhất về phát triển chatbot

Trong khi xây dựng một chatbot, điều quan trọng là phải hiểu rằng có một số thực tiễn tốt nhất có thể được tận dụng. Điều này sẽ giúp tạo ra một chatbot thân thiện với người dùng thành công có thể hoàn thành mục đích của nó để có một cuộc trò chuyện liền mạch với người dùng.

Một trong những điều quan trọng nhất trong mối quan hệ này là phải biết rõ đối tượng mục tiêu. Tiếp đến là những thứ khác như xác định các

tình huống sử dụng, đặt âm của cuộc trò chuyện và xác định các nền tảng nhán tin.

Bằng cách tuân thủ các thực tiễn tốt nhất sau đây, mong muốn đảm bảo các cuộc hội thoại liền mạch với người dùng có thể trở thành hiện thực.

5.3.4.1. Người dùng tiềm năng

Hiểu biết sâu sắc về đối tượng mục tiêu là bước đầu tiên để xây dựng một chatbot thành công. Giai đoạn tiếp theo là để biết mục đích mà chatbot được tạo ra.

Dưới đây là một số điểm cần nhớ:

- Biết mục đích của chatbot cụ thể là gì. Nó có thể là một chatbot để giải trí khán giả, tạo điều kiện cho người dùng giao dịch, cung cấp tin tức hoặc phục vụ như một kênh dịch vụ khách hàng.
- Làm cho chatbot thân thiện hơn với khách hàng bằng cách tìm hiểu về sản phẩm của khách hàng.

TÀI LIỆU THAM KHẢO

- [1] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Volume 1 - Volume 01. CVPR ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: <http://dx.doi.org/10.1109/CVPR.2005.177> (cited on pages 25, 51, 124).
- [2] T. Ojala, M. Pietikainen, and T. Maenpaa. “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns”. In: Pattern Analysis and Machine Intelligence, IEEE Transactions on 24.7 (2002), pages 971–987 (cited on pages 25, 51, 124).
- [3] Richard HR Hahnloser et al. “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit”. In: Nature 405.6789 (2000), page 947 (cited on pages 126, 128).
- [4] Richard H. R. Hahnloser, H. Sebastian Seung, and Jean-Jacques Slotine. “Permitted and Forbidden Sets in Symmetric Threshold-dòngar Networks”. In: Neural Comput. 15.3 (Mar. 2003), pages 621–638. ISSN: 0899-7667.
- [5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: Nature 521.7553 (2015), pages 436–444 (cited on pages 21, 126, 128).
- [6] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. “Rectifier nondòngarities improve neural network acoustic models”. In: in ICML Workshop on Deep Learning for Audio, Speech and Language Processing. 2013 (cited on page 126).
- [7] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: CoRR abs/1512.03385 (2015). URL: <http://arxiv.org/abs/1512.03385> (cited on pages 113, 126, 188, 192, 279).
- [8] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and Accurate Deep Network Learning by Exponential Dòngar Units (ELUs)”. In: CoRR abs/1511.07289 (2015). URL: <http://arxiv.org/abs/1511.07289> (cited on page 126).
- [9] Donald Hebb. The organization of behavior: A neuropsychological theory. Wiley, 1949 (cited on page 128).
- [10] Jürgen Schmidhuber. “Deep Learning in Neural Networks: An Overview”. URL: <http://arxiv.org/abs/1404.7828> (cited on pages 29, 128).
- [11] Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. Elements of

Artificial Neural Networks. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0-262-13328-8 (cited on pages 128, 132).

- [12] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: Psychological Review (1958), pages 65–386 (cited on pages 22, 129, 130).
- [13] M. Minsky and S. Papert. Perceptrons. Cambridge, MA: MIT Press, 1969 (cited on pages 22, 129).
- [14] Mikel Olazaran. “A Sociological Study of the Official History of the Perceptrons Controversy”. In: Social Studies of Science 26.3 (1996), pages 611–659. ISSN: 03063127. URL: <http://www.jstor.org/stable/285702> (cited on page 129).
- [15] P. J. Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences”. PhD thesis. Harvard University, 1974 (cited on pages 23, 129).
- [16] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Neurocomputing: Foundations of Research”. In: edited by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chapter Learning Representations by Back-propagating Errors, pages 696–699. ISBN: 0-262-01097-6.
- [17] <https://www.coursera.org/learn/machine-learning> (cited on pages 88, 132, 137, 141).
- [18] Kishan Mehrotra, Chilukuri K. Mohan, and Sanjay Ranka. Elements of Artificial Neural Networks. Cambridge, MA, USA: MIT Press, 1997. ISBN: 0-262-13328-8 (cited on pages 128, 132).
- [19] Michael Nielsen. Chapter 2: How the backpropagation algorithm works. <http://neuralnetworksanddeeplearning.com/chap2.html>. 2017 (cited on pages 137, 141).
- [20] Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition. [http:](http://)
- [21] Matt Mazur. A Step by Step Backpropagation Example. <https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>. 2015 (cited on pages 137, 141).
- [22] Geoffrey Hinton. Neural Networks for Machine Learning. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (cited on pages 112, 164).
- [23] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: J. Mach.

Learn. Res. 12 (July 2011), pages 2121– 2159. ISSN: 1532-4435.

- [24] Matthew D. Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. In: CoRR abs/1212.5701 (2012). URL: <http://arxiv.org/abs/1212.5701> (cited on page 164).
- [25] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: CoRR abs/1412.6980 (2014). URL: <http://arxiv.org/abs/1412.6980> (cited on pages 164, 311).
- [26] Greg Heinrich. NVIDIA DIGITS: Weight Initialization. <https://github.com/NVIDIA/DIGITS/blob/master/examples/weight-init/README.md>. 2015 (cited on pages 165, 167).
- [27] Yann LeCun et al. “Efficient BackProp”. In: Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop. London, UK, UK: Springer-Verlag, 1998, pages 9–50. ISBN: 3-540-65311-2. URL: <http://dl.acm.org/citation.cfm?id=645754.668382> (cited on pages 23, 166).
- [28] https://keras.io/initializers/#glorot_uniform. 2016 (cited on page 167).
- [29] Warren S. McCulloch and Walter Pitts. “Neurocomputing: Foundations of Research”. In: edited by James A. Anderson and Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Chapter A Logical Calculus of the Ideas Immanent in Nervous Activity, pages 15–27. ISBN: 0-262-01097-6.
- [30] F. Rosenblatt. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan, 1962 (cited on page 22).
- [31] Balázs Csanad Csaji. “Approximation with Artificial Neural Networks”. In: MSc Thesis, Etvos Lornd University (ELTE), Budapest, Hungary (2001) (cited on page 23).
- [32] Yann Lecun et al. “Gradient-based learning applied to document recognition”. In: Proceedings of the IEEE. 1998, pages 2278–2324 (cited on pages 24, 195, 219, 227).
- [33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning. <http://www.deeplearningbook.org>. MIT Press, 2016 (cited on pages 22, 24, 27, 42, 54, 56, 82, 95, 98, 113, 117, 169, 194, 252).
- [34] Jason Brownlee. What is Deep Learning? <http://machinelearningmastery.com/what-is-deep-learning/>. 2016 (cited on page 24).
- [35] Robert M. Haralick, K. Shanmugam, and Its’Hak Dinstein. “Textural Features for Image Classification”. In: IEEE Transactions on Systems, Man, and Cybernetics SMC-3.6 (Nov. 1973), pages 610–621. ISSN:

- 0018-9472. DOI: 10 .1109 /tsmc .1973 .4309314. URL: <http://dx.doi.org/10.1109/tsmc.1973.4309314> (cited on page 25).
- [36] Ming-Kuei Hu. “Visual pattern recognition by moment invariants”. In: Information Theory, IRE Transactions on 8.2 (Feb. 1962), pages 179–187. ISSN: 0096-1000 (cited on page 25).
- [37] A. Khotanzad and Y. H. Hong. “Invariant Image Recognition by Zernike Moments”. In: IEEE Trans. Pattern Anal. Mach. Intell. 12.5 (May 1990), pages 489–497. ISSN: 0162-8828. DOI: 10.1109/34.55109. URL: <http://dx.doi.org/10.1109/34.55109> (cited on page 25).
- [38] Jing Huang et al. “Image Indexing Using Color Correlograms”. In: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR ’97). CVPR ’97. Washington, DC, USA: IEEE Computer Society, 1997, pages 762–. ISBN: 0-8186-7822-4. URL: <http://dl.acm.org/citation.cfm?id=794189.794514> (cited on page 25).
- [39] Edward Rosten and Tom Drummond. “Fusing Points and Dongs for High Performance Tracking”. In: Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2. ICCV ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 1508–1515. ISBN: 0-7695-2334-X-02.
- [40] Chris Harris and Mike Stephens. “A combined corner and edge detector”. In: In Proc. of Fourth Alvey Vision Conference. 1988, pages 147–151 (cited on page 25).
- [41] David G. Lowe. “Object Recognition from Local Scale-Invariant Features”. In: Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2. ICCV ’99. Washington, DC, USA: IEEE Computer Society, 1999, pages 1150–. ISBN: 0-7695-0164-8. URL: <http://dl.acm.org/citation.cfm?id=850924.851523> (cited on page 25).
- [42] Herbert Bay et al. “Speeded-Up Robust Features (SURF)”. In: Comput. Vis. Image Underst. 110.3 (June 2008), pages 346–359. ISSN: 1077-3142. DOI: 10.1016/j.cviu.2007.09.014.
- [43] Michael Calonder et al. “BRIEF: Binary Robust Independent Elementary Features”. In: Proceedings of the 11th European Conference on Computer Vision: Part IV. ECCV’10. Heraklion, Crete, Greece: Springer-Verlag, 2010, pages 778–792. ISBN: 3-642-15560-X, 978-3-642-15560-4. URL: <http://dl.acm.org/citation.cfm?id=1888089.1888148> (cited on page 25).
- [44] Ethan Rublee et al. “ORB: An Efficient Alternative to SIFT or SURF”. In: Proceedings of the 2011 International Conference on Computer Vision. ICCV ’11. Washington, DC, USA: IEEE Computer Society,

- 2011, pages 2564–2571. ISBN: 978-1-4577-1101-5. DOI: 10.1109/ICCV.2011.6126544. URL: <http://dx.doi.org/10.1109/ICCV.2011.6126544> (cited on page 25).
- [45] Navneet Dalal and Bill Triggs. “Histograms of Oriented Gradients for Human Detection”. In: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Volume 1 - Volume 01. CVPR ’05. Washington, DC, USA: IEEE Computer Society, 2005, pages 886–893. ISBN: 0-7695-2372-2. DOI: 10.1109/CVPR.2005.177. URL: <http://dx.doi.org/10.1109/CVPR.2005.177> (cited on pages 25, 51, 124).
- [46] Pedro F. Felzenszwalb et al. “Object Detection with Discriminatively Trained Part-Based Models”. In: IEEE Trans. Pattern Anal. Mach. Intell. 32.9 (Sept. 2010), pages 1627–1645. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2009.167. URL: <http://dx.doi.org/10.1109/TPAMI.2009.167> (cited on page 26).
- [47] Tomasz Malisiewicz, Abhinav Gupta, and Alexei A. Efros. “Ensemble of Exemplar-SVMs for Object Detection and Beyond”. In: ICCV. 2011 (cited on page 26).
- [48] Richard Szeliski. Computer Vision: Algorithms and Applications. 1st. New York, NY, USA: Springer-Verlag New York, Inc., 2010. ISBN: 1848829345, 9781848829343 (cited on pages 171, 177).
- [49] Victor Powell. Image Kernels Explained Visually. <http://setosa.io/ev/image-kernels/>. 2015 (cited on page 177).
- [50] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: Advances in Neural Information Processing Systems 25. Edited by F. Pereira et al. Curran Associates, Inc., 2012, pages 1097–1105.
- [51] Andrej Karpathy. Convolutional Networks. <http://cs231n.github.io/convolutional-networks/> (cited on pages 186, 187, 191).
- [52] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: CoRR abs/1502.03167 (2015). URL: <http://arxiv.org/abs/1502.03167> (cited on pages 189, 193).
- [53] Dmytro Mishkin. CaffeNet - Benchmark - Batch Norm. <https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md> (cited on page 190).
- [54] Reddit community contributors. Batch Normalization before or after ReLU?

- [55] Christian Szegedy et al. “Going Deeper with Convolutions”. In: Computer Vision and Pattern Recognition (CVPR). 2015. URL: <http://arxiv.org/abs/1409.4842> (cited on pages 113, 192, 280).
- [56] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size”. In: CoRR abs/1602.07360 (2016). URL: <http://arxiv.org/abs/1602.07360> (cited on pages 192, 280, 281).
- [57] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large- Scale Image Recognition”. In: CoRR abs/1409.1556 (2014). URL: <http://arxiv.org/abs/1409.1556> (cited on pages 113, 192, 195, 227, 229, 278).
- [58] Pierre Sermanet et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: CoRR abs/1312.6229 (2013). URL: <http://arxiv.org/abs/1312.6229> (cited on page 229).
- [59] David Mount and Sunil Arya. ANN: A Library for Approximate Nearest Neighbor Searching. <https://www.cs.umd.edu/~mount/ANN/>. 2010 (cited on page 81).
- [60] Marius Muja and David G. Lowe. “Scalable Nearest Neighbor Algorithms for High Dimensional Data”. In: Pattern Analysis and Machine Intelligence, IEEE Transactions on 36 (2014) (cited on pages 79, 81).
- [61] Erik Bernhardsson. Annoy: Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk. <https://github.com/spotify/annoy>. 2015 (cited on page 81).
- [62] Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition.
- [63] Ian H. Witten, Eibe Frank, and Mark A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123748569, 9780123748560 (cited on page 88).
- [64] Peter Harrington. Machine Learning in Action. Greenwich, CT, USA: Manning Publications Co., 2012. ISBN: 1617290181, 9781617290183 (cited on page 88).
- [65] Stephen Marsland. Machine Learning: An Algorithmic Perspective. 1st. Chapman & Hall/CRC, 2009. ISBN: 1420067184, 9781420067187 (cited on page 88).
- [66] Corinna Cortes and Vladimir Vapnik. “Support-Vector Networks”. In: Mach. Learn. 20.3 (Sept. 1995), pages 273–297. ISSN: 0885-6125. DOI: 10.1023/A:1022627411411.

- [67] Andrej Karpathy. Dòngar Classification. <http://cs231n.github.io/dòngar-classify/> (cited on pages 82, 94).
- [68] Dmytro Mishkin and Jiri Matas. “All you need is a good init”. In: CoRR abs/1511.06422 (2015). URL: <http://arxiv.org/abs/1511.06422> (cited on page 102).
- [69] Stanford University. Stanford Electronics Laboratories Et al. Adaptive “adadòng” neuron using chemical “memistors.”. 1960.
- [70] Ning Qian. “On the Momentum Term in Gradient Descent Learning Algorithms”. In: Neural Netw. 12.1 (Jan. 1999), pages 145–151. ISSN: 0893-6080.
- [71] Yurii Nesterov. “A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ”. In: Soviet Mathematics Doklady. Volume 27. 2. 1983, pages 372–376 (cited on pages 111, 112).
- [72] Sebastian Ruder. “An overview of gradient descent optimization algorithms”. URL: <http://arxiv.org/abs/1609.04747> (cited on page 112).
- [73] Richard S. Sutton. “Two Problems with Backpropagation and Other Steepest-Descent Learning Procedures for Networks”. In: Proceedings of the Eighth Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Erlbaum, 1986 (cited on page 112).
- [74] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. “Advances in Optimizing Recurrent Networks”. In: CoRR abs/1212.0901 (2012). URL: <http://arxiv.org/abs/1212.0901> (cited on page 112).
- [75] Ilya Sutskever. “Training Recurrent Neural Networks”. AAINS22066. PhD thesis. Toronto, Ont., Canada, Canada, 2013. ISBN: 978-0-499-22066-0 (cited on page 112).
- [76] Andrej Karpathy. Neural Networks (Part III). <http://cs231n.github.io/neural-networks-3/> (cited on pages 113, 253).
- [77] Hui Zou and Trevor Hastie. “Regularization and variable selection via the Elastic Net”. In: Journal of the Royal Statistical Society, Series B 67 (2005), pages 301–320 (cited on pages 113, 116).
- [78] <http://deeplearning.net/tutorial/gettingstarted.html#regularization> (cited on page 117)
- [79] Andrej Karpathy. Neural Networks (Part II). <http://cs231n.github.io/neural-networks-2/> (cited on page 117).
- [80] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: International Journal of Computer Vision (IJCV) 115.3

- (2015), pages 211–252. DOI: 10.1007/s11263-015-0816-y (cited on pages 45, 58, 68, 81, 277).
- [81] Kaiming He et al. “Identity Mappings in Deep Residual Networks”. In: CoRR abs/1603.05027 (2016). URL: <http://arxiv.org/abs/1603.05027> (cited on page 279).
- [82] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: CoRR abs/1512.00567 (2015). URL: <http://arxiv.org/abs/1512.00567> (cited on page 280).
- [83] François Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: CoRR abs/1610.02357 (2016).
- [84] Forrest N. Iandola et al. “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size”. In: CoRR abs/1602.07360 (2016). URL: <http://arxiv.org/abs/1602.07360> (cited on pages 192, 280, 281).
- [85] Stéfan van der Walt et al. “scikit-image: image processing in Python”. In: PeerJ 2 (June 2014), e453.
- [86] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: Journal of Machine Learning Research 12 (2011), pages 2825–2830 (cited on pages 19, 64).
- [87] Satya Mallick. Why does OpenCV use BGR color format? <http://www.learnopencv.com/why-does-opencv-use-bgr-color-format/>. 2015 (cited on page 36).
- [88] Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik. “A Training Algorithm for Optimal Margin Classifiers”. In: Proceedings of the Fifth Annual Workshop on Computational Learning Theory. COLT ’92. Pittsburgh, Pennsylvania, USA: ACM, 1992, pages 144–152
- [89] Leo Breiman. “Random Forests”. In: Mach. Learn. 45.1 (Oct. 2001), pages 5–32. ISSN: 0885-6125. DOI: 10.1023/A:1010933404324. URL: <http://dx.doi.org/10.1023/A:1010933404324> (cited on page 45).
- [90] Denny Zhou et al. “Learning with Local and Global Consistency”. In: Advances in Neural Information Processing Systems 16. Edited by S. Thrun, L. K. Saul, and P. B. Schölkopf. MIT Press, 2004, pages 321–328.
- [91] Xiaojin Zhu and Zoubin Ghahramani. Learning from Labeled and Unlabeled Data with Label Propagation. Technical report. 2002 (cited on page 48).
- [92] Antti Rasmus et al. “Semi-Supervised Learning with Ladder Network”.

In: CoRR abs/1507.02672 (2015). URL: <http://arxiv.org/abs/1507.02672> (cited on page 48).

- [93] Avrim Blum and Tom Mitchell. “Combining Labeled and Unlabeled Data with Co-training”. In: Proceedings of the Eleventh Annual Conference on Computational Learning Theory. COLT’ 98. Madison, Wisconsin, USA: ACM, 1998, pages 92–100. ISBN: 1-58113-057-0.
- [94] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR-10 and CIFAR-100. <http://www.cs.toronto.edu/~kriz/cifar.html> (cited on page 55).
- [95] <https://github.com/hromi/SMILEsmileD>. 2010 (cited on pages 55, 307).
- [96] Maria-Elena Nilsback and Andrew Zisserman. “A Visual Vocabulary for Flower Classification.” In: CVPR (2). IEEE Computer Society, 2006, pages 1447–1454.
- [97] L. Fei-Fei, R. Fergus, and Pietro Perona. “Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories”. In: 2004 (cited on page 57).
- [98] Kristen Grauman and Trevor Darrell. “The Pyramid Match Kernel: Efficient Learning with Sets of Features”. In: J. Mach. Learn. Res. 8 (May 2007), pages 725–760. ISSN: 1532-4435.
- [99] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. “Beyond Bags of Features: SpatialPyramidMatchingforRecognizingNaturalSceneCategories”. In: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition – Volume CVPR ’06. Washington, DC, USA: IEEE Computer Society, 2006, pages 2169–2178.
- [100] Hao Zhang et al. “SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition”. In: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2. CVPR ’06. Washington, DC, USA: IEEE Computer Society, 2006, pages 2126–2136.
- [101] Eran Eidinger, Roee Enbar, and Tal Hassner. “Age and Gender Estimation of Unfiltered Faces”. In: Trans. Info. For. Sec. 9.12 (Dec. 2014), pages 2170–2179.
- [102] WordNet. About WordNet. <http://wordnet.princeton.edu>. 2010 (cited on page 58).
- [103] A. Quattoni and A. Torralba. “Recognizing indoor scenes”. In: Computer Vision and Pattern Recognition, IEEE Computer Society Conference on. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pages 413–420 (cited on page 60).

- [104] Jonathan Krause et al. “3D Object Representations for Fine-Grained Categorization”. In: 4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13). Sydney, Australia, 2013 (cited on page 60).
- [105] Jon Louis Bentley. “Multidimensional Binary Search Trees Used for Associative Searching”. In: Commun. ACM 18.9 (Sept. 1975), pages 509–517.
- [106] Sanjoy Dasgupta. “Experiments with Random Projection”. In: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence. UAI ’00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pages 143–151.
- [107] Ella Bingham and Heikki Mannila. “Random Projection in Dimensionality Reduction: Applications to Image and Text Data”. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD ’01. San Francisco, California: ACM, 2001, pages 245–250.
- [108] Sanjoy Dasgupta and Anupam Gupta. “An Elementary Proof of a Theorem of Johnson and Lindenstrauss”. In: Random Struct. Algorithms 22.1 (Jan. 2003), pages 60–65.
- [109] Pedro Domingos. “A Few Useful Things to Know About Machine Learning”. In: Commun. ACM 55.10 (Oct. 2012), pages 78–87. ISSN: 0001-0782.
- [110] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: CoRR abs/1412.6980 (2014). URL: <http://arxiv.org/abs/1412.6980> (cited on pages 164, 311).
- [111] Paul Viola and Michael Jones. “Rapid object detection using a boosted cascade of simple features”. In: 2001, pages 511–518 (cited on page 313).



CHÍNH SÁCH CHẤT LƯỢNG

*Không ngừng nâng cao chất lượng dạy, học, nghiên cứu khoa học
và phục vụ cộng đồng nhằm mang đến cho người học những điều kiện tốt nhất
để phát triển toàn diện các năng lực đáp ứng nhu cầu phát triển và hội nhập quốc tế.*

ISBN: 978-604-73-7082-5
9 786047 370825
NXB ĐHQG HCM