

O'REILLY®

ПОЛНОЦВЕТНОЕ
ИЗДАНИЕ

Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow

КОНЦЕПЦИИ, ИНСТРУМЕНТЫ И ТЕХНИКИ
ДЛЯ СОЗДАНИЯ ИНТЕЛЛЕКТУАЛЬНЫХ
СИСТЕМ



Орельен Жерон

Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow

*Концепции, инструменты и техники
для создания интеллектуальных систем*

Hands-On Machine Learning with Scikit-Learn and TensorFlow

*Concepts, Tools, and Techniques to Build
Intelligent Systems*

Aurélien Géron

Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow

*Концепции, инструменты и техники
для создания интеллектуальных систем*

Орельен Жерон



Москва · Санкт-Петербург · Киев
2018

ББК 32.973.26-018.2.75

Ж61

УДК 681.3.07

Компьютерное издательство “Диалектика”

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция Ю.Н. Артеменко

По общим вопросам обращайтесь в издательство “Диалектика”
по адресу: info@dialektika.com, http://www.dialektika.com

Жерон, Орельен.

Ж61 Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow: концепции, инструменты и техники для создания интеллектуальных систем. : Пер. с англ. — СПб. : ООО “Альфа-книга”, 2018. — 688 с. : ил. — Парал. тит. англ.

ISBN 978-5-9500296-2-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly & Associates.

Authorized Russian translation of the English edition of *Hands-On Machine Learning with Scikit-Learn and TensorFlow* (ISBN 978-1-491-96229-9) © 2017 Aurélien Géron. All rights reserved.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Орельен Жерон

**Прикладное машинное обучение с помощью
Scikit-Learn и TensorFlow:
концепции, инструменты и техники для создания
интеллектуальных систем**

Верстка *Т.Н. Артеменко*

Художественный редактор *В.Г. Павлютин*

Подписано в печать 29.01.2018. Формат 70×100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 55,47. Уч.-изд. л. 53,5.

Тираж 500 экз. Заказ № 0000.

ООО “Альфа-книга”, 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит А, пом. 848

ISBN 978-5-9500296-2-2 (рус.)

© Компьютерное издательство “Диалектика”, 2018,
перевод, оформление, макетирование

ISBN 978-1-491-96229-9 (англ.)

© 2017 Aurélien Géron

Оглавление

Часть I. Основы машинного обучения	25
Глава 1. Введение в машинное обучение	27
Глава 2. Полный проект машинного обучения	63
Глава 3. Классификация	121
Глава 4. Обучение моделей	153
Глава 5. Методы опорных векторов	199
Глава 6. Деревья принятия решений	223
Глава 7. Ансамблевое обучение и случайные леса	239
Глава 8. Понижение размерности	267
Часть II. Нейронные сети и глубокое обучение	291
Глава 9. Подготовка к работе с TensorFlow	293
Глава 10. Введение в искусственные нейронные сети	323
Глава 11. Обучение глубоких нейронных сетей	351
Глава 12. Использование TensorFlow для распределения вычислений между устройствами и серверами	401
Глава 13. Сверточные нейронные сети	453
Глава 14. Рекуррентные нейронные сети	483
Глава 15. Автокодировщики	523
Глава 16. Обучение с подкреплением	553
Приложение А. Решения упражнений	593
Приложение Б. Контрольный перечень для проекта машинного обучения	633
Приложение В. Двойственная задача SVM	640
Приложение Г. Автоматическое дифференцирование	643
Приложение Д. Другие популярные архитектуры искусственных нейронных сетей	651
Предметный указатель	662

Содержание

Об авторе	15
Благодарности	15
Предисловие	17
Цунами машинного обучения	17
Машинное обучение в ваших проектах	18
Цель и подход	18
Предварительные требования	19
Дорожная карта	20
Другие ресурсы	21
Типографские соглашения, используемые в книге	22
Использование примеров кода	23
Ждем ваших отзывов!	24
Часть I. Основы машинного обучения	25
Глава 1. Введение в машинное обучение	27
Что такое машинное обучение?	28
Для чего используют машинное обучение?	29
Типы систем машинного обучения	32
Обучение с учителем и без учителя	33
Пакетное и динамическое обучение	40
Обучение на основе образцов или на основе моделей	43
Основные проблемы машинного обучения	50
Недостаточный размер обучающих данных	50
Нерепрезентативные обучающие данные	52
Данные плохого качества	54
Несущественные признаки	54
Переобучение обучающих данных	55
Недообучение обучающих данных	57
Шаг назад	58
Испытание и проверка	59
Упражнения	61
Глава 2. Полный проект машинного обучения	63
Работа с реальными данными	63
Выяснение общей картины	65
Постановка задачи	65
Выбор критерия качества работы	68
Проверка допущений	71

Получение данных	71
Создание рабочей области	71
Загрузка данных	75
Беглый взгляд на структуру данных	77
Создание испытательного набора	81
Обнаружение и визуализация данных для понимания их сущности	87
Визуализация географических данных	87
Поиск связей	90
Экспериментирование с комбинациями атрибутов	93
Подготовка данных для алгоритмов машинного обучения	94
Очистка данных	95
Обработка текстовых и категориальных атрибутов	98
Специальные трансформаторы	102
Масштабирование признаков	103
Конвейеры трансформации	104
Выбор и обучение модели	106
Обучение и оценка с помощью обучающего набора	107
Более подходящая оценка с использованием перекрестной проверки	108
Точная настройка модели	111
Решетчатый поиск	111
Рандомизированный поиск	114
Ансамблевые методы	115
Анализ лучших моделей и их ошибок	115
Оценка системы с помощью испытательного набора	116
Запуск, наблюдение и сопровождение системы	117
Пробуйте!	118
Упражнения	118
Глава 3. Классификация	121
MNIST	121
Обучение двоичного классификатора	124
Показатели производительности	125
Измерение правильности с использованием перекрестной проверки	125
Матрица неточностей	127
Точность и полнота	129
Соотношение точность/полнота	131
Кривая ROC	135
Многоклассовая классификация	139
Анализ ошибок	142
Многозначная классификация	146
Многовыходовая классификация	148
Упражнения	150

Глава 4. Обучение моделей	153
Линейная регрессия	154
Нормальное уравнение	156
Вычислительная сложность	159
Градиентный спуск	160
Пакетный градиентный спуск	163
Стохастический градиентный спуск	167
Мини-пакетный градиентный спуск	170
Полиномиальная регрессия	172
Кривые обучения	174
Регуляризованные линейные модели	179
Гребневая регрессия	179
Лассо-регрессия	182
Эластичная сеть	184
Раннее прекращение	185
Логистическая регрессия	187
Оценивание вероятностей	187
Обучение и функция издержек	189
Границы решений	190
Многoperеменная логистическая регрессия	193
Упражнения	197
Глава 5. Методы опорных векторов	199
Линейная классификация SVM	199
Классификация с мягким зазором	200
Нелинейная классификация SVM	203
Полиномиальное ядро	204
Добавление признаков близости	206
Гауссово ядро RBF	207
Вычислительная сложность	209
Регрессия SVM	210
Внутренняя кухня	212
Функция решения и прогнозы	212
Цель обучения	213
Квадратичное программирование	215
Двойственная задача	216
Параметрически редуцированные методы SVM	217
Динамические методы SVM	220
Упражнения	222
Глава 6. Деревья принятия решений	223
Обучение и визуализация дерева принятия решений	223
Вырабатывание прогнозов	225

Оценивание вероятностей классов	227
Алгоритм обучения CART	228
Вычислительная сложность	229
Загрязненность Джини или энтропия?	230
Гиперпараметры регуляризации	230
Регрессия	232
Неустойчивость	235
Упражнения	236
Глава 7. Ансамблевое обучение и случайные леса	239
Классификаторы с голосованием	240
Бэггинг и вставка	243
Бэггинг и вставка в Scikit-Learn	245
Оценка на неиспользуемых образцах	246
Методы случайных участков и случайных подпространств	248
Случайные леса	248
Особо случайные деревья	250
Значимость признаков	250
Бустинг	251
AdaBoost	252
Градиентный бустинг	256
Стекинг	261
Упражнения	264
Глава 8. Понижение размерности	267
“Проклятие размерности”	268
Основные подходы к понижению размерности	270
Проекция	270
Обучение на основе многообразий	272
PCA	274
Предохранение дисперсии	274
Главные компоненты	275
Проецирование до d измерений	277
Использование Scikit-Learn	277
Коэффициент объясненной дисперсии	278
Выбор правильного количества измерений	278
Алгоритм PCA для сжатия	279
Инкрементный анализ главных компонентов	281
Рандомизированный анализ главных компонентов	282
Ядерный анализ главных компонентов	282
Выбор ядра и подстройка гиперпараметров	283
LLE	286
Другие приемы понижения размерности	288
Упражнения	289

Часть II. Нейронные сети и глубокое обучение	291
Глава 9. Подготовка к работе с TensorFlow	293
Установка	297
Создание первого графа и его прогон в сеансе	297
Управление графами	299
Жизненный цикл значения узла	300
Линейная регрессия с помощью TensorFlow	301
Реализация градиентного спуска	302
Расчет градиентов вручную	303
Использование autodiff	304
Использование оптимизатора	306
Передача данных алгоритму обучения	306
Сохранение и восстановление моделей	308
Визуализация графа и кривых обучения с использованием TensorBoard	309
Пространства имен	313
Модульность	314
Совместное использование переменных	316
Упражнения	320
Глава 10. Введение в искусственные нейронные сети	323
От биологических нейронов к искусственным нейронам	324
Биологические нейроны	325
Логические вычисления с помощью нейронов	326
Персептрон	328
Многослойный персептрон и обратная связь	333
Обучение многослойного персептрона с помощью высокоуровневого API-интерфейса TensorFlow	336
Обучение глубокой нейронной сети с использованием только TensorFlow	338
Стадия построения	338
Стадия выполнения	343
Использование нейронной сети	344
Точная настройка гиперпараметров нейронной сети	345
Количество скрытых слоев	346
Количество нейронов на скрытый слой	347
Функции активации	348
Упражнения	349
Глава 11. Обучение глубоких нейронных сетей	351
Проблемы исчезновения и взрывного роста градиентов	352
Инициализация Ксавье и Хе	353
Ненасыщаемые функции активации	356
Пакетная нормализация	359
Отсечение градиентов	365

Повторное использование заранее обученных слоев	366
Повторное использование модели TensorFlow	366
Повторное использование моделей из других фреймворков	370
Замораживание низкоуровневых слоев	371
Кеширование замороженных слоев	372
Подстройка, отбрасывание или замена слоев верхних уровней	373
Зоопарки моделей	373
Предварительное обучение без учителя	374
Предварительное обучение на вспомогательной задаче	375
Более быстрые оптимизаторы	376
Моментная оптимизация	377
Ускоренный градиент Нестерова	378
AdaGrad	380
RMSProp	382
Оптимизация Adam	382
Планирование скорости обучения	385
Избегание переобучения посредством регуляризации	388
Раннее прекращение	388
Регуляризация ℓ_1 и ℓ_2	389
Отключение	390
Регуляризация на основе max-нормы	393
Дополнение данных	395
Практические рекомендации	397
Упражнения	398

Глава 12. Использование TensorFlow для распределения вычислений между устройствами и серверами 401

Множество устройств на единственной машине	402
Установка	403
Управление оперативной памятью графического процессора	406
Размещение операций на устройствах	408
Параллельное выполнение	412
Зависимости управления	414
Множество устройств на множестве серверов	415
Открытие сеанса	417
Службы мастера и исполнителя	418
Прикрепление операций между задачами	418
Фрагментация переменных среди множества серверов параметров	419
Разделение состояния между сеансами с использованием контейнеров ресурсов	421
Асинхронное взаимодействие с использованием очередей TensorFlow	423
Загрузка данных напрямую из графа	429

Распараллеливание нейронных сетей в кластере TensorFlow	438
Одна нейронная сеть на устройство	439
Репликация внутри графа или между графами	440
Параллелизм модели	443
Параллелизм данных	445
Упражнения	451
Глава 13. Сверточные нейронные сети	453
Строение зрительной коры головного мозга	454
Сверточный слой	456
Фильтры	458
Наложение множества карт признаков	459
Реализация с помощью TensorFlow	462
Требования к памяти	464
Объединяющий слой	466
Архитектуры сверточных нейронных сетей	467
LeNet-5	469
AlexNet	470
GoogLeNet	472
ResNet	476
Упражнения	482
Глава 14. Рекуррентные нейронные сети	485
Рекуррентные нейроны	486
Ячейки памяти	489
Входные и выходные последовательности	489
Базовые рекуррентные нейронные сети в TensorFlow	491
Статическое развертывание во времени	492
Динамическое развертывание во времени	495
Обработка входных последовательностей переменной длины	495
Обработка выходных последовательностей переменной длины	497
Обучение рекуррентных нейронных сетей	497
Обучение классификатора последовательностей	498
Обучение для прогнозирования временных рядов	500
Креативная рекуррентная нейронная сеть	505
Глубокие рекуррентные нейронные сети	506
Распределение глубокой рекуррентной нейронной сети между множеством графических процессоров	507
Применение отключения	508
Трудность обучения в течение многих временных шагов	510
Ячейка LSTM	511
Смотровые связи	514
Ячейка GRU	514

Обработка естественного языка	516
Векторные представления слов	516
Сеть “кодировщик–декодировщик” для машинного перевода	519
Упражнения	523
Глава 15. Автокодировщики	525
Эффективные представления данных	526
Выполнение анализа главных компонентов с помощью понижающего линейного автокодировщика	528
Многослойные автокодировщики	529
Реализация с помощью TensorFlow	530
Связывание весов	532
Обучение по одному автокодировщику за раз	533
Визуализация реконструкций	536
Визуализация признаков	537
Предварительное обучение без учителя с использованием многослойных автокодировщиков	538
Шумоподавляющие автокодировщики	540
Реализация с помощью TensorFlow	541
Разреженные автокодировщики	543
Реализация с помощью TensorFlow	544
Вариационные автокодировщики	545
Генерирование цифр	549
Другие автокодировщики	550
Упражнения	552
Глава 16. Обучение с подкреплением	555
Обучение для оптимизации наград	556
Поиск политики	558
Введение в OpenAI Gym	560
Политики в форме нейронных сетей	564
Оценка действий: проблема присваивания коэффициентов доверия	567
Градиенты политики	568
Марковские процессы принятия решений	574
Обучение методом временных разностей и Q-обучение	579
Политики исследования	581
Приближенное Q-обучение и глубокое Q-обучение	582
Обучение играть в игру Ms. Pac-Man с использованием алгоритма сети DQN	584
Упражнения	594
Спасибо!	595
Приложение А. Решения упражнений	596
Глава 1. Введение в машинное обучение	596
Глава 2. Полный проект машинного обучения	599

Глава 3. Классификация	599
Глава 4. Обучение моделей	599
Глава 5. Методы опорных векторов	602
Глава 6. Деревья принятия решений	604
Глава 7. Ансамблевое обучение и случайные леса	605
Глава 8. Понижение размерности	607
Глава 9. Подготовка к работе с TensorFlow	610
Глава 10. Введение в искусственные нейронные сети	613
Глава 11. Обучение глубоких нейронных сетей	616
Глава 12. Использование TensorFlow для распределения вычислений между устройствами и серверами	618
Глава 13. Сверточные нейронные сети	621
Глава 14. Рекуррентные нейронные сети	624
Глава 15. Автокодировщики	626
Глава 16. Обучение с подкреплением	629
Приложение Б. Контрольный перечень для проекта машинного обучения	633
Постановка задачи и выяснение общей картины	633
Получение данных	634
Исследование данных	635
Подготовка данных	636
Составление окончательного списка перспективных моделей	637
Точная настройка системы	638
Представление своего решения	639
Запуск!	639
Приложение В. Двойственная задача SVM	640
Приложение Г. Автоматическое дифференцирование	643
Ручное дифференцирование	643
Символическое дифференцирование	644
Численное дифференцирование	645
Автоматическое дифференцирование в прямом режиме	647
Автоматическое дифференцирование в обратном режиме	648
Приложение Д. Другие популярные архитектуры искусственных нейронных сетей	651
Сети Хопфилда	651
Машины Больцмана	653
Ограниченные машины Больцмана	655
Глубокие сети доверия	656
Самоорганизующиеся карты	659
Предметный указатель	662

Орельен Жерон — консультант по машинному обучению. Бывший работник компании Google, он руководил командой классификации видеороликов YouTube с 2013 по 2016 год. С 2002 по 2012 год он также был основателем и руководителем технического отдела в компании Wifirst, ведущего поставщика услуг беспроводного доступа к Интернету во Франции, а в 2001 году — основателем и руководителем технического отдела в фирме Polyconseil, которая сейчас управляет сервисом совместного пользования электромобилями Autolib'.

Благодарности

Я хотел бы поблагодарить своих коллег из Google, особенно команду классификации видеороликов YouTube, за то, что они научили меня настолько многому в области машинного обучения. Без них я никогда бы не начал такой проект. Выражаю специальную благодарность своим персональным гуру по машинному обучению: Клемену Курбе, Жюльену Дюбуа, Матиасу Кенде, Дэниелу Китачевски, Джеймсу Пэку, Александру Паку, Аношу Раджу, Витору Сессаку, Виктору Томчаку, Ингрид фон Глен, Ричу Уошингтону и всем из YouTube Paris.

Я чрезвычайно признателен всем замечательным людям, находящим время в своей занятой жизни для критического анализа моей книги до мельчайших деталей. Спасибо Питу Уордену за ответы на все мои вопросы по TensorFlow, рецензирование части II, предоставление многих интересных идей и, конечно же, за участие в основной команде TensorFlow. Вы непременно должны заглянуть в его блог (<https://petewarden.com/>)! Большое спасибо Лукасу Бивальду за его очень серьезный пересмотр части II: он приложил все старания, проверил весь код (и выявил несколько ошибок), дал множество замечательных советов, а его энтузиазм был заразительным. Вы просто обязаны посетить его блог (<https://lukasbiewald.com/>) и взглянуть на его классных роботов (<https://goo.gl/Eu5u28>)! Благодарю Джастина Фрэнсиса за то, что он также очень тщательно проанализировал часть II, отловил ошибки и предоставил прекрасные идеи, в частности по главе 16. Следите за его записями, посвященными TensorFlow (<https://goo.gl/28ve8z>)!

Огромное спасибо Дэвиду Анджиевски, который пересмотрел часть I и дал невероятно полезный отзыв, указав на непонятные разделы и предложив способы их улучшения. Зайдите на его веб-сайт (<http://www.david-andrzejewski.com/>)! Благодарю Грегуара Мениля, который проанализировал часть II и внес очень интересную практическую рекомендацию по обучению нейронных сетей. Также спасибо Эдди Хонгу, Салиму Семауну, Кариму Матра, Ингрид фон Глен, Иэну Смесу и Венсану Гильбо за рецензирование части I и множество полезных советов. И я благодарен своему тестю Мишелю Тесье, бывшему учителю математики, а теперь замечательному переводчику произведений Антона Чехова, за то, что он помог уладить дело с некоторыми математическими выкладками в книге, и проанализировал тетрадь Jupyter по линейной алгебре.

Разумеется, гигантское спасибо моему дорогому брату Сильвену, который пересматривал каждую главу, проверял каждую строчку кода, давал отзывы буквально на каждый раздел и поддерживал меня от первой строки до последней. Люблю тебя, братишка!

Благодарю фантастических сотрудников издательства O'Reilly, в особенности Николь Таш, обеспечивавшую проницательные отзывы, всегда радостную, ободряющую и любезную. Спасибо также Мари Богуро, Бену Лорика, Майку Лукидесу и Лорель Рюма за то, что они верили в проект и помогали мне определить его рамки. Благодарю Мета Хакера и всех из команды Atlas за ответы на мои технические вопросы по форматированию, AsciiDoc и LaTeX, а также Рэйчел Монаган, Ника Адамса и всех из производственной команды за окончательную проверку и сотни поправок.

И последнее, но не менее важное: я бесконечно признателен моей любимой жене Эмманюэль и трем замечательным детям, Александру, Реми и Габриэль, за то, что они поддерживали меня в интенсивной работе над этой книгой, задавали много вопросов (кто сказал, что вы не сумеете обучить нейронным сетям семилетку?) и даже приносили мне печенье и кофе. О чем еще можно было мечтать?

Цунами машинного обучения

В 2006 году Джеффри Хинтон и др. опубликовали статью¹, в которой было показано, как обучать глубокую нейронную сеть, способную распознавать рукописные цифры с передовой точностью (>98%). Они назвали такой прием “глубоким обучением” (“Deep Learning”). Обучение глубокой нейронной сети в то время считалось невозможным² и с 1990-х годов большинство исследователей отказалось от этой затеи. Указанная статья возродила интерес научной общественности и вскоре многочисленные новые статьи продемонстрировали, что глубокое обучение было не только возможным, но способным к умопомрачительным достижениям. Никакой другой прием машинного обучения (Machine Learning — ML) не мог даже приблизиться к таким достижениям (при помощи гигантской вычислительной мощности и огромных объемов данных). Этот энтузиазм быстро распространился на многие другие области машинного обучения.

Промелькнуло 10 лет и машинное обучение завоевало отрасль: теперь оно лежит в основе большей части магии сегодняшних высокотехнологичных продуктов, упорядочивая результаты ваших поисковых запросов, приводя в действие распознавание речи в вашем смартфоне и рекомендуя видеоролики. Вдобавок оно еще и одержало победу над чемпионом мира по игре го. Не успеете оглянуться, и машинное обучение начнет вести ваш автомобиль.

¹ Статья доступна на домашней странице Хинтона, находящейся по адресу <http://www.cs.toronto.edu/~hinton/>.

² Несмотря на тот факт, что глубокие сверточные нейронные сети Яна Лекуна хорошо работали при распознавании изображений с 1990-годов, они были не настолько универсальны.

Машинное обучение в ваших проектах

Итак, естественно вы потрясены машинным обучением и желали бы присоединиться к компании!

Возможно, вы хотите дать своему домашнему роботу собственный мозг? Добиться, чтобы он распознавал лица? Научить его ходить?

А может быть у вашей компании имеется масса данных (журналы пользователей, финансовые данные, производственные данные, данные машинных датчиков, статистические данные от линии оперативной поддержки, отчеты по персоналу и т.д.) и, скорее всего, вы сумели бы найти там несколько скрытых жемчужин, просто зная, где искать. Ниже перечислены примеры того, что можно было бы делать:

- сегментировать заказчиков и установить лучшую маркетинговую стратегию для каждой группы;
- рекомендовать товары каждому клиенту на основе того, что покупают похожие клиенты;
- определять, какие транзакции, возможно, являются мошенническими;
- прогнозировать доход в следующем году;
- многое другое (<https://www.kaggle.com/wiki/DataScienceUseCases>).

Какой бы ни была причина, вы решили освоить машинное обучение и внедрить его в свои проекты. Великолепная идея!

Цель и подход

В книге предполагается, что вы почти ничего не знаете о машинном обучении. Ее цель — предоставить вам концепции, идеи и инструменты, которые необходимы для фактической реализации программ, способных *обучаться на основе данных*.

Мы рассмотрим многочисленные приемы, начиная с простейших и самых часто используемых (таких как линейная регрессия) и заканчивая рядом методов глубокого обучения, которые регулярно побеждают в состязаниях.

Вместо того чтобы реализовывать собственную миниатюрную версию каждого алгоритма, мы будем применять реальные фреймворки Python производственного уровня.

- Библиотека Scikit-Learn (<http://scikit-learn.org/>) очень проста в использовании, при этом она эффективно реализует многие алгоритмы машинного обучения, что делает ее великолепной отправной точкой для изучения машинного обучения.
- TensorFlow (<http://tensorflow.org/>) является более сложной библиотекой для распределенных численных расчетов с применением графов потоков данных. Это позволяет эффективно обучать и запускать очень большие нейронные сети, потенциально распределяя вычисления между тысячами серверов с множеством графических процессоров. Библиотека TensorFlow была создана в Google и поддерживает много крупномасштабных приложений машинного обучения. В ноябре 2015 года она стала продуктом с открытым кодом.

В книге отдается предпочтение практическому подходу, стимулируя интуитивное понимание машинного обучения через конкретные работающие примеры, поэтому теории здесь совсем немного. Хотя вы можете читать книгу, не прибегая к ноутбуку, настоятельно рекомендуется экспериментировать с примерами кода, которые доступны в виде тетрадей Jupyter по адресу <https://github.com/ageron/handson-ml>.

Предварительные требования

В книге предполагается, что вы обладаете некоторым опытом программирования на Python и знакомы с главными библиотеками Python для научных расчетов, в частности NumPy (<http://numpy.org/>), Pandas (<http://pandas.pydata.org/>) и Matplotlib (<http://matplotlib.org/>).

К тому же, если вас интересует, что происходит внутри, тогда вы должны также понимать математику на уровне колледжа (исчисление, линейную алгебру, теорию вероятностей и статистику).

Если вы пока еще не знаете язык Python, то веб-сайт <http://learnpython.org/> станет прекрасным местом, чтобы приступить к его изучению. Также неплохим ресурсом будет официальное руководство на [python.org](https://docs.python.org/3/tutorial/) (<https://docs.python.org/3/tutorial/>).

Если вы никогда не работали с Jupyter, то в главе 2 будет описан процесс его установки и основы: это замечательный инструмент, который полезно иметь в своем арсенале.

Если вы не знакомы с библиотеками Python для научных расчетов, то предоставленные тетради Jupyter включают несколько подходящих руководств. Доступно также краткое математическое руководство по линейной алгебре.

Дорожная карта

Книга содержит две части. В **части I** раскрываются перечисленные ниже темы.

- Что такое машинное обучение? Какие задачи оно пытается решать? Каковы основные категории и фундаментальные концепции систем машинного обучения?
- Главные шаги в типовом проекте машинного обучения.
- Обучение путем подгонки модели к данным.
- Оптимизация функции издержек.
- Обработка, очистка и подготовка данных.
- Выбор и конструирование признаков.
- Выбор модели и подстройка гиперпараметров с использованием перекрестной проверки.
- Главные проблемы машинного обучения, в частности недообучение и переобучение (компромисс между смещением и дисперсией).
- Понижение размерности обучающих данных в целях борьбы с “проклятием размерности”.
- Наиболее распространенные алгоритмы обучения: линейная и полиномиальная регрессия, логистическая регрессия, метод k ближайших соседей, метод опорных векторов, деревья принятия решений, случайные леса и ансамблевые методы.

В **части II** рассматриваются следующие темы.

- Что собой представляют нейронные сети? Для чего они пригодны?
- Построение и обучение нейронных сетей с применением TensorFlow.
- Самые важные архитектуры нейронных сетей: нейронные сети прямого распространения, сверточные сети, рекуррентные сети, сети с долгой краткосрочной памятью (LSTM) и автокодировщики.

- Приемы обучения глубоких нейронных сетей.
- Масштабирование нейронных сетей для гигантских наборов данных.
- Обучение с подкреплением.

Первая часть основана главным образом на использовании Scikit-Learn, в то время как вторая — на применении TensorFlow.



Не спешите с погружением: несмотря на то, что глубокое обучение, без всякого сомнения, является одной из самых захватывающих областей в машинном обучении, вы должны сначала овладеть основами. Кроме того, большинство задач могут довольно хорошо решаться с использованием простых приемов, таких как случайные леса и ансамблевые методы (обсуждаются в части I). Глубокое обучение лучше всего подходит для решения сложных задач, подобных распознаванию изображений, распознаванию речи или обработке естественного языка, при условии, что имеется достаточный объем данных, вычислительная мощность и терпение.

Другие ресурсы

Для освоения машинного обучения доступно много ресурсов. Учебные курсы по машинному обучению Эндрю Ына на Coursera (<https://www.coursera.org/learn/machine-learning/>) и учебные курсы по нейронным сетям и глубокому обучению Джеффри Хинтона (<https://www.coursera.org/course/neuralnets>) изумительны, хотя требуют значительных затрат времени (вероятно, нескольких месяцев).

Кроме того, существует много интересных веб-сайтов о машинном обучении, в числе которых, конечно же, веб-сайт с выдающимся руководством пользователя Scikit-Learn (http://scikit-learn.org/stable/user_guide.html). Вам также может понравиться веб-сайт Dataquest (<https://www.dataquest.io/>), предлагающий очень полезные интерактивные руководства, и блоги по машинному обучению вроде перечисленных на веб-сайте Quora (<http://goo.gl/GwtU3A>). Наконец, веб-сайт по глубокому обучению (<http://deeplearning.net/>) содержит хороший список ресурсов для дальнейшего изучения.

Разумеется, доступны многие другие вводные книги по машинному обучению, включая перечисленные ниже.

- Джоэл Грус, *Data Science from Scratch* (<http://shop.oreilly.com/product/0636920033400.do>) (O'Reilly). В этой книге представлены основы машинного обучения и реализации ряда основных алгоритмов с помощью только кода Python (с нуля).
- Стивен Марсленд, *Machine Learning: An Algorithmic Perspective* (Chapman and Hall). Эта книга является замечательным введением в машинное обучение. В ней подробно раскрыт широкий спектр вопросов и приведены примеры кода на Python (также с нуля, но с применением NumPy).
- Себастьян Рашка, *Python Machine Learning* (Packt Publishing). Также великолепное введение в машинное обучение, в котором задействованы библиотеки Python с открытым кодом (Pylearn 2 и Theano).
- Ясер С. Абу-Мустафа, Малик Магдон-Исмаил и Сюань-Тянь Линь, *Learning from Data* (AMLBook). За счет достаточно объемного теоретического подхода к машинному обучению эта книга обеспечивает глубокое понимание многих аспектов, в частности компромисса между смещением и дисперсией (глава 4).
- Стюарт Рассел и Питер Норвиг, *Artificial Intelligence: A Modern Approach, 3rd Edition* (Pearson) (*Искусственный интеллект. Современный подход*, 2-е изд., Диалектика). Прекрасная (и большая) книга, в которой раскрыт невероятный объем вопросов, включая машинное обучение. Она помогает уловить общую картину машинного обучения.

Наконец, замечательный способ изучения предусматривает присоединение к веб-сайтам состязаний по машинному обучению, таким как Kaggle.com (<https://www.kaggle.com/>). Это даст вам возможность приложить свои навыки к решению реальных задач, получая помощь от ряда выдающихся профессионалов в области машинного обучения.

Типографские соглашения, используемые в книге

В книге применяются следующие типографские соглашения.

Курсив

Используется для новых терминов.

Моноширинный

Применяется для URL, адресов электронной почты, имен и расширений файлов, листингов программ, а также внутри абзацев для ссылки на программные элементы наподобие имен переменных и функций, баз данных, типов данных, переменных среды и ключевых слов.

Моноширинный полужирный

Используется для представления команд или другого текста, который должен набираться пользователем буквально.

Моноширинный курсив

Применяется для текста, который должен быть заменен значениями, предоставленными пользователем, или значениями, определенными контекстом.



Этот элемент содержит совет или указание.



Этот элемент содержит замечание общего характера.



Этот элемент содержит предупреждение или предостережение.

Использование примеров кода

Добавочные материалы (примеры кода, упражнения и т.д.) доступны для загрузки по адресу <https://github.com/ageron/handson-ml>.

Настоящая книга призвана помочь вам выполнять свою работу. Обычно, если в книге предлагается пример кода, то вы можете применять его в собственных программах и документации. Вы не обязаны обращаться к нам за разрешением, если только не используете значительную долю кода. Скажем,

написание программы, в которой задействовано несколько фрагментов кода из этой книги, разрешения не требует.

Для продажи или распространения компакт-диска с примерами из книг O'Reilly разрешение обязательно. Ответ на вопрос путем цитирования данной книги и ссылки на пример кода разрешения не требует. Для встраивания значительного объема примеров кода, рассмотренных в этой книге, в документацию по вашему продукту разрешение обязательно.

Мы высоко ценим указание авторства, хотя и не требуем этого. Установление авторства обычно включает название книги, фамилию и имя автора, издательство и номер ISBN. Например: “*Hands-On Machine Learning with Scikit-Learn and TensorFlow* by Aurélien Géron (O'Reilly). Copyright 2017 Aurélien Géron, 978-1-491-96229-9”.

Если вам кажется, что способ использования вами примеров кода выходит за законные рамки или упомянутые выше разрешения, тогда свяжитесь с нами по следующему адресу электронной почты: permissions@oreilly.com.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com

WWW: <http://www.dialektika.com>

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116

в Украине: 03150, Киев, а/я 152

Основы машинного обучения

Введение в машинное обучение

Когда большинство людей слышат словосочетание “машинное обучение” (МО), они представляют себе робота: надежного дворецкого или неумолимого Терминатора в зависимости от того, кого вы спросите. Но машинное обучение — не только футуристическая фантазия, оно уже здесь. На самом деле оно десятилетиями существовало в ряде специализированных приложений, таких как программы для *оптического распознавания знаков* (*Optical Character Recognition — OCR*). Но первое приложение МО, которое действительно получило широкое распространение, улучшив жизнь сотен миллионов людей, увидело свет еще в 1990-х годах: это был *фильтр спама*. Он не претендует быть вездесущим Скайнетом, но формально квалифицируется как приложение с МО (фактически фильтр обучается настолько хорошо, что вам редко когда приходится маркировать какое-то сообщение как спам). За ним последовали сотни приложений МО, которые теперь тихо приводят в действие сотни продуктов и средств, используемых вами регулярно, от выдачи лучших рекомендаций до голосового поиска.

Где машинное обучение начинается и где заканчивается? Что в точности означает для машины *изучить* что-то? Если я загрузил копию Википедии, то действительно ли мой компьютер “обучился” чему-нибудь? Стал ли он неожиданно умнее? В этой главе вы начнете прояснять для себя, что такое машинное обучение и почему у вас может возникать желание применять его.

Затем, прежде чем приступить к исследованию континента МО, мы взглянем на карту, чтобы узнать о главных регионах и наиболее заметных ориентирах: обучение с учителем и без учителя, динамическое и пакетное обучение, обучение на основе образцов и на основе моделей. Далее мы рассмотрим рабочий поток типового проекта МО, обсудим основные проблемы, с которыми вы можете столкнуться, и покажем, как оценивать и отлаживать систему МО.

В настоящей главе вводится много фундаментальных концепций (и жаргона), которые каждый специалист по работе с данными должен знать наизусть. Это будет высокоуровневый обзор (единственная глава, не изобилующая кодом); материал довольно прост, но до того как переходить к чтению остальных глав книги, вы должны удостовериться в том, что вам здесь все совершенно ясно. Итак, запаситесь кофе и поехали!



Если вы уже знаете все основы машинного обучения, тогда можете перейти прямо к главе 2. Если такой уверенности нет, то прежде чем двигаться дальше, попробуйте дать ответы на вопросы в конце главы.

Что такое машинное обучение?

Машинное обучение представляет собой науку (и искусство) программирования компьютеров для того, чтобы они могли *обучаться на основе данных*.

Вот более общее определение.

[Машинное обучение — это] научная дисциплина, которая наделяет компьютеры способностью учиться, не будучи явно запрограммированными.

Артур Самуэль, 1959 год

А ниже приведено определение, больше ориентированное на разработку.

Говорят, что компьютерная программа обучается на основе опыта E по отношению к некоторой задаче T и некоторой оценке производительности P , если ее производительность на T , измеренная посредством P , улучшается с опытом E .

Том Митчелл, 1997 год

Скажем, ваш фильтр спама является программой МО, которая способна научиться отмечать спам на заданных примерах спам-сообщений (возможно, маркированных пользователями) и примерах нормальных почтовых сообщений (не спама). Примеры, которые система использует для обучения, называются *обучающим набором* (*training set*). Каждый обучающий пример называется *обучающим образцом* (*training instance* или *training sample*). В рассматриваемой ситуации задача T представляет собой отметку спама для новых сообщений, опыт E — *обучающие данные* (*training data*), а оценка производительности P нуждается в определении; например, вы можете применять коэффициент корректно классифицированных почтовых сооб-

щений. Эта конкретная оценка производительности называется *точностью* (*accuracy*) и часто используется в задачах классификации.

Если вы просто загрузите копию Википедии, то ваш компьютер будет содержать намного больше данных, но не станет внезапно выполнять лучше какую-нибудь задачу. Таким образом, это не машинное обучение.

Для чего используют машинное обучение?

Обдумайте, как бы вы писали фильтр спама с применением приемов традиционного программирования (рис. 1.1).

1. Сначала вы бы посмотрели, каким образом обычно выглядит спам. Вы могли бы заметить, что в теме сообщения в большом количестве встречаются определенные слова или фразы. Возможно, вы также обратили бы внимание на несколько других паттернов (шаблонов) в имени отправителя, теле сообщения и т.д.
2. Вы написали бы алгоритм обнаружения для каждого замеченного паттерна, и программа маркировала бы сообщения как спам в случае выявления некоторого числа таких паттернов.
3. Вы бы протестировали программу и повторяли шаги 1 и 2 до тех пор, пока она не стала достаточно хорошей.

Поскольку задача нетривиальна, в вашей программе с высокой вероятностью появится длинный список сложных правил, который довольно трудно сопровождать.

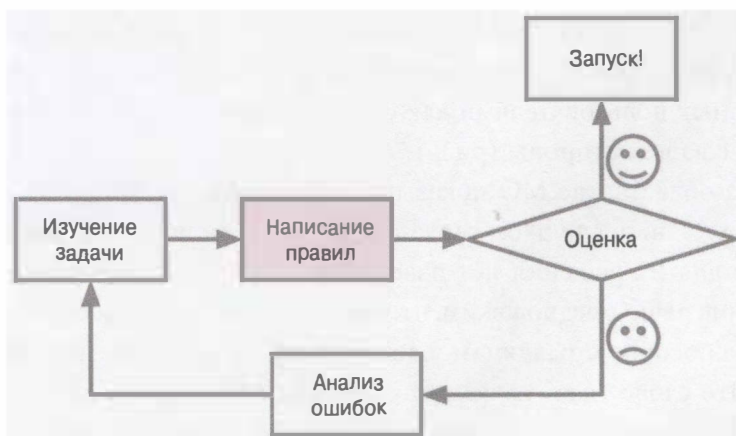


Рис. 1.1. Традиционный подход

В противоположность этому подходу фильтр спама, основанный на приемах МО, автоматически узнает, какие слова и фразы являются хорошими прогнозаторами спама, обнаруживая необычно часто встречающиеся шаблоны слов в примерах спам-сообщений по сравнению с примерами нормальных сообщений (рис. 1.2). Программа оказывается гораздо более короткой, легкой в сопровождении и, скорее всего, более точной.

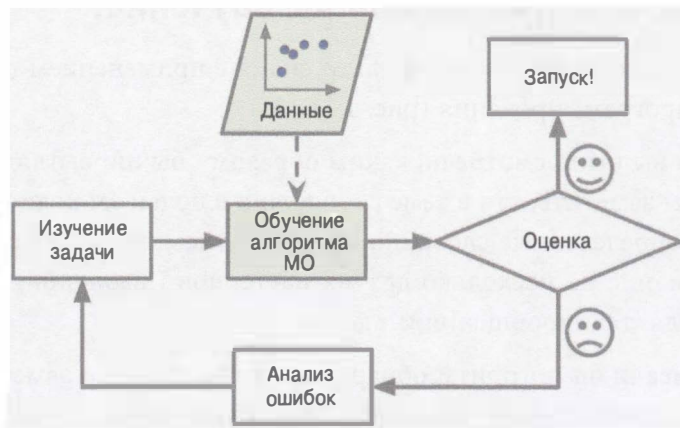


Рис. 1.2. Подход с машинным обучением

Кроме того, если спамеры обнаружат, что все их сообщения, которые содержат, к примеру, “4U”, блокируются, тогда они могут взамен писать “For U”. Фильтр спама, построенный с использованием приемов традиционного программирования, понадобится обновить, чтобы он маркировал сообщения с “For U”. Если спамеры продолжают обходить ваш фильтр спама, то вам придется постоянно писать новые правила.

Напротив, фильтр спама, основанный на приемах МО, автоматически заметит, что фраза “For U” стала необычно часто встречаться в сообщениях, маркированных пользователями как спам, и начнет маркировать их без вмешательства с вашей стороны (рис. 1.3).

Еще одна область, где МО показывает блестящие результаты, охватывает задачи, которые либо слишком сложно решать с помощью традиционных подходов, либо для их решения нет известных алгоритмов. Например, возьмем распознавание речи: предположим, что вы хотите начать с простого и написать программу, способную различать слова “один” и “два”. Вы можете обратить внимание, что слово “два” начинается с высокого звука (“Д”), а потому жестко закодировать алгоритм, который измеряет интенсивность высокого звука и применяет это для проведения различий между упомянутыми словами.

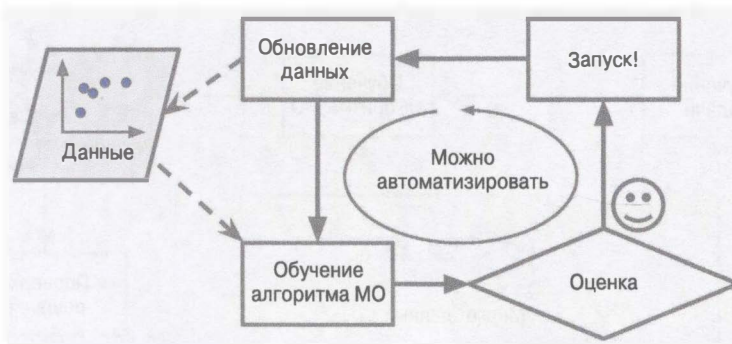


Рис. 1.3. Автоматическая адаптация к изменениям

Очевидно, такой прием не будет масштабироваться на тысячи слов, произносимых миллионами очень разных людей в шумных окружениях и на десятках языков. Лучшее решение (во всяком случае, на сегодняшний день) предусматривает написание алгоритма, который учится самостоятельно, имея много звукозаписей с примерами произношения каждого слова.

Наконец, машинное обучение способно помочь учиться людям (рис. 1.4): алгоритмы МО можно инспектировать с целью выяснения, чему они научились (хотя для некоторых алгоритмов это может быть непросто). Например, после того, как фильтр спама был обучен на достаточном объеме спам-сообщений, его легко обследовать для выявления списка слов и словосочетаний, которые он считает лучшими прогнозаторами спама. Временами удастся найти неожиданные взаимосвязи или новые тенденции, приводя к лучшему пониманию задачи.

Применение приемов МО для исследования крупных объемов данных может помочь в обнаружении паттернов, которые не были замечены сразу. Это называется *интеллектуальным* или *глубинным анализом данных* (*data mining*).

Подводя итоги, машинное обучение замечательно подходит для:

- задач, существующие решения которых требуют большого объема ручной настройки или длинных списков правил — один алгоритм МО часто способен упростить код и выполняться лучше;
- сложных задач, для которых традиционный подход вообще не предлагает хороших решений — лучшие приемы МО могут найти решение;
- изменяющихся сред — система МО способна адаптироваться к новым данным;
- получения сведений о сложных задачах и крупных объемах данных.

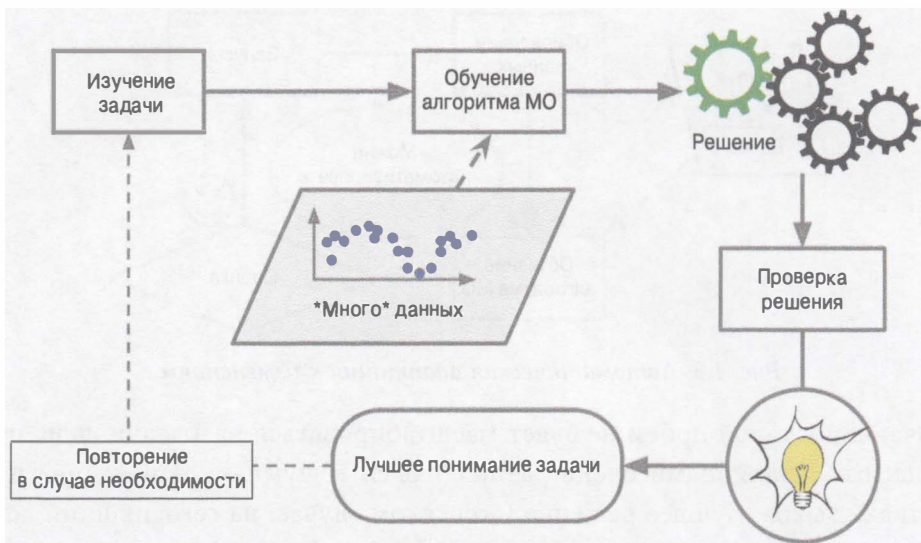


Рис. 1.4. Машинное обучение может помочь учиться людям

Типы систем машинного обучения

Существует так много разных типов систем машинного обучения, что их удобно сгруппировать в обширные категории на основе следующих признаков:

- обучаются ли они с человеческим контролем (обучение с учителем, обучение без учителя, частичное обучение (semisupervised learning) и обучение с подкреплением (Reinforcement Learning));
- могут ли они обучаться постепенно на лету (динамическое или пакетное обучение);
- работают ли они, просто сравнивая новые точки данных с известными точками данных, или взамен обнаруживают паттерны в обучающих данных и строят прогнозирующую модель подобно тому, как поступают ученые (обучение на основе образцов или на основе моделей).

Перечисленные критерии не являются взаимоисключающими; вы можете комбинировать их любым желаемым образом. Например, современный фильтр спама способен обучаться на лету, используя модель глубокой нейронной сети, которая обучена с применением примеров спам-сообщений и нормальных сообщений; это делает его динамической, основанной на моделях системой обучения с учителем.

Давайте более подробно рассмотрим каждый из указанных выше критериев.

Обучение с учителем и без учителя

Системы МО могут быть классифицированы согласно объему и типу контроля, которым они подвергаются во время обучения. Есть четыре главных категории: обучение с учителем, обучение без учителя, частичное обучение и обучение с подкреплением.

Обучение с учителем

При *обучении с учителем* обучающие данные, поставляемые вами алгоритму, включают желательные решения, называемые *метками* (*label*), как показано на рис. 1.5.



Рис. 1.5. Помеченный обучающий набор для обучения с учителем (пример с классификацией спама)

Типичной задачей обучения с учителем является *классификация*. Фильтр спама служит хорошим примером классификации: он обучается на многих примерах сообщений с их *классом* (спам или не спам), а потому должен знать, каким образом классифицировать новые сообщения.

Другая типичная задача — прогнозирование *целевого* числового значения, такого как цена автомобиля, располагая набором *характеристик* или *признаков* (пробег, возраст, марка и т.д.), которые называются *прогнозаторами* (*predictor*). Задачу подобного рода именуют *регрессией*¹ (рис. 1.6). Чтобы обучить систему, вам необходимо предоставить ей много примеров автомобилей, включая их прогнозаторы и метки (т.е. цены).

¹ Забавный факт: это необычно звучащее название является термином из области статистики, который ввел Фрэнсис Гальтон (Голтон), когда исследовал явление, что дети высоких людей обычно оказываются ниже своих родителей. Поскольку дети были ниже, он назвал данный факт *возвращением (регрессией) к среднему*. Затем такое название было применено к методам, которые он использовал для анализа взаимосвязей между переменными.



Рис. 1.6. Регрессия



В машинном обучении *атрибут* представляет собой тип данных (например, “Пробег”), тогда как *признак* в зависимости от контекста имеет несколько смыслов, но в большинстве случаев подразумевает атрибут плюс его значение (скажем, “Пробег = 15 000”). Тем не менее, многие люди используют слова *атрибут* и *признак* взаимозаменяемо.

Обратите внимание, что некоторые алгоритмы регрессии могут применяться также для классификации и наоборот. Например, *логистическая регрессия* (*logistic regression*) обычно используется для классификации, т.к. она способна производить значение, которое соответствует вероятности принадлежности к заданному классу (скажем, спам с вероятностью 20%).

Ниже перечислены некоторые из самых важных алгоритмов обучения с учителем (раскрываемые в настоящей книге):

- к ближайших соседей (k-nearest neighbors)
- линейная регрессия (linear regression)
- логистическая регрессия (logistic regression)
- метод опорных векторов (Support Vector Machine — SVM)
- деревья принятия решений (decision tree) и случайные леса (random forest)
- нейронные сети (neural network)².

² Некоторые архитектуры нейронных сетей могут быть без учителя, такие как автокодировщики (autoencoder) и ограниченные машины Больцмана (restricted Boltzmann machine). Они также могут быть частичными, как в глубоких сетях доверия (deep belief network) и предварительном обучении без учителя.

Обучение без учителя

При *обучении без учителя*, как вы могли догадаться, обучающие данные не помечены (рис. 1.7). Система пытается обучаться без учителя.

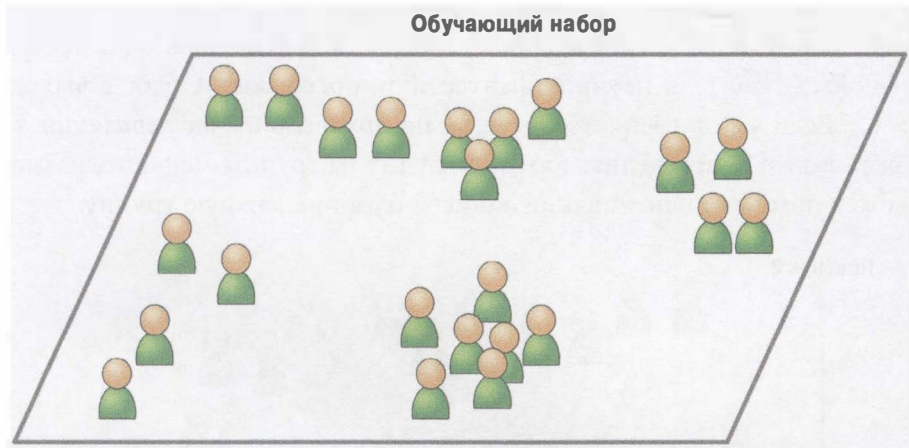


Рис. 1.7. Непомеченный обучающий набор для обучения без учителя

Далее приведен список наиболее важных алгоритмов обучения без учителя (мы рассмотрим понижение размерности (dimensionality reduction) в главе 8):

- Кластеризация
 - k-средние (k-means)
 - иерархический кластерный анализ (Hierarchical Cluster Analysis — HCA)
 - максимизация ожиданий (expectation maximization)
- Визуализация и понижение размерности
 - анализ главных компонент (Principal Component Analysis — PCA)
 - ядерный анализ главных компонент (kernel PCA)
 - локальное линейное вложение (Locally-Linear Embedding — LLE)
 - стохастическое вложение соседей с t-распределением (t-distributed Stochastic Neighbor Embedding — t-SNE)
- Обучение ассоциативным правилам (association rule learning)
 - Apriori
 - Eclat

Например, предположим, что вы имеете много данных о посетителях своего блога. Вы можете запустить алгоритм кластеризации, чтобы попытаться выявить группы похожих посетителей (рис. 1.8). Алгоритму совершенно ни к чему сообщать, к какой группе принадлежит посетитель: он найдет такие связи без вашей помощи. Скажем, алгоритм мог бы заметить, что 40% посетителей — мужчины, которые любят комиксы и читают ваш блог вечерами, 20% — юные любители научной фантастики, посещающие блог в выходные дни, и т.д. Если вы применяете алгоритм *иерархической кластеризации*, тогда он может также подразделить каждую группу на группы меньших размеров. Это может помочь в нацеливании записей блога на каждую группу.

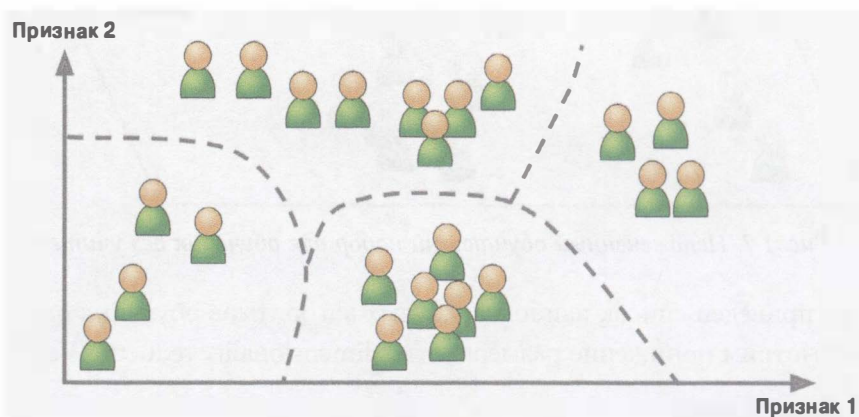


Рис. 1.8. Кластеризация

Алгоритмы *визуализации* тоже являются хорошими примерами алгоритмов обучения без учителя: вы обеспечиваете их большим объемом сложных и непомеченных данных, а они выводят двухмерное или трехмерное представление данных, которое легко вычерчивать (рис. 1.9). Такие алгоритмы стараются сохранить столько структуры, сколько могут (например, пытаются не допустить наложения при визуализации отдельных кластеров во входном пространстве), поэтому вы можете понять, как данные организованы, и возможно идентифицировать непредвиденные паттерны.

Связанной задачей является *понижение размерности*, цель которой — упростить данные без потери слишком многой информации. Один из способов предусматривает слияние нескольких связанных признаков в один. Например, пробег автомобиля может находиться в тесной связи с его возрастом, так что алгоритм понижения размерности объединит их в один признак, который представляет степень износа автомобиля. Это называется *выделением признаков (feature extraction)*.

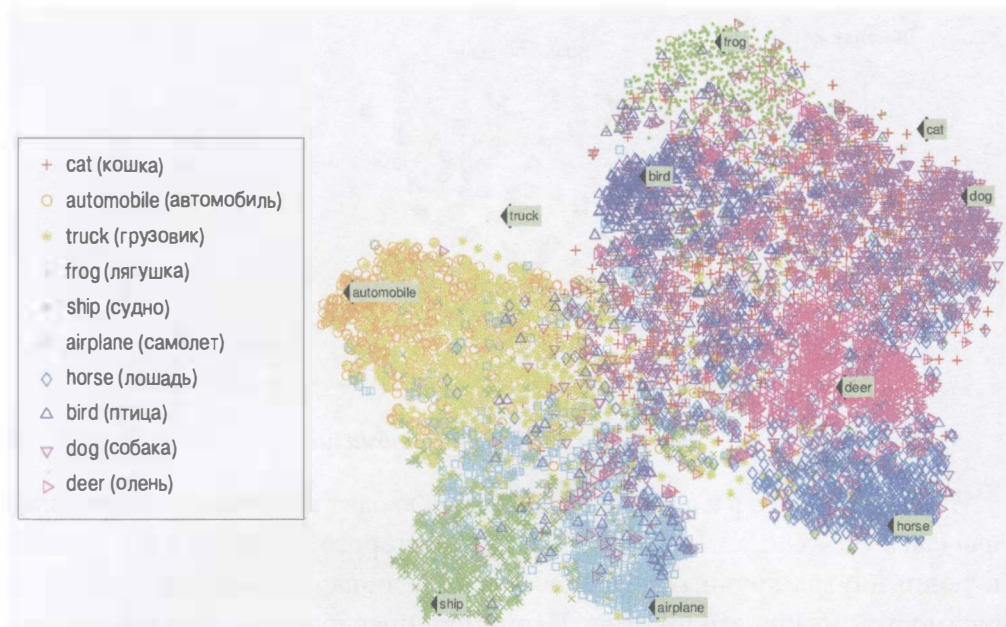


Рис. 1.9. Пример визуализации t -SNE, выделяющей семантические кластеры³



Часто имеет смысл попытаться сократить размерность обучающих данных, используя алгоритм понижения размерности, до их передачи в другой алгоритм МО (такой как алгоритм обучения с учителем). В итоге другой алгоритм МО станет намного быстрее, данные будут занимать меньше места на диске и в памяти, а в ряде случаев он может также эффективнее выполняться.

Еще одной важной задачей обучения без учителя является **обнаружение аномалий** (*anomaly detection*) — например, выявление необычных транзакций на кредитных картах в целях предотвращения мошенничества, отлавливание производственных дефектов или автоматическое удаление выбросов из набора данных перед его передачей другому алгоритму обучения. Система обучалась на нормальных образцах, и когда она видит новый образец, то может сообщить, выглядит он как нормальный или вероятно представляет собой аномалию (рис. 1.10).

³ Обратите внимание на то, что животные довольно хорошо отделены от транспортных средств, лошади близки к оленям, но далеки от птиц, и т.д. Рисунок воспроизводится с разрешения из работы “T-SNE visualization of the semantic word space” (“Визуализация t -SNE семантического пространства слов”) Ричарда Сокера, Милинда Ганджу, Кристофера Маннинга и Эндрю Ына (2013 год).

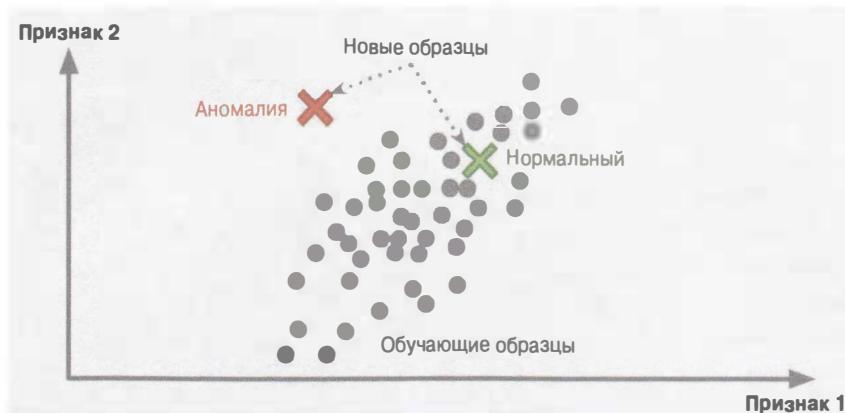


Рис. 1.10. Обнаружение аномалий

Наконец, в число распространенных задач входит *обучение ассоциативным правилам* (*association rule learning*), цель которого заключается в проникновении внутрь крупных объемов данных и обнаружении интересных зависимостей между атрибутами. Например, предположим, что вы владеете супермаркетом. Запуск ассоциативного правила на журналах продаж может выявить, что люди, покупающие соус для барбекю и картофельные чипсы, также склонны приобретать стейк. Таким образом, может возникнуть желание разместить указанные товары ближе друг к другу.

Частичное обучение

Некоторые алгоритмы способны работать с частично помеченными обучающими данными, в состав которых обычно входит много непомеченных данных и чуть-чуть помеченных. Процесс называется *частичным обучением* (рис. 1.11).

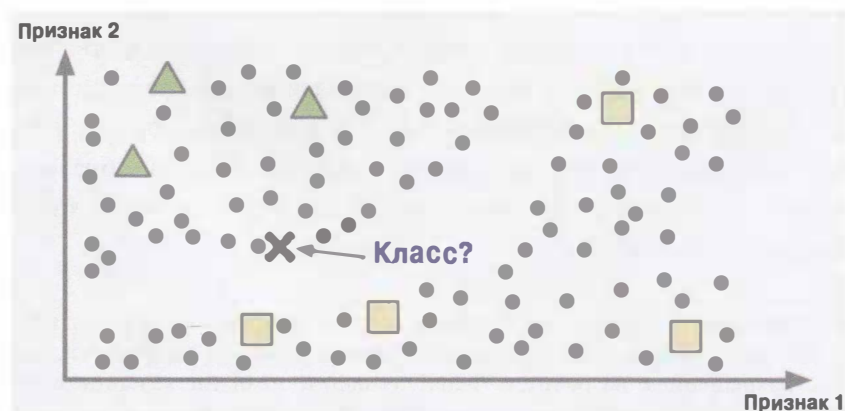


Рис. 1.11. Частичное обучение

Хорошими примерами могут быть некоторые службы для размещения фотографий наподобие Google Фото. После загрузки в такую службу ваших семейных фотографий она автоматически распознает, что одна и та же особа А появляется на фотографиях 1, 5 и 11, а другая особа В — на фотографиях 2, 5 и 7. Так действует часть алгоритма без учителя (кластеризация). Теперь системе нужно узнать у вас, кто эти люди. Всего одной метки на особу⁴ будет достаточно для того, чтобы служба смогла назвать всех на каждой фотографии, что полезно для поиска фотографий.

Большинство алгоритмов частичного обучения являются комбинациями алгоритмов обучения без учителя и с учителем. Например, *глубокие сети доверия (DBN)* основаны на компонентах обучения без учителя, называемых *ограниченными машинами Больцмана (RBM)*, уложенными друг поверх друга. Машины RBM обучаются последовательно способом без учителя, после чего целая система точно настраивается с применением приемов обучения с учителем.

Обучение с подкреплением

Обучение с подкреплением — совершенно другая вещь. Обучающая система, которая в данном контексте называется *агентом*, может наблюдать за средой, выбирать и выполнять действия, выдавая в ответ *награды* (или *штрафы* в форме отрицательных наград, как показано на рис. 1.12). Затем она должна самостоятельно узнать, в чем заключается наилучшая стратегия, называемая *политикой*, чтобы со временем получать наибольшую награду. Политика определяет, какое действие агент обязан выбирать, когда он находится в заданной ситуации.

Например, многие роботы реализуют алгоритмы обучения с подкреплением, чтобы учиться ходить. Также хорошим примером обучения с подкреплением является программа AlphaGo, разработанная DeepMind: она широко освещалась в прессе в мае 2017 года, когда выиграла в *го* у чемпиона мира Кэ Цзе. Программа обучилась своей выигрышной политике благодаря анализу миллионов игр и затем многократной игре против самой себя. Следует отметить, что во время игр с чемпионом обучение было отключено; программа AlphaGo просто применяла политику, которой научилась ранее.

⁴ В ситуации, когда система работает безупречно. На практике она часто создает несколько кластеров на особу, а временами смешивает двух людей, выглядящих похожими, поэтому вам необходимо предоставить несколько меток на особу и вручную очистить некоторые кластеры.



Рис. 1.12. Обучение с подкреплением

Пакетное и динамическое обучение

Еще один критерий, используемый для классификации систем МО, связан с тем, может ли система обучаться постепенно на основе потока входящих данных.

Пакетное обучение

При *пакетном обучении* система неспособна обучаться постепенно: она должна учиться с применением всех доступных данных. В общем случае процесс будет требовать много времени и вычислительных ресурсов, поэтому обычно он проходит автономно. Сначала система обучается, а затем помещается в производственную среду и функционирует без дальнейшего обучения; она просто применяет то, что ранее узнала. Это называется *автономным обучением* (*offline learning*).

Если вы хотите, чтобы система пакетного обучения узнала о новых данных (вроде нового типа спама), тогда понадобится обучить новую версию системы с нуля на полном наборе данных (не только на новых, но также и на старых данных), затем остановить старую систему и заменить ее новой.

К счастью, весь процесс обучения, оценки и запуска системы МО можно довольно легко автоматизировать (как было показано на рис. 1.3), так что даже система пакетного обучения будет способной адаптироваться к изменениям. Просто обновляйте данные и обучайте новую версию системы с нуля настолько часто, насколько требуется.

Такое решение просто и зачастую прекрасно работает, но обучение с использованием полного набора данных может занять много часов, поэтому вы обычно будете обучать новую систему только каждые 24 часа или даже раз в неделю. Если ваша система нуждается в адаптации к быстро меняющимся данным (скажем, для прогнозирования курса акций), то понадобится более реактивное решение.

Кроме того, обучение на полном наборе данных требует много вычислительных ресурсов (процессорного времени, памяти, дискового пространства, дискового ввода-вывода, сетевого ввода-вывода и т.д.). При наличии большого объема данных и автоматизации системы для обучения с нуля ежедневно все закончится тем, что придется тратить много денег. Если объем данных громаден, то применение алгоритма пакетного обучения может даже оказаться невозможным.

Наконец, если ваша система должна быть способной обучаться автономно, и она располагает ограниченными ресурсами (например, приложение смартфона или марсоход), тогда доставка крупных объемов обучающих данных и расходование многочисленных ресурсов для ежедневного обучения в течение часов станет непреодолимой проблемой.

Но во всех упомянутых случаях есть лучший вариант — использование алгоритмов, которые допускают постепенное обучение.

Динамическое обучение

При *динамическом обучении* вы обучаете систему постепенно за счет последовательного предоставления ей образцов данных либо по отдельности, либо небольшими группами, называемыми *мини-пакетами*. Каждый шаг обучения является быстрым и недорогим, так что система может узнавать о новых данных на лету по мере их поступления (рис. 1.13).

Динамическое обучение отлично подходит для систем, которые получают данные в виде непрерывного потока (скажем, курс акций) и должны адаптироваться к изменениям быстро или самостоятельно. Вдобавок динамическое обучение будет хорошим вариантом в случае ограниченных вычислительных ресурсов: после того, как система динамического обучения узнала о новых образцах данных, она в них больше не нуждается, а потому может их отбросить (если только вы не хотите иметь возможность производить откат к предыдущему состоянию и “воспроизведение” данных). В итоге можно сберечь громадный объем пространства.

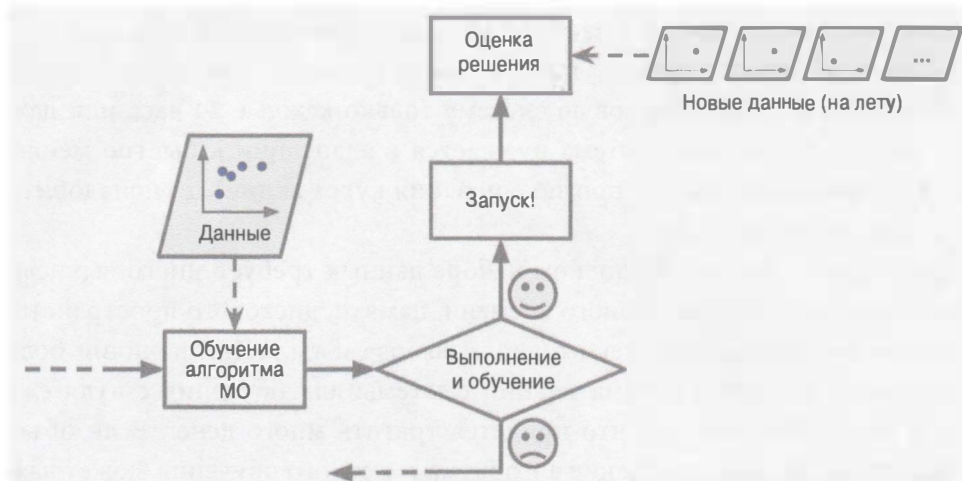


Рис. 1.13. Динамическое обучение

Алгоритмы динамического обучения также могут применяться для обучения систем на гигантских наборах данных, которые не умещаются в основную память одной машины (прием называется *внешним обучением* (*out-of-core learning*)). Алгоритм загружает часть данных, выполняет шаг обучения на этих данных и повторяет процесс до тех пор, пока не пройдет все данные (рис. 1.14).

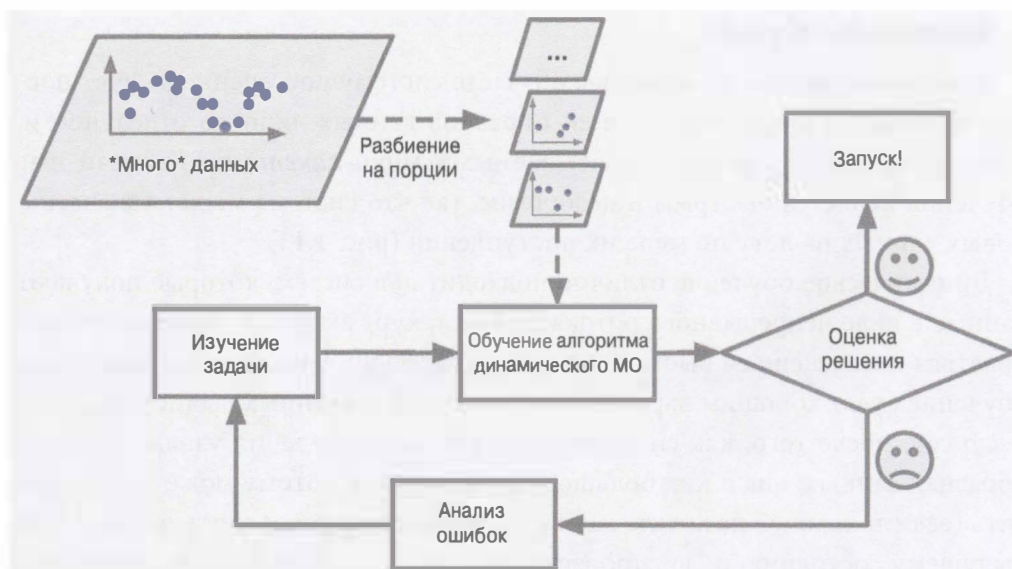


Рис. 1.14. Использование динамического обучения для обработки гигантских наборов данных



Весь этот процесс обычно выполняется автономно (т.е. не на действующей системе), так что название “динамическое обучение” может сбивать с толку. Думайте о нем как о *постепенном обучении* (*incremental learning*).

Важный параметр систем динамического обучения касается того, насколько быстро они должны адаптироваться к меняющимся данным: он называется *скоростью обучения* (*learning rate*). Если вы установите высокую скорость обучения, тогда ваша система будет быстро адаптироваться к новым данным, но также быть склонной скоро забывать старые данные (вряд ли кого устроит фильтр спама, маркирующий только самые последние виды спама, которые ему были показаны). И наоборот, если вы установите низкую скорость обучения, то система станет обладать большей инертностью; т.е. она будет обучаться медленнее, но также окажется менее чувствительной к шуму в новых данных или к последовательностям нерепрезентативных точек данных.

Крупная проблема с динамическим обучением связана с тем, что в случае передачи в систему неправильных данных ее производительность будет понемногу снижаться. Если речь идет о действующей системе, тогда ваши клиенты заметят такое снижение. Например, неправильные данные могли бы поступать из некорректно функционирующего датчика в роботе либо от кого-то, кто заваливает спамом поисковый механизм в попытках повышения рейтинга в результатах поиска. Чтобы сократить этот риск, вам необходимо тщательно следить за системой и, обнаружив уменьшение производительности, сразу же отключать обучение (и возможно возвратиться в предыдущее рабочее состояние). Может также понадобится отслеживать входные данные и реагировать на аномальные данные (скажем, с применением алгоритма обнаружения аномалий).

Обучение на основе образцов или на основе моделей

Очередной способ категоризации систем МО касается того, как они *обобщают*. Большинство задач МО имеют отношение к выработке прогнозов. Это значит, что система должна быть в состоянии обобщить заданное количество обучающих примеров на примеры, которые она никогда не видела ранее. Наличие подходящего показателя производительности на обучающих данных — условие хорошее, но недостаточное; истинная цель в том, чтобы приемлемо работать на новых образцах.

Существуют два основных подхода к обобщению: обучение на основе образцов и обучение на основе моделей.

Обучение на основе образцов

Возможно, наиболее тривиальной формой обучения является просто заучивание на память. Если бы вы создавали фильтр спама в подобной манере, то он маркировал бы лишь сообщения, идентичные сообщениям, которые уже были маркированы пользователями — не худшее, но определенно и не лучшее решение.

Вместо того чтобы маркировать только сообщения, которые идентичны известным спам-сообщениям, ваш фильтр спама мог бы программироваться для маркирования также и сообщений, очень похожих на известные спам-сообщения. Такая задача требует *измерения сходства* между двумя сообщениями. Элементарное измерение сходства между двумя сообщениями могло бы предусматривать подсчет количества совместно используемых ими слов. Система маркировала бы сообщение как спам при наличии в нем многих слов, присутствующих в известном спам-сообщении.

Это называется *обучением на основе образцов*: система учит примеры на память и затем обобщает их на новые примеры с применением измерения сходства (рис. 1.15).

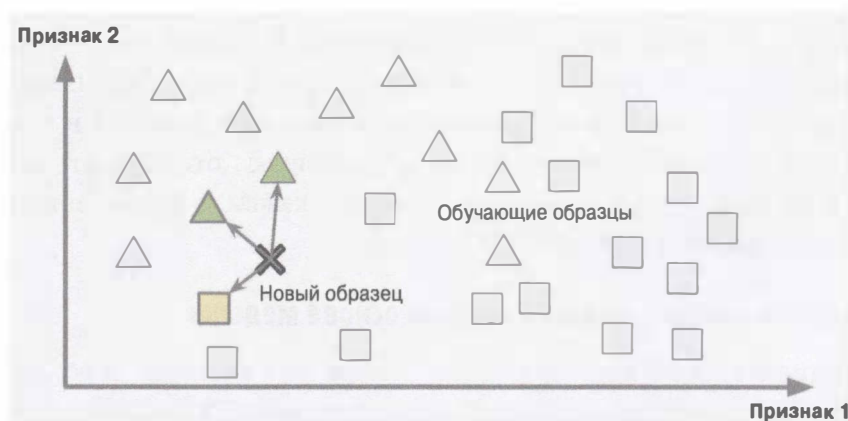


Рис. 1.15. Обучение на основе образцов

Обучение на основе моделей

Другой метод обобщения набора примеров предполагает построение модели этих примеров и ее использование для выработки *прогнозов*. Это называется *обучением на основе моделей* (рис. 1.16).

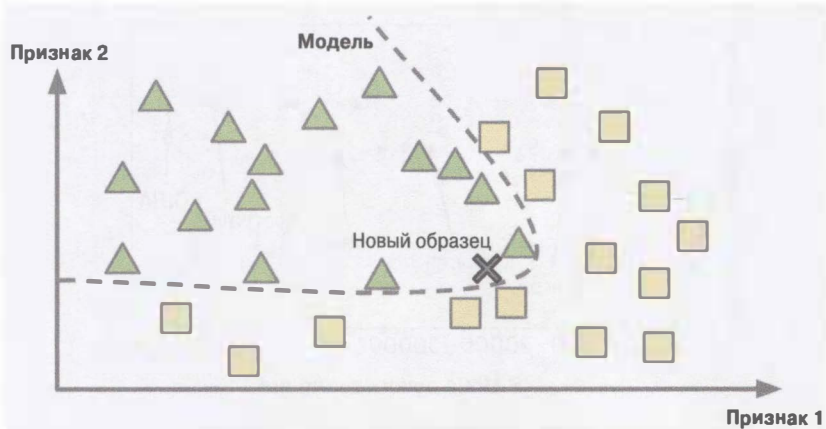


Рис. 1.16. Обучение на основе моделей

Предположим, что вас интересует, делают ли деньги людей счастливыми, а потому вы загружаете данные “Индекс лучшей жизни” из веб-сайта Организации экономического сотрудничества и развития (ОЭСР), а также статистические данные по валовому внутреннему продукту (ВВП) на душу населения из веб-сайта Международного валютного фонда. Затем вы соединяете таблицы и выполняете сортировку по ВВП на душу населения. В табл. 1.1 представлена выборка того, что вы получите.

Таблица 1.1. Делают ли деньги людей счастливыми?

Страна	ВВП на душу населения (в долларах США)	Удовлетворенность жизнью
Венгрия	12 240	4.9
Корея	27 195	5.8
Франция	37 675	6.5
Австралия	50 962	7.3
США	55 805	7.2

Давайте графически представим данные для нескольких произвольно выбранных стран (рис. 1.17).

Кажется, здесь есть тенденция! Хотя данные *зашумлены* (т.е. отчасти произвольны), похоже на то, что удовлетворенность жизнью растет более или менее линейно с увеличением ВВП на душу населения. Итак, вы решили моделировать удовлетворенность жизнью как линейную функцию от ВВП на душу населения. Такой шаг называется *выбором модели*: вы выбрали линейную модель удовлетворенности жизнью с только одним атрибутом — ВВП на душу населения (уравнение 1.1).

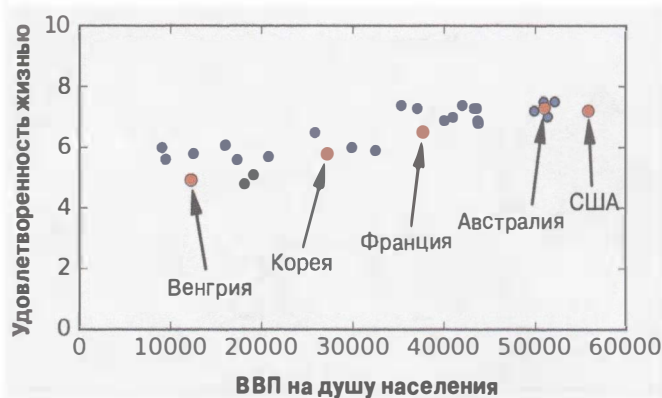


Рис. 1.17. Уловили здесь тенденцию?

Уравнение 1.1. Простая линейная модель

$$\text{удовлетворенность_жизнью} = \theta_0 + \theta_1 \times \text{ВВП_на_душу_населения}$$

Модель имеет два *параметра модели*, θ_0 и θ_1 ⁵. Подстраивая эти параметры, вы можете заставить свою модель представлять любую линейную функцию, как видно на рис. 1.18.



Рис. 1.18. Несколько возможных линейных моделей

Прежде чем модель можно будет использовать, вы должны определить значения параметров θ_0 и θ_1 . Как вы узнаете значения, которые обеспечат лучшее выполнение модели? Чтобы ответить на данный вопрос, понадобится указать измерение производительности.

⁵ По соглашению для представления параметров модели часто применяется греческая буква θ (тета).

Вы можете определить либо *функцию полезности* (*utility function*) (или *функцию приспособленности* (*fitness function*)), которая измеряет, насколько *хороша* ваша модель, либо *функцию издержек* или *функцию стоимости* (*cost function*), которая измеряет, насколько модель *плоха*. Для задач линейной регрессии люди обычно применяют функцию издержек, измеряющую расстояние между прогнозами линейной модели и обучающими примерами; цель заключается в минимизации этого расстояния.

Именно здесь в игру вступает алгоритм линейной регрессии: вы предоставляете ему обучающие примеры, а он находит параметры, которые делают линейную модель лучше подогнанной к имеющимся данным. Процесс называется *обучением* модели. В рассматриваемом случае алгоритм обнаруживает, что оптимальными значениями параметров являются $\theta_0 = 4.85$ и $\theta_1 = 4.91 \times 10^{-5}$.

Теперь модель подогнана к обучающим данным настолько близко, насколько возможно (для линейной модели), что отражено на рис. 1.19.

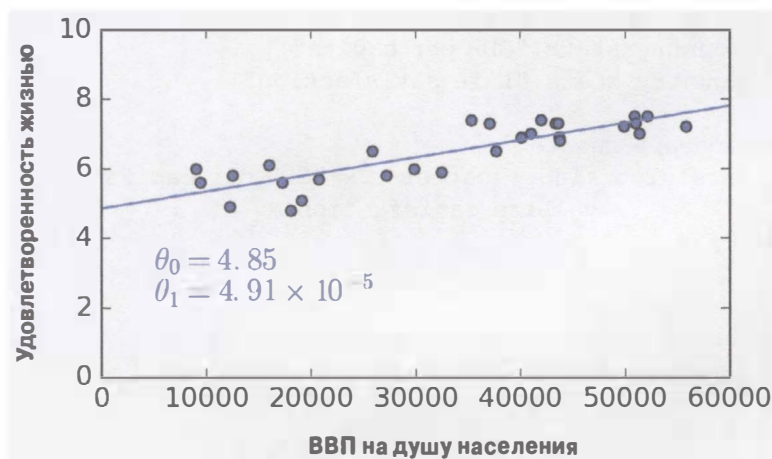


Рис. 1.19. Линейная модель, которая лучше подогнана к обучающим данным

Вы окончательно готовы к запуску модели для выработки прогнозов. Например, пусть вы хотите выяснить, насколько счастливы киприоты, и данные ОЭСР не дают искомого ответа. К счастью, вы можете воспользоваться своей моделью, чтобы получить хороший прогноз: вы ищете ВВП на душу населения Кипра, находите \$22 587, после чего применяете модель и устанавливаете, что удовлетворенность жизнью, вероятно, должна быть где-то $4.85 + 22\,587 \times 4.91 \times 10^{-5} = 5.96$.

Чтобы разжечь ваш аппетит, в примере 1.1 показан код Python, который загружает данные, подготавливает их⁶, создает диаграмму рассеяния для визуализации, затем обучает линейную модель и вырабатывает прогноз⁷.

Пример 1.1. Обучение и прогон линейной модели с использованием Scikit-Learn

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn

# Загрузить данные
oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita = pd.read_csv("gdp_per_capita.csv", thousands=',',
                             delimiter='\t', encoding='latin1', na_values="n/a")

# Подготовить данные
country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]

# Визуализировать данные
country_stats.plot(kind='scatter', x="GDP per capita",
                    y='Life satisfaction')
plt.show()

# Выбрать линейную модель
model = sklearn.linear_model.LinearRegression()

# Обучить модель
model.fit(X, y)

# Выработать прогноз для Кипра
X_new = [[22587]] # ВВП на душу населения Кипра
print(model.predict(X_new)) # выводит [[ 5.96242338]]
```

⁶ В коде предполагается, что функция `prepare_country_stats()` уже определена: она объединяет данные ВВП и удовлетворенности жизнью в единый Pandas-объект `DataFrame`.

⁷ Не переживайте, если пока не понимаете весь код; библиотека Scikit-Learn будет представлена в последующих главах.



Если бы вы применяли взамен алгоритм обучения на основе образцов, тогда выяснили бы, что Словения имеет самое близкое к Кипру значение ВВП на душу населения (\$20 732). Поскольку данные ОЭСР сообщают нам, что удовлетворенность жизнью в Словении составляет 5.7, вы могли бы спрогнозировать для Кипра удовлетворенность жизнью 5.7. Слегка уменьшив масштаб и взглянув на следующие две близко лежащие страны, вы обнаружите Португалию и Испанию с уровнями удовлетворенности жизнью соответственно 5.1 и 6.5. Усредняя указанные три значения, вы получаете 5.77, что довольно близко к прогнозу на основе модели. Такой простой алгоритм называется регрессией *методом k ближайших соседей* (в этом примере $k = 3$).

Замена модели линейной регрессии моделью с регрессией k ближайших соседей в предыдущем коде сводится к замене строки:

```
model = sklearn.linear_model.LinearRegression()
```

следующей строкой:

```
model = sklearn.neighbors.KNeighborsRegressor(n_neighbors=3)
```

Если все прошло нормально, тогда ваша модель будет вырабатывать хорошие прогнозы. Если же нет, то может потребоваться учесть больше атрибутов (уровень занятости, здоровье, загрязнение воздуха и т.д.), получить обучающие данные большего объема или качества либо выбрать более мощную модель (скажем, *полиномиальную регрессионную модель* (*polynomial regression model*)).

Подведем итог вышесказанному:

- вы исследовали данные;
- вы выбрали модель;
- вы обучили модель на обучающих данных (т.е. обучающий алгоритм искал для параметров модели значения, которые доводят до минимума функцию издержек);
- наконец, вы применили модель, чтобы вырабатывать прогнозы на новых образцах (называется *выведением* (*inference*)), надеясь на то, что эта модель будет хорошо обобщаться.

Именно так выглядит типичный проект машинного обучения. В главе 2 вы обретете личный опыт, пройдя проект подобного рода от начала до конца.

До сих пор было раскрыто много основ: теперь вы знаете, что собой в действительности представляет машинное обучение, почему оно полезно, каковы самые распространенные категории систем МО и на что похож рабочий поток типичного проекта. Настало время посмотреть, что может пойти не так в обучении и воспрепятствовать выработке точных прогнозов.

Основные проблемы машинного обучения

Вкратце, поскольку ваша главная задача — выбор алгоритма обучения и его обучение на определенных данных, двумя вещами, которые могут пойти не так, являются “плохой алгоритм” и “плохие данные”. Давайте начнем с примеров плохих данных.

Недостаточный размер обучающих данных

Чтобы научить ребенка, что такое яблоко, вам нужно всего лишь указать на яблоко и произнести слово “яблоко” (возможно повторив процедуру несколько раз). Теперь ребенок способен опознавать яблоки во всех расцветках и формах. Гениальная личность.

Машинному обучению до этого пока далеко; большинству алгоритмов МО для надлежащей работы требуется много данных. Даже для очень простых задач вам обычно нужны тысячи примеров, а для сложных задач, таких как распознавание изображений или речи, понадобятся миллионы примеров (если только вы не можете многократно использовать части существующей модели).

Необоснованная эффективность данных

В знаменитой статье, опубликованной в 2001 году, исследователи из Microsoft Мишель Банко и Эрик Брилл показали, что очень разные алгоритмы МО, включая достаточно простые, выполняются почти одинаково хорошо на сложной задаче устранения неоднозначности в естественном языке⁸, когда им было предоставлено достаточно данных (рис. 1.20).

⁸ Например, в случае английского языка знание, когда в зависимости от контекста писать “to”, “two” или “too”.

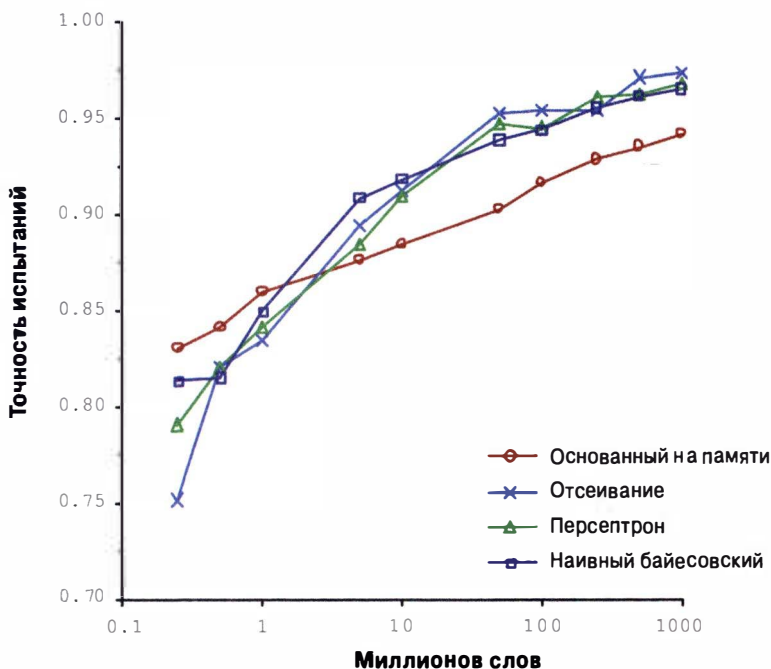


Рис. 1.20. Важность данных в сравнении с алгоритмами⁹

Авторы отмечают: “такие результаты наводят на мысль, что у нас может возникнуть желание пересмотреть компромисс между тратой времени и денег на разработку алгоритмов и тратой их на совокупную разработку”.

Идея о том, что для сложных задач данные более существенны, чем алгоритмы, была в дальнейшем популяризирована Питером Норвигом и другими в статье под названием “The Unreasonable Effectiveness of Data” (“Необоснованная эффективность данных”), опубликованной в 2009 году¹⁰. Однако следует отметить, что наборы данных малых и средних размеров по-прежнему очень распространены, и не всегда легко или дешево получать дополнительные обучающие данные, а потому не отказывайтесь от алгоритмов прямо сейчас.

⁹ Рисунок воспроизведен с разрешения Мишель Банко и Эрика Бриля из работы “Learning Curves for Confusion Set Disambiguation” (“Кривые обучения для устранения неоднозначности путающих наборов”), опубликованной в 2001 году.

¹⁰ “The Unreasonable Effectiveness of Data”, Питер Норvig и другие (2009 год).

Нерепрезентативные обучающие данные

Для успешного обобщения критически важно, чтобы обучающие данные были репрезентативными в отношении новых примеров, на которые вы хотите их обобщить. Это справедливо при обучении как на основе образцов, так и на основе моделей.

Скажем, набор стран, который мы применяли ранее для обучения линейной модели, не был репрезентативным в полной мере; в нем отсутствовало несколько стран. На рис. 1.21 показано, на что похожи данные после добавления недостающих стран.

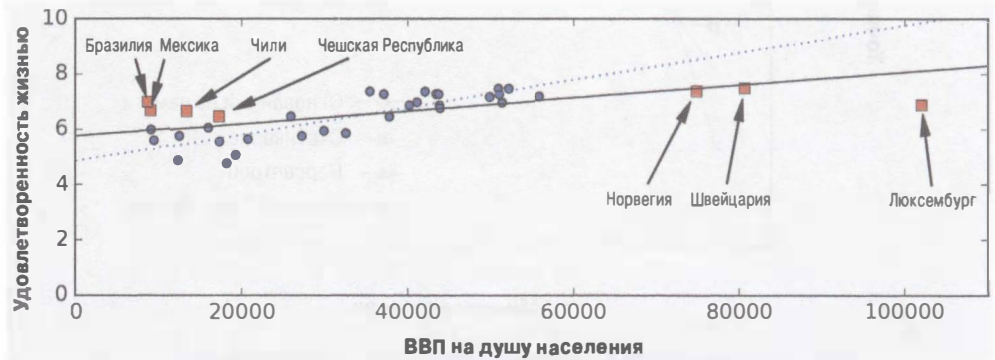


Рис. 1.21. Более репрезентативный обучающий образец

Обучение линейной модели на новых данных дает сплошную линию, в то время как старая модель изображена с помощью точечной линии. Как видите, добавление нескольких отсутствующих ранее стран не только значительно изменяет модель, но также проясняет, что такая простая линейная модель вероятно никогда не будет работать хорошо. Выглядит так, что люди в очень богатых странах не счастливее людей в умеренно богатых странах (фактически они выглядят несчастливыми), и наоборот, люди в некоторых бедных странах кажутся счастливее, чем люди во многих богатых странах.

За счет использования нерепрезентативного обучающего набора мы обучили модель, которая вряд ли будет вырабатывать точные прогнозы, особенно для очень бедных и очень богатых стран.

Крайне важно применять обучающий набор, репрезентативный для примеров, на которые вы хотите обобщить. Достичь такой цели часто труднее, чем может показаться: если образец слишком мал, то вы получите *шум выборки* (*sampling noise*), т.е. нерепрезентативные данные как исход шанса, но даже очень крупные образцы могут быть нерепрезентативными в случае дефектного метода выборки. Это называется *смещением выборки* (*sampling bias*).

Знаменитый пример смещения выборки

Пожалуй, самый знаменитый пример смещения выборки случился во время президентских выборов в США в 1936 году, когда противостояли Лэндон и Рузвельт: журнал *Literary Digest* проводил сверхбольшой опрос, отправив письма приблизительно 10 миллионам людей. Было получено 2.4 миллиона ответов и предсказано, что с высокой вероятностью Лэндон наберет 57% голосов. Взамен выиграл Рузвельт с 62% голосов. Ошибка скрывалась в методе выборки, используемом в *Literary Digest*.

- Во-первых, чтобы получить адреса для отправки писем с опросом, в *Literary Digest* применяли телефонные справочники, перечни подписчиков журналов, списки членов различных клубов и т.п. Указанные списки обычно включали более состоятельных людей, которые с наибольшей вероятностью проголосовали бы за республиканца (следовательно, Лэндона).
- Во-вторых, ответило менее 25% людей, получивших письма с опросом. Это опять приводит к смещению выборки из-за исключения людей, не особо интересующихся политикой, людей, которым не нравится журнал *Literary Digest*, и другие ключевые группы людей. Такой особый тип смещения выборки называется *погрешностью, вызванной неполучением ответов* (*nonresponse bias*).

Вот еще один пример: пусть вы хотите построить систему для распознавания видеоклипов в стиле фанк. Один из способов создания обучающего набора предусматривает поиск в YouTube по запросу “музыка фанк” и использование результирующих видеоклипов. Но такой прием предполагает, что поисковый механизм YouTube возвращает набор видеоклипов, который является репрезентативным для всех видеоклипов в стиле фанк на YouTube. В действительности результаты поиска, скорее всего, окажутся смещенными в пользу популярных исполнителей (а если вы живете в Бразилии, то получите много видеоклипов в стиле фанк-кариока, которые звучат совершенно не похоже на Джеймса Брауна). С другой стороны, как еще можно получить крупный обучающий набор?

Данные плохого качества

Очевидно, если ваши обучающие данные полны ошибок, выбросов и шума (например, вследствие измерений плохого качества), тогда они затруднят для системы выявление лежащих в основе паттернов, и потому менее вероятно, что система будет функционировать хорошо. Часто стоит выделить время на очистку обучающих данных. На самом деле большинство специалистов по работе с данными тратят значительную часть своего времени, занимаясь именно этим. Ниже перечислены возможные меры.

- Если некоторые примеры являются несомненными выбросами, то может помочь простое их отбрасывание или попытка вручную исправить ошибки.
- Если в некоторых примерах отсутствуют какие-то признаки (скажем, 5% ваших заказчиков не указали свой возраст), тогда вам потребуется решить, что делать: игнорировать такие атрибуты; игнорировать примеры с недостающими атрибутами; заполнить отсутствующие значения (возможно, средним возрастом); обучать одну модель с признаками и одну модель без признаков; и т.д.

Несущественные признаки

Как говорится, мусор на входе — мусор на выходе. Ваша система будет способна обучаться, только если обучающие данные содержат достаточное количество существенных признаков и не очень много несущественных. Важная часть успеха проекта МО вытекает из хорошего набора признаков, на котором производится обучение. Такой процесс, называемый *конструированием признаков* (*feature engineering*), включает в себя:

- *выбор признаков* (*feature selection*) — выбор самых полезных признаков для обучения на существующих признаках;
- *выделение признаков* (*feature extraction*) — сочетание существующих признаков для выпуска более полезного признака (как было показано ранее, помочь может алгоритм понижения размерности);
- *создание новых признаков* путем сбора новых данных.

Теперь, когда были продемонстрированы многие примеры плохих данных, давайте рассмотрим пару примеров плохих алгоритмов.

Переобучение обучающих данных

Предположим, что во время вашего путешествия по зарубежной стране вас обворовал таксист. Вы можете поддаться искушению и заявить, что *все* таксисты в этой стране являются ворами. Чрезмерное обобщение представляет собой то, что мы, люди, делаем слишком часто, и к несчастью машины могут попасть в аналогичную ловушку, если не соблюдать осторожность. В контексте МО такая ситуация называется *переобучением* (*overfitting*): это означает, что модель хорошо выполняется на обучающих данных, но не обобщается как следует.

На рис. 1.22 показан пример полиномиальной модели высокого порядка для степени удовлетворенности жизнью, где обучающие данные чрезмерно переобучаются. Хотя она гораздо лучше выполняется на обучающих данных, чем простая линейная модель, действительно ли вы будете доверять ее прогнозам?

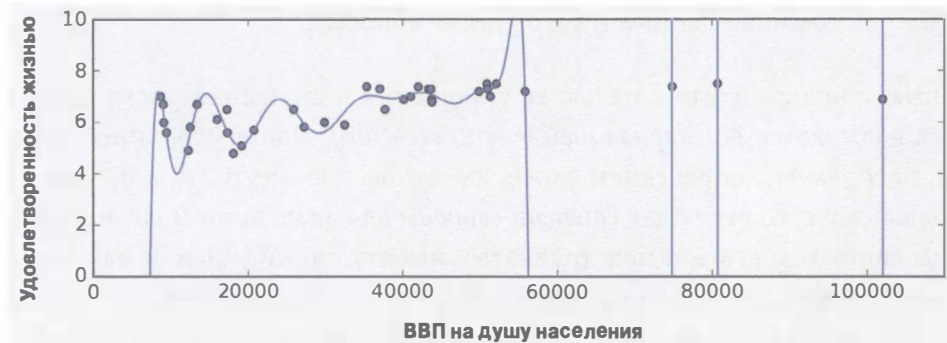


Рис. 1.22. Переобучение обучающих данных

Сложные модели, такие как глубокие нейронные сети, могут обнаруживать едва различимые паттерны в данных, но если обучающий набор содержит шумы или слишком мал (что привносит шум выборки), то модель, скорее всего, станет выявлять паттерны в самом шуме. Очевидно, такие паттерны не будут обобщаться на новые примеры. Предположим, что вы снабдили свою модель для степени удовлетворенности жизнью многими дополнительными атрибутами, включая неинформативные атрибуты наподобие названия страны. В таком случае сложная модель может обнаруживать паттерны вроде того факта, что все страны в обучающем наборе, название которых содержит букву “в”, имеют удовлетворенность жизнью выше 7: Новая Зеландия (7.3), Норвегия (7.4), Швеция (7.2) и Швейцария (7.5).

Насколько вы уверены в том, что правило, касающееся наличия буквы “в” в названии страны, должно быть обобщено на Зимбабве или Малави? Понятно, что такой паттерн возник в обучающих данных по чистой случайности, но модель не в состоянии сообщить, реален ли паттерн или же является просто результатом шума в данных.



Переобучение происходит, когда модель слишком сложна относительно объема и зашумленности обучающих данных. Ниже перечислены возможные решения.

- Упростить модель, выбрав вариант с меньшим числом параметров (например, линейную модель вместо полиномиальной модели высокого порядка), сократив количество атрибутов в обучающих данных или ограничив модель.
- Накопить больше обучающих данных.
- Понизить шум в обучающих данных (например, исправить ошибки данных и устранить выбросы).

Ограничение модели с целью ее упрощения и снижения риска переобучения называется *регуляризацией* (*regularization*). Например, линейная модель, которую мы определили ранее, имеет два параметра, θ_0 и θ_1 . Это дает обучающему алгоритму две *степени свободы* для адаптации модели к обучающим данным: он может подстраивать и высоту линии (θ_0), и ее наклон (θ_1). Если мы принудительно установим θ_1 в 0, то алгоритм получит только одну степень свободы, и ему будет гораздо труднее подгонять данные надлежащим образом: он сможет лишь перемещать линию вверх или вниз, чтобы максимально приблизиться к обучающим образцам, что закончится вокруг среднего. Действительно очень простая модель! Если мы разрешим алгоритму модифицировать параметр θ_1 , но вынудим удерживать его небольшим, тогда обучающий алгоритм фактически будет находиться где-то между одной и двумя степенями свободы. Он породит более простую модель, чем с двумя степенями свободы, но сложнее, чем только с одной. Вы хотите добиться правильного баланса между идеальной подгонкой к данным и сохранением модели достаточно простой для того, чтобы обеспечить ее хорошее обобщение.

На рис. 1.23 показаны три модели: точечная линия представляет исходную модель, которая обучалась при нескольких отсутствующих странах, пунктир-

ная линия — вторую модель, обучающуюся со всеми странами, а сплошная линия — линейную модель, обучающуюся на тех же данных, что и первая модель, но с ограничением регуляризации. Как видите, регуляризация вынудила модель иметь меньший наклон, что чуть хуже подгоняет ее к обучающим данным, на которых она обучалась, но фактически позволяет модели лучше обобщаться на новые примеры.

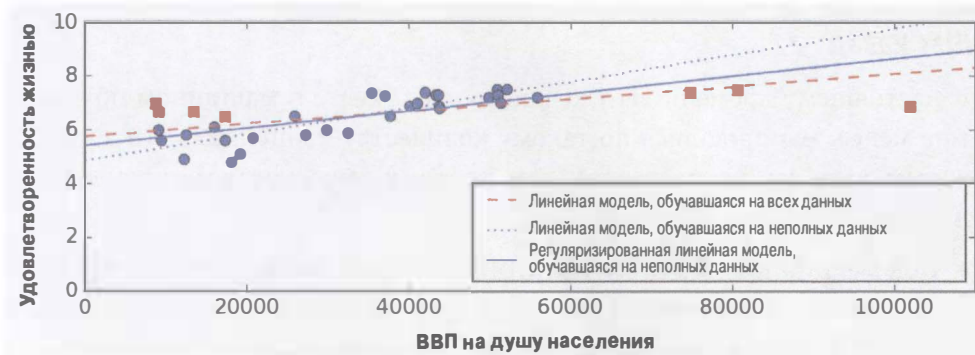


Рис. 1.23. Регуляризация снижает риск переобучения

Объемом регуляризации, подлежащим применению во время обучения, можно управлять с помощью *гиперпараметра*. Гиперпараметр — это параметр обучающего алгоритма (не модели). По существу сам обучающий алгоритм на него не влияет; гиперпараметр должен быть установлен перед обучением, а во время обучения оставаться постоянным. Если вы установите гиперпараметр регуляризации в очень большую величину, то получите почти плоскую модель (с наклоном, близким к нулю); обучающий алгоритм почти наверняка не допустит переобучения обучающих данных, но с меньшей вероятностью найдет хорошее решение. Регулировка гиперпараметров — важная часть построения системы МО (подробный пример приводится в следующей главе).

Недообучение обучающих данных

Как вы могли догадаться, *недообучение* (*underfitting*) является противоположностью переобучения: оно происходит, когда ваша модель слишком проста, чтобы узнать лежащую в основе структуру данных. Например, линейная модель для степени удовлетворенности жизнью предрасположена к недообучению; реальность сложнее модели, так что ее прогнозы неизбежно будут неточными, даже на обучающих примерах.

Вот основные варианты исправления проблемы.

- Выбор более мощной модели с большим числом параметров.
- Предоставление обучающему алгоритму лучших признаков (конструирование признаков).
- Уменьшение ограничений на модели (например, снижение величины гиперпараметра регуляризации).

Шаг назад

К настоящему времени вы уже много чего знаете о машинном обучении. Тем не менее, мы прошли по такому количеству концепций, что вы можете испытывать легкую растерянность, поэтому давайте сделаем шаг назад и посмотрим на общую картину.

- Под машинным обучением понимается обеспечение лучшей обработки машинами определенной задачи за счет их обучения на основе данных вместо явного кодирования правил.
- Существует много разных типов систем машинного обучения: с учителем или без, пакетное или динамическое, на основе образцов или на основе моделей и т.д.
- В проекте МО вы накапливаете данные в обучающем наборе и предоставляете его обучающему алгоритму. Если алгоритм основан на моделях, тогда он настраивает ряд параметров, чтобы подогнать модель к обучающему набору (т.е., чтобы вырабатывать качественные прогнозы на самом обучающем наборе), и затем есть надежда, что алгоритм будет способен вырабатывать пригодные прогнозы также и на новых образцах. Если алгоритм основан на образцах, то он просто заучивает образцы на память и использует измерение сходства для обобщения на новые образцы.
- Система не будет хорошо работать, если обучающий набор слишком мал или данные нерепрезентативны, зашумлены либо загрязнены несущественными признаками (мусор на входе — мусор на выходе). В конце концов, ваша модель не должна быть ни чересчур простой (и переобучаться), ни излишне сложной (и недообучаться).

Осталось раскрыть лишь одну важную тему: после обучения модели вы вовсе не хотите просто “надеяться” на то, что она обобщится на новые образцы. Вы желаете ее оценить и при необходимости точно настроить. Давайте посмотрим как.

Испытание и проверка

Единственный способ узнать, насколько хорошо модель будет обобщаться на новые образцы, предусматривает ее действительное испытание на новых образцах. Для этого модель можно поместить в производственную среду и понаблюдать за ее функционированием. Вполне нормальный прием, но если ваша модель крайне плоха, то пользователи выразят недовольство — не самый лучший план.

Более подходящий вариант заключается в том, чтобы разделить данные на два набора: *обучающий набор* (*training set*) и *испытательный набор* (*test set*). Как следует из названий, вы обучаете свою модель с применением обучающего набора и испытываете ее, используя испытательный набор. Частота ошибок на новых образцах называется *ошибкой обобщения* (*generalization error*) или *ошибкой выхода за пределы выборки* (*out-of-sample error*), и путем оценивания модели на испытательном наборе вы получаете оценку такой ошибки. Результирующее значение сообщает вам, насколько хорошо модель будет работать с образцами, которые она никогда не видела ранее.

Если ошибка обучения низкая (т.е. ваша модель допускает немного погрешностей на обучающем наборе), но ошибка обобщения высокая, то это значит, что модель переобучена обучающими данными.



Принято применять 80% данных для обучения и *требовать* 20% данных для испытания.

Итак, оценка модели достаточно проста: нужно лишь воспользоваться испытательным набором. Теперь предположим, что вы колеблетесь между двумя моделями (скажем, линейной и полиномиальной): как принять решение? Один из вариантов — обучить обе и сравнить, насколько хорошо они обобщаются, с помощью испытательного набора.

Далее представим, что линейная модель обобщается лучше, но во избежание переобучения вы хотите применить какую-то регуляризацию. Вопрос в том, каким образом вы выберете значение гиперпараметра регуляризации? Можно обучить 100 разных моделей, используя для этого гиперпараметра 100 отличающихся значений. Пусть вы нашли лучшее значение гиперпараметра, которое производит модель с наименьшей ошибкой обобщения — всего 5%.

Затем вы запускаете такую модель в производство, но к несчастью она функционирует не настолько хорошо, как ожидалось, давая 15% ошибок. Что же произошло?

Проблема в том, что вы измерили ошибку обобщения на испытательном наборе много раз, после чего адаптировали модель и гиперпараметры, чтобы выпустить лучшую модель *для указанного набора*. Это означает, что модель вряд ли выполняется столь же хорошо на новых данных.

Общее решение проблемы предусматривает наличие второго набора, который называется *проверочным набором* (*validation set*). Вы обучаете множество моделей с разнообразными гиперпараметрами, применяя обучающий набор, выбираете модель и гиперпараметры, которые обеспечивают лучшее выполнение на проверочном наборе, и когда модель устраивает, запускаете финальный тест на испытательном наборе, чтобы получить оценку ошибки обобщения.

Во избежание “излишней траты” слишком большого объема обучающих данных в проверочных наборах распространенный прием заключается в использовании *перекрестной проверки* (*cross-validation*): обучающий набор разбивается на дополняющие поднаборы, а каждая модель обучается на разной комбинации таких поднаборов и проверяется на оставшихся поднаборах. После того, как были выбраны тип модели и гиперпараметры, окончательная модель обучается с применением этих гиперпараметров на полном обучающем наборе, и ошибка обобщения измеряется на испытательном наборе.

Теорема об отсутствии бесплатных завтраков

Модель является упрощенной версией наблюдений. Упрощения означают отбрасывание избыточных деталей, которые вряд ли обобщатся на новые образцы. Однако чтобы решить, какие данные отбрасывать, а какие оставлять, вы должны делать *предположения*. Например, линейная модель выдвигает предположение о том, что данные фундаментально линейны и расстояние между образцами и прямой линией — просто шум, который можно безопасно игнорировать.

В известной работе 1996 года¹¹ Дэвид Вольперт продемонстрировал, что если вы не делаете абсолютно никаких предположений о данных, тогда нет оснований отдавать предпочтение одной модели перед любой другой.

¹¹ “The Lack of A Priori Distinctions Between Learning Algorithms” (“Отсутствие априорных различий между обучающими алгоритмами”), Д. Вольперт (1996 год).

Это теорема *об отсутствии бесплатных завтраков (No Free Lunch — NFL)* (встречаются варианты “бесплатных завтраков не бывает”, “бесплатных обедов не бывает” и т.п. — *примеч. пер.*). Для одних наборов данных наилучшей моделью является линейная, в то время как для других ею будет нейронная сеть. Нет модели, которая *априори* гарантировала бы лучшую работу (отсюда и название теоремы). Единственный способ достоверно знать, какая модель лучше, предусматривает оценку всех моделей. Поскольку это невозможно, на практике вы делаете некоторые разумные предположения о данных и оцениваете лишь несколько рациональных моделей. Например, для простых задач вы можете оценивать линейные модели с различными уровнями регуляризации, а для сложных задач — разнообразные нейронные сети.

Упражнения

В этой главе рассматривались некоторые из самых важных концепций машинного обучения. В последующих главах мы будем погружаться более глубоко и писать больше кода, но прежде чем переходить к ним, удостоверьтесь в том, что способны дать ответы на перечисленные ниже вопросы.

1. Как бы вы определили машинное обучение?
2. Можете ли вы назвать четыре типа задач, где МО показывает блестящие результаты?
3. Что такое помеченный обучающий набор?
4. Каковы две наиболее распространенных задачи обучения с учителем?
5. Можете ли вы назвать четыре распространенных задачи обучения без учителя?
6. Какой тип алгоритма МО вы бы использовали, чтобы сделать возможным прохождение роботом по разнообразным неизведанным территориям?
7. Какой тип алгоритма вы бы применяли для сегментирования своих заказчиков в несколько групп?
8. Как бы вы представили задачу выявления спама — как задачу обучения с учителем или как задачу обучения без учителя?

9. Что такое система динамического обучения?
10. Что такое внешнее обучение?
11. Какой тип обучающего алгоритма при выработке прогнозов полагается на измерение сходства?
12. В чем разница между параметром модели и гиперпараметром обучающего алгоритма?
13. Что ищут алгоритмы обучения на основе моделей? Какую наиболее распространенную стратегию они используют для достижения успеха? Как они вырабатывают прогнозы?
14. Можете ли вы назвать четыре основных проблемы в машинном обучении?
15. Что происходит, если ваша модель хорошо работает с обучающими данными, но плохо обобщается на новые образцы? Можете ли вы назвать три возможных решения?
16. Что такое испытательный набор и почему он может применяться?
17. В чем заключается цель проверочного набора?
18. Что может пойти не так при настройке гиперпараметров с использованием испытательного набора?
19. Что такое перекрестная проверка и почему ей отдается предпочтение перед проверочным набором?

Решения приведенных упражнений доступны в приложении А.

Полный проект машинного обучения

В этой главе вы подробно разберете образец проекта, разыгрывая из себя недавно нанятого специалиста по работе с данными в компании, которая занимается недвижимостью¹. Вы пройдете через следующие основные шаги.

1. Выяснение общей картины.
2. Получение данных.
3. Обнаружение и визуализация данных для понимания их сущности.
4. Подготовка данных для алгоритмов машинного обучения.
5. Выбор модели и ее обучение.
6. Точная настройка модели.
7. Представление своего решения.
8. Запуск, наблюдение и сопровождение системы.

Работа с реальными данными

При освоении машинного обучения лучше действительно экспериментировать с реально существующими данными, а не только с искусственными наборами данных. К счастью, на выбор доступны тысячи открытых баз данных, охватывающих все виды предметных областей. Ниже перечислено несколько мест, куда вы можете заглянуть, чтобы получить данные.

- Популярные открытые хранилища данных:
 - хранилище для машинного обучения Калифорнийского университета в Ирвайне (UC Irvine Machine Learning Repository; <http://archive.ics.uci.edu/ml/index.php>)

¹ Образец проекта полностью выдуман; наша цель в том, чтобы продемонстрировать основные шаги проекта МО, а не узнать что-то о сфере работы с недвижимостью.

- наборы данных Kaggle (<https://www.kaggle.com/datasets>)
- наборы данных AWS в Amazon (<https://aws.amazon.com/public-datasets>)
- Метапорталы (содержат списки открытых хранилищ данных):
 - <http://dataportals.org/>
 - <http://opendatamonitor.eu/>
 - <http://quandl.com/>
- Другие страницы со списками многих популярных открытых хранилищ данных:
 - список наборов данных для машинного обучения в англоязычной Википедии (https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research)
 - вопрос в Quora.com (<https://www.quora.com/Where-can-I-find-large-datasets-open-to-the-public>)
 - сабреддит Datasets (<https://www.reddit.com/r/datasets/>)

Для настоящей главы мы выбрали набор данных California Housing Prices (Цены на жилье в Калифорнии) из хранилища StatLib² (рис. 2.1).

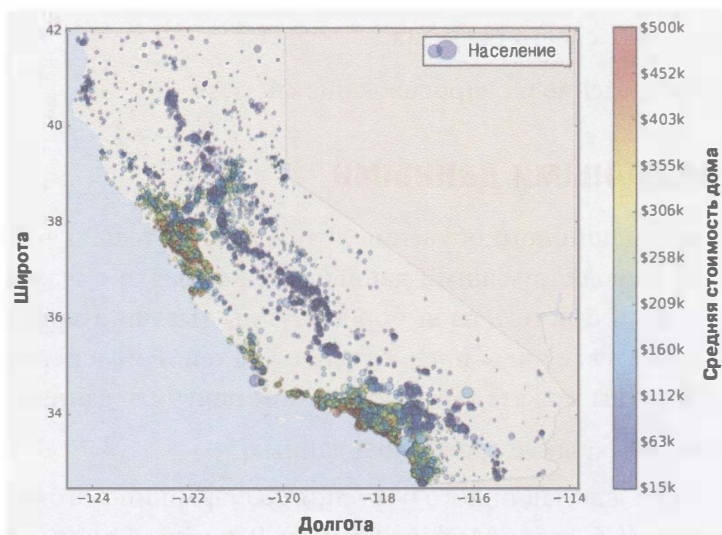


Рис. 2.1. Цены на жилье в Калифорнии

² Первоначально этот набор данных появился в статье Р. Келли Пейса и Рональда Барри “Sparse Spatial Autoregressions” (“Разреженные пространственные авторегрессии”), опубликованной в журнале *Statistics & Probability Letters*, том 33, выпуск 3 (1997 год): стр. 291–297.

Этот набор был основан на данных переписи населения Калифорнии 1990 года. Он не особенно новый, но обладает многими качествами для обучения, потому мы притворимся, что в нем содержатся последние данные. В целях обучения мы также добавили категориальный атрибут и удалили несколько признаков.

Выяснение общей картины

Добро пожаловать в Корпорацию по жилью, вооруженную машинным обучением! Первой задачей, которую вас попросили выполнить, стало построение модели цен на жилье в Калифорнии, используя данные местной переписи населения. Эти данные имеют метрики, такие как население, медианный доход, средняя стоимость дома и т.д., для каждой группы блоков в Калифорнии. Группы блоков являются наименьшими географическими единицами, для которых Бюро переписи населения США публикует выборочные данные (группа блоков обычно имеет население от 600 до 3000 человек). Для краткости мы будем называть их просто “округами”. Ваша модель должна обучаться на таких данных и быть способной прогнозировать среднюю стоимость дома в любом округе, учитывая все остальные метрики.



Как хорошо организованный специалист по работе с данными первым делом вы достаете свой контрольный перечень для проекта МО. Можете начать с перечня, приведенного в приложении Б; он должен работать достаточно хорошо в большинстве проектов МО, но позаботьтесь о его приспособлении к имеющимся нуждам. Мы пройдемся в этой главе по многим пунктам контрольного перечня, но и пропустим несколько либо потому, что они самоочевидны, либо из-за того, что они будут обсуждаться в последующих главах.

Постановка задачи

Первый вопрос, который вы задаете своему шефу, касается того, что собой в точности представляет бизнес-задача; возможно, построение модели не является конечной целью. Каким образом компания рассчитывает применять и извлекать пользу от модели? Это важно, поскольку определяет то, как будет поставлена задача, какие алгоритмы будут выбраны, какой критерий качества работы будет использоваться для оценки модели и сколько усилий придется потратить для ее подстройки.

Ваш шеф отвечает, что выходные данные модели (прогноз средней стоимости дома в каком-то округе) будут передаваться в другую систему МО

(рис. 2.2) наряду со многими другими *сигналами*³. Такая нисходящая система будет определять, стоит или нет инвестировать в выбранную область. Правильность ее работы критически важна, потому что данное обстоятельство напрямую влияет на доход.

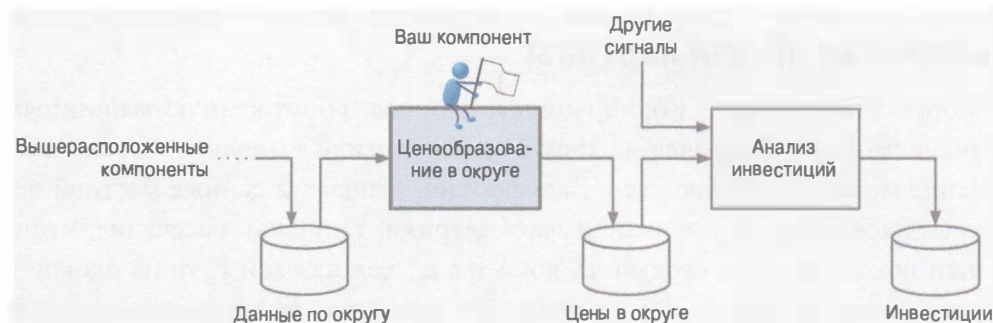


Рис. 2.2. Конвейер машинного обучения для инвестиций в недвижимость

Конвейеры

Последовательность *компонентов* обработки данных называется *конвейером*. Конвейеры очень распространены в системах МО, т.к. в них существует большой объем данных для манипулирования и много трансформаций данных для применения.

Компоненты обычно выполняются асинхронно. Каждый компонент захватывает массу данных, обрабатывает их и выдает результат в другое хранилище данных, после чего некоторое время спустя следующий компонент в конвейере захватывает эти данные и выдает собственный вывод и т.д. Каждый компонент довольно самодостаточен: в качестве интерфейса между компонентами выступает хранилище данных. В итоге понять систему достаточно просто (с помощью графа потока данных), и разные команды разработчиков могут сосредоточиться на разных компонентах. Более того, если компонент выходит из строя, то нижерасположенные компоненты часто способны продолжить нормальное функционирование (во всяком случае, какое-то время), используя вывод из неисправного компонента. Такой подход делает архитектуру действительно надежной.

С другой стороны, неисправный компонент в течение некоторого времени может оставаться незамеченным, если не реализовано надлежащее наблюдение. Данные устаревают, и общая производительность системы падает.

³ Порцию информации, передаваемой в систему МО, часто называют *сигналом*, ссылаясь на теорию информации Шеннона: вам нужно высокое соотношение сигнал/шум.

Следующим вы задаете вопрос, на что похоже текущее решение (если оно есть). Это часто даст опорную производительность, а также догадки о том, как решать задачу. Ваш шеф отвечает, что в настоящий момент приблизительные цены в округе подсчитываются экспертами вручную: команда собирает новейшую информацию об округе, и когда получить среднюю стоимость дома не удастся, она оценивается с применением сложных правил.

Такое решение является дорогостоящим и отнимающим много времени, а оценки — далеко не замечательными; в случаях, когда экспертам удастся узнать действительную среднюю стоимость дома, они нередко осознают, что их оценки расходятся более чем на 10%. Именно потому в компании полагают, что было бы полезно обучить модель для прогнозирования средней стоимости дома по округу, располагая другими данными об округе. Данные переписи выглядят прекрасным набором данных для эксплуатации в таких целях, поскольку он включает средние стоимости домов в сотнях округов, а также другие данные.

Хорошо, располагая всей упомянутой информацией, теперь вы готовы приступить к проектированию системы. Прежде всего, вам необходимо определиться с задачей: она будет обучением с учителем, обучением без учителя или обучением с подкреплением? Это задача классификации, задача регрессии или что-то еще? Должны ли вы использовать технологии пакетного обучения или динамического обучения? До того как читать дальше, остановитесь и попробуйте сами ответить на перечисленные вопросы.

Удалось ли вам найти ответы? Давайте посмотрим, что у нас есть: вы имеете дело с типичной задачей обучения с учителем, т.к. вам предоставили *помеченные* обучающие образцы (к каждому образцу прилагается ожидаемый вывод, т.е. средняя стоимость дома по округу). Кроме того, она также является типовой задачей регрессии, поскольку вам предложено спрогнозировать значение. Точнее говоря, это задача *многомерной регрессии* (*multivariate regression*), потому что для выработки прогноза система будет применять множество признаков (она будет использовать население округа, медианный доход и т.д.). В первой главе вы прогнозировали степень удовлетворенности жизнью на основе всего лишь одного признака, ВВП на душу населения, так что это была задача *одномерной регрессии* (*univariate regression*). Наконец, отсутствует непрерывный поток поступающих в систему данных, нет специфической потребности в быстром приспособлении к меняющимся данным, а объем самих данных достаточно мал, позволяя им уместиться в памяти. Таким образом, должно прекрасно подойти простое пакетное обучение.



Если бы данные были гигантскими, тогда вы могли бы разнести работу пакетного обучения на несколько серверов (с применением технологии *MapReduce*, как будет показано позже) либо взамен использовать прием динамического обучения.

Выбор критерия качества работы

Ваш следующий шаг заключается в выборе критерия качества работы. Типичный критерий качества для задач регрессии — *квадратный корень из среднеквадратической ошибки* (*Root Mean Squared Error — RMSE*). Она дает представление о том, насколько большую ошибку система обычно допускает в своих прогнозах, с более высоким весом для крупных ошибок. В уравнении 2.1 показана математическая формула для вычисления RMSE.

Уравнение 2.1. Квадратный корень из среднеквадратической ошибки (RMSE)

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

Обозначения

Уравнение 2.1 вводит несколько очень распространенных обозначений МО, которые мы будем применять повсюду в книге.

- m — количество образцов в наборе данных, на которых измеряется ошибка RMSE.
 - Например, если вы оцениваете ошибку RMSE на проверочном наборе из 2 000 округов, то $m = 2000$.
- $\mathbf{x}^{(i)}$ — вектор всех значений признаков (исключая метку) i -го образца в наборе данных, а $y^{(i)}$ — его метка (желательное выходное значение для данного образца).
 - Например, если первый округ в наборе данных находится в месте с долготой -118.29° и широтой 33.91° , имеет 1 416 жителей с медианным доходом \$38 372, а средняя стоимость дома составляет \$156 400 (пока игнорируя другие признаки), тогда:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1\,416 \\ 38\,372 \end{pmatrix}$$

и

$$y^{(1)} = 156\,400$$

- \mathbf{X} — матрица, содержащая все значения признаков (исключая метки) всех образцов в наборе данных. Предусмотрена одна строка на образец, а i -тая строка эквивалентна транспонированию⁴ $\mathbf{x}^{(i)}$, что обозначается как $(\mathbf{x}^{(i)})^T$.
- Например, если первый округ оказывается таким, как только что описанный, тогда матрица \mathbf{X} выглядит следующим образом:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^T \\ (\mathbf{x}^{(2)})^T \\ \vdots \\ (\mathbf{x}^{(1999)})^T \\ (\mathbf{x}^{(2000)})^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1416 & 38372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- h — функция прогнозирования системы, также называемая *гипотезой* (*hypothesis*). Когда системе предоставляется вектор признаков образца $\mathbf{x}^{(i)}$, она выводит для этого образца прогнозируемое значение $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$.
- Например, если система прогнозирует, что средняя стоимость дома в первом округе равна \$158 400, тогда $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158\,400$. Ошибка прогноза для этого округа составляет $\hat{y}^{(1)} - y^{(1)} = 2000$.
- $\text{RMSE}(\mathbf{X}, h)$ — функция издержек, измеренная на наборе образцов с использованием гипотезы h .

Мы применяем строчные курсивные буквы для обозначения скалярных значений (таких как m или $y^{(i)}$) и имен функций (наподобие h), строчные полужирные буквы для векторов (вроде $\mathbf{x}^{(i)}$) и прописные полужирные буквы для матриц (таких как \mathbf{X}).

⁴ Вспомните, что операция транспонирования заменяет вектор столбца вектором строки (и наоборот).

Несмотря на то что ошибка RMSE в целом является предпочтительным критерием качества для задач регрессии, в некоторых контекстах вы можете использовать другую функцию. Предположим, что существует много округов с выбросами. В такой ситуации вы можете обдумать применение *средней абсолютной ошибки* (*Mean Absolute Error — MAE*), также называемой *средним абсолютным отклонением* (*average absolute deviation*), которая показана в уравнении 2.2:

Уравнение 2.2. Средняя абсолютная ошибка (MAE)

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

Показатели RMSE и MAE представляют собой способы измерения расстояния между двумя векторами: вектором прогнозов и вектором целевых значений. Существуют разнообразные меры расстояния, или *нормы*.

- Вычисление квадратного корня из суммы квадратов (RMSE) соответствует *евклидовой норме* (*euclidian norm*): это понятие расстояния, с которым вы знакомы. Она также называется *нормой ℓ_2* и обозначается как $\|\cdot\|_2$ (или просто $\|\cdot\|$).
- Вычисление суммы абсолютных величин (MAE) соответствует *норме ℓ_1* и обозначается как $\|\cdot\|_1$. Иногда ее называют *нормой Манхэттена* (*Manhattan norm*), потому что она измеряет расстояние между двумя точками в городе, если вы можете перемещаться только вдоль прямоугольных городских кварталов.
- В более общем смысле *норма ℓ_k* вектора \mathbf{v} , содержащего n элементов, определяется как $\|\mathbf{v}\|_k = (|v_0|^k + |v_1|^k + \dots + |v_n|^k)^{\frac{1}{k}}$. ℓ_0 просто дает количество ненулевых элементов в векторе, а ℓ_∞ — максимальную абсолютную величину в векторе.
- Чем выше индекс нормы, тем больше она концентрируется на крупных значениях и пренебрегает мелкими значениями. Вот почему ошибка RMSE чувствительнее к выбросам, чем ошибка MAE. Но когда выбросы экспоненциально редкие (как в колоколообразной кривой), ошибка RMSE работает очень хорошо и обычно является предпочтительной.

Проверка допущений

Наконец, хорошая практика предусматривает перечисление и проверку допущений, которые были сделаны до сих пор (вами или другими); это может выявить серьезные проблемы на ранней стадии. Например, цены на дома в округе, которые выводит ваша система, планируется передавать нижерасположенной системе МО, и мы допускаем, что эти цены будут использоваться как таковые. Но что, если нижерасположенная система на самом деле преобразует цены в категории (скажем, “дешевая”, “средняя” или “дорогая”) и затем применяет такие категории вместо самих цен? Тогда получение совершенно правильной цены вообще не важно; вашей системе необходимо лишь получить правильную категорию. Если ситуация выглядит именно так, то задача должна быть сформулирована как задача классификации, а не регрессии. Вряд ли вам захочется обнаружить это после многомесячной работы над регрессионной системой.

К счастью, после разговора с командой, ответственной за нижерасположенную систему, вы обрели уверенность в том, что им действительно нужны фактические цены, а не просто категории. Великолепно! Вы в полной готовности, дан зеленый свет, и можно приступать к написанию кода прямо сейчас!

Получение данных

Пришло время засучить рукава. Без колебаний доставайте свой ноутбук и займитесь проработкой приведенных далее примеров кода в тетради Jupyter (notebook; также встречается вариант “записная книжка” — *примеч. пер.*). Все тетради Jupyter доступны для загрузки по адресу <https://github.com/ageron/handson-ml>.

Создание рабочей области

Первое, что необходимо иметь — установленную копию Python. Возможно, она уже есть в вашей системе. Если нет, тогда загрузите и установите ее из веб-сайта <https://www.python.org/>⁵.

⁵ Рекомендуется использовать последнюю версию Python 3. Версия Python 2.7+ также будет хорошо работать, но она устарела. Если вы применяете Python 2, то должны добавить в начало своего кода `from __future__ import (division, print_function, unicode_literals)`.

Далее понадобится создать каталог рабочей области для кода и наборов данных МО. Откройте терминал и введите следующие команды (после подсказок \$):

```
$ export ML_PATH="$HOME/ml" # При желании можете изменить этот путь
$ mkdir -p $ML_PATH
```

Вам потребуется несколько модулей Python: Jupyter, NumPy, Pandas, Matplotlib и Scikit-Learn. При наличии установленного инструмента Jupyter, функционирующего со всеми указанными модулями, вы можете спокойно переходить к чтению раздела “Загрузка данных” далее в главе. Если модули пока еще отсутствуют, то есть много способов их установки (вместе с зависимостями). Вы можете использовать систему пакетов своей операционной системы (скажем, apt-get для Ubuntu либо MacPorts или HomeBrew для macOS), установить дистрибутив, ориентированный на научные вычисления с помощью Python, такой как Anaconda, и применять его систему пакетов или просто использовать собственную систему пакетов Python, pip, которая по умолчанию включается в двоичные установщики Python (начиная с версии Python 2.7.9)⁶. Проверить, установлен ли модуль pip, можно посредством следующей команды:

```
$ pip3 --version
pip 9.0.1 from [...] /lib/python3.5/site-packages (python 3.5)
```

Вы обязаны удостовериться в том, что установлена самая последняя версия pip, по меньшей мере, новее 1.4, для поддержки установки двоичных модулей (также известных под названием колеса (wheel)). Вот как обновить модуль pip⁷:

```
$ pip3 install --upgrade pip
Collecting pip
[...]
Successfully installed pip-9.0.1
```

⁶ Шаги установки будут показаны с применением pip в оболочке bash системы Linux или macOS. Вам может понадобиться адаптировать приведенные команды к своей системе. В случае Windows рекомендуется взамен установить дистрибутив Anaconda.

⁷ Для запуска данной команды могут потребоваться права администратора; если это так, тогда снабдите ее префиксом `sudo`.

Создание изолированной среды

Если вы пожелаете работать в изолированной среде (что настоятельно рекомендуется, т.к. вы сможете трудиться над разными проектами без возникновения конфликтов между версиями библиотек), установите инструмент `virtualenv`, запустив следующую команду `pip`:

```
$ pip3 install --user --upgrade virtualenv
Collecting virtualenv
[...]  
Successfully installed virtualenv
```

Далее вы можете создать изолированную среду Python:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix ' [...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

Затем каждый раз, когда вы хотите активизировать такую среду, просто откройте терминал и введите:

```
$ cd $ML_PATH
$ source env/bin/activate
```

Пока изолированная среда активна, в нее будут устанавливаться любые пакеты, которые вы выберете для установки с помощью `pip`, и Python получит доступ только к этим пакетам (если также нужен доступ к пакетам сайта системы, тогда вы должны создавать среду с указанием параметра `system-site-packages` инструмента `virtualenv`). Дополнительные сведения доступны в документации по `virtualenv`.

Теперь вы можете установить все обязательные модули и их зависимости, используя следующую простую команду `pip`:

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
[...]
```

Чтобы проверить успешность установки, попробуйте импортировать каждый модуль:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy, sklearn"
```

Никакого вывода и сообщений об ошибках быть не должно. Далее можете запустить Jupyter:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory: [...]ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at:
  http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server
  and shut down all kernels (twice to skip confirmation).
```

Итак, сервер Jupyter функционирует в вашем терминале, прослушивая порт 8888. Вы можете посетить его, открыв в веб-браузере страницу <http://localhost:8888/> (обычно она открывается автоматически при запуске сервера). Вы должны увидеть пустой каталог своей рабочей области (содержащий единственный каталог `env`, если вы следовали предшествующим инструкциям для `virtualenv`).

Создайте новую тетрадь Python, щелкнув на кнопке **New** (Новая) и выбрав подходящую версию Python⁸ (рис. 2.3).

Такое действие приводит к трем вещам: во-первых, в рабочей области создается новый файл тетради по имени `Untitled.ipynb`; во-вторых, запускается ядро Jupyter Python для выполнения этой тетради; в-третьих, тетрадь открывается в новой вкладке. Вы должны первым делом переименовать тетрадь в `Housing` (файл автоматически переименуется в `Housing.ipynb`), щелкнув на `Untitled` и введя новое имя.

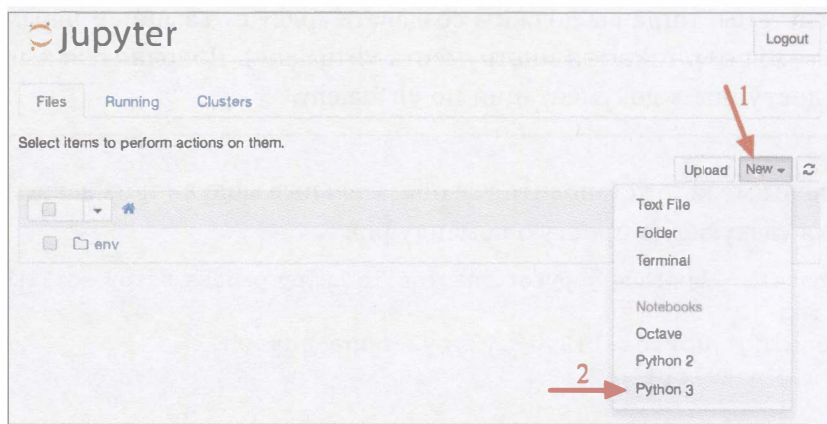


Рис. 2.3. Ваша рабочая область в Jupyter

⁸ Обратите внимание, что Jupyter способен поддерживать несколько версий Python и даже многие другие языки наподобие R или Octave.

Тетрадь содержит список ячеек. Каждая ячейка может вмещать исполняемый код или форматированный текст. Прямо сейчас тетрадь включает только одну пустую ячейку кода, помеченную как `In [1]:`. Наберите в ячейке `print("Hello world!")` и щелкните на кнопке запуска (рис. 2.4) или нажмите `<Shift+Enter>`. Текущая ячейка опрашивается ядром Python этой тетради, которое выполнит ее и возвратит вывод. Результат отображается ниже ячейки, а поскольку мы достигли конца тетради, то автоматически создается новая ячейка. Чтобы изучить основы, выберите пункт `User Interface Tour` (Тип по пользовательскому интерфейсу) в меню `Help` (Справка) инструмента Jupyter.

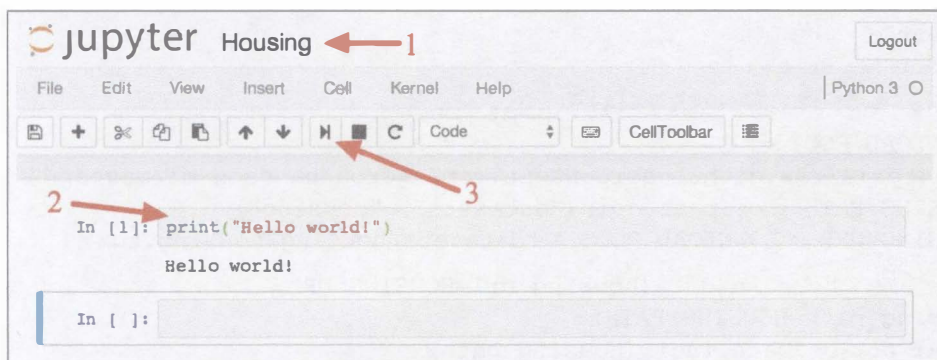


Рис. 2.4. Простейшая тетрадь Python

Загрузка данных

В типовых средах ваши данные будут доступными в реляционной базе данных (или в каком-то другом общем хранилище данных) и разнесенными по множеству таблиц/документов/файлов. Для обращения к ним вам сначала понадобится получить учетные данные и авторизацию доступа⁹, а также ознакомиться со схемой данных. Однако в рассматриваемом проекте ситуация гораздо проще: вы лишь загрузите единственный сжатый файл `housing.tgz`, внутри которого содержится файл в формате разделенных запятыми значений (comma-separated value — CSV) по имени `housing.csv` со всеми данными.

⁹ Может также возникнуть необходимость проверить ограничения правового характера, такие как персональные поля, которые никогда не должны копироваться в ненадежные хранилища данных.

Вы могли бы загрузить файл посредством веб-браузера и запустить команду `tar xzf housing.tgz`, чтобы распаковать файл CSV, но предпочтительнее создать функцию, которая будет делать это. Такой прием особенно удобен, если данные регулярно изменяются, поскольку он позволяет написать небольшой сценарий и запускать его всякий раз, когда нужно извлечь самые последние данные (или можно настроить запланированное задание для автоматического выполнения этой работы через равные промежутки времени). Автоматизация процесса извлечения данных также удобна, когда набор данных необходимо устанавливать на многочисленных машинах.

Ниже показана функция для извлечения данных¹⁰.

```
import os
import tarfile
from six.moves import urllib

DOWNLOAD_ROOT =
    "https://raw.githubusercontent.com/ageron/handson-ml/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL,
    housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Теперь вызов `fetch_housing_data()` приводит к созданию каталога `datasets/housing` в рабочей области, загрузке файла `housing.tgz`, извлечению из него файла `housing.csv` и его помещению в указанный каталог.

Давайте загрузим данные с применением Pandas. Вам снова придется написать небольшую функцию для загрузки данных:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

Эта функция возвращает Pandas-объект `DataFrame`, содержащий все данные.

¹⁰ В реальном проекте вы сохранили бы этот код в файле Python, но пока его можно просто ввести в тетради Jupyter.

Беглый взгляд на структуру данных

Давайте бегло ознакомимся с верхними пятью строками, используя метод `head()` объекта `DataFrame` (рис. 2.5).

```
In [5]: housing = load_housing_data()
housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Рис. 2.5. Верхние пять строк набора данных

Каждая строка представляет один округ. Имеется 10 атрибутов (первые 6 видны на снимке экрана): `longitude`, `latitude`, `housing_median_age`, `total_rooms`, `total_bedrooms`, `population`, `households`, `median_income`, `median_house_value` и `ocean_proximity`.

Метод `info()` полезен для получения краткого описания данных, в частности общего числа строк, а также типа и количества ненулевых значений каждого атрибута (рис. 2.6).

В наборе данных есть 20 640 образцов и потому по стандартам машинного обучения он довольно мал, но идеален для того, чтобы начать.

```
In [6]: housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude                20640 non-null float64
latitude                 20640 non-null float64
housing_median_age       20640 non-null float64
total_rooms              20640 non-null float64
total_bedrooms           20433 non-null float64
population               20640 non-null float64
households               20640 non-null float64
median_income            20640 non-null float64
median_house_value       20640 non-null float64
ocean_proximity          20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

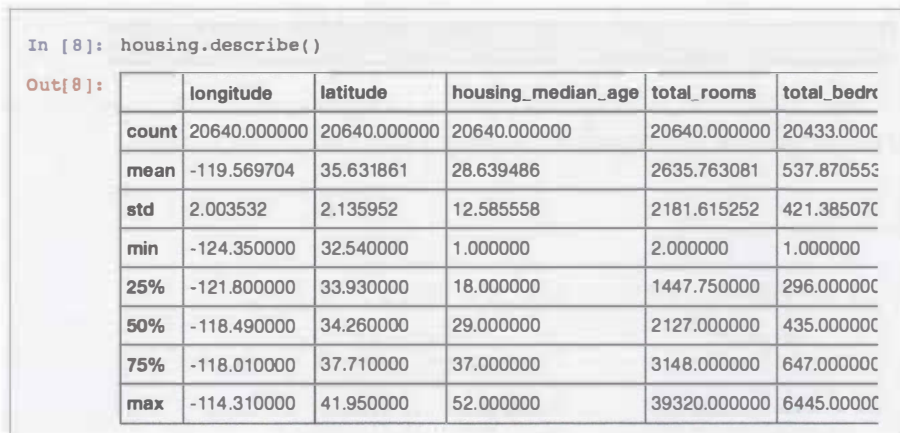
Рис. 2.6. Результаты выполнения метода `info()`

Обратите внимание, что атрибут `total_bedrooms` имеет только 20 433 ненулевых значений, т.е. у 207 округов упомянутый признак отсутствует. Позже нам придется позаботиться об этом.

Все атрибуты являются числовыми кроме поля `ocean_proximity`. Оно имеет тип `object`, поэтому способно содержать объект Python любого вида, но поскольку вы загрузили данные из файла CSV, то знаете, что поле `ocean_proximity` должно быть текстовым атрибутом. Взглянув на верхние пять строк, вы наверняка заметите, что значения в столбце `ocean_proximity` повторяются, т.е. вероятно он представляет собой категориальный атрибут. С применением метода `value_counts()` можно выяснить, какие категории существуют, и сколько округов принадлежит к каждой категории:

```
>>> housing["ocean_proximity"].value_counts()
<1H OCEAN          9136
INLAND             6551
NEAR OCEAN         2658
NEAR BAY           2290
ISLAND              5
Name: ocean_proximity, dtype: int64
```

Давайте посмотрим на остальные поля. Метод `describe()` отображает сводку по числовым атрибутам (рис. 2.7).



```
In [8]: housing.describe()
```

Out[8]:	longitude	latitude	housing_median_age	total_rooms	total_bedrooms
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.385070
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000

Рис. 2.7. Сводка по числовым атрибутам

Строки `count`, `mean`, `min` и `max` не требуют объяснений. Обратите внимание, что нулевые значения проигнорированы (и потому, скажем, `count` для `total_bedrooms` составляет 20 433, а не 20 640). В строке `std` показано *стандартное отклонение* (*standard deviation*), которое измеряет разброс зна-

чений¹¹. В строках 25%, 50% и 75% приведены соответствующие *процентили* (*percentile*): процентиль указывает значение, ниже которого падает заданный процент наблюдений в группе замеров. Например, 25% округов имеют средний возраст домов (*housing_median_age*) ниже 18, в то время как в 50% округов средний возраст домов ниже 29 и в 75% округов — ниже 37. Такие показатели часто называют 25-м процентилем (или 1-м *квартилем* (*quartile*)), медианой и 75-м процентилем (или 3-м *квартилем*).

Другой быстрый способ получить представление о данных, с которыми вы имеете дело, предусматривает вычерчивание гистограммы для каждого числового атрибута. Гистограмма показывает количество образцов (по вертикальной оси), которые имеют заданный диапазон значений (по горизонтальной оси). Вы можете вычерчивать гистограмму для одного атрибута за раз или вызвать метод `hist()` на целом наборе данных, и он вычертит гистограммы для всех числовых атрибутов (рис. 2.8).

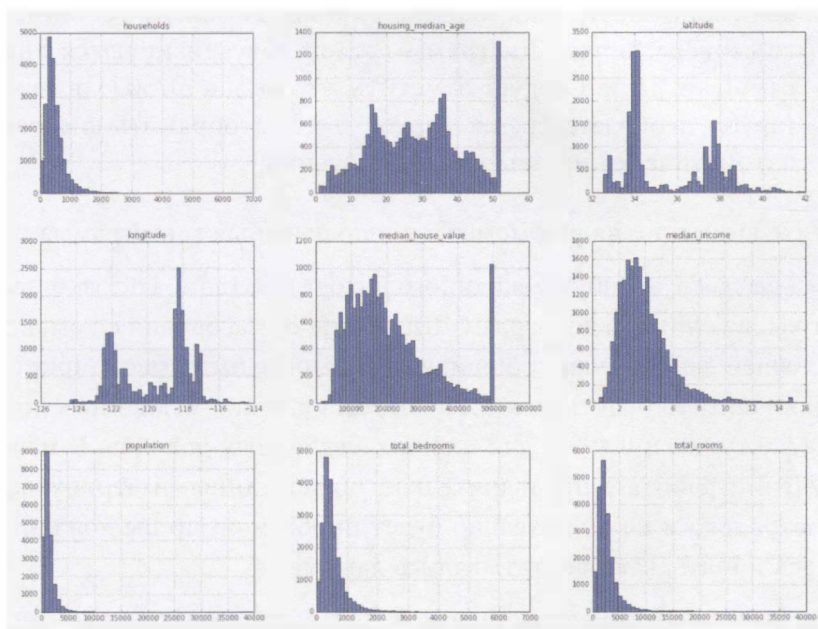


Рис. 2.8. Гистограммы для числовых атрибутов

¹¹ Стандартное отклонение обычно обозначается как σ (греческая буква “сигма”) и представляет собой квадратный корень из *дисперсии* (*variance*), которая является усреднением квадратичного отклонения от среднего значения. Когда признак имеет колоколообразное *нормальное распределение* (также называемое *распределением Гаусса*), что очень распространено, то применяется правило “68-95-99.7”: около 68% значений находятся внутри 1σ среднего, 95% — внутри 2σ и 99.7% — внутри 3σ .

Например, вы можете заметить, что чуть более 800 округов имеют значение `median_house_value` равное приблизительно \$100 000.

```
%matplotlib inline # только в тетради Jupyter
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Метод `hist()` полагается на библиотеку `Matplotlib`, а та в свою очередь — на указанный пользователем графический сервер для рисования на экране. Таким образом, прежде чем можно будет что-то вычерчивать, вам нужно указать, какой сервер библиотека `Matplotlib` должна использовать. Простейший вариант — применить магическую команду (magic command; встречается также вариант “волшебная команда” — *примеч. пер.*) `%matplotlib inline` в `Jupyter`. Она сообщает `Jupyter` о необходимости настройки `Matplotlib` на использование собственного сервера `Jupyter`. Диаграммы затем визуализируются внутри самой тетради. Следует отметить, что вызов `show()` в тетради `Jupyter` необязателен, т.к. `Jupyter` будет отображать диаграммы автоматически при выполнении ячейки.

Обратите внимание на ряд моментов в полученных гистограммах.

1. Прежде всего, атрибут медианного дохода (`median_income`) не выглядит выраженным в долларах США. После согласования с командой, собиравшей данные, вам сообщили, что данные были масштабированы и ограничены 15 (фактически 15.0001) для высших медианных доходов и 0.5 (фактически 0.4999) для низших медианных доходов. В машинном обучении работа с предварительно обработанными атрибутами распространена и не обязательно будет проблемой, но вы должны попытаться понять, как рассчитывались данные.
2. Атрибуты среднего возраста домов (`housing_median_age`) и средней стоимости домов (`median_house_value`) также были ограничены. Последний может стать серьезной проблемой, поскольку он является вашим целевым атрибутом (метками). Ваши алгоритмы МО могут узнать, что цены никогда не выходят за эти пределы. Вам понадобится выяснить у клиентской команды (команды, которая будет потреблять вывод вашей системы), будет это проблемой или нет. Если они сооб-

щают, что нуждаются в точных прогнозах даже за рамками \$500 000, тогда у вас по большому счету есть два варианта.

- а) Собрать надлежащие метки для округов, чьи метки были ограничены.
 - б) Удалить такие округа из обучающего набора (и также из испытательного набора, потому что ваша система не должна плохо оцениваться, если она прогнозирует цены за пределами \$500 000).
3. Атрибуты имеют очень разные масштабы. Мы обсудим данный вопрос позже в главе, когда будем исследовать масштабирование признаков.
4. Наконец, многие гистограммы имеют *медленно убывающие хвосты* (*tail heavy*): они простираются гораздо дальше вправо от медианы, чем влево. Это может несколько затруднить некоторым алгоритмам МО выявлять паттерны. Позже мы попытаемся трансформировать такие атрибуты, чтобы получить более колоколообразные распределения.

Будем надеяться, что теперь вы лучше понимаете природу данных, с которыми имеете дело.



Подождите! До того как продолжить дальнейшее рассмотрение данных, вы должны создать испытательный набор, отложить его и больше никогда в него заглядывать.

Создание испытательного набора

Рекомендация добровольно отложить часть данных в сторону на этой стадии может показаться странной. В конце концов, вы лишь мельком взглянули на данные и непременно должны узнать о них намного больше, прежде чем решить, какие алгоритмы применять, не так ли? Это верно, но ваш мозг является удивительной системой обнаружения паттернов, что означает его крайнюю predisposedность к переобучению: взглянув на испытательный набор, вы можете натолкнуться на какой-то с виду интересный паттерн в тестовых данных, который приведет к выбору вами определенного типа модели МО. Когда вы оцениваете ошибку обобщения, используя испытательный набор, ваша оценка будет слишком оптимистичной и приведет к выпуску системы, которая не работает настолько хорошо, насколько ожидалось. Это называется смещением *из-за информационного просмотра данных* (*data snooping bias*).

Создать испытательный набор теоретически довольно просто: нужно лишь произвольно выбрать некоторые образцы, обычно 20% набора данных, и отложить их:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

Функцию `split_train_test()` затем можно применять следующим образом:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

Да, все работает, но далеко от совершенства: если вы запустите программу еще раз, то она сгенерирует отличающийся испытательный набор! Со временем вы (или ваши алгоритмы МО) увидят полный набор данных, чего вы хотите избежать.

Одно из решений заключается в сохранении испытательного набора после первого запуска и его загрузка при последующих запусках. Другой вариант предусматривает установку начального значения генератора случайных чисел (например, `np.random.seed(42)`)¹² до вызова `np.random.permutation()`, чтобы он всегда генерировал те же самые перетасованные индексы.

Но оба решения перестанут работать, когда вы в следующий раз извлечете обновленный набор данных. Распространенное решение предполагает использование идентификатора каждого образца при решении, должен ли он быть помещен в испытательный набор (при условии, что образцы имеют уникальные и неизменяемые идентификаторы). Например, вы могли бы вычислять хеш идентификатора каждого образца, оставлять только последний байт хеша и помещать образец в испытательный набор, если значение

¹² Вы часто будете видеть, как люди устанавливают 42 в качестве начального значения. Это число не обладает никакими особыми свойствами помимо того, что служит ответом на “главный вопрос жизни, вселенной и всего такого” (https://ru.wikipedia.org/wiki/Ответ_на_главный_вопрос_жизни,_вселенной_и_всего_такого — *примеч. пер.*).

последнего байта хеша меньше или равно 51 (приблизительно 20% из 256). Такой подход гарантирует, что испытательный набор будет сохраняться согласованным между множеством запусков, даже если набор данных обновляется. Новый испытательный набор будет содержать 20% новых образцов, но не будет включать образцы, которые присутствовали в испытательном наборе ранее. Вот возможная реализация:

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column,
                           hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_:
        test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

К сожалению, набор данных `housing` не имеет столбца для идентификатора. Простейшее решение предусматривает применение индекса строки в качестве идентификатора:

```
housing_with_id = housing.reset_index() # добавляет столбец `index`
train_set, test_set =
    split_train_test_by_id(housing_with_id, 0.2, "index")
```

Если вы используете индекс строки как уникальный идентификатор, то должны удостовериться в том, что новые данные добавляются в конец набора данных и никакие строки из набора не удаляются. В ситуации, когда добиться этого невозможно, вы можете попробовать применять для построения уникального идентификатора самые стабильные признаки. Скажем, широта и долгота округа гарантированно будут стабильными на протяжении нескольких миллионов лет, так что вы могли бы скомбинировать их в идентификатор¹³:

```
housing_with_id["id"] =
    housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set =
    split_train_test_by_id(housing_with_id, 0.2, "id")
```

¹³ На самом деле информация о местоположении является довольно грубой, в результате чего многие округа будут иметь в точности один и тот же идентификатор, а потому окажутся в том же самом наборе (испытательном или обучающем). Это привносит неуместное смещение выборки.

Модуль Scikit-Learn предлагает несколько функций для разделения наборов данных на множество поднаборов разнообразными способами. Простейшая из них, `train_test_split()`, делает практически то же, что и определенная ранее функция `split_train_test()`, с парой дополнительных особенностей. Во-первых, она принимает параметр `random_state`, который позволяет устанавливать начальное значение генератора случайных чисел, как объяснялось ранее. Во-вторых, функции `train_test_split()` можно передавать несколько наборов данных с одинаковыми количествами строк и она разобьет их по тем же самым индексам (что очень удобно, к примеру, при наличии отдельного объекта `DataFrame` для меток):

```
from sklearn.model_selection import train_test_split
train_set, test_set =
    train_test_split(housing, test_size=0.2, random_state=42)
```

До сих пор мы рассматривали чисто случайные методы выборки. Как правило, они хороши, если набор данных достаточно большой (в особенности относительно количества атрибутов), но когда это не так, возникает риск внесения значительного смещения выборки. Если специалисты в исследовательской компании решают обзвонить 1000 людей, чтобы задать им несколько вопросов, то они не просто произвольно выбирают 1000 человек из телефонного справочника. Они стараются гарантировать, что такая группа из 1000 людей является репрезентативной для всего населения. Например, население США состоит из 51.3% женщин и 48.7% мужчин, так что при хорошо организованном анкетировании в США попытаются соблюсти такое соотношение в выборке: 513 женщин и 487 мужчин. Это называется *стратифицированной выборкой* (*stratified sampling*): население делится на однородные подгруппы, называемые *стратами* (*strata*), и из каждой страты извлекается правильное число образцов, чтобы гарантировать репрезентативность испытательного набора для всего населения. При использовании чисто случайной выборки был бы примерно 12%-ный шанс получить скошенный испытательный набор, содержащий менее чем 49% женщин или более чем 54% женщин. В обеих ситуациях результаты исследования оказались бы значительно смещенными.

Пусть вы побеседовали с экспертами, которые сказали вам, что медианный доход является очень важным атрибутом для прогнозирования средней стоимости домов. У вас может возникнуть желание обеспечить репрезентативность испытательного набора для различных категорий дохода в целом наборе данных. Поскольку медианный доход представляет собой непрерывный

числовой атрибут, сначала понадобится создать атрибут категории дохода. Давайте рассмотрим гистограмму для медианного дохода (`median_income`) более внимательно (см. рис. 2.8): большинство значений медианного дохода сгруппированы около \$20 000–\$50 000, но некоторые медианные доходы выходят далеко за пределы \$60 000. В наборе данных важно иметь достаточное количество образцов для каждой страты, иначе оценка важности страты может быть смещена. Другими словами, вы не должны иметь слишком много страт, а каждая страта должна быть достаточно крупной. Приведенный ниже код создает атрибут категории дохода (`income_cat`) путем деления медианного дохода на 1.5 (чтобы ограничить число категорий дохода), округления результата с применением `ceil()` (чтобы получить дискретные категории) и затем объединения всех категорий больше 5 в категорию 5:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0,
                           inplace=True)
```

Категории дохода представлены на рис. 2.9.

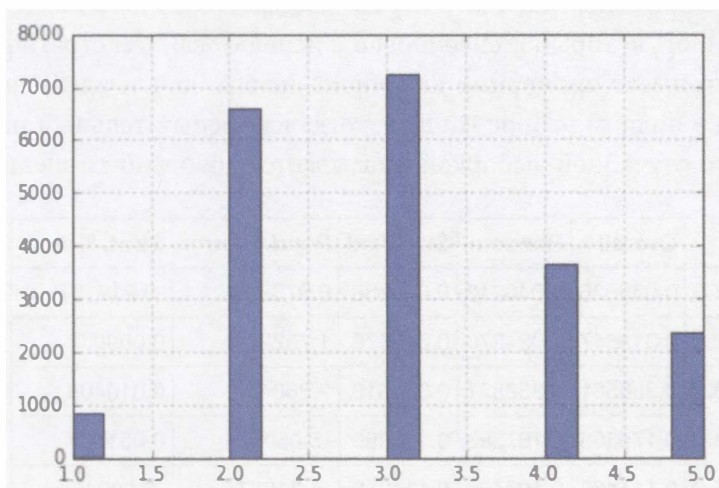


Рис. 2.9. Гистограмма категорий дохода

Теперь вы готовы делать стратифицированную выборку на основе категории дохода. Для этого вы можете использовать класс `StratifiedShuffleSplit` из Scikit-Learn:

```
from sklearn.model_selection import StratifiedShuffleSplit
split = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
                               random_state=42)
```

```
for train_index, test_index in split.split(housing,
                                           housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

Выясним, работает ли код ожидаемым образом. Вы можете начать с просмотра пропорций категорий дохода в полном наборе данных `housing`:

```
>>> housing["income_cat"].value_counts() / len(housing)
3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
```

С помощью похожего кода вы можете измерить пропорции категорий дохода в испытательном наборе. На рис. 2.10 сравниваются пропорции категорий дохода в полном наборе данных, в испытательном наборе, который сгенерирован посредством стратифицированной выборки, и в испытательном наборе, полученном с применением чисто случайной выборки. Здесь видно, что испытательный набор, который сгенерирован с использованием стратифицированной выборки, имеет пропорции категорий дохода, почти идентичные таким пропорциям в полном наборе данных, тогда как испытательный набор, полученный чисто случайной выборкой, оказывается довольно скошенным.

	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

Рис. 2.10. Сравнение смещения выборки стратифицированной и чисто случайной выборки

Теперь вы должны удалить атрибут `income_cat`, чтобы вернуть данные обратно в первоначальное состояние:

```
for set_in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

Мы потратили порядочно времени на генерацию испытательного набора по уважительной причине: это часто игнорируемая, но критически важная часть проекта МО. Более того, многие из представленных идей будут полезны позже, когда мы начнем обсуждать перекрестную проверку. Самое время переходить к следующей стадии: исследованию данных.

Обнаружение и визуализация данных для понимания их сущности

Пока что вы только мельком взглянули на данные, чтобы обрести общее понимание типа данных, которыми предстоит манипулировать. Цель теперь — продвинуться немного глубже.

Прежде всего, удостоверьтесь в том, что отложили испытательный набор в сторону и занимаетесь исследованием только обучающего набора. К тому же если обучающий набор очень крупный, тогда у вас может появиться желание выполнять выборку из исследуемого набора, чтобы сделать манипуляции легкими и быстрыми. В нашем случае набор довольно мал, поэтому можно просто работать напрямую с полным набором. Давайте создадим копию, чтобы вы могли попрактиковаться с ней, не причиняя вреда обучающему набору:

```
housing = strat_train_set.copy()
```

Визуализация географических данных

Поскольку есть географическая информация (широта и долгота), хорошей мыслью будет создать график рассеяния всех округов для визуализации данных (рис. 2.11):

```
housing.plot(kind="scatter", x="longitude", y="latitude")
```

Очертания вполне напоминают Калифорнию, но трудно разглядеть какой-то конкретный паттерн. Установка параметра `alpha` в `0.1` значительно облегчает обнаружение мест, где имеется высокая плотность точек данных (рис. 2.12):

```
housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)
```

Теперь стало гораздо лучше: вы можете четко видеть области с высокой плотностью, а именно — область залива Сан-Франциско и поблизости Лос-Анджелеса и Сан-Диего плюс длинная линия довольно высокой плотности в Калифорнийской долине, в частности около Сакраменто и Фресно.

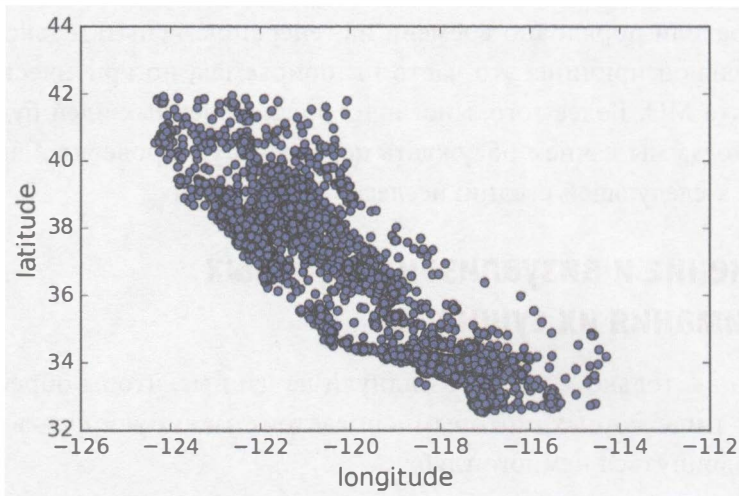


Рис. 2.11. График рассеяния географических данных

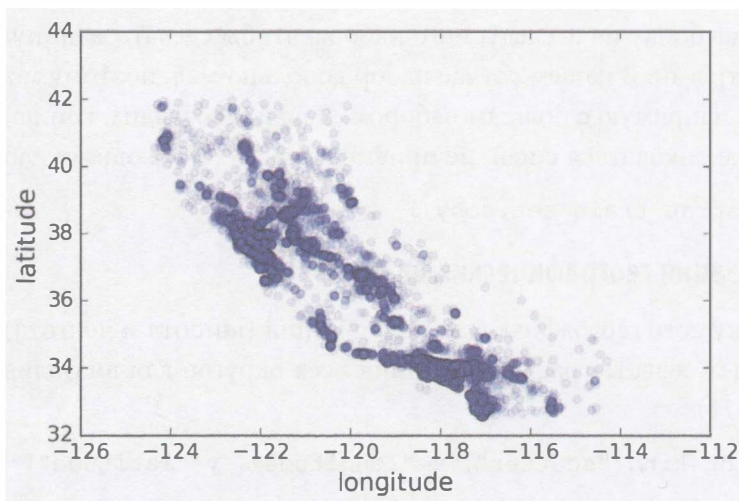


Рис. 2.12. Улучшенная визуализация выделяет области с высокой плотностью

Вообще говоря, наши мозги очень хороши в выявлении паттернов на рисунках, но может возникнуть необходимость поэкспериментировать с параметрами визуализации, чтобы сделать паттерны более заметными.

Обратимся к ценам на дома (рис. 2.13). Радиус каждого круга представляет население округа (параметр s), а цвет — цену (параметр c).

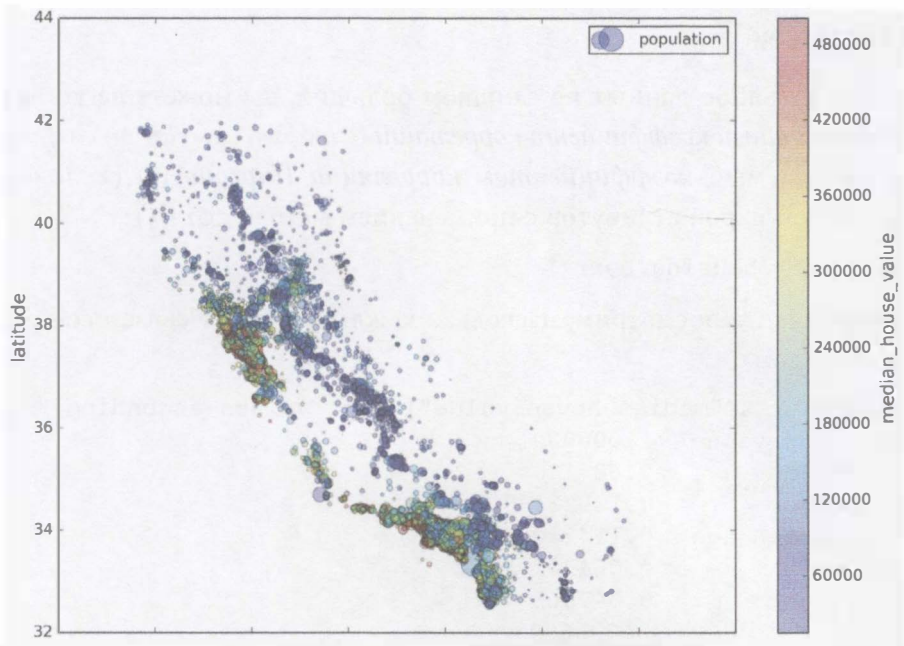


Рис. 2.13. Цены на дома в Калифорнии

Мы будем применять предварительно определенную карту цветов (параметр `cmap`) по имени `jet`, которая простирается от синего цвета (низкие цены) до красного (высокие цены)¹⁴:

```
housing.plot(kind="scatter", x="longitude", y="latitude",
             alpha=0.4, s=housing["population"]/100, label="population",
             figsize=(10,7), c="median_house_value",
             cmap=plt.get_cmap("jet"), colorbar=True,
             )
plt.legend()
```

Изображение говорит о том, что цены на дома очень сильно зависят от местоположения (например, близко к океану) и плотности населения, о чем вам вероятно уже известно. Возможно, будет полезно воспользоваться каким-то алгоритмом кластеризации, чтобы выявить главные кластеры, и добавить новые признаки, которые измеряют близость к центрам кластеров. Атрибут близости к океану также может быть практичен, хотя в северной Калифорнии цены на дома в прибрежных округах не слишком высоки, так что это не является простым правилом.

¹⁴ Если вы просматриваете это изображение в оттенках серого, тогда возьмите красный фломастер и пометьте им большую часть береговой линии от области залива вниз до Сан-Диего (что вы и могли ожидать). Можете также добавить небольшой участок желтого около Сакраменто.

Поиск связей

Поскольку набор данных не слишком большой, вы можете легко вычислить *стандартный коэффициент корреляции* (*standard correlation coefficient*), также называемый *коэффициентом корреляции Пирсона* (r) (*Pearson's r*), между каждой парой атрибутов с применением метода `corr()`:

```
corr_matrix = housing.corr()
```

Теперь давайте посмотрим, насколько каждый атрибут связан со средней стоимостью дома:

```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.687170
total_rooms           0.135231
housing_median_age    0.114220
households            0.064702
total_bedrooms        0.047865
population            -0.026699
longitude             -0.047279
latitude              -0.142826
Name: median_house_value, dtype: float64
```

Коэффициент корреляции колеблется от -1 до 1 . Когда он близок к 1 , это означает наличие сильной положительной корреляции; например, средняя стоимость дома имеет тенденцию увеличиваться с ростом медианного дохода. Когда коэффициент корреляции близок к -1 , тогда существует сильная отрицательная корреляция; вы можете видеть небольшую отрицательную корреляцию между широтой и средней стоимостью дома (т.е. цены имеют незначительную тенденцию к снижению при движении на север). Наконец, коэффициенты, близкие к нулю, означают отсутствие линейной корреляции. На рис. 2.14 приведены разнообразные графики наряду с коэффициентом корреляции между их горизонтальными и вертикальными осями.



Коэффициент корреляции измеряет только линейную корреляцию (“если x повышается, тогда y обычно повышается/понижается”). Он может совершенно упустить из виду нелинейные связи (например, “если x близко к нулю, тогда y обычно повышается”). Обратите внимание, что все графики в нижнем ряду имеют коэффициент корреляции, равный нулю, несмотря на очевидный факт, что их оси не являются независимыми: они представляют примеры нелинейных связей.

Кроме того, во втором ряду показаны примеры, где коэффициент корреляции равен 1 или -1; как видите, он совершенно не соотносится с наклоном. К примеру, ваш рост в сантиметрах имеет коэффициент корреляции 1 с вашим ростом в метрах или в нанометрах.

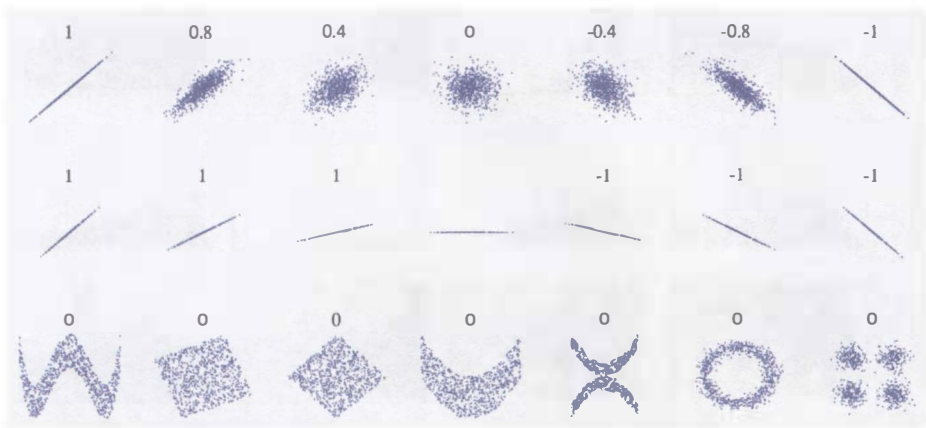


Рис. 2.14. Стандартный коэффициент корреляции различных наборов данных (Википедия: источники изображений)

Еще один способ проверки корреляции между атрибутами предусматривает использование Pandas-функции `scatter_matrix()`, которая вычерчивает каждый числовой атрибут по отношению к каждому другому числовому атрибуту. Так как в настоящий момент есть 11 числовых атрибутов, вы получили бы $11^2 = 121$ график, что не уместилось бы на печатной странице, а потому давайте сосредоточимся лишь на нескольких многообещающих атрибутах, которые представляются наиболее связанными со средней стоимостью дома (рис. 2.15):

```
from pandas.plotting import scatter_matrix
attributes = ["median_house_value", "median_income", "total_rooms",
             "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```

Если бы модуль Pandas вычерчивал каждую переменную по отношению к самой себе, то главная диагональ (с левого верхнего угла в правый нижний угол) была бы переполнена прямыми линиями, что не особенно полезно. Таким образом, Pandas взамен отображает гистограмму каждого атрибута (доступны другие варианты; за дополнительной информацией обращайтесь в документацию Pandas).

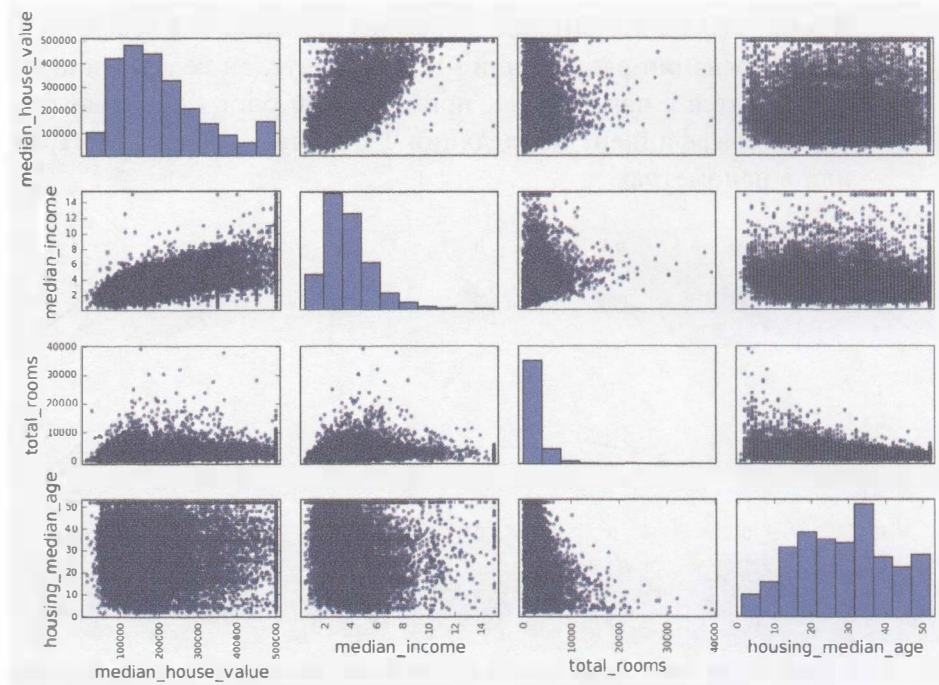


Рис. 2.15. Матрица рассеяния

Самый многообещающим атрибутом для прогнозирования средней стоимости дома является медианный доход, поэтому давайте увеличим его график рассеяния корреляции (рис. 2.16):

```
housing.plot(kind="scatter", x="median_income",
              y="median_house_value", alpha=0.1)
```

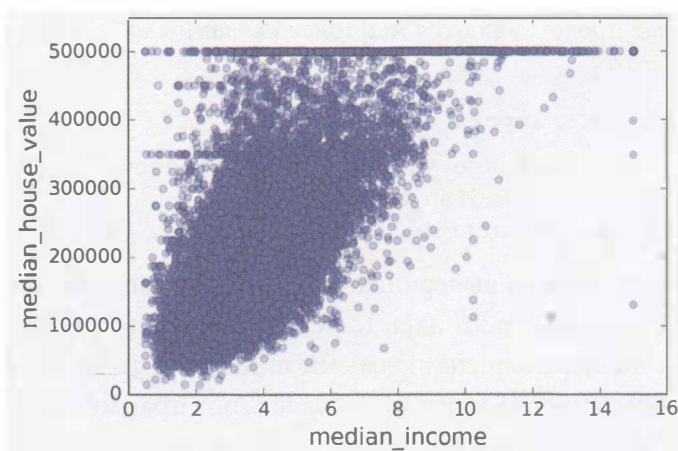


Рис. 2.16. Медианный доход в сравнении со средней стоимостью дома

График выявляет несколько фактов. Во-первых, корреляция действительно очень сильная; вы ясно видите тенденцию к повышению и точки не слишком рассеяны. Во-вторых, граница цены, которую мы отметили ранее, четко видна как горизонтальная линия напротив значения \$500 000. Но график раскрывает и другие менее очевидные прямые линии: горизонтальную линию вблизи \$450 000, еще одну возле \$350 000, возможно горизонтальную линию около \$280 000 и несколько линий ниже. Вы можете попробовать удалить соответствующие окружности, чтобы ваши алгоритмы не научились воспроизводить такие индивидуальные особенности данных.

Экспериментирование с комбинациями атрибутов

Будем надеяться, что предшествующие разделы дали вам представление о нескольких способах, которыми можно исследовать данные и понять их сущность. Вы идентифицировали несколько индивидуальных особенностей данных, от которых при желании можно избавиться перед тем, как передавать данные алгоритму машинного обучения, и нашли интересные связи между атрибутами, в частности с целевым атрибутом. Вы также заметили, что некоторые атрибуты имеют распределение с медленно убывающим хвостом, поэтому такие атрибуты может понадобиться трансформировать (скажем, за счет вычисления их логарифма). Конечно, степень вашего продвижения будет значительно варьироваться от проекта к проекту, но общие идеи похожи.

Последнее, что вы можете захотеть сделать перед фактической подготовкой данных для алгоритмов МО — опробовать разнообразные комбинации атрибутов. Например, общее количество комнат в округе не особенно полезно, если вы не знаете, сколько есть домов. То, что вам действительно необходимо — это количество комнат на дом. Аналогично общее количество спален само по себе не очень полезно: возможно, его следовало бы сравнить с количеством комнат. Население на дом также представляется интересной для рассмотрения комбинацией атрибутов. Давайте создадим упомянутые новые атрибуты:

```
housing["rooms_per_household"] =  
    housing["total_rooms"]/housing["households"]  
housing["bedrooms_per_room"] =  
    housing["total_bedrooms"]/housing["total_rooms"]  
housing["population_per_household"] =  
    housing["population"]/housing["households"]
```


А теперь снова взглянем на матрицу корреляции:

```
>>> corr_matrix = housing.corr()
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value      1.000000
median_income           0.687160
rooms_per_household     0.146285
total_rooms             0.135097
housing_median_age      0.114110
households              0.064506
total_bedrooms          0.047689
population_per_household -0.021985
population              -0.026920
longitude               -0.047432
latitude                -0.142724
bedrooms_per_room       -0.259984
Name: median_house_value, dtype: float64
```

Эй, неплохо! Новый атрибут `bedrooms_per_room` (количество спален на количество комнат) намного больше связан со средней стоимостью дома, чем общее количество комнат или спален. По всей видимости, дома с меньшим соотношением спальни/комнаты имеют тенденцию быть более дорогостоящими. Количество комнат на дом также более информативно, нежели общее число комнат в округе — очевидно, чем крупнее дома, тем они дороже.

Данный раунд исследований вовсе не обязан быть абсолютно исчерпывающим; вопрос в том, чтобы начать все как следует и быстро ухватить суть, что поможет получить первый достаточно хороший прототип. Но это итеративный процесс: после того, как прототип готов, вы можете проанализировать его вывод, чтобы обрести еще лучшее понимание, и снова возвратиться к шагу исследования.

Подготовка данных для алгоритмов машинного обучения

Наступило время подготовки данных для ваших алгоритмов машинного обучения. Вместо того чтобы просто выполнить подготовку вручную, вы должны написать для нее функции по ряду веских причин.

- Появится возможность легко воспроизводить эти трансформации на любом наборе данных (скажем, при очередном получении свежего набора данных).

- Вы будете постепенно строить библиотеку функций трансформации, которые можно многократно применять в будущих проектах.
- Готовые функции можно использовать в действующей системе, чтобы трансформировать новые данные перед передачей в алгоритмы.
- Станет возможным опробование разнообразных трансформаций в целях выяснения, какие сочетания трансформаций работают лучше всего.

Но сначала давайте вернемся к чистому обучающему набору (еще раз скопировав `strat_train_set`), а также разделим прогнозаторы и метки, поскольку мы не обязательно хотим применять те же самые трансформации к прогнозаторам и целевым значениям (обратите внимание, что `drop()` создает копию данных и не влияет на `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

Очистка данных

Большинство алгоритмов МО не способны работать с недостающими признаками, поэтому мы создадим несколько функций, которые позаботятся о них. Ранее вы видели, что в атрибуте `total_bedrooms` не хватало ряда значений, так что давайте исправим положение. Есть три варианта:

- избавиться от соответствующих округов;
- избавиться от всего атрибута;
- установить недостающие значения в некоторую величину (ноль, среднее, медиана и т.д.).

Все варианты легко реализуются с использованием методов `dropna()`, `drop()` и `fillna()` объекта `DataFrame`:

```
housing.dropna(subset=["total_bedrooms"])           # вариант 1
housing.drop("total_bedrooms", axis=1)              # вариант 2
median = housing["total_bedrooms"].median()         # вариант 3
housing["total_bedrooms"].fillna(median, inplace=True)
```

Если вы выбрали вариант 3, то должны подсчитать медиану обучающего набора и применять ее для значений, отсутствующих в обучающем наборе, но также не забывайте сохранить вычисленную медиану. Она вам будет нужна позже для замены недостающих значений в испытательном наборе, когда понадобится оценивать систему, и также после ввода систему в эксплуатацию для замены недостающих значений в новых данных.

Модуль Scikit-Learn предлагает удобный класс, который заботится об отсутствующих значениях: `Imputer`. Вот как им пользоваться. Первым делом необходимо создать экземпляр `Imputer`, указывая на то, что недостающие значения каждого атрибута следует заменять медианой этого атрибута:

```
from sklearn.preprocessing import Imputer
imputer = Imputer(strategy="median")
```

Так как медиана может подсчитываться только на числовых атрибутах, нам требуется создать копию данных без текстового атрибута `ocean_proximity`:

```
housing_num = housing.drop("ocean_proximity", axis=1)
```

Теперь экземпляр `imputer` можно подогнать к обучающим данным с применением метода `fit()`:

```
imputer.fit(housing_num)
```

Экземпляр `imputer` просто подсчитывает медиану каждого атрибута и сохраняет результат в своей переменной экземпляра `statistics_`. Некоторые значения отсутствуют только у атрибута `total_bedrooms`, но мы не можем иметь уверенность в том, что после начала эксплуатации системы подобное не случится с новыми данными, а потому надежнее применить `imputer` ко всем числовым атрибутам:

```
>>> imputer.statistics_
array([ -118.51 ,  34.26 ,  29. , 2119.5 ,  433. , 1164. ,  408. ,  3.5409])
>>> housing_num.median().values
array([ -118.51 ,  34.26 ,  29. , 2119.5 ,  433. , 1164. ,  408. ,  3.5409])
```

Теперь вы можете использовать этот “обученный” экземпляр `imputer` для трансформации обучающего набора путем замены недостающих значений известными медианами:

```
X = imputer.transform(housing_num)
```

Результатом является обыкновенный массив NumPy, содержащий трансформированные признаки. Если его нужно поместить обратно в Pandas-объект `DataFrame`, то это просто:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
```

Проект Scikit-Learn

API-интерфейс Scikit-Learn необыкновенно хорошо спроектирован. Ниже перечислены главные проектные принципы¹⁵.

- **Согласованность.** Все объекты разделяют согласованный и простой интерфейс.
 - **Оценщики** (*estimator*). Любой объект, который может проводить оценку параметров на основе набора данных, называется оценщиком (например, `imputer` является оценщиком). Сама оценка производится с помощью метода `fit()`, принимающего в качестве параметра единственный набор данных (или два для алгоритмов обучения с учителем; второй набор данных содержит метки). Любой другой параметр, необходимый для управления процессом оценки, считается гиперпараметром (вроде `strategy` в `imputer`) и должен быть указан как переменная экземпляра (обычно через параметр конструктора).
 - **Трансформаторы** (*transformer*). Некоторые оценщики (такие как `imputer`) могут также трансформировать набор данных; они называются трансформаторами. И снова API-интерфейс довольно прост: трансформация выполняется методом `transform()`, которому в параметре передается набор данных, подлежащий трансформации. Он возвращает трансформированный набор данных. Трансформация обычно полагается на изученные параметры, как в случае `imputer`. Все трансформаторы также имеют удобный метод по имени `fit_transform()`, который представляет собой эквивалент вызова `fit()` и затем `transform()` (но благодаря оптимизации метод `fit_transform()` временами выполняется намного быстрее).
 - **Прогнозаторы** (*predictor*). Наконец, некоторые оценщики способны вырабатывать прогнозы, имея набор данных; они называются прогнозаторами.

¹⁵ За дополнительными сведениями по проектным принципам обращайтесь к работе “API design for machine learning software: experiences from the scikit-learn project” (“Проектирование API-интерфейса для ПО машинного обучения: опыт проекта scikit-learn”) (<https://arxiv.org/pdf/1309.0238v1.pdf>).

Например, модель `LinearRegression` в предыдущей главе была прогнозатором: она прогнозировала уровень удовлетворенности жизнью при заданном ВВП на душу населения в стране. Прогнозатор располагает методом `predict()`, который принимает набор данных с новыми образцами и возвращает набор данных с соответствующими прогнозами. Прогнозатор также имеет метод `score()`, измеряющий качество прогнозов с помощью указанного испытательного набора (и соответствующих меток в случае алгоритмов обучения с учителем)¹⁶.

- **Инспектирование.** Все гиперпараметры прогнозаторов доступны напрямую через переменные экземпляра (например, `imputer.strategy`), и все изученные параметры прогнозаторов также доступны через открытые переменные экземпляра с суффиксом в виде подчеркивания (например, `imputer.statistics_`).
- **Нераспространение классов.** Наборы данных представляются как массивы NumPy или разреженные матрицы SciPy вместо самодельных классов. Гиперпараметры — это просто обычные строки или числа Python.
- **Композиция.** Существующие строительные блоки максимально возможно используются повторно. Например, как будет показано далее, из произвольной последовательности трансформаторов легко создать прогнозатор `Pipeline`, за которым находится финальный прогнозатор.
- **Разумные стандартные значения.** Scikit-Learn предоставляет обобщенные стандартные значения для большинства параметров, облегчая быстрое создание базовой рабочей системы.

Обработка текстовых и категориальных атрибутов

Ранее мы пропустили категориальный атрибут `ocean_proximity`, потому что он был текстовым атрибутом, поэтому вычислить его медиану невозможно:

¹⁶ Некоторые прогнозаторы также предоставляют методы для измерения степени достоверности своих прогнозов.

```
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat.head(10)
17606    <1H OCEAN
18632    <1H OCEAN
14650    NEAR OCEAN
3230      INLAND
3555     <1H OCEAN
19480      INLAND
8879     <1H OCEAN
13685      INLAND
4937     <1H OCEAN
4861     <1H OCEAN
Name: ocean_proximity, dtype: object
```

Как бы то ни было, большинство алгоритмов МО предпочитают работать с числами, так что давайте преобразуем категории из текста в числа. Для этого можно применить Pandas-метод `factorize()`, который сопоставляет каждую категорию с отличающимся целым числом:

```
>>> housing_cat_encoded, housing_categories = housing_cat.factorize()
>>> housing_cat_encoded[:10]
array([0, 0, 1, 2, 0, 2, 0, 2, 0, 0])
```

Ситуация улучшилась: атрибут `housing_cat_encoded` теперь исключительно числовой. Метод `factorize()` также возвращает список категорий ("`<1H OCEAN`" была сопоставлена с 0, "`NEAR OCEAN`" — с 1 и т.д.):

```
>>> housing_categories
Index(['<1H OCEAN', 'NEAR OCEAN', 'INLAND', 'NEAR BAY', 'ISLAND'],
      dtype='object')
```

С таким представлением связана одна проблема: алгоритмы МО будут предполагать, что два соседних значения более похожи, чем два отдаленных значения. Очевидно, что это не так (скажем, категории 0 и 4 более похожи, чем категории 0 и 2). Распространенное решение по исправлению проблемы предусматривает создание одного двоичного атрибута на категорию: один атрибут равен 1, когда категорией является "`<1H OCEAN`" (и 0 в противном случае), другой атрибут равен 1, когда категория представляет собой "`NEAR OCEAN`" (и 0 в противном случае), и т.д. Такой прием называется *кодированием с одним активным состоянием* или *унитарным кодированием* (*one-hot encoding*), поскольку только один атрибут будет равен 1 (активный), в то время как остальные — 0 (пассивный).

Модуль Scikit-Learn предлагает кодировщик `OneHotEncoder` для преобразования целочисленных категориальных значений в векторы в унитарном коде.

Закодируем категории как векторы в унитарном коде:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> encoder = OneHotEncoder()
>>> housing_cat_lhot =
    encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
>>> housing_cat_lhot
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
    with 16512 stored elements in Compressed Sparse Row format>
```

Обратите внимание, что метод `fit_transform()` ожидает двумерный массив, но `housing_cat_encoded` — одномерный массив, а потому его форму необходимо изменить¹⁷. Кроме того, выводом является *разреженная матрица* SciPy, а не массив NumPy. Это очень удобно, когда имеются категориальные атрибуты с тысячами категорий. После унитарного кодирования получается матрица, содержащая тысячи столбцов, которая полна нулей за исключением единственной единицы на строку. Расходование массы памяти для представления преимущественно нулей было бы расточительством, так что взамен разреженная матрица хранит только позиции ненулевых элементов. Ее можно использовать во многом подобно нормальному двумерному массиву¹⁸ но если вы действительно хотите преобразовать разреженную матрицу в (плотный) массив NumPy, тогда просто вызовите метод `toarray()`:

```
>>> housing_cat_lhot.toarray()
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  1.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

Вы можете применить обе трансформации (текстовых категорий в целочисленные категории и затем целочисленных категорий в векторы в унитарном коде) за один раз, используя класс `CategoricalEncoder`. Он не входит в состав Scikit-Learn 0.19.0 и предшествующих версий, но вскоре будет добавлен, так что на момент чтения книги упомянутый класс может уже быть доступным. Если это не так, то возьмите его из тетради Jupyter для настоящей главы (код был скопирован из запроса на принятие изменений (Pull Request) #9151). Вот как применять класс `CategoricalEncoder`:

¹⁷ NumPy-функция `reshape()` разрешает одному измерению быть `-1`, что означает “неопределенное”: значение выводится из длины массива и оставшихся измерений.

```
>>> from sklearn.preprocessing import CategoricalEncoder
# или возьмите из тетради
>>> cat_encoder = CategoricalEncoder()
>>> housing_cat_resaped = housing_cat.values.reshape(-1, 1)
>>> housing_cat_lhot = cat_encoder.fit_transform(housing_cat_resaped)
>>> housing_cat_lhot
<16512x5 sparse matrix of type '<class 'numpy.float64''>'
with 16512 stored elements in Compressed Sparse Row format>
```

По умолчанию класс `CategoricalEncoder` выводит разреженную матрицу, но вы можете установить кодирование в `"onehot-dense"`, если предпочитаете плотную матрицу:

```
>>> cat_encoder = CategoricalEncoder(encoding="onehot-dense")
>>> housing_cat_lhot = cat_encoder.fit_transform(housing_cat_resaped)
>>> housing_cat_lhot
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.],
       ...,
       [ 0.,  1.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.]])
```

С использованием переменной экземпляра `categories_` можно получить список категорий. Он представляет собой список, содержащий одномерный массив категорий для каждого категориального атрибута (в данном случае список, который содержит единственный массив, т.к. есть только один категориальный атрибут):

```
>>> cat_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```



Если категориальный атрибут имеет крупное число возможных категорий (скажем, код страны, профессия, группы и т.д.), тогда унитарное кодирование приведет к большому количеству входных признаков. В результате может замедлиться обучение и ухудшиться производительность. Если такое случилось, то вы захотите создать более плотные представления, называемые *вложениями* (*embedding*), но для этого требуется хорошее понимание нейронных сетей (за дополнительными сведениями обращайтесь в главу 14).

Специальные трансформаторы

Хотя Scikit-Learn предлагает много полезных трансформаторов, вам придется писать собственные трансформаторы для задач наподобие специальных операций очистки или комбинирования специфических атрибутов. Вы будете стремиться обеспечить бесшовную работу собственных трансформаторов с функциональностью Scikit-Learn (такой как конвейеры), а поскольку Scikit-Learn полагается на утиную типизацию (не наследование), то вам потребуется лишь создать класс и реализовать три метода: `fit()` (возвращающий `self`), `transform()` и `fit_transform()`. Последний метод можно получить свободно, просто добавив `TransformerMixin` в качестве базового класса. К тому же, если вы добавите `BaseEstimator` как базовый класс (и откажетесь от присутствия `*args` и `**kwargs` в своем конструкторе), тогда получите два дополнительных метода (`get_params()` и `set_params()`), которые будут полезны для автоматической настройки гиперпараметров. Например, ниже показан небольшой класс трансформатора, добавляющий скомбинированные атрибуты, которые мы обсуждали ранее:

```
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, household_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # нет *args
                                                    # или **kwargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # не остается ничего другого
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, household_ix]
        population_per_household =
            X[:, population_ix] / X[:, household_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                          bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

В приведенном примере трансформатор имеет один параметр, `add_bedrooms_per_room`, по умолчанию установленный в `True` (что часто

удобно для предоставления разумных стандартных значений). Этот гиперпараметр позволит легко выяснять, помогает ли алгоритмам МО добавление данного атрибута или нет. В более общем случае вы можете добавлять гиперпараметр для управления любым шагом подготовки данных, в котором нет 100%-ной уверенности.

Чем больше вы автоматизируете такие шаги подготовки данных, тем больше комбинаций можете опробовать автоматически и тем выше вероятность, что вы найдете замечательную комбинацию (сэкономив много времени).

Масштабирование признаков

Одной из самых важных трансформаций, которые вам понадобятся применить к своим данным, является *масштабирование признаков* (*feature scaling*). За немногими исключениями алгоритмы МО не особенно хорошо выполняются, когда входные числовые атрибуты имеют очень разные масштабы. Это касается данных о домах: общее количество комнат колеблется от 6 до 39 320 наряду с тем, что медианный доход находится в пределах лишь от 0 до 15. Обратите внимание, что масштабирование целевых значений обычно не требуется.

Существуют два распространенных способа обеспечения того же самого масштаба у всех атрибутов: *масштабирование по минимаксу* (*min-max scaling*) и *стандартизация* (*standardization*).

Масштабирование по минимаксу (многие его называют *нормализацией* (*normalization*)) выполняется довольно просто: значения смещаются и изменяются так, чтобы в итоге находиться в диапазоне от 0 до 1. Это делается путем вычитания минимального значения и деления на разность максимального и минимального значений. Для решения такой задачи в Scikit-Learn предназначен трансформатор по имени `MinMaxScaler`. У него есть гиперпараметр `feature_range`, который позволяет изменять диапазон, если по какой-то причине 0–1 не устраивает.

Стандартизация совершенно другая: сначала вычитается среднее значение (поэтому стандартизованные значения всегда имеют нулевое среднее) и затем производится деление на дисперсию, так что результирующее распределение имеет единичную дисперсию. В отличие от масштабирования по минимаксу стандартизация вовсе не привязывает значения к специфическому диапазону, что может оказаться проблемой для ряда алгоритмов (скажем, нейронные сети часто ожидают входного значения в диапазоне от 0 до 1). Тем не менее, стандартизация гораздо менее подвержена влиянию выбросов.

Например, предположим, что округ имеет медианный доход равный 100 (по ошибке). Масштабирование по минимаксу втиснуло бы все остальные значения из 0–15 в диапазон 0–0.15, тогда как стандартизация была бы менее затронутой. Для стандартизации Scikit-Learn предлагает трансформатор по имени `StandardScaler`.



Как и со всеми трансформациями, масштабирующие трансформаторы важно подгонять только к обучающим данным, а не к полному набору данных (включая испытательный набор). Лишь затем их можно использовать для трансформации обучающего набора и испытательного набора (и новых данных).

Конвейеры трансформации

Вы видели, что есть много шагов трансформации данных, которые необходимо выполнять в правильном порядке. К счастью, Scikit-Learn предоставляет класс `Pipeline`, призванный помочь справиться с такими последовательностями трансформаций. Ниже показан небольшой конвейер для числовых атрибутов:

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Конструктор `Pipeline` принимает список пар “имя/оценщик”, определяющий последовательность шагов. Все кроме последнего оценщика обязаны быть трансформаторами (т.е. они должны иметь метод `fit_transform()`). Имена могут быть какими угодно (пока они не содержат два подчеркивания (`__`)).

Вызов метода `fit()` конвейера приводит к последовательному вызову методов `fit_transform()` всех трансформаторов с передачей вывода каждого вызова в качестве параметра следующему вызову до тех пор, пока не будет достигнут последний оценщик, для которого просто вызывается метод `fit()`.

Конвейер открывает доступ к тем же самым методам, что и финальный оценщик. В текущем примере последним оценщиком является `StandardScaler`, представляющий собой трансформатор, поэтому конвейер имеет метод

`transform()`, который применяет все трансформации к данным в последовательности (в нем также есть метод `fit_transform()`, который можно было бы использовать вместо вызова `fit()` и затем `transform()`).

Теперь неплохо бы иметь возможность передавать Pandas-объект `DataFrame` прямо в конвейер вместо того, чтобы сначала вручную извлекать числовые столбцы в массив NumPy. В Scikit-Learn нет ничего для обработки Pandas-объектов `DataFrame`¹⁸, но мы можем написать специальный трансформатор для такой задачи:

```
from sklearn.base import BaseEstimator, TransformerMixin

class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

Наш класс `DataFrameSelector` будет трансформировать данные за счет выбора желаемых аргументов, отбрасывания остальных и преобразования результирующего объекта `DataFrame` в массив NumPy. С его помощью вы сможете легко написать конвейер, который будет принимать Pandas-объект `DataFrame` и обрабатывать только числовые значения: конвейер мог бы начинаться с объекта `DataFrameSelector` для отбора лишь числовых атрибутов, за которым следуют остальные шаги предварительной обработки, обсужденные ранее. С той же легкостью вы можете написать еще один конвейер для категориальных атрибутов, тоже просто выбирая категориальные атрибуты с применением объекта `DataFrameSelector` и затем объекта `CategoricalEncoder`.

```
num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
    ('selector', DataFrameSelector(num_attribs)),
    ('imputer', Imputer(strategy="median")),
    ('attribs_adder', CombinedAttributesAdder()),
    ('std_scaler', StandardScaler()),
])
```

¹⁸ Но проверьте запрос на принятие изменений #3886, который может ввести класс `ColumnTransformer`, облегчающий специфичные для атрибутов трансформации. Вы также могли бы запустить команду `pip3 install sklearn-pandas`, чтобы получить класс `DataFrameMapper`, предназначенный для похожей цели.


```
cat_pipeline = Pipeline([
    ('selector', DataFrameSelector(cat_attribs)),
    ('cat_encoder', CategoricalEncoder(encoding="onehot-dense")),
])
```

Но как можно объединить эти два конвейера в единственный конвейер? Ответ — воспользоваться классом `FeatureUnion` из Scikit-Learn. Вы передаете ему список трансформаторов (который может содержать целые конвейеры трансформаторов); когда вызывается его метод `transform()`, он параллельно запускает методы `transform()` всех трансформаторов, ожидает их вывода, затем объединяет выводы и возвращает результат (конечно, вызов его метода `fit()` приводит к вызову метода `fit()` каждого трансформатора). Полный конвейер, обрабатывающий числовые и категориальные атрибуты, может выглядеть следующим образом:

```
from sklearn.pipeline import FeatureUnion
full_pipeline = FeatureUnion(transformer_list=[
    ("num_pipeline", num_pipeline),
    ("cat_pipeline", cat_pipeline),
])
```

Целый конвейер запускается просто:

```
>>> housing_prepared = full_pipeline.fit_transform(housing)
>>> housing_prepared
array([[ -1.15604281,  0.77194962,  0.74333089, ...,  0.          ,
         0.          ,  0.          ],
       [ -1.17602483,  0.6596948 , -1.1653172 , ...,  0.          ,
         0.          ,  0.          ],
       [...]])
>>> housing_prepared.shape
(16512, 16)
```

Выбор и обучение модели

Наконец-то! Вы завершили постановку задачи, получили и исследовали данные, произвели выборку обучающего и испытательного наборов и написали конвейеры трансформации, чтобы автоматически очистить и подготовить данные для алгоритмов МО. Теперь вы готовы к выбору и обучению модели МО.

Обучение и оценка с помощью обучающего набора

Хорошая новость в том, что благодаря предшествующим шагам все должно быть гораздо проще, чем может показаться. Давайте сначала обучим модель с линейной регрессией (`LinearRegression`), как делалось в предыдущей главе:

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

Сделано! У вас есть готовая модель с линейной регрессией. Опробуем ее на нескольких образцах из обучающего набора:

```
>>> some_data = housing.iloc[:5]
>>> some_labels = housing_labels.iloc[:5]
>>> some_data_prepared = full_pipeline.transform(some_data)
>>> print("Прогнозы:", lin_reg.predict(some_data_prepared))
Прогнозы: [ 210644.6045 317768.8069 210956.4333 59218.9888 189747.5584]
>>> print("Метки:", list(some_labels))
Метки: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Модель работает, хотя прогнозы не вполне точны (например, первый прогноз отличается почти на 40%). Давайте измерим ошибку RMSE этой регрессионной модели на целом обучающем наборе, используя функцию `mean_squared_error()` из Scikit-Learn:

```
>>> from sklearn.metrics import mean_squared_error
>>> housing_predictions = lin_reg.predict(housing_prepared)
>>> lin_mse = mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse = np.sqrt(lin_mse)
>>> lin_rmse
68628.198198489219
```

Ладно, это лучше, чем ничего, но очевидно не крупный успех: значения `median_house_value` большинства округов колеблются между \$120 000 и \$265 000, так что типичная ошибка прогноза \$68 628 не особенно устраивает. Мы имеем пример недообучения модели на обучающих данных. Когда такое происходит, может быть, признаки не предоставляют достаточный объем информации для выработки качественных прогнозов или модели не хватает мощности. Как было указано в предыдущей главе, главными способами устранения причин недообучения являются выбор более мощной модели, снабжение алгоритма обучения лучшими признаками или сокращение огра-

ничений модели. Модель не регуляризирована, поэтому последний вариант исключается. Можно было бы добавить больше признаков (скажем, логарифм численности населения), но сначала опробуем более сложную модель, чтобы посмотреть, как она себя поведет.

Давайте обучим `DecisionTreeRegressor`. Это мощная модель, способная находить сложные нелинейные взаимосвязи в данных (деревья принятия решений (decision tree) подробно рассматриваются в главе 6). Код должен выглядеть знакомо:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

После обучения модели оценим ее на обучающем наборе:

```
>>> housing_predictions = tree_reg.predict(housing_prepared)
>>> tree_mse = mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse = np.sqrt(tree_mse)
>>> tree_rmse
0.0
```

Подождите, что?! Вообще нет ошибок? Действительно ли эта модель может быть полностью совершенной? Разумеется, гораздо более вероятно, что модель крайне переобучена данными. Откуда такая уверенность? Как мы отмечали ранее, вы не хотите прикасаться к испытательному набору до тех пор, пока не будете готовы запустить модель, в которой уверены, поэтому должны применять часть обучающего набора для обучения и часть для проверки модели.

Более подходящая оценка с использованием перекрестной проверки

Один из способов оценки моделей с деревьями принятия решений мог бы предусматривать применение функции `train_test_split()` для разделения обучающего набора на меньший обучающий набор и проверочный набор с последующим обучением моделей на меньшем обучающем наборе и их оценкой на проверочном наборе. Объем работы приличный, но особых сложностей с ней не связано, а результаты довольно хороши.

Замечательной альтернативой является использование средства *перекрестной проверки* Scikit-Learn. Приведенный ниже код выполняет *перекрестную проверку (с контролем) по K блокам (K-fold cross-validation)*. Он произвольно

разбивает обучающий набор на 10 несовпадающих поднаборов, называемых *блоками* (*fold*), затем обучает их и оценивает модель с деревом принятия решений 10 раз, каждый раз выбирая для оценки другой блок и проводя обучение на оставшихся 9 блоках. Результатом будет массив, содержащий 10 сумм оценок:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared,
                        housing_labels, scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```



Средство перекрестной проверки Scikit-Learn ожидает функцию полезности (больше означает лучше), а не функцию издержек (меньше означает лучше), поэтому функция подсчета фактически является противоположностью ошибки MSE (т.е. имеет отрицательное значение), из-за чего в предыдущем коде перед вычислением квадратного корня подсчитывается `-scores`.

Взглянем на результаты:

```
>>> def display_scores(scores):
...     print("Суммы оценок:", scores)
...     print("Среднее:", scores.mean())
...     print("Стандартное отклонение:", scores.std())
...
>>> display_scores(tree_rmse_scores)
Суммы оценок: [ 70232.0136482  66828.46839892  72444.08721003
  70761.50186201  71125.52697653  75581.29319857  70169.59286164
  70055.37863456  75370.49116773  71222.39081244]
Среднее: 71379.0744771
Стандартное отклонение: 2458.31882043
```

Теперь модель с деревом принятия решений выглядит не настолько хорошо, как было ранее. На самом деле похоже, что она выполняется хуже модели с линейной регрессией! Обратите внимание, что перекрестная проверка позволяет получить не только оценку производительности модели, но также меру, насколько эта оценка точна (т.е. стандартное отклонение). Модель с деревом принятия решений имеет сумму оценки приблизительно 71 379, обычно $\pm 2\,458$. Вы бы не располагали такой информацией, если бы просто применяли один проверочный набор. Но перекрестная проверка достигается ценой многократного обучения модели, а потому не всегда возможна.

Чисто ради уверенности подсчитаем те же самые суммы оценок для модели с линейной регрессией:

```
>>> lin_scores = cross_val_score(lin_reg, housing_prepared,
                                housing_labels, scoring="neg_mean_squared_error", cv=10)
...
>>> lin_rmse_scores = np.sqrt(-lin_scores)
>>> display_scores(lin_rmse_scores)
Суммы оценок: [ 66782.73843989  66960.118071  70347.95244419
  74739.57052552  68031.13388938  71193.84183426  64969.63056405
  68281.61137997  71552.91566558  67665.10082067]
Среднее: 69052.4613635
Стандартное отклонение: 2731.6740018
```

Все правильно: модель с деревом принятия решений переобучена настолько сильно, что она выполняется хуже модели с линейной регрессией.

А теперь опробуем последнюю модель: `RandomForestRegressor`. Как вы увидите в главе 7, случайные леса (random forest) работают путем обучения многочисленных деревьев принятия решений на произвольных поднаборах признаков и усредняют их прогнозы. Построение модели поверх множества других моделей называется *ансамблевым обучением* (*ensemble learning*) и часто представляет собой великолепный способ еще большего продвижения алгоритмов МО. Мы пропустим большую часть кода, потому что он такой же, как для других моделей:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> forest_reg = RandomForestRegressor()
>>> forest_reg.fit(housing_prepared, housing_labels)
>>> [...]
>>> forest_rmse
21941.911027380233
>>> display_scores(forest_rmse_scores)
Суммы оценок: [ 51650.94405471  48920.80645498  52979.16096752
  54412.74042021  50861.29381163  56488.55699727  51866.90120786
  49752.24599537  55399.50713191  53309.74548294]
Среднее: 52564.1902524
Стандартное отклонение: 2301.87380392
```

Здорово, положение намного улучшилось: случайные леса выглядят весьма многообещающими. Однако обратите внимание, что сумма оценки на обучающем наборе по-прежнему гораздо ниже, чем на проверочных наборах, а это значит, что модель все еще переобучена обучающим набором. Возможными решениями проблемы переобучения являются упрощение модели, ее ограничение (т.е. регуляризация) или получение намного большего объема обучаю-

щих данных. Тем не менее, прежде чем всецело заняться случайными лесами, вы должны опробовать многие другие модели из разнообразных категорий алгоритмов МО (ряд методов опорных векторов (support vector machine) с разными ядрами, возможно нейронную сеть и т.д.), не тратя слишком много времени на подстройку гиперпараметров. Цель в том, чтобы включить в окончательный список лишь несколько (от двух до пяти) перспективных моделей.



Вы должны сохранять каждую модель, с которой экспериментируете, чтобы позже можно было легко возвратиться к любой желаемой модели. Удостоверьтесь в том, что сохраняете и гиперпараметры, и изученные параметры, а также суммы оценок перекрестной проверки и возможно фактические прогнозы. Такой подход позволит легко сравнивать суммы оценок среди моделей разных типов и сопоставлять типы ошибок, которые они допускают. Модели Scikit-Learn можно сохранять с использованием модуля Python по имени `pickle` или библиотеки `sklearn.externals.joblib`, которая более эффективна при сериализации крупных массивов NumPy:

```
from sklearn.externals import joblib

joblib.dump(my_model, "my_model.pkl")
# и позже...
my_model_loaded = joblib.load("my_model.pkl")
```

Точная настройка модели

Предположим, что у вас уже есть окончательный список перспективных моделей. Теперь вам необходимо провести их точную настройку. Давайте рассмотрим несколько способов, к которым вы можете прибегнуть.

Решетчатый поиск

Первый способ заключается в том, чтобы вручную возиться с гиперпараметрами до тех пор, пока не обнаружится замечательная комбинация их значений. Работа может оказаться крайне утомительной и возможно у вас просто не будет времени исследовать многие комбинации.

Взамен вы должны задействовать класс `GridSearchCV` из Scikit-Learn, выполняющий *решетчатый поиск* (*grid search*). Вам нужно лишь сообщить ему, с какими параметрами вы хотите поэкспериментировать, и какие значе-

ния опробовать, а `GridSearchCV` оценит все возможные комбинации значений гиперпараметров, применяя перекрестную проверку. Например, следующий код ищет наилучшую комбинацию значений гиперпараметров для `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10],
     'max_features': [2, 3, 4]},
]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error')

grid_search.fit(housing_prepared, housing_labels)
```



Когда вы не имеете представления о том, какое значение должен иметь гиперпараметр, то простой подход предусматривает опробование последовательных степеней 10 (или меньшего числа, если вас интересует мелкозернистый поиск, как демонстрируется в этом примере с гиперпараметром `n_estimators`).

Параметр `param_grid` сообщает Scikit-Learn о том, что сначала необходимо оценить все $3 \times 4 = 12$ комбинаций значений гиперпараметров `n_estimators` и `max_features`, указанных в первом словаре (пока не беспокойтесь о смысле данных гиперпараметров; они будут объясняться в главе 7), затем опробовать $2 \times 3 = 6$ комбинаций значений гиперпараметров во втором словаре, но на этот раз с гиперпараметром `bootstrap`, установленным в `False`, а не `True` (стандартное значение для `bootstrap`).

В общем, решетчатый поиск будет исследовать $12 + 6 = 18$ комбинаций значений гиперпараметров класса `RandomForestRegressor` и обучать каждую модель пять раз (поскольку мы используем перекрестную проверку по пяти блокам). Другими словами, в целом будет проводиться $18 \times 5 = 90$ циклов обучения! Процесс может занять долгое время, но после его завершения вы можете получить наилучшую комбинацию параметров вроде такой:

```
>>> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```



Поскольку 8 и 30 являются максимальными значениями, которые были оценены, вероятно, вы должны поискать более высокие значения, потому что сумма оценки может продолжить улучшаться.

Вы также можете получить лучший оценщик напрямую:

```
>>> grid_search.best_estimator_  
RandomForestRegressor(bootstrap=True, criterion='mse',  
                        max_depth=None, max_features=8, max_leaf_nodes=None,  
                        min_impurity_decrease=0.0, min_impurity_split=None,  
                        min_samples_leaf=1, min_samples_split=2,  
                        min_weight_fraction_leaf=0.0, n_estimators=30,  
                        n_jobs=1, oob_score=False, random_state=42, verbose=0,  
                        warm_start=False)
```



Если объект `GridSearchCV` инициализируется с `refit=True` (по умолчанию), тогда после нахождения лучшего оценщика с применением перекрестной проверки он переобучит его на полном обучающем наборе. Как правило, это хорошая идея, потому что снабжение дополнительными данными, скорее всего, улучшит его производительность.

Конечно, суммы оценок также доступны:

```
>>> cvres = grid_search.cv_results_  
>>> for mean_score, params in zip(cvres["mean_test_score"],  
                                cvres["params"]):  
...     print(np.sqrt(-mean_score), params)  
...  
63647.854446 {'n_estimators': 3, 'max_features': 2}  
55611.5015988 {'n_estimators': 10, 'max_features': 2}  
53370.0640736 {'n_estimators': 30, 'max_features': 2}  
60959.1388585 {'n_estimators': 3, 'max_features': 4}  
52740.5841667 {'n_estimators': 10, 'max_features': 4}  
50374.1421461 {'n_estimators': 30, 'max_features': 4}  
58661.2866462 {'n_estimators': 3, 'max_features': 6}  
52009.9739798 {'n_estimators': 10, 'max_features': 6}  
50154.1177737 {'n_estimators': 30, 'max_features': 6}  
57865.3616801 {'n_estimators': 3, 'max_features': 8}  
51730.0755087 {'n_estimators': 10, 'max_features': 8}  
49694.8514333 {'n_estimators': 30, 'max_features': 8}  
62874.4073931 {'n_estimators': 3, 'bootstrap': False, 'max_features': 2}  
54643.4998083 {'n_estimators': 10, 'bootstrap': False, 'max_features': 2}  
59437.8922859 {'n_estimators': 3, 'bootstrap': False, 'max_features': 3}
```

```
52735.3582936 {'n_estimators': 10, 'bootstrap': False, 'max_features': 3}
57490.0168279 {'n_estimators': 3, 'bootstrap': False, 'max_features': 4}
51008.2615672 {'n_estimators': 10, 'bootstrap': False, 'max_features': 4}
```

В рассматриваемом примере мы получаем лучшее решение, устанавливая гиперпараметр `max_features` в 8, а гиперпараметр `n_estimators` — в 30. Сумма оценки RMSE для такой комбинации составляет 49 694, что немного лучше, чем сумма оценки, полученная ранее с использованием стандартных значений гиперпараметров (52 564). Примите поздравления, вы успешно провели точную настройку своей лучшей модели!



Не забывайте, что вы можете трактовать некоторые шаги подготовки данных как гиперпараметры. Например, решетчатый поиск будет автоматически выяснять, добавлять или нет признак, в котором вы не были уверены (скажем, применение гиперпараметра `add_bedrooms_per_room` трансформатора `CombinedAttributesAdder`). Он может аналогичным образом использоваться для автоматического нахождения лучшего способа обработки выбросов, недостающих признаков, выбора признаков и т.д.

Рандомизированный поиск

Подход с решетчатым поиском хорош, когда вы исследуете относительно небольшое число комбинаций, как в предыдущем примере, но если *пространство поиска* гиперпараметра является крупным, то часто предпочтительнее применять *рандомизированный поиск* (*randomized search*), который реализован классом `RandomizedSearchCV`. Этот класс может использоваться аналогично классу `GridSearchCV`, но вместо опробования всех возможных комбинаций он оценивает заданное количество случайных комбинаций, выбирая случайное значение для каждого гиперпараметра на любой итерации. Такой подход обладает двумя главными преимуществами.

- Если вы позволите рандомизированному поиску выполняться, скажем, для 1000 итераций, тогда будет исследовано 1000 разных значений для каждого гиперпараметра (а не лишь несколько значений на гиперпараметр при подходе с решетчатым поиском).
- Вы имеете больший контроль над вычислительными ресурсами, которые желали бы выделить на поиск значений гиперпараметров, просто устанавливая количество итераций.

Ансамблевые методы

Еще один способ точной настройки вашей системы предусматривает попытку объединения моделей, которые выполняются лучше. Группа (или “ансамбль” (ensemble)) зачастую будет выполняться лучше, чем наилучшие отдельные модели (точно так же, как случайные леса выполняются лучше индивидуальных деревьев принятия решений, на которые они опираются), особенно если эти отдельные модели допускают очень разные типы ошибок. Мы раскроем данную тему более подробно в главе 7.

Анализ лучших моделей и их ошибок

Часто получить хорошее представление о задаче можно за счет изучения лучших моделей. Например, `RandomForestRegressor` может обозначить относительную важность каждого атрибута для выработки точных прогнозов:

```
>>> feature_importances =
    grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([ 7.33442355e-02,  6.29090705e-02,  4.11437985e-02,
        1.46726854e-02,  1.41064835e-02,  1.48742809e-02,
        1.42575993e-02,  3.66158981e-01,  5.64191792e-02,
        1.08792957e-01,  5.33510773e-02,  1.03114883e-02,
        1.64780994e-01,  6.02803867e-05,  1.96041560e-03,
        2.85647464e-03])
```

Давайте отобразим оценки важности рядом с именами соответствующих атрибутов:

```
>>> extra_attribs =
    ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = cat_pipeline.named_steps["cat_encoder"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
```

```
(0.014672685420543237, 'total_rooms'),  
(0.014257599323407807, 'households'),  
(0.014106483453584102, 'total_bedrooms'),  
(0.010311488326303787, '<1H OCEAN'),  
(0.0028564746373201579, 'NEAR OCEAN'),  
(0.0019604155994780701, 'NEAR BAY'),  
(6.0280386727365991e-05, 'ISLAND')]
```

Располагая такой информацией, вы можете попытаться отбросить какие-то менее полезные признаки (например, по-настоящему важной видимо является только категория `ocean_proximity`, так что вы могли бы попробовать отбросить остальные).

Вы также должны посмотреть на специфические ошибки, которые допускает система, затем постараться понять, почему она их допускает, и что могло бы исправить проблему (добавление дополнительных признаков или наоборот избавление от неинформативных признаков, очистка выбросов и т.д.).

Оценка системы с помощью испытательного набора

После точной настройки вы в итоге получите систему, которая функционирует достаточно хорошо. Подошло время провести оценку финальной модели на испытательном наборе. В этом процессе нет ничего особенного; просто получите прогнозаторы и метки из испытательного набора, запустите свой полный конвейер (`full_pipeline`), чтобы трансформировать данные (вызывайте `transform()`, *не* `fit_transform()`!), и оцените финальную модель с помощью испытательного набора:

```
final_model = grid_search.best_estimator_  
X_test = strat_test_set.drop("median_house_value", axis=1)  
y_test = strat_test_set["median_house_value"].copy()  
X_test_prepared = full_pipeline.transform(X_test)  
final_predictions = final_model.predict(X_test_prepared)  
final_mse = mean_squared_error(y_test, final_predictions)  
final_rmse = np.sqrt(final_mse) # => оценивается в 47 766.0
```

Производительность обычно будет несколько хуже, чем та, которую вы измеряли с применением перекрестной проверки, если вы проводили большой объем настройки гиперпараметров (поскольку ваша система в итоге настроена на хорошее выполнение с проверочными данными и вряд ли будет выполняться столь же хорошо с неизвестными наборами данных). В этом

примере подобное не происходит, но когда такое случается, то вы должны устоять перед соблазном подстроить гиперпараметры, чтобы обеспечить хорошие показатели на испытательном наборе; улучшения вряд ли будут обобщены на новые данные.

Теперь наступает стадия предварительного запуска проекта: вам необходимо представить свое решение (выделяя то, что вы узнали, что сработало, а что нет, какие предположения были сделаны, и каковы ограничения вашей системы), документировать все аспекты и создать подходящие презентации с четкими визуализациями и легкими для запоминания формулировками (скажем, “медианный доход — это прогнозатор номер один цен на дома”).

Запуск, наблюдение и сопровождение системы

Отлично, вы получили разрешение на запуск! Вы должны подготовить свое решение для помещения в производственную среду, в частности подключить к системе источники производственных входных данных и написать тесты.

Вам также необходимо написать код наблюдения для контроля реальной производительности системы через регулярные интервалы и включать сигналы тревоги, когда она падает. Это важно для выявления не только неожиданных отказов, но также факта снижения производительности. Такой подход довольно обычен, потому что по мере эволюции данных с течением времени модели склонны “портиться”, если только не будут методично обучаться на свежих данных.

Оценка производительности вашей системы потребует выборки вырабатываемых ею прогнозов и их оценки. Как правило, для этого понадобятся аналитики, которыми могут быть эксперты на местах или работники на краудсорсинговой основе (такой как Amazon Mechanical Turk или CrowdFlower). Так или иначе, вам нужно подключить к системе конвейер человеческой оценки.

Вы также должны обеспечить оценку качества входных данных системы. Иногда производительность системы будет слегка снижаться из-за низкого качества сигнала (например, неисправный датчик посылает произвольные значения или выходные данные другой команды становятся устаревшими), но может пройти какое-то время, прежде чем производительность снизится настолько, чтобы сработал сигнал тревоги. Если вы наблюдаете за входными данными своей системы, то можете перехватить это раньше. Наблюдение за входными данными особенно важно для систем динамического обучения.

Наконец, вы обычно будете стремиться обучать свои модели на регулярной основе, используя свежие данные. Вы должны как можно больше автоматизировать такой процесс. Иначе весьма вероятно, что вы будете обновлять свою модель только каждые полгода (в лучшем случае), а производительность системы с течением времени может сильно колебаться. Если вы имеете дело с системой динамического обучения, тогда должны обеспечить сохранение снимков ее состояния через регулярные интервалы, чтобы можно было легко произвести откат к предыдущему работоспособному состоянию.

Пробуйте!

Будем надеяться, настоящая глава дала вам хорошее представление о том, на что похож проект машинного обучения, и продемонстрировала ряд инструментов, которые вы можете применять для обучения крупной системы. Как было показано, большая часть работы связана с шагом подготовки, построением инструментов наблюдения, организацией конвейеров человеческой оценки и автоматизацией регулярного обучения моделей. Разумеется, алгоритмы МО также важны, но вероятно предпочтительнее освоить общий процесс и хорошо знать три или четыре алгоритма, чем тратить все свое время на исследование передовых алгоритмов и не уделять достаточного времени полному процессу.

Итак, если вы еще этого не сделали, то сейчас самое подходящее время взять ноутбук, выбрать интересующий набор данных и попробовать пройти через весь процесс от А до Я. Хорошим местом для старта будет веб-сайт состязаний вроде <http://kaggle.com/>: у вас будет набор данных для работы, ясная цель и люди для обмена опытом.

Упражнения

Выполните следующие упражнения с использованием набора данных `housing`.

1. Испытайте метод опорных векторов для регрессии (`sklearn.svm.SVR`) с разнообразными гиперпараметрами, такими как `kernel="linear"` (с различными значениями для гиперпараметра `C`) или `kernel="rbf"` (с различными значениями для гиперпараметров `C` и `gamma`). Пока не беспокойтесь о смысле этих гиперпараметров. Насколько лучше работает прогнозатор `SVR`?

2. Попробуйте заменить `GridSearchCV` классом `RandomizedSearchCV`.
3. Попробуйте добавить трансформатор в конвейер подготовки, чтобы выбирать только самые важные атрибуты.
4. Попробуйте создать единственный конвейер, который выполняет полную подготовку данных и вырабатывает финальный прогноз.
5. Автоматически исследуйте некоторые варианты подготовки с применением `GridSearchCV`.

Решения упражнений доступны в онлайн-тетрадах Jupyter по адресу <http://github.com/ageron/handson-ml>.

Классификация

В главе 1 упоминалось, что самыми распространенными задачами обучения с учителем являются регрессия (прогнозирование значений) и классификация (прогнозирование классов). В главе 2 была исследована задача регрессии, прогнозирующая стоимость домов с использованием разнообразных алгоритмов, таких как линейная регрессия, деревья принятия решений и случайные леса (которые будут объясняться более подробно в последующих главах). Теперь мы сосредоточим внимание на системах классификации.

MNIST

В настоящей главе мы будем применять набор данных MNIST (Mixed National Institute of Standards and Technology — смешанный набор данных Национального института стандартов и технологий США), который содержит 70 000 небольших изображений цифр, написанных от руки учащимися средних школ и служащими Бюро переписи населения США. Каждое изображение помечено цифрой, которую оно представляет. Этот набор изучался настолько досконально, что его часто называют первым примером (“Hello World”) машинного обучения: всякий раз, когда люди строят новый алгоритм классификации, им любопытно посмотреть, как он будет выполняться на наборе MNIST. Во время освоения машинного обучения рано или поздно любому доведется столкнуться с MNIST.

Библиотека Scikit-Learn предоставляет много вспомогательных функций для загрузки популярных наборов данных. В их число входит MNIST. Следующий код извлекает набор данных MNIST¹:

```
>>> from sklearn.datasets import fetch_mldata
>>> mnist = fetch_mldata('MNIST original')
>>> mnist
```

¹ По умолчанию Scikit-Learn кеширует загруженные наборы данных в каталоге по имени `$HOME/scikit_learn_data`.

```
{ 'COL_NAMES': ['label', 'data'],
  'DESCR': 'mldata.org dataset: mnist-original',
  'data': array([[0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 ...,
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0],
                 [0, 0, 0, ..., 0, 0, 0]], dtype=uint8),
  'target': array([ 0.,  0.,  0., ...,  9.,  9.,  9.]}}
```

Наборы данных, загруженные Scikit-Learn, обычно имеют похожие словарные структуры, в состав которых входят:

- ключ `DESCR`, описывающий набор данных;
- ключ `data`, содержащий массив с одной строкой на образец и одним столбцом на признак;
- ключ `target`, содержащий массив с метками.

Посмотрим, что это за массивы:

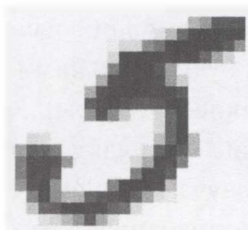
```
>>> X, y = mnist["data"], mnist["target"]
>>> X.shape
(70000, 784)
>>> y.shape
(70000,)
```

Существует 70 000 изображений и с каждым изображением связано 784 признака. Дело в том, что каждое изображение имеет размер 28×28 пикселей, а каждый признак просто представляет интенсивность одного пикселя, от 0 (белый) до 255 (черный). Давайте хоть одним глазком взглянем на какую-нибудь цифру из набора данных. Понадобится лишь извлечь вектор признаков образца, придать ему форму массива 28×28 и отобразить его содержимое, используя функцию `imshow()` из библиотеки Matplotlib:

```
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt

some_digit = X[36000]
some_digit_image = some_digit.reshape(28, 28)

plt.imshow(some_digit_image, cmap = matplotlib.cm.binary,
           interpolation="nearest")
plt.axis("off")
plt.show()
```



Похоже на цифру 5 и метка действительно это подтверждает:

```
>>> y[36000]  
5.0
```

На рис. 3.1 показано чуть больше изображений из набора данных MNIST, чтобы вы смогли ощутить сложность задачи классификации.

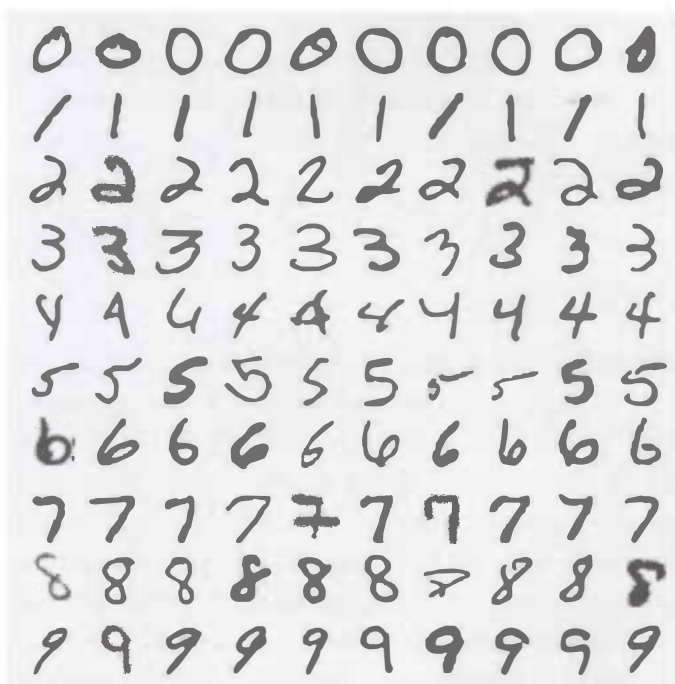


Рис. 3.1. Несколько изображений цифр из набора данных MNIST

Но подождите! Вы же должны всегда создавать испытательный набор и откладывать его в сторону, прежде чем внимательно обследовать данные. На самом деле набор данных MNIST уже разделен на обучающий набор (первые 60 000 изображений) и испытательный набор (последние 10 000 изображений):

```
X_train, X_test, y_train, y_test =  
X[:60000], X[60000:], y[:60000], y[60000:]
```


Давайте также перетасуем обучающий набор; это гарантирует, что все блоки перекрестной проверки будут похожими (вы вовсе не хотите, чтобы в одном блоке отсутствовали какие-то цифры). Кроме того, некоторые обучающие алгоритмы чувствительны к порядку следования обучающих образцов и плохо выполняются, если получают в одной строке много похожих образцов. Тасование набора данных гарантирует, что это не случится²:

```
import numpy as np

shuffle_index = np.random.permutation(60000)
X_train, y_train = X_train[shuffle_index], y_train[shuffle_index]
```

Обучение двоичного классификатора

Упростим на время задачу и попробуем опознать только одну цифру — скажем, 5. Такой “детектор пятерок” будет примером *двоичного классификатора* (*binary classifier*), способного проводить различие между всего лишь двумя классами, “пятерка” и “не пятерка”. Давайте создадим целевые векторы для этой задачи классификации:

```
y_train_5 = (y_train == 5) # True для всех пятерок,
                             # False для всех остальных цифр.
y_test_5 = (y_test == 5)
```

А теперь возьмем классификатор и обучим его. Хорошим местом для старта будет классификатор на основе метода *стохастического градиентного спуска* (*Stochastic Gradient Descent* — *SGD*), который применяет класс *SGDClassifier* из Scikit-Learn. Преимущество такого классификатора в том, что он способен эффективно обрабатывать очень крупные наборы данных. Отчасти это связано с тем, что SGD использует обучающие образцы независимым образом по одному за раз (обстоятельство, которое делает SGD также подходящим для *динамического обучения*), как мы увидим позже. Создадим экземпляр *SGDClassifier* и обучим его на целом обучающем наборе:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

² Тасование может оказаться неудачной идеей в ряде контекстов — например, при работе с данными временных рядов (такими как биржевые курсы или метеорологические сводки). Мы будем исследовать это в последующих главах.



Класс `SGDClassifier` полагается на неупорядоченность во время обучения (отсюда и название метода “стохастический”). Если вас интересуют воспроизводимые результаты, тогда вы должны установить параметр `random_state`.

Теперь его можно применять для обнаружения изображений цифры 5:

```
>>> sgd_clf.predict([some_digit])  
array([ True], dtype=bool)
```

Классификатор считает, что это изображение представляет пятерку (`True`). Похоже, в данном случае он отгадал! Давайте оценим производительность модели.

Показатели производительности

Оценка классификатора часто значительно сложнее, чем оценка регрессора, а потому мы посвятим этой теме большую часть главы. Доступно много показателей производительности, так что снова запаситесь кофе и приготовьтесь к ознакомлению с многочисленными новыми концепциями и аббревиатурами!

Измерение правильности с использованием перекрестной проверки

Хороший способ оценки модели предусматривает применение перекрестной проверки, как делалось в главе 2.

Реализация перекрестной проверки

Иногда вам будет необходим больший контроль над процессом перекрестной проверки, нежели тот, который предлагает Scikit-Learn в готовом виде. В таких случаях вы можете реализовать перекрестную проверку самостоятельно; в действительности это довольно просто. Следующий код делает приблизительно то же, что и функция `cross_val_score()` из Scikit-Learn, выводя тот же самый результат:

```
from sklearn.model_selection import StratifiedKFold  
from sklearn.base import clone  
  
skfolds = StratifiedKFold(n_splits=3, random_state=42)
```

```

for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # выводит 0.9502, 0.96565
                                   # и 0.96495

```

Класс `StratifiedKfold` производит стратифицированную выборку (как объяснялось в главе 2), чтобы создать блоки, которые содержат репрезентативную пропорцию каждого класса. На каждой итерации код создает клон классификатора, обучает его на обучающих блоках и вырабатывает прогнозы на испытательном блоке. Затем он подсчитывает количество корректных прогнозов и выводит коэффициент корректных прогнозов.

Давайте используем функцию `cross_val_score()` для оценки модели `SGDClassifier` с применением перекрестной проверки по K блокам, имея три блока. Вспомните, что перекрестная проверка по K блокам означает разбиение обучающего набора на K блоков (в этом случае на три), выработку прогнозов и их оценку на каждом блоке с использованием модели, обученной на остальных блоках (см. главу 2):

```

>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])

```

Здорово! *Правильность* (*accuracy*; коэффициент корректных прогнозов) выше 95% на всех блоках перекрестной проверки? Выглядит поразительно, не так ли? Ну, прежде чем приходить в слишком сильное возбуждение, посмотрим на очень глупый классификатор, который просто относит все изображения без исключения к классу “не пятерка”:

```

from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)

```

Можете угадать правильность такой модели? Давайте выясним:

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([ 0.909 ,  0.90715,  0.9128 ])
```

Все так, правильность модели выше 90%! Дело в том, что лишь около 10% изображений являются пятерками, поэтому если вы неизменно полагаете, что изображение — *не* пятерка, то и будете правы примерно в 90% случаев. Нострадамус побежден.

Это демонстрирует причину, по которой правильность обычно не считается предпочтительным показателем производительности для классификаторов, особенно когда вы работаете с *ассиметричными наборами данных* (т.е. одни классы встречаются намного чаще других).

Матрица неточностей

Гораздо лучший способ оценки производительности классификатора предусматривает просмотр *матрицы неточностей* (*confusion matrix*). Общая идея заключается в том, чтобы подсчитать, сколько раз образцы класса А были отнесены к классу В. Например, для выяснения, сколько раз классификатор путал изображения пятерок с тройками, вы могли бы заглянуть в 5-ю строку и 3-й столбец матрицы неточностей.

Для расчета матрицы неточностей сначала понадобится иметь набор прогнозов, чтобы их можно было сравнивать с фактическими целями. Вы могли бы выработать прогнозы на испытательном наборе, но давайте пока оставим его незатронутым (вспомните, что вы хотите применять испытательный набор только в самом конце проекта после того, как у вас будет классификатор, готовый к запуску). Взамен мы можете использовать функцию `cross_val_predict()`:

```
from sklearn.model_selection import cross_val_predict
y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)
```

Как и `cross_val_score()`, функция `cross_val_predict()` выполняет перекрестную проверку по К блокам, но вместо сумм очков оценки она возвращает прогнозы, выработанные на каждом испытательном блоке. Это значит, что вы получите чистый прогноз для каждого образца в обучающем наборе (“чистый” означает, что прогноз сделан моделью, которая никогда не видела данные во время обучения).

Теперь вы готовы получить матрицу неточностей с применением функции `confusion_matrix()`. Просто передайте функции целевые классы (`y_train_5`) и спрогнозированные классы (`y_train_pred`):

```
>>> from sklearn.metrics import confusion_matrix
>>> confusion_matrix(y_train_5, y_train_pred)
array([[53272, 1307],
       [1077, 4344]])
```

Каждая строка в матрице неточностей представляет *фактический класс*, а каждый столбец — *спрогнозированный класс*. Первая строка матрицы учитывает изображения не пятерок (*отрицательный класс (negative class)*): 53 272 их них были корректно классифицированы как не пятерки (*истинно отрицательные классификации (true negative — TN)*), тогда как оставшиеся 1 307 были ошибочно классифицированы как пятерки (*ложноположительные классификации (false positive — FP)*). Вторая строка матрицы учитывает изображения пятерок (*положительный класс (positive class)*): 1 077 были ошибочно классифицированы как не пятерки (*ложноотрицательные классификации (false negative — FN)*), в то время как оставшиеся 4 344 были корректно классифицированы как пятерки (*истинно положительные классификации (true positive — TP)*). Безупречный классификатор имел бы только истинно положительные и истинно отрицательные классификации, так что его матрица неточностей содержала бы ненулевые значения только на своей главной диагонали (от левого верхнего до правого нижнего угла):

```
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579, 0],
       [0, 5421]])
```

Матрица неточностей дает вам много информации, но иногда вы можете предпочесть более сжатую метрику. Интересной метрикой такого рода является аккуратность положительных классификаций; она называется *точностью (precision)* классификатора (уравнение 3.1).

Уравнение 3.1. Точность

$$\text{точность} = \frac{TP}{TP + FP}$$

TP — это количество истинно положительных классификаций, а *FP* — количество ложноположительных классификаций.

Тривиальный способ добиться совершенной точности заключается в том, чтобы вырабатывать одну положительную классификацию и удостоверяться в ее корректности (*точность* = $1/1 = 100\%$). Это было бы не особенно полезно, т.к. классификатор игнорировал бы все кроме одного положительного образца. Таким образом, точность обычно используется наряду с еще одной метрикой под названием *полнота* (*recall*), также называемой *чувствительностью* (*sensitivity*) или *долей истинно положительных классификаций* (*True Positive Rate* — *TPR*): коэффициент положительных образцов, которые корректно обнаружены классификатором (уравнение 3.2).

Уравнение 3.2. Полнота

$$\text{полнота} = \frac{TP}{TP + FN}$$

Естественно, *FN* — это число ложноотрицательных классификаций.

Если вы запутались с матрицей неточностей, то устранить путаницу может помочь рис. 3.2.



Рис. 3.2. Иллюстрированная матрица неточностей

Точность и полнота

В Scikit-Learn предлагается несколько функций для подсчета метрик классификаторов, в том числе точности и полноты:


```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1307)
0.76871350203503808
>>> recall_score(y_train_5, y_train_pred) # == 4344 / (4344 + 1077)
0.80132816823464303
```

Теперь ваш детектор пятерок не выглядит настолько блестящим, как при просмотре его правильности. Его заявление о том, что изображение представляет пятерку, корректно только 77% времени. Кроме того, он обнаруживает только 80% пятерок.

Точность и полноту часто удобно объединять в единственную метрику под названием *мера F_1* (F_1 score), особенно если вам нужен простой способ сравнения двух классификаторов. Мера F_1 — это *среднее гармоническое* (*harmonic mean*) точности и полноты (уравнение 3.3). В то время как обычное среднее трактует все значения одинаково, среднее гармоническое придает низким значениям больший вес. В результате классификатор получит высокую меру F_1 , только если высокими являются и полнота, и точность.

Уравнение 3.3. Мера F_1

$$F_1 = \frac{2}{\frac{1}{\text{точность}} + \frac{1}{\text{полнота}}} = 2 \times \frac{\text{точность} \times \text{полнота}}{\text{точность} + \text{полнота}} = \frac{TP}{TP + \frac{FN + FP}{2}}$$

Чтобы вычислить меру F_1 , просто вызовите функцию `f1_score()`:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.78468208092485547
```

Мера F_1 поддерживает классификаторы, которые имеют подобные точность и полноту. Это не всегда то, что вы хотите: в одних контекстах вы главным образом заботитесь о точности, а в других — фактически о полноте. Например, если вы обучаете классификатор выявлению видеороликов, безопасных для просмотра детьми, тогда наверняка отдадите предпочтение классификатору, который отклоняет многие хорошие видеоролики (низкая полнота), но сохраняет только безопасные видеоролики (высокая точность), а не классификатору, имеющему гораздо более высокую полноту, но позволяющему проникнуть в продукт нескольким по-настоящему нежелательным видеороликам (в таких случаях вы можете даже решить добавить человеческий конвейер для проверки результатов выбора видеороликов классификатором).

С другой стороны, пусть вы обучаете классификатор обнаружению магазинных воришек по изображениям с камер наблюдения: вполне вероятно будет хорошо, если ваш классификатор имеет только 30%-ную точность при условии 99%-ной полноты (конечно, охранники будут получать незначительное число ложных сигналов тревоги, но почти все магазинные воришки окажутся пойманными).

К сожалению, невозможно получить то и другое одновременно: увеличение точности снижает полноту и наоборот. Это называется *соотношением точность/полнота* (*precision/recall tradeoff*).

Соотношение точность/полнота

Чтобы понять данное соотношение, давайте посмотрим, каким образом класс `SGDClassifier` принимает свои решения по классификации. Для каждого образца он вычисляет сумму очков на основе *функции решения* (*decision function*), и если сумма очков больше какого-то порогового значения, тогда образец приписывается положительному классу, а иначе — отрицательному классу. На рис. 3.3 показано несколько цифр, расположенных в порядке от самой низкой суммы очков слева до самой высокой справа. Предположим, что *порог принятия решения* (*decision threshold*) находится на центральной стрелке (между двумя пятерками): вы обнаружите справа от этого порога четыре истинно положительные классификации (реальные пятерки) и одну ложноположительную классификацию (фактически шестерку). Следовательно, при таком пороге точность составляет 80% (4 из 5). Но из шести фактических пятерок классификатор обнаруживает только четыре, так что полнота равна 67% (4 из 6).

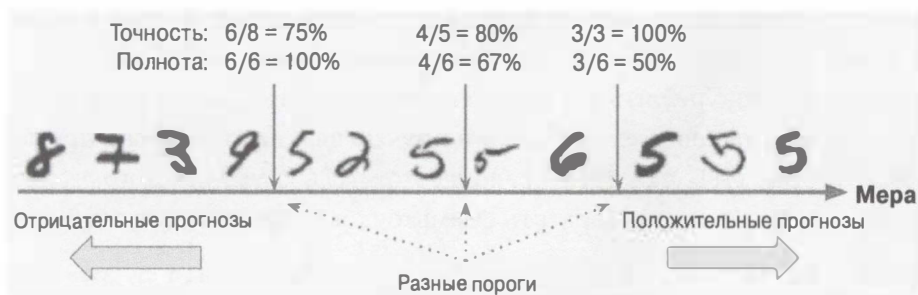


Рис. 3.3. Порог принятия решения и соотношение точность/полнота

Если вы поднимете порог (перемещая его по стрелке вправо), тогда ложноположительная классификация (шестерка) становится истинно отрицательной, увеличивая тем самым точность (вплоть до 100% в данном случае), но одна истинно положительная классификация становится ложноотрицательной, снижая полноту до 50%. И наоборот, снижение порога увеличивает полноту и сокращает точность.

Библиотека Scikit-Learn не позволяет устанавливать порог напрямую, но предоставляет доступ к суммам очков, управляющих решением, которые она применяет для выработки прогнозов. Вместо вызова метода `predict()` классификатора вы можете вызвать его метод `decision_function()`, который возвращает сумму очков для каждого образца, и затем выработать прогнозы на основе этих сумм очков, используя любой желаемый порог:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([ 161855.74572176])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
array([ True], dtype=bool)
```

Класс `SGDClassifier` применяет порог равный 0, так что предыдущий код возвращает такой же результат, как и метод `predict()` (т.е. `True`). Давайте поднимем порог:

```
>>> threshold = 200000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False], dtype=bool)
```

Результат подтверждает, что поднятие порога снижает полноту. Изображение фактически представляет пятерку и классификатор опознает ее, когда порог равен 0, но не замечает, когда порог увеличивается до 200 000.

Так как же можно решить, какой порог использовать? Сначала понадобится получить суммы очков всех образцов в обучающем наборе, снова применяя функцию `cross_val_predict()`, но на этот раз с указанием того, что вместо прогнозов она должна возвратить суммы очков, управляющие решением:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

Теперь с помощью сумм очков можно вычислить точность и полноту для всех возможных порогов, используя функцию `precision_recall_curve()`:

```
from sklearn.metrics import precision_recall_curve
precisions, recalls, thresholds =
    precision_recall_curve(y_train_5, y_scores)
```

Наконец, с применением Matplotlib вы можете вычертить точность и полноту как функции значения порога (рис. 3.4):

```
def plot_precision_recall_vs_threshold(precisions,
                                      recalls, thresholds):
    plt.plot(thresholds, precisions[:-1], "b--", label="Точность")
    plt.plot(thresholds, recalls[:-1], "g-", label="Полнота")
    plt.xlabel("Порог")
    plt.legend(loc="center left")
    plt.ylim([0, 1])

plot_precision_recall_vs_threshold(precisions, recalls, thresholds)
plt.show()
```

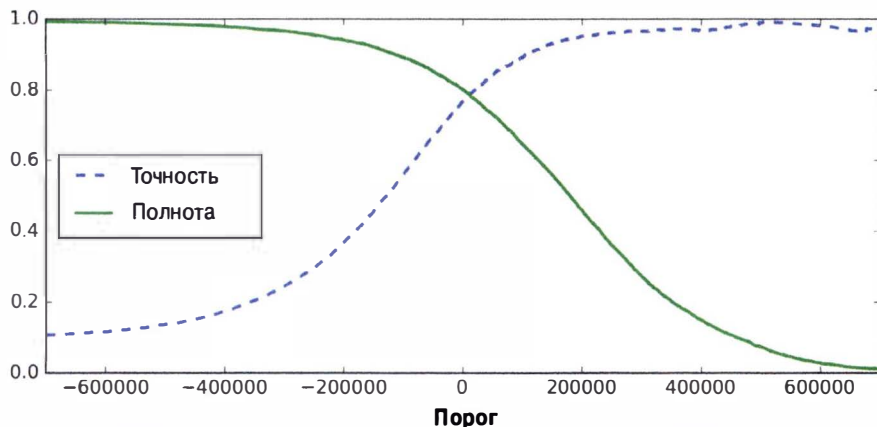


Рис. 3.4. Точность и полнота в сравнении с порогом принятия решения



Вас может интересовать, почему на рис. 3.4 кривая точности имеет больше выбоин, чем кривая полноты. Дело в том, что при поднятии порога точность иногда может снижаться (хотя в общем случае она будет расти). Чтобы понять причину, снова взгляните на рис. 3.3 и обратите внимание, что происходит, когда вы начинаете с центрального порога и перемещаете его вправо только на одну цифру: точность из $4/5$ (80%) снижается до $3/4$ (75%). С другой стороны, при увеличении порога полнота может только понижаться, что и объясняет гладкость представляющей ее кривой.

Теперь вы можете просто выбрать значение порога, которое позволяет достичь наилучшего соотношения точность/полнота для имеющейся задачи. Другой способ выбора хорошего соотношения точность/полнота предусматривает вычерчивание графика точности в сравнении с полнотой, как показано на рис. 3.5.

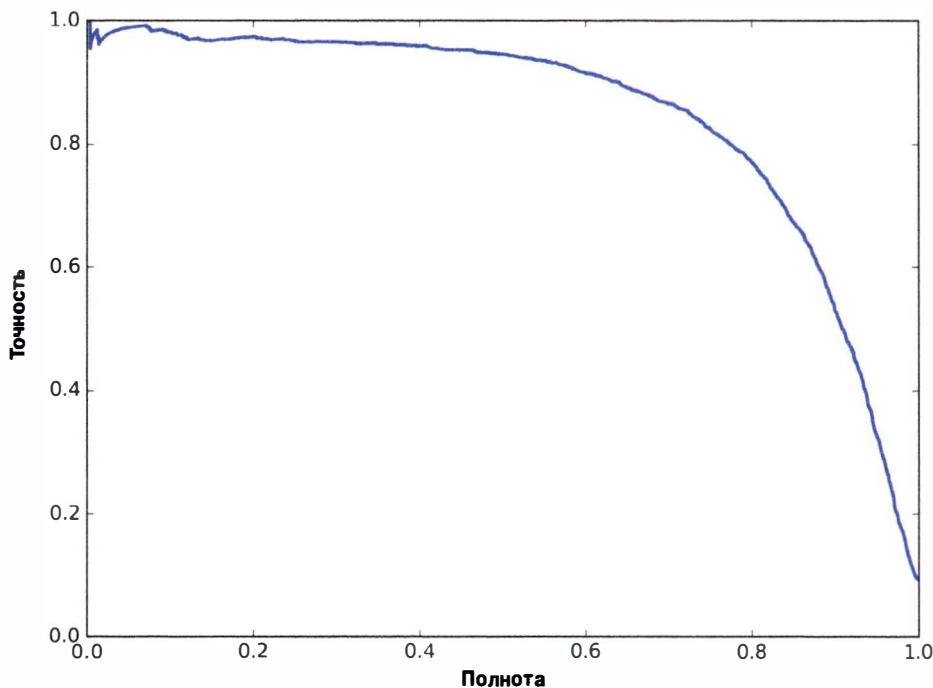


Рис. 3.5. Точность в сравнении с полнотой

Легко заметить, что точность действительно начинает резко падать при полноте около 80%. Вероятно, вам захочется выбрать соотношение точность/полнота прямо перед этим падением — например, поблизости к полноте 60%. Но, конечно же, выбор зависит от выполняемого проекта.

Давайте предположим, что вы решили нацелиться на точность 90%. Вы смотрите на первый график (слегка увеличив его) и обнаруживаете, что нужно использовать порог приблизительно 70 000. Чтобы вырабатывать прогнозы (пока что на обучающем наборе), вместо вызова метода `predict()` классификатора вы можете просто запустить следующий код:

```
y_train_pred_90 = (y_scores > 70000)
```

Проверим точность и полноту таких прогнозов:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.86592051164915484
>>> recall_score(y_train_5, y_train_pred_90)
0.69931746910164172
```

Замечательно, вы располагаете классификатором с точностью 90% (или достаточно близкой)! Как видите, создать классификатор практически любой желаемой точности довольно легко: лишь установите достаточно высокий порог и все готово. Но не торопитесь. Высокоточный классификатор не особенно полезен, если его полнота слишком низкая!



Если кто-то говорит: “давайте достигнем точности 99%”, то вы обязаны спросить: “а при какой полноте?”

Кривая ROC

Кривая *рабочей характеристики приемника* (*Receiver Operating Characteristic* — ROC) представляет собой еще один распространенный инструмент, применяемый с двоичными классификаторами. Она очень похожа на *кривую точности-полноты* (*precision-recall (PR) curve*), но вместо вычерчивания точности в сравнении с полнотой кривая ROC изображает *долю истинно положительных классификаций* (другое название полноты) по отношению к *доле ложноположительных классификаций* (*False Positive Rate* — FPR). Доля FPR — это пропорция отрицательных образцов, которые были некорректно классифицированы как положительные. Она равна единице минус *доля истинно отрицательных классификаций* (*True Negative Rate* — TNR), представляющая собой пропорцию отрицательных образцов, которые были корректно классифицированы как отрицательные.

Доля TNR также называется *специфичностью* (*specificity*). Следовательно, кривая ROC изображает *чувствительность* (*sensitivity*), т.е. полноту, в сравнении с 1 – *специфичность*.

Чтобы вычертить кривую ROC, сначала необходимо вычислить TPR и FPR для разнообразных значений порога, используя функцию `roc_curve()`:

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```


Затем можно вычертить график FPR в сравнении с TPR, применяя Matplotlib. Приведенный ниже код выводит график, показанный на рис. 3.6:

```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.axis([0, 1, 0, 1])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')

plot_roc_curve(fpr, tpr)
plt.show()
```

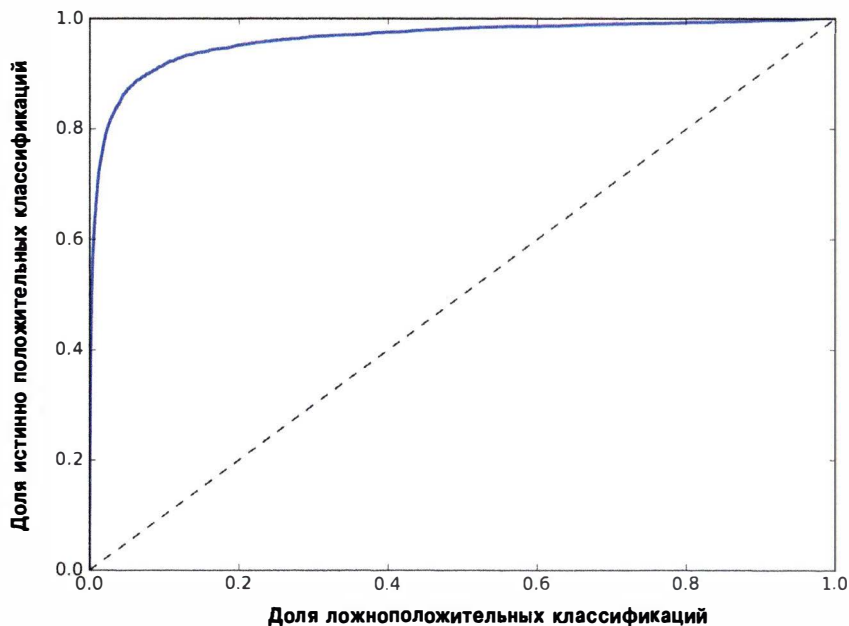


Рис. 3.6. Кривая ROC

И снова имеется соотношение: чем выше полнота (TPR), тем больше классификатор дает ложноположительных классификаций (FPR). Пунктирная линия представляет кривую ROC чисто случайного классификатора; хороший классификатор отстоит от указанной линии настолько далеко, насколько это возможно (стремясь к левому верхнему углу).

Один из способов сравнения классификаторов предусматривает измерение *площади под кривой* (Area Under the Curve — AUC). Безупречный классификатор будет иметь *площадь под кривой ROC* (ROC AUC), равную 1, тогда как чисто случайный классификатор — площадь 0.5.

В Scikit-Learn предлагается функция для расчета площади под кривой ROC:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(y_train_5, y_scores)
0.96244965559671547
```



Поскольку кривая ROC настолько похожа на кривую точности-полноты, вы можете задаться вопросом, как решать, какую из кривых использовать. Вот эмпирическое правило: всякий раз, когда положительный класс является редким или ложноположительные классификации заботят больше, чем ложноотрицательные, предпочтение должно отдаваться кривой PR, а иначе — кривой ROC. Например, взглянув на предыдущую кривую ROC (и показатель ROC AUC), вы можете подумать, что классификатор действительно хорош. Но это главным образом потому, что существует мало положительных классификаций (пятерок) в сравнении с отрицательными классификациями (не пятерками). Напротив, кривая PR дает ясно понять, что классификатор имеет возможности для улучшения (кривая могла бы стать ближе к правому верхнему углу).

Давайте обучим классификатор `RandomForestClassifier` и сравним его кривую ROC и показатель ROC AUC с классификатором `SGDClassifier`. Первым делом понадобится получить показатели для каждого образца в обучающем наборе. Но из-за особенностей работы (глава 7) класс `RandomForestClassifier` не имеет метода `decision_function()`. Взамен у него есть метод `predict_proba()`. Классификаторы Scikit-Learn обычно имеют тот или другой метод. Метод `predict_proba()` возвращает массив, который содержит строку на образец и столбец на класс, содержащий вероятность того, что заданный образец принадлежит заданному классу (скажем, 70%-ный шанс, что изображение представляет пятерку):

```
from sklearn.ensemble import RandomForestClassifier
forest_clf = RandomForestClassifier(random_state=42)
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5,
                                     cv=3, method="predict_proba")
```

Но для вычерчивания кривой ROC нужны показатели, а не вероятности. Простым решением будет применение вероятности положительного класса в качестве показателя:

```

y_scores_forest = y_probas_forest[:, 1] # показатель = вероятность
                                         # положительного класса
fpr_forest, tpr_forest, thresholds_forest =
    roc_curve(y_train_5, y_scores_forest)

```

Теперь вы готовы к вычерчиванию кривой ROC. Полезно также вычертить первую кривую ROC, чтобы посмотреть, как они соотносятся (рис. 3.7):

```

plt.plot(fpr, tpr, "b:", label="SGD")
plot_roc_curve(fpr_forest, tpr_forest, "Случайный лес")
plt.legend(loc="lower right")
plt.show()

```

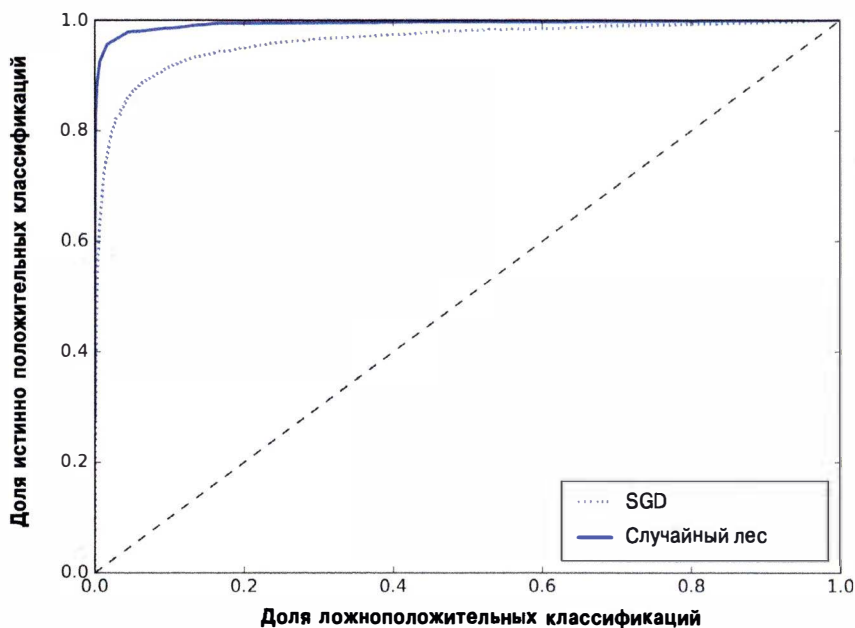


Рис. 3.7. Сравнение кривых ROC

На рис. 3.7 видно, что кривая ROC классификатора `RandomForestClassifier` выглядит гораздо лучше кривой ROC классификатора `SGDClassifier`: она расположена намного ближе к левому верхнему углу. Как результат, его показатель ROC AUC также значительно лучше:

```

>>> roc_auc_score(y_train_5, y_scores_forest)
0.99312433660038291

```

Попробуйте измерить показатели точности и полноты: вы должны обнаружить точность 98.5% и полноту 82.8%. Не так уж плохо!

Будем надеяться, что к настоящему моменту вы знаете, как обучать двоичные классификаторы, выбирать подходящую метрику для своей задачи, оценивать классификаторы с использованием перекрестной проверки, устанавливать соотношение точность/полнота, которое соответствует имеющимся потребностям, и сравнивать разнообразные модели с применением кривых ROC и показателей ROC AUC. А теперь попробуем распознать больше, чем только пятерки.

Многоклассовая классификация

В то время как двоичные классификаторы проводят различие между двумя классами, *многоклассовые классификаторы* (*multiclass classifier*), также называемые *полиномиальными классификаторами* (*multinomial classifier*), могут различать более двух классов.

Некоторые алгоритмы (такие как классификаторы методом случайных лесов или наивные байесовские классификаторы (naive Bayes classifier)) способны обращаться с множеством классов напрямую. Другие (вроде классификаторов методом опорных векторов или линейных классификаторов) являются строго двоичными классификаторами. Однако существуют различные стратегии, которые можно использовать для проведения многоклассовой классификации, применяя несколько двоичных классификаторов.

Например, один из способов создания системы, которая может группировать изображения цифр в 10 классов (от 0 до 9), заключается в том, чтобы обучить 10 двоичных классификаторов, по одному для каждой цифры (детектор нулей, детектор единиц, детектор двоек и т.д.). Затем, когда необходимо классифицировать изображение, вы получаете из каждого классификатора сумму баллов решения для данного изображения и выбираете класс, чей классификатор выдал наивысший балл. Такой прием называется стратегией “один против всех” (*one-versus-all* — *OvA*), также известной под названием “один против остальных” (*one-versus-the-rest*).

Другая стратегия предусматривает обучение двоичного классификатора для каждой пары цифр: одного для проведения различия между нулями и единицами, одного для различения нулей и двоек, одного для единиц и двоек и т.д. Это называется стратегией “один против одного” (*one-versus-one* — *OvO*). Если есть N классов, тогда понадобится обучить $N \times (N - 1) / 2$ классификаторов. Для задачи с набором данных MNIST получается, что нужно обучить 45 двоичных классификаторов! Когда вы хотите классифицировать изобра-

жение, то должны прогнать его через все 45 классификаторов и посмотреть, какой класс выиграл большинство дуэлей. Главное преимущество стратегии OvO в том, что каждый классификатор нуждается в обучении только на части обучающего набора для двух классов, которые он обязан различать.

Некоторые алгоритмы (наподобие классификаторов методом опорных векторов) плохо масштабируются с ростом размера обучающего набора, так что для таких алгоритмов стратегия OvO предпочтительнее, потому что быстрее обучить много классификаторов на небольших обучающих наборах, чем обучить несколько классификаторов на крупных обучающих наборах. Тем не менее, для большинства алгоритмов двоичной классификации предпочтительной является стратегия OvA.

Библиотека Scikit-Learn обнаруживает попытку использования алгоритма двоичной классификации для задачи многоклассовой классификации и автоматически иницирует стратегию OvA (за исключением классификаторов методом опорных векторов, для которых применяется OvO). Проверим сказанное на классификаторе `SGDClassifier`:

```
>>> sgd_clf.fit(X_train, y_train)      # y_train, не y_train_5
>>> sgd_clf.predict([some_digit])
array([ 5.])
```

Это было легко! Показанный выше код обучает `SGDClassifier` на обучающем наборе, используя исходные целевые классы от 0 до 9 (`y_train`) вместо целевых классов “пятерка против всех” (`y_train_5`). Затем он вырабатывает прогноз (в данном случае корректный). “За кулисами” библиотека Scikit-Learn фактически обучила 10 двоичных классификаторов, получила баллы решений для каждого изображения и выбрала класс с наивысшим баллом.

Чтобы убедиться в этом, вы можете вызвать метод `decision_function()`. Вместо возвращения только одной суммы баллов на образец она возвращает 10 сумм баллов, по одной на класс:

```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[ -311402.62954431, -363517.28355739, -446449.5306454 ,
        -183226.61023518, -414337.15339485,  161855.74572176,
        -452576.39616343, -471957.14962573, -518542.33997148,
        -536774.63961222]])
```

Самый высокий балл действительно соответствует классу 5:

```
>>> np.argmax(some_digit_scores)
5
>>> sgd_clf.classes_
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> sgd_clf.classes_[5]
5.0
```



Когда классификатор обучен, он сохраняет в своем атрибуте `classes_` список целевых классов, упорядоченный по значению. В этом случае индекс каждого класса в массиве `classes_` удобно соответствует самому классу (например, так уж вышло, что класс по индексу 5 оказывается классом 5), но обычно подобное везение — не частая картина.

Если вы хотите заставить Scikit-Learn применять стратегию OvO или OvA, тогда можете использовать классы `OneVsOneClassifier` или `OneVsRestClassifier`. Просто создайте экземпляр класса, передав двоичный классификатор его конструктору. Например, следующий код создает многоклассовый классификатор, применяющий стратегию OvO, на основе `SGDClassifier`:

```
>>> from sklearn.multiclass import OneVsOneClassifier
>>> ovo_clf = OneVsOneClassifier(SGDClassifier(random_state=42))
>>> ovo_clf.fit(X_train, y_train)
>>> ovo_clf.predict([some_digit])
array([ 5.])
>>> len(ovo_clf.estimators_)
45
```

Обучить `RandomForestClassifier` также легко:

```
>>> forest_clf.fit(X_train, y_train)
>>> forest_clf.predict([some_digit])
array([ 5.])
```

На этот раз библиотека Scikit-Learn не обязана запускать OvA или OvO, поскольку классификаторы методом случайных лесов способны напрямую группировать образцы во множество классов. Вы можете вызвать метод `predict_proba()`, чтобы получить список вероятностей, с которыми классификатор назначит каждый образец каждому классу:


```
>>> forest_clf.predict_proba([some_digit])
array([[ 0.1,  0. ,  0. ,  0.1,  0. ,  0.8,  0. ,  0. ,  0. ,  0.]])
```

Как видите, классификатор довольно уверен в своем прогнозе: 0.8 в 5-м элементе массива означает, что модель с 80%-ной вероятностью оценивает изображение как представляющее пятерку. Она также считает, что иначе изображение могло бы представлять ноль или тройку (с 10%-ной вероятностью).

Конечно, теперь вы хотите оценить созданные классификаторы, как обычно, используя перекрестную проверку. Давайте оценим правильность классификатора `SGDClassifier` с применением функции `cross_val_score()`:

```
>>> cross_val_score(sgd_clf, X_train, y_train,
                    cv=3, scoring="accuracy")
array([ 0.84063187,  0.84899245,  0.86652998])
```

Он дает правильность выше 84% на всех испытательных блоках. В случае использования случайного классификатора вы получили бы правильность 10%, так что это неплохой показатель, но улучшения по-прежнему возможны. Например, простое масштабирование входных данных (как обсуждалось в главе 2) увеличивает правильность до более 90%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype(np.float64))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train,
                    cv=3, scoring="accuracy")
array([ 0.91011798,  0.90874544,  0.906636  ])
```

Анализ ошибок

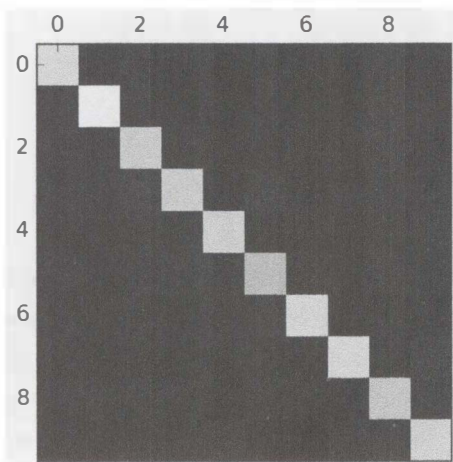
Разумеется, в реальном проекте вы бы следовали шагам контрольного перечня для проекта МО (см. приложение Б): исследование вариантов подготовки данных, опробование множества моделей, включение в окончательный список лучших моделей и точная настройка их гиперпараметров с применением `GridSearchCV`, а также максимально возможная автоматизация, как делалось в предыдущей главе. Здесь мы будем предполагать, что вы отыскиали перспективную модель и хотите найти способы ее усовершенствования. Одним таким способом является анализ типов ошибок, допускаемых моделью.

Первым делом вы можете взглянуть на матрицу неточностей. Вам необходимо выработать прогнозы, используя функцию `cross_val_predict()`, и затем вызвать функцию `confusion_matrix()`, как вы поступали ранее:

```
>>> y_train_pred = cross_val_predict(sgd_clf, X_train_scaled,
                                     y_train, cv=3)
>>> conf_m = confusion_matrix(y_train, y_train_pred)
>>> conf_m
array([[5725,   3,   24,   9,  10,  49,  50,  10,  39,   4],
       [  2, 6493,  43,  25,   7,  40,   5,  10, 109,   8],
       [ 51,  41, 5321, 104,  89,  26,  87,  60, 166,  13],
       [ 47,  46, 141, 5342,   1, 231,  40,  50, 141,  92],
       [ 19,  29,  41,  10, 5366,   9,  56,  37,  86, 189],
       [ 73,  45,  36, 193,  64, 4582, 111,  30, 193,  94],
       [ 29,  34,  44,   2,  42,  85, 5627,  10,  45,   0],
       [ 25,  24,  74,  32,  54,  12,   6, 5787,  15, 236],
       [ 52, 161,  73, 156,  10, 163,  61,  25, 5027, 123],
       [ 43,  35,  26,  92, 178,  28,   2, 223,  82, 5240]])
```

Как много чисел. Часто гораздо удобнее смотреть на представление матрицы неточностей в виде изображения, для чего применяется функция `matshow()` из Matplotlib:

```
plt.matshow(conf_m , cmap=plt.cm.gray)
plt.show()
```



Итоговая матрица неточностей выглядит неплохо, т.к. большинство изображений расположено на главной диагонали, что означает их корректную классификацию. Пятерки чуть темнее других цифр, что могло бы означать наличие в наборе данных небольшого количества изображений пятерок или не настолько хорошую работу классификатора на пятерках, как на других цифрах. В действительности вы можете проверить оба факта.

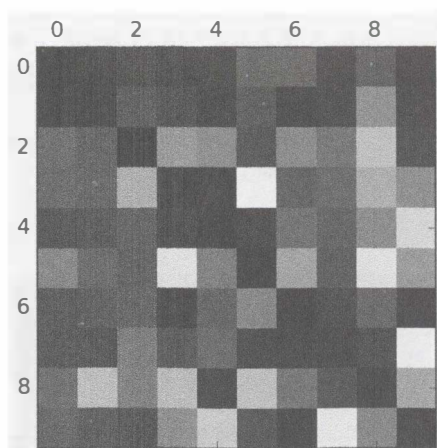
Давайте сконцентрируем график на ошибках. Прежде всего, каждое значение в матрице неточностей понадобится разделить на число изображений

в соответствующем классе, чтобы можно было сравнивать частоту ошибок вместо их абсолютного количества (что сделало бы избыливающие ошибками классы несправедливо плохими):

```
row_sums = conf_mx.sum(axis=1, keepdims=True)
norm_conf_mx = conf_mx / row_sums
```

Давайте заполним диагональ нулями, сохранив только ошибки, и вычертим результирующий график:

```
np.fill_diagonal(norm_conf_mx, 0)
plt.matshow(norm_conf_mx, cmap=plt.cm.gray)
plt.show()
```



Теперь вы можете ясно видеть типы ошибок, которые допускает классификатор. Вспомните, что строки представляют фактические классы, а столбцы — спрогнозированные классы. Столбцы для классов 8 и 9 довольно светлые, а это говорит о том, что многие изображения неправильно классифицированы как восьмерки или девятки. Подобным же образом строки для классов 8 и 9 также довольно светлые, сообщая о том, что восьмерки и девятки часто путались с другими цифрами. И наоборот, некоторые строки изрядно темные, скажем, строка 1: это означает, что большинство единиц классифицировано корректно (несколько были перепутаны с восьмерками, вот и все). Обратите внимание, что ошибки не являются идеально симметричными; например, существует больше пятерок, неправильно классифицированных как восьмерки, чем наоборот.

Анализ матрицы неточностей часто может давать подсказки о способах улучшения классификатора. Глядя на этот график, кажется, что усилия долж-

ны быть направлены на усовершенствование классификации восьмерок и девяток, а также устранение характерной путаницы между тройками и пятерками. Скажем, вы могли бы попробовать накопить больше обучающих данных для указанных цифр. Или же вы могли бы сконструировать новые признаки, которые помогали бы классификатору, к примеру, написав алгоритм для подсчета количества замкнутых контуров (8 имеет два таких контура, 6 — один, 5 — ни одного). Либо вы могли бы предварительно обработать изображения (скажем, используя библиотеку Scikit-Image, Pillow или OpenCV), чтобы паттерны вроде замкнутых контуров стали более заметными.

Анализ индивидуальных ошибок также может оказаться хорошим способом получения знаний о том, что ваш классификатор делает и почему он отказывает, но он более сложный и затратный в плане времени. Давайте нарисует примеры троек и пятерок (функция `plot_digits()` просто применяет функцию `imshow()` из Matplotlib; за деталями обращайтесь к тетради Jupyter для настоящей главы):

```
cl_a, cl_b = 3, 5
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]

plt.figure(figsize=(8,8))
plt.subplot(221); plot_digits(X_aa[:25], images_per_row=5)
plt.subplot(222); plot_digits(X_ab[:25], images_per_row=5)
plt.subplot(223); plot_digits(X_ba[:25], images_per_row=5)
plt.subplot(224); plot_digits(X_bb[:25], images_per_row=5)
plt.show()
```



Два блока 5×5 слева показывают изображения, классифицированные как тройки, а два блока 5×5 справа — изображения, классифицированные как пятерки. Несколько цифр, которые классификатор неправильно понял (т.е. блоки слева внизу и справа вверху), написаны настолько плохо, что даже человек испытывал бы затруднения с их классификацией (к примеру, пятерка в строке 8 и столбце 1 действительно напоминает тройку). Однако большинство неправильно классифицированных изображений выглядят для нас как очевидные ошибки, и нелегко понять, почему классификатор допустил ошибки³. Причина в том, что мы используем простой класс `SGDClassifier`, который представляет собой линейную модель. Он всего лишь назначает каждой точке вес по классу и когда встречает новое изображение, то суммирует взвешенные интенсивности точек, чтобы получить показатель для каждого класса. Таким образом, поскольку тройки и пятерки отличаются только немногими точками, модель будет легко их путать.

Главное отличие между тройками и пятерками — положение маленькой линии, которая соединяет верхнюю линию с нижней дугой. Если вы изобразите тройку, слегка сместив соединение влево, тогда классификатор может опознать ее как пятерку, и наоборот. Другими словами, данный классификатор крайне чувствителен к сдвигам и поворотам изображений. Следовательно, один из способов сокращения путаницы между тройками и пятерками предусматривал бы предварительную обработку изображений для гарантии, что они хорошо отцентрированы и не слишком повернуты. Возможно, этот способ также поможет уменьшить число других ошибок.

Многозначная классификация

До сих пор каждый образец всегда назначался только одному классу. В ряде случаев может быть желательно, чтобы классификатор выдавал множество классов для каждого образца. Например, возьмем классификатор распознавания лиц: что он должен делать в ситуации, если на той же самой фотографии распознает несколько людей? Конечно, он обязан прикреплять по одной метке на распознанную особу. Скажем, классификатор был обучен распознавать три лица, Алису, Боба и Чарли; позже, когда он видит фотогра-

³ Но не забывайте о том, что наш мозг является фантастической системой распознавания паттернов, и наша зрительная система выполняет массу сложной предварительной обработки, прежде чем любая информация проникнет в наше сознание, поэтому тот факт, что все выглядит простым, вовсе не означает, что так оно и есть.

фию Алисы и Чарли, то должен выдать [1, 0, 1] (т.е. “Алиса — да, Боб — нет и Чарли — да”). Система классификации такого рода, которая выдает множество двоичных меток, называется системой *многозначной классификации* (*multilabel classification*).

Прямо сейчас мы не будем глубоко вдаваться в распознавание лиц, но в иллюстративных целях рассмотрим более простой пример:

```
from sklearn.neighbors import KNeighborsClassifier
y_train_large = (y_train >= 7)
y_train_odd = (y_train % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]
knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

Приведенный код создает массив `y_multilabel`, содержащий две целевые метки для каждого изображения цифры: первая метка указывает, является ли цифра большой (7, 8 или 9), а вторая — нечетная ли цифра. Создается экземпляр классификатора `KNeighborsClassifier` (поддерживающий многозначную классификацию, но не все классификаторы делают это), который обучается с применением массива множества целей. Теперь можно вырабатывать прогноз; обратите внимание, что он выводит две метки:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]], dtype=bool)
```

И прогноз правильный! Цифра 5 на самом деле небольшая (`False`) и нечетная (`True`).

Существует много способов оценки многозначного классификатора и выбор надлежащей метрики крайне зависит от проекта. Например, один из подходов заключается в том, чтобы определить для каждой отдельной метки меру F_1 (или любую другую метрику двоичных классификаторов, которые обсуждались ранее) и затем просто подсчитать среднюю сумму очков. Следующий код вычисляет среднюю меру F_1 по всем меткам:

```
>>> y_train_knn_pred =
    cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.97709078477525002
```

Здесь предполагается равная степени важности всех меток, что может быть и не так. В частности, если фотографий Алисы намного больше, чем

Боба или Чарли, тогда вы можете предоставить больший вес оценке классификатора на фотографиях Алисы. Простой вариант — дать каждой метке вес, равный ее *поддержке* (т.е. количеству образцов с такой целевой меткой). Чтобы сделать это, просто установите `average="weighted"` в предыдущем коде⁴.

Многовыходовая классификация

Последний тип задачи классификации, который мы здесь обсудим, называется *многовыходовой-многоклассовой классификацией* (или просто *многовыходовой классификацией* (*multioutput classification*)). Это просто обобщение многозначной классификации, где каждая метка может быть многоклассовой (т.е. способна иметь более двух возможных значений).

В целях иллюстрации давайте построим систему, которая устраняет шум в изображениях. На входе она будет получать зашумленное изображение цифры и (в перспективе) выдавать чистое изображение цифры, представленное в виде массива интенсивностей пикселей, точно как в случае изображений MNIST. Обратите внимание, что вывод классификатора является многозначным (одна метка на пиксель) и каждая метка может иметь множество значений (интенсивность пикселя колеблется от 0 до 255). Таким образом, это пример системы многовыходовой классификации.



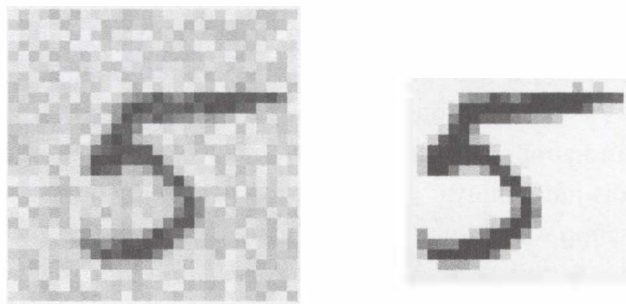
Граница между классификацией и регрессией временами размыта, как в рассмотренном примере. Возможно, прогнозирование интенсивности пикселей ближе к регрессии, чем к классификации. Кроме того, многовыходовые системы не ограничены задачами классификации; вы могли бы даже иметь систему, которая выдает множество меток на образец, включая метки классов и метки значений.

Начнем с создания обучающего и испытательного наборов, взяв изображения MNIST и добавив шум к интенсивностям их пикселей с использованием функции `randint()` из NumPy. Целевые изображения будут исходными изображениями:

⁴ Библиотека Scikit-Learn предлагает несколько других вариантов усреднения и метрик многозначных классификаторов; дополнительные сведения ищите в документации.

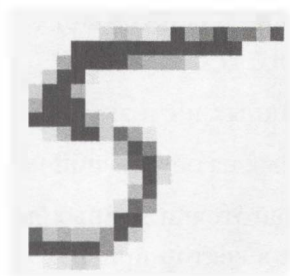
```
noise = np.random.randint(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise
noise = np.random.randint(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise
y_train_mod = X_train
y_test_mod = X_test
```

Давайте одним глазом взглянем на изображение из испытательного набора (да-да, мы суем свой нос в испытательные данные, так что сейчас вы должны грозно нахмурить брови):



Слева показано зашумленное входное изображение, а справа — чистое целевое изображение. Обучим классификатор и очистим изображение:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```



Выглядит достаточно близким к цели! На этом наше путешествие в классификацию завершено. Теперь вы должны знать, каким образом выбирать хорошие метрики для задач классификации, устанавливать подходящее соотношение точность/полнота, сравнивать классификаторы и в целом строить эффективные системы классификации для разнообразных задач.

Упражнения

1. Попробуйте построить классификатор для набора данных MNIST, который достигает 97%-ной правильности на испытательном наборе. Подсказка: с этой задачей неплохо справится `KNeighborsClassifier`; вам просто нужно найти подходящие значения гиперпараметров (воспользуйтесь решетчатым поиском для гиперпараметров `weights` и `n_neighbors`).
2. Напишите функцию, которая может смещать изображение MNIST в любом направлении (влево, вправо, вверх или вниз) на один пиксель⁵. Затем для каждого изображения в обучающем наборе создайте четыре смещенных копии (по одной на каждое направление) и добавьте их в обучающий набор. Наконец, обучите на этом расширенном обучающем наборе свою наилучшую модель и измерьте ее правильность на испытательном наборе. Вы должны заметить, что модель стала работать даже лучше! Такой прием искусственного увеличения обучающего набора называется *дополнением данных* (*data augmentation*) или *расширением обучающего набора* (*training set expansion*).
3. Займитесь набором данных *Titanic*. Хорошим местом для старта будет веб-сайт Kaggle (<https://www.kaggle.com/c/titanic>).
4. Постройте классификатор спама (более серьезное упражнение).
 - Загрузите примеры спама и нормальных сообщений из открытых наборов данных Apache SpamAssassin (<https://spamassassin.apache.org/old/publiccorpus/>).
 - Распакуйте наборы данных и ознакомьтесь с форматом данных.
 - Разделите наборы данных на обучающий набор и испытательный набор.
 - Создайте конвейер подготовки данных для преобразования каждого почтового сообщения в вектор признаков. Ваш конвейер подготовки данных должен трансформировать почтовое сообщение в (разреженный) вектор, указывающий наличие или отсутствие каждого возможного слова. Например, если все почтовые сообщения содержат только

⁵ Вы можете применить функцию `shift()` из модуля `scipy.ndimage.interpolation`. Например, `shift(image, [2, 1], cval=0)` смещает изображение `image` на 2 пикселя вниз и 1 пиксель вправо.

четыре слова, “Hello”, “how”, “are”, “you”, тогда сообщение “Hello you Hello Hello you” было бы преобразовано в вектор [1, 0, 0, 1] (означающий [“Hello” присутствует, “how” отсутствует, “are” отсутствует, “you” присутствует]), или [3, 0, 0, 2], если вы предпочитаете подсчитывать количество вхождений каждого слова.

- Вы можете пожелать добавить гиперпараметры к своему конвейеру подготовки данных, чтобы управлять тем, производить ли разбор заголовков почтовых сообщений, приводить ли символы каждого сообщения к нижнему регистру, удалять ли символы пунктуации, замещать ли URL строкой “URL”, заменять ли все числа строкой “NUMBER” или даже выполнять ли *морфологический поиск* (stemming), т.е. исключение окончаний слов (для этого доступно несколько библиотек Python).
- Затем испытайте несколько классификаторов и посмотрите, можете ли вы построить замечательный классификатор спама с высокими показателями полноты и точности.

Решения упражнений доступны в онлайн-тетрадах Jupyter по адресу <https://github.com/ageron/handson-ml>.

Обучение моделей

До сих пор мы трактовали модели МО и их алгоритмы обучения главным образом как черные ящики. Если вы прорабатывали какие-то упражнения из предшествующих глав, то могли быть удивлены тем, насколько много удалось сделать, ничего не зная о “внутренней кухне”. Вы оптимизировали регрессионную систему, усовершенствовали классификатор изображений с цифрами и даже построили с нуля классификатор спама, не располагая сведениями о том, как это фактически функционирует. Действительно, во многих ситуациях вам вовсе не нужно знать детали реализации.

Тем не менее, хорошее понимание того, каким образом все работает, может помочь быстро нацелиться на подходящую модель, использовать правильный алгоритм обучения и выбрать оптимальный набор гиперпараметров для имеющейся задачи. Понимание “внутренней кухни” также будет содействовать более эффективной отладке проблем и анализу ошибок. Наконец, большинство тем, обсуждаемых в настоящей главе, будут важны для освоения, построения и обучения нейронных сетей (рассматриваются в части II книги).

В этой главе мы начнем с исследования *линейной регрессионной модели* (*linear regression model*), одной из простейших доступных моделей. Мы обсудим два очень разных способа ее обучения.

- Применение прямого уравнения в *аналитическом виде*, непосредственно вычисляющего параметры модели, которые лучше всего подгоняют модель к обучающему набору (т.е. параметры модели, сводящие к минимуму значение функции издержек на обучающем наборе).
- Использование подхода итеративной оптимизации, называемого *градиентным спуском* (*Gradient Descent* — *GD*), который постепенно корректирует параметры модели, чтобы довести до минимума значение функции издержек на обучающем наборе, в итоге сходясь к тому же са-

тому набору параметров, что и при первом способе. Мы рассмотрим несколько вариантов градиентного спуска, которые периодически будут применяться при изучении нейронных сетей в части II: пакетный (Batch GD — BGD), мини-пакетный (mini-batch GD) и стохастический (stochastic GD).

Затем мы взглянем на *полиномиальную регрессию* (*polynomial regression*), более сложную модель, которая может подгоняться к нелинейным наборам данных. Поскольку такая модель имеет больше параметров, чем линейная регрессионная модель, она более предрасположена к переобучению обучающими данными, поэтому мы посмотрим, как обнаруживать подобную ситуацию, используя *кривые обучения* (*learning curve*), и опишем несколько приемов регуляризации, которые способны снизить риск переобучения обучающим набором.

В заключение мы рассмотрим еще две модели, широко применяемые для задач классификации: *логистическую регрессию* (*logistic regression*) и *многопеременную логистическую регрессию* (*softmax regression*).



В главе есть довольно много математических уравнений, в которых используются основные понятия линейной алгебры и линейных вычислений. Для понимания этих уравнений вы должны знать, что такое векторы и матрицы, как их транспонировать, что собой представляет скалярное произведение, что такое обратная матрица и частные производные. Если вы не знакомы с указанными концепциями, тогда изучите вводные обучающие руководства по линейной алгебре и линейным вычислениям, доступные в виде тетрадей Jupyter в сопровождающих онлайн-материалах. Если вы из тех, кто по-настоящему не выносит математику, то все равно проработайте эту главу, просто пропуская уравнения; есть надежда, что одного текста окажется достаточно для того, чтобы помочь вам понять большинство концепций.

Линейная регрессия

В главе 1 рассматривалась простая регрессионная модель для удовлетворенности жизнью:

$$\text{удовлетворенность_жизнью} = \theta_0 + \theta_1 \times \text{ВВП_на_душу_населения}.$$

Эта модель представляет собой всего лишь линейную функцию от входного признака *ВВП_на_душу_населения*. θ_0 и θ_1 — параметры модели.

Как правило, линейная модель вырабатывает прогноз, просто вычисляя взвешенную сумму входных признаков плюс константы под названием *член смещения* (*bias term*), также называемой *свободным членом* (*intercept term*), как показано в уравнении 4.1.

Уравнение 4.1. Прогноз линейной регрессионной модели

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

- \hat{y} — спрогнозированное значение.
- n — количество признаков.
- x_i — значение i -того признака.
- θ_j — j -тый параметр модели (включая член смещения θ_0 и веса признаков $\theta_1, \theta_2, \dots, \theta_n$).

Это может быть записано гораздо компактнее с применением векторизованной формы, как показано в уравнении 4.2.

Уравнение 4.2. Прогноз линейной регрессионной модели (векторизованная форма)

$$\hat{y} = h_{\theta}(\mathbf{x}) = \theta^T \cdot \mathbf{x}$$

- θ — *вектор параметров* модели, содержащий член смещения θ_0 и веса признаков от θ_1 до θ_n .
- θ^T — транспонированный θ (вектор-строка вместо вектора-столбца).
- \mathbf{x} — *вектор признаков* образца, содержащий от x_0 до x_n , где x_0 всегда равно 1.
- $\theta^T \cdot \mathbf{x}$ — скалярное произведение θ^T и \mathbf{x} .
- h_{θ} — функция гипотезы, использующая параметры модели θ .

Хорошо, мы имеем линейную регрессионную модель, но как мы будем ее обучать? Вспомните, что обучение модели означает установку ее параметров так, чтобы модель была наилучшим образом подогнана к обучающему набору. Для этой цели нам первым делом нужна мера того, насколько хорошо (или плохо) модель подогнана к обучающим данным. В главе 2 мы выяснили, что самой распространенной мерой производительности регрессионной мо-

дели является квадратный корень из среднеквадратической ошибки (RMSE) (см. уравнение 2.1). Следовательно, для обучения линейной регрессионной модели необходимо найти значение θ , которое сводит к минимуму RMSE. На практике проще довести до минимума *среднеквадратическую ошибку (Mean Squared Error — MSE)*, чем RMSE, что приведет к тому же самому результату (поскольку значение, которое сводит к минимуму функцию, также сводит к минимуму ее квадратный корень)¹.

Ошибка MSE гипотезы линейной регрессии h_θ на обучающем наборе X вычисляется с использованием уравнения 4.3.

Уравнение 4.3. Функция издержек MSE для линейной регрессионной модели

$$\text{MSE}(X, h_\theta) = \frac{1}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)})^2$$

Большинство обозначений было представлено в главе 2 (см. врезку “Обозначения”). Единственное отличие в том, что вместо просто h мы пишем h_θ , подчеркивая факт параметризации модели по вектору θ . Чтобы упростить обозначения, мы будем записывать $\text{MSE}(\theta)$ вместо $\text{MSE}(X, h_\theta)$.

Нормальное уравнение

Для нахождения значения θ , которое сводит к минимуму функцию издержек, имеется *решение в аналитическом виде (closed-form solution)* — иными словами, математическое уравнение, дающее результат напрямую. Оно называется *нормальным уравнением (normal equation)* и представлено в уравнении 4.4².

¹ Часто бывает так, что алгоритм обучения будет пытаться оптимизировать другую функцию, отличающуюся от меры производительности, которая применяется для оценки финальной модели. Обычно причины связаны с тем, что такую функцию легче вычислять, она обладает удобными свойствами по установлению различий, которых лишена мера производительности, или требуется ограничить модель во время обучения, как будет показано при обсуждении регуляризации.

² Доказательство того, что уравнение возвращает значение θ , которое доводит до минимума функцию издержек, выходит за рамки настоящей книги.

Уравнение 4.4. Нормальное уравнение

$$\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$$

- $\hat{\theta}$ — значение θ , которое сводит к минимуму функцию издержек.
- y — вектор целевых значений, содержащий от $y^{(1)}$ до $y^{(m)}$.

Давайте сгенерируем данные, выглядящие как линейные, для проверки на этом уравнении (рис. 4.1):

```
import numpy as np
X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

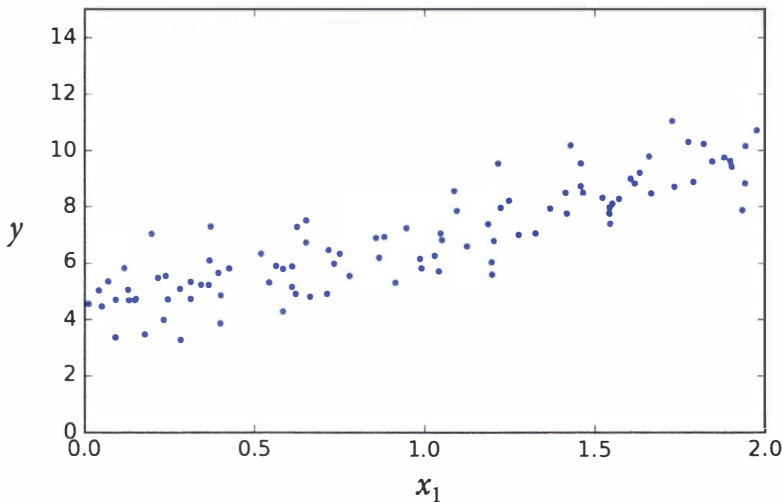


Рис. 4.1. Случайно сгенерированный набор линейных данных

Теперь вычислим θ с применением нормального уравнения. Мы будем использовать функцию `inv()` из модуля линейной алгебры (linear algebra), входящего в состав NumPy (`np.linalg`), для получения обратной матрицы и метод `dot()` для умножения матриц:

```
X_b = np.c_[np.ones((100, 1)), X] # добавляет x0 = 1 к каждому образцу
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Фактической функцией, которая применялась для генерации данных, была $y = 4 + 3x_1 + \text{гауссов шум}$. Посмотрим, что нашло уравнение:

```
>>> theta_best
array([[ 4.21509616],
       [ 2.77011339]])
```

Мы надеялись на $\theta_0 = 4$ и $\theta_1 = 3$, а не $\theta_0 = 4.215$ и $\theta_1 = 2.770$. Довольно близко, но шум делает невозможным восстановление точных параметров исходной функции.

Далее можно вырабатывать прогнозы, используя $\hat{\theta}$:

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = np.c_[np.ones((2, 1)), X_new] # добавляет x0 = 1
                                              # к каждому образцу
>>> y_predict = X_new_b.dot(theta_best)
>>> y_predict
array([[ 4.21509616],
       [ 9.75532293]])
```

Давайте вычертим прогнозы этой модели (рис. 4.2):

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

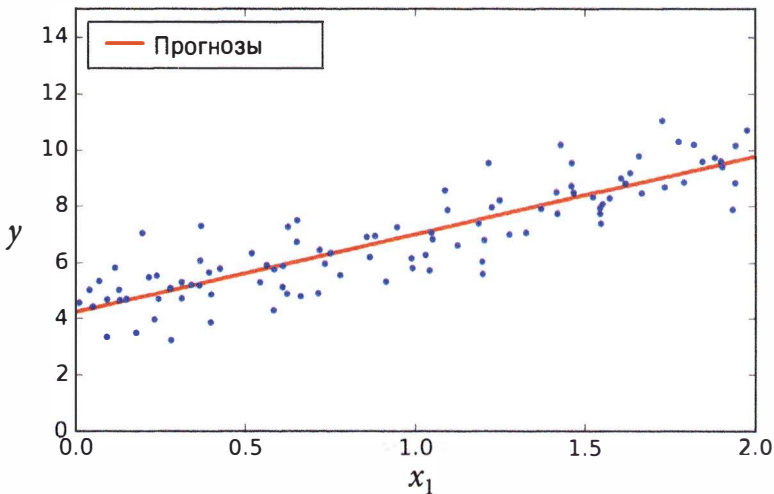


Рис. 4.2. Прогнозы линейной регрессионной модели

Эквивалентный код, в котором применяется Scikit-Learn, выглядит примерно так³:

³ Обратите внимание, что Scikit-Learn отделяет член смещения (`intercept_`) от весов признаков (`coef_`).

```
>>> from sklearn.linear_model import LinearRegression
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 4.21509616]), array([[ 2.77011339]]))
>>> lin_reg.predict(X_new)
array([[ 4.21509616],
       [ 9.75532293]])
```

Вычислительная сложность

Нормальное уравнение вычисляет инверсию $X^T \cdot X$, которая представляет собой матрицу $n \times n$ (где n — количество признаков). *Вычислительная сложность* (*computational complexity*) инвертирования такой матрицы обычно составляет примерно от $O(n^{2.4})$ до $O(n^3)$ (в зависимости от реализации). Другими словами, если вы удвоите количество признаков, то умножите время вычислений на значение приблизительно от $2^{2.4} = 5.3$ до $2^3 = 8$.



Нормальное уравнение становится очень медленным, когда количество признаков серьезно возрастает (скажем, до 100 000).

В качестве положительной стороны следует отметить, что это уравнение является линейным относительно числа образцов в обучающем наборе ($O(m)$), а потому оно эффективно обрабатывает крупные обучающие наборы, если только они могут уместиться в память.

К тому же после обучения линейной регрессионной модели (с использованием нормального уравнения или любого другого алгоритма) прогнозирование будет очень быстрым: вычислительная сложность линейна в отношении и количества образцов, на которых необходимо вырабатывать прогнозы, и числа признаков. Другими словами, выработка прогнозов на удвоенном количестве образцов (или удвоенном числе признаков) будет примерно в два раза дольше.

Теперь мы взглянем на совершенно другие способы обучения линейной регрессионной модели, лучше подходящие в случаях, когда имеется большое число признаков или слишком много образцов в обучающем наборе, чтобы уместиться в память.

Градиентный спуск

Градиентный спуск представляет собой самый общий алгоритм оптимизации, способный находить оптимальные решения широкого диапазона задач. Основная идея градиентного спуска заключается в том, чтобы итеративно подстраивать параметры для сведения к минимуму функции издержек.

Предположим, вы потерялись в горах в густом тумане; вы способны прощупывать только крутизну поверхности под ногами. Хорошая стратегия быстро добраться до дна долины предусматривает движение вниз по самому крутому спуску (не поступайте так в реальных горах, потому что быстро — не значит безопасно; ходите по указанным на карте тропам — *примеч. пер.*). Это в точности то, что делает *градиентный спуск*: он измеряет локальный градиент функции ошибок применительно к вектору параметров θ и двигается в направлении убывающего градиента. Как только градиент становится нулевым, вы достигли минимума!

Выражаясь более конкретно, вы начинаете с заполнения вектора θ случайными значениями (т.н. *случайная инициализация*). Затем вы постепенно улучшаете его, предпринимая по одному маленькому шагу за раз и на каждом шаге пытаетесь снизить функцию издержек (например, MSE) до тех пор, пока алгоритм не *сойдется* в минимуме (рис. 4.3).

Важным параметром в *градиентном спуске* является размер шагов, определяемый гиперпараметром *скорости обучения* (*learning rate*). Если скорость обучения слишком мала, тогда алгоритму для сведения придется пройти через множество итераций, что потребует длительного времени (рис. 4.4).



Рис. 4.3. Градиентный спуск

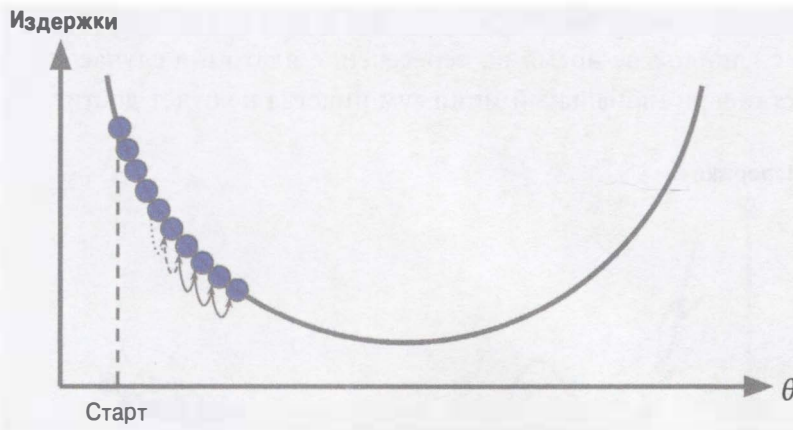


Рис. 4.4. Скорость обучения слишком мала

С другой стороны, если скорость обучения слишком высока, тогда вы можете перескочить долину и оказаться на другой стороне, возможно даже выше, чем находились ранее. Это способно сделать алгоритм расходящимся, что приведет к выдаче постоянно увеличивающихся значений и неудаче в поиске хорошего решения (рис. 4.5).

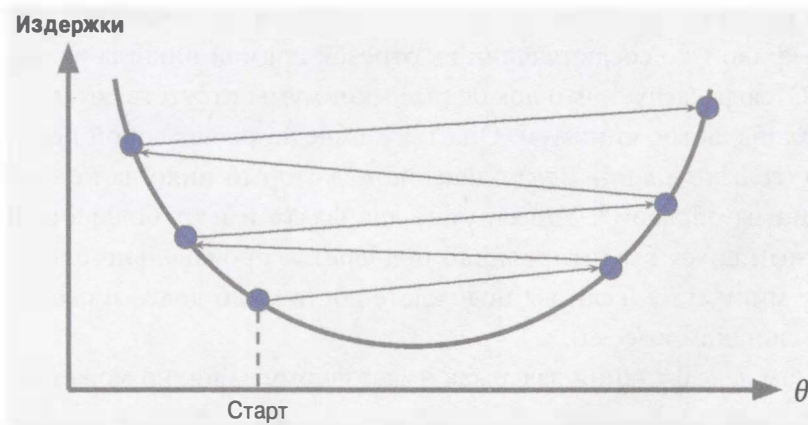


Рис. 4.5. Скорость обучения слишком высока

Наконец, не все функции издержек выглядят как точные правильные чаши. Могут существовать впадины, выступы, плато и самые разнообразные участки нерегулярной формы, которые крайне затрудняют сходжение к минимуму. На рис. 4.6 проиллюстрированы две главные проблемы с *градиентным спуском*: если случайная инициализация начинает алгоритм слева, то он сойдется в точке *локального минимума*, который не настолько

хорош как *глобальный минимум*. Если алгоритм начнется справа, тогда он потратит очень долгое время на пересечение плато и в случае его слишком ранней остановки глобальный минимум никогда не будет достигнут.

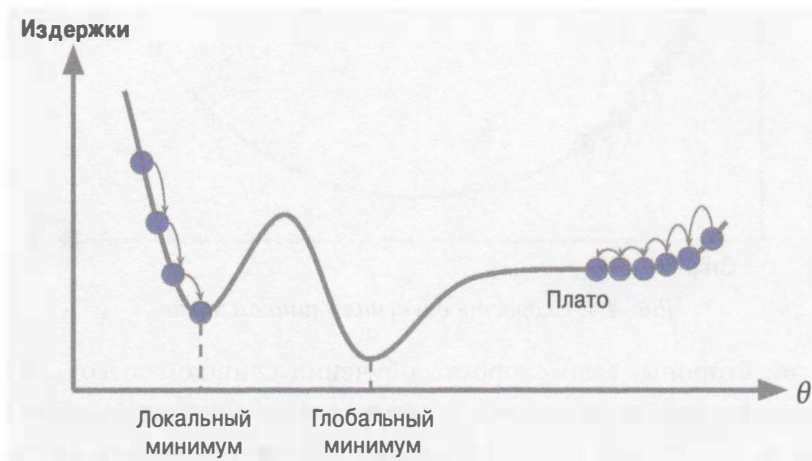


Рис. 4.6. Просчеты градиентного спуска

К счастью, функция издержек MSE для линейной регрессионной модели является *выпуклой функцией (convex function)*, т.е. если выбрать любые две точки на кривой, то соединяющий их отрезок прямой никогда не пересекает кривую. Отсюда следует, что локальные минимумы отсутствуют, а есть только один глобальный минимум. Она также представляет собой *непрерывную функцию (continuous function)* с наклоном, который никогда не изменяется неожиданным образом⁴. Упомянутые два факта имеют большое значение: градиентный спуск гарантированно подберется произвольно близко к глобальному минимуму (если вы подождете достаточно долго и скорость обучения не слишком высока).

На самом деле функция издержек имеет форму чаши, но может быть продолговатой чашей, если масштабы признаков сильно отличаются. На рис. 4.7 показан градиентный спуск на обучающем наборе, где признаки 1 и 2 имеют тот же самый масштаб (слева), и на обучающем наборе, где признак 1 содержит гораздо меньшие значения, чем признак 2 (справа)⁵.

⁴ Выражаясь формально, ее производная является *липшиц-непрерывной (Lipschitz continuous)*.

⁵ Так как признак 1 меньше, для воздействия на функцию издержек требуется большее изменение в θ_1 , что и объясняет вытянутость чаши вдоль оси θ_1 .

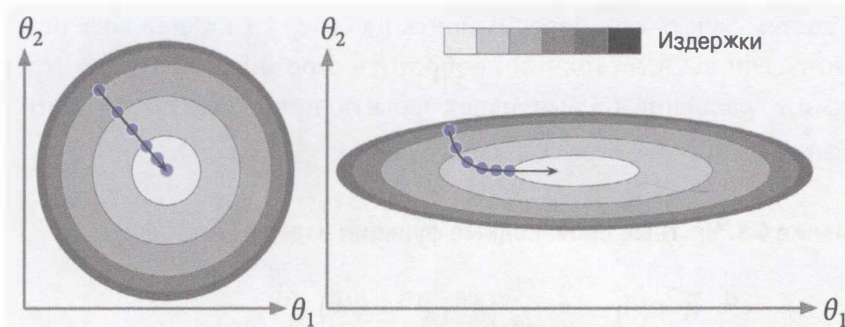


Рис. 4.7. Градиентный спуск с масштабированием признаков и без него

Как видите, слева алгоритм градиентного спуска устремляется прямо к минимуму, из-за чего достигает его быстро, а справа он сначала движется в направлении, которое практически перпендикулярно направлению к глобальному минимуму, и заканчивает длинным маршем по почти плоской долине. Минимум в итоге достигается, но за долгое время.



Когда используется градиентный спуск, вы должны обеспечить наличие у всех признаков похожего масштаба (например, с применением класса `StandardScaler` из Scikit-Learn), иначе он потребует гораздо большего времени на сходжение.

Диаграмма на рис. 4.7 также демонстрирует тот факт, что обучение модели означает поиск комбинации параметров модели, которая сводит к минимуму функцию издержек (на обучающем наборе). Это поиск в *пространстве параметров* модели: чем больше параметров имеет модель, тем больше будет измерений в пространстве параметров и тем труднее окажется поиск: искать иголку в 300-мерном стоге сена намного сложнее, чем в трехмерном. К счастью, поскольку функция издержек выпуклая в случае линейной регрессии, иголка находится просто на дне чаши.

Пакетный градиентный спуск

Чтобы реализовать градиентный спуск, вам понадобится вычислить градиент функции издержек в отношении каждого параметра модели θ_j . Другими словами, вам необходимо подсчитать, насколько сильно функция издержек будет изменяться при небольшом изменении θ_j . Результат называется *частной производной* (*partial derivative*). Это похоже на то, как задать вопрос: “Каков угол наклона горы под моими ногами, если я стою лицом на

восток?”, затем задать его, повернувшись на север (и т.д. для всех остальных измерений, если вы в состоянии вообразить себе мир с более чем тремя измерениями). Уравнение 4.5 вычисляет частную производную функции издержек в отношении параметра θ_j , известную как $\frac{\partial}{\partial \theta_j} \text{MSE}(\theta)$.

Уравнение 4.5. Частные производные функции издержек

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\theta) = \frac{2}{m} \sum_{i=1}^m (\theta^T \cdot \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Вместо вычисления таких частных производных по отдельности вы можете воспользоваться уравнением 4.6, чтобы вычислить их все сразу. **Вектор-градиент** (*gradient vector*), известный как $\nabla_{\theta} \text{MSE}(\theta)$, содержит все частные производные функции издержек (по одной для каждого параметра модели).

Уравнение 4.6. Вектор-градиент функции издержек

$$\nabla_{\theta} \text{MSE}(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\theta) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\theta) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\theta) \end{pmatrix} = \frac{2}{m} \mathbf{X}^T \cdot (\mathbf{X} \cdot \theta - \mathbf{y})$$



Обратите внимание, что формула включает в себя вычисления с полным обучающим набором \mathbf{X} на каждом шаге градиентного спуска! Вот почему алгоритм называется **пакетным градиентным спуском** (*batch gradient descent*): на каждом шаге он потребляет целый пакет обучающих данных. В результате он оказывается крайне медленным на очень крупных обучающих наборах (но вскоре мы рассмотрим гораздо более быстрые алгоритмы градиентного спуска). Однако градиентный спуск хорошо масштабируется в отношении количества признаков; обучение линейной регрессионной модели при наличии сотен тысяч признаков проходит намного быстрее с применением градиентного спуска, чем с использованием нормального уравнения.

Имея вектор-градиент, который указывает вверх, вы просто двигаетесь в противоположном направлении вниз. Это означает вычитание $\nabla_{\theta}\text{MSE}(\theta)$ из θ . Именно здесь в игру вступает скорость обучения η ⁶: умножение вектора-градиента на η дает размер шага вниз (уравнение 4.7).

Уравнение 4.7. Шаг градиентного спуска

$$\theta \text{ (следующий шаг)} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Давайте взглянем на быструю реализацию данного алгоритма:

```
eta = 0.1                                # скорость обучения
n_iterations = 1000
m = 100
theta = np.random.randn(2,1)            # случайная инициализация
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients
```

Задача была не слишком сложной! Посмотрим на результирующее значение `theta`:

```
>>> theta
array([[ 4.21509616],
       [ 2.77011339]])
```

Эй, ведь это в точности то, что нашло нормальное уравнение! Градиентный спуск работает прекрасно. Но что, если применить другую скорость обучения `eta`? На рис. 4.8 показаны первые 10 шагов градиентного спуска, использующих три разных скорости обучения (пунктирная линия представляет начальную точку).

Слева скорость обучения слишком низкая: в конце концов, алгоритм достигнет решения, но это займет долгое время. Посредине скорость обучения выглядит довольно хорошей: алгоритм сходится к решению всего за несколько итераций. Справа скорость обучения чересчур высокая: алгоритм расходится, беспорядочно перескакивая с места на место и фактически с каждым шагом все больше удаляясь от решения.

Для нахождения хорошей скорости обучения вы можете применять решетчатый поиск (см. главу 2).

⁶ Эта (η) — седьмая буква греческого алфавита.

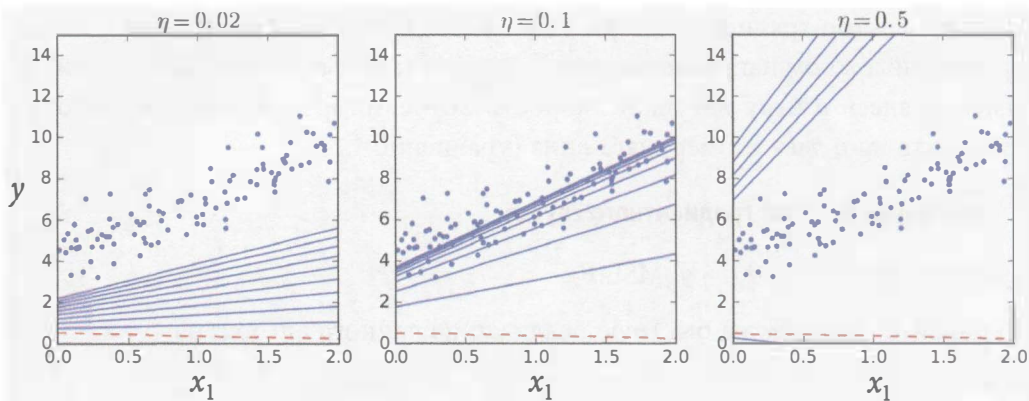


Рис. 4.8. Градиентный спуск с различными скоростями обучения

Тем не менее, количество итераций можно ограничить, если желательно, чтобы решетчатый поиск был в состоянии исключить модели, которые требуют слишком длительного времени на сходжение.

Вас может интересовать, каким образом устанавливать количество итераций. Если оно слишком мало, тогда вы по-прежнему окажетесь далеко от оптимального решения, когда алгоритм остановится, но если количество итераций очень велико, то вы будете понапрасну растрачивать время наряду с тем, что параметры модели больше не изменятся. Простое решение предусматривает установку очень большого числа итераций, но прекращение работы алгоритма, как только вектор-градиент становится маленьким, т.е. норма делается меньше, чем крошечное число ε (называемое **допуском** (*tolerance*)), поскольку такое случается, когда градиентный спуск (почти) достиг минимума.

Скорость сходимости

Когда функция издержек выпуклая и ее наклон резко не изменяется (как в случае функции издержек MSE), то пакетный градиентный спуск с фиксированной скоростью обучения в итоге сойдется к оптимальному решению, но возможно вам придется немного подождать. В зависимости от формы функции издержек он может требовать $O(1/\varepsilon)$ итераций для достижения оптимума внутри диапазона ε . Если вы разделите допуск на 10, чтобы иметь более точное решение, тогда алгоритму придется выполняться примерно в 10 раз дольше.

Стохастический градиентный спуск

Главная проблема с пакетным градиентным спуском — тот факт, что он использует полный обучающий набор для вычисления градиентов на каждом шаге, который делает его очень медленным в случае крупного обучающего набора. Как противоположная крайность, *стохастический градиентный спуск* (*Stochastic Gradient Descent* — *SGD*) на каждом шаге просто выбирает из обучающего набора случайный образец и вычисляет градиенты на основе только этого единственного образца. Очевидно, алгоритм становится гораздо быстрее, т.к. на каждой операции ему приходится манипулировать совсем малым объемом данных. Также появляется возможность проводить обучение на гигантских обучающих наборах, потому что на каждой итерации в памяти должен находиться только один образец (SGD может быть реализован как алгоритм внешнего обучения⁷).

С другой стороны, из-за своей стохастической (т.е. случайной) природы этот алгоритм гораздо менее нормален, чем пакетный градиентный спуск: вместо умеренного понижения вплоть до достижения минимума функция издержек будет скачками изменяться вверх и вниз, понижаясь только в среднем. Со временем алгоритм будет очень близок к минимуму, но как только он туда доберется, скачкообразные изменения продолжатся, никогда не уходясь (рис. 4.9). Таким образом, после окончания алгоритма финальные значения параметров оказываются хорошими, но не оптимальными.

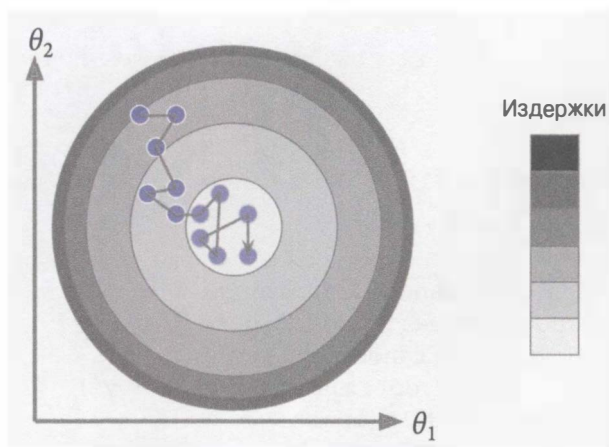


Рис. 4.9. Стохастический градиентный спуск

⁷ Внешнее обучение обсуждалось в главе 1.

Когда функция издержек крайне нерегулярна (как на рис. 4.6), то это в действительности может помочь алгоритму выбраться из локальных минимумов, так что у стохастического градиентного спуска есть больше шансов отыскать глобальный минимум, чем у пакетного градиентного спуска.

Следовательно, случайность хороша для избегания локальных оптимумов, но плоха потому, что означает, что алгоритм может никогда не осесть на минимуме. Одним из решений такой дилеммы является постепенное сокращение скорости обучения. Шаги начинаются с больших (которые содействуют в достижении быстрого прогресса и избегании локальных минимумов), а затем становятся все меньше и меньше, позволяя алгоритму обосноваться на глобальном минимуме. Такой процесс называется *имитацией отжига* (*simulated annealing*), поскольку он имеет сходство с процессом закалки в металлургии, когда расплавленный металл медленно охлаждается. Функция, которая определяет скорость обучения на каждой итерации, называется *графиком обучения* (*learning schedule*). Если скорость обучения снижается слишком быстро, то вы можете застрять в локальном минимуме или даже остаться на полпути к минимуму. Если скорость обучения снижается чересчур медленно, тогда вы можете долго прыгать возле минимума и в конечном итоге получить квазиоптимальное решение в случае прекращения обучения слишком рано.

Приведенный ниже код реализует стохастический градиентный спуск с применением простого графика обучения:

```
n_epochs = 50
t0, t1 = 5, 20                                # гиперпараметры графика обучения

def learning_schedule(t):
    return t0 / (t + t1)

theta = np.random.randn(2,1)                  # случайная инициализация

for epoch in range(n_epochs):
    for i in range(m):
        random_index = np.random.randint(m)
        xi = X_b[random_index:random_index+1]
        yi = y[random_index:random_index+1]
        gradients = 2 * xi.T.dot(xi.dot(theta) - yi)
        eta = learning_schedule(epoch * m + i)
        theta = theta - eta * gradients
```

По соглашению мы повторяем раундами по *m* итераций; каждый раунд называется *эпохой* (*epoch*). Наряду с тем, что код пакетного градиентного

спуска повторяется 1000 раз на всем обучающем наборе, показанный выше код проходит через обучающий набор только 50 раз и добирается до приемлемого решения:

```
>>> theta
array([[ 4.21076011],
       [ 2.74856079]])
```

На рис. 4.10 представлены первые 10 шагов обучения (заметьте, насколько нерегулярны шаги).

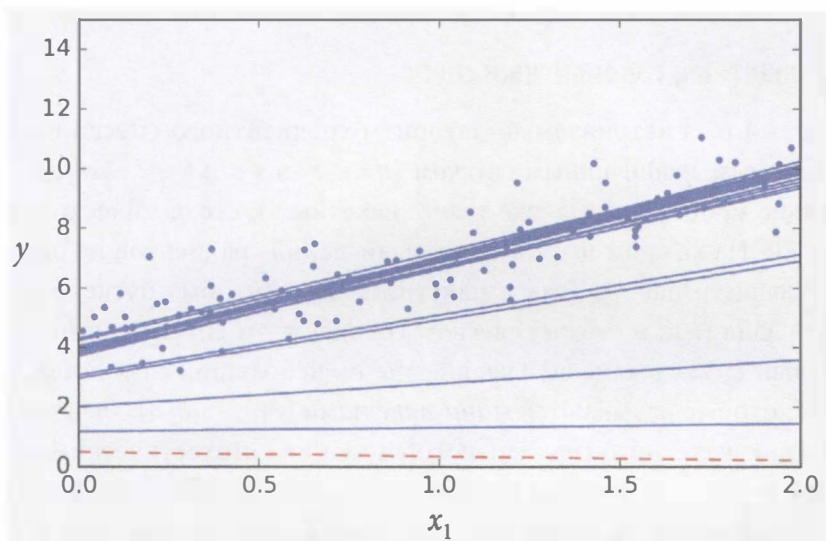


Рис. 4.10. Первые 10 шагов стохастического градиентного спуска

Обратите внимание, что поскольку образцы выбираются случайно, некоторые из них могут быть выбраны несколько раз за эпоху, тогда как другие — вообще не выбираться. Если вы хотите обеспечить, чтобы алгоритм в рамках каждой эпохи проходил по каждому образцу, то другой подход предусматривает тасование обучающего набора, проход по нему образцом за образцом, снова тасование и т.д. Однако при таком подходе сходжение обычно происходит медленнее.

Чтобы выполнить линейную регрессию, использующую SGD, с помощью Scikit-Learn, вы можете применять класс `SGDRegressor`, который по умолчанию настроен на оптимизацию функции издержек в виде квадратичной ошибки. Следующий код прогоняет 50 эпох, начиная со скорости обучения 0.1 (`eta0=0.1`) и используя стандартный график обучения

(отличающийся от предыдущего), но не применяет какой-либо регуляризации (`penalty=None`; вскоре мы поговорим об этом более подробно):

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(n_iter=50, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

И снова вы находите решение, очень близкое к тому, что возвращалось нормальным уравнением:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([ 4.16782089]), array([ 2.72603052]))
```

Мини-пакетный градиентный спуск

Последний рассматриваемый алгоритм градиентного спуска называется *мини-пакетным градиентным спуском* (*mini-batch gradient descent*). Понять его довольно просто, т.к. вы уже знаете пакетный и стохастический градиентные спуски. На каждом шаге вместо вычисления градиентов на основе полного обучающего набора (как в пакетном градиентном спуске) или только одного образца (как в стохастическом градиентном спуске) мини-пакетный градиентный спуск вычисляет градиенты на небольших случайных наборах образцов, которые называются *мини-пакетами* (*mini-batch*). Главное превосходство мини-пакетного градиентного спуска над стохастическим градиентным спуском в том, что вы можете получить подъем производительности от аппаратной оптимизации матричных операций, особенно когда используются графические процессоры.

Продвижение этого алгоритма в пространстве параметров не настолько нерегулярно, как в случае SGD, в особенности при довольно крупных мини-пакетах. В результате мини-пакетный градиентный спуск закончит блуждания чуть ближе к минимуму, чем SGD. Но с другой стороны ему может быть труднее уйти от локальных минимумов (в случае задач, которые страдают от локальных минимумов, в отличие от линейной регрессии, как мы видели ранее). На рис. 4.11 показаны пути, проходимые тремя алгоритмами градиентного спуска в пространстве параметров во время обучения. В итоге все они оказываются близко к минимуму, но путь пакетного градиентного спуска фактически останавливается на минимуме, тогда как стохастический и мини-пакетный градиентные спуски продолжают двигаться около минимума. Тем не менее, не забывайте о том, что пакетный градиентный спуск требует длительного времени при выполнении каждого шага, а стохастический и ми-

ни-пакетный градиентные спуски также смогут достичь минимума, если вы примените хороший график обучения.

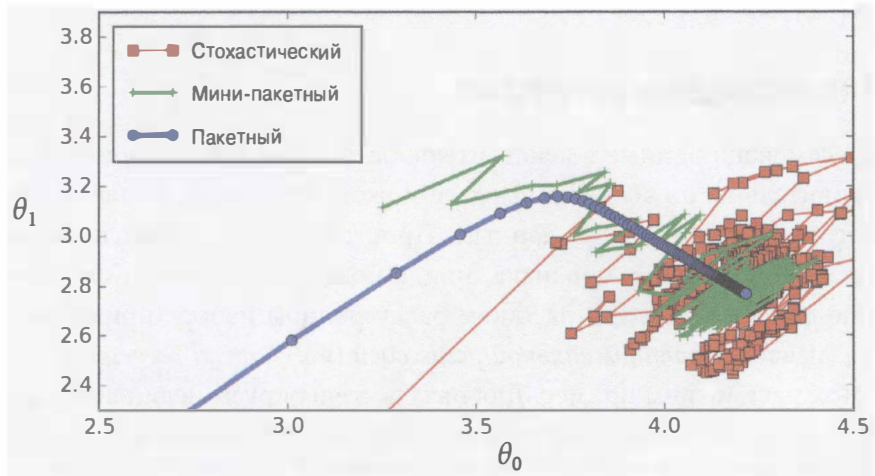


Рис. 4.11. Пути алгоритмов градиентного спуска в пространстве параметров

Давайте сравним алгоритмы, которые мы обсуждали до сих пор, для линейной регрессии⁸ (вспомните, что m — количество обучающих образцов, а n — количество признаков); взгляните на табл. 4.1.

Таблица 4.1. Сравнение алгоритмов для линейной регрессии

Алгоритм	Большое m	Поддерживает ли внешнее обучение	Большое n	Гиперпараметры	Требуется ли масштабирование	Scikit-Learn
Нормальное уравнение	Быстрый	Нет	Медленный	0	Нет	Linear Regression
Пакетный градиентный спуск	Медленный	Нет	Быстрый	2	Да	—
Стохастический градиентный спуск	Быстрый	Да	Быстрый	≥ 2	Да	SGDRegressor
Мини-пакетный градиентный спуск	Быстрый	Да	Быстрый	≥ 2	Да	SGDRegressor

⁸ В то время как нормальное уравнение может выполнять только линейную регрессию, вы увидите, что алгоритмы градиентного спуска могут использоваться для обучения многих других моделей.



После обучения практически нет никаких отличий: все алгоритмы заканчивают очень похожими моделями и вырабатывают прогнозы точно таким же образом.

Полиномиальная регрессия

Что, если ваши данные в действительности сложнее обычной прямой линии? Удивительно, но вы на самом деле можете применять линейную модель для подгонки к нелинейным данным. Простой способ предполагает добавление степеней каждого признака в виде новых признаков и последующее обучение линейной модели на таком расширенном наборе признаков. Этот прием называется *полиномиальной регрессией* (*polynomial regression*).

Давайте рассмотрим пример. Для начала сгенерируем нелинейные данные, основываясь на простом *квадратном уравнении*⁹ (плюс некоторый шум, как показано на рис. 4.12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

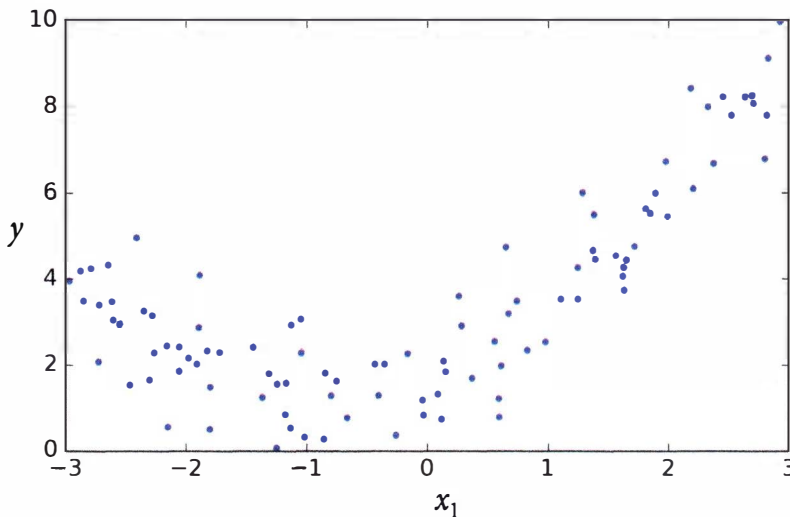


Рис. 4.12. Сгенерированный нелинейный и зашумленный набор данных

⁹ Квадратное уравнение имеет форму $y = ax^2 + bx + c$.

Безусловно, прямая линия никогда не будет подогнана под такие данные должным образом. Потому воспользуемся классом `PolynomialFeatures` из `Scikit-Learn`, чтобы преобразовать наши обучающие данные, добавив в качестве новых признаков квадрат (полином 2-й степени) каждого признака (в этом случае есть только один признак):

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)
>>> X_poly = poly_features.fit_transform(X)
>>> X[0]
array([-0.75275929])
>>> X_poly[0]
array([-0.75275929,  0.56664654])
```

Теперь `X_poly` содержит первоначальный признак `X` плюс его квадрат. Далее вы можете подогнать модель `LinearRegression` к таким расширенным обучающим данным (рис. 4.13):

```
>>> lin_reg = LinearRegression()
>>> lin_reg.fit(X_poly, y)
>>> lin_reg.intercept_, lin_reg.coef_
(array([ 1.78134581]), array([[ 0.93366893,  0.56456263]]))
```

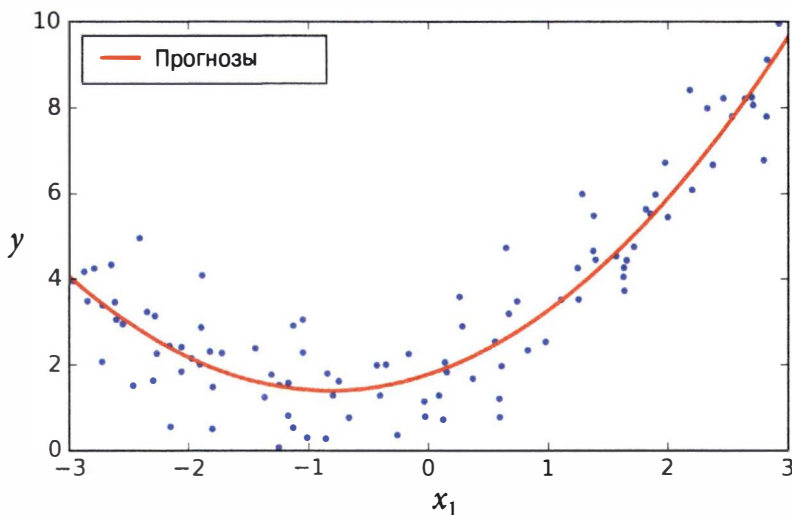


Рис. 4.13. Прогнозы полиномиальной регрессионной модели

Неплохо: модель оценивает функцию как $y = 0.56x_1^2 + 0.93x_1 + 1.78$, когда на самом деле исходной функцией была $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{гауссов шум}$.

Обратите внимание, что при наличии множества признаков полиномиальная регрессия способна отыскать связи между признаками (то, что простая линейная регрессионная модель делать не в состоянии). Это становится возможным благодаря тому факту, что класс `PolynomialFeatures` также добавляет все комбинации признаков вплоть до заданной степени. Например, если есть два признака a и b , тогда `PolynomialFeatures` с `degree=3` добавил бы не только признаки a^2 , a^3 , b^2 и b^3 , но и комбинации ab , a^2b и ab^2 .



`PolynomialFeatures(degree=d)` трансформирует массив, содержащий n признаков, в массив, содержащий $\frac{(n+d)!}{d!n!}$ признаков, где $n!$ — *факториал* числа n , который равен $1 \times 2 \times 3 \times \dots \times n$. Остерегайтесь комбинаторного бурного роста количества признаков!

Кривые обучения

В случае выполнения полиномиальной регрессии высокой степени, вероятно, вы гораздо лучше подгоните обучающие данные, чем с помощью линейной регрессии. Например, на рис. 4.14 демонстрируется применение полиномиальной модели 300-й степени к предшествующим обучающим данным, а результат сравнивается с чистой линейной моделью и квадратичной моделью (полиномиальной 2-й степени).

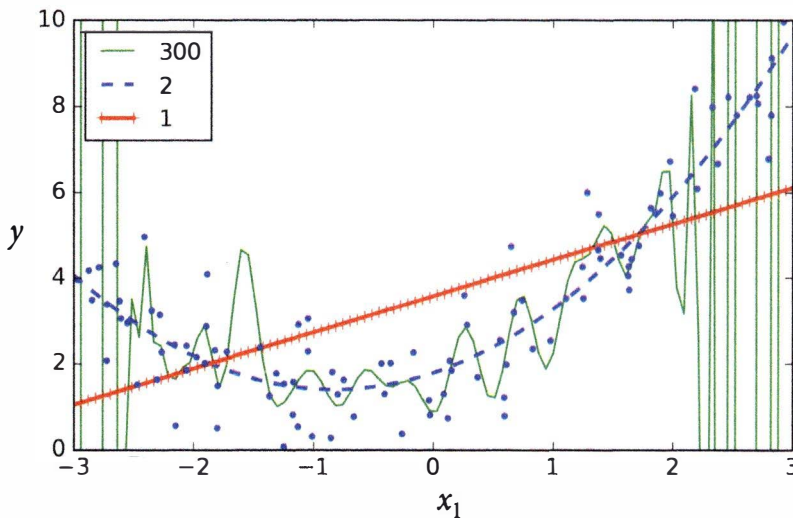


Рис. 4.14. Полиномиальная регрессия высокой степени

Обратите внимание на то, как полиномиальная модель 300-й степени колеблется, чтобы как можно больше приблизиться к обучающим образцам.

Разумеется, такая полиномиальная регрессионная модель высокой степени вызывает сильное переобучение обучающими данными, тогда как линейная модель — недообучение на них. В рассматриваемом случае будет хорошо обобщаться квадратичная модель. Это имеет смысл, поскольку данные генерировались с использованием квадратного уравнения, но с учетом того, что обычно вы не будете знать функцию, применяемую для генерации данных, каким же образом принимать решение о том, насколько сложной должна быть модель? Как выяснить, что модель переобучается или недообучается на данных?

В главе 2 с помощью перекрестной проверки оценивалась производительность обобщения модели. Если согласно метрикам перекрестной проверки модель хорошо выполняется на обучающих данных, но плохо обобщается, то модель переобучена. Если модель плохо выполняется в обоих случаях, тогда она недообучена. Так выглядит один из способов сказать, что модель слишком проста или чрезмерно сложна.

Другой способ предусматривает просмотр *кривых обучения* (*learning curve*): они представляют собой графики производительности модели на обучающем наборе и проверочном наборе как функции от размера обучающего набора (или итерации обучения). Чтобы получить такие графики, нужно просто обучить модель несколько раз на подмножествах разных размеров, взятых из обучающего набора. В следующем коде определяется функция, которая вычерчивает кривые обучения модели для установленных обучающих данных:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val =
        train_test_split(X, y, test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict,
                                                y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict, y_val))
    plt.plot(np.sqrt(train_errors), "r--", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Давайте взглянем на кривые обучения обыкновенной линейной регрессионной модели (прямая линия на рис. 4.15):

```
lin_reg = LinearRegression()  
plot_learning_curves(lin_reg, X, y)
```

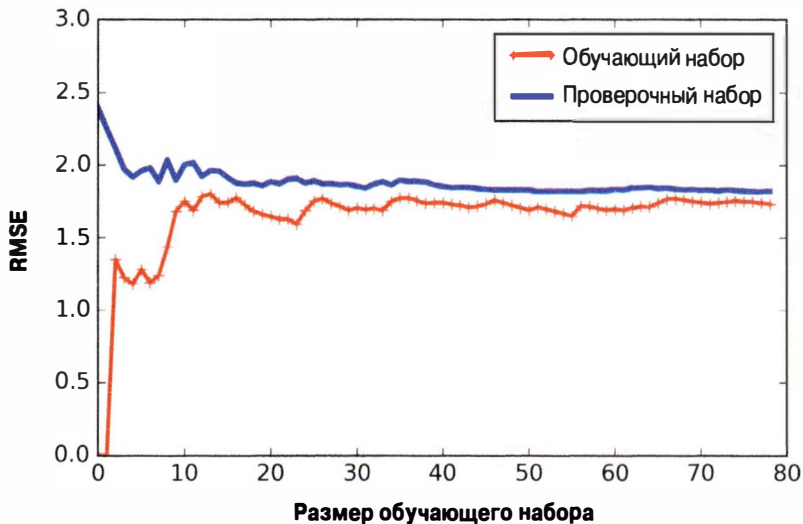


Рис. 4.15. Кривые обучения для линейной модели

Здесь необходимы некоторые пояснения. Первым делом обратите внимание на производительность модели в случае использования обучающих данных: когда в обучающем наборе есть только один или два образца, модель может быть в полной мере подогнана к ним, что и объясняет начало кривой с нулевой ошибки. Но по мере добавления образцов в обучающий набор идеальная подгонка модели к обучающим данным становится невозможной, как из-за того, что данные зашумлены, так и потому, что они совершенно отличаются от линейных. Следовательно, ошибка на обучающих данных движется вверх, пока не стабилизируется, когда добавление новых образцов в обучающий набор не делает среднюю ошибку намного лучше или хуже. Теперь перейдем к производительности модели на проверочных данных. Когда модель обучалась на незначительном количестве обучающих образцов, она неспособна обобщаться надлежащим образом, а потому ошибка проверки изначально довольно велика. Затем по мере того, как модель видит все больше обучающих образцов, она обучается, а ошибка проверки соответственно медленно снижается. Однако прямая линия снова не в состоянии хорошо смоделировать данные, так что ошибка стабилизируется поблизости к другой кривой.

Такие кривые обучения типичны для недообученной модели. Обе кривые стабилизируются; они расположены близко друг к другу и довольно высоко.



Если ваша модель недообучена на обучающих данных, тогда добавление дополнительных обучающих образцов не поможет. Вам нужно выбрать более сложную модель или отыскать лучшие признаки.

Теперь рассмотрим кривые обучения полиномиальной модели 10-й степени на тех же самых данных (рис. 4.16):

```
from sklearn.pipeline import Pipeline
polynomial_regression = Pipeline([
    ("poly_features",
     PolynomialFeatures(degree=10, include_bias=False)),
    ("lin_reg", LinearRegression()),
])
plot_learning_curves(polynomial_regression, X, y)
```

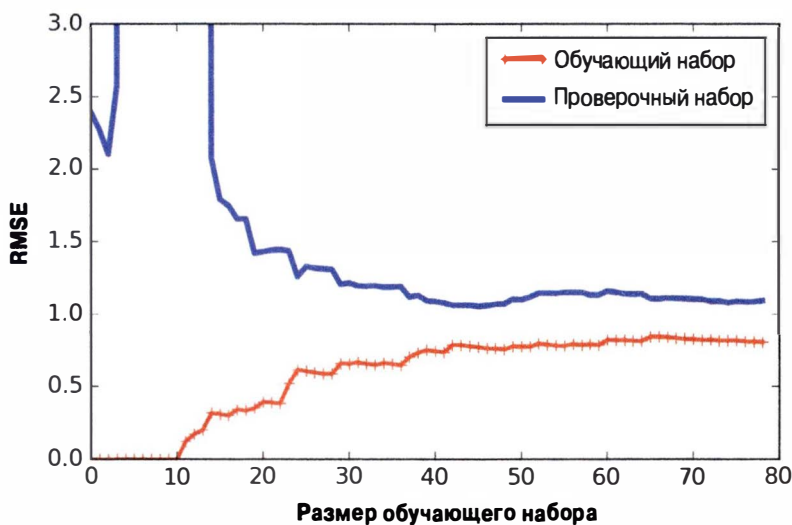


Рис. 4.16. Кривые обучения для полиномиальной модели

Кривые обучения выглядят чуть лучше предыдущих, но есть два очень важных отличия.

- Ошибка на обучающих данных гораздо ниже, чем в случае линейной регрессионной модели.

- Между кривыми имеется промежуток. Это значит, что модель выполняется существенно лучше на обучающих данных, чем на проверочных данных, демонстрируя признак переобучения. Тем не менее, если вы примените намного более крупный обучающий набор, то две кривые продолжат сближение.



Один из способов улучшения переобученной модели заключается в предоставлении ей добавочных обучающих данных до тех пор, пока ошибка проверки не достигнет ошибки обучения.

Компромисс между смещением и дисперсией

Важным теоретическим результатом статистики и машинного обучения является тот факт, что ошибка обобщения модели может быть выражена в виде суммы трех очень разных ошибок.

Смещение. Эта часть ошибки обобщения связана с неверными предположениями, такими как допущение того, что данные линейные, когда они на самом деле квадратичные. Модель с высоким смещением, скорее всего, недообучится на обучаемых данных¹⁰.

Дисперсия. Эта часть объясняется чрезмерной чувствительностью модели к небольшим изменениям в обучающих данных. Модель со многими степенями свободы (такая как полиномиальная модель высокой степени), вероятно, будет иметь высокую дисперсию и потому переобучаться обучающими данными.

Неустраняемая погрешность. Эта часть появляется вследствие зашумленности самих данных. Единственный способ сократить неустраняемую погрешность в ошибке предусматривает очистку данных (например, приведение в порядок источников данных, таких как неисправные датчики, или выявление и устранение выбросов).

Возрастание сложности модели обычно увеличивает ее дисперсию и уменьшает смещение. И наоборот, сокращение сложности модели увеличивает ее смещение и уменьшает дисперсию. Вот почему это называется компромиссом.

¹⁰ Такое понятие смещения не следует путать с понятием члена смещения в линейных моделях.

Регуляризованные линейные модели

Как было показано в главах 1 и 2, хороший способ сократить переобучение заключается в том, чтобы регуляризовать модель (т.е. ограничить ее): чем меньше степеней свободы она имеет, тем труднее ее будет переобучить данными. Например, простой метод регуляризации полиномиальной модели предполагает понижение количества полиномиальных степеней.

Для линейной модели регуляризация обычно достигается путем ограничения весов модели. Мы рассмотрим *гребневую регрессию* (*ridge regression*), *лассо-регрессию* (*lasso regression*) и *эластичную сеть* (*elastic net*), которые реализуют три разных способа ограничения весов.

Гребневая регрессия

Гребневая регрессия (также называемая *регуляризацией Тихонова*) является регуляризированной версией линейной регрессии: к функции издержек добавляется *член регуляризации* (*regularization term*), равный $\alpha \sum_{i=1}^n \theta_i^2$. Это заставляет алгоритм обучения не только приспосабливаться к данным, но также удерживать веса модели насколько возможно небольшими. Обратите внимание, что член регуляризации должен добавляться к функции издержек только во время обучения. После того как модель обучена, вы захотите оценить производительность модели с использованием нерегуляризированной меры производительности.



Отличие функции издержек, применяемой во время обучения, от меры производительности, используемой для проверки — довольно распространенное явление. Еще одна причина, по которой они могут быть разными, не считая регуляризации, заключается в том, что хорошая функция издержек при обучении должна иметь удобные для оптимизации производные, тогда как мера производительности, применяемая для проверки, обязана быть насколько возможно близкой к финальной цели. Подходящим примером может служить классификатор, который обучается с использованием такой функции издержек, как *логарифмическая потеря* (*log loss*; обсуждается далее в главе), но оценивается с применением точности/полноты.

Гиперпараметр α управляет тем, насколько необходимо регуляризовать модель. Когда $\alpha = 0$, гребневая регрессия оказывается просто линейной регрес-

сий. При очень большом значении α все веса становятся крайне близкими к нулю, и результатом будет ровная линия, проходящая через середину данных. В уравнении 4.8 представлена функция издержек гребневой регрессии¹¹.

Уравнение 4.8. Функция издержек для гребневой регрессии

$$J(\theta) = \text{MSE}(\theta) + \alpha \frac{1}{2} \sum_{i=1}^n \theta_i^2$$

Обратите внимание, что член смещения θ_0 не регуляризован (сумма начинается с $i = 1$, не 0). Если мы объявим \mathbf{w} как весовой вектор признаков (от θ_1 до θ_n), тогда член регуляризации просто равен $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, где $\|\cdot\|_2$ представляет норму ℓ_2 весового вектора¹². Для градиентного спуска нужно просто добавить $\alpha \mathbf{w}$ к вектору-градиенту MSE (уравнение 4.6).



Перед выполнением гребневой регрессии важно масштабировать данные (скажем, посредством `StandardScaler`), т.к. она чувствительна к масштабу входных признаков. Это справедливо для большинства регуляризованных моделей.

На рис. 4.17 показано несколько гребневых моделей, обученных на некоторых линейных данных с использованием разных значений α .

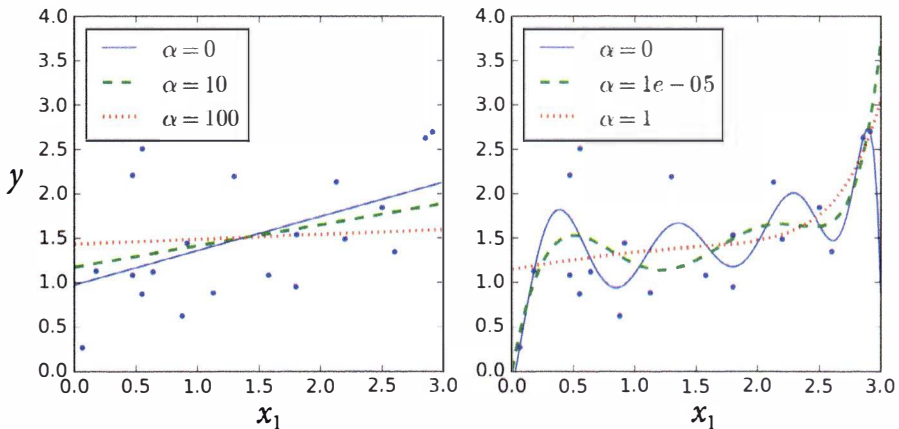


Рис. 4.17. Гребневая регрессия

¹¹ Для функций издержек, не имеющих краткого имени, принято использовать обозначение $J(\theta)$; мы будем часто применять такое обозначение в оставшихся главах книги. Контекст прояснит, какая функция издержек обсуждается.

¹² Нормы обсуждались в главе 2.

Слева применялись обыкновенные гребневые модели, приводя к линейным прогнозам. Справа данные были сначала расширены с применением `PolynomialFeatures(degree=10)`, затем масштабированы с использованием `StandardScaler` и в заключение к результирующим признакам были применены гребневые модели: это полиномиальная регрессия с гребневой регуляризацией. Обратите внимание на то, как увеличение α ведет к более ровным (т.е. менее предельным, более рациональным) прогнозам; в итоге повышается дисперсия модели, но снижается ее смещение.

Как и в случае линейной регрессии, мы можем производить гребневую регрессию, либо вычисляя уравнение *в аналитическом виде*, либо выполняя градиентный спуск. Доводы за и против одинаковы. В уравнении 4.9 *решение в аналитическом виде* (где A — это *единичная матрица*¹³ (*identity matrix*) $n \times n$ за исключением нуля в левой верхней ячейке, соответствующего члену смещения).

Уравнение 4.9. Решение в аналитическом виде для гребневой регрессии

$$\hat{\theta} = (X^T \cdot X + \alpha A)^{-1} \cdot X^T \cdot y$$

Вот как выполнять гребневую регрессию с помощью Scikit-Learn, используя решение в *аналитическом виде* (вариант уравнения 4.9 применяет метод разложения матрицы Андре-Луи Холецкого):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[ 1.55071465]])
```

И с применением стохастического градиентного спуска¹⁴:

¹³ Квадратная матрица, полная нулей, за исключением единиц на главной диагонали (от левого верхнего до правого нижнего угла).

¹⁴ В качестве альтернативы можно применять класс `Ridge` с `solver="sag"`. *Стохастический усредненный градиентный спуск* (*stochastic average GD*) представляет собой разновидность SGD. За дополнительными деталями обращайтесь к презентации “Minimizing Finite Sums with the Stochastic Average Gradient Algorithm” (“Сведение к минимуму конечных сумм с помощью алгоритма стохастического усредненного градиентного спуска”) Марка Шмидта и др. из Университета Британской Колумбии (<http://goo.gl/vxVyA2>).

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([ 1.13500145])
```

Гиперпараметр `penalty` устанавливает используемый тип члена регуляризации. Указание `"l2"` обозначает то, что вы хотите, чтобы алгоритм SGD добавлял к функции издержек член регуляризации, равный одной второй квадрата нормы ℓ_2 весового вектора: это просто гребневая регрессия.

Лассо-регрессия

Регрессия методом наименьшего абсолютного сокращения и выбора (*least absolute shrinkage and selection operator (lasso) regression*), называемая просто *лассо-регрессией*, представляет собой еще одну регуляризованную версию линейной регрессии: в точности как гребневая регрессия она добавляет к функции издержек член регуляризации, но вместо одной второй квадрата нормы ℓ_2 весового вектора использует норму ℓ_1 весового вектора (уравнение 4.10).

Уравнение 4.10. Функция издержек для лассо-регрессии

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^n |\theta_i|$$

На рис. 4.18 демонстрируется то же самое, что и на рис. 4.17, но гребневые модели заменены лассо-моделями и применяются меньшие значения α .

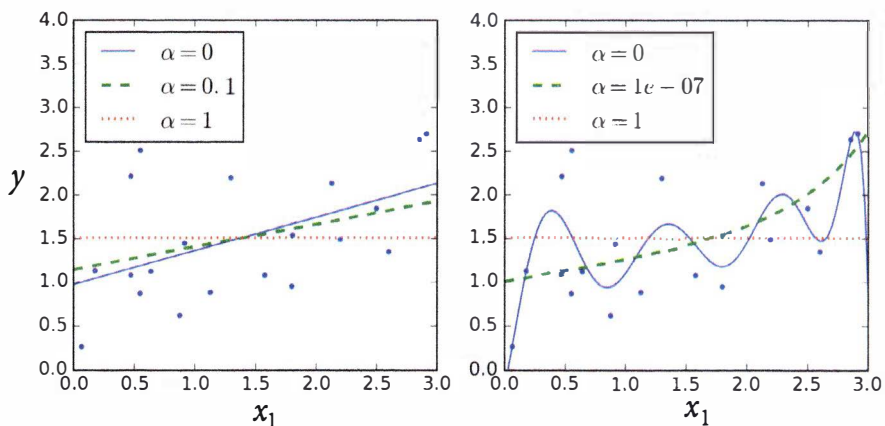


Рис. 4.18. Лассо-регрессия

Важной характеристикой лассо-регрессии является то, что она стремится полностью исключить веса наименее важных признаков (т.е. устанавливает их в ноль). Например, пунктирная линия на графике справа на рис. 4.18 (с $\alpha = 10^{-7}$) выглядит квадратичной, почти линейной: все веса для полиномиальных признаков высокой степени равны нулю. Другими словами, лассо-регрессия автоматически выполняет выбор признаков и выпускает *разреженную модель* (т.е. с незначительным числом ненулевых весов признаков).

Осознать, почему это так, можно с помощью рис. 4.19: на левом верхнем графике фоновые контуры (эллипсы) представляют нерегуляризованную функцию издержек MSE ($\alpha = 0$), а белые кружочки показывают путь пакетного градиентного спуска (BGD) с этой функцией издержек. Контуры переднего плана (ромбы) представляют штраф ℓ_1 , а треугольники показывают путь BGD только для данного штрафа ($\alpha \rightarrow \infty$). Обратите внимание на то, как путь сначала достигает $\theta_1 = 0$, после чего катится вниз по желобу до тех пор, пока не добирается до $\theta_2 = 0$. На правом верхнем графике контуры представляют ту же самую функцию издержек плюс штраф ℓ_1 с $\alpha = 0.5$. Глобальный минимум находится на оси $\theta_2 = 0$. Путь BGD сначала достигает $\theta_2 = 0$ и затем катится по желобу, пока не доходит до глобального минимума.

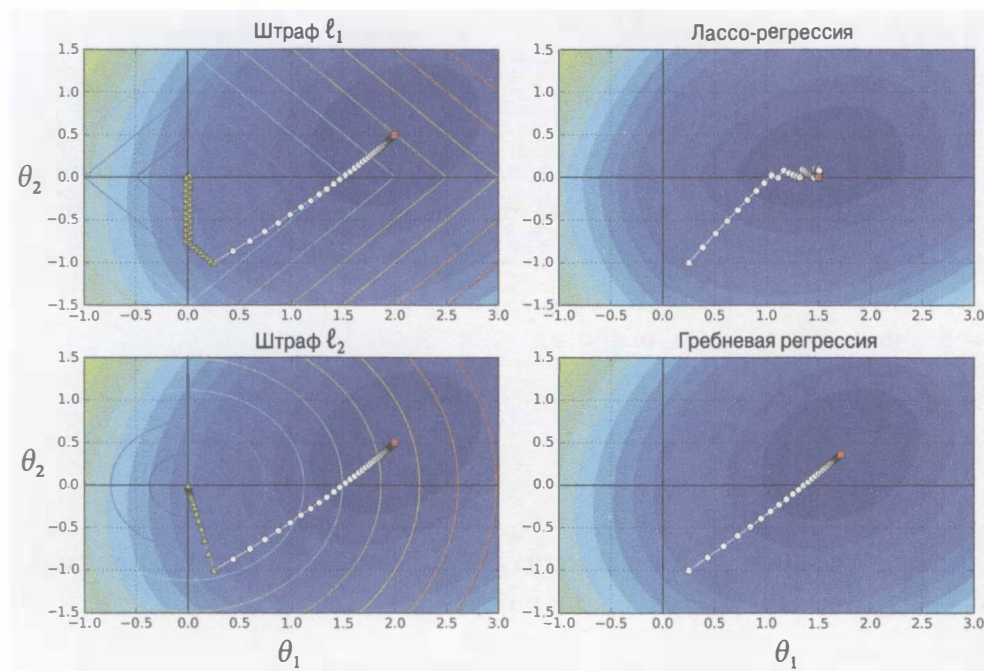


Рис. 4.19. Сравнение лассо-регрессии и гребневой регрессии

Два нижних графика демонстрируют те же вещи, но взамен используют штраф ℓ_2 . Регуляризованный минимум ближе к $\theta = 0$, чем нерегуляризованный минимум, но веса не полностью устранены.



С функцией издержек лассо-регрессии путь BGD имеет тенденцию колебаться поперек желоба ближе к концу. Причина в том, что в точке $\theta_2 = 0$ наклон внезапно изменяется. Чтобы действительно сойтись в глобальном минимуме, вам понадобится постепенно снижать скорость обучения.

Функция издержек лассо-регрессии не является дифференцируемой при $\theta_i = 0$ (для $i = 1, 2, \dots, n$), но градиентный спуск по-прежнему хорошо работает, если вы взамен применяете **вектор-субградиент** (*subgradient vector*) \mathbf{g}^{15} , когда $\theta_i = 0$. В уравнении 4.11 показано уравнение вектора-субградиента, которое можно использовать для градиентного спуска с функцией издержек лассо-регрессии.

Уравнение 4.11. Вектор-субградиент лассо-регрессии

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix}, \quad \text{где} \quad \text{sign}(\theta_i) = \begin{cases} -1, & \text{если } \theta_i < 0 \\ 0, & \text{если } \theta_i = 0 \\ +1, & \text{если } \theta_i > 0 \end{cases}$$

Ниже приведен небольшой пример Scikit-Learn, в котором применяется класс `Lasso`. Обратите внимание, что взамен вы могли бы использовать `SGDRegressor(penalty="l1")`.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

Эластичная сеть

Эластичная сеть — это середина между гребневой регрессией и лассо-регрессией. Член регуляризации представляет собой просто смесь чле-

¹⁵ Вы можете представлять вектор-субградиент в недифференцируемой точке как промежуточный вектор между векторами-градиентами вокруг этой точки.

нов регуляризации гребневой регрессии и лассо-регрессии, к тому же можно управлять отношением смеси r . При $r = 0$ эластичная сеть эквивалентна гребневой регрессии, а при $r = 1$ она эквивалентна лассо-регрессии (уравнение 4.12).

Уравнение 4.12. Функция издержек эластичной сети

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^n |\theta_i| + \frac{1-r}{2} \alpha \sum_{i=1}^n \theta_i^2$$

Итак, когда вы должны применять обыкновенную линейную регрессию (т.е. без какой-либо регуляризации), гребневую регрессию, лассо-регрессию или эластичную сеть? Почти всегда предпочтительнее иметь хотя бы немного регуляризации, поэтому в целом вам следует избегать использования обыкновенной линейной регрессии. Гребневая регрессия — хороший вариант по умолчанию, но если вы полагаете, что только несколько признаков будут фактически полезными, то должны отдавать предпочтение лассо-регрессии или эластичной сети, поскольку, как уже обсуждалось, они имеют тенденцию понижать веса бесполезных признаков до нуля. В общем случае эластичная сеть предпочтительнее лассо-регрессии, т.к. лассо-регрессия может работать с переборами, когда количество признаков больше числа обучающих образцов или некоторые признаки сильно связаны.

Вот короткий пример, в котором применяется класс `ElasticNet` из Scikit-Learn (`l1_ratio` соответствует отношению смеси r):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

Раннее прекращение

Совершенно другой способ регуляризации итеративных алгоритмов обучения, подобных градиентному спуску, предусматривает остановку обучения, как только ошибка проверки достигает минимума. Такой прием называется *ранним прекращением* (*early stopping*). На рис. 4.20 показана сложная модель (в данном случае полиномиальная регрессионная модель высокой степени) во время обучения с использованием пакетного градиентного спуска. По мере прохождения эпох алгоритм обучается и его ошибка прогноза (RMSE)

на обучающем наборе естественным образом снижается и то же самое делает его ошибка прогноза на проверочном наборе. Однако через некоторое время ошибка проверки перестает уменьшаться и фактически возвращается к росту. Такое положение дел указывает на то, что модель начала переобучаться обучающими данными. С помощью раннего прекращения вы просто останавливаете обучение, как только ошибка проверки достигла минимума. Это настолько простой и эффективный прием регуляризации, что Джеффри Хинтон назвал его “превосходным бесплатным завтраком”.

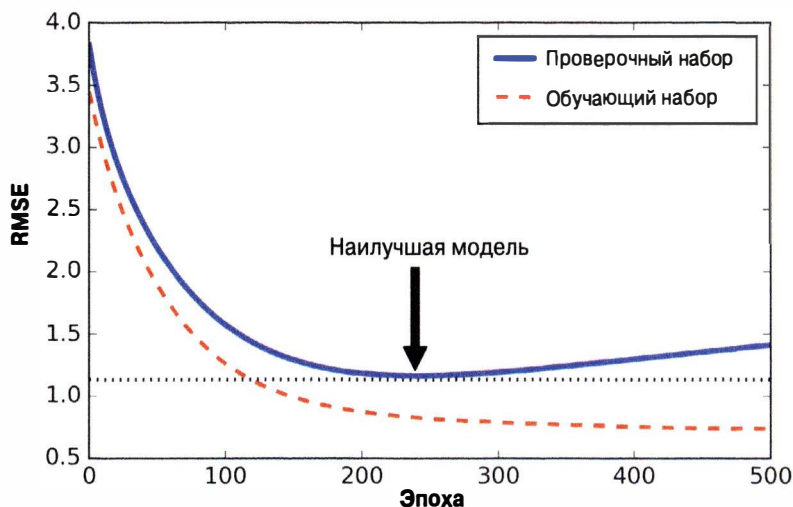


Рис. 4.20. Регуляризация с ранним прекращением



При стохастическом и мини-пакетном градиентном спуске кривые не до такой степени гладкие, а потому узнать, достигнут минимум или нет, может быть трудно. Решение заключается в том, чтобы остановить обучение только после того, как ошибка проверки находилась над минимумом в течение некоторого времени (когда вы уверены, что модель больше не будет улучшаться), и затем откатить параметры модели в точку, где ошибка проверки была на минимуме.

Ниже приведена базовая реализация раннего прекращения:

```
from sklearn.base import clone
# ПОДГОТОВИТЬ ДАННЫЕ
poly_scaler = Pipeline([
```

```

    ("poly_features",
     PolynomialFeatures(degree=90, include_bias=False)),
    ("std_scaler", StandardScaler()) ])
X_train_poly_scaled = poly_scaler.fit_transform(X_train)
X_val_poly_scaled = poly_scaler.transform(X_val)

sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train) # продолжается с места,
                                             # которое было оставлено
    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)

```

Обратите внимание, что при `warm_start=True`, когда вызывается метод `fit()`, он просто продолжает обучение с места, которое он оставил, а не перезапускается с самого начала.

Логистическая регрессия

Как обсуждалось в главе 1, некоторые алгоритмы регрессии могут применяться также для классификации (и наоборот). *Логистическая регрессия* (также называемая *логит-регрессией* (*logit regression*)) обычно используется для оценки вероятности того, что образец принадлежит к определенному классу (например, какова вероятность того, что заданное почтовое сообщение является спамом?). Если оценочная вероятность больше 50%, тогда модель прогнозирует, что образец принадлежит к данному классу (называемому положительным классом, помеченным “1”), а иначе — что не принадлежит (т.е. относится к отрицательному классу, помеченному “0”). Это делает ее двоичным классификатором.

Оценивание вероятностей

Итак, каким образом все работает? Подобно линейной регрессионной модели логистическая регрессионная модель подсчитывает взвешенные сум-

мы входных признаков (плюс член смещения), но взамен выдачи результата напрямую, как делает линейная регрессионная модель, он выдает *логистику* (*logistic*) результата (уравнение 4.13).

Уравнение 4.13. Логистическая регрессионная модель оценивает вероятность (векторизованная форма)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x})$$

Логистика, также называемая *логитом* (*logit*) и обозначаемая $\sigma(\cdot)$, представляет собой *сигмоидальную* (т.е. S-образной формы) функцию, которая выдает число между 0 и 1. Она определена, как показано в уравнении 4.14 и на рис. 4.21.

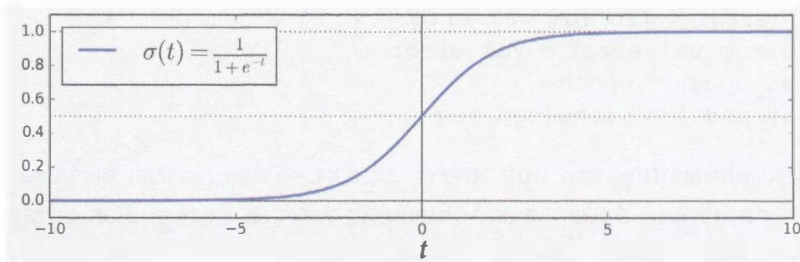


Рис. 4.21. Логистическая функция

Уравнение 4.14. Логистическая функция

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

После того как логистическая регрессионная модель оценила вероятность $\hat{p} = h_{\theta}(\mathbf{x})$, что образец \mathbf{x} принадлежит положительному классу, она может легко выработать прогноз \hat{y} (уравнение 4.15).

Уравнение 4.15. Прогноз логистической регрессионной модели

$$\hat{y} = \begin{cases} 0, & \text{если } \hat{p} < 0.5, \\ 1, & \text{если } \hat{p} \geq 0.5. \end{cases}$$

Обратите внимание, что $\sigma(t) < 0.5$, когда $t < 0$, а $\sigma(t) \geq 0.5$, когда $t \geq 0$, так что логистическая регрессионная модель прогнозирует 1, если $\theta^T \cdot \mathbf{x}$ положительно, и 0, если отрицательно.

Обучение и функция издержек

Хорошо, теперь вы знаете, каким образом логистическая регрессионная модель оценивает вероятности и вырабатывает прогнозы. Но как ее обучить? Целью обучения является установка вектора параметров θ так, чтобы модель выдавала оценки в виде высокой вероятности для положительных образцов ($y = 1$) и низкой вероятности для отрицательных образцов ($y = 0$). Указанная идея воплощена в функции издержек, приведенной в уравнении 4.16, для одиночного обучающего образца x .

Уравнение 4.16. Функция издержек одиночного обучающего образца

$$c(\theta) = \begin{cases} -\log(\hat{p}) & , \text{ если } y = 1, \\ -\log(1 - \hat{p}) & , \text{ если } y = 0. \end{cases}$$

Такая функция издержек имеет смысл, потому что $-\log(t)$ растет очень медленно, когда t приближается к 0, поэтому издержки будут большими, если модель оценивает вероятность близко к 0 для положительного образца, и они также будут сверхбольшими, если модель оценивает вероятность близко к 1 для отрицательного образца. С другой стороны, $-\log(t)$ близко к 0, когда t близко к 1, а потому издержки будут близки к 0, если оценочная вероятность близка к 0 для отрицательного образца или близка к 1 для положительного образца, что в точности является тем, чего мы хотим.

Функция издержек на полном обучающем наборе представляет собой просто средние издержки на всех обучающих образцах. Она также может быть записана в виде одного выражения (в чем вы можете легко удостовериться), называемого *логарифмической потерей* (*log loss*), как показано в уравнении 4.17.

Уравнение 4.17. Функция издержек логистической регрессии (логарифмическая потеря)

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m \left[y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)}) \right]$$

Плохая новость в том, что нет известных уравнений *в аналитическом виде для вычисления значения θ* , которое сводит к минимуму такую функцию издержек (эквивалента нормального уравнения не существует). Но хорошая новость — функция издержек выпуклая, поэтому градиентный спуск (или любой другой алгоритм оптимизации) гарантированно отыщет глобальный

минимум (если скорость обучения не слишком высока и вы подождете достаточно долго). Частные производные функции издержек относительно j -того параметра модели θ_j имеют вид, представленный в уравнении 4.18.

Уравнение 4.18. Частные производные функции издержек логистической регрессии

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)}$$

Уравнение 4.18 выглядит очень похожим на уравнение 4.5: для каждого образца оно подсчитывает ошибку прогноза и умножает ее на значение j -того признака, после чего вычисляет среднее по всем обучающим образцам. Имея вектор-градиент, содержащий все частные производные, вы можете применять его в алгоритме пакетного градиентного спуска. Итак, теперь вы знаете, как обучать логистическую регрессионную модель. Для стохастического градиентного спуска вы, конечно же, просто брали бы по одному образцу за раз, а для мини-пакетного градиентного спуска использовали бы мини-пакет за раз.

Границы решений

Чтобы продемонстрировать работу логистической регрессии, мы применим набор данных об ирисах. Это знаменитый набор данных, который содержит длины и ширины чашелистиков и лепестков цветков ириса трех разных видов: ирис щетинистый (*iris setosa*), ирис разноцветный (*iris versicolor*) и ирис виргинский (*iris virginica*) (рис. 4.22).

Попробуем построить классификатор для выявления вида ириса виргинского на основе только признака ширины лепестка (*petal width*). Сначала загрузим данные:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:]          # ширина лепестка
>>> y = (iris["target"] == 2).astype(np.int) # 1, если ирис
                                           # виргинский, иначе 0
```

Теперь обучим логистическую регрессионную модель:

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```



Рис. 4.22. Цветы ирисов трех видов¹⁶

Давайте взглянем на оценочные вероятности модели для цветов, ширина лепестка которых варьируется от 0 до 3 сантиметров (рис. 4.23):

```
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-", label="Ирис виргинский")
plt.plot(X_new, y_proba[:, 0], "b--", label="Не ирис виргинский")
# + дополнительный код Matplotlib для улучшения изображения
```

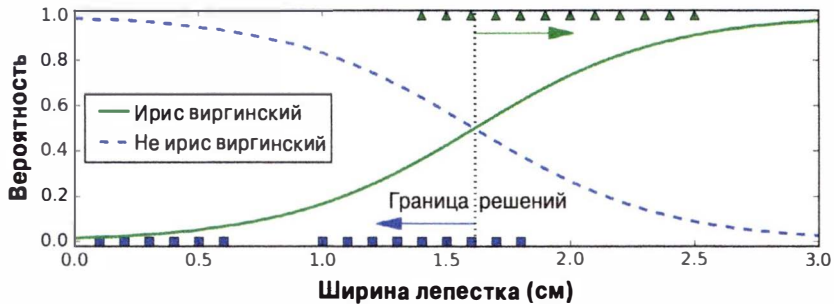


Рис. 4.23. Оценочные вероятности и граница решений

¹⁶ Фотографии воспроизведены из соответствующих страниц Википедии. Фотография ириса виргинского принадлежит Френку Мейфилду (Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>)), фотография ириса разноцветного принадлежит Д. Гордону И. Робертсону (Creative Commons BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)), а фотография ириса щетинистого находится в публичной собственности.

Ширина лепестков цветов ириса виргинского (представленного треугольниками) простирается от 1.4 до 2.5 см, в то время как цветы ириса других видов (представленные квадратами) обычно имеют меньшую ширину лепестка, находящуюся в диапазоне от 0.1 до 1.8 см.

Обратите внимание на наличие некоторого перекрытия. Выше примерно 2 см классификатор весьма уверен в том, что цветок представляет собой ирис виргинский (он выдает высокую вероятность принадлежности к данному классу), тогда как ниже 1 см он весьма уверен в том, что цветок не является ирисом виргинским (высокая вероятность принадлежности к классу “Не ирис виргинский”). В промежутке между указанными противоположностями классификатор не имеет уверенности. Тем не менее, если вы предложите ему спрогнозировать класс (используя метод `predict()`, а не `predict_proba()`), то он возвратит любой класс как наиболее вероятный. Следовательно, возле примерно 1.6 см существует *граница решений* (*decision boundary*), где обе вероятности равны 50%: если ширина лепестка больше 1.6 см, тогда классификатор спрогнозирует, что цветок — ирис виргинский, а иначе — не ирис виргинский (хотя и не будучи очень уверенным):

```
>>> log_reg.predict([[1.7], [1.5]])  
array([1, 0])
```

На рис. 4.24 показан тот же самый набор данных, но на этот раз с отображением двух признаков: ширина лепестка и длина лепестка. После обучения классификатор, основанный на логистической регрессии, может оценивать вероятность того, что новый цветок является ирисом виргинским, базируясь на упомянутых двух признаках. Пунктирная линия представляет точки, где модель производит оценку с 50%-ной вероятностью: это граница решений модели. Обратите внимание, что граница линейна¹⁷. Каждая параллельная линия представляет точки, в которых модель выдает специфическую вероятность, от 15% (слева внизу) до 90% (справа вверху). В соответствии с моделью все цветки выше правой верхней линии имеют более чем 90%-ный шанс быть ирисом виргинским.

Подобно другим линейным моделям логистические регрессионные модели могут быть регуляризованы с применением штрафов ℓ_1 или ℓ_2 . На самом деле Scikit-Learn по умолчанию добавляет штраф ℓ_2 .

¹⁷ Она представляет собой набор точек \mathbf{x} , так что справедливо уравнение $\theta_0 + \theta_1 x_1 + \theta_2 x_2 = 0$, определяющее прямую линию.

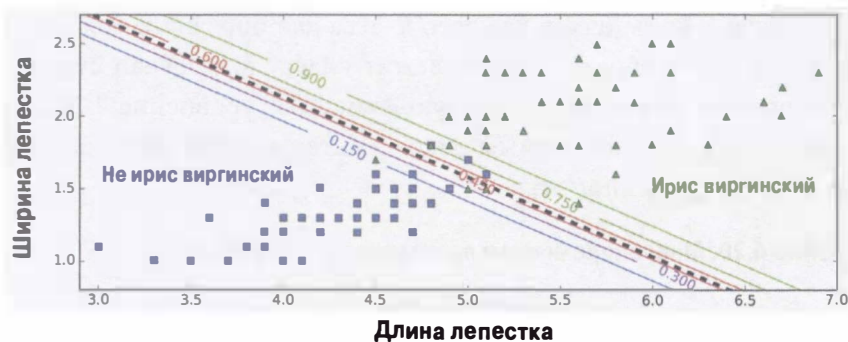


Рис. 4.24. Линейная граница решений



Гиперпараметром, управляющим силой регуляризации модели `LogisticRegression` из `Scikit-Learn`, является не `alpha` (как в других линейных моделях), а его инверсия: `C`. Чем выше значение `C`, тем *меньше* модель регуляризуется.

Многoperеменная логистическая регрессия

Логистическая регрессионная модель может быть обобщена для поддержки множества классов напрямую, без необходимости в обучении и комбинировании многочисленных двоичных классификаторов (как обсуждалось в главе 3). В результате получается *многoperеменная логистическая регрессия* (*softmax regression*), или *полиномиальная логистическая регрессия* (*multinomial logistic regression*).

Идея довольно проста: имея образец \mathbf{x} , многoperеменная логистическая регрессионная модель сначала вычисляет сумму очков $s_k(\mathbf{x})$ для каждого класса k и затем оценивает вероятность каждого класса, применяя к суммам очков *многoperеменную логистическую функцию* (*softmax function*), также называемую *нормализованной экспоненциальной* (*normalized exponential*) функцией. Уравнение для подсчета $s_k(\mathbf{x})$ должно выглядеть знакомым, т.к. оно точно такое же, как для прогноза линейной регрессии (уравнение 4.19).

Уравнение 4.19. Многoperеменная логистическая сумма очков для класса k

$$s_k(\mathbf{x}) = (\theta^{(k)})^T \cdot \mathbf{x}$$

Обратите внимание, что каждый класс имеет собственный отдельный вектор параметров $\theta^{(k)}$. Все эти векторы обычно хранятся как строки в *матрице параметров* Θ .

После подсчета сумм очков каждого класса для образца \mathbf{x} можно оценить вероятность \hat{p}_k , что образец принадлежит классу k , прогнав суммы очков через многопеременную логистическую функцию (уравнение 4.20): она вычисляет экспоненту каждой суммы очков и затем нормализует их (путем деления на сумму всех экспонент).

Уравнение 4.20. Многопеременная логистическая функция

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

- K — количество классов.
- $\mathbf{s}(\mathbf{x})$ — вектор, содержащий суммы очков каждого класса для образца \mathbf{x} .
- $\sigma(\mathbf{s}(\mathbf{x}))_k$ — оценочная вероятность того, что образец \mathbf{x} принадлежит классу k при заданных суммах очков каждого класса для этого образца.

Подобно классификатору, основанному на логистической регрессии, классификатор на базе многопеременной логистической регрессии прогнозирует класс с наивысшей оценочной вероятностью (который является просто классом с самой высокой суммой очков), как показано в уравнении 4.21.

Уравнение 4.21. Прогноз классификатора, основанного на многопеременной логистической регрессии

$$\hat{y} = \operatorname{argmax}_k \sigma(\mathbf{s}(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left((\theta^{(k)})^T \cdot \mathbf{x} \right)$$

- Операция **argmax** возвращает значение переменной, которая обращает функцию в максимум. В приведенном уравнении она возвращает значение k , обращающее в максимум оценочную вероятность $\sigma(\mathbf{s}(\mathbf{x}))_k$.



Классификатор на основе многопеременной логистической регрессии прогнозирует только один класс за раз (т.е. он многоклассовый, а не многовыходовый), поэтому он должен использоваться только с взаимоисключающими классами, такими как разные виды растений. Его нельзя применять для распознавания множества людей на одной фотографии.

Теперь, когда вам известно, каким образом модель оценивает вероятности и вырабатывает прогнозы, давайте взглянем на обучение. Цель в том, чтобы получить модель, которая дает оценку в виде высокой вероятности для целе-

вого класса (и, следовательно, низкую вероятность для остальных классов). Сведение к минимуму функции издержек, которая называется *перекрестной энтропией* (*cross entropy*) и показана в уравнении 4.22, должно привести к этой цели, поскольку она штрафует модель, когда та дает оценку в виде низкой вероятности для целевого класса. Перекрестная энтропия часто используется для измерения, насколько хорошо набор оценочных вероятностей классов соответствует целевым классам (мы еще будем ее применять несколько раз в последующих главах).

Уравнение 4.22. Функция издержек в форме перекрестной энтропии

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

- Значение $y_k^{(i)}$ равно 1, если целевым классом для i -того образца является k ; иначе оно равно 0.

Обратите внимание, что при наличии только двух классов ($K=2$) эта функция издержек эквивалентна функции издержек логистической регрессии (логарифмической потере; см. уравнение 4.17).

Перекрестная энтропия

Перекрестная энтропия происходит из теории информации. Предположим, что вы хотите эффективно передавать ежедневную информацию о погоде. Если есть восемь вариантов (солнечно, дождливо и т.д.), тогда вы могли бы кодировать каждый вариант, используя 3 бита, т.к. $2^3 = 8$. Однако если вы считаете, что почти каждый день будет солнечно, то было бы гораздо эффективнее кодировать “солнечно” только на одном бите (0), а остальные семь вариантов — на четырех битах (начиная с 1). Перекрестная энтропия измеряет среднее количество битов, действительно отправляемых на вариант. Если ваше допущение о погоде безупречно, тогда перекрестная энтропия будет просто равна энтропии самой погоды (т.е. присущей ей непредсказуемости). Но если ваше допущение ошибочно (скажем, часто идет дождь), то перекрестная энтропия будет больше на величину, называемую *расстоянием* (*расхождением*) *Кульбака–Лейблера* (*Kullback–Leibler divergence*).

Перекрестная энтропия между двумя распределениями вероятностей p и q определяется как $H(p, q) = -\sum_x p(x) \log q(x)$ (по крайней мере, когда распределения дискретны).

Вектор-градиент этой функции издержек в отношении $\theta^{(k)}$ имеет вид, показанный в уравнении 4.23.

Уравнение 4.23. Вектор-градиент перекрестной энтропии для класса k

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Теперь вы можете рассчитать вектор-градиент для каждого класса, после чего применить градиентный спуск (или любой другой алгоритм оптимизации) для нахождения матрицы параметров Θ , которая сводит к минимуму функцию издержек.

Давайте воспользуемся многопеременной логистической регрессией для классификации цветков ириса во все три класса. По умолчанию класс `LogisticRegression` из Scikit-Learn применяет стратегию “один против всех”, когда он обучается на более чем двух классах, но вы можете установить гиперпараметр `multi_class` в `"multinomial"`, чтобы переключить его на многопеременную логистическую регрессию. Вдобавок вы должны указать решатель, который поддерживает многопеременную логистическую регрессию, такой как решатель `"lbfgs"` (за дополнительными сведениями обращайтесь в документацию Scikit-Learn). По умолчанию он также применяет регуляризацию ℓ_2 , которой можно управлять с использованием гиперпараметра `C`.

```
X = iris["data"][:, (2, 3)] # длина лепестка, ширина лепестка
y = iris["target"]
softmax_reg = LogisticRegression(multi_class="multinomial",
                                solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

Таким образом, найдя в следующий раз ирис с лепестками длиной 5 см и шириной 2 см, вы можете предложить модели сообщить вид такого ириса, и она ответит, что с вероятностью 94.2% это ирис виргинский (класс 2) или с вероятностью 5.8% ирис разноцветный:

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[ 6.33134078e-07,  5.75276067e-02,  9.42471760e-01]])
```

На рис. 4.25 показаны результирующие границы решений, представленные с помощью фоновых цветов. Обратите внимание, что границы решений между любыми двумя классами линейны. На рис. 4.25 также приведены вероятности для класса “Ирис разноцветный”, представленного кривыми линиями (скажем, линия с меткой 0.450 представляет границу 45%-ной вероятности). Имейте в виду, что модель способна прогнозировать класс, который имеет оценочную вероятность ниже 50%. Например, в точке, где встречаются все границы решений, все классы имеют одинаковые оценочные вероятности 33%.

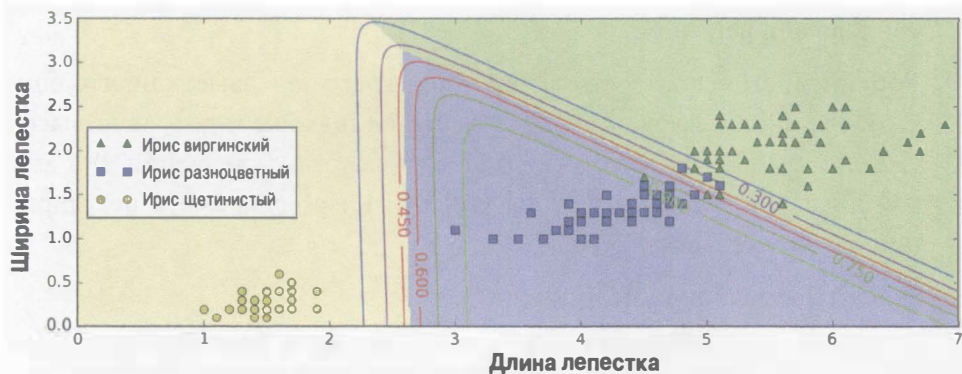


Рис. 4.25. Границы решений многопеременной логистической регрессии

Упражнения

1. Какой алгоритм обучения линейной регрессии вы можете применить, если у вас есть обучающий набор с миллионами признаков?
2. Предположим, что признаки в вашем обучающем наборе имеют очень разные масштабы. Какие алгоритмы могут пострадать от этого и как? Что вы можете с этим поделать?
3. Может ли градиентный спуск застрять в локальном минимуме при обучении логистической регрессионной модели?
4. Все ли алгоритмы градиентного спуска приводят к той же самой модели при условии, что вы позволяете им выполняться достаточно долго?
5. Допустим, вы используете пакетный градиентный спуск и вычерчиваете ошибку проверки на каждой эпохе. Если вы заметите, что ошибка проверки последовательно растет, тогда что, скорее всего, происходит? Как можно это исправить?

6. Хороша ли идея немедленно останавливать мини-пакетный градиентный спуск, когда ошибка проверки возрастает?
7. Какой алгоритм градиентного спуска (среди тех, которые обсуждались) быстрее всех достигнет окрестностей оптимального решения? Какой алгоритм действительно сойдется? Как вы можете заставить сойтись также остальные алгоритмы?
8. Предположим, что вы применяете *полиномиальную регрессию*. Вы вычерчиваете кривые обучения и замечаете крупный промежуток между ошибкой обучения и ошибкой проверки. Что произошло? Назовите три способа решения.
9. Допустим, вы используете гребневую регрессию и заметили, что ошибка обучения и ошибка проверки почти одинаковы и довольно высоки. Сказали бы вы, что модель страдает от высокого смещения или высокой дисперсии? Должны ли вы увеличить гиперпараметр регуляризации α или уменьшить его?
10. Почему вы захотели бы применять:
 - гребневую регрессию вместо обыкновенной линейной регрессии (т.е. без какой-либо регуляризации)?
 - лассо-регрессию вместо гребневой регрессии?
 - эластичную сеть вместо лассо-регрессии?
11. Предположим, что вы хотите классифицировать фотографии как сделанные снаружи/внутри и днем/ночью. Должны ли вы реализовать два классификатора, основанные на логистической регрессии, или один классификатор на базе многопеременной логистической регрессии?
12. Реализуйте пакетный градиентный спуск с ранним прекращением для многопеременной логистической регрессии (не используя Scikit-Learn).

Решения приведенных упражнений доступны в приложении А.

Методы опорных векторов

Метод опорных векторов (Support Vector Machine — SVM) — это очень мощная и универсальная модель машинного обучения, способная выполнять линейную или нелинейную классификацию, регрессию и даже выявление выбросов. Она является одной из самых популярных моделей в МО, и любой интересующийся МО обязан иметь ее в своем инструментальном комплекте. Методы SVM особенно хорошо подходят для классификации сложных, но небольших или средних наборов данных.

В настоящей главе объясняются ключевые концепции методов SVM, способы их использования и особенности их работы.

Линейная классификация SVM

Фундаментальную идею, лежащую в основе методов SVM, лучше раскрыть с применением иллюстраций. На рис. 5.1 показана часть набора данных об ирисах, который был введен в конце главы 4. Два класса могут быть легко и ясно разделены с помощью прямой линии (они *линейно сепарабельные*).

На графике слева приведены границы решений трех возможных линейных классификаторов. Модель, граница решений которой представлена пунктирной линией, до такой степени плоха, что даже не разделяет классы надлежащим образом. Остальные две модели прекрасно работают на данном обучающем наборе, но их границы решений настолько близки к образцам, что эти модели, вероятно, не будут выполняться так же хорошо на новых образцах. По контрасту сплошной линией на графике справа обозначена граница решений классификатора SVM; линия не только разделяет два класса, но также находится максимально возможно далеко от ближайших обучающих образцов. Вы можете считать, что классификатор SVM устанавливает самую широкую, какую только возможно, полосу (представленную параллельными пунктирными линиями) между классами. Это называется *классификацией с широким зазором (large margin classification)*.

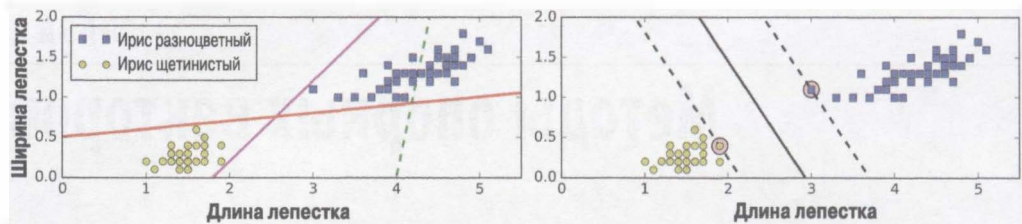


Рис. 5.1. Классификация с широким зазором

Обратите внимание, что добавление дополнительных обучающих образцов “вне полосы” вообще не будет влиять на границу решений: она полностью определяется образцами, расположенными по краям полосы (или “опирается” на них). Такие образцы называются *опорными векторами* (*support vector*); на рис. 5.1 они обведены окружностями.



Методы SVM чувствительны к масштабам признаков, как можно видеть на рис. 5.2: график слева имеет масштаб по вертикали, намного превышающий масштаб по горизонтали, поэтому самая широкая полоса близка к горизонтали. После масштабирования признаков (например, с использованием класса `StandardScaler` из Scikit-Learn) граница решений выглядит гораздо лучше (на графике справа).

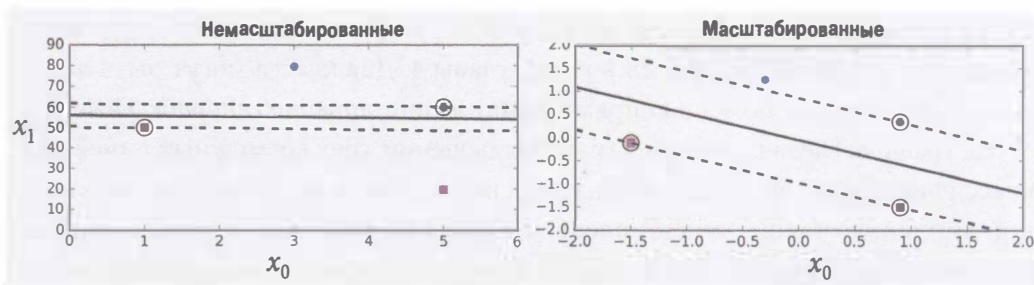


Рис. 5.2. Чувствительность к масштабам признаков

Классификация с мягким зазором

Если мы строго зафиксируем, что все образцы находятся вне полосы и на правой стороне, то получим *классификацию с жестким зазором* (*hard margin classification*). Классификации с жестким зазором присущи две главные проблемы. Во-первых, она работает, только если данные являются линейно сепарабельными. Во-вторых, она довольно чувствительна к выбросам.

На рис. 5.3 приведен набор данных об ирисах с только одним дополнительным выбросом: слева невозможно найти жесткий зазор, а справа граница решений сильно отличается от той, которую мы видели на рис. 5.1 без выброса, и модель, вероятно, не будет обобщаться с тем же успехом.

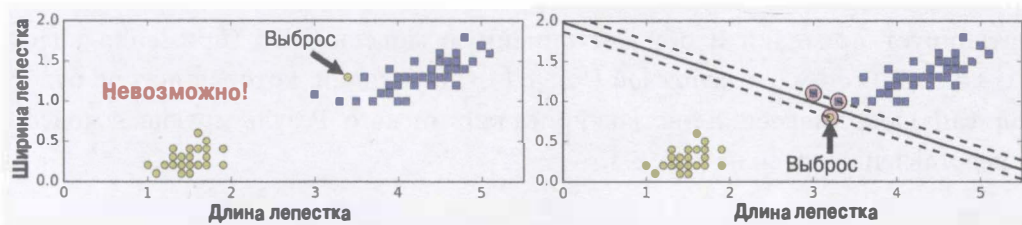


Рис. 5.3. Чувствительность к выбросам жесткого зазора

Чтобы избежать таких проблем, предпочтительнее применять более гибкую модель. Цель заключается в том, чтобы отыскать хороший баланс между удержанием полосы как можно более широкой и ограничением количества *нарушений зазора* (т.е. появления экземпляров, которые оказываются посредине полосы или даже на неправильной стороне). Это называется *классификацией с мягким зазором* (*soft margin classification*).

В классах SVM библиотеки Scikit-Learn вы можете управлять упомянутым балансом, используя гиперпараметр C : меньшее значение C ведет к более широкой полосе, но большему числу нарушений зазора. На рис. 5.4 показаны границы решений и зазоры двух классификаторов SVM с мягким зазором на нелинейно сепарабельном наборе данных. Слева за счет применения высокого значения C классификатор делает меньше нарушений зазора, но имеет меньший зазор. Справа из-за использования низкого значения C зазор гораздо больше, но многие образцы попадают на полосу. Однако, похоже, что второй классификатор будет лучше обобщаться: фактически даже на этом обучающем наборе он делает совсем немного ошибок в прогнозах, т.к. большинство нарушений зазора фактически находятся на корректной стороне границы решений.

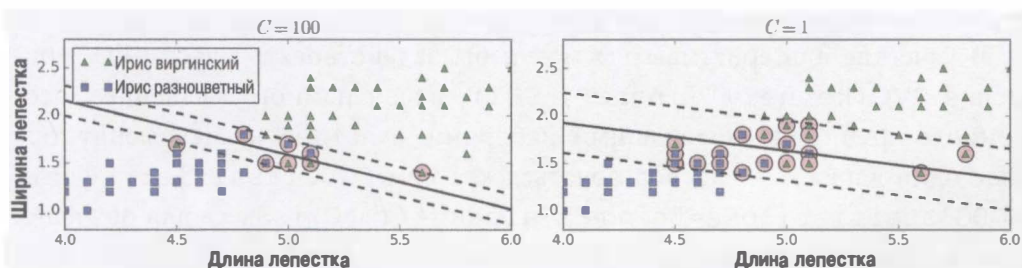


Рис. 5.4. Меньшее количество нарушений зазора или больший зазор



Если ваша модель SVM переобучается, тогда можете попробовать ее регуляризовать путем сокращения C .

Следующий код Scikit-Learn загружает набор данных об ирисах, масштабирует признаки и обучает линейную модель SVM (применяя класс `LinearSVC` с $C=1$ и *петлевой* (*hinge loss*) функцией, которая вскоре будет описана) для выявления цветков ириса виргинского. Результирующая модель представлена справа на рис. 5.4.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)] # длина лепестка, ширина лепестка
y = (iris["target"] == 2).astype(np.float64) # ирис виргинский

svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("linear_svc", LinearSVC(C=1, loss="hinge")),
])

svm_clf.fit(X, y)
```

Затем, как обычно, вы можете использовать модель для выработки прогнозов:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```



В отличие от классификаторов, основанных на логистической регрессии, классификаторы SVM не выдают вероятности для каждого класса.

В качестве альтернативы вы могли бы задействовать класс `SVC`, применив `SVC(kernel="linear", C=1)`, но он намного медленнее, особенно с крупными обучающими наборами, а потому не рекомендуется. Еще один вариант — воспользоваться классом `SGDClassifier` в форме `SGDClassifier(loss="hinge", alpha=1/(m*C))`. Здесь для обучения линейного классификатора SVM применяется стохастический градиентный спуск (см. главу 4). Он не сходится настолько быстро, как класс `LinearSVC`,

но может быть полезным для обработки гигантских наборов данных, которые не умещаются в памяти (внешнее обучение), или для решения задач динамической классификации.



Класс `LinearSVC` регуляризует член смещения, так что вы обязаны сначала центрировать обучающий набор, вычтя его среднее значение. Если вы масштабируете данные с использованием класса `StandardScaler`, то это делается автоматически. Вдобавок удостоверьтесь в том, что установили гиперпараметр `loss` в `"hinge"`, т.к. указанное значение не выбирается по умолчанию. Наконец, для лучшей производительности вы должны установить гиперпараметр `dual` в `False`, если только не существуют дополнительные признаки кроме тех, что есть у обучающих образцов (мы обсудим двойственность позже в настоящей главе).

Нелинейная классификация SVM

Хотя линейные классификаторы SVM эффективны и работают на удивление хорошо в многочисленных случаях, многие наборы данных далеки от того, чтобы быть линейно сепарабельными. Один из подходов к обработке нелинейных наборов данных предусматривает добавление дополнительных признаков, таких как полиномиальные признаки (как делалось в главе 4); в ряде ситуаций результатом может оказаться линейно сепарабельный набор данных. Рассмотрим график слева на рис. 5.5: он представляет простой набор данных с только одним признаком x_1 . Как видите, этот набор данных не является линейно сепарабельным. Но если вы добавите второй признак $x_2 = (x_1)^2$, тогда результирующий двумерный набор данных станет полностью линейно сепарабельным.

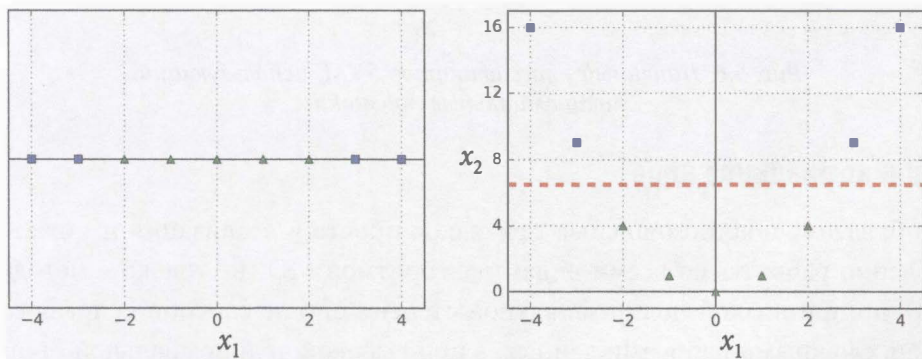


Рис. 5.5. Добавление признаков для превращения набора данных в линейно сепарабельный

Чтобы воплотить указанную идею с применением Scikit-Learn, вы можете создать экземпляр `Pipeline`, содержащий трансформатор `PolynomialFeatures` (как обсуждалось в разделе “Полиномиальная регрессия” главы 4), за которым следуют экземпляры `StandardScaler` и `LinearSVC`. Давайте проверим это на наборе данных `moons` (рис. 5.6):

```
from sklearn.datasets import make_moons
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures

polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])

polynomial_svm_clf.fit(X, y)
```

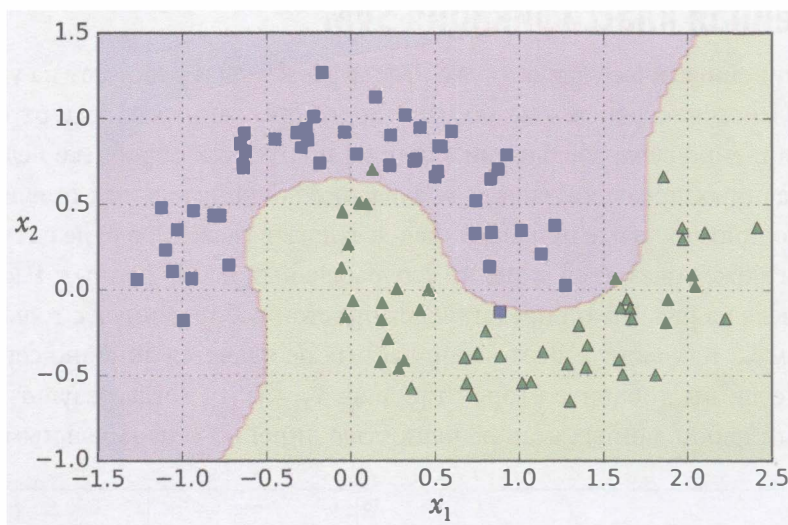


Рис. 5.6. Линейный классификатор SVM, использующий полиномиальные признаки

Полиномиальное ядро

Добавление полиномиальных признаков просто в реализации и может великолепно работать со всеми видами алгоритмов МО (не только с методами SVM), но при низкой полиномиальной степени оно не способно справиться с очень сложными наборами данных, а при высокой полиномиальной степени оно создает огромное количество признаков, делая модель крайне медленной.

К счастью, когда используются методы SVM, вы можете применить почти чудодейственный математический прием, называемый *ядерным трюком* (*kernel trick*), который вскоре будет объяснен. Он позволяет получить тот же самый результат, как если бы вы добавили много полиномиальных признаков, даже при полиномах очень высокой степени, без фактического их добавления. Таким образом, не происходит комбинаторного бурного роста количества признаков, поскольку в действительности вы не добавляете никаких признаков. Ядерный трюк выполняется классом `SVC`. Давайте проверим его на наборе данных `moons`:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
])
poly_kernel_svm_clf.fit(X, y)
```

Этот код обучает классификатор SVM, использующий полиномиальное ядро 3-й степени. Он представлен слева на рис. 5.7. Справа показан еще один классификатор SVM, который применяет полиномиальное ядро 10-й степени. Понятно, что если ваша модель переобучается, то вы можете сократить полиномиальную степень. И наоборот, если модель недообучается, тогда вы можете попробовать увеличить полиномиальную степень. Гиперпараметр `coef0` управляет тем, насколько сильно полиномы высокой степени влияют на модель в сравнении с полиномами низкой степени.

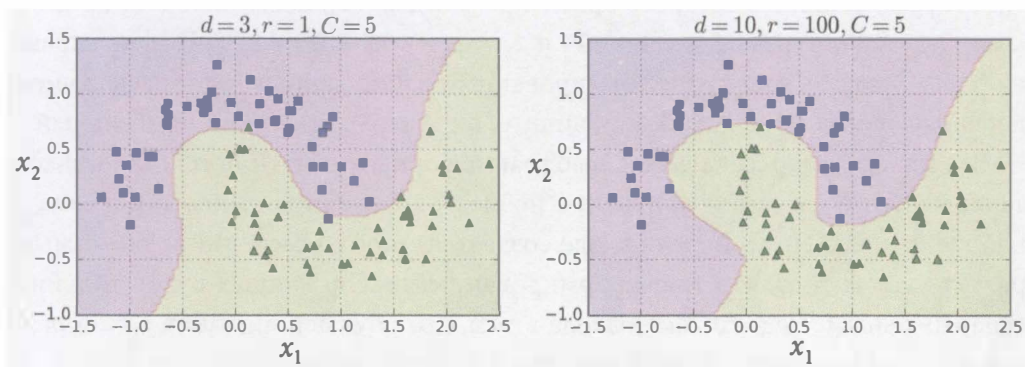


Рис. 5.7. Классификаторы SVM с полиномиальным ядром



Распространенный подход к поиску правильных значений гиперпараметров заключается в использовании решетчатого поиска (см. главу 2). Часто быстрее сначала сделать очень грубый решетчатый поиск, а затем провести более точный решетчатый поиск вокруг найденных лучших значений. Наличие хорошего представления о том, что фактически делает каждый гиперпараметр, также может помочь производить поиск в правильной части пространства гиперпараметров.

Добавление признаков близости

Еще одна методика решения нелинейных задач предусматривает добавление признаков, подсчитанных с применением *функции близости* (*similarity function*), которая измеряет, сколько сходства каждый образец имеет с отдельным *ориентиром* (*landmark*). Например, возьмем обсуждаемый ранее одномерный набор данных и добавим к нему два ориентира в $x_1 = -2$ и $x_1 = 1$ (график слева на рис. 5.8). Затем определим функцию близости как гауссову *радиальную базисную функцию* (*Radial Basis Function — RBF*) с $\gamma = 0.3$ (уравнение 5.1).

Уравнение 5.1. Гауссова функция RBF

$$\phi_{\gamma}(\mathbf{x}, \ell) = \exp \left(-\gamma \| \mathbf{x} - \ell \|^2 \right)$$

Это колоколообразная функция, изменяющаяся от 0 (очень далеко от ориентира) до 1 (на ориентире). Теперь мы готовы вычислить новые признаки. Взглянем на образец $x_1 = -1$: он находится на расстоянии 1 от первого ориентира и расстоянии 2 от второго ориентира. Следовательно, его новыми признаками будут $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ и $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. График справа на рис. 5.8 показывает трансформированный набор данных (с отбрасыванием первоначальных признаков). Как видите, он теперь линейно сепарабельный.

Вас может интересовать, как выбираются ориентиры. Простейший подход заключается в создании ориентира по местоположению каждого образца в наборе данных. При таком подходе создается много измерений и тем самым растут шансы того, что трансформированный набор данных будет линейно сепарабельным. Недостаток подхода в том, что обучающий набор с m образцами и n признаками трансформируется в обучающий набор с m образцами и m признаками (предполагая отбрасывание первоначальных признаков). Если обучающий набор очень крупный, тогда вы получите в равной степени большое количество признаков.

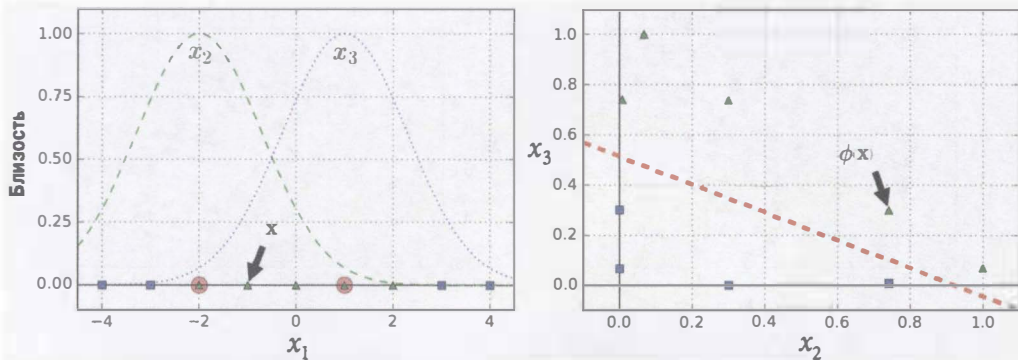


Рис. 5.8. Признаки близости, использующие гауссову функцию RBF

Гауссово ядро RBF

Подобно методу полиномиальных признаков метод признаков близости способен принести пользу любому алгоритму МО, но он может быть вычислительно затратным при подсчете всех дополнительных признаков, особенно в крупных обучающих наборах. Тем не менее, ядерный трюк снова делает свою “магию” SVM: он позволяет получить похожий результат, как если бы добавлялись многочисленные признаки близости, без фактического их добавления. Давайте испытаем *гауссово ядро RBF* (*Gaussian RBF kernel*) с применением класса SVC:

```
rbf_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
])
rbf_kernel_svm_clf.fit(X, y)
```

Модель представлена слева внизу на рис. 5.9. На других графиках изображены модели, обученные с разными значениями гиперпараметров `gamma` (γ) и `C`. Увеличение `gamma` приводит к сужению колоколообразной кривой (см. график слева на рис. 5.8), в результате чего сфера влияния каждого образца уменьшается: граница решений становится более неравномерной, извивающейся вблизи индивидуальных образцов. И наоборот, небольшое значение `gamma` делает колоколообразную кривую шире, поэтому образцы имеют большую сферу влияния, а граница решений оказывается более гладкой. Таким образом, γ действует аналогично гиперпараметру регуляризации: если ваша модель переобучается, тогда вы должны уменьшить значение `gamma`, а если недообучается — то увеличить его (подобно гиперпараметру `C`).

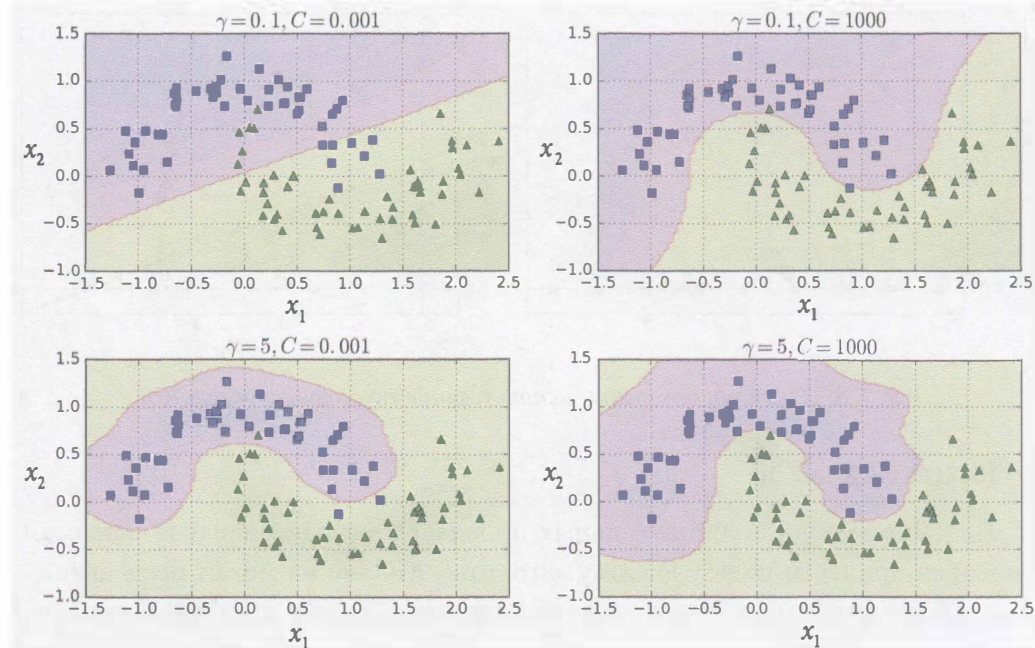


Рис. 5.9. Классификаторы SVM, использующие ядро RBF

Существуют и другие ядра, но они применяются гораздо реже. Например, некоторые ядра приспособлены к специфическим структурам данных. *Строковые ядра (string kernel)* иногда используются при классификации текстовых документов или цепочек ДНК (например, с применением *ядра строковых подпоследовательностей (string subsequence kernel)* или ядер на основе *расстояния Левенштейна (Levenshtein distance)*).



Имея на выбор так много ядер, как принять решение, какое ядро использовать? Примите в качестве эмпирического правила: вы должны всегда первым пробовать линейное ядро (помните, что `LinearSVC` гораздо быстрее `SVC(kernel="linear")`), особенно если обучающий набор очень большой либо изобилует признаками. Если обучающий набор не слишком большой, тогда вы должны испытать также гауссово ядро RBF; оно работает хорошо в большинстве случаев. При наличии свободного времени и вычислительной мощности вы также можете поэкспериментировать с рядом других ядер, применяя перекрестную проверку и решетчатый поиск, в особенности, когда существуют ядра, которые приспособлены к структуре данных вашего обучающего набора.

Вычислительная сложность

Класс `LinearSVC` основан на библиотеке *liblinear*, которая реализует оптимизированный алгоритм (<http://goo.gl/R635CH>) для линейных методов SVM¹. Он не поддерживает ядерный трюк, но масштабируется почти линейно с количеством обучающих образцов и количеством признаков: сложность его времени обучения составляет ориентировочно $O(m \times n)$.

Алгоритм занимает больше времени, когда требуется очень высокая точность. Точность управляется гиперпараметром допуска ϵ (называется `tol` в Scikit-Learn). Большинству задач классификации подходит стандартный допуск.

Класс `SVC` основан на библиотеке *libsvm*, которая реализует алгоритм (<http://goo.gl/a8HkE3>), поддерживающий ядерный трюк². Сложность времени обучения обычно находится между $O(m^2 \times n)$ и $O(m^3 \times n)$. К сожалению, это означает, что он становится невероятно медленным при большом количестве обучающих образцов (скажем, в случае сотен тысяч образцов). Такой алгоритм идеален для сложных, но небольших или средних обучающих наборов. Тем не менее, он хорошо масштабируется с количеством признаков, особенно *разреженных (sparse) признаков* (т.е. когда каждый образец имеет лишь немного ненулевых признаков). В этом случае алгоритм масштабируется приблизительно со средним числом ненулевых признаков на образце. В табл. 5.1 сравниваются классы классификации SVM из Scikit-Learn.

Таблица 5.1. Сравнение классов Scikit-Learn для классификации SVM

Класс	Сложность времени обучения	Поддержка внешнего обучения	Требуется ли масштабирование	Ядерный трюк
<code>LinearSVC</code>	$O(m \times n)$	Нет	Да	Нет
<code>SGDClassifier</code>	$O(m \times n)$	Да	Да	Нет
<code>SVC</code>	от $O(m^2 \times n)$ до $O(m^3 \times n)$	Нет	Да	Да

¹ “A Dual Coordinate Descent Method for Large-scale Linear SVM” (“Метод двойного покоординатного спуска для крупномасштабного линейного метода опорных векторов”), Лин и др. (2008 год).

² “Sequential Minimal Optimization (SMO)” (“Последовательная минимальная оптимизация”), Дж. Платт (1998 год).

Регрессия SVM

Как упоминалось ранее, алгоритм SVM довольно универсален: он поддерживает не только линейную и нелинейную классификацию, но также линейную и нелинейную регрессию. Прием заключается в инвертировании цели: вместо попытки приспособиться к самой широкой из возможных полосе между двумя классами, одновременно ограничивая нарушения зазора, регрессия SVM пробует уместить как можно больше образцов на полосе наряду с ограничением нарушений зазора (т.е. образцов *вне* полосы). Ширина полосы управляется гиперпараметром ϵ . На рис. 5.10 показаны две модели линейной регрессии SVM, обученные на случайных линейных данных, одна с широким зазором ($\epsilon = 1.5$) и одна с узким зазором ($\epsilon = 0.5$).

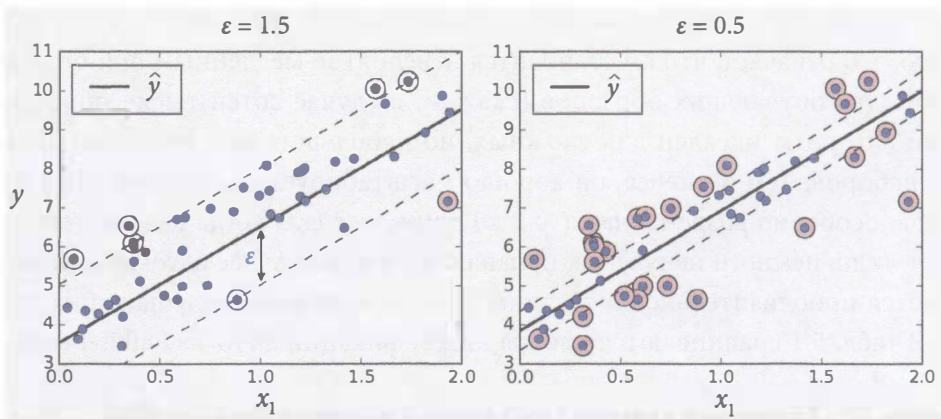


Рис. 5.10. Регрессия SVM

Добавление дополнительных обучающих образцов внутри зазора не влияет на прогнозы модели; соответственно, говорят, что модель *нечувствительна к ϵ* .

Для выполнения линейной регрессии SVM можно использовать класс `LinearSVR` из Scikit-Learn. Следующий код производит модель, представленную слева на рис. 5.10 (обучающие данные должны быть предварительно масштабированы и отцентрированы):

```
from sklearn.svm import LinearSVR
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

Для решения задач нелинейной регрессии можно применять *параметрически редуцированную (kernelized) модель SVM* (“kernelization” иногда пере-

водят как “кERNELИЗАЦИЯ” — *примеч. пер.*). Например, на рис. 5.11 демонстрируется регрессия SVM на случайном квадратичном обучающем наборе, использующая полиномиальное ядро 2-го порядка. На графике слева производилось немного регуляризации (т.е. крупное значение C), а на графике справа — гораздо больше регуляризации (т.е. небольшое значение C).

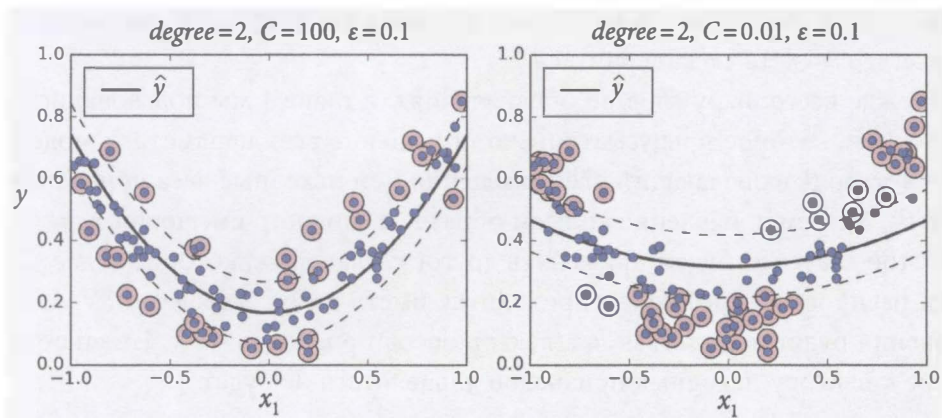


Рис. 5.11. Регрессия SVM, применяющая полиномиальное ядро 2-го порядка

Показанный ниже код порождает модель, представленную слева на рис. 5.11, с использованием класса `SVR` из Scikit-Learn (который поддерживает ядерный трюк). Класс `SVR` — это регрессионный эквивалент класса `SVC`, а класс `LinearSVR` — регрессионный эквивалент класса `LinearSVC`.

Класс `LinearSVR` масштабируется линейно с размером обучающего набора (подобно классу `LinearSVC`), в то время как класс `SVR` становится не в меру медленным, когда обучающий набор вырастает до крупного (подобно классу `SVC`).

```
from sklearn.svm import SVR
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```



Методы SVM также могут применяться для выявления выбросов; за дополнительными сведениями обращайтесь в документацию Scikit-Learn.

Внутренняя кухня

В настоящем разделе объясняется, каким образом методы SVM вырабатывают прогнозы и как работают их обучающие алгоритмы, начиная с линейных классификаторов SVM. Если вы только начали изучать МО, тогда можете благополучно пропустить раздел и перейти прямо к упражнениям в конце главы. Позже, когда возникнет желание глубже понять методы SVM, вы всегда сможете сюда вернуться.

Прежде всего, пару слов об обозначениях: в главе 4 мы пользовались соглашением, которое предусматривало помещение всех параметров модели в один вектор θ , включающий член смещения θ_0 и исходные веса признаков от θ_1 до θ_n , а затем добавление ко всем образцам входного смещения $x_0 = 1$.

В этой главе мы будем применять другое соглашение, которое более удобно (и распространено), когда приходится иметь дело с методами SVM: член смещения будет называться b , а вектор весов признаков — w . Никакое смещение к вектору исходных признаков добавляться не будет.

Функция решения и прогнозы

Модель линейной классификации SVM прогнозирует класс нового образца x , просто вычисляя функцию решения $w^T \cdot x + b = w_1x_1 + \dots + w_nx_n + b$: если результат положительный, то спрогнозированный класс \hat{y} является положительным (1), а иначе — отрицательным (0); см. уравнение 5.2.

Уравнение 5.2. Прогноз линейного классификатора SVM

$$\hat{y} = \begin{cases} 0, & \text{если } w^T \cdot x + b < 0, \\ 1, & \text{если } w^T \cdot x + b \geq 0 \end{cases}$$

На рис. 5.12 показана функция решения, которая соответствует модели справа на рис. 5.4: это двумерная плоскость, т.к. набор данных имеет два признака (ширина лепестка и длина лепестка). Граница решений — множество точек, где функция решения равна 0: это пересечение двух плоскостей, которое является прямой (представленной толстой сплошной линией)³.

³ В более общих чертах, когда имеется n признаков, функция решения представляет собой n -мерную гиперплоскость, а граница решений — $(n - 1)$ -мерную гиперплоскость.

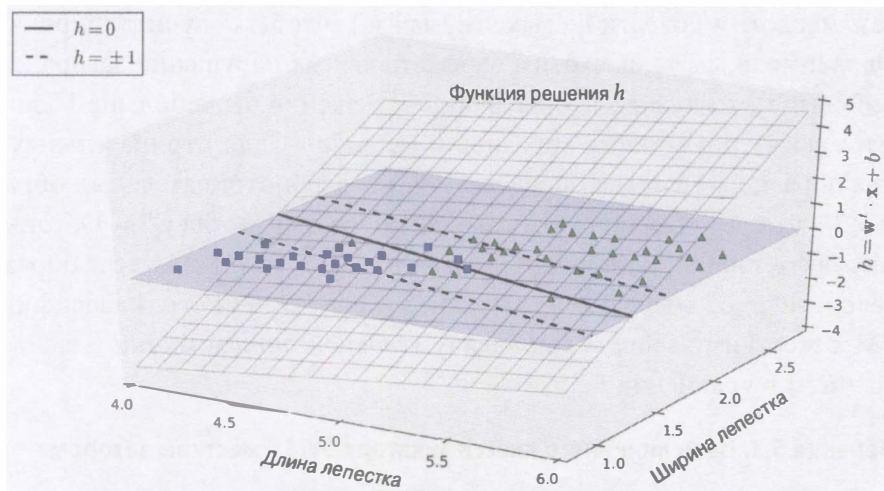


Рис. 5.12. Функция решения для набора данных об ирисах

Пунктирные линии представляют точки, где функция решения равна 1 или -1 : они параллельны и находятся на одинаковом расстоянии от границы решений, формируя вокруг нее зазор. Обучение линейного классификатора SVM означает нахождение таких значений \mathbf{w} и b , которые делают этот зазор как можно более широким, одновременно избегая нарушений зазора (жесткий зазор) или ограничивая их (мягкий зазор).

Цель обучения

Рассмотрим наклон функции решения: он тождественен норме вектора весов, $\|\mathbf{w}\|$. Если мы разделим наклон на 2, тогда точки, в которых функция решения равна ± 1 , будут в два раза дальше от границы решений. Другими словами, деление наклона на 2 умножит зазор на 2. Возможно, это легче представить себе в двумерном виде на рис. 5.13. Чем меньше вектор весов \mathbf{w} , тем шире зазор.

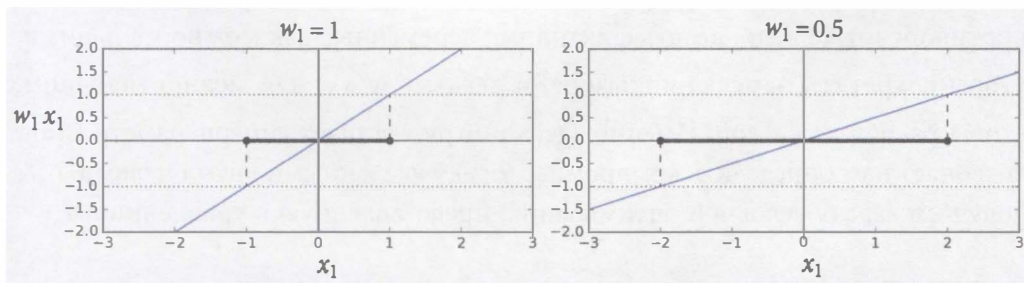


Рис. 5.13. Меньший вектор весов приводит к более широкому зазору

Итак, мы хотим довести до максимума $\|\mathbf{w}\|$, чтобы получить широкий зазор. Однако если мы также хотим избежать любых нарушений зазора (иметь жесткий зазор), то нужно, чтобы функция решения была больше 1 для всех положительных обучающих образцов и меньше -1 для отрицательных обучающих образцов. Если мы определим $t^{(i)} = -1$ для отрицательных образцов (когда $y^{(i)} = 0$) и $t^{(i)} = 1$ для положительных образцов (когда $y^{(i)} = 1$), тогда можем выразить такое ограничение как $t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ для всех образцов.

Следовательно, мы можем выразить цель линейного классификатора SVM с жестким зазором как задачу *условной оптимизации* (*constrained optimization*) в уравнении 5.3.

Уравнение 5.3. Цель линейного классификатора SVM с жестким зазором

$$\begin{aligned} &\underset{\mathbf{w}, b}{\text{минимизировать}} \quad \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} \\ &\text{при условии} \quad t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{для } i = 1, 2, \dots, m \end{aligned}$$



Мы минимизируем $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$, что равносильно $\frac{1}{2} \|\mathbf{w}\|^2$, вместо минимизации $\|\mathbf{w}\|$. Причина в том, что это даст тот же самый результат (поскольку значения \mathbf{w} и b , которые минимизируют какое-то значение, также минимизируют половину его квадрата), но $\frac{1}{2} \|\mathbf{w}\|^2$ имеет подходящую и простую производную (именно \mathbf{w}), в то время как $\|\mathbf{w}\|$ не дифференцируется для $\mathbf{w} = 0$. Алгоритмы оптимизации гораздо лучше работают с дифференцируемыми функциями.

Чтобы достичь цели мягкого зазора, нам необходимо ввести *фиктивную переменную* (*slack variable*) $\zeta^{(i)} \geq 0$ для каждого образца⁴: $\zeta^{(i)}$ измеряет, насколько i -тому образцу разрешено нарушать зазор. Теперь мы имеем две противоречивые цели: делать фиктивные переменные как можно меньшими, чтобы сократить нарушения зазора, и делать $\frac{1}{2} \mathbf{w}^T \cdot \mathbf{w}$ как можно меньшим, чтобы расширить зазор. Именно здесь в игру вступает гиперпараметр C : он позволяет нам определить компромисс между указанными двумя целями. Мы получаем задачу условной оптимизации, представленную в уравнении 5.4.

⁴ Дзета (ζ) — шестая буква греческого алфавита.

Уравнение 5.4. Цель линейного классификатора SVM с мягким зазором

$$\begin{aligned} & \underset{\mathbf{w}, b, \zeta}{\text{минимизировать}} \quad \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)} \\ & \text{при условии} \quad t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{и} \quad \zeta^{(i)} \geq 0 \quad \text{для} \quad i = 1, 2, \dots, m \end{aligned}$$

Квадратичное программирование

Задачи жесткого и мягкого зазора являются задачами выпуклой квадратичной оптимизации с линейными ограничениями. Такие задачи известны как задачи *квадратичного программирования* (*Quadratic Programming — QP*). Для решения задач QP доступны многие готовые решатели, использующие разнообразные методики, исследование которых выходит за рамки настоящей книги⁵. В уравнении 5.5 дана общая постановка задачи.

Уравнение 5.5. Задача квадратичного программирования

$$\begin{aligned} & \underset{\mathbf{p}}{\text{минимизировать}} \quad \frac{1}{2} \mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} + \mathbf{f}^T \cdot \mathbf{p} \\ & \text{при условии} \quad \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b} \end{aligned}$$

$$\text{где} \quad \left\{ \begin{array}{ll} \mathbf{p} & \text{— } n_p\text{-размерный вектор (} n_p = \text{ количество параметров),} \\ \mathbf{H} & \text{— матрица } n_p \times n_p, \\ \mathbf{f} & \text{— } n_p\text{-размерный вектор,} \\ \mathbf{A} & \text{— матрица } n_c \times n_p \text{ (} n_c = \text{ количество ограничений),} \\ \mathbf{b} & \text{— } n_c\text{-размерный вектор.} \end{array} \right.$$

Обратите внимание, что выражение $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ фактически определяет n_c ограничений: $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ для $i = 1, 2, \dots, n_c$, где $\mathbf{a}^{(i)}$ — вектор, содержащий элементы i -той строки \mathbf{A} , а $b^{(i)}$ — i -тый элемент \mathbf{b} .

Вы можете легко проверить, что если установить параметры QP следующим образом, то достигается цель линейного классификатора SVM с жестким зазором:

⁵ Чтобы узнать больше о квадратичном программировании, можете начать с чтения книги Стивена Бойда и Ливена Ванденберга *Convex Optimization* (Cambridge University Press, 2004 год) (<http://goo.gl/FGXuLw>) или просмотреть курс видеолекций (на английском языке) от Ричарда Брауна (<http://goo.gl/rTo3Af>).

- $n_p = n + 1$, где n — количество признаков (+1 предназначено для члена смещения);
- $n_c = m$, где m — количество обучающих образцов;
- \mathbf{H} — единичная матрица $n_p \times n_p$ кроме нуля в левой верхней ячейке (чтобы проигнорировать член смещения);
- $\mathbf{f} = \mathbf{0}$, n_p -размерный вектор, заполненный нулями;
- $\mathbf{b} = \mathbf{1}$, n_c -размерный вектор, заполненный единицами;
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{X}}^{(i)}$, где $\dot{\mathbf{X}}^{(i)}$ тождественно $\mathbf{x}^{(i)}$ с добавочным признаком смещения $\dot{\mathbf{x}}_0$.

Таким образом, один из способов обучения линейного классификатора SVM с жестким зазором предусматривает просто применение готового решателя QP с передачей ему предшествующих параметров. Результирующий вектор \mathbf{p} будет содержать член смещения $b = p_0$ и веса признаков $w_i = p_i$ для $i = 1, 2, \dots, m$. Аналогично вы можете использовать решатель QP для решения задачи мягкого зазора (взгляните на упражнения в конце главы).

Тем не менее, чтобы применить ядерный трюк, мы собираемся рассмотреть другую задачу условной оптимизации.

Двойственная задача

Имея задачу условной оптимизации, известную как *прямая задача (primal problem)*, можно выразить отличающуюся, но тесно связанную задачу, которая называется *двойственной задачей (dual problem)*. Решение двойственной задачи обычно дает нижнюю границу решения прямой задачи, но при некоторых условиях двойственная задача может даже иметь те же самые решения, что и прямая задача. К счастью, задача SVM удовлетворяет этим условиям⁶, так что вы можете выбирать, решать прямую задачу или двойственную задачу; обе они будут иметь то же самое решение. В уравнении 5.6 показана двойственная форма цели линейного классификатора SVM (если вас интересует, как выводить двойственную задачу из прямой задачи, тогда обратитесь в приложение B).

⁶ Целевая функция является выпуклой, а ограничения неравенства представляют собой непрерывно дифференцируемые и выпуклые функции.

Уравнение 5.6. Двойственная форма цели линейного классификатора SVM

$$\begin{aligned} & \underset{\alpha}{\text{минимизировать}} \quad \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ & \text{при условии} \quad \alpha^{(i)} \geq 0 \quad \text{для} \quad i = 1, 2, \dots, m \end{aligned}$$

После нахождения вектора $\hat{\alpha}$, сводящего к минимуму это уравнение (используя решатель QR), с применением уравнения 5.7 вы можете вычислить $\hat{\mathbf{w}}$ и \hat{b} , которые минимизируют прямую задачу.

Уравнение 5.7. От решения двойственной задачи к решению прямой задачи

$$\begin{aligned} \hat{\mathbf{w}} &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)} \\ \hat{b} &= \frac{1}{n_s} \sum_{i=1}^m \left(t^{(i)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)} \right) \\ & \quad \hat{\alpha}^{(i)} > 0 \end{aligned}$$

Двойственная задача решается быстрее прямой, когда количество обучающих образцов меньше количества признаков. Что более важно, становится возможным ядерный трюк, в то время как при решении прямой задачи он невозможен. Итак, что же собой представляет этот самый ядерный трюк?

Параметрически редуцированные методы SVM

Предположим, что вы хотите применять полиномиальную трансформацию второй степени к двумерному обучающему набору (такому как `moons`) и затем обучать линейный классификатор SVM на трансформированном обучающем наборе. В уравнении 5.8 показана *полиномиальная отображающая функция* (*mapping function*) второй степени ϕ , которую желательно применять.

Уравнение 5.8. Полиномиальное отображение второй степени

$$\phi(\mathbf{x}) = \phi \left(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \right) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$$

Обратите внимание, что трансформированный вектор является трехмерным, а не двумерным. Давайте посмотрим, что получится в результате

применения такого полиномиального отображения второй степени к паре двумерных векторов, \mathbf{a} и \mathbf{b} , и вычисления скалярного произведения трансформированных векторов (уравнение 5.9).

Уравнение 5.9. Ядерный трюк для полиномиального отображения второй степени

$$\begin{aligned}\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2} a_1 a_2 \\ a_2^2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1^2 \\ \sqrt{2} b_1 b_2 \\ b_2^2 \end{pmatrix} = a_1^2 b_1^2 + 2a_1 b_1 a_2 b_2 + a_2^2 b_2^2 \\ &= (a_1 b_1 + a_2 b_2)^2 = \left(\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \cdot \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \cdot \mathbf{b})^2\end{aligned}$$

Как вам это? Скалярное произведение трансформированных векторов равно квадрату скалярного произведения исходных векторов:

$$\phi(\mathbf{a})^T \cdot \phi(\mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2.$$

Теперь о сути: если вы примените трансформацию ϕ ко всем обучающим образцам, тогда двойственная задача (см. уравнение 5.6) будет содержать скалярное произведение $\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(j)})$. Но если ϕ — полиномиальная трансформация второй степени, определенная в уравнении 5.8, то вы можете заменить это скалярное произведение трансформированных векторов просто на $(\mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)})^2$. Таким образом, в действительности вы вообще не нуждаетесь в трансформации обучающих образцов: всего лишь замените скалярное произведение в уравнении 5.6 его квадратом. Результат будет абсолютно тем же самым, как если бы вы не поленились фактически трансформировать обучающий набор и затем подогнали какой-то линейный алгоритм SVM, но такой трюк делает весь процесс гораздо более эффективным с вычислительной точки зрения. В этом заключается сущность ядерного трюка.

Функция $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \cdot \mathbf{b})^2$ называется *полиномиальным ядром* второй степени. В машинном обучении *ядро* (*kernel*) — это функция, которая способна вычислять скалярное произведение $\phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$, базируясь только на исходных векторах \mathbf{a} и \mathbf{b} , без необходимости в вычислении трансформации ϕ (или даже знания о ней).

В уравнении 5.10 приведен список самых распространенных ядер.

Уравнение 5.10. Распространенные ядра

Линейное: $K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$

Полиномиальное: $K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$

Гауссово RBF: $K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$

Сигмоидальное: $K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \cdot \mathbf{b} + r)$

Теорема Мерсера

Согласно *теореме Мерсера* (*Mercer's theorem*), если функция $K(\mathbf{a}, \mathbf{b})$ соблюдает несколько математических условий, называемых *условиями Мерсера* (K должна быть непрерывной, симметричной в своих аргументах, так что $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, и т.д.), тогда существует функция ϕ , которая отображает \mathbf{a} и \mathbf{b} на другое пространство (возможно, с гораздо большей размерностью), такое что $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \cdot \phi(\mathbf{b})$. Таким образом, вы можете использовать K как ядро, поскольку знаете, что ϕ существует, даже если вам неизвестно, что собой представляет ϕ . В случае гауссова ядра RBF можно показать, что ϕ фактически отображает каждый обучающий образец на бесконечномерное пространство, поэтому хорошо, что вам не придется действительно выполнять отображение!

Следует отметить, что некоторые часто употребляемые ядра (такие как сигмоидальное ядро) не соблюдают все условия Мерсера, но на практике в целом работают нормально.

По-прежнему осталась одна загвоздка, которую нужно уладить. Уравнение 5.7 показывает, как перейти от двойственного решения к прямому решению в случае линейного классификатора SVM, но если вы применяете ядерный трюк, то оказываетесь с уравнениями, которые включают $\phi(\mathbf{x}(i))$. На самом деле $\hat{\mathbf{w}}$ обязано иметь то же количество измерений, что и $\phi(\mathbf{x}(i))$, которое может быть гигантским или даже бесконечным, поэтому вы не будете в состоянии вычислить его. Но как можно вырабатывать прогнозы, не зная $\hat{\mathbf{w}}$? Хорошая новость в том, что вы можете включить формулу для $\hat{\mathbf{w}}$ из уравнения 5.7 в функцию решения для нового образца $\mathbf{x}^{(n)}$ и получить уравнение с только скалярными произведениями между входными векторами. Это снова делает возможным использование ядерного трюка (уравнение 5.11).

Уравнение 5.11. Выработка прогнозов с помощью параметрически редуцированного метода SVM

$$\begin{aligned}
 h_{\widehat{\mathbf{W}}, \hat{b}}(\phi(\mathbf{x}^{(n)})) &= \widehat{\mathbf{W}}^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} = \left(\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^T \cdot \phi(\mathbf{x}^{(n)}) + \hat{b} \\
 &= \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \left(\phi(\mathbf{x}^{(i)})^T \cdot \phi(\mathbf{x}^{(n)}) \right) + \hat{b} \\
 &= \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \hat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \hat{b}
 \end{aligned}$$

Обратите внимание, что поскольку $\alpha^{(i)} \neq 0$ лишь для опорных векторов, выработка прогнозов включает в себя вычисление скалярного произведения нового входного вектора $\mathbf{x}^{(n)}$ только с опорными векторами, а не со всеми обучающими образцами. Конечно, вам также необходимо подсчитать член смещения \hat{b} , применяя тот же трюк (уравнение 5.12).

Уравнение 5.12. Вычисление члена смещения с использованием ядерного трюка

$$\begin{aligned}
 \hat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \widehat{\mathbf{W}}^T \cdot \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \left(\sum_{j=1}^m \hat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \cdot \phi(\mathbf{x}^{(i)}) \right) \\
 &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left(1 - t^{(i)} \sum_{\substack{j=1 \\ \hat{\alpha}^{(j)} > 0}}^m \hat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right)
 \end{aligned}$$

Если у вас разболелась голова, то это совершенно нормально: таков печальный побочный эффект от ядерного трюка.

Динамические методы SVM

Прежде чем завершить главу, давайте мельком взглянем на динамические классификаторы SVM (вспомните, что динамическое обучение означает постепенное обучение, обычно по мере поступления новых образцов).

Для линейных классификаторов SVM один из методов предусматривает применение градиентного спуска (например, используя `SGDClassifier`) для сведения к минимуму функции издержек в уравнении 5.13, которое является производным от прямой задачи. К сожалению, она сходится намного медленнее, чем методы, основанные на QR.

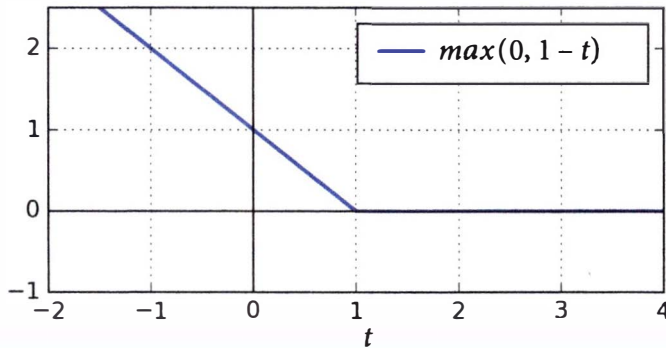
Уравнение 5.13. Функция издержек линейного классификатора SVM

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))$$

Первая сумма в функции издержек вынудит модель иметь небольшой вектор весов \mathbf{w} , приводя к более широкому зазору. Вторая сумма подсчитывает общее количество нарушений зазора. Нарушение зазора образца равно 0, если он расположен вне полосы на корректной стороне, либо иначе пропорционально расстоянию до корректной стороны полосы. Сведение к минимуму этого члена гарантирует, что модель будет нарушать зазор мало и редко.

Петлевая функция

Функция $\max(0, 1 - t)$ называется *петлевой* (*hinge loss*) и представлена ниже. Она равна 0, когда $t \geq 1$. Ее производная (наклон) равна -1 , если $t < 1$, и 0, если $t > 1$. Петлевая функция не является дифференцируемой при $t = 1$, но подобно лассо-регрессии (см. раздел “Лассо-регрессия” в главе 4) вы по-прежнему можете применять градиентный спуск, используя любой *субдифференциал* при $t = 1$ (т.е. любое значение между -1 и 0).



Также возможно реализовать динамические параметрически редуцированные методы SVM — например, руководствуясь работами “Инкрементное и декрементное обучение SVM” (<http://goo.gl/JEqVui>)⁷ или “Быстрые параметрически редуцированные классификаторы с динамическим

⁷ “Incremental and Decremental Support Vector Machine Learning” (“Инкрементное и декрементное обучение методами опорных векторов”), Ж. Кофенбергс, Т. Поджо (2001 год).

и активным обучением” (<https://goo.gl/hsoUHA>)⁸. Однако они реализованы в Matlab и C++. Для крупномасштабных нелинейных задач вы можете обдумать применение нейронных сетей (см. часть II).

Упражнения

1. Какая фундаментальная идея лежит в основе методов опорных векторов?
2. Что такое опорный вектор?
3. Почему важно масштабировать входные образцы при использовании методов SVM?
4. Может ли классификатор SVM выдать меру доверия, когда он классифицирует образец? Как насчет вероятности?
5. Какую форму задачи SVM — прямую или двойственную — вы должны применять для обучения модели на обучающем наборе с миллионами образцов и сотнями признаков?
6. Пусть вы обучаете классификатор SVM с ядром RBF. Кажется, он недообучается на обучающем наборе: вам следует увеличить или же уменьшить γ (γ (gamma))? Что скажете о C ?
7. Как вы должны установить параметры QP (H , f , A и b), чтобы решить задачу линейного классификатора SVM с мягким зазором, используя готовый решатель QP?
8. Обучите классификатор `LinearSVC` на линейно сепарабельном наборе данных. Затем обучите на том же наборе данных классификаторы `SVC` и `SGDClassifier`. Посмотрите, можете ли вы заставить их выдавать примерно одинаковые модели.
9. Обучите классификатор SVM на наборе данных MNIST. Поскольку классификаторы SVM являются двоичными, для классификации всех 10 цифр вам придется применять стратегию “один против всех”. Вы можете решить отрегулировать гиперпараметры с использованием небольшого проверочного набора, чтобы ускорить процесс. Какой правильности вы можете достичь?
10. Обучите регрессор SVM с помощью набора данных, содержащего цены на жилье в Калифорнии.

Решения приведенных упражнений доступны в приложении А.

⁸ “Fast Kernel Classifiers with Online and Active Learning” (“Быстрые параметрически редуцированные классификаторы с динамическим и активным обучением”) А. Борд, С. Эртекин, Д. Вестон, Л. Ботту (2005 год).

Деревья принятия решений

Подобно методам опорных векторов *деревья принятия решений* (*decision tree*) являются универсальными алгоритмами машинного обучения, которые могут заниматься задачами классификации и регрессии, включая даже многовыходовые задачи. Они представляют собой очень мощные алгоритмы, способные подгоняться к сложным наборам данных. Например, в главе 2 вы обучали модель *DecisionTreeRegressor* на наборе данных, содержащем цены на жилье в Калифорнии, великолепно подгоняя ее (фактически переобучая).

Деревья принятия решений также являются фундаментальными компонентами случайных лесов (см. главу 7), которые входят в число самых мощных алгоритмов МО, доступных на сегодняшний день.

В настоящей главе мы начнем с обсуждения того, как обучать, визуализировать и вырабатывать прогнозы с помощью деревьев принятия решений. Затем мы рассмотрим алгоритм обучения CART, используемый библиотекой Scikit-Learn, а также выясним, как регуляризировать деревья и применять их для задач регрессии. Наконец, мы взглянем на некоторые ограничения деревьев принятия решений.

Обучение и визуализация дерева принятия решений

Чтобы понять деревья принятия решений, давайте просто построим одно такое дерево и посмотрим, как оно вырабатывает прогнозы. Следующий код обучает классификатор *DecisionTreeClassifier* на наборе данных *iris* (см. главу 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # длина и ширина лепестка
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

Вы можете визуализировать обученное дерево принятия решений, сначала используя метод `export_graphviz()` для вывода файла определения диаграммы по имени `iris_tree.dot`:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file=image_path("iris_tree.dot"),
    feature_names=iris.feature_names[2:],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

Затем вы можете преобразовывать этот файл `.dot` в разнообразные форматы, такие как PDF или PNG, с применением инструмента командной строки `dot` из пакета `graphviz`¹. Приведенная ниже команда преобразует файл `.dot` в файл изображения `.png`:

```
$ dot -Tpng iris_tree.dot -o iris_tree.png
```

Ваше первое дерево принятия решений выглядит так, как показано на рис. 6.1.

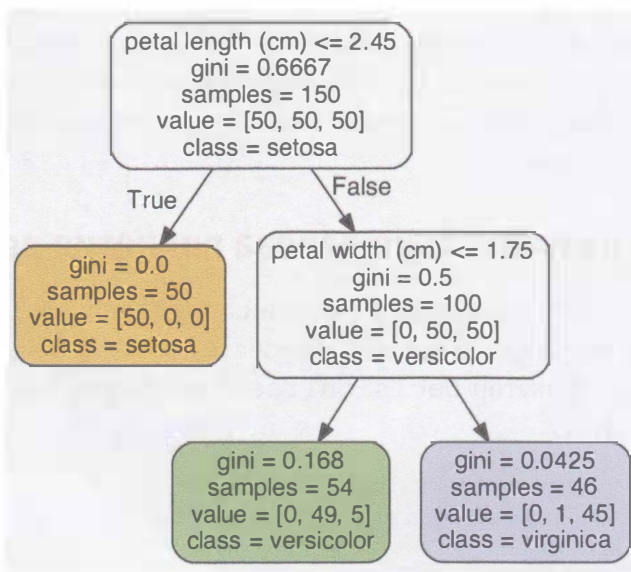


Рис. 6.1. Дерево принятия решений `iris_tree`

¹ `graphviz` — это программный пакет с открытым кодом для визуализации диаграмм, доступный по адресу <http://www.graphviz.org/>.

Вырабатывание прогнозов

Давайте выясним, как дерево, представленное на рис. 6.1, вырабатывает прогнозы. Предположим, что вы нашли цветок ириса и хотите его классифицировать. Вы начинаете с корневого узла (глубина 0, сверху): корневой узел спрашивает, меньше ли 2.45 см длина лепестка у цветка? Если меньше, тогда вы спускаетесь к левому дочернему узлу корневого узла (глубина 1, слева). В этом случае это *листовой узел* (т.е. он не имеет дочерних узлов), а потому он не задает никаких вопросов: вы можете просто посмотреть на спрогнозированный класс для данного узла, и дерево принятия решений прогнозирует, что ваш цветок — ирис щетинистый (`class=setosa`).

Теперь представим, что вы нашли еще один цветок, но на этот раз длина лепестка больше 2.45 см. Вы должны спуститься к правому дочернему узлу корневого узла (глубина 1, справа), который не является листовым, так что он задает новый вопрос: меньше ли 1.75 см ширина лепестка у цветка? Если меньше, тогда весьма вероятно, что ваш цветок — ирис разноцветный (глубина 2, слева), а если нет, то ирис виргинский (глубина 2, справа). Действительно, все так просто.



Одним из многих качеств деревьев принятия решений является то, что они требуют совсем небольшой подготовки данных. В частности, для них вообще не нужно масштабирование или центрирование признаков.

Атрибут `samples` узла подсчитывает, к скольким обучающим образцам он применяется. Например, 100 обучающих образцов имеют длину лепестка больше 2.45 см (глубина 1, справа), среди которых 54 образца имеют ширину лепестка меньше 1.75 см (глубина 2, слева). Атрибут `value` узла сообщает, к скольким обучающим образцам каждого класса применяется этот узел: например, правый нижний узел применяется к 0 образцам ириса щетинистого, 1 образцу ириса разноцветного и 45 образцам ириса виргинского. В заключение, атрибут `gini` (показатель Джини (Gini)) узла измеряет его *загрязненность* (*impurity*): узел “чист” (`gini=0`), если все обучающие образцы, к которым он применяется, принадлежат одному и тому же классу. Скажем, поскольку узел на глубине 1 слева применяется только к обучающим образцам ириса щетинистого, он чистый и его показатель Джини равен 0.

В уравнении 6.1 показано, как алгоритм обучения подсчитывает показатель Джини G_i для i -того узла. Например, узел на глубине 2 слева имеет показатель Джини, равный $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Вскоре мы обсудим еще одну *меру загрязненности*.

Уравнение 6.1. Загрязненность Джини

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

- $p_{i,k}$ — доля образцов класса k среди обучающих образцов в i -том узле.



Библиотека Scikit-Learn использует алгоритм CART, который выпускает только *двоичные деревья*: нелистовые узлы всегда имеют два дочерних узла (т.е. для вопросов существуют только ответы “да”/“нет”). Однако другие алгоритмы вроде ID3 могут выпускать деревья принятия решений с узлами, имеющими больше двух дочерних узлов.

На рис. 6.2 изображены границы решений для этого дерева принятия решений. Толстая вертикальная линия представляет границу решений корневого узла (глубина 0): длина лепестка = 2.45 см. Поскольку левая область чистая (только ирис щетинистый), она не может быть дополнительно расщеплена. Тем не менее, правая область загрязнена, и потому узел на глубине 1 справа расщепляет ее при ширине лепестка = 1.75 см (представленной пунктирной линией). Так как гиперпараметр `max_depth` был установлен в 2, дерево принятия решений останавливается прямо здесь. Однако если вы установите `max_depth` в 3, тогда каждый из двух узлов на глубине 2 добавит еще границы решений (представленные точечными линиями).

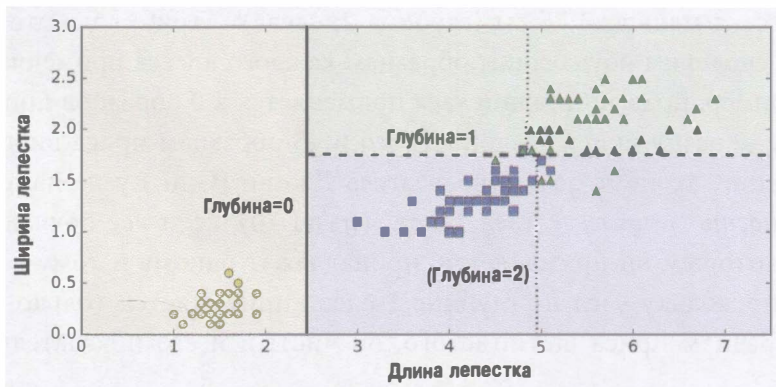


Рис. 6.2. Границы решений в дереве принятия решений

Интерпретация модели: белый ящик или черный ящик

Вы могли заметить, что деревья принятия решений довольно просты для понимания, а их решения легко интерпретировать. Такие модели часто называют *моделями белого ящика*. Как вы увидите, в противоположность этому случайные леса или нейронные сети в большинстве случаев рассматриваются как *модели черного ящика*. Они вырабатывают замечательные прогнозы, и вы можете легко проверить вычисления, которые выполняются для выдачи прогнозов; тем не менее, обычно трудно объяснить простыми терминами, почему были выработаны именно такие прогнозы. Например, если нейронная сеть сообщает о присутствии на фотографии конкретного человека, то трудно узнать, что на самом деле способствовало такому прогнозу: распознала ли модель глаза этого человека? Его рот? Его нос? Его обувь? Или даже диван, на котором он сидел? И наоборот, деревья принятия решений предоставляют простые и аккуратные правила классификации, которые в случае необходимости можно даже применять вручную (например, для классификации цветков).

Оценивание вероятностей классов

Дерево принятия решений также в состоянии оценивать вероятность принадлежности образца определенному классу k : сначала происходит обход дерева, чтобы найти листовой узел для данного образца, и затем возвращается пропорция обучающих образцов класса k в найденном узле. Для примера предположим, что вы обнаружили цветок с лепестками длиной 5 см и шириной 1.5 см. Соответствующий листовой узел находится на глубине 2 слева, поэтому дерево принятия решений должно выдать следующие вероятности: 0% для ириса щетинистого (0/54), 90.7% для ириса разноцветного (49/54) и 9.3% для ириса виргинского (5/54). И, конечно, если вы предложите спрогнозировать класс, дерево принятия решений должно выдать ирис разноцветный (класс 1), т.к. он имеет самую высокую вероятность. Давайте проверим сказанное:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Великолепно! Обратите внимание, что оценочные вероятности будут идентичными в любом другом месте правого нижнего прямоугольника на рис. 6.2 — например, если бы лепестки имели длину 6 см и ширину 1.5 см (хотя кажется очевидным, что в данном случае цветок с высокой вероятностью был бы ирисом виргинским).

Алгоритм обучения CART

Для обучения деревьев принятия решений (также называемых “растущими” деревьями) библиотека Scikit-Learn использует алгоритм *деревя классификации и регрессии* (*Classification And Regression Tree* — CART). Идея на самом деле довольно проста: алгоритм сначала расщепляет обучающий набор на два поднабора с применением единственного признака k и порога t_k (скажем, “длина лепестка ≤ 2.45 см”). Как он выбирает k и t_k ? Алгоритм ищет пару (k, t_k) , которая производит самые чистые поднаборы (взвешенные по их размеру). Функция издержек, которую он пытается минимизировать, имеет вид, представленный в уравнении 6.2.

Уравнение 6.2. Функция издержек CART для классификации

$$J(k, t_k) = \frac{m_{\text{левый}}}{m} G_{\text{левый}} + \frac{m_{\text{правый}}}{m} G_{\text{правый}},$$

где $\begin{cases} G_{\text{левый/правый}} & \text{измеряет загрязненность левого/правого поднабора,} \\ m_{\text{левый/правый}} & \text{— количество образцов в левом/правом поднаборе.} \end{cases}$

После того как алгоритм успешно расщепил обучающий набор на два поднабора, он расщепляет эти поднаборы, используя ту же самую логику, затем рекурсивно расщепляет подподнаборы и т.д. Алгоритм останавливает рекурсию по достижении максимальной глубины (определенной гиперпараметром `max_depth`) или когда не может найти расщепление, которое сократило бы загрязненность. Дополнительные условия остановки управляются рядом других параметров, которые мы вскоре рассмотрим (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf` и `max_leaf_nodes`).



Как видите, алгоритм CART является *поглощающим* или “жадным” (*greedy algorithm*): он жадно ищет оптимальное расщепление на верхнем уровне и затем повторяет процесс на каждом уровне. Он не проверяет, приведет ли расщепление к как можно более низкой загрязненности на несколько уровней ниже. Жадный алгоритм часто вырабатывает достаточно хорошее решение, но не гарантирует, что оно будет оптимальным.

К сожалению, нахождение оптимального дерева известно как *NP-полная* (*NP-complete*) задача²: она требует $O(\exp(m))$ времени, делая задачу трудной для решения даже в случае довольно малых обучающих наборов. Именно потому мы должны довольствоваться “достаточно хорошим” решением.

Вычислительная сложность

Вырабатывание прогнозов требует обхода дерева принятия решений от корня до какого-то листа. Деревья принятия решений обычно близки к сбалансированным, так что обход дерева принятия решений требует прохождения через приблизительно $O(\log_2(m))$ узлов³. Поскольку каждый узел требует проверки значения лишь одного признака, общая сложность прогноза составляет только $O(\log_2(m))$ независимо от количества признаков. Таким образом, прогнозы будут очень быстрыми даже при работе с крупными обучающими наборами.

Однако алгоритм обучения сравнивает все признаки (или не все, если установлен параметр `max_features`) на всех образцах в каждом узле. В результате получается сложность обучения $O(n \times m \log(m))$. Для небольших обучающих наборов (менее нескольких тысяч образцов) библиотека Scikit-Learn может ускорить обучение за счет предварительной сортировки данных (`presort=True`), но это значительно замедлит обучение при более крупных обучающих наборах.

² P — это набор задач, которые могут быть решены за полиномиальное время. NP — это набор задач, решения которых могут быть проверены за полиномиальное время. NP-трудная (NP-hard) задача — это такая задача, к которой может быть сокращена любая NP-задача за полиномиальное время. NP-полная задача — это NP-задача и NP-трудная задача. Крупным открытым математическим вопросом является выяснение $P = NP$ или нет. Если $P \neq NP$ (что кажется вероятным), тогда для любой NP-полной задачи не будет найдено ни одного полиномиального алгоритма (разве что на квантовом компьютере).

³ \log_2 — это двоичный логарифм. Он равносителен $\log_2(m) = \log(m) / \log(2)$.

Загрязненность Джини или энтропия?

По умолчанию применяется мера загрязненности Джини (Gini impurity; также называемая неоднородностью Джини — *примеч. пер.*), но вместо нее вы можете выбрать меру *энтропии* загрязненности, установив гиперпараметр `criterion` в `"entropy"`. Концепция энтропии появилась в термодинамике как мера молекулярного беспорядка: энтропия приближается к нулю, когда молекулы неподвижны и вполне упорядочены. Позже энтропия распространилась на разнообразные предметные области, включая *теорию информации Шеннона*, где она измеряет среднее количество информации сообщения⁴: энтропия равна нулю, когда все сообщения идентичны. В машинном обучении она часто используется в качестве меры загрязненности: энтропия набора равна нулю, когда он содержит образцы только одного класса. В уравнении 6.3 показано определение энтропии i -того узла. Например, узел на глубине 2 слева (см. рис. 6.1) имеет энтропию равную $-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31$.

Уравнение 6.3. Энтропия

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log(p_{i,k})$$

Итак, что вы должны применять — загрязненность Джини или энтропию? По правде говоря, большую часть времени особой разницы нет: они приводят к похожим деревьям. Загрязненность Джини слегка быстрее подсчитывать, поэтому она является хорошим вариантом по умолчанию. Тем не менее, когда разница есть, загрязненность Джини имеет тенденцию изолировать самый часто встречающийся класс в собственной ветви дерева, а энтропия — производить чуть более сбалансированные деревья⁵.

Гиперпараметры регуляризации

Деревья принятия решений выдвигают очень мало предположений об обучающих данных (в противоположность линейным моделям, кото-

⁴ Сокращение энтропии часто называют *приростом информации* (*information gain*).

⁵ Для получения дополнительных сведений ознакомьтесь с интересным анализом Себастьяна Рашки по адресу <http://goo.gl/UndTrO>.

рые очевидным образом предполагают, что данные линейны, к примеру). Оставленная не связанной ограничениями, древовидная структура будет адаптировать себя к обучающим данным, очень близко подгоняясь к ним и, скорее всего, допуская переобучение. Такая модель часто называется *непараметрической моделью* (*nonparametric model*), но не из-за отсутствия каких-либо параметров (они имеются и нередко в изобилии), а по той причине, что количество параметров перед обучением не определено, оттого структура модели вольна тесно привязываться к данным. В противоположность этому *параметрическая модель* (*parametric model*), подобная линейной модели, имеет предопределенное количество параметров, так что ее степень свободы ограничивается, сокращая риск переобучения (но увеличивая риск недообучения).

Во избежание переобучения обучающими данными вы должны ограничивать свободу дерева принятия решений во время обучения. Как вам уже известно, такой прием называется регуляризацией. Гиперпараметры регуляризации зависят от используемого алгоритма, но обычно вы можете, по крайней мере, ограничить максимальную глубину дерева принятия решений. В Scikit-Learn максимальная глубина управляется гиперпараметром `max_depth` (стандартным значением является `None`, которое означает отсутствие ограничения). Уменьшение `max_depth` будет регуляризовать модель и соответственно сокращать риск переобучения.

Класс `DecisionTreeClassifier` имеет несколько других параметров, которые похожим образом ограничивают форму дерева принятия решений: `min_samples_split` (минимальное число образцов, которые должны присутствовать в узле, прежде чем его можно будет расщепить), `min_samples_leaf` (минимальное количество образцов, которое должен иметь листовой узел), `min_weight_fraction_leaf` (то же, что и `min_samples_leaf`, но выраженное в виде доли от общего числа взвешенных образцов), `max_leaf_nodes` (максимальное количество листовых узлов) и `max_features` (максимальное число признаков, которые оцениваются при расщеплении каждого узла). Увеличение гиперпараметров `min_*` или уменьшение гиперпараметров `max_*` будет регуляризовать модель.



Другие алгоритмы работают, сначала обучая дерево принятия решений без ограничений и затем *отсекая* (удаляя) излишние узлы. Узел, все дочерние узлы которого листовые, считается излишним, если обеспечиваемое им улучшение чистоты не является *статистически значимым*. Стандартные статистические тесты, такие как *тест χ^2* , применяются для оценки вероятности того, что улучшение представляет собой исключительно результат случайности (которая называется *нулевой гипотезой* (*null hypothesis*)). Если эта вероятность, называемая *p-значением*, выше заданного порога (обычно составляющего 5% и управляемого гиперпараметром), тогда узел считается излишним и его дочерние узлы удаляются. Отсечение продолжается до тех пор, пока не будут удалены все излишние узлы.

На рис. 6.3 показаны два дерева принятия решений, обученные на наборе данных *moons* (введенном в главе 5). Дерево принятия решений слева обучалось со стандартными гиперпараметрами (т.е. без ограничений), а дерево принятия решений справа обучалось с `min_samples_leaf=4`. Совершенно очевидно, что модель слева переобучена, а модель справа, вероятно, будет обобщаться лучше.

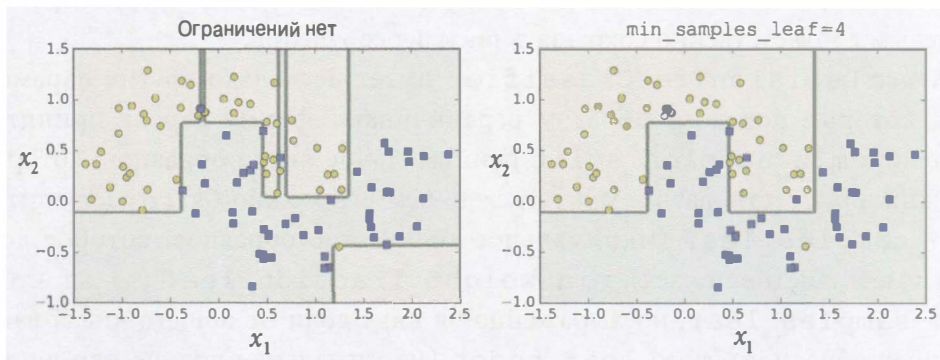


Рис. 6.3. Регуляризация с использованием `min_samples_leaf`

Регрессия

Деревья принятия решений также способны иметь дело с задачами регрессии. Давайте построим дерево регрессии с применением класса `DecisionTreeRegressor` из Scikit-Learn, обучив его на зашумленном квадратичном наборе данных с `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

Результирующее дерево изображено на рис. 6.4.

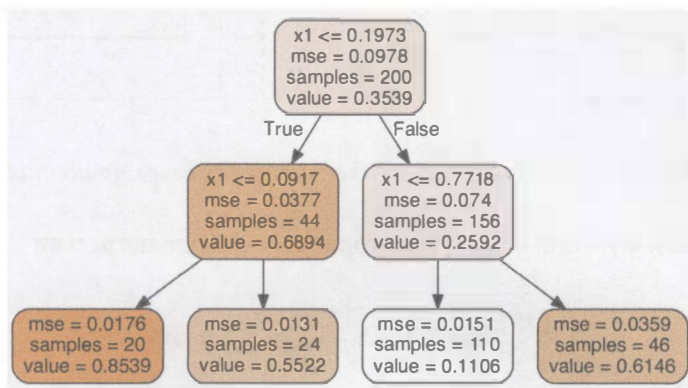


Рис. 6.4. Дерево принятия решений для регрессии

Это дерево выглядит очень похожим на дерево классификации, которое вы строили ранее. Главное отличие в том, что вместо прогнозирования класса в каждом узле оно прогнозирует значение. Например, пусть вы хотите выработать прогноз для нового образца с $x_1 = 0.6$. Вы обходите дерево, начиная с корневого узла, и в итоге доберетесь до листового узла, который прогнозирует $value = 0.1106$. Прогноз является просто средним целевым значением 110 обучающих образцов, ассоциированных с этим листовым узлом. Прогноз приводит к тому, что среднеквадратическая ошибка (MSE) на таких 110 образцах составляет 0.0151.

Прогнозы рассматриваемой модели показаны слева на рис. 6.5. Если вы установите `max_depth=3`, то получите прогнозы, представленные справа на рис. 6.5. Обратите внимание, что спрогнозированное значение для каждой области всегда будет средним целевым значением образцов в этой области. Алгоритм расщепляет каждую область так, чтобы расположить большинство обучающих образцов как можно ближе к спрогнозированному значению.

Алгоритм CART работает главным образом так же, как раньше, но только вместо попытки расщеплять обучающий набор способом, сводящим к минимуму загрязненность, он пробует расщеплять его способом, который минимизирует MSE. В уравнении 6.4 приведена функция издержек, которую алгоритм пытается довести до минимума.

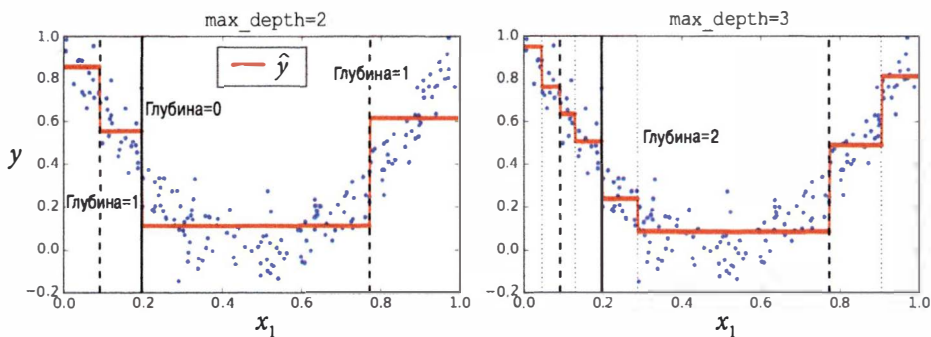


Рис. 6.5. Прогнозы регрессионных моделей в виде деревьев принятия решений

Уравнение 6.4. Функция издержек алгоритма CART для регрессии

$$J(k, t_k) = \frac{m_{\text{левый}}}{m} \text{MSE}_{\text{левый}} + \frac{m_{\text{правый}}}{m} \text{MSE}_{\text{правый}},$$

$$\text{где} \begin{cases} \text{MSE}_{\text{узел}} = \sum_{i \in \text{узел}} (\hat{y}_{\text{узел}} - y^{(i)})^2 \\ \hat{y}_{\text{узел}} = \frac{1}{m_{\text{узел}}} \sum_{i \in \text{узел}} y^{(i)} \end{cases}$$

Как и при задачах классификации, деревья принятия решений склонны к переобучению, когда имеют дело с задачами регрессии. Без какой-либо регуляризации (т.е. в случае использования стандартных значений гиперпараметров) вы получите прогнозы, показанные слева на рис. 6.6. Здесь очевидно крайне сильное переобучение обучающим набором. Простая установка `min_samples_leaf=10` в результате дает гораздо более рациональную модель, представленную справа на рис. 6.6.

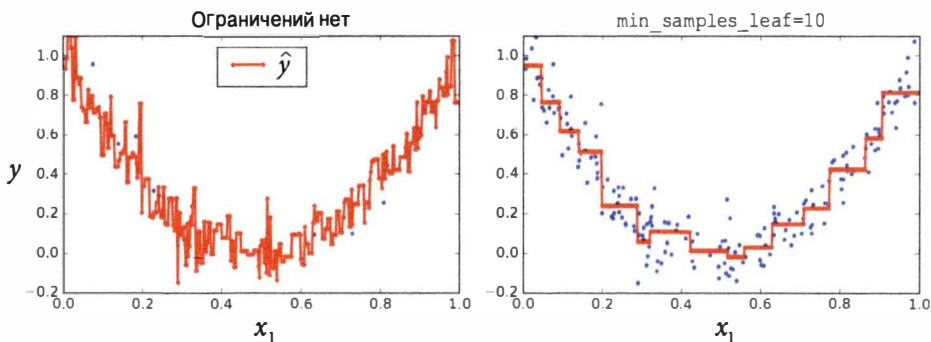


Рис. 6.6. Регуляризация регрессора в виде дерева принятия решений

Неустойчивость

Надо надеяться, что к настоящему моменту вы убеждены в том, что деревья принятия решений обладают многими достоинствами: они простые для понимания и интерпретации, легкие в применении, универсальные и мощные. Однако с ними связано несколько ограничений. Прежде всего, как вы могли заметить, деревья принятия решений предпочитают прямоугольные границы решений (все расщепления перпендикулярны той или иной оси), которые делают их чувствительными к поворотам обучающего набора. Например, на рис. 6.7 показан простой линейно сепарабельный набор данных: слева дерево принятия решений может его легко расщепить, тогда как справа, после поворота набора данных на 45° , граница решений выглядит излишне извилистой. Хотя оба дерева принятия решений идеально подогнаны к обучающему набору, весьма вероятно, что модель справа не будет хорошо обобщаться. Один из способов ограничения такой проблемы предусматривает использование метода PCA (Principal Component Analysis — анализ главных компонент; см. главу 8), который в результате дает лучшую ориентацию обучающих данных.

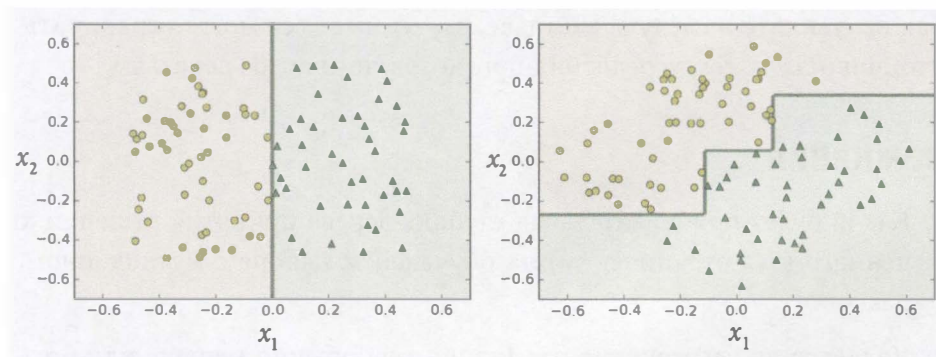


Рис. 6.7. Чувствительность к поворотам обучающего набора

Обычно основной вопрос с деревьями принятия решений связан с тем, что они сильно чувствительны к небольшим изменениям в обучающих данных. Например, если вы просто удалите из обучающего набора образец для самого широкого ириса разноцветного (с лепестками длиной 4.8 см и шириной 1.8 см) и обучите новое дерево принятия решений, то можете получить модель, представленную на рис. 6.8. Как видите, она выглядит крайне отличающейся от предыдущего дерева принятия решений (см. рис. 6.2).

В действительности, поскольку применяемый библиотекой Scikit-Learn алгоритм обучения является стохастическим⁶, вы можете получать очень разные модели даже на тех же самых обучающих данных (если только вы не установили гиперпараметр `random_state`).

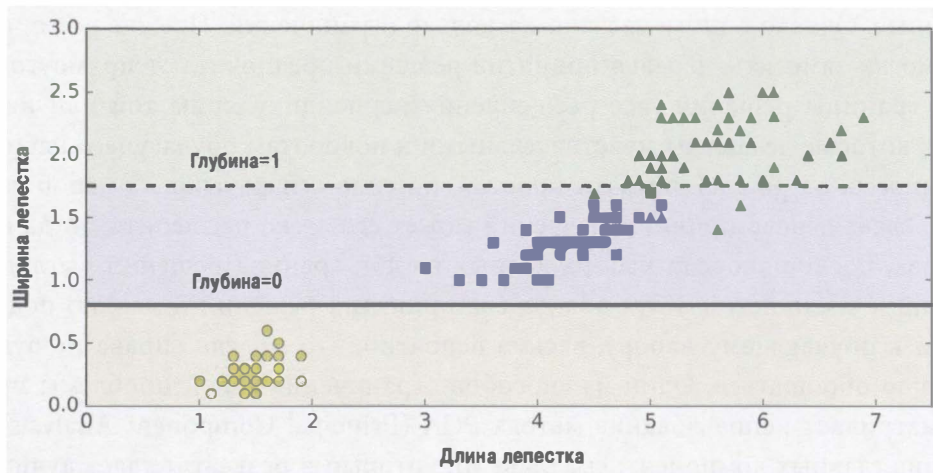


Рис. 6.8. Чувствительность к деталям обучающего набора

Как вы увидите в следующей главе, случайные леса могут ограничить эту неустойчивость путем усреднения прогнозов по многим деревьям.

Упражнения

1. Какой будет приблизительная глубина дерева принятия решений, обученного (без ограничений) на обучающем наборе с 1 миллионом образцов?
2. Является ли загрязненность Джини узла обычно меньше или больше, чем у его родительского узла? Она *обычно* меньше/больше или *всегда* меньше/больше?
3. Если дерево принятия решений переобучается обучающим набором, то хорошей ли идеей будет уменьшение `max_depth`?
4. Если дерево принятия решений недообучается на обучающем наборе, то хорошей ли идеей будет масштабирование входных признаков?

⁶ Он случайно выбирает набор признаков для оценки в каждом узле.

5. Если обучение дерева принятия решений на обучающем наборе, содержащем 1 миллион образцов, занимает один час, то сколько примерно времени потребуется для обучения другого дерева принятия решений на обучающем наборе, содержащем 10 миллионов образцов?
6. Если ваш обучающий набор содержит 100 000 образцов, то ускорит ли процесс обучения установка `presort=True`?
7. Обучите и точно настройте дерево принятия решений для набора данных `moons`.
 - а) Сгенерируйте набор данных `moons`, используя `make_moons(n_samples=10000, noise=0.4)`.
 - б) Расщепите его на обучающий набор и испытательный набор, применяя `train_test_split()`.
 - в) Воспользуйтесь решетчатым поиском с перекрестной проверкой (с помощью класса `GridSearchCV`), чтобы отыскать хорошие значения гиперпараметров для `DecisionTreeClassifier`. Подсказка: опробуйте разнообразные значения для `max_leaf_nodes`.
 - г) Обучите дерево принятия решений на полном обучающем наборе с применением найденных значений гиперпараметров и измерьте производительность модели на испытательном наборе. Вы должны получить правильность примерно от 85% до 87%.
8. Создайте лес.
 - а) Продолжая предыдущее упражнение, сгенерируйте 1 000 поднаборов обучающего набора, каждый из которых содержит 100 случайно выбранных образцов. Подсказка: для этого можете использовать класс `ShuffleSplit` из Scikit-Learn.
 - б) Обучите по одному дереву принятия решений на каждом поднаборе с применением найденных ранее значений гиперпараметров. Оцените полученные 1 000 деревьев принятия решений на испытательном наборе. Поскольку эти деревья принятия решений обучались на наборах меньшего размера, то они, скорее всего, будут выполняться хуже, чем первое дерево принятия решений, давая правильность около 80%.

- в) Теперь время для магии. Для каждого образца в испытательном наборе сгенерируйте прогнозы посредством 1 000 деревьев принятия решений и сохраните только наиболее частый прогноз (для этого можете воспользоваться функцией `mode ()` из SciPy). Это дает вам *мажоритарные прогнозы* (*majority-vote prediction*) на испытательном наборе.
- г). Оцените эти прогнозы на испытательном наборе: вы должны получить чуть *большую* правильность, чем у первой модели (выше от примерно 0.5% до 1.5%). Примите поздравления, ведь вы обучили классификатор на основе случайного леса!

Решения приведенных упражнений доступны в приложении А.

Ансамблевое обучение и случайные леса

Предположим, вы задаете сложный вопрос тысячам случайных людей и затем агрегируете их ответы. Во многих случаях вы обнаружите, что такой агрегированный ответ оказывается лучше, чем ответ эксперта. Это называется *коллективным разумом* (*wisdom of the crowd*). Аналогично если вы агрегируете прогнозы группы прогнозаторов (таких как классификаторы или регрессоры), то часто будете получать лучшие прогнозы, чем прогноз от наилучшего индивидуального прогнозатора. Группа прогнозаторов называется *ансамблем* (*ensemble*); соответственно, прием носит название *ансамблевого обучения* (*ensemble learning*), а алгоритм ансамблевого обучения именуется *ансамблевым методом* (*ensemble method*).

Например, вы можете обучать группу классификаторов на основе деревьев принятия решений, задействовав для каждого отличающийся случайный поднабор обучающего набора. Для выработки прогнозов вы лишь получаете прогнозы всех индивидуальных деревьев и прогнозируете класс, который стал обладателем большинства голосов (см. последнее упражнение в главе 6). Такой ансамбль деревьев принятия решений называется *случайным лесом* (*random forest*) и, несмотря на свою простоту, является одним из самых мощных алгоритмов МО, доступных на сегодняшний день.

Кроме того, как упоминалось в главе 2, вы часто будете использовать ансамблевые методы ближе к концу проекта, когда несколько хороших прогнозаторов уже построены, для объединения их в еще лучший прогнозатор. На самом деле выигрышные решения в состязаниях по МО зачастую включают в себя некоторое количество ансамблевых методов (самые знаменитые относятся к состязанию Netflix Prize (<http://netflixprize.com/>)).

В настоящей главе мы обсудим наиболее популярные ансамблевые методы, в том числе *бэггинг* (*bagging*), *бустинг* (*boosting*), *стекинг* (*stacking*) и ряд других. Мы также исследуем случайные леса.

Классификаторы с голосованием

Допустим, вы обучили несколько классификаторов, и каждый из них обеспечивает правильность около 80%. У вас может быть классификатор на основе логистической регрессии, классификатор SVM, классификатор на основе случайного леса, классификатор методом k ближайших соседей и вероятно ряд других (рис. 7.1).

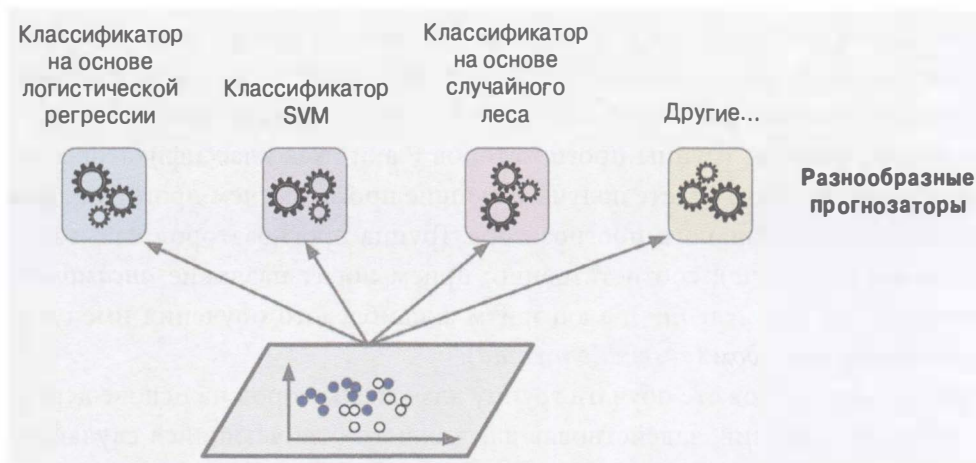


Рис. 7.1. Обучение разнообразных классификаторов

Очень простой способ создания еще лучшего классификатора предусматривает агрегирование прогнозов всех классификаторов и прогнозирование класса, который получает наибольшее число голосов. Такой мажоритарный классификатор называется классификатором с жестким голосованием (*hard voting classifier*) и демонстрируется на рис. 7.2.

Отчасти удивительно, но данный классификатор с голосованием часто достигает большей правильности, чем наилучший классификатор в ансамбле. На самом деле, даже если каждый классификатор является *слабым учеником* (*weak learner*), т.е. он лишь немногим лучше случайного угадывания, то ансамбль по-прежнему может быть *сильным учеником* (*strong learner*), обеспечивая высокую правильность, при условии, что есть достаточное количество слабых учеников и они достаточно разнообразны.

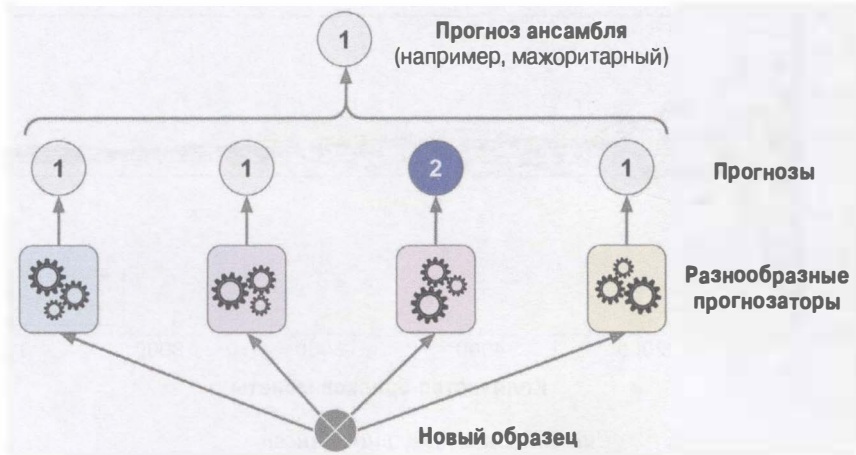


Рис. 7.2. Прогнозы классификатора с жестким голосованием

Как подобное возможно? Следующая аналогия сможет пролить свет на эту тайну. Предположим, у вас есть слегка несимметричная монета, которая имеет 51%-ный шанс упасть на лицевую сторону (орел) и 49%-ный шанс — на обратную сторону (решка). Если вы бросите ее 1 000 раз, то в целом получите примерно 510 орлов и 490 решек, и таким образом большинство орлов. Обратившись к математике, вы обнаружите, что вероятность получения большинства орлов после 1 000 бросков близка к 75%. Чем больше вы будете бросать монету, тем выше эта вероятность (скажем, при 10 000 бросков вероятность преодолевает планку 97%). Это связано с *законом больших чисел* (*law of large numbers*): по мере продолжения бросания монеты доля выпадения орлов становится все ближе и ближе к вероятности орлов (51%). На рис. 7.3 показано 10 серий бросков несимметричной монеты. Вы можете видеть, что с увеличением числа бросков доля выпадения орлов приближается к 51%. Со временем все 10 серий становятся настолько близкими к 51%, что оказываются устойчиво выше 50%.

Подобным же образом допустим, что вы строите ансамбль, содержащий 1 000 классификаторов, которые по отдельности корректны только 51% времени (едва ли лучше случайного угадывания). Если вы прогнозируете мажоритарный класс, то можете надеяться на правильность до 75%! Однако это справедливо, только если все классификаторы полностью независимы, допуская несвязанные ошибки, что явно не наша ситуация, т.к. они обучались на тех же самых данных. Скорее всего, классификаторы будут допускать ошибки одинаковых типов, а потому во многих случаях большинство голосов отдается некорректному классу, снижая правильность ансамбля.

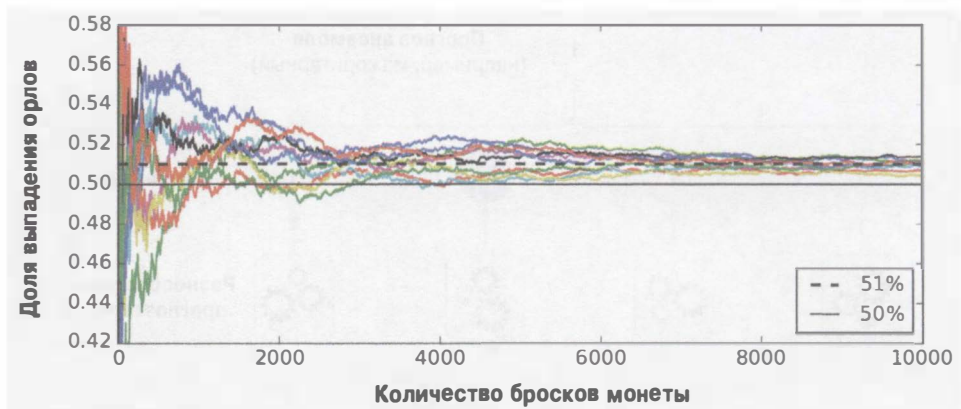


Рис. 7.3. Закон больших чисел



Ансамблевые методы работают лучше, когда прогнозаторы являются как можно более независимыми друг от друга. Один из способов получить несходные классификаторы заключается в том, чтобы обучать их с применением очень разных алгоритмов. Это улучшит шансы, что они будут допускать ошибки сильно различающихся типов, способствуя повышению правильности ансамбля.

Следующий код создает и обучает с помощью Scikit-Learn классификатор с голосованием, состоящий из несходных классификаторов (в качестве обучающего набора используется набор данных `moons`, представленный в главе 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Давайте выясним правильность каждого классификатора на испытательном наборе:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.872
SVC 0.888
VotingClassifier 0.896
```

Вы сделали это! Классификатор с голосованием слегка превосходит все индивидуальные классификаторы.

Если все классификаторы в состоянии оценивать вероятности классов (т.е. имеют метод `predict_proba()`), тогда вы можете сообщить Scikit-Learn о необходимости прогнозирования класса с наивысшей вероятностью класса, усредненной по всем индивидуальным классификаторам. Это называется *мягким голосованием* (*soft voting*). Оно часто добивается более высокой эффективности, чем жесткое голосование, потому что придает больший вес голосам с высоким доверием. Все, что вам потребуется — поменять `voting="hard"` на `voting="soft"` и удостовериться в способности классификаторов оценивать вероятности классов. По умолчанию такой вариант в классе `SVC` не принимается, поэтому вам понадобится установить его гиперпараметр `probability` в `True` (что заставит класс `SVC` применять перекрестную проверку для оценки вероятностей классов, замедляя обучение, и добавить метод `predict_proba()`). Если вы модифицируете предыдущий код для использования мягкого голосования, то обнаружите, что классификатор с голосованием добивается правильности свыше 91%!

Бэггинг и вставка

Как только что обсуждалось, один из способов получения наборов несходных классификаторов заключается в применении очень разных алгоритмов обучения. Другой подход предусматривает использование для каждого прогнозатора одного и того же алгоритма обучения, но обучение прогнозаторов на разных случайных поднаборах обучающего набора. Когда выборка осуществляется с заменой, такой метод называется *бэггингом* (*bagging*¹ — сокра-

¹ “Bagging Predictors” (“Прогнозаторы с бэггингом”), Л. Брейман (1996 год) (<http://goo.gl/o42tml>).

шение от *bootstrap aggregating*² (*бутстрэп-агрегирование*)). Когда выборка выполняется *без* замены, такой метод называется *вставкой* или *вклеиванием* (*pasting*³).

Другими словами, бэггинг и вставка позволяют производить выборку обучающих образцов по несколько раз множеством прогнозаторов, но только бэггинг разрешает осуществлять выборку обучающих образцов по несколько раз одним и тем же прогнозатором. Процесс выборки и обучения представлен на рис. 7.4.

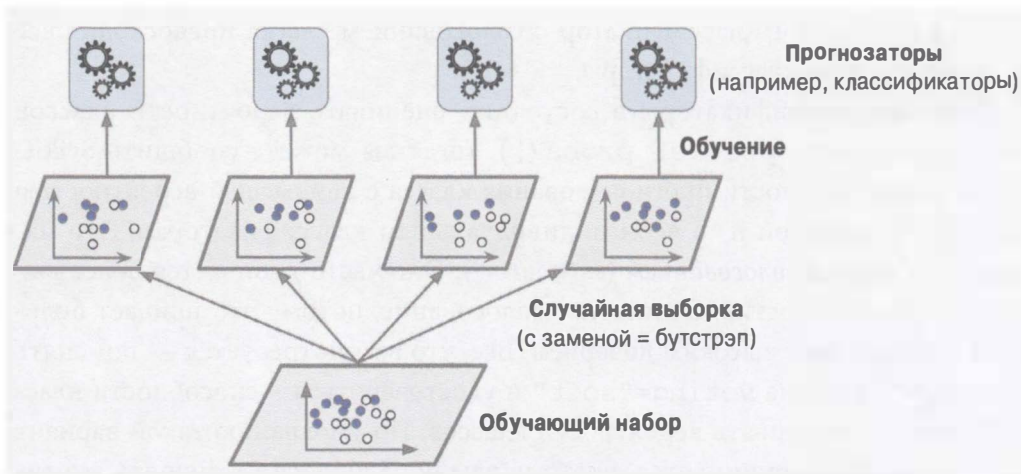


Рис. 7.4. Выборка и обучение с вставкой/бэггингом обучающего набора

После того, как все прогнозаторы обучены, ансамбль может вырабатывать прогноз для нового образца, просто агрегируя прогнозы всех прогнозаторов. Функция агрегирования обычно представляет собой *статистическую моду* (*statistical mode*) для классификации (т.е. самый частый прогноз, как и классификатор с жестким голосованием) или среднее для регрессии. Каждый индивидуальный прогнозатор имеет более высокое смещение, нежели если бы он обучался на исходном обучающем наборе, но агрегирование сокращает и смещение, и дисперсию⁴.

² В статистике повторная выборка с заменой (возвращением) называется *бутстрэппингом* (*bootstrapping*).

³ “Pasting small votes for classification in large databases and on-line” (“Вставка небольших голосов для классификации в крупных базах данных и динамический режим”), Л. Брейман (1999 год) (<http://goo.gl/BXm0pm>).

⁴ Смещение и дисперсия были введены в главе 4.

Обычно совокупный результат состоит в том, что ансамбль имеет похожее смещение, но меньшую дисперсию, чем одиночный прогнозатор, обученный на исходном обучающем наборе.

На рис. 7.4 вы видели, что все прогнозаторы могут обучаться параллельно, через разные процессорные ядра или даже разные серверы. Аналогично параллельно могут вырабатываться и прогнозы. Это одна из причин, по которой бэггинг и вставка являются настолько популярными методами: они очень хорошо масштабируются.

Бэггинг и вставка в Scikit-Learn

Библиотека Scikit-Learn предлагает простой API-интерфейс в виде класса `BaggingClassifier` для бэггинга и вставки (или `BaggingRegressor` для регрессии). Показанный ниже код обучает ансамбль из 500 классификаторов на основе деревьев принятия решений⁵, каждый из которых обучается на 100 обучающих экземплярах, случайно выбранных из обучающего набора с заменой (пример бэггинга, но если вы хотите взамен применять вставку, тогда просто установите `bootstrap=False`). Параметр `n_jobs` сообщает Scikit-Learn количество процессорных ядер для использования при обучении и прогнозировании (-1 указывает на необходимость участия всех доступных ядер):

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```



Класс `BaggingClassifier` автоматически выполняет вместо жесткого голосования мягкое голосование, если базовый классификатор может оценивать вероятности классов (т.е. если он имеет метод `predict_proba()`), как обстоит дело с классификаторами на основе деревьев принятия решений.

⁵ В качестве альтернативы параметр `max_samples` может быть установлен в число с плавающей точкой между 0.0 и 1.0, в случае чего максимальное количество образцов, подлежащих выборке, равно размеру обучающего набора, умноженному на `max_samples`.

На рис. 7.5 сравниваются границы решений одиночного дерева и ансамбля с бэггингом из 500 деревьев (созданного предыдущим кодом), которые обучались на наборе данных `moons`. Как видите, прогнозы ансамбля, вероятно, будут обобщаться гораздо лучше прогнозов одиночного дерева принятия решений: ансамбль имеет сопоставимое смещение, но меньшую дисперсию (он допускает приблизительно то же количество ошибок на обучающем наборе, но граница решений характеризуется меньшей нерегулярностью).

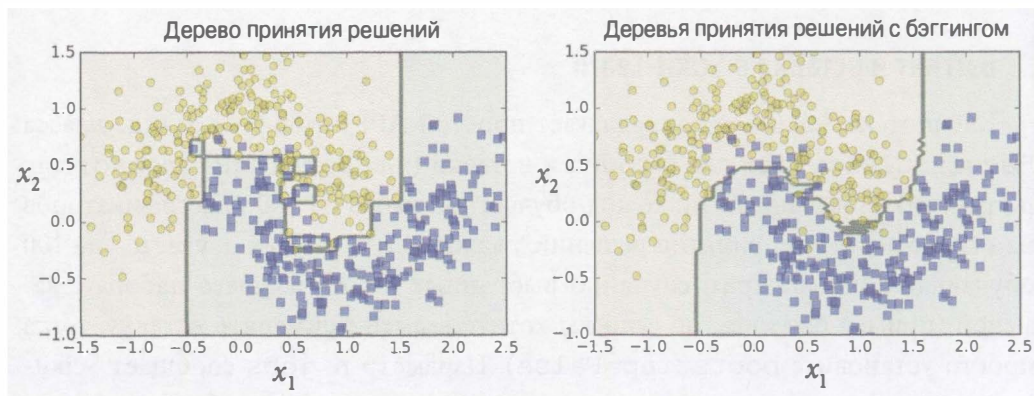


Рис. 7.5. Сравнение одиночного дерева принятия решений и ансамбля с бэггингом из 500 деревьев

Бутстрэппинг привносит чуть больше несходства в поднаборы, на которых обучается каждый прогнозатор, и потому бэггинг в итоге дает слегка выше смещение, чем вставка, но это также означает, что прогнозаторы будут менее зависимыми друг от друга, сокращая дисперсию ансамбля. В целом бэггинг часто приводит к лучшим моделям, что и является причиной, по которой ему обычно отдают предпочтение. Тем не менее, имея свободное время и вычислительную мощность, вы можете применить перекрестную проверку для оценки бэггинга и вставки и выбрать подход, который работает лучше.

Оценка на неиспользуемых образцах

При бэггинге некоторые образцы могут быть выбраны несколько раз для любого заданного прогнозатора, тогда как другие могут не выбираться вообще. По умолчанию класс `BaggingClassifier` производит выборку `m` обучающих образцов с заменой (`bootstrap=True`), где `m` — размер обучающего набора. Это означает, что для каждого прогнозатора будет выби-

ратся в среднем только около 63% обучающих образцов⁶. Оставшиеся 37% обучающих образцов, которые не выбираются, называются *неиспользуемыми* (*out-of-bag* — *oob*) образцами. Обратите внимание, что такие 37% образцов не одинаковы для всех прогнозаторов.

Поскольку прогнозатор никогда не видит образцы *oob* во время обучения, его можно оценивать на образцах *oob* без необходимости в наличии отдельного проверочного набора или проведении перекрестной проверки. Вы можете оценить сам ансамбль, усредняя оценки *oob* каждого прогнозатора.

При создании экземпляра `BaggingClassifier` в Scikit-Learn можно установить `oob_score=True`, чтобы запросить автоматическую оценку *oob* после обучения. Прием демонстрируется в следующем коде. Результирующая сумма оценки доступна через переменную `oob_score_`:

```
>>> bag_clf = BaggingClassifier(  
...     DecisionTreeClassifier(), n_estimators=500,  
...     bootstrap=True, n_jobs=-1, oob_score=True)  
...  
>>> bag_clf.fit(X_train, y_train)  
>>> bag_clf.oob_score_  
0.9013333333333332
```

Согласно проведенной оценке *oob* классификатор `BaggingClassifier`, скорее всего, достигнет правильности около 90.1% на испытательном наборе. Давайте проверим это:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred = bag_clf.predict(X_test)  
>>> accuracy_score(y_test, y_pred)  
0.91200000000000003
```

Мы получили правильность 91.2% на испытательном наборе — достаточно близко!

Функция решения *oob* для каждого обучающего образца также доступна через переменную `oob_decision_function_`. В данном случае (так как базовый оценщик имеет метод `predict_proba()`) функция решения возвращает вероятности классов для каждого обучающего образца. Например, оценка *oob* устанавливает, что первый обучающий образец с вероятностью 68.25% принадлежит положительному классу (и с вероятностью 31.75% — отрицательному классу):

⁶ С ростом *m* эта доля приближается к $1 - \exp(-1) \approx 63.212\%$.

```
>>> bag_clf.oob_decision_function_  
array([[ 0.31746032,  0.68253968],  
       [ 0.34117647,  0.65882353],  
       [ 1.          ,  0.          ],  
       ...  
       [ 1.          ,  0.          ],  
       [ 0.03108808,  0.96891192],  
       [ 0.57291667,  0.42708333]])
```

Методы случайных участков и случайных подпространств

Класс `BaggingClassifier` также поддерживает выборку признаков. Процесс управляется двумя гиперпараметрами: `max_features` и `bootstrap_features`. Они работают в таком же духе, как `max_samples` и `bootstrap`, но предназначены для выборки признаков, а не выборки образцов. Таким образом, каждый прогнозатор будет обучаться на случайном поднаборе входных признаков.

Это особенно полезно, когда вы имеете дело с исходными данными высокой размерности (такими как изображения). Выборка сразу обучающих образцов и признаков называется методом *случайных участков* (*random patches method*⁷). Сбережение всех обучающих образцов (т.е. `bootstrap=False` и `max_samples=1.0`), но проведение выборки признаков (т.е. `bootstrap_features=True` и/или `max_features` меньше 1.0) называется методом *случайных подпространств* (*random subspaces method*⁸).

Выборка признаков обеспечивает даже большее несходство прогнозаторов, обменивая чуть более высокое смещение на низкую дисперсию.

Случайные леса

Ранее уже упоминалось, что случайный лес (*random forest*⁹) — это ансамбль деревьев принятия решений, которые обычно обучены посредством метода бэггинга (либо иногда вставки), как правило, с параметром `max_samples`, установленным в размер обучающего набора.

⁷ “Ensembles on Random Patches” (“Ансамбли на случайных участках”), Ж. Люпп и П. Гер (2012 год) (<http://goo.gl/B2EcM2>).

⁸ “The random subspace method for constructing decision forests” (“Метод случайных подпространств для построения лесов принятия решений”), Тин Кам Хо (1998 год) (<http://goo.gl/NPi5vH>).

⁹ “Random Decision Forests” (“Случайные леса принятия решений”), Т. Хо (1995 год) (<http://goo.gl/zVOGQ1>).

Вместо построения экземпляра `BaggingClassifier` и его передачи экземпляру `DecisionTreeClassifier` вы можете применить класс `RandomForestClassifier`, который является более удобным и оптимизированным для деревьев принятия решений¹⁰ (аналогичным образом имеется класс `RandomForestRegressor` для задач регрессии). Показанный ниже код обучает классификатор на основе случайного леса с 500 деревьями (каждое ограничено максимум 16 узлами), используя все доступные процессорные ядра:

```
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500,
                               max_leaf_nodes=16, n_jobs=-1)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

С несколькими исключениями класс `RandomForestClassifier` имеет все гиперпараметры класса `DecisionTreeClassifier` (для управления ростом деревьев) плюс все гиперпараметры класса `BaggingClassifier` для управления самим ансамблем¹¹.

Алгоритм случайного леса вводит добавочную случайность, когда выращивает деревья; вместо поиска лучшего из лучших признаков при расщеплении узла (см. главу 6) он ищет наилучший признак в случайном поднаборе признаков. В результате получается более значительное несходство деревьев, которое (снова) обменивает более высокое смещение на низкую дисперсию, как правило, выдавая в целом лучшую модель. Следующий классификатор `BaggingClassifier` примерно эквивалентен предыдущему классификатору `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1)
```

¹⁰ Класс `BaggingClassifier` остается полезным, если вам нужен пакет чего-то, отличающегося от деревьев принятия решений.

¹¹ Существует ряд заметных исключений: отсутствуют гиперпараметры `splitter` (принудительно установлен в `"random"`), `presort` (принудительно установлен в `False`), `max_samples` (принудительно установлен в `1.0`) и `base_estimator` (принудительно установлен в `DecisionTreeClassifier` с предоставленными гиперпараметрами).

Особо случайные деревья

При выращивании дерева в случайном лесе для каждого узла, подлежащего расщеплению, рассматривается только случайный поднабор признаков (как обсуждалось ранее). Можно сделать деревья еще более случайными за счет применения случайных порогов для каждого признака вместо поиска наилучших возможных порогов (подобно тому, как поступают обыкновенные деревья принятия решений).

Лес с такими чрезвычайно случайными деревьями называется просто ансамблем *особо случайных деревьев* (*extremely randomized trees ensemble*¹² или *extra-trees* для краткости). И снова более высокое смещение обменивается на низкую дисперсию. Кроме того, особо случайные деревья обучаются намного быстрее, чем обыкновенные случайные леса, поскольку нахождение наилучшего возможного порога для каждого признака в каждом узле является одной из самых затратных в плане времени задач по выращиванию дерева.

Вы можете создать классификатор на основе особо случайных деревьев, используя класс `ExtraTreesClassifier` из Scikit-Learn. Его API-интерфейс идентичен API-интерфейсу класса `RandomForestClassifier`. Подобным образом класс `ExtraTreesRegressor` имеет такой же API-интерфейс, как у класса `RandomForestRegressor`.



Трудно сказать заранее, будет ли класс `RandomForestClassifier` выполняться лучше или хуже класса `ExtraTreesClassifier`. Как правило, единственный способ узнать — попробовать оба и сравнить их с применением перекрестной проверки (подстраивая гиперпараметры с использованием решетчатого поиска).

Значимость признаков

Еще одно замечательное качество случайных лесов заключается в том, что они облегчают измерение относительной значимости каждого признака. Библиотека Scikit-Learn измеряет значимость признака путем выяснения, насколько узлы дерева, применяющие этот признак, снижают загрязненность в среднем (по всем деревьям в лесе). Выражаясь точнее, значимость признака представляет собой взвешенное среднее, где вес каждого узла равен количеству обучающих образцов, которые с ним ассоциированы (см. главу 6).

¹² “Extremely randomized trees” (“Особо случайные деревья”), П. Гер, Д. Эрнст, Л. Веенкель (2005 год) (<http://goo.gl/RHGEA4>).

Данный показатель подсчитывается в Scikit-Learn автоматически для каждого признака после обучения, а результаты масштабируются так, что сумма всех значимостей равна 1. Вы можете обратиться к итоговому признаку с использованием переменной `feature_importances_`. Например, следующий код обучает классификатор `RandomForestClassifier` на наборе данных `iris` (введенном в главе 4) и выдает значимость каждого признака. Кажется, самыми важными признаками являются длина лепестка в сантиметрах (`petal length (cm)`) (44%) и ширина лепестка в сантиметрах (`petal width (cm)`) (42%), в то время как значимость длины чашелистика в сантиметрах (`sepal length (cm)`) и ширины чашелистика в сантиметрах (`sepal width (cm)`) по сравнению с ними намного ниже (соответственно 11% и 2%).

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"],
                           rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Аналогично, если вы обучите классификатор на основе случайного леса с помощью набора данных MNIST (введенного в главе 3) и вычертите график значимости каждого пикселя, то получите изображение, представленное на рис. 7.6.

Случайные леса очень удобны для быстрого понимания того, какие признаки действительно имеют значение, в особенности, если вам необходимо осуществлять выбор признаков.

Бустинг

Бустинг (первоначально называемый *усилением гипотезы* (*hypothesis boosting*)) относится к любому ансамблевому методу, который способен комбинировать нескольких слабых учеников в одного сильного ученика. Основная идея большинства методов бустинга предусматривает последовательное обучение прогнозаторов, причем каждый из них старается исправить своего предшественника.

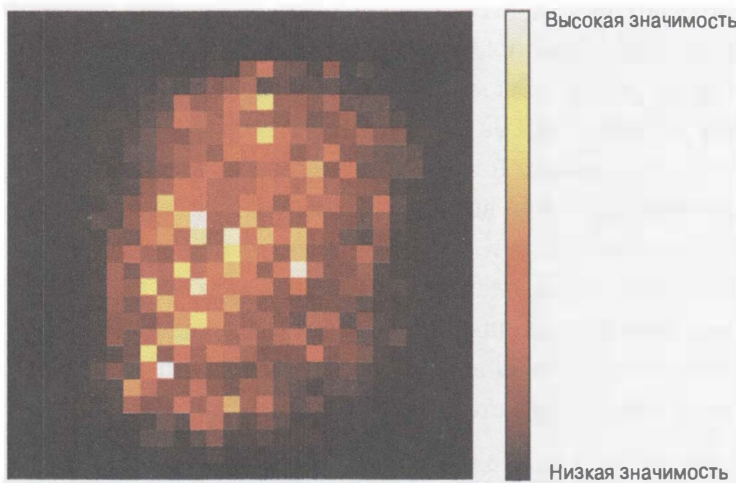


Рис. 7.6. Значимость пикселей в наборе данных MNIST (согласно классификатору на основе случайного леса)

Доступно много методов бустинга, но безоговорочно самыми популярными являются *AdaBoost*¹³ (сокращение от *Adaptive Boosting* — адаптивный бустинг) и *градиентный бустинг* (*gradient boosting*). Давайте начнем с AdaBoost.

AdaBoost

Один из способов, которым новый прогнозатор может исправлять своего предшественника, заключается в том, что он уделяет чуть больше внимания обучающим образцам, на которых у предшественника было недообучение. В результате новые прогнозаторы все больше и больше концентрируются на трудных случаях. Именно такой прием применяет метод AdaBoost.

Например, при построении классификатора AdaBoost сначала обучается первый базовый классификатор (такой как дерево принятия решений), который используется для выработки прогнозов на обучающем наборе. Относительный вес некорректно классифицированных им обучающих образцов увеличивается. Второй классификатор обучается уже с применением обновленных весов, после чего он используется для выработки прогнозов на обучающем наборе, веса снова обновляются и т.д. (рис. 7.7).

¹³ “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting” (“Обобщение динамического обучения с точки зрения теории принятия решений и применение к бустингу”), Йоав Фройнд, Роберт Шапайр (1997 год) (<http://goo.gl/OIduRW>).

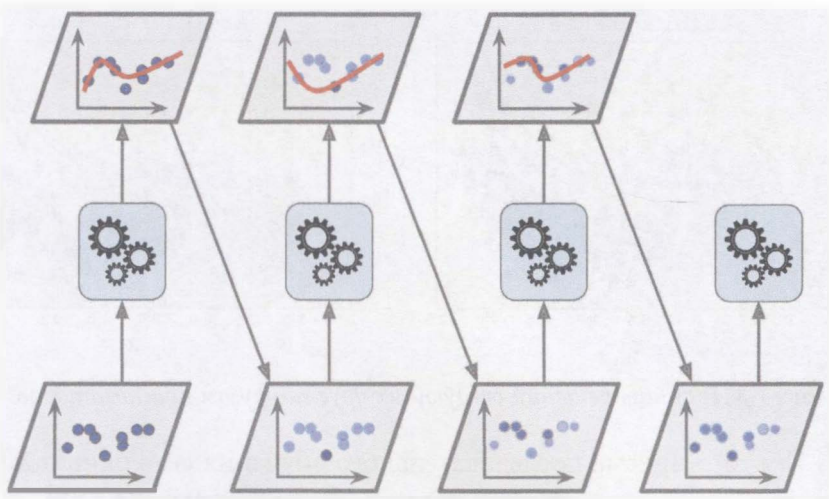


Рис. 7.7. Последовательное обучение в методе AdaBoost с обновлением весов образцов

На рис. 7.8 показаны границы решений пяти последовательных прогнозаторов на наборе данных `moons` (в этом примере каждый прогнозатор является сильно регуляризируемым классификатором SVM с ядром RBF¹⁴). Первый классификатор воспринимает многие образцы неправильно, поэтому их веса повышаются. Вследствие этого второй классификатор справляется с такими образцами лучше и т.д. На графике справа представлена та же самая последовательность прогнозаторов, но скорость обучения (`learning_rate`) уменьшена вдвое (т.е. на каждой итерации веса некорректно классифицированных образцов поднимаются максимум наполовину). Как видите, такой прием последовательного обучения имеет некоторые сходные черты с градиентным спуском, но вместо подстройки параметров одиночного прогнозатора для сведения к минимуму функции издержек AdaBoost добавляет прогнозаторы в ансамбль, постепенно делая его лучше.

После того как все прогнозаторы обучены, ансамбль вырабатывает прогнозы очень похоже на бэггинг или вставку за исключением того, что прогнозаторы имеют разные веса в зависимости от их общей правильности на взвешенном обучающем наборе.

¹⁴ Только в целях иллюстрации. Методы SVM в целом не являются хорошими прогнозаторами для AdaBoost, потому что они медленные и склонны к нестабильной работе с AdaBoost.

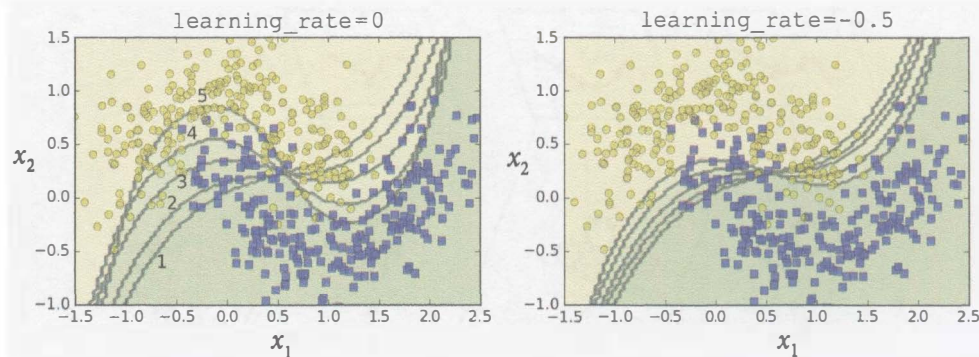


Рис. 7.8. Границы решений следующих друг за другом прогнозаторов



У этого приема последовательного обучения есть один важный недостаток: он не допускает распараллеливания (или только частично), поскольку каждый прогнозатор можно обучать лишь после того, как был обучен и оценен предыдущий прогнозатор. В результате он не масштабируется настолько хорошо, как бэггинг или вставка.

Давайте подробнее рассмотрим алгоритм AdaBoost. Вес каждого образца $w^{(i)}$ изначально установлен в $\frac{1}{m}$. Первый прогнозатор обучается и подсчитывается его взвешенная частота ошибок r_1 на обучающем наборе (уравнение 7.1).

Уравнение 7.1. Взвешенная частота ошибок j -того прогнозатора

$$r_j = \frac{\sum_{i=1}^m w^{(i)} \mathbb{1}_{\hat{y}_j^{(i)} \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}} ,$$

где $\hat{y}_j^{(i)}$ — прогноз j -того прогнозатора для i -того образца.

Затем вычисляется вес α_j прогнозатора с применением уравнения 7.2, где η — гиперпараметр скорости обучения (со стандартным значением 1)¹⁵. Чем более правильным является прогнозатор, тем выше будет его вес. Если прогнозатор всего лишь случайно угадывает, тогда его вес будет близок к нулю. Однако если он почти всегда ошибается (т.е. правильность ниже случайного угадывания), то его вес будет отрицательным.

¹⁵ В исходном алгоритме AdaBoost гиперпараметр скорости обучения не используется.

Уравнение 7.2. Вес прогнозатора

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Далее веса образцов обновляются с применением уравнения 7.3: некорректно классифицированные образцы поднимаются.

Уравнение 7.3. Правило обновления весов

для $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)}, & \text{если } \hat{y}_j^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j), & \text{если } \hat{y}_j^{(i)} \neq y^{(i)} \end{cases}$$

После этого веса образцов нормализуются (т.е. делятся на $\sum_{i=1}^m w^{(i)}$).

Наконец, новый прогнозатор обучается с использованием обновленных весов, после чего весь процесс повторяется (вычисление веса нового прогнозатора, обновление весов образцов, обучение еще одного прогнозатора и т.д.). Алгоритм останавливается, когда достигнуто желаемое количество прогнозаторов или найден совершенный прогнозатор.

При прогнозировании алгоритм AdaBoost просто подсчитывает прогнозы всех прогнозаторов и взвешивает их с применением весов прогнозаторов α_j . Спрогнозированным классом будет тот, который получает большинство взвешенных голосов (уравнение 7.4).

Уравнение 7.4. Прогнозы AdaBoost

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j, \quad \hat{y}_j(\mathbf{x}) = k$$

где N — количество прогнозаторов.

В действительности Scikit-Learn использует многоклассовую версию алгоритма AdaBoost, называемую SAMME¹⁶ (что означает *Stagewise Additive Modeling using a Multiclass Exponential loss function* — ступенчатое аддитивное моделирование с применением многоклассовой экспоненциальной функции потерь). Когда классов только два, алгоритм SAMME эквивалентен AdaBoost.

¹⁶ За дополнительными сведениями обращайтесь в статью “Multi-Class AdaBoost” (“Многоклассовый алгоритм AdaBoost”), Джи Жу и др. (2006 год) (<http://goo.gl/Eji2vR>).

Кроме того, если прогнозаторы в состоянии оценивать вероятности классов (т.е. имеют метод `predict_proba()`), то Scikit-Learn может использовать вариант SAMME под названием SAMME.R (здесь R означает Real — вещественный), который полагается на вероятности классов, а не на прогнозы, и в целом выполняется лучше.

Приведенный ниже код обучает классификатор AdaBoost, основанный на 200 *пеньках решений* (*Decision Stump*), с применением класса `AdaBoostClassifier` из Scikit-Learn (как вы могли догадаться, имеется также класс `AdaBoostRegressor`). Пенек решения представляет собой дерево принятия решений с `max_depth=1` — иными словами, дерево, состоящее из одного узла решения и двух листовых узлов. Это стандартный базовый оценщик для класса `AdaBoostClassifier`.

```
from sklearn.ensemble import AdaBoostClassifier
ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```



Если ваш ансамбль AdaBoost переобучается обучающим набором, тогда можете сократить количество оценщиков или более строго регуляризовать базовый оценщик.

Градиентный бустинг

Другим популярным алгоритмом бустинга является *градиентный бустинг*¹⁷. Подобно AdaBoost градиентный бустинг работает, последовательно добавляя в ансамбль прогнозаторы, каждый из которых корректирует своего предшественника. Тем не менее, вместо подстройки весов образцов на каждой итерации, как делает AdaBoost, этот метод старается подогнать новый прогнозатор к *остаточным ошибкам* (*residual error*), допущенным предыдущим прогнозатором.

Давайте рассмотрим простой пример регрессии, использующий деревья принятия решений в качестве базовых прогнозаторов (разумеется, градиентный бустинг прекрасно работает также и с задачами регрессии). Это называется *градиентным бустингом на основе деревьев* (*gradient tree boosting*) или *деревьями*

¹⁷ Впервые был представлен в работе “Arcing the Edge” (“Образование дуги на краю”), Л. Брейман (1997 год) (<http://goo.gl/Ezw4jL>).

регрессии с градиентным бустингом (*gradient boosted regression tree* — GBRT). Первым делом подгоним регрессор `DecisionTreeRegressor` к обучающему набору (например, зашумленному квадратичному обучающему набору):

```
from sklearn.tree import DecisionTreeRegressor
tree_reg1 = DecisionTreeRegressor(max_depth=2)
tree_reg1.fit(X, y)
```

Теперь обучим второй регрессор `DecisionTreeRegressor` на остаточных ошибках, допущенных первым прогнозатором:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2)
tree_reg2.fit(X, y2)
```

Затем обучим третий регрессор на остаточных ошибках, допущенных вторым прогнозатором:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2)
tree_reg3.fit(X, y3)
```

Мы получили ансамбль, содержащий три дерева. Он может вырабатывать прогнозы на новом образце, просто суммируя прогнозы всех деревьев:

```
y_pred = sum(tree.predict(X_new)
              for tree in (tree_reg1, tree_reg2, tree_reg3))
```

На рис. 7.9 представлены прогнозы этих трех деревьев в левой колонке и прогнозы ансамбля в правой колонке. В первой строке ансамбль имел только одно дерево, потому его прогнозы в точности такие же, как у первого дерева. Во второй строке новое дерево обучено на остаточных ошибках первого дерева. Справа видно, что прогнозы ансамбля равны сумме прогнозов первых двух деревьев. Аналогично в третьей строке еще одно дерево обучено на остаточных ошибках второго дерева. Легко заметить, что по мере добавления деревьев к ансамблю его прогнозы постепенно становятся лучше.

Более простой способ обучения ансамблей GBRT предусматривает применение класса `GradientBoostingRegressor` из Scikit-Learn. Во многом подобно классу `RandomForestRegressor` он имеет гиперпараметры для управления ростом деревьев принятия решений (например, `max_depth`, `min_samples_leaf` и т.д.), а также гиперпараметры для управления обучением ансамбля вроде количества деревьев (`n_estimators`).

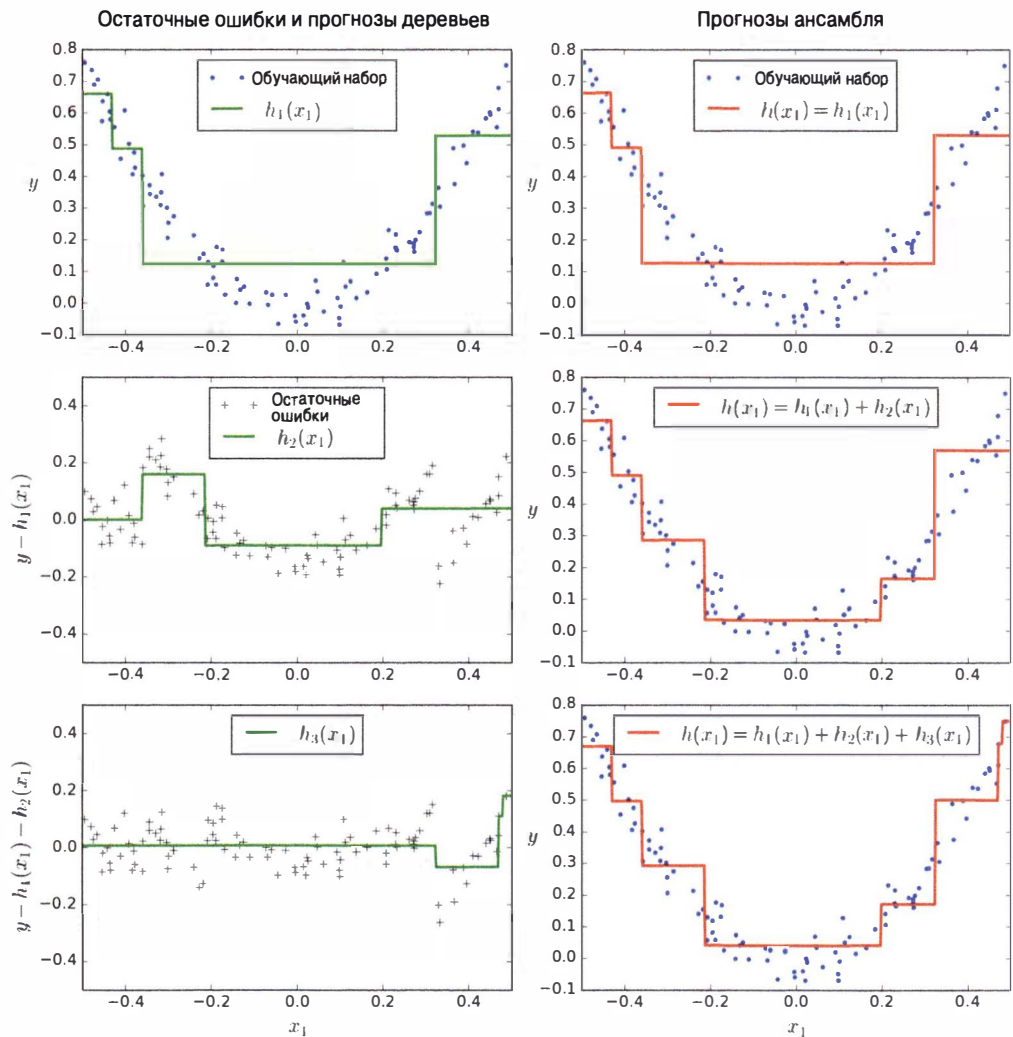


Рис. 7.9. Градиентный бустинг

Следующий код создает тот же самый ансамбль, что и предыдущий код:

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                  learning_rate=1.0)
gbrt.fit(X, y)
```

Гиперпараметр скорости обучения (`learning_rate`) устанавливает степень вклада каждого дерева. Если вы установите его в низкое значение, такое как `0.1`, тогда придется подгонять к обучающему набору больше деревьев в ансамбле, но прогнозы обычно будут обобщаться лучше. Это прием регуля-

ризации, называемый *сжатием* (*shrinkage*). На рис. 7.10 показаны два ансамбля GBRT, обученные с низкой скоростью обучения: ансамбль слева не имеет достаточного числа деревьев, чтобы подогнаться к обучающему набору, в то время как ансамбль справа имеет слишком много деревьев и переобучается обучающим набором.

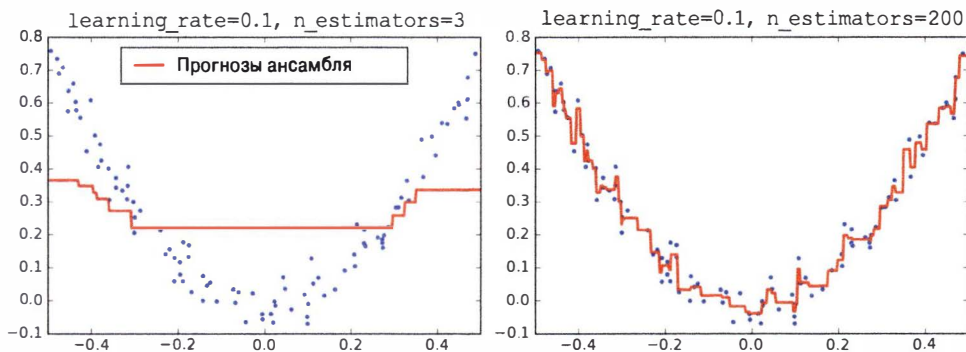


Рис. 7.10. Ансамбли GBRT, в которых недостаточно (слева) и слишком много (справа) прогнозаторов

Чтобы отыскать оптимальное количество деревьев, вы можете использовать раннее прекращение (см. главу 4). Его несложно реализовать с применением метода `staged_predict()`: он возвращает итератор по прогнозам, выработанным ансамблем на каждой стадии обучения (с одним деревом, двумя деревьями и т.д.). Приведенный ниже код обучает ансамбль GBRT, содержащий 120 деревьев, измеряет ошибку проверки на каждой стадии обучения для нахождения оптимального числа деревьев и в заключение обучает еще один ансамбль GBRT, используя оптимальное количество деревьев:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=120)
gbrt.fit(X_train, y_train)

errors = [mean_squared_error(y_val, y_pred)
           for y_pred in gbrt.staged_predict(X_val)]
bst_n_estimators = np.argmin(errors)

gbrt_best = GradientBoostingRegressor(max_depth=2,
                                       n_estimators=bst_n_estimators)
gbrt_best.fit(X_train, y_train)
```

На рис. 7.11 слева представлены ошибки проверки, а справа — прогнозы наилучшей модели.

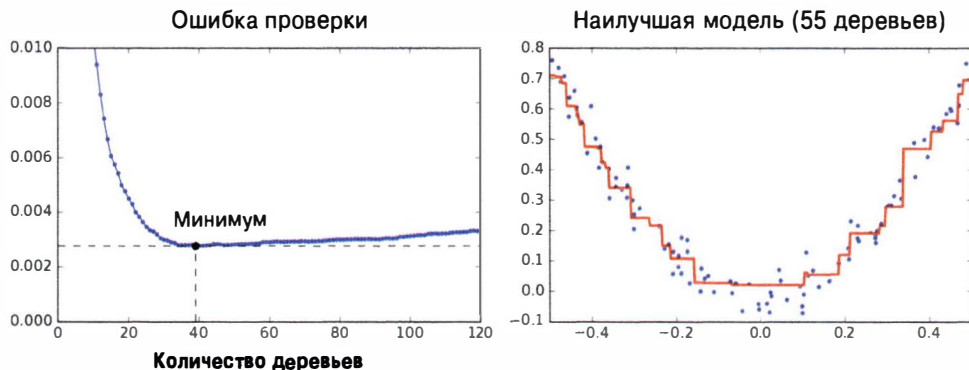


Рис. 7.11. Подстройка количества деревьев с применением раннего прекращения

Раннее прекращение также можно реализовать, фактически останавливая обучение досрочно (вместо того, чтобы сначала обучить крупное число деревьев и затем обернуться назад в поисках оптимального количества). Для этого понадобится установить `warm_start=True`, что заставляет библиотеку Scikit-Learn сохранять существующие деревья, когда вызывается метод `fit()`, делая возможным постепенное обучение. Следующий код останавливает обучение, когда ошибка проверки не улучшается для пяти итераций в строке:

```
gbrt = GradientBoostingRegressor(max_depth=2, warm_start=True)
min_val_error = float("inf")
error_going_up = 0
for n_estimators in range(1, 120):
    gbrt.n_estimators = n_estimators
    gbrt.fit(X_train, y_train)
    y_pred = gbrt.predict(X_val)
    val_error = mean_squared_error(y_val, y_pred)
    if val_error < min_val_error:
        min_val_error = val_error
        error_going_up = 0
    else:
        error_going_up += 1
        if error_going_up == 5:
            break # раннее прекращение
```

Класс `GradientBoostingRegressor` также поддерживает гиперпараметр `subsample`, который указывает долю обучающих образцов, под-

лежащих использованию для обучения каждого дерева. Например, если `subsample=0.25`, тогда каждое дерево обучается на 25% обучающих образцов, выбранных произвольно. Как вы, вероятно, уже догадались, это обменивает более высокое смещение на низкую дисперсию. Вдобавок обучение значительно ускоряется. Такой прием называется *стохастическим градиентным бустингом* (*stochastic gradient boosting*).



Градиентный бустинг можно применять с другими функциями потерь, что управляется гиперпараметром `loss` (за дополнительными сведениями обращайтесь в документацию Scikit-Learn).

Стекинг

Последний ансамблевый метод, который мы обсудим в главе, называется *стекингом* (“stacking” — сокращенное название “stacked generalization”¹⁸ (стековое обобщение)). Он основан на простой идее: вместо того, чтобы использовать тривиальные функции (такие как с жестким голосованием) для агрегирования прогнозов всех прогнозаторов в ансамбле, почему бы нам не научить какую-то модель делать это агрегирование? На рис. 7.12 демонстрируется ансамбль такого рода, выполняющий задачу регрессии на новом образце.

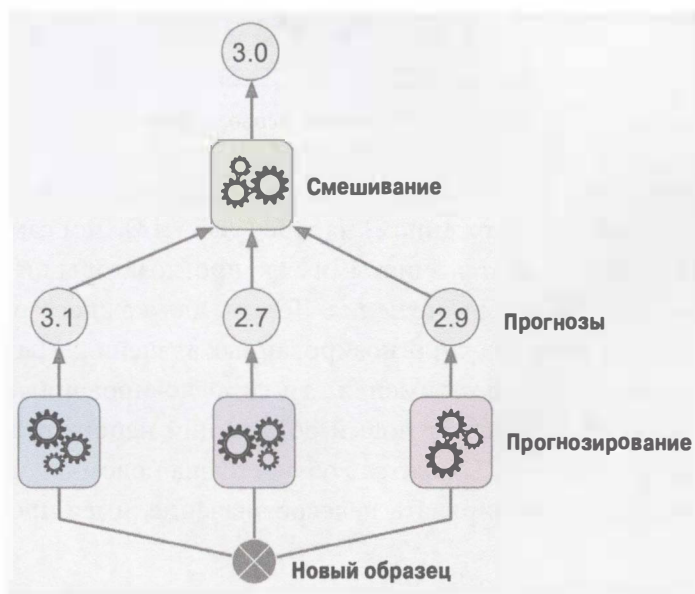


Рис. 7.12. Агрегирование прогнозов с применением смешивающего прогнозатора

¹⁸ “Stacked Generalization” (“Стековое обобщение”), Д. Уолперт (1992 год).

Каждый из трех нижних прогнозаторов прогнозирует отличающееся значение (3.1, 2.7 и 2.9), после чего финальный прогнозатор (называемый *смесителем* (*blender*) или *мета-учеником* (*meta learner*)) получает на входе такие прогнозы и вырабатывает окончательный прогноз (3.0).

Распространенный подход к обучению смесителя предполагает использование удерживаемого (hold-out) набора¹⁹. Давайте посмотрим, как это работает. Для начала обучающий набор разбивается на два поднабора. Первый поднабор применяется для обучения прогнозаторов на первом уровне (рис. 7.13).

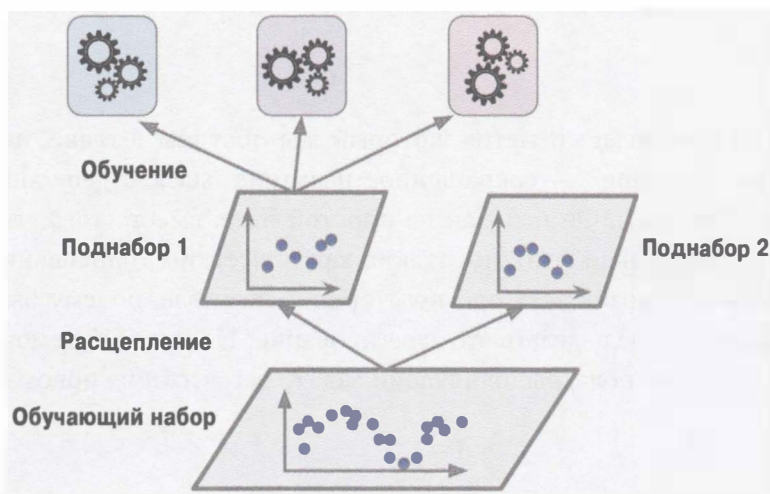


Рис. 7.13. Обучение первого уровня

Затем прогнозаторы первого уровня используются для вырабатывания прогнозов на втором (удержанном) наборе (рис. 7.14). Тем самым гарантируется, что прогнозы являются “чистыми”, т.к. прогнозаторы ни разу не видели данных образцов во время обучения. Теперь для каждого образца в удерживаемом наборе есть три спрогнозированных значения. Мы можем создать новый обучающий набор, применяя эти спрогнозированные значения как входные признаки (что делает новый обучающий набор трехмерным) и сохраняя целевые значения. Смеситель обучается на новом обучающем наборе, а потому учится прогнозировать целевое значение, имея прогнозы первого уровня.

¹⁹ В качестве альтернативы можно использовать внеблочные (out-of-fold) прогнозы. В некоторых контекстах это называется *стекингом*, тогда как применение удерживаемого набора — *смешиванием*. Однако для многих людей упомянутые термины синонимичны.

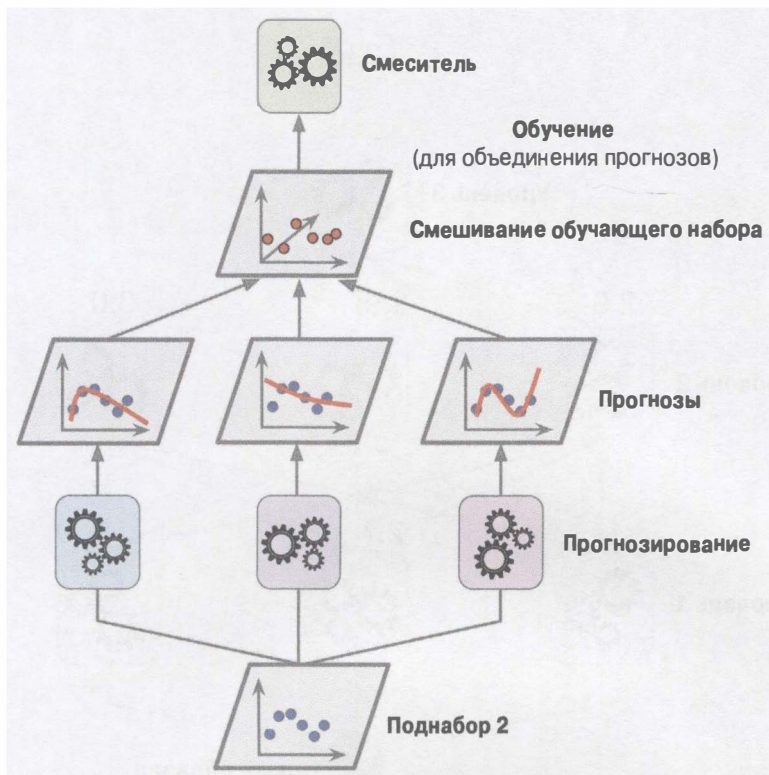


Рис. 7.14. Обучение смесителя

На самом деле подобным образом можно обучить несколько разных смесителей (например, смеситель, использующий линейную регрессию, еще один смеситель, применяющий регрессию на основе случайного леса, и т.д.): мы получаем целый уровень смесителей. Трюк предусматривает расщепление обучающего набора на три поднабора. Первый поднабор используется для обучения первого уровня. Второй поднабор предназначен для создания обучающего набора, который применяется при обучении второго уровня (и использует прогнозы, выработанные прогнозаторами первого уровня). Третий поднабор позволяет создать обучающий набор для обучения третьего уровня (и применяет прогнозы, сделанные прогнозаторами второго уровня). После этого мы можем вырабатывать прогноз для нового образца, последовательно проходя по всем уровням (рис. 7.15).

К сожалению, Scikit-Learn не поддерживает стекинг напрямую, но не слишком сложно развернуть собственную реализацию (см. упражнения ниже). В качестве альтернативы вы можете использовать реализацию с открытым кодом, такую как `brew` (доступную по адресу <https://github.com/viisar/brew>).

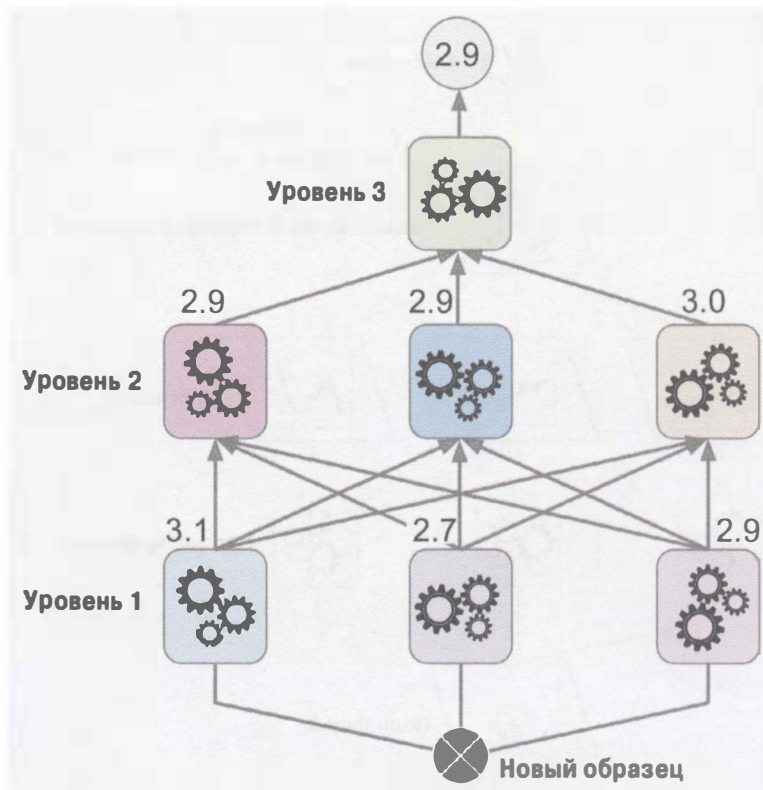


Рис. 7.15. Прогнозы в многоуровневом ансамбле со стекингом

Упражнения

1. Если вы обучили пять разных моделей на точно тех же обучающих данных, и все они достигают точности 95%, то можно ли как-нибудь скомбинировать эти модели, чтобы получить лучшие результаты? Если да, то как? Если нет, то почему?
2. В чем разница между классификаторами с жестким и с мягким голосованием?
3. Можно ли ускорить обучение ансамбля с бэггингом, распределив его по множеству серверов? Как насчет ансамблей с вставкой, ансамблей с бустингом, случайных лесов или ансамблей со стекингом?
4. В чем преимущество оценки с помощью неиспользуемых образцов (*oob*)?

5. Что делает особо случайные деревья (Extra-Trees) в большей степени случайными, чем обыкновенные случайные леса? Чем может помочь такая добавочная случайность? Особо случайные деревья медленнее или быстрее обыкновенных случайных лесов?
6. Если ваш ансамбль AdaBoost недообучается на обучающих данных, то какие гиперпараметры вы должны подстраивать и каким образом?
7. Если ваш ансамбль с градиентным бустингом переобучается обучающим набором, то вы должны увеличить или уменьшить скорость обучения?
8. Загрузите данные MNIST (представленные в главе 3) и расщепите их на обучающий набор, проверочный набор и испытательный набор (например, применив 40 000 образцов для обучения, 10 000 — для проверки и 10 000 — для испытания). Затем обучите разнообразные классификаторы, такие как классификатор на основе случайного леса, классификатор на базе особо случайных деревьев и классификатор SVM. Далее попробуйте объединить их в ансамбль, который превосходит их всех на проверочном наборе, используя классификатор с мягким или с жестким голосованием. После получения ансамбля опробуйте его на испытательном наборе. Насколько лучше он выполняется в сравнении с индивидуальными классификаторами?
9. Запустите индивидуальные классификаторы из предыдущего упражнения, чтобы выработать прогнозы на проверочном наборе, и создайте новый обучающий набор с результирующими прогнозами: каждый обучающий образец представляет собой вектор, содержащий набор прогнозов от всех классификаторов для изображения, и целью является класс изображения. Примите поздравления, вы только что обучили смеситель, который вместе с классификаторами образует ансамбль со стекингом! Теперь оцените ансамбль на испытательном наборе. Для каждого изображения в испытательном наборе выработайте прогнозы с помощью всех имеющихся классификаторов, после чего передайте эти прогнозы смесителю, чтобы получить прогнозы ансамбля. Как результаты соотносятся с обученным ранее классификатором с голосованием?

Решения приведенных упражнений доступны в приложении А.

Понижение размерности

Многие задачи машинного обучения имеют дело с тысячами или даже миллионами признаков для каждого обучающего образца. Как вы увидите далее, это не только крайне замедляет обучение, но и значительно затрудняет нахождение хорошего решения. На такую проблему часто ссылаются как на “*проклятие размерности*” (*curse of dimensionality*).

К счастью, в реальных задачах количество признаков нередко можно существенно сократить, переведя трудноразрешимую задачу в разрешимую. Например, рассмотрим изображения MNIST (представленные в главе 3): пиксели на краях изображений почти всегда белые, так что вы могли бы полностью отбросить такие пиксели из обучающего набора, не теряя много информации. На рис. 7.6 подтверждается, что эти пиксели совсем не важны для задачи классификации. Кроме того, два смежных пикселя часто сильно связаны: если вы сольете их в один пиксель (скажем, взяв среднее интенсивностей двух пикселей), то не потеряете много информации.



Понижение размерности на самом деле вызывает утрату некоторой информации (подобно тому, как сжатие изображения в формат JPEG может ухудшить его качество). И хотя понижение размерности ускорит обучение, оно может привести к тому, что система станет выполняться чуть хуже. Оно также немного усложнит конвейеры, сделав их труднее в сопровождении. Следовательно, вы сначала должны попробовать обучить свою систему с применением исходных данных, прежде чем обдумывать использование понижения размерности, когда обучение оказывается слишком медленным. Однако в ряде случаев понижение размерности обучающих данных может отфильтровать некоторый шум и ненужные детали, обеспечив более высокую производительность (но в общем случае это не так; оно просто ускоряет обучение).

Вдобавок к ускорению обучения понижение размерности также чрезвычайно полезно при визуализации данных. Уменьшение количества измерений до двух (или трех) делает возможным графическое представление обучающих наборов с высоким числом измерений и часто появление ряда важных догадок за счет зрительного обнаружения закономерностей, таких как скопления.

В настоящей главе мы обсудим “проклятие размерности” и дадим представление о том, что происходит в многомерном пространстве. Затем мы рассмотрим два основных подхода к понижению размерности (*проекция* (*projection*) и *обучение на основе многообразий* (*manifold learning*)) и пройдемся по трем самым популярным приемам понижения размерности: PCA, Kernel PCA и LLE.

“Проклятие размерности”

Мы настолько привыкли жить в трех измерениях¹, что наша интуиция не справляется с попытками вообразить многомерное пространство. Даже базовый четырехмерный гиперкуб невероятно трудно изобразить в нашем уме (рис. 8.1), не говоря уже о 200-мерном эллипсоиде, привязанном к 1000-мерному пространству.

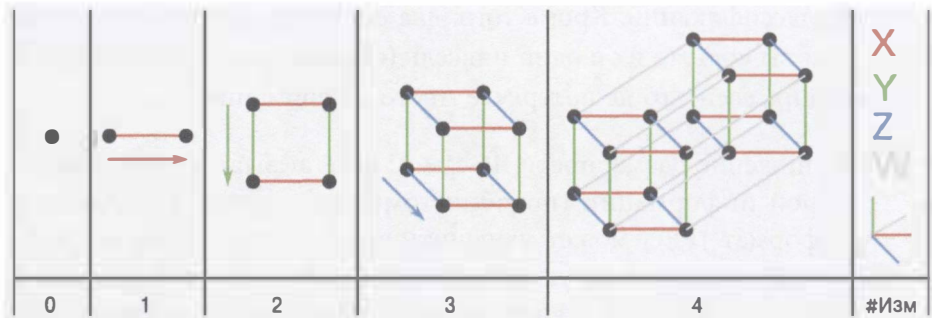


Рис. 8.1. Точка, отрезок, квадрат, куб и тессеракт (от нулевой размерности до четырехмерного гиперкуба)²

- ¹ Хорошо, в четырех измерениях, если учесть время, и в еще большем количестве, если вы занимаетесь теорией струн.
- ² Понаблюдайте за вращением тессеракта, спроецированного на трехмерное пространство: <http://goo.gl/OM7ktJ>. Изображение принадлежит пользователю Википедии NerdBoy1392 (Creative Commons BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)). Воспроизведено из <https://en.wikipedia.org/wiki/Tesseract>.

Оказывается, что в многомерном пространстве многие вещи ведут себя совсем иначе. Например, если вы выбираете произвольную точку внутри единичного квадрата (квадрата 1×1), то она имеет только приблизительно 0.4%-ный шанс находиться ближе, чем 0.001 от края (другими словами, крайне маловероятно, что произвольная точка окажется “предельной” по любому измерению). Но в случае 10 000-мерного единичного гиперкуба (куба $1 \times 1 \times \dots \times 1$, т.е. в сумме десять тысяч единиц) такая вероятность превышает 99.999999%. Большинство точек внутри многомерного гиперкуба очень близки к краю³.

Существует отличие, причиняющее больше беспокойства: если вы произвольно выбираете две точки внутри единичного квадрата, то расстояние между ними будет составлять в среднем приблизительно 0.52. Если же вы выбираете две произвольные точки внутри единичного трехмерного куба, тогда средним расстоянием будет примерно 0.66. Но что можно сказать о двух точках, произвольно выбранных в единичном 1 000 000-мерном гиперкубе? Верите или нет, среднее расстояние составит около 408.25 (приблизительно $\sqrt{1\,000\,000/6}$)! Это совершенно парадоксально: как две точки могут оказаться настолько далеко друг от друга, когда они обе находятся внутри того же единичного гиперкуба? Из указанного факта следует, что многомерные наборы данных подвержены риску быть крайне разреженными: большинство обучающих образцов, скорее всего, будут находиться далеко друг от друга. Разумеется, это также означает, что новый образец, вероятно, окажется далеко от любого обучающего образца, делая прогнозы гораздо менее надежными, чем при меньшем количестве измерений, поскольку они будут основаны на значительно большем числе экстраполяций. Короче говоря, чем больше измерений имеет обучающий набор, тем выше риск переобучения им.

Теоретически одним из решений “проклятия размерности” могло быть увеличение размера обучающего набора с целью достижения достаточной плотности обучающих образцов. К сожалению, на практике количество обучающих образцов, требуемых для достижения заданной плотности, растет экспоненциально в отношении числа измерений. Имея лишь 100 признаков (значительно меньше, нежели в задаче MNIST), вам понадобилось бы больше обучающих образцов, чем атомов в наблюдаемой вселенной, чтобы обучающие образцы находились на расстоянии 0.1 друг от друга в среднем при условии их равномерного распределения по всем измерениям.

³ Забавный факт: любой, кого вы знаете, вероятно, занимает крайнюю позицию минимум в одном измерении (скажем, по количеству сахара, добавляемого в кофе), если вы учтете достаточное число измерений.

Основные подходы к понижению размерности

Прежде чем погружаться в исследование специфических алгоритмов понижения размерности, давайте рассмотрим два основных подхода к понижению размерности: проекция и обучение на основе многообразий.

Проекция

В большинстве реальных задач обучающие образцы *не* распределены равномерно по всем измерениям. Многие признаки являются почти постоянными, в то время как другие — тесно зависимыми (что обсуждалось ранее для MNIST). В результате все обучающие образцы фактически находятся внутри *подпространства* (или близко к нему) с гораздо меньшим числом измерений в рамках многомерного пространства. Звучит очень абстрактно, поэтому стоит взглянуть на пример. На рис. 8.2 показан трехмерный набор данных, представленный кружочками.

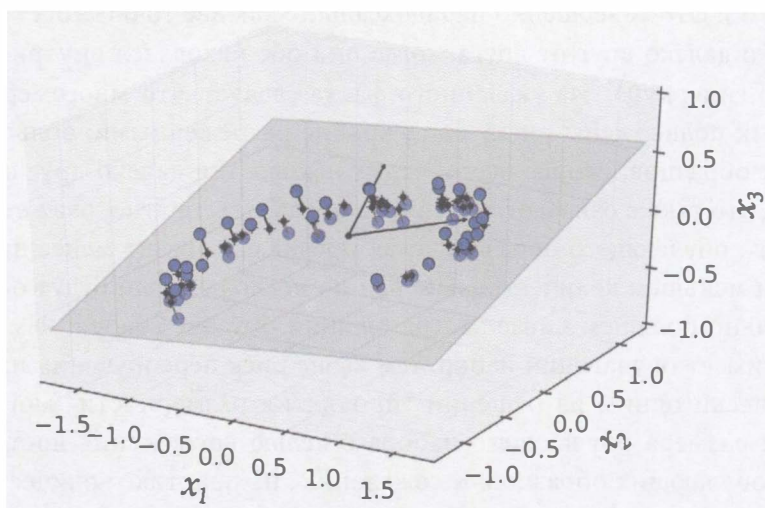


Рис. 8.2. Трехмерный набор данных, лежащий близко к двумерному подпространству

Обратите внимание, что все обучающие образцы расположены близко к некоторой плоскости: это подпространство с меньшим количеством измерений (двумерное) в рамках многомерного (трехмерного) пространства. Если теперь мы спроецируем каждый обучающий образец перпендикулярно на такое подпространство (представлено короткими линиями, соединяющими образцы с плоскостью), то получим новый двумерный набор данных, при-

веденный на рис. 8.3. Та-дам! Мы только что понизили размерность набора данных от 3-х до 2-х. Имейте в виду, что оси соответствуют новым признакам z_1 и z_2 (координаты проекций на плоскость).

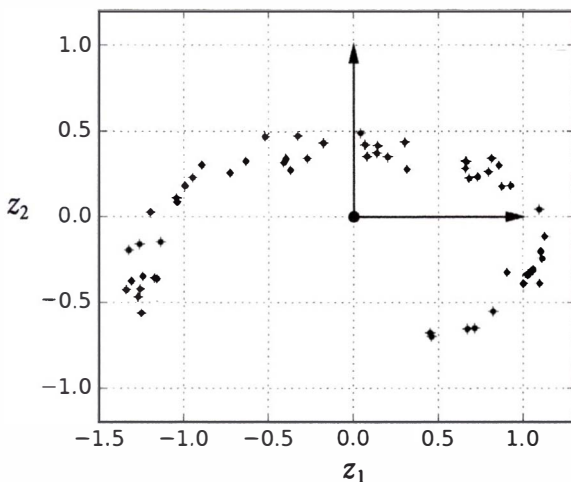


Рис. 8.3. Новый двумерный набор данных после проекции

Тем не менее, проекция не всегда оказывается наилучшим подходом к понижению размерности. Во многих случаях подпространство может скручиваться и поворачиваться, как в известном наборе данных *Swiss roll* (швейцарский рулет), показанном на рис. 8.4.

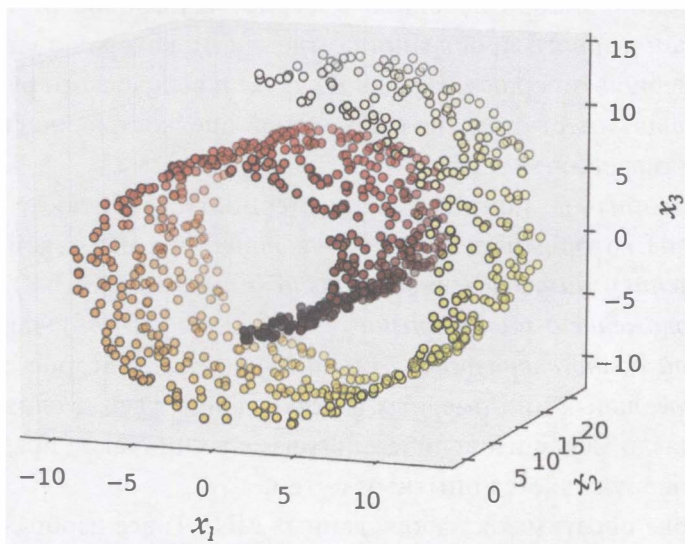


Рис. 8.4. Набор данных *Swiss roll*

Простое проектирование на плоскость (например, путем отбрасывания x_3) сплющило бы разные уровни швейцарского рулета, как видно слева на рис. 8.5. Однако на самом деле мы хотим развернуть швейцарский рулет, чтобы получить двумерный набор данных, показанный справа на рис. 8.5.

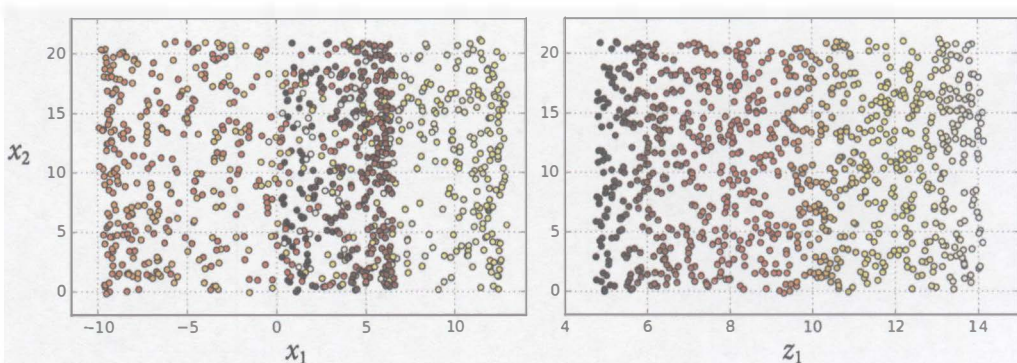


Рис. 8.5. Сплющивание швейцарского рулета за счет проектирования на плоскость (слева) в противоположность его разворачиванию (справа)

Обучение на основе многообразий

Швейцарский рулет является примером двумерного *многообразия* (*manifold*). Выражаясь просто, двумерное многообразие — это двумерная форма, которую можно изгибать и скручивать в пространстве с большим числом измерений. Говоря в общем, d -мерное многообразие представляет собой часть n -мерного пространства (где $d < n$), которая локально имеет сходство с d -мерной гиперплоскостью. В случае швейцарского рулета $d = 2$ и $n = 3$: он локально имеет сходство с двумерной плоскостью, но сворачивается в третьем измерении.

Многие алгоритмы понижения размерности работают, моделируя *многообразие*, на котором находятся обучающие образцы; такой подход называется *обучением на основе многообразий* (*manifold learning*). Он опирается на *предположение о многообразии* (*manifold assumption*), также называемое *гипотезой о многообразии* (*manifold hypothesis*), которое считает, что большинство реальных многомерных наборов данных лежат близко к многообразию с гораздо меньшим количеством измерений. Такое предположение очень часто обнаруживается опытным путем.

Давайте снова обратимся к набору данных MNIST: все изображения рукописных цифр имеют некоторые сходные черты. Они образованы из соедине-

тельных линий, кромки являются белыми, они более или менее центрированы и т.д. Если вы сгенерировали изображения случайным образом, то лишь смехотворно малая их часть будет выглядеть похожими на рукописные цифры. Другими словами, степень свободы, доступная вам при создании изображения цифры, значительно ниже степени свободы, которая имеется, когда разрешено генерировать любое желаемое изображение. Такие ограничения стремятся сжать набор данных в многообразии с меньшим количеством измерений.

Предположение о многообразии часто сопровождается другим неявным предположением: что решаемая задача (скажем, классификации или регрессии) будет проще, если ее выразить в пространстве многообразия с меньшим числом измерений. Например, в верхней строке на рис. 8.6 набор данных Swiss roll расщепляется на два класса: в трехмерном пространстве (слева) граница решений была бы довольно сложной, но в двумерном пространстве многообразия (справа) граница решений представляет собой простую прямую линию.

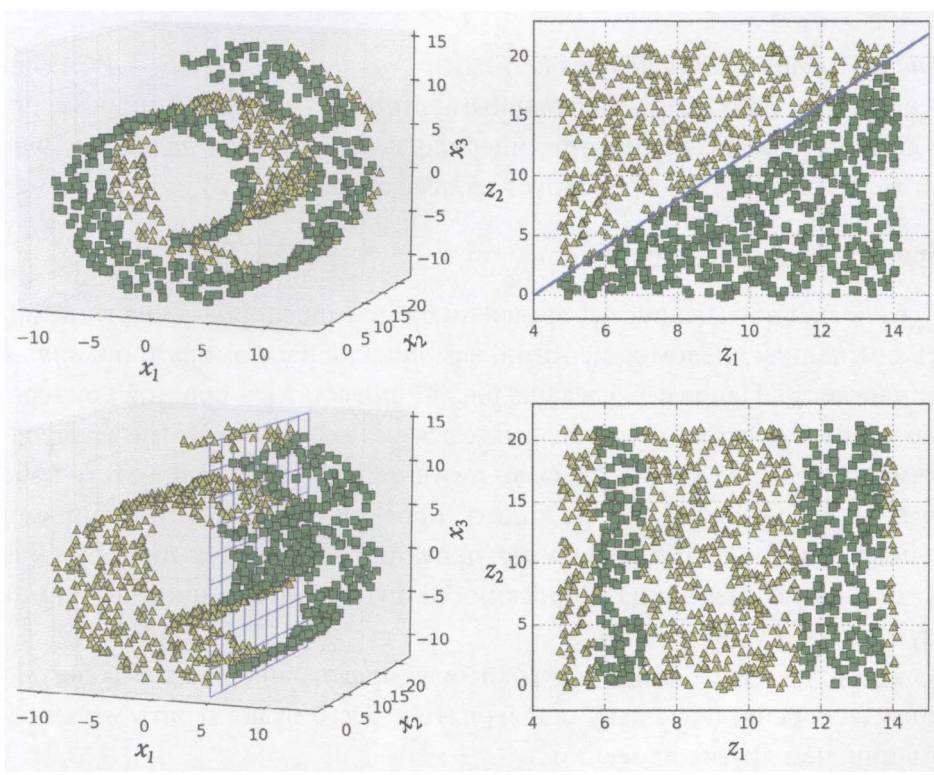


Рис. 8.6. Граница решений не всегда может упрощаться при меньшем количестве измерений

Тем не менее, указанное предположение не всегда поддерживается. Скажем, в нижней строке на рис. 8.6 граница решений расположена в месте $x_1 = 5$. Такая граница решений выглядит очень простой в исходном трехмерном пространстве (вертикальная плоскость), но более сложной в развернутом многообразии (совокупность из четырех независимых линейных сегментов).

Короче говоря, если перед обучением модели вы понижаете размерность своего обучающего набора, то это определенно ускорит обучение, но не всегда может приводить к лучшему или более простому решению; все зависит от набора данных.

К настоящему времени вы должны иметь хорошее представление о том, что такое “проклятие размерности” и как алгоритмы понижения размерности могут с ним справиться, особенно в случае поддержки предположения о многообразии. Остаток главы посвящен исследованию ряда самых популярных алгоритмов.

РСА

Анализ главных компонент (Principal Component Analysis — PCA) является безоговорочно самым популярным алгоритмом понижения размерности. Сначала он идентифицирует гиперплоскость, которая находится ближе всего к данным, и затем проецирует на нее данные.

Предохранение дисперсии

Прежде чем обучающий набор можно будет спроецировать на гиперплоскость с меньшим числом измерений, вам понадобится выбрать правильную гиперплоскость. Например, слева на рис. 8.7 представлен простой двумерный набор данных вместе с тремя разными осями (т.е. одномерными гиперплоскостями). Справа на рис. 8.7 показан результат проецирования этого набора данных на каждую из осей. Как видите, проекция на сплошную линию предохраняет максимальную дисперсию, проекция на точечную линию — очень незначительную дисперсию и проекция на пунктирную линию — промежуточную величину дисперсии.

Кажется разумным выбрать ось, которая предохраняет максимальную величину дисперсии, поскольку она вероятнее всего будет терять меньше информации, чем другие проекции.

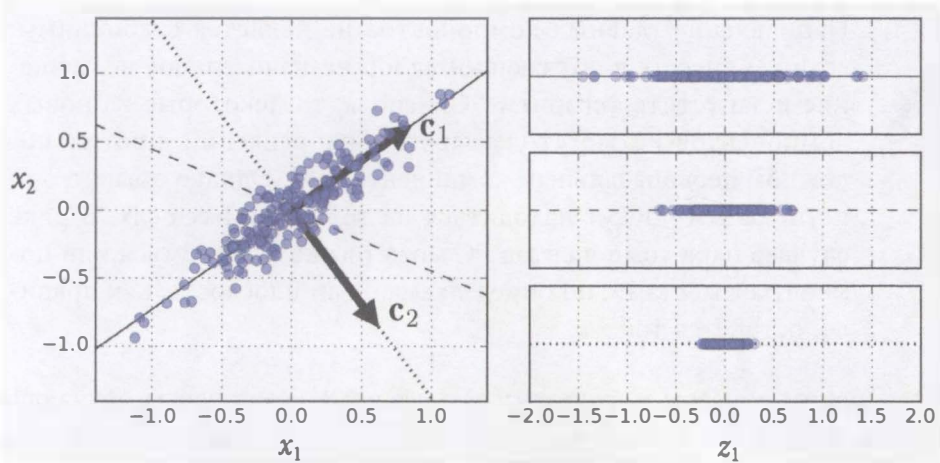


Рис. 8.7. Выбор подпространства для проецирования

Обосновать выбор можно и по-другому: эта ось сводит к минимуму средний квадрат расстояния между исходным набором данных и его проекцией на ось. Так выглядит довольно простая идея, лежащая в основе PCA⁴.

Главные компоненты

Алгоритм PCA идентифицирует ось, на долю которой приходится самая крупная величина дисперсии в обучающем наборе. На рис. 8.7 она представлена сплошной линией. Алгоритм PCA также находит вторую ось, перпендикулярную первой, на долю которой приходится самая крупная величина оставшейся дисперсии. В рассматриваемом двумерном примере выбора нет: это точечная линия. Если бы речь шла о многомерном наборе данных, тогда PCA также нашел бы третью ось, перпендикулярную обеим предшествующим осям, четвертую, пятую и т.д. — столько осей, сколько есть измерений в наборе данных.

Единичный вектор, который определяет i -тую ось, называется i -тым *главным компонентом* (*Principal Component* — *PC*). На рис. 8.7 первым компонентом PC является c_1 , а вторым компонентом PC — c_2 . На рис. 8.2 первые два компонента PC представлены перпендикулярными стрелками на плоскости; третий компонент PC был бы перпендикулярным к плоскости (указывая вверх или вниз).

⁴ “On Lines and Planes of Closest Fit to Systems of Points in Space” (“О линиях и плоскостях наиболее точного соответствия системам точек в пространстве”), К. Пирсон (1901 год) (<http://goo.gl/e2Qc8T>).



Направление главных компонент не является стабильным: если вы внесете в обучающий набор незначительное возмущение и запустите алгоритм PCA снова, то некоторые из новых компонент PC могут указывать в направлении, противоположном первоначальным компонентам PC. Однако обычно они по-прежнему будут находиться на тех же самых осях. В ряде случаев пара компонент PC может даже повернуться или поменяться местами, но определяемая ими плоскость, как правило, останется той же.

Итак, каким образом можно отыскать главные компоненты обучающего набора? К счастью, существует стандартный прием матричного разложения под названием *сингулярное разложение* (*Singular Value Decomposition* — *SVD*), который может провести декомпозицию матрицы X обучающего набора в скалярное произведение трех матриц $U \cdot \Sigma \cdot V^T$, где V содержит все искомые главные компоненты (уравнение 8.1).

Уравнение 8.1. Матрица главных компонент

$$V = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \dots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

Следующий код Python применяет функцию `svd()` из NumPy для получения всех главных компонент обучающего набора и затем извлекает первые два компонента PC:

```
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt.T[:, 0]
c2 = Vt.T[:, 1]
```



Алгоритм PCA предполагает, что набор данных центрирован относительно начала. Вы увидите, что классы PCA из Scikit-Learn сами позаботятся о центрировании данных. Тем не менее, если вы реализуете алгоритм PCA самостоятельно (как в предыдущем примере) либо используете другие библиотеки, то не забывайте сначала центрировать данные.

Проецирование до d измерений

После идентификации всех главных компонентов вы можете понизить размерность набора данных до d измерений за счет его проецирования на гиперплоскость, определенную первыми d главными компонентами. Выбор такой гиперплоскости гарантирует, что проекция будет предохранять максимально возможную дисперсию. Например, на рис. 8.2 трехмерный набор данных проецируется на двумерную плоскость, определенную первыми двумя главными компонентами, с предохранением крупной части дисперсии набора данных. В результате двумерная проекция выглядит очень похожей на исходный трехмерный набор данных.

Чтобы спроецировать обучающий набор на гиперплоскость, вы можете просто подсчитать скалярное произведение матрицы X обучающего набора и матрицы W_d , которая определена как матрица, содержащая первые d главных компонентов (т.е. матрица, состоящая из первых d столбцов V), как показано в уравнении 8.2.

Уравнение 8.2. Проецирование обучающего набора до d измерений

$$X_{d\text{-проекция}} = X \cdot W_d$$

Следующий код Python проецирует обучающий набор на плоскость, определенную первыми двумя главными компонентами:

```
W2 = Vt.T[:, :2]
X2D = X_centered.dot(W2)
```

Вот и все! Теперь вы знаете, как понизить размерность любого набора данных до любого количества измерений, одновременно предохраняя максимально возможную дисперсию.

Использование Scikit-Learn

Класс `PCA` из Scikit-Learn реализует алгоритм PCA с применением разложения SVD, как мы делали ранее. Приведенный далее код использует алгоритм PCA, чтобы понизить размерность набора данных до двух измерений (обратите внимание, что он автоматически заботится о центрировании данных):

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

После подгонки трансформатора PCA к набору данных вы можете обращаться к главным компонентам посредством переменной `components_` (она хранит главные компоненты в виде горизонтальных векторов, так что, скажем, первым главным компонентом будет `pca.components_.T[:, 0]`).

Коэффициент объясненной дисперсии

Еще одной очень полезной порцией информации является *коэффициент объясненной дисперсии* (*explained variance ratio*) каждого главного компонента, доступный через переменную `explained_variance_ratio_`. Он указывает долю дисперсии набора данных, которая лежит вдоль оси каждого главного компонента. Например, давайте посмотрим на коэффициенты объясненной дисперсии первых двух компонентов трехмерного набора данных, представленного на рис. 8.2:

```
>>> pca.explained_variance_ratio_  
array([ 0.84248607,  0.14631839])
```

Результат говорит о том, что 84.2% дисперсии набора данных лежит вдоль первой оси, а 14.6% — вдоль второй оси. Третьей оси остается менее 1.2%, потому разумно предположить, что она несет в себе мало информации.

Выбор правильного количества измерений

Вместо выбора произвольного числа измерений для понижения размерности обычно предпочтительнее выбирать такое количество измерений, которое соответствует достаточно большой порции дисперсии (скажем, 95%). Конечно, рекомендация не относится к понижению размерности с целью визуализации данных — в этом случае вы, как правило, хотите понизить размерность до 2 или 3.

Следующий код реализует алгоритм PCA без понижения размерности и вычисляет минимальное количество измерений, требуемых для предохранения 95% дисперсии обучающего набора:

```
pca = PCA()  
pca.fit(X_train)  
cumsum = np.cumsum(pca.explained_variance_ratio_)  
d = np.argmax(cumsum >= 0.95) + 1
```

Вы затем могли бы установить `n_components=d` и запустить алгоритм PCA снова. Однако есть намного лучший вариант: вместо указания количес-

тва главных компонентов, подлежащих предохранению, вы можете установить `n_components` в число с плавающей точкой между 0.0 и 1.0, указывающее долю дисперсии, которую желательно предохранить:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Еще один вариант предусматривает вычерчивание графика объясненной дисперсии как функции количества измерений (нужно просто вычертить `cumsum`; рис. 8.8). На кривой обычно будет крутой изгиб, где объясненная дисперсия прекращает быстрый рост. Вы можете считать это внутренне присущей размерностью набора данных. В нашем случае видно, что понижение размерности до примерно 100 измерений не приведет к потере слишком многой объясненной дисперсии.

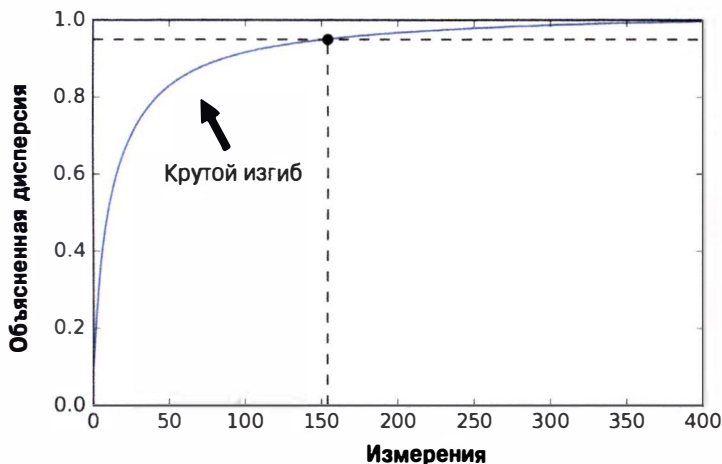


Рис. 8.8. Объясненная дисперсия как функция количества измерений

Алгоритм PCA для сжатия

Вполне очевидно, после понижения размерности обучающий набор требует гораздо меньше пространства. Например, попробуйте применить алгоритм PCA к набору данных MNIST, предохраняя 95% его дисперсии. Вы должны обнаружить, что каждый образец будет иметь лишь чуть выше 150 признаков вместо исходных 784 признаков. Итак, наряду с тем, что большая часть дисперсии предохранена, набор данных теперь меньше, чем 20% его первоначального размера! Это приемлемая степень сжатия, и вы заметите, что она способна значительно ускорить алгоритм классификации (такой как SVM).

Сокращенный набор данных также можно вернуть к 784 измерениям, применив обратную трансформацию проекции PCA. Разумеется, первоначальные данные не будут восстановлены, т.к. проекция привела к утрате некоторой информации (в рамках дисперсии 5%, которая была отброшена), но результат, вероятно, будет довольно близок к исходным данным. Средний квадрат расстояния между первоначальными и восстановленными данными (сжатыми и затем распакованными) называется *ошибкой восстановления* (*reconstruction error*). Например, показанный далее код сжимает набор данных MNIST до 154 измерений и с помощью метода `inverse_transform()` распаковывает его обратно в 784 измерения. На рис. 8.9 представлено несколько цифр из первоначального обучающего набора (слева) и те же цифры после сжатия и распаковки. Как видите, качество изображений слегка снизилось, но цифры по большей части остались незатронутыми.

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

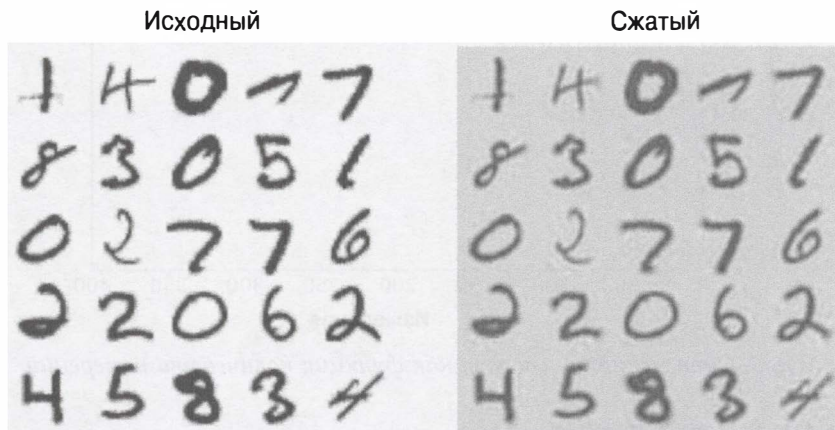


Рис. 8.9. Сжатие набора данных MNIST с предохранением 95% дисперсии

В уравнении 8.3 демонстрируется обратная трансформация.

Уравнение 8.3. Обратная трансформация PCA, восстанавливающая исходное количество измерений

$$\mathbf{X}_{\text{восстановленная}} = \mathbf{X}_{d\text{-проекция}} \cdot \mathbf{W}_d^T$$

Инкрементный анализ главных компонент

Проблема с предшествующей реализацией алгоритма PCA в том, что она требует помещения в память целого обучающего набора, чтобы алгоритм SVD мог работать. К счастью, были разработаны алгоритмы *инкрементного анализа главных компонент* (*Incremental PCA* — *IPCA*): вы можете расщепить обучающий набор на мини-пакеты и передавать алгоритму ИПСА по одному мини-пакету за раз. Такой прием удобен для крупных обучающих наборов и также для динамического применения алгоритма PCA (т.е. на лету, по мере поступления новых образцов).

Следующий код расщепляет набор данных MNIST на 100 мини-пакетов (используя функцию `array_split()` из NumPy) и передает их классу `IncrementalPCA`⁵ из Scikit-Learn, чтобы понизить размерность набора данных MNIST до 154 измерений (как и ранее). Обратите внимание, что вы обязаны вызывать метод `partial_fit()` с каждым мини-пакетом, а не метод `fit()` с целым обучающим набором:

```
from sklearn.decomposition import IncrementalPCA
n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)
X_reduced = inc_pca.transform(X_train)
```

В качестве альтернативы вы можете применять класс `mmap` из NumPy, который позволяет манипулировать крупным массивом, хранящимся в двоичном файле на диске, как если бы он целиком был размещен в памяти; класс загружает в память только нужные данные, когда они ему необходимы. Поскольку в каждый момент времени класс `IncrementalPCA` использует только небольшую часть массива, расходование памяти остается под контролем. Это делает возможным вызов обычного метода `fit()`:

```
X_mm = np.mmap(filename, dtype="float32",
               mode="readonly", shape=(m, n))
batch_size = m // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X_mm)
```

⁵ Библиотека Scikit-Learn использует алгоритм, описанный в статье “Incremental Learning for Robust Visual Tracking” (“Постепенное обучение для надежного визуального слежения”), Д. Росс и др. (2007 год) (<http://goo.gl/FmdhUP>).

Рандомизированный анализ главных компонентов

Библиотека Scikit-Learn предлагает еще один способ выполнения алгоритма PCA, называемый *рандомизированным анализом главных компонентов* (*randomized PCA*). Он представляет собой стохастический алгоритм, который быстро находит аппроксимацию первых d главных компонент. Его вычислительная сложность составляет $O(m \times d^2) + O(d^3)$ вместо $O(m \times n^2) + O(n^3)$, так что когда d намного меньше n , он значительно быстрее предшествующих алгоритмов.

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

Ядерный анализ главных компонентов

В главе 5 мы обсуждали ядерный трюк, т.е. математический прием, который неявно отображает образцы на пространство с очень большим числом измерений (называемое *пространством признаков* (*feature space*)), делающее возможным нелинейную классификацию и регрессию с помощью методов опорных векторов. Вспомните, что линейная граница решений в многомерном пространстве признаков соответствует сложной нелинейной границе решений в *исходном пространстве*.

Оказывается, тот же самый трюк может быть применен к PCA, позволяя выполнять сложные нелинейные проекции для понижения размерности. Это называется *ядерным анализом главных компонентов* (*kernel PCA* — *kPCA*)⁶. Часто полезно предохранять кластеры образцов после проекции или временно даже разворачивать наборы данных, которые лежат близко к скрученному многообразию.

Например, следующий код использует класс `KernelPCA` из Scikit-Learn для выполнения kPCA с ядром RBF (за дополнительными сведениями о ядре RBF и других ядрах обращайтесь в главу 5):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

⁶ “Kernel Principal Component Analysis” (“Ядерный анализ главных компонентов”), Б. Шолькопф, А. Смола, К. Мюллер (1999 год) (<http://goo.gl/5lQT5Q>).

На рис. 8.10 показан набор данных Swiss roll, пониженный до двух измерений с применением линейного ядра (эквивалент простого использования класса PCA), ядра RBF и сигмоидального ядра (логистическая потеря).

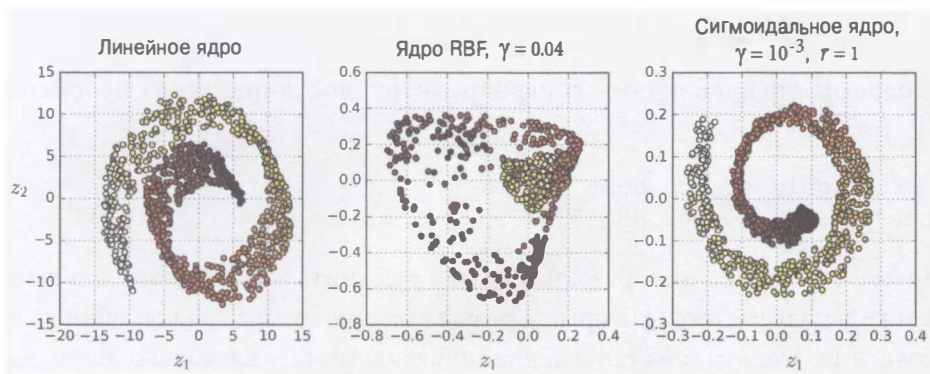


Рис. 8.10. Набор данных Swiss roll, пониженный до двух измерений за счет применения kPCA с разнообразными ядрами

Выбор ядра и подстройка гиперпараметров

Так как kPCA является алгоритмом обучения без учителя, нет никаких очевидных показателей производительности, которые помогли бы выбрать наилучшее ядро и значения гиперпараметров. Тем не менее, понижение размерности зачастую предпринимается как подготовительный шаг для задачи обучения с учителем (скажем, классификации), поэтому вы можете просто воспользоваться решетчатым поиском, чтобы выбрать ядро и гиперпараметры, которые приводят к наилучшей производительности на такой задаче. Например, показанный ниже код создает двухшаговый конвейер, сначала понижающий размерность до двух измерений с применением kPCA и затем использующий логистическую регрессию для классификации. Далее он применяет класс `GridSearchCV`, чтобы найти наилучшее ядро и значение `gamma` для kPCA с целью достижения наибольшей правильности классификации в конце конвейера.

```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline

clf = Pipeline([
    ("kpca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
])
```

```
param_grid = [{
    "kpcsa__gamma": np.linspace(0.03, 0.05, 10),
    "kpcsa__kernel": ["rbf", "sigmoid"]
}]
grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Теперь наилучшее ядро и гиперпараметры доступны через переменную `best_params_`:

```
>>> print(grid_search.best_params_)
{'kpcsa__gamma': 0.043333333333333335, 'kpcsa__kernel': 'rbf'}
```

Другой подход, на этот раз совершенно без учителя, заключается в выборе ядра и гиперпараметров, которые в результате дают наименьшую ошибку восстановления. Однако восстановление производится не настолько легко, как в случае линейного PCA, и вот почему. На рис. 8.11 представлен первоначальный трехмерный набор данных Swiss roll (слева сверху) и результирующий двумерный набор данных после использования kPCA с ядром RBF (справа сверху).

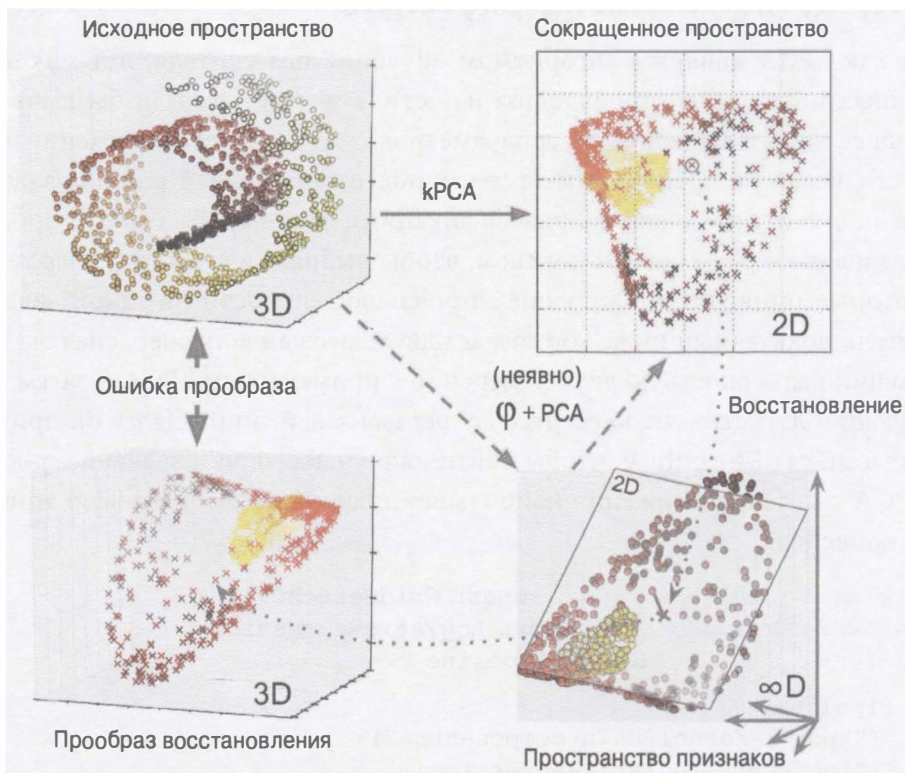


Рис. 8.11. Ядерный PCA и ошибка прообраза восстановления

Благодаря ядерному трюку это математически эквивалентно отображению обучающего набора на бесконечномерное пространство признаков (справа внизу) с применением *отображения признака ϕ* и последующему проецированию трансформированного обучающего набора в двумерный набор, используя линейный PCA. Обратите внимание, что если бы мы могли инвертировать шаг линейного PCA для заданного образца в пространстве с сокращенным числом измерений, то восстановленная точка располагалась бы в пространстве признаков, а не в исходном пространстве (например, подобно точке, представленной на диаграмме посредством **x**). Поскольку пространство признаков является бесконечномерным, мы не можем вычислить восстановленную точку и, следовательно, не можем подсчитать точную ошибку восстановления. К счастью, в исходном пространстве можно найти точку, которая будет отображаться близко к восстановленной точке. Это называется *прообразом (pre-image)* восстановления. Имея такой прообраз, вы можете измерить квадрат его расстояния до исходного образца и затем выбрать ядро и гиперпараметры, которые сводят к минимуму ошибку прообраза восстановления.

Вас может интересовать, как выполняется такое восстановление. Одно из решений предусматривает обучение регрессионной модели с учителем, при котором спроецированные образцы выступают в качестве обучающего набора, а исходные образцы — в качестве целей. Библиотека Scikit-Learn будет делать это автоматически, если вы установите `fit_inverse_transform=True`, как демонстрируется в следующем коде⁷:

```
rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.0433,
                    fit_inverse_transform=True)
X_reduced = rbf_pca.fit_transform(X)
X_preimage = rbf_pca.inverse_transform(X_reduced)
```



По умолчанию `fit_inverse_transform=False` и класс `KernelPCA` не имеет метода `inverse_transform()`. Этот метод создается только в случае установки `fit_inverse_transform=True`.

⁷ Библиотека Scikit-Learn применяет алгоритм, основанный на ядерной гребневой регрессии (Kernel Ridge Regression), который описан в статье “Learning to Find Pre-images” (“Обучение для нахождения прообразов”), Г. Бакир, Д. Вестон и Б. Шолькопф (Тюбинген, Германия: Институт биологической кибернетики общества имени Макса Планка, 2004 год) (<http://goo.gl/d0ydY6>).

Затем можно подсчитать ошибку прообраза восстановления:

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(X, X_preimage)
32.786308795766132
```

Теперь вы можете воспользоваться решетчатым поиском с перекрестной проверкой, чтобы найти ядро и гиперпараметры, которые сводят к минимуму ошибку прообраза восстановления.

LLE

*Локальное линейное вложение (Locally-Linear Embedding — LLE)*⁸ является еще одним очень мощным приемом *нелинейного понижения размерности (nonlinear dimensionality reduction — NLDR)*. Это методика обучения на основе многообразий, которая не полагается на проекции подобно предшествующим алгоритмам. Выражаясь кратко, LLE сначала измеряет, как каждый обучающий образец линейно связан со своими ближайшими соседями, и затем ищет представление обучающего набора с меньшим количеством измерений, где такие локальные связи лучше всего предохраняются (вскоре мы рассмотрим детали). В результате прием LLE оказывается очень эффективным при разворачивании скрученных многообразий, особенно когда шум не слишком большой.

Например, приведенный ниже код применяет класс `LocallyLinearEmbedding` из Scikit-Learn для разворачивания набора данных Swiss roll. Результирующий двумерный набор данных показан на рис. 8.12. Как видите, набор данных Swiss roll полностью развернут и расстояния между образцами локально хорошо предохранены. Тем не менее, расстояния не предохраняются при большем масштабе: левая часть неразвернутого набора данных Swiss roll сдвинута, в то время как правая часть растянута. Несмотря на это, прием LLE выполнил неплохую работу по моделированию многообразия.

```
from sklearn.manifold import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

⁸ “Nonlinear Dimensionality Reduction by Locally Linear Embedding” (“Нелинейное понижение размерности путем локального линейного вложения”), С. Ровайс, Л. Саул (2000 год) (<https://goo.gl/iA9bns>).

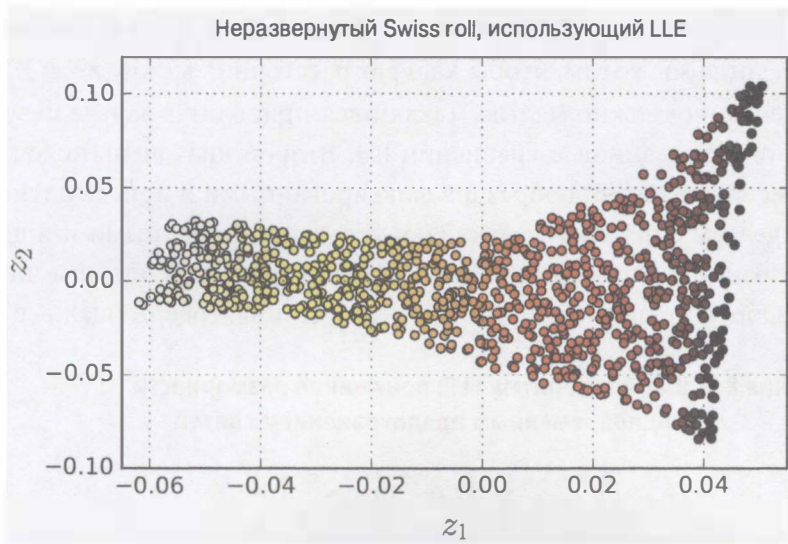


Рис. 8.12. Неразвернутый набор данных Swiss roll, использующий LLE

Вот как действует LLE: сначала для каждого обучающего образца $\mathbf{x}^{(i)}$ алгоритм идентифицирует его k ближайших соседей (в предыдущем коде $k = 10$) и затем пытается восстановить $\mathbf{x}^{(i)}$ как линейную функцию этих соседей. Точнее говоря, он ищет такие веса $w_{i,j}$, чтобы квадрат расстояния между $\mathbf{x}^{(i)}$ и $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$ был как можно меньше, при условии, что $w_{i,j} = 0$, если $\mathbf{x}^{(j)}$ не является одним из k ближайших соседей $\mathbf{x}^{(i)}$. Таким образом, первым шагом LLE оказывается задача условной оптимизации, описанная в уравнении 8.4, где \mathbf{W} — матрица весов, содержащая все веса $w_{i,j}$. Второе ограничение просто нормализует веса для каждого обучающего образца $\mathbf{x}^{(i)}$.

Уравнение 8.4. Шаг 1 алгоритма LLE: линейное моделирование локальных связей

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2$$

$$\text{при условии} \begin{cases} w_{i,j} = 0, \text{ если } \mathbf{x}^{(j)} \text{ не является одним из } k \text{ ближайших соседей } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 \text{ для } i = 1, 2, \dots, m \end{cases}$$

После этого шага матрица весов $\widehat{\mathbf{W}}$ (содержащая веса $\widehat{w}_{i,j}$) представляет локальные линейные связи между обучающими образцами. Вторым шагом заключается в отображении обучающих образцов на d -мерное пространство (где $d < n$) с одновременным предохранением как можно большего числа име-

ющихся локальных связей. Если $\mathbf{z}^{(i)}$ — отражение $\mathbf{x}^{(i)}$ в этом d -мерном пространстве, тогда мы хотим, чтобы квадрат расстояния между $\mathbf{z}^{(i)}$ и $\sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)}$ был насколько возможно малым. Такая идея приводит к задаче безусловной оптимизации, описанной в уравнении 8.5. Второй шаг очень похож на первый, но вместо удержания образцов фиксированными и поиска оптимальных весов мы делаем обратное: удерживаем веса фиксированными и ищем оптимальную позицию отражений образцов в пространстве с низким числом измерений. Обратите внимание, что \mathbf{Z} — это матрица, содержащая все $\mathbf{z}^{(i)}$.

Уравнение 8.5. Шаг 2 алгоритма LLE: понижение размерности с одновременным предохранением связей

$$\hat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left(\mathbf{z}^{(i)} - \sum_{j=1}^m \hat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Реализация алгоритма LLE в Scikit-Learn имеет следующую вычислительную сложность: $O(m \log(m) n \log(k))$ для нахождения k ближайших соседей, $O(m n k^3)$ для оптимизации весов и $O(dm^2)$ для построения представлений с низким числом измерений. К сожалению, наличие m^2 в последнем элементе делает этот алгоритм плохо масштабируемым для очень крупных наборов данных.

Другие приемы понижения размерности

Существует много других приемов понижения размерности, ряд которых доступен в Scikit-Learn. Ниже перечислены наиболее популярные из них.

- **Многомерное шкалирование (Multidimensional Scaling — MDS)** понижает размерность, одновременно пытаясь предохранить расстояния между образцами (рис. 8.13).
- **Изометрическое отображение (isomap)** создает граф, соединяя каждый образец с его ближайшими соседями и затем понижая размерность с попыткой предохранения *геодезических расстояний (geodesic distance)*⁹ между образцами.
- **Стохастическое вложение соседей с t -распределением (t-distributed Stochastic Neighbor Embedding — t-SNE)** понижает размерность, одно-

⁹ Геодезическое расстояние между двумя узлами в графе представляет собой количество узлов на кратчайшем пути между этими узлами.

временно пытаюсь сохранять похожие образцы поблизости и непохожие образцы на отдалении. Оно главным образом применяется для визуализации, в частности визуального представления кластеров образцов в многомерном пространстве (например, для двумерной визуализации изображений MNIST).

- **Линейный дискриминантный анализ** (*Linear Discriminant Analysis — LDA*) на самом деле представляет собой алгоритм классификации, но во время обучения он узнает наиболее отличающиеся оси между классами, которые затем могут использоваться для определения гиперплоскости, куда будут проецироваться данные. Преимущество в том, что проекция будет удерживать классы как можно дальше друг от друга, поэтому LDA является хорошим приемом для понижения размерности перед запуском другого алгоритма классификации, такого как SVM.

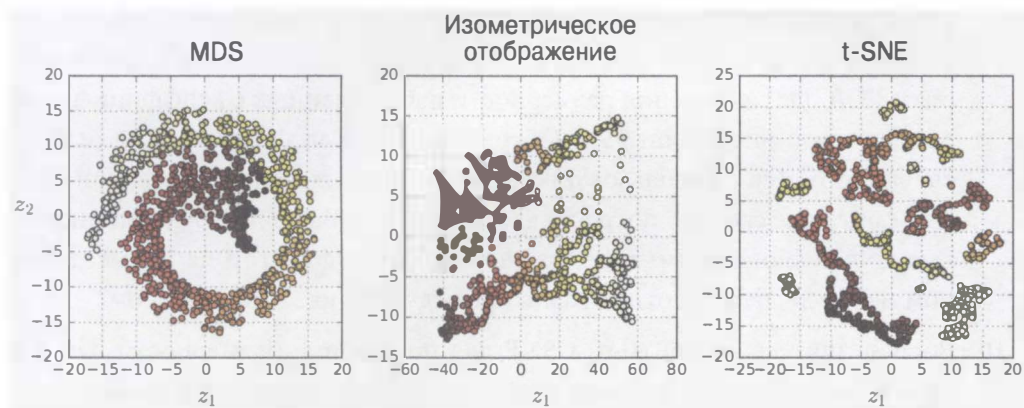


Рис. 8.13. Понижение размерности набора данных *Swiss roll* до двух измерений с применением разных приемов

Упражнения

1. Каковы главные мотивы для понижения размерности набора данных? В чем заключаются основные недостатки понижения размерности?
2. Что такое “проклятие размерности”?
3. После того как размерность набора данных была понижена, можно ли обратить операцию? Если да, то как? Если нет, то почему?
4. Можно ли использовать алгоритм PCA для понижения размерности крайне нелинейного набора данных?

5. Предположим, что вы выполняете алгоритм PCA на 1000-мерном наборе данных, установив коэффициент объясненной дисперсии в 95%. Сколько измерений будет иметь результирующий набор данных?
6. В каких случаях вы бы применяли простой алгоритм PCA, инкрементный PCA, рандомизированный PCA или ядерный PCA?
7. Как вы можете оценить производительность алгоритма понижения размерности на своем наборе данных?
8. Имеет ли какой-нибудь смысл соединять в цепочку два разных алгоритма понижения размерности?
9. Загрузите набор данных MNIST (введенный в главе 3) и расщепите его на обучающий набор и испытательный набор (возьмите первые 60 000 образцов для обучения, а оставшиеся 10 000 для испытаний). Обучите классификатор на основе случайного леса с использованием набора данных и зафиксируйте время, сколько это заняло, после чего оцените результирующую модель на испытательном наборе. Далее примените алгоритм PCA для понижения размерности набора данных с коэффициентом объясненной дисперсии 95%. Обучите новый классификатор на основе случайного леса с использованием набора данных меньшей размерности и посмотрите, сколько потребовалось времени. Было ли обучение значительно более быстрым? Оцените новый классификатор на испытательном наборе: как он соотносится с предыдущим классификатором?
10. Воспользуйтесь алгоритмом t-SNE для понижения размерности набора данных MNIST до двух измерений и вычертите результат с применением Matplotlib. Для представления целевого класса каждого изображения можете использовать график рассеяния с 10 разными цветами. В качестве альтернативы на месте каждого образца вы можете выводить цветные цифры или даже рисовать версии с уменьшенным размером самих изображений цифр (в случае вычерчивания всех цифр визуализация будет слишком перегруженной, поэтому вы должны либо рисовать случайную выборку, либо вычерчивать образец, только если не были вычерчены другие образцы на близком расстоянии). Вы должны получить аккуратную визуализацию с вполне разделенными кластерами цифр. Попробуйте применить другие алгоритмы понижения размерности, такие как PCA, LLE или MDS, и сравните результирующие визуализации.

Решения приведенных упражнений доступны в приложении А.

Нейронные сети и глубокое обучение

Подготовка к работе с TensorFlow

TensorFlow — это мощная программная библиотека с открытым кодом, предназначенная для численных расчетов. Она особенно хорошо подходит и точно подогнана под крупномасштабное машинное обучение. Ее базовый принцип прост: вы определяете в Python граф вычислений, подлежащих выполнению (например, такой как на рис. 9.1), а TensorFlow берет этот граф и эффективно прогоняет с использованием оптимизированного кода C++.

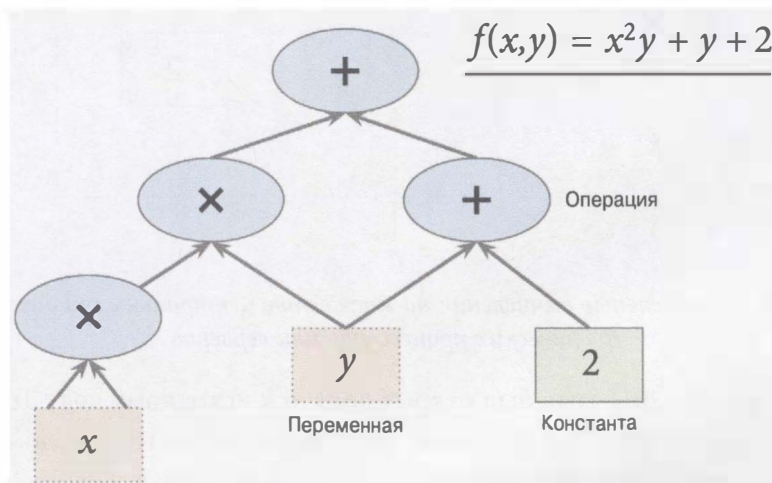


Рис. 9.1. Простой вычислительный граф

Самое главное, граф можно разбивать на части и прогонять их параллельно на множестве центральных процессоров или графических процессоров (рис. 9.2). Библиотека TensorFlow также поддерживает распределенные вычисления, поэтому вы в состоянии обучать громадные нейронные сети на невероятно больших обучающих наборах за приемлемое время, распределяя вычисления по сотням серверов (см. главу 12). Библиотека TensorFlow способна обучать сеть со многими миллионами параметров на обучающих наборах, состоящих из миллиардов образцов с миллионами признаков в

каждом. Сказанное не должно вызывать удивления, поскольку библиотека TensorFlow была разработана командой Google Brain и приводит в действие многие крупномасштабные службы Google, такие как Google Cloud Speech, Google Photos и Google Search.

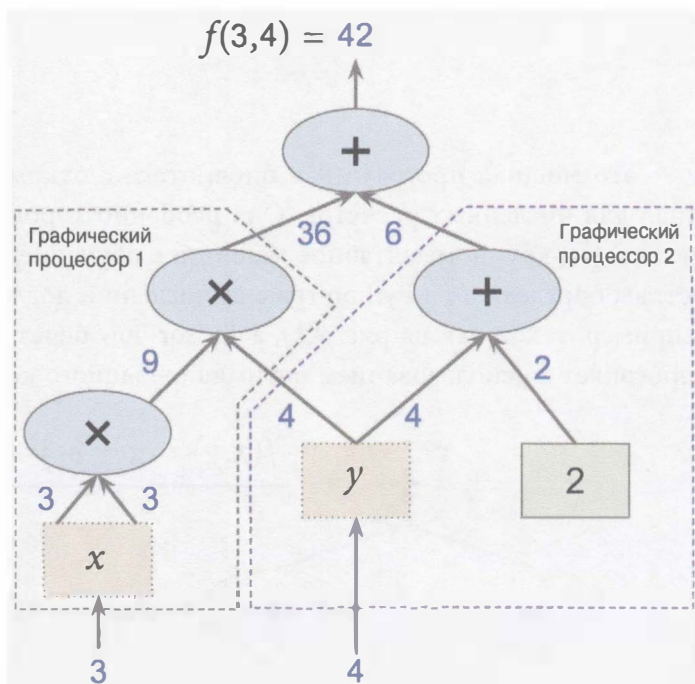


Рис. 9.2. Параллельные вычисления на множестве центральных процессоров, графических процессоров или серверов

Когда в ноябре 2015 года был открыт доступ к исходному коду TensorFlow, уже существовало много популярных библиотек с открытым кодом для глубокого обучения (Deep Learning), ряд которых перечислен в табл. 9.1, и надо признать, что большинство средств TensorFlow было доступно в той или иной библиотеке. Тем не менее, ясное проектное решение, масштабируемость, гибкость¹ и обширная документация (не говоря уже о наличии Google в названии) быстро продвинуло TensorFlow на вершину списка. Выражаясь кратко, библиотека TensorFlow с самого начала задумывалась быть гибкой, масштабируемой и готовой к работе, а существующие фреймворки обладали только двумя из указанных трех характеристик.

¹ Библиотека TensorFlow не ограничена нейронными сетями или даже машинным обучением; при желании вы можете запускать имитационные модели из квантовой физики.

Ниже перечислены некоторые важные особенности библиотеки TensorFlow.

- Она работает не только на компьютерах с Windows, Linux и macOS, но также на мобильных устройствах, функционирующих под управлением iOS и Android.
- Она предоставляет очень простой API-интерфейс для Python под названием *TFLearn*² ([tensorflow.contrib.learn](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/learn)), совместимый с библиотекой Scikit-Learn. Как вы увидите, его можно применять для обучения нейронных сетей разнообразных типов, написав лишь несколько строк кода. Ранее это бы независимый проект *Scikit Flow* (или *skflow*).
- Она также предоставляет еще один простой API-интерфейс, называемый *TF-slim* ([tensorflow.contrib.slim](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim)), который упрощает построение, обучение и оценку нейронных сетей.
- Поверх TensorFlow был независимо построен ряд других высокоуровневых API-интерфейсов, таких как *Keras* (<http://keras.io/>; теперь доступный в [tensorflow.contrib.keras](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/keras)) или Pretty Tensor (<https://github.com/google/prettytensors/>).
- Ее главный API-интерфейс для Python обеспечивает намного большую гибкость (ценой более высокой сложности) при создании всех видов вычислений, включая любую архитектуру нейронной сети, какую только можно вообразить.
- Она включает высокоэффективные реализации на C++ многих операций МО, особенно тех, которые нужны для построения нейронных сетей. Имеется также API-интерфейс для C++, позволяющий определять собственные высокопроизводительные операции.
- Она предоставляет несколько расширенных узлов оптимизации для поиска параметров, которые доводят до минимума функцию издержек. Их очень легко использовать, поскольку TensorFlow автоматически позаботится о вычислении градиентов функций, которые вы определите. Это называется *автоматическим дифференцированием* (*automatic differentiating* — *autodiff*).
- Она поступает с великолепным инструментом визуализации под названием *TensorBoard*, который позволяет просматривать вычислительный граф, показывать кривые обучения и т.д.

² Не спутайте с библиотекой TFLearn, которая является независимым проектом.

- Компания Google также запустила облачную службу для прогона графов TensorFlow (<https://cloud.google.com/ml>).
- Последнее, но не менее важное: с ней связана команда преданных и опытных разработчиков, а также растущее сообщество, которое содействует ее улучшению. Она является одним из самых популярных проектов с открытым кодом на GitHub, и на ее основе строится все больше и больше замечательных проектов (к примеру, просмотрите страницу ресурсов по адресу <https://www.tensorflow.org/> или <https://github.com/jtoy/awesome-tensorflow>). Вопросы технического характера следует задавать на сайте <http://stackoverflow.com/>, снабжая их меткой "tensorflow". Вы можете регистрировать дефекты и запросы средств посредством GitHub. Для участия в общих обсуждениях запишитесь в группу Google (<http://goo.gl/N7kRF9>).

В настоящей главе мы пройдемся по основам TensorFlow, от установки до создания, запуска, сохранения и визуализации простых вычислительных графов. Важно овладеть этими основами, прежде чем строить свою первую нейронную сеть (чем мы займемся в следующей главе).

Таблица 9.1. Библиотеки глубокого обучения с открытым кодом (список неполон)

Библиотека	API-интерфейс	Платформа	Кто начал	Год
Caffe	Python, C++, Matlab	Linux, macOS, Windows	Я. Цзя, UC Berkeley (BVLG)	2013
Deeplearning4j	Java, Scala, Clojure	Linux, macOS, Windows, Android	А. Гибсон, Д. Паттерсон	2014
H2O	Python, R	Linux, macOS, Windows	H2O.ai	2014
MXNet	Python, C++ и др.	Linux, macOS, Windows, iOS, Android	DMLC	2015
TensorFlow	Python, C++	Linux, macOS, Windows, iOS, Android	Google	2015
Theano	Python	Linux, macOS, iOS	Монреальский университет	2010
Torch	C++, Lua	Linux, macOS, iOS, Android	Р. Коллобер, К. Кавукчуоглу, К. Фарабе	2002

Установка

Поехали! Предполагая, что вы установили Jupyter и Scikit-Learn согласно инструкциям в главе 2, для установки TensorFlow можете просто применить `pip`. Если вы создали изолированную среду с использованием `virtualenv`, тогда сначала понадобится ее активировать:

```
$ cd $ML_PATH # Ваш рабочий каталог ML (например, $HOME/ml)
$ source env/bin/activate
```

Затем установите TensorFlow (если `virtualenv` не применяется, то вам нужны права администратора или добавьте параметр `--user`):

```
$ pip3 install --upgrade tensorflow
```



Для поддержки графических процессоров вместо `tensorflow` вам необходимо установить `tensorflow-gpu`. За дополнительными сведениями обращайтесь в главу 12.

Чтобы проверить результаты установки, введите приведенную ниже команду. Она должна вывести версию установленной библиотеки TensorFlow.

```
$ python3 -c 'import tensorflow; print(tensorflow.__version__)'
1.3.0
```

Создание первого графа и его прогон в сеансе

Следующий код создает граф, представленный на рис. 9.1:

```
import tensorflow as tf
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

Вот и вся история! Важно понимать, что этот код в действительности не выполняет никаких вычислений, хотя и выглядит так, будто бы выполняет (особенно последняя строка). Он только создает вычислительный граф. На самом деле он даже не инициализирует переменные. Чтобы прогнать такой граф, понадобится открыть *сеанс* (*session*) TensorFlow и с его использованием инициализировать переменные и оценить `f`. Сеанс TensorFlow позаботится о помещении операций в *устройства* (*device*), такие как центральные про-

цессоры и графические процессоры, их выполнении и сохранении значений всех переменных³. Показанный далее код создает сеанс, инициализирует переменные, вычисляет `f` и закрывает сеанс (освобождая ресурсы):

```
>>> sess = tf.Session()
>>> sess.run(x.initializer)
>>> sess.run(y.initializer)
>>> result = sess.run(f)
>>> print(result)
42
>>> sess.close()
```

Повторение `sess.run()` каждый раз может стать обременительным, но к счастью есть лучший способ:

```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Внутри блока `with` сеанс устанавливается в качестве стандартного. Вызов `x.initializer.run()` эквивалентен вызову `tf.get_default_session().run(x.initializer)` и подобным же образом вызов `f.eval()` эквивалентен вызову `tf.get_default_session().run(f)`. Такой прием делает код легче для чтения. Кроме того, в конце блока сеанс автоматически закрывается.

Вместо запуска инициализатора вручную для каждой одиночной переменной вы можете применить функцию `global_variables_initializer()`. Обратите внимание, что она на самом деле не выполняет инициализацию немедленно, а взамен создает граф, который будет инициализировать все переменные, когда он прогоняется:

```
init = tf.global_variables_initializer()    # подготовка инициализирующего узла

with tf.Session() as sess:
    init.run()    # действительная инициализация всех переменных
    result = f.eval()
```

Внутри Jupyter или в командной оболочке Python вы можете отдать предпочтение созданию экземпляра `InteractiveSession`. Единственное от-

³ В главе 12 вы узнаете, что в распределенной версии TensorFlow значения переменных хранятся на серверах, а не в сеансах.

личие `InteractiveSession` от обыкновенного `Session` в том, что после создания экземпляр `InteractiveSession` автоматически делает себя стандартным сеансом, поэтому блок `with` не нужен (но после завершения работы с сеансом его придется закрывать вручную):

```
>>> sess = tf.InteractiveSession()
>>> init.run()
>>> result = f.eval()
>>> print(result)
42
>>> sess.close()
```

Программа TensorFlow, как правило, делится на две части: первая часть строит вычислительный граф (*стадия построения*), а вторая часть его прогоняет (*стадия выполнения*). Стадия построения обычно создает вычислительный граф, представляющий модель МО, и вычисления, требующиеся для обучения модели. Стадия выполнения в целом запускает цикл, который неоднократно производит шаг обучения (скажем, один шаг на мини-пакет), постепенно улучшая параметры модели. Вскоре мы разберем пример.

Управление графами

Любой созданный вами узел автоматически добавляется к стандартному графу:

```
>>> x1 = tf.Variable(1)
>>> x1.graph is tf.get_default_graph()
True
```

В большинстве случаев такое поведение устраивает, но иногда может возникнуть необходимость в управлении множеством независимых графов. Для этого понадобится создать новый экземпляр `Graph` и временно сделать его стандартным графом внутри блока `with`:

```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
True
>>> x2.graph is tf.get_default_graph()
False
```



Во время экспериментирования в Jupyter (или в командной оболочке Python) те же самые команды часто запускают более одного раза. Результатом оказывается стандартный граф, содержащий много дублированных узлов. Одно из решений может заключаться в перезапуске ядра Jupyter (или командной оболочки Python), но более приемлемое решение предусматривает просто возвращение стандартного графа в исходное состояние посредством `tf.reset_default_graph()`.

Жизненный цикл значения узла

При оценке узла TensorFlow автоматически определяет набор узлов, от которых он зависит, и оценивает такие узлы первыми. Например, взгляните на следующий код:

```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval()) # 10
    print(z.eval()) # 15
```

Прежде всего, в коде определяется очень простой граф. Затем создается сеанс, в котором полученный граф прогоняется для оценки `y`: библиотека TensorFlow автоматически обнаруживает, что `y` зависит от узла `x`, который зависит от `w`, поэтому она сначала оценивает `w`, затем `x`, затем `y` и возвращает значение `y`. Наконец, код прогоняет граф для оценки `z`. И снова TensorFlow замечает, что первыми должны оцениваться `w` и `x`. Важно отметить, что TensorFlow *не* будет повторно использовать результат предыдущей оценки `w` и `x`. Короче говоря, предшествующий код оценивает `w` и `x` дважды.

Между прогонами графа значения всех узлов отбрасываются кроме значений переменных, которые поддерживаются сеансом параллельно прогонам графа (как будет показано в главе 12, классы очередей и чтения также поддерживают состояние). Переменная начинает свое существование, когда запускается ее инициализатор, и заканчивает его, когда закрывается сеанс.

Если вы хотите эффективно оценивать `y` и `z`, не оценивая `w` и `x` дважды, как в предшествующем коде, тогда потребуется запросить у TensorFlow оценку `y` и `z` только в одном прогоне графа, как демонстрируется в следующем коде:

```
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
    print(y_val) # 10
    print(z_val) # 15
```



В версии TensorFlow с единственным процессом множество сеансов не разделяют какого-либо состояния, даже если они повторно применяют тот же самый граф (каждый сеанс будет иметь собственную копию каждой переменной). В распределенной версии TensorFlow (см. главу 12) состояние переменных хранится на серверах, а не в сеансах, потому множество сеансов способны разделять те же самые переменные.

Линейная регрессия с помощью TensorFlow

Операции TensorFlow могут принимать любое количество входных данных и выдавать любое количество выходных данных. Например, операции сложения и умножения принимают два входных значения и производят одно выходное значение. Константы и переменные не принимают входные данные (они называются *операциями источника*). Входные и выходные данные представляют собой многомерные массивы, называемые *тензорами* (*tensor*); отсюда и происходит имя библиотеки: “tensor flow” — “поток тензоров”. Подобно массивам NumPy тензоры имеют тип и форму. На самом деле тензоры в API-интерфейсе для Python просто представлены массивами `ndarray` из NumPy. Они обычно содержат числа с плавающей точкой, но их также можно использовать для передачи строк (произвольных байтовых массивов).

В приведенных до сих пор примерах тензоры содержали только одиночное скалярное значение, но, конечно же, вычисления можно производить на массивах любой формы. Приведенный ниже код манипулирует двумерными массивами для выполнения линейной регрессии на наборе данных, содержащем цены на недвижимость в Калифорнии (см. главу 2). Код начинается с извлечения набора данных. Затем он добавляет дополнительный входной признак смещения ($x_0 = 1$) ко всем обучающим образцам (с применением NumPy, так что действие выполняется немедленно). Далее код создает два константных

узла TensorFlow, X и y , для хранения этих данных и целей⁴, после чего использует ряд матричных операций, предоставляемых TensorFlow, для определения θ . Такие матричные функции — `transpose()`, `matmul()` и `matrix_inverse()` — не требуют пояснений, но как обычно, они не выполняют вычисления немедленно; взамен создаются узлы в графе, которые будут выполнены при прогоне графа. Вы можете выявить, что определение θ соответствует нормальному уравнению $(\hat{\theta} = (X^T \cdot X)^{-1} \cdot X^T \cdot y$; см. главу 4). Наконец, код создает сеанс и применяет его для оценки θ .

```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1),
                dtype=tf.float32, name="y")

XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

Главное преимущество показанного кода по сравнению с вычислением нормального уравнения напрямую, используя NumPy, заключается в том, что TensorFlow будет автоматически выполнять его на плате графического процессора при ее наличии (конечно, в случае установки TensorFlow с поддержкой графических процессоров; за деталями обращайтесь в главу 12).

Реализация градиентного спуска

Давайте попробуем вместо нормального уравнения применить пакетный градиентный спуск (введенный в главе 4). Сначала мы будем делать это путем расчета градиентов вручную, затем используем средство autodiff из TensorFlow, чтобы позволить TensorFlow вычислять градиенты автоматически, и в заключение применим пару готовых оптимизаторов TensorFlow.

⁴ Обратите внимание, что `housing.target` представляет собой одномерный массив, но для вычисления θ нам нужно придать ему форму вектора-столбца. Вспомните, что NumPy-функция `reshape()` разрешает указывать для одного из измерений `-1` (означает “неопределенное”): это измерение будет вычисляться на основе длины массива и остальных измерений.



При использовании градиентного спуска помните, что важно сначала нормализовать входные векторы признаков, иначе обучение может значительно замедлиться. Вы можете делать это с применением TensorFlow, NumPy, класса `StandardScaler` из Scikit-Learn или любого другого решения по вашему предпочтению. В представленном далее коде предполагается, что нормализация уже проведена.

Расчет градиентов вручную

Показанный далее код не требует пояснений за исключением нескольких новых элементов.

- Функция `random_uniform()` создает в графе узел, который будет генерировать тензор, содержащий случайные значения, с заданной формой и диапазоном во многом подобно функции `rand()` из NumPy.
- Функция `assign()` создает узел, который будет присваивать переменной новое значение. В данном случае он реализует шаг пакетного градиентного спуска $\theta(\text{следующий шаг}) = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$.
- Главный цикл многократно выполняет шаг обучения (`n_epochs` раз) и через каждые 100 итераций выводит текущее значение среднеквадратической ошибки (`mse`). Вы должны заметить, что MSE уменьшается с каждой итерацией.

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias,
                dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1),
                dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0),
                    name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
```



```

for epoch in range(n_epochs):
    if epoch % 100 == 0:
        print("Эпоха", epoch, "MSE =", mse.eval())
        sess.run(training_op)

best_theta = theta.eval()

```

Использование autodiff

Предыдущий код работает хорошо, но требует математического выведения градиентов из функции издержек (MSE). В случае линейной регрессии это довольно легко, но если выведение пришлось бы делать с глубокими нейронными сетями, то головной боли оказалось бы достаточно: оно утомительно и подвершено ошибкам. Вы могли бы применить *символическое дифференцирование* (*symbolic differentiation*), чтобы автоматически найти уравнения для частных производных, но результирующий код не обязательно был бы очень эффективным.

Чтобы понять причину, возьмем функцию $f(x) = \exp(\exp(\exp(x)))$. Если вы знаете исчисление, то можете вывести ее производную $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$. В случае написания кода $f(x)$ и $f'(x)$ отдельно и в точности, как они выглядят, код не будет настолько эффективным, как мог бы быть. Более эффективное решение предусматривает написание функции, которая сначала вычисляет $\exp(x)$, затем $\exp(\exp(x))$, далее $\exp(\exp(\exp(x)))$ и возвращает все три значения. Это напрямую дает $f(x)$ (третий элемент), и если нужна производная, тогда можно просто перемножить все три элемента. При наивном подходе для вычисления $f(x)$ и $f'(x)$ функцию `exp` пришлось вызывать бы девять раз. В случае только что рассмотренного подхода ее нужно вызывать лишь три раза.

Все становится хуже, когда функция определена некоторым произвольным кодом. Сможете ли вы отыскать уравнение (или код) для вычисления частных производных следующей функции? Совет: даже не пытайтесь.

```

def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z

```

К счастью, на выручку приходит средство autodiff библиотеки TensorFlow: оно способно автоматически и эффективно вычислять градиенты. Просто замените строку `gradients = ...` в коде градиентного спуска, приведенно-

го в предыдущем разделе, показанной ниже строкой, и код продолжит прекрасно работать:

```
gradients = tf.gradients(mse, [theta])[0]
```

Функция `gradients()` берет операцию (в данном случае `mse`) и список переменных (здесь только `theta`) и создает список операций (по одной на переменную) для вычисления градиентов операции относительно каждой переменной. Таким образом, узел `gradients` будет вычислять вектор-градиент MSE по отношению к `theta`.

Существуют четыре основных подхода к автоматическому вычислению градиентов, которые сведены в табл. 9.2. Библиотека TensorFlow использует *автоматическое дифференцирование в обратном режиме* (*reverse-mode autodiff*), которое идеально (в плане эффективности и правильности), когда есть много входных и мало выходных данных, как часто происходит в нейронных сетях. Оно вычисляет все частные производные выходов относительно всех входов лишь за $n_{\text{выходов}} + 1$ обходов графа.

Таблица 9.2. Основные решения для автоматического вычисления градиентов

Прием	Количество обходов графа для вычисления всех градиентов	Правильность	Поддержка произвольного кода	Примечание
Численное дифференцирование (numerical differentiation)	$n_{\text{входов}} + 1$	Низкая	Да	Тривиально в реализации
Символическое дифференцирование (symbolic differentiation)	—	Высокая	Нет	Строит очень непохожий граф
Автоматическое дифференцирование в прямом режиме (forward-mode autodiff)	$n_{\text{входов}}$	Высокая	Да	Применяет дуальные числа (dual number)
Автоматическое дифференцирование в обратном режиме (reverse-mode autodiff)	$n_{\text{выходов}} + 1$	Высокая	Да	Реализовано в TensorFlow

Если вас интересует, как работает эта магия, тогда загляните в приложение Г.

Использование оптимизатора

Итак, библиотека TensorFlow вычисляет для вас градиенты. Но все даже еще проще: она также предоставляет несколько готовых оптимизаторов, включая оптимизатор градиентного спуска. Вы можете лишь заменить предшествующие строки `gradients = ...` и `training_op = ...` следующим кодом и он снова будет успешно работать:

```
optimizer =  
    tf.train.GradientDescentOptimizer(learning_rate=learning_rate)  
training_op = optimizer.minimize(mse)
```

При желании применить оптимизатор другого типа вам необходимо изменить только одну строку. Например, вы можете использовать *моментный оптимизатор* (`momentum optimizer`), который часто сходится гораздо быстрее градиентного спуска (см. главу 11), определив его так:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,  
                                         momentum=0.9)
```

Передача данных алгоритму обучения

Давайте попробуем модифицировать предыдущий код, чтобы реализовать мини-пакетный градиентный спуск. Для этого нам нужен способ замены `x` и `y` на каждой итерации следующим мини-пакетом. Простейший способ предполагает применение *узлов-заполнителей* (`placeholder node`). Такие узлы являются специальными, поскольку они фактически не делают какие-то вычисления, а только выводят данные, которые вы сообщаете им вывести во время выполнения. Обычно они используются для передачи обучающих данных в TensorFlow во время обучения. Если вы не укажете значение во время выполнения для узла-заполнителя, тогда возникнет исключение.

Чтобы создать узел-заполнитель, вы должны вызвать функцию `placeholder()` и указать тип данных выходного тензора. Дополнительно можно также задать его форму, если вы хотите навязать ее. Указание `None` для измерения означает “любой размер”. Например, следующий код создает узел-заполнитель `A` и узел `B = A + 5`. Когда оценивается `B`, мы передаем методу `eval()` экземпляр `feed_dict`, который указывает значение `A`. Обратите внимание, что узел `A` обязан иметь ранг 2 (т.е. быть двумерным) и должен содержать три столбца (иначе сгенерируется исключение), но может включать любое количество строк.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[ 9. 10. 11.]
 [12. 13. 14.]]
```



На самом деле можно передавать вывод *любых* операций, а не только заполнителей. В таком случае библиотека TensorFlow не пытается вычислять эти операции; она применяет значения, которые вы им передаете.

Чтобы реализовать мини-пакетный градиентный спуск, нам необходимо лишь слегка подкорректировать имеющийся код. Первым делом изменим определение *X* и *y* на стадии построения, сделав их узлами-заполнителями:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Затем определим размер пакета и подсчитаем общее количество пакетов:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

Наконец, на стадии выполнения извлечем мини-пакеты один за другим и предоставим значение *X* и *y* через параметр *feed_dict* при оценке узла, который зависит от любого из них.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # загрузка данных с диска
    return X_batch, y_batch

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
```



Передавать значение X и y при оценке θ не требуется ввиду отсутствия зависимости от них.

Сохранение и восстановление моделей

После обучения модели вы должны сохранить ее параметры на диск, чтобы при необходимости позже к ней можно было возвратиться, использовать в другой программе, сравнить с другими моделями и т.д. Кроме того, может быть, вы пожелаете сохранять контрольные точки через регулярные интервалы во время обучения, чтобы в случае аварийного отказа компьютера обучение удалось продолжить с последней контрольной точки, а не начинать снова с нуля.

Библиотека TensorFlow делает сохранение и восстановление модели очень легким. Просто создайте узел `Saver` в конце стадии построения (после того, как созданы все узлы переменных); затем на стадии выполнения вызывайте метод `save()` узла `Saver` всякий раз, когда хотите сохранить модель, передавая ему сеанс и путь к файлу контрольных точек:

```
[...]
theta = tf.Variable(tf.random_uniform([n+1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
        if epoch % 100 == 0: # контрольная точка через каждые 100 эпох
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

            sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Восстанавливать модель в равной степени легко: вы создаете объект `Saver` в конце стадии построения, как и ранее, но затем в начале стадии выполнения вместо инициализации переменных с применением инициализирующего узла вызываете метод `restore()` объекта `Saver`:

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

По умолчанию `Saver` сохраняет и восстанавливает все переменные под их собственными именами, но если вы заинтересованы в большем контроле, тогда можете указать, какие переменные сохранять или восстанавливать и какие имена использовать. Например, следующий объект `Saver` будет сохранять или восстанавливать только переменную `theta` под именем `weights`:

```
saver = tf.train.Saver({"weights": theta})
```

По умолчанию метод `save()` сохраняет структуру графа во втором файле с тем же самым именем и расширением `.meta`. Вы можете загрузить структуру графа с применением метода `tf.train.import_meta_graph()`. Указанный метод добавляет граф к стандартному графу и возвращает экземпляр `Saver`, который можно использовать для восстановления состояния графа (т.е. значений переменных):

```
saver = tf.train.import_meta_graph("/tmp/my_model_final.ckpt.meta")
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```

Это позволяет полностью восстановить сохраненную модель, включая структуру графа и значения переменных, без необходимости в поиске кода, который ее построил.

Визуализация графа и кривых обучения с использованием TensorBoard

Итак, теперь мы имеем вычислительный граф, который обучает линейную регрессионную модель с применением мини-пакетного градиентного спуска, и сохраняем контрольные точки через регулярные интервалы. Звучит замысловато, не так ли? Однако при визуализации хода обучения мы все еще полагаемся на функцию `print()`. Есть более эффективный способ: инструмент `TensorBoard`. Если вы передадите ему статистические данные по обучению, то он отобразит в вашем веб-браузере элегантные интерактивные визуализации статистических данных (скажем, кривые обучения). Вы также можете предоставить инструменту `TensorBoard` определение графа, а он предложит великолепный интерфейс для его просмотра. Это очень удобно для выявления ошибок в графе, поиска узких мест и т.д.

Первый шаг предусматривает небольшую корректировку программы, чтобы она записывала определение графа и статистические данные по обучению — к примеру, ошибку MSE — в журнальный каталог, из которого будет читать TensorBoard. При каждом запуске программы вы должны использовать другой журнальный каталог, иначе TensorBoard будет объединять статистические данные из разных запусков, что испортит визуализации. Простейшим решением является включение отметки времени в имя журнального каталога. Добавьте в начало программы следующий код:

```
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{} /run-{}".format(root_logdir, now)
```

Затем добавьте в самый конец стадии построения такой код:

```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

Первая строка создает в графе узел, который будет подсчитывать значение MSE и записывать его в строку совместимого с TensorBoard двоичного журнала, называемую *сводкой* (*summary*). Вторая строка кода создает объект `FileWriter`, который будет применяться для записывания сводок в журнальные файлы внутри журнального каталога. В первом параметре указывается путь к журнальному каталогу (в данном случае что-то вроде `tf_logs/run-20160906091959/` относительно текущего каталога). Второй (необязательный параметр) — это граф, который вы хотите визуализировать. Новый объект `FileWriter` создает журнальный каталог, если тот еще не существует (и при необходимости его родительские каталоги), и записывает определение графа в двоичный журнальный файл, называемый *файлом событий* (*events file*).

Затем вам понадобится обновить стадию выполнения, чтобы регулярно оценивать узел `mse_summary` во время обучения (например, через каждые 10 мини-пакетов). В результате будет выдана сводка, которую вы сможете записать в файл событий, используя `file_writer`. Вот обновленный код:

```
[...]
for batch_index in range(n_batches):
    X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
    if batch_index % 10 == 0:
        summary_str =
            mse_summary.eval(feed_dict={X: X_batch, y: y_batch})
```

```

    step = epoch * n_batches + batch_index
    file_writer.add_summary(summary_str, step)
    sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
[...]
```



Избегайте регистрации статистических данных по обучению на каждом отдельном шаге обучения, т.к. это значительно замедлит обучение.

Наконец, в конце программы `FileWriter` желательно закрыть:

```
file_writer.close()
```

Теперь запустите программу: она создаст журнальный каталог и запишет в него файл событий, содержащий определение графа и значения MSE. Откройте окно командной оболочки, перейдите в свой рабочий каталог и введите команду `ls -l tf_logs/run*`, чтобы просмотреть содержимое журнального каталога:

```

$ cd $ML_PATH # Ваш рабочий каталог МО (например, $HOME/ml)
$ ls -l tf_logs/run*
total 40
-rw-r--r-- 1 ageron staff 18620 Sep 6 11:10 events.out.
tfevents.1472553182.mymac
```

Запустив программу второй раз, вы должны увидеть в каталоге `tf_logs/` второй каталог:

```

$ ls -l tf_logs/
total 0
drwxr-xr-x 3 ageron staff 102 Sep 6 10:07 run-20160906091959
drwxr-xr-x 3 ageron staff 102 Sep 6 10:22 run-20160906092202
```

Замечательно! Наступило время запустить сервер TensorBoard. Если вы создавали среду `virtualenv`, тогда активируйте ее и затем запустите сервер с помощью команды `tensorboard`, указав ей корневой журнальный каталог. Следующая команда запускает веб-сервер TensorBoard, прослушивающий порт 6006 (перевернутое “goog”):

```

$ source env/bin/activate
$ tensorboard --logdir tf_logs/
Starting TensorBoard on port 6006
(You can navigate to http://0.0.0.0:6006)
Запуск TensorBoard на порте 6006
(Вы можете перейти на http://0.0.0.0:6006)
```

Откройте браузер и перейдите на <http://0.0.0.0:6006/> (или <http://localhost:6006/>). Добро пожаловать в TensorBoard! На вкладке Events (События) вы должны видеть справа элемент MSE. Щелчок на нем приводит к отображению графика MSE во время обучения для обоих запусков (рис. 9.3). Вы можете отмечать, какие запуски хотите видеть, увеличивать или уменьшать масштаб отображения, наводить курсор на кривую для просмотра деталей и т.д.



Рис. 9.3. Визуализация статистических данных по обучению с применением TensorBoard

Щелкните на вкладке Graphs (Графы). Вы должны увидеть граф, показанный на рис. 9.4.

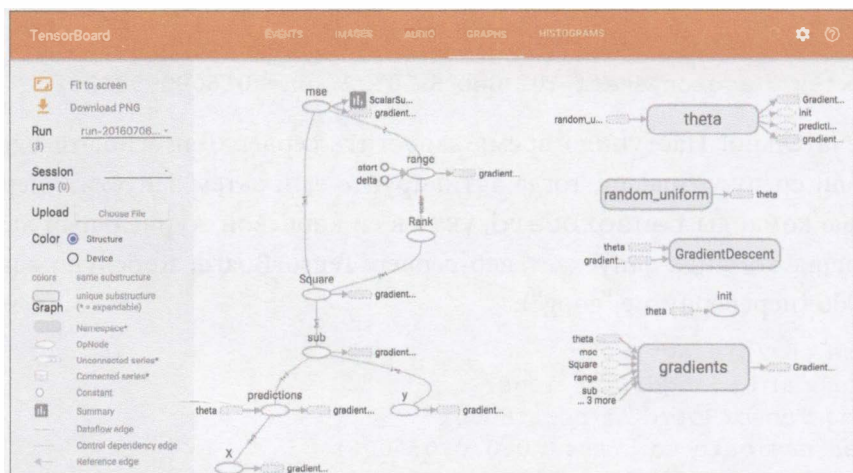


Рис. 9.4. Визуализация графа с использованием TensorBoard

Чтобы уменьшить беспорядок, узлы с множеством *ребер* (т.е. соединений с другими узлами) вынесены во вспомогательную область справа (вы можете перемещать узел между основным графом и вспомогательной областью, щелкая на нем правой кнопкой мыши). По умолчанию некоторые части графа свернуты. Например, попробуйте навести курсор на узел `gradients` и щелкнуть на значке \oplus , чтобы развернуть этот подграф. Затем в развернутом подграфе попытайтесь развернуть подграф `mse_grad`.



Если вы хотите просмотреть граф прямо внутри Jupyter, то можете применять функцию `show_graph()`, доступную в тетради для настоящей главы. Изначально она была написана А. Мордвинцевым в его великолепной учебной тетради по DeepDream (<http://goo.gl/EtCWUc>). Еще один вариант предусматривает установку инструмента отладки TensorFlow, написанного Э. Янгом (<https://github.com/ericjang/tdb>), который включает расширение Jupyter для визуализации графов (и многое другое).

Пространства имен

При работе с более сложными моделями, такими как нейронные сети, граф легко становится загроможденным тысячами узлов. Во избежание этого вы можете создавать *пространства имен* (*name scope*) для группирования связанных узлов. Например, давайте модифицируем предыдущий код, чтобы определить операции `error` и `mse` внутри пространства имени под названием `"loss"`:

```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

Имя каждой операции, определенной внутри пространства, теперь снабжено префиксом `"loss/"`:

```
>>> print(error.op.name)
loss/sub
>>> print(mse.op.name)
loss/mse
```

В TensorBoard узлы `mse` и `error` находятся внутри пространства имени `loss`, которое по умолчанию свернуто (рис. 9.5).

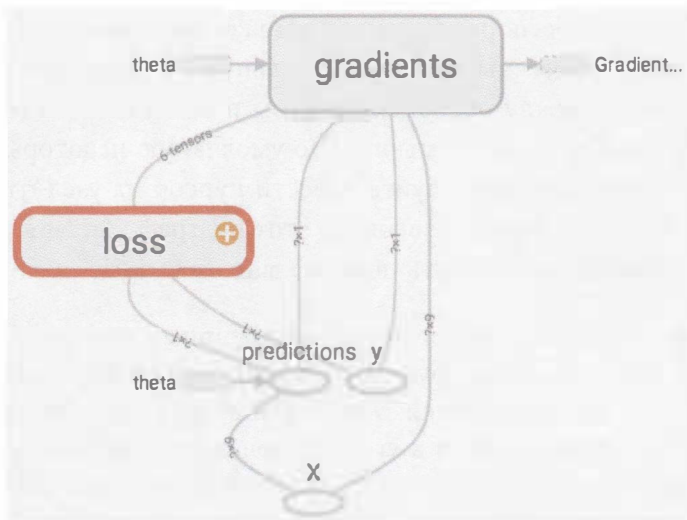


Рис. 9.5. Свернутое пространство имени в TensorBoard

Модульность

Предположим, что вы хотите создать граф, который суммирует выходные данные двух *выпрямленных линейных элементов* (*rectified linear unit* — *ReLU*). Выпрямленный линейный элемент вычисляет линейную функцию входных данных, выдавая результат, если она положительная, и 0 в противном случае (уравнение 9.1).

Уравнение 9.1. Выпрямленный линейный элемент

$$h_{w,b}(X) = \max(X \cdot w + b, 0)$$

Приведенный ниже код выполняет необходимую работу, но имеет довольно много повторяющихся фрагментов:

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")
z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")
relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")
output = tf.add(relu1, relu2, name="output")
```

Такой повторяющийся код труден в сопровождении и подвержен к ошибкам (на самом деле код содержит ошибку, связанную с вырезанием и вставкой; кстати, вы заметили ее?). Ситуация станет еще хуже, если вы пожелаете добавить несколько дополнительных элементов ReLU. К счастью, TensorFlow позволяет придерживаться принципа “не повторяться”: просто создайте функцию для построения ReLU. Показанный далее код создает пять элементов ReLU и выдает их сумму (обратите внимание, что `add_n()` создает операцию, которая будет вычислять сумму списка тензоров):

```
def relu(X):
    w_shape = (int(X.get_shape()[1]), 1)
    w = tf.Variable(tf.random_normal(w_shape), name="weights")
    b = tf.Variable(0.0, name="bias")
    z = tf.add(tf.matmul(X, w), b, name="z")
    return tf.maximum(z, 0., name="relu")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Когда вы создаете узел, TensorFlow проверяет, существует ли уже такое имя, и если оно существует, то дополняет его символом подчеркивания с последующим индексом, чтобы обеспечить уникальность имени. Таким образом, первый элемент ReLU содержит узлы с именами "weights", "bias", "z" и "relu" (плюс многие другие узлы с их стандартными именами, такие как "MatMul").

Второй элемент ReLU содержит узлы с именами "weights_1", "bias_1" и т.д. Третий элемент ReLU содержит узлы с именами "weights_2", "bias_2" и т.д. Библиотека TensorBoard идентифицирует такие последовательности и сворачивает их, уменьшая беспорядок (рис. 9.6).

С использованием пространств имен вы можете сделать граф гораздо яснее. Просто переместите все содержимое функции `relu()` внутрь пространства имени. На рис. 9.7 представлен результирующий граф.

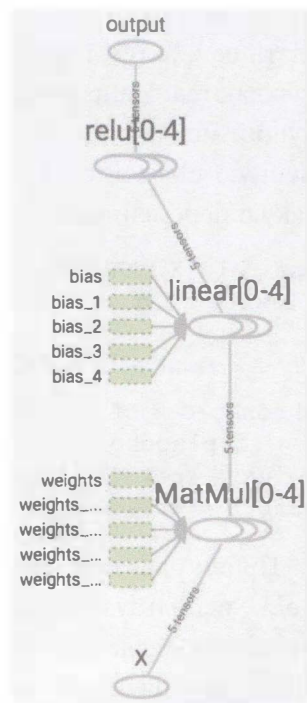


Рис. 9.6. Свёрнутые последовательности узлов

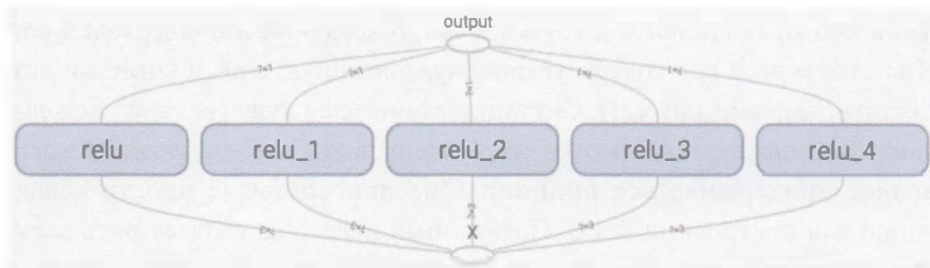


Рис. 9.7. Более ясный граф за счет применения элементов в пространствах имен

Обратите внимание, что TensorFlow также обеспечивает уникальность названий пространств имен, дополняя их суффиксами `_1`, `_2` и т.д.

```
def relu(X):
    with tf.name_scope("relu"):
        [...]
```

Совместное использование переменных

Если вы хотите разделять переменную между разнообразными компонентами графа, то один простой вариант заключается в том, чтобы сначала создать ее и затем передавать в качестве параметра функциям, которые в ней нуждаются. Например, допустим, вы желаете управлять порогом ReLU (в текущий момент жестко закодированным как 0), применяя разделяемую переменную `threshold` для всех элементов ReLU. Вы могли бы просто создать такую переменную и передать ее в функцию `relu()`:

```
def relu(X, threshold):
    with tf.name_scope("relu"):
        [...]
        return tf.maximum(z, threshold, name="max")

threshold = tf.Variable(0.0, name="threshold")
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X, threshold) for i in range(5)]
output = tf.add_n(relus, name="output")
```

Прием работает: теперь вы можете управлять порогом для всех элементов ReLU, используя переменную `threshold`. Тем не менее, при наличии множества разделяемых параметров вроде `threshold` необходимость их постоянной передачи как параметров будет доставлять немало хлопот. Одни разработчики создают словарь Python, содержащий все переменные в модели,

и передают его в каждую функцию. Другие создают класс для каждого модуля (например, класс `ReLU`, в котором переменные класса применяются для поддержки разделяемого параметра). Третьи устанавливают разделяемую переменную в качестве атрибута функции `relu()` при первом вызове:

```
def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        [...]
    return tf.maximum(z, relu.threshold, name="max")
```

Библиотека TensorFlow предлагает еще один вариант, который может привести к более ясному и модульному коду, нежели предшествующие варианты⁵. Поначалу такое решение чуть труднее для понимания, но поскольку оно широко используется в TensorFlow, полезно вникнуть в детали. Идея заключается в том, чтобы применять функцию `get_variable()` для создания разделяемой переменной, если она пока не существует, либо использовать ее повторно, если она уже имеется. Желаемое поведение (создание или повторное применение) управляется атрибутом текущего пространства переменной `variable_scope()`. Скажем, следующий код будет создавать переменную по имени `"relu/threshold"` (в виде скаляра, т.к. `shape=()`, и используя `0.0` для начального значения):

```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(),
                               initializer=tf.constant_initializer(0.0))
```

Обратите внимание, что если переменная уже была создана ранее вызовом `get_variable()`, то данный код сгенерирует исключение. Такое поведение предотвращает повторное применение переменных по ошибке. Чтобы повторно использовать переменную, вы должны явно заявить об этом, устанавливая атрибут `reuse` пространства переменной в `True` (в таком случае указывать форму или инициализатор необязательно):

```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

⁵ Создание класса `ReLU` является, возможно, самым ясным вариантом, но довольно тяжеловесным.

Данный код извлечет существующую переменную `"relu/threshold"` либо сгенерирует исключение, если она не существует или не была создана с применением `get_variable()`. В качестве альтернативы вы можете установить атрибут `reuse` в `True` внутри блока, вызвав метод `reuse_variables()` пространства переменных:

```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```



После того как атрибут `reuse` установлен в `True`, он не может быть установлен обратно в `False` внутри блока. Кроме того, если вы определяете другие пространства переменных внутри данного, то они автоматически унаследуют `reuse=True`. Наконец, повторно использоваться таким способом могут только переменные, созданные посредством `get_variable()`.

Теперь вы располагаете всем необходимым для доступа функции `relu()` к переменной `threshold`, не передавая ее как параметр:

```
def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")          # повторно
                                                           # использовать существующую переменную
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):                          # создать переменную
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
```

Код определяет функцию `relu()`, создает переменную `relu/threshold` (в виде скаляра, который позже будет инициализирован значением `0.0`) и строит пять элементов ReLU, вызывая функцию `relu()`.

Функция `relu()` повторно применяет переменную `relu/threshold` и создает другие узлы ReLU.



Переменные, созданные с использованием `get_variable()`, всегда именуются с применением имени своих пространств переменных в качестве префикса (скажем, `"relu/threshold"`), но для всех остальных узлов (включая переменные, которые созданы с помощью `tf.Variable()`) пространство переменной действует подобно новому пространству имени. В частности, если пространство имени с идентичным названием уже было создано, тогда к нему добавляется суффикс, чтобы обеспечить уникальность. Например, все узлы, созданные в предшествующем коде (кроме переменной `threshold`), имеют имена, предваренные префиксами от `"relu_1/"` до `"relu_5/"` (рис. 9.8).

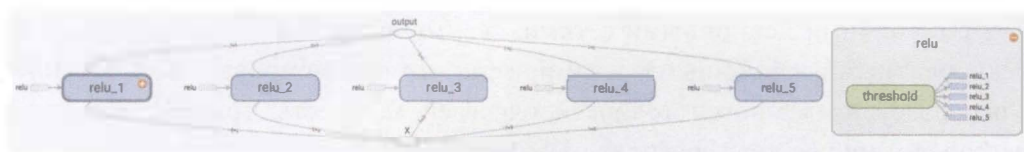


Рис. 9.8. Пять элементов ReLU, разделяющие переменную `threshold`

Заслуживает сожаления тот факт, что переменная `threshold` должна определяться за пределами функции `relu()`, где находится весь остальной код ReLU. Чтобы исправить положение, следующий код создает переменную `threshold` внутри функции `relu()` при ее первом вызове и повторно использует при последующих вызовах. Теперь функция `relu()` не обязана беспокоиться о пространствах имен или разделении переменных: она просто вызывает метод `get_variable()`, который будет создавать или повторно применять переменную `threshold` (ей не нужно знать, что конкретно произойдет). В остатке кода функция `relu()` вызывается пять раз, обеспечивая установку `reuse=False` для первого вызова и `reuse=True` — для всех остальных.

```
def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    [...]
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
reus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        reus.append(relu(X))
output = tf.add_n(reus, name="output")
```

Результирующий граф слегка отличается от предыдущего, потому что разделяемая переменная находится внутри первого элемента ReLU (рис. 9.9).

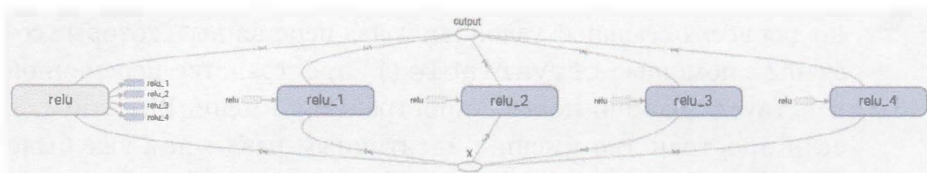


Рис. 9.9. Пять элементов ReLU, разделяющие переменную *threshold*, которая находится внутри первого ReLU

На этом введение в TensorFlow завершено. В последующих главах мы обсудим более сложные темы, в частности, многочисленные операции, связанные с глубокими нейронными сетями, сверточными нейронными сетями и рекуррентными нейронными сетями, а также масштабирование TensorFlow с использованием многопоточности, очередей, множества графических процессоров и множества серверов.

Упражнения

1. Каковы основные преимущества создания вычислительного графа вместо выполнения вычислений напрямую? Каковы главные недостатки?
2. Эквивалентен ли оператор `a_val = a.eval(session=sess)` оператору `a_val = sess.run(a)`?
3. Эквивалентен ли оператор `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` оператору `a_val, b_val = sess.run([a,b])`?
4. Можно ли запускать два вычислительных графа в одном сеансе?
5. Если вы создаете граф `g`, содержащий переменную `w`, затем запускаете два потока и открываете в каждом потоке сеанс с применением того же самого графа `g`, то будет ли каждый сеанс иметь собственную копию переменной `w` или она будет разделяться?
6. Когда переменная инициализируется? Когда она уничтожается?
7. В чем разница между заполнителем и переменной?
8. Что случается, когда вы запускаете граф для оценки операции, которая зависит от заполнителя, но не передаете его значение? Что происходит, если операция не зависит от заполнителя?

9. Можно ли при запуске графа передавать выходное значение любой операции или только значения заполнителей?
10. Как можно установить переменную в любое желаемое значение (во время стадии выполнения)?
11. Сколько раз автоматическое дифференцирование в обратном режиме должно совершить обход графа, чтобы вычислить градиенты функции издержек относительно 10 переменных? Что можно сказать об автоматическом дифференцировании в прямом режиме? А о символическом дифференцировании?
12. Реализуйте логистическую регрессионную модель с мини-пакетным градиентным спуском, используя TensorFlow. Обучите ее и оцените на обучающем наборе `moons` (введенном в главе 5). Попытайтесь добавить все перечисленные ниже украшения.
 - Определите внутри функции `logistic_regression()` граф, который можно легко повторно применять.
 - Используя `Saver`, сохраните контрольные точки через регулярные интервалы во время обучения, а в конце обучения сохраните финальную модель.
 - Восстановите последнюю контрольную точку при запуске, если обучение было прервано.
 - Определите граф с применением пространств имен, чтобы граф хорошо выглядел в TensorBoard.
 - Добавьте сводки для визуализации кривых обучения в TensorBoard.
 - Попробуйте подстроить некоторые гиперпараметры, такие как скорость обучения или размер мини-пакета, и посмотрите на форму кривой обучения.

Решения приведенных упражнений доступны в приложении А.

Введение в искусственные нейронные сети

Птицы вдохновили нас летать, репейники подтолкнули к созданию застежки типа “липучка” и сама природа способствовала многим другим изобретениям. В таком случае при творческих исканиях способов построения интеллектуальной машины вполне логично взглянуть на архитектуру мозга. Это и есть ключевая идея, которая легла в основу *искусственных нейронных сетей* (*Artificial Neural Network* — ANN). Однако хотя появление самолетов было вдохновлено птицами, они вовсе не обязаны махать крыльями. Аналогично сети ANN постепенно становились все больше отличающимися от своих биологических родственников. Некоторые исследователи даже утверждают, что мы вообще должны отказаться от биологической аналогии (например, говорить “единицы” вместо “нейроны”), чтобы не ограничивать наши творческие возможности биологически правдоподобными системами¹.

Сети ANN находятся в самом сердце глубокого обучения. Они являются универсальными, мощными и масштабируемыми, что делает их идеальным вариантом для решения сложных задач МО, таких как классификация миллиардов изображений (например, инструмент Google Картинки), поддержка служб распознавания речи (скажем, Apple’s Siri), рекомендация лучших видеороликов для просмотра сотнями миллионов людей ежедневно (например, YouTube) или обучение с целью победы чемпиона мира в игре го, исследуя миллионы прошедших игр и затем играя против самого себя (DeepMind’s AlphaGo).

В этой главе мы представим искусственные нейронные сети, начав с краткого турне в самые ранние архитектуры ANN. Затем мы рассмотрим *многослойные перцептроны* (*Multi-Layer Perceptron* — MLP) и реализуем

¹ Вы можете извлечь лучшее из обоих миров, если сохраните открытость к вдохновляющим идеям из биологии, но не будете бояться создавать биологически нереалистичные модели при условии их приемлемой работы.

один с использованием TensorFlow для решения задачи классификации цифр MNIST (см. главу 3).

От биологических нейронов к искусственным нейронам

Удивительно, но сети ANN существуют довольно долго: впервые они были введены в далеком 1943 году нейрофизиологом Уорреном Мак-Каллоком и математиком Уолтером Питтсом. В своей знаменательной статье² Мак-Каллок и Питтс представили упрощенную вычислительную модель того, как биологические нейроны могут работать вместе для выполнения сложных вычислений с применением *логики высказываний* (*propositional logic*). Это была первая архитектура искусственной нейронной сети. Как мы увидим, с тех пор было создано много других архитектур.

Первые успехи сетей ANN вплоть до 1960-х годов привели к широко распространенной уверенности в том, что вскоре мы будем общаться с действительно интеллектуальными машинами. Когда стало ясно, что такое обещание неосуществимо (по крайней мере, какое-то время), финансирование улетучилось, и для сетей ANN начался долгий безрадостный период. В начале 1980-х годов интерес к ANN возобновился, поскольку были изобретены новые архитектуры сетей и разработаны лучшие приемы обучения. Но до наступления 1990-х годов большинство исследователей отдавали предпочтение мощным альтернативным приемам МО вроде методов опорных векторов (см. главу 5), т.к. казалось, что они обеспечивали лучшие результаты и обладали более строгими теоретическими основами. Наконец, сейчас мы наблюдаем еще одну волну интереса к сетям ANN. Ослабнет ли эта волна подобно предшествующим? Есть несколько веских причин полагать, что здесь ситуация иная и она окажет гораздо более сильное влияние на нашу жизнь.

- В настоящее время доступно гигантское количество данных для обучения нейронных сетей, и в случае очень крупных и сложных задач сети ANN часто превосходят другие методики МО.
- Потрясающий рост вычислительной мощности по сравнению с 1990-х годами делает возможным обучение больших нейронных сетей за приемлемое время. Отчасти это связано с законом Мура, но также благо-

² “A Logical Calculus of Ideas Immanent in Nervous Activity” (“Логическое исчисление идей, присущих нервной деятельности”), У. Мак-Каллок и У. Питтс (1943 год) (<https://goo.gl/U14mxW>).

даря индустрии компьютерных игр, которая выпускала мощные графические процессоры миллионами единиц.

- Алгоритмы обучения были усовершенствованы. Справедливости ради следует отметить, что они лишь незначительно отличаются от алгоритмов, используемых в 1990-х годах, но внесенные относительно небольшие корректировки вызвали сильное положительное воздействие.
- Некоторые теоретические ограничения сетей ANN на практике оказались благоприятными. Например, многие считали, что алгоритмы обучения ANN были обречены, поскольку они, скорее всего, застревали бы в локальных оптимумах, но выяснилось, что на деле такая ситуация возникает довольно редко (или если случается, то обычно достаточно близко к глобальному оптимуму).
- Похоже, сети ANN вошли в эффективный виток финансирования и развития. Ошеломительные продукты, основанные на сетях ANN, регулярно освещаются в прессе, что притягивает все большее и большее внимание, а также инвестиции в них, приводя к большему развитию и даже более изумительным продуктам.

Биологические нейроны

Прежде чем обсуждать искусственные нейроны, давайте кратко рассмотрим, что собой представляет биологический нейрон (рис. 10.1). Эта необычно выглядящая клетка, встречающаяся главным образом в коре головного мозга животных (скажем, в вашем мозге), состоит из *тела клетки* (*cell body*), содержащего ядро и большинство сложных компонентов клетки, а также многие ветвящиеся удлинения, называемые *дендритами* (*dendrite*), плюс одно очень крупное удлинение, называемое *аксоном* (*axon*). Длина аксона может быть больше тела клетки как в несколько раз, так и в десятки тысяч раз. Поблизости к концу аксон расщепляется на множество ветвей, называемых *телодендронами* (*telodendria*), а на кончике этих ветвей располагаются очень маленькие структуры, называемые *синаптическими окончаниями* (*synaptic terminal*) или просто *синапсами* (*synapse*), которые соединяются с дендритами (или прямо с телом клетки) других нейронов. Через такие синапсы биологические нейроны принимают от других нейронов короткие электрические импульсы, называемые *сигналами* (*signal*). Когда в пределах нескольких миллисекунд нейрон получает достаточное количество сигналов от других нейронов, он выдает собственные сигналы.

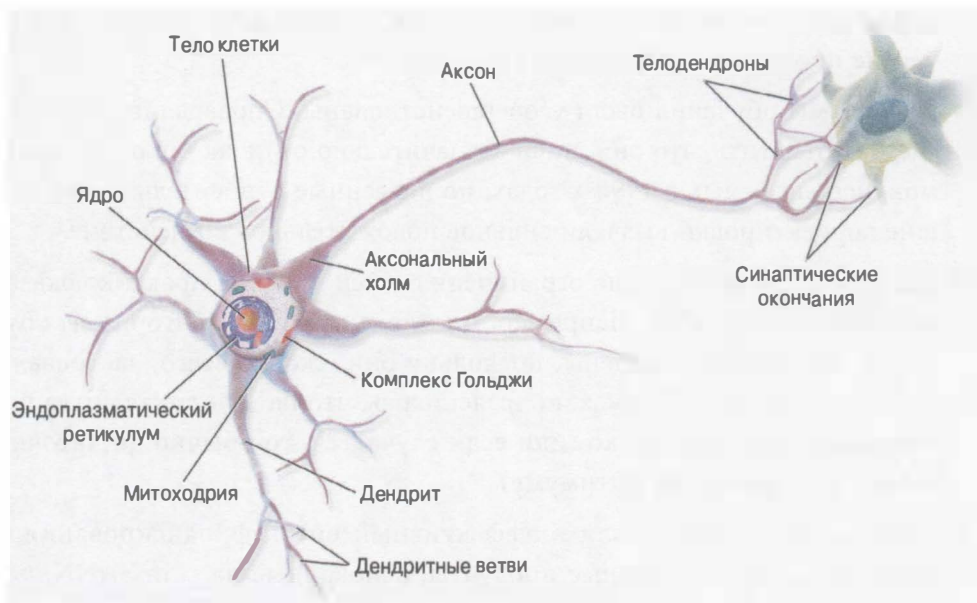


Рис. 10.1. Биологический нейрон³

Таким образом, похоже, что отдельные биологические нейроны ведут себя довольно просто, но они организованы в обширную сеть из миллиардов нейронов, каждый из которых обычно соединяется с тысячами других нейронов. С помощью обширной сети довольно простых нейронов можно выполнять очень сложные вычисления подобно тому, как из объединенных усилий простых муравьев может появиться сложный муравейник. Архитектура *биологических нейронных сетей* (*Biological Neural Networks — BNN*)⁴ все еще является объектом активных исследований, но некоторые части мозга были отображены, и кажется, что нейроны часто организованы в последовательные слои (рис. 10.2).

Логические вычисления с помощью нейронов

Уоррен Мак-Каллок и Уолтер Питтс предложили очень простую модель биологического нейрона, которая позже стала известной как *искусственный нейрон* (*artificial neuron*): он имеет один или большее количество двоичных (включено/выключено) входов и один двоичный выход.

³ Рисунок Брюса Блауса (Creative Commons 3.0 (<https://creativecommons.org/licenses/by/3.0/>)). Воспроизведен из <https://en.wikipedia.org/wiki/Neuron>.

⁴ В контексте машинного обучения выражение “нейронные сети” в большинстве случаев относится к сетям ANN, а не BNN.

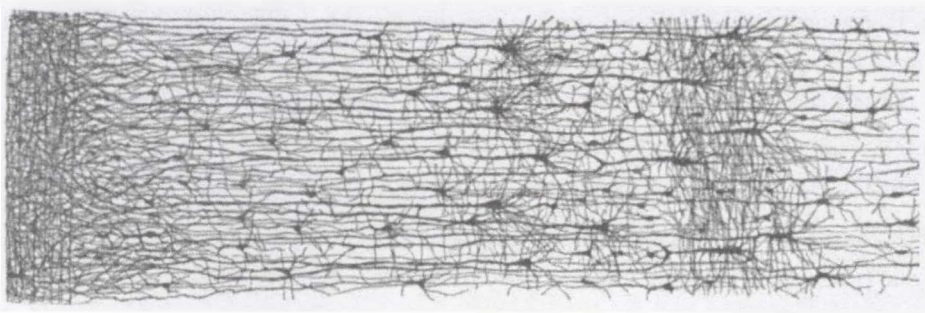


Рис. 10.2. Множество слоев в биологической нейронной сети
(кора головного мозга человека)⁵

Искусственный нейрон просто активирует свой выход, когда активно больше, чем определенное число его входов. Мак-Каллок и Питтс показали, что даже посредством такой упрощенной модели можно построить сеть искусственных нейронов, которая вычисляет любое желаемое логическое утверждение. Например, давайте построим несколько сетей ANN, которые выполняют разнообразные логические вычисления (рис. 10.3), исходя из предположения, что нейрон активируется, когда активны хотя бы два его входа.

- Первая сеть слева — это просто функция тождественного отображения: если нейрон A активирован, тогда нейрон C также активируется (т.к. он получает два входных сигнала от нейрона A), но если нейрон A отключен, то нейрон C также отключен.

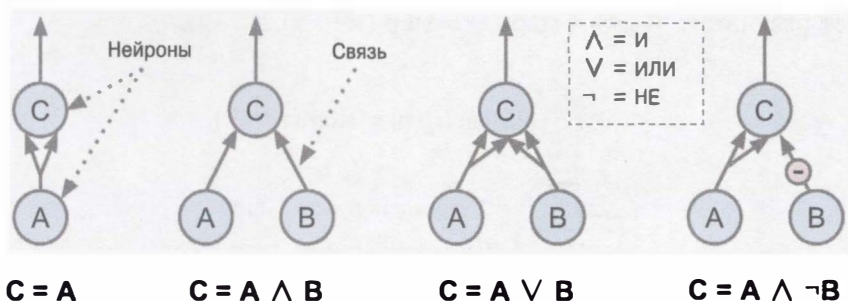


Рис. 10.3. Сети ANN, выполняющие простые логические вычисления

⁵ Изображение расслоения коры головного мозга Сантьяго Рамон-и-Кахаля (публичная собственность). Воспроизведено из https://en.wikipedia.org/wiki/Cerebral_cortex.

- Вторая сеть выполняет операцию логического “И”: нейрон С активируется, только когда активированы и нейрон А, и нейрон В (для активации нейрона С одиночного входного сигнала недостаточно).
- Третья сеть выполняет операцию логического “ИЛИ”: нейрон С активируется, если активирован либо нейрон А, либо нейрон В (или оба).
- Наконец, если мы предположим, что входная связь способна подавлять активность нейрона (как в случае биологических нейронов), тогда четвертая сеть вычисляет чуть более сложное логическое утверждение: нейрон С активируется, только если нейрон А активен, а нейрон В отключен. Если нейрон А активен все время, тогда получается операция логического “НЕ”: нейрон С активен, когда нейрон В отключен, и наоборот.

Вы легко можете представить, каким образом такие сети можно комбинировать для вычисления сложных логических выражений (взгляните на упражнения в конце главы).

Перцептрон

Перцептрон (*perceptron*) — одна из простейших архитектур ANN, придуманная Фрэнком Розенблаттом в 1957 году. Она основана на несколько отличающемся искусственном нейроне (рис. 10.4), который называется *линейным пороговым элементом* (*Linear Threshold Unit — LTU*): теперь входы и выход представляют собой числа (а не двоичные значения “включено/выключено”), вдобавок с каждой входной связью ассоциирован вес. Элемент LTU вычисляет взвешенную сумму своих входов ($z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = \mathbf{w}^T \cdot \mathbf{x}$), затем применяет к полученной сумме *ступенчатую функцию* (*step function*) и выдает результат: $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z) = \text{step}(\mathbf{w}^T \cdot \mathbf{x})$.

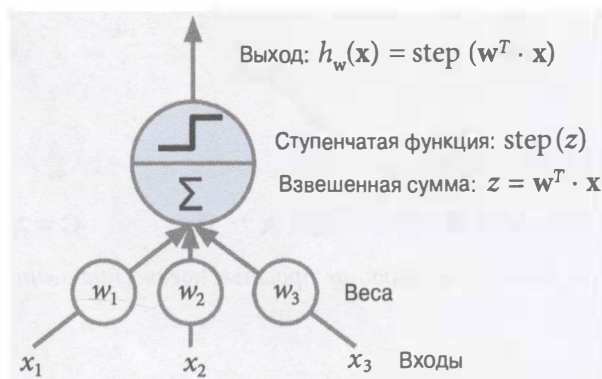


Рис. 10.4. Линейный пороговый элемент

Самой распространенной ступенчатой функцией, используемой в перцептронах, является *ступенчатая функция Хевисайда (Heaviside step function)*, приведенная в уравнении 10.1. Иногда вместо нее применяется функция знака (или *сигнум-функция; sign function*).

Уравнение 10.1. Распространенная ступенчатая функция, применяемая в перцептронах

$$\text{heaviside}(z) = \begin{cases} 0, & \text{если } z < 0 \\ 1, & \text{если } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1, & \text{если } z < 0 \\ 0, & \text{если } z = 0 \\ +1, & \text{если } z > 0 \end{cases}$$

Одиночный элемент LTU может использоваться для простой линейной двоичной классификации. Он рассчитывает линейную комбинацию входов и если результат превышает некоторый порог, то выдает положительный класс, а иначе — отрицательный (подобно классификатору на основе логистической регрессии или линейному методу опорных векторов).

Например, вы могли бы применять одиночный элемент LTU для классификации цветков ириса на базе длины и ширины лепестков (также добавляя дополнительный признак смещения $x_0 = 1$, как мы делали в предшествующих главах). Обучение элемента LTU означает нахождение правильных значений для w_0 , w_1 и w_2 (алгоритм обучения мы вскоре обсудим).

Перцептрон состоит из единственного слоя элементов LTU⁶, причем каждый нейрон соединяется со всеми входами. Такие связи часто представляются с использованием специальных сквозных нейронов, называемых *входными нейронами (input neuron)*: они просто передают на выход все, что получают на входе. Кроме того, как правило, добавляется дополнительный признак смещения ($x_0 = 1$). Этот признак смещения обычно представляется с применением нейрона специального типа, называемого *нейроном смещения (bias neuron)*, который просто все время выдает 1.

На рис. 10.5 показан перцептрон с двумя входами и тремя выходами. Такой перцептрон способен классифицировать образцы одновременно в три разных двоичных класса, что делает его многовыходовым классификатором.

⁶ Название “перцептрон” иногда используется для обозначения крошечной сети с единственным элементом LTU.

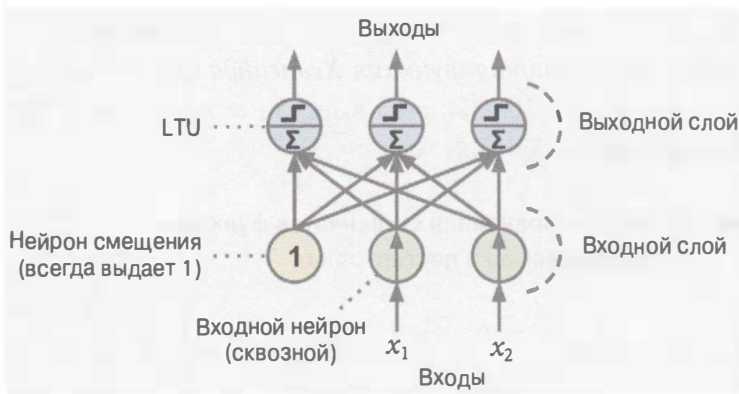


Рис. 10.5. Диаграмма персептрона

Итак, каким же образом обучается персептрон? Алгоритм обучения персептрона, предложенный Фрэнком Розенблаттом, был в значительной степени навеян **правилом Хебба (Hebb's rule)**. В своей книге “The Organization of Behavior” (“Организация поведения”), опубликованной в 1949 году, Дональд Хебб предположил, что когда биологический нейрон часто вызывает срабатывание другого нейрона, то связь между этими двумя нейронами усиливается. Позже идея была резюмирована Зигридом Левелем в его легко запоминающейся фразе: “Клетки, которые срабатывают вместе, связаны вместе” (“Cells that fire together, wire together”). Впоследствии правило стало известным под названием **правило Хебба (или обучение по Хеббу (Hebbian learning))**; т.е. вес связи между двумя нейронами увеличивается всякий раз, когда на своих выходах они выдают одно и то же. Персептроны обучаются с применением варианта этого правила, в котором во внимание принимается ошибка, допущенная сетью; он не укрепляет связи, которые ведут к неправильному выходу. Говоря точнее, персептрону передается один обучающий образец за раз, и для каждого образца он вырабатывает свои прогнозы. Для каждого выходного нейрона, который выдает неправильный прогноз, увеличиваются веса связей из входов, которые способствовали корректному прогнозу. Правило описано в уравнении 10.2.

Уравнение 10.2. Правило обучения персептрона (обновление весов)

$$w_{i,j} \text{ (следующий шаг)} = w_{i,j} + \eta (y_j - \hat{y}_j) x_i$$

- $w_{i,j}$ — вес связи между i -тым входным нейроном и j -тым выходным нейроном.

- x_i — i -тое входное значение текущего обучающего образца.
- \hat{y}_j — выход j -того выходного нейрона для текущего обучающего образца.
- y_j — целевой выход j -того выходного нейрона для текущего обучающего образца.
- η — скорость обучения.

Граница решений каждого выходного нейрона линейна, так что персептроны неспособны к обучению на сложных паттернах (подобно классификаторам на основе логистической регрессии). Тем не менее, если обучающие образцы являются линейно сепарабельными, то Розенблатт демонстрирует, что алгоритм будет сходиться в решение⁷. Это называется *теоремой о сходимости персептрона* (*perceptron convergence theorem*).

Библиотека Scikit-Learn предоставляет класс `Perceptron`, который реализует сеть с одним элементом LTU. Он может использоваться практически так, как вы могли ожидать — скажем, с набором данных `iris` (введенным в главе 4):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris()
X = iris.data[:, (2, 3)]           # длина лепестка, ширина лепестка
y = (iris.target == 0).astype(np.int) # ирис щетинистый?
per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

y_pred = per_clf.predict([[2, 0.5]])
```

Вы можете обнаружить, что алгоритм обучения персептрона сильно напоминает стохастический градиентный спуск. На самом деле употребление класса `Perceptron` из Scikit-Learn эквивалентно применению класса `SGDClassifier` со следующими гиперпараметрами: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (скорость обучения) и `penalty=None` (без регуляризации).

⁷ Обратите внимание, что решение в целом не уникально: в общем случае, когда данные линейно сепарабельны, существует бесконечное число гиперплоскостей, которые могут их разделять.

Обратите внимание, что в противоположность классификаторам на основе логистической регрессии персептроны не выдают вероятность класса; взамен они только вырабатывают прогнозы, базируясь на жестком пороге. Это одна из веских причин отдать предпочтение логистической регрессии перед персептронами.

В своей монографии *Perceptrons* (*Персептроны*), вышедшей в 1969 году, Марвин Мински и Сеймур Пейперт выделили несколько серьезных недостатков персептронов, в частности тот факт, что они неспособны решить некоторые тривиальные задачи (скажем, задачу классификации на основе *исключающего "ИЛИ"* (*Exclusive OR — XOR*), изображенную слева на рис. 10.6). Разумеется, это также справедливо в отношении любой другой модели линейной классификации (вроде классификаторов на основе логистической регрессии), но исследователи ожидали от персептронов намного большего, потому их разочарование было огромным: в результате многие исследователи вообще перестали заниматься *коннекционизмом* (*connectionism*), т.е. изучением нейронных сетей, отдав предпочтение высокоуровневым задачам, таким как логика, решение и поиск.

Однако выяснилось, что некоторые ограничения персептронов можно устранить за счет укладывания друг на друга множества персептронов. Результирующая сеть ANN называется *многослойным персептроном* (*Multi-Layer Perceptron — MLP*). В частности, MLP способен решать задачу XOR, в чем вы можете удостовериться путем вычисления выхода MLP, представленного справа на рис. 10.6, для каждой комбинации входов: для входов (0, 0) или (1, 1) сеть выдает 0, а для входов (0, 1) или (1, 0) — 1.

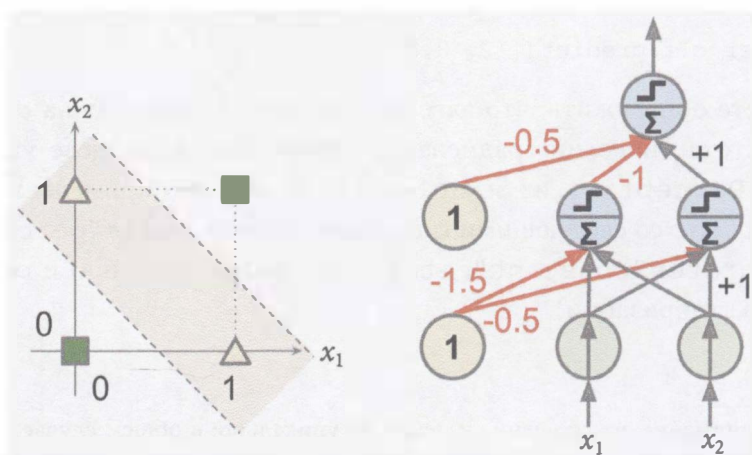


Рис. 10.6. Задача классификации XOR и решающий ее многослойный персептрон

Многослойный персептрон и обратная связь

Многослойный персептрон (рис. 10.7) состоит из одного (сквозного) входного слоя, одного или большего числа слоев элементов LTU, называемых *скрытыми слоями* (*hidden layer*), и еще одного слоя элементов LTU, который называется *выходным слоем* (*output layer*). Каждый слой кроме выходного включает нейрон смещения и полностью связан со следующим слоем. Когда сеть ANN имеет два и более скрытых слоя, она называется *глубокой нейронной сетью* (*Deep Neural Network — DNN*).

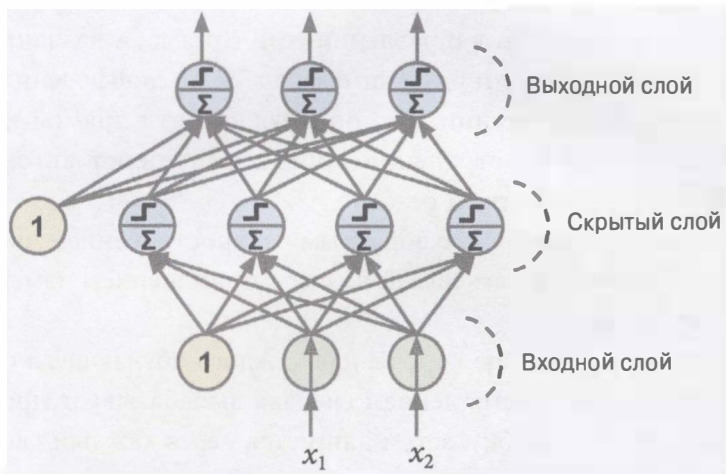


Рис. 10.7. Многослойный персептрон

В течение многих лет исследователи безуспешно пытались найти способ обучения многослойных персептронов. Наконец, в 1986 году Д.Э. Румельхарт и др. опубликовали революционную статью⁸, представляющую алгоритм обучения с *обратным распространением* (*backpropagation training algorithm*; backpropagation — обратное распространение ошибки обучения)⁹. В наши дни мы можем описать его как градиентный спуск, использующий автоматическое дифференцирование в обратном режиме (градиентный спуск обсуждался в главе 4, а автоматическое дифференцирование — в главе 9).

⁸ “Learning Internal Representations by Error Propagation” (“Изучение внутренних представлений путем распространения ошибки”), Д. Румельхарт, Г. Хинтон, Р. Уильямс (1986 год) (<https://goo.gl/Wl7Xyc>).

⁹ В действительности этот алгоритм был придуман несколько раз разными исследователями в различных областях, начиная с П. Уэрбоса в 1974 году.

Алгоритм передает сети каждый обучающий образец и вычисляет выход каждого нейрона в каждом следующем друг за другом слое (это прямой проход, как и при выработке прогнозов). Затем он измеряет выходную ошибку сети (т.е. разницу между желаемым и действительным выходом сети) и подсчитывает вклад каждого нейрона из последнего скрытого слоя в ошибку каждого выходного нейрона.

Далее алгоритм продолжает измерять, сколько таких вкладов в ошибку поступает от каждого нейрона в предыдущем скрытом слое, и делает это до тех пор, пока не достигнет входного слоя. Такой обратный проход эффективно измеряет градиент ошибок по всем весам связей в сети, распространяя градиент ошибок в обратном направлении сети (отсюда и название алгоритма). Если вы просмотрите алгоритм автоматического дифференцирования в обратном режиме в приложении Г, то обнаружите, что прямой и обратный проходы обратного распространения просто выполняют автоматическое дифференцирование в обратном режиме.

Последним шагом алгоритма с обратным распространением является градиентный спуск по всем весам связей в сети с применением измеренных ранее градиентов ошибок.

Давайте сформулируем еще короче: для каждого обучающего образца алгоритм с обратным распространением сначала вырабатывает прогноз (прямой проход), измеряет ошибку, затем двигается через каждый слой в обратном направлении, чтобы измерить вклад в ошибку каждой связи (обратный проход), и в заключение немного подстраивает веса связей с целью уменьшения ошибки (шаг градиентного спуска).

Для надлежащей работы такого алгоритма авторы внесли ключевое изменение в архитектуру MLP: они заменили ступенчатую функцию логистической функцией, $\sigma(z) = 1 / (1 + \exp(-z))$. Это было существенным, поскольку ступенчатая функция содержит только плоские сегменты, так что градиенты для работы отсутствуют (градиентный спуск не в состоянии перемещаться по плоской поверхности), в то время как логистическая функция везде имеет четко определенную ненулевую производную, позволяя градиентному спуску продвигаться на каждом шаге.

Алгоритм с обратным распространением может использоваться с другими *функциями активации* (*activation function*), а не только с логистической функцией.

Ниже описаны еще две популярных функции активации.

- **Функция гиперболического тангенса** $\tanh(z) = 2\sigma(2z) - 1$.

Подобно логистической функции она является S-образной, непрерывной и дифференцируемой, но ее выходное значение находится в диапазоне от -1 до 1 (а не от 0 до 1 , как в случае логистической функции), что имеет тенденцию делать выход каждого слоя более или менее нормализованным (т.е. центрированным относительно 0) в начале обучения. Это часто помогает ускорить сходимость.

- **Функция ReLU**, введенная в главе 9, $\text{ReLU}(z) = \max(0, z)$.

Она непрерывная, но, к сожалению, не дифференцируемая в точке $z=0$ (наклон резко меняется, что может привести к скачкам градиентного спуска поблизости данной точки). Тем не менее, на практике эта функция работает очень хорошо и обладает преимуществом более быстрого вычисления. Кроме того, тот факт, что она не имеет максимального выходного значения, также помогает ослабить ряд проблем во время градиентного спуска (в главе 11 мы еще к этому вернемся).

Перечисленные популярные функции активации и их производные представлены на рис. 10.8.

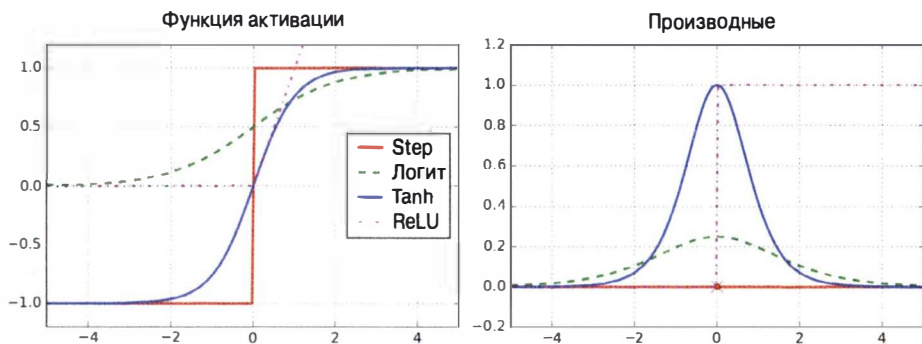


Рис. 10.8. Функции активации и их производные

Многослойный перцептрон часто применяется для классификации, причем каждый выход соответствует отдельному двоичному классу (скажем, спам/не спам, срочный/не срочный и т.д.). Когда классы исключают (например, классы от 0 до 9 для классификации изображений цифр), то выходной слой обычно модифицируется путем замены индивидуальных функций активации разделяемой **многопеременной** (*softmax*) функцией (рис. 10.9).

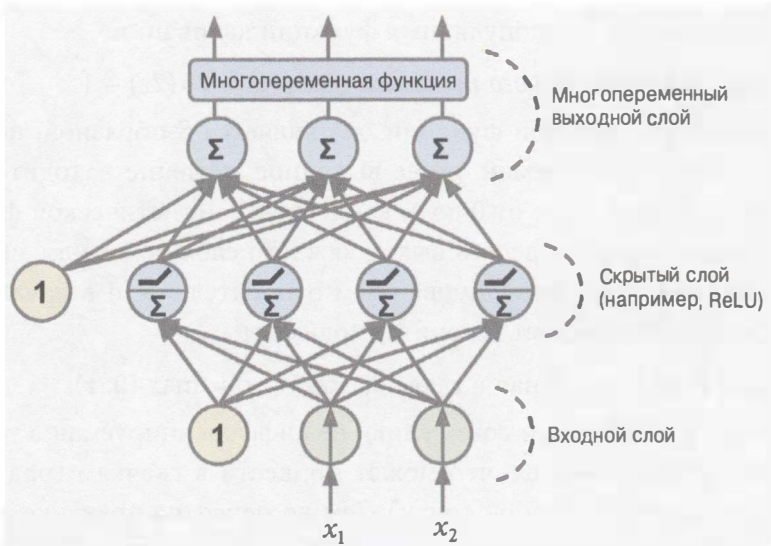


Рис. 10.9. Современный многослойный персептрон (включающий ReLU и многопеременную функцию), предназначенный для классификации

Многопеременная функция была введена в главе 4. Выход каждого нейрона соответствует оценочной вероятности надлежащего класса. Обратите внимание, что сигнал протекает только в одном направлении (от входов к выходам), поэтому такая архитектура является примером *нейронной сети прямого распространения* (*Feedforward Neural Network — FNN*).



По-видимому, биологические нейроны реализуют приблизительно сигмоидальную (S-образную) функцию активации, поэтому исследователи придерживались сигмоидальных функций в течение очень долгого времени. Но оказывается, что в сетях ANN функция активации ReLU в целом работает лучше. Это один из случаев, когда биологическая аналогия ввела в заблуждение.

Обучение многослойного персептрона с помощью высокоуровневого API-интерфейса TensorFlow

Простейший способ обучить MLP посредством TensorFlow предусматривает использование высокоуровневого API-интерфейса `TF.Learn`, совместимого с библиотекой `Scikit-Learn`. Класс `DNNClassifier` позволяет довольно легко обучить глубокую нейронную сеть с любым количеством скрытых слоев и многопеременным выходным слоем для выдачи оценочных вероят-

ностей классов. Например, следующий код обучает классификационную сеть DNN с двумя скрытыми слоями (один из 300 нейронов и еще один из 100 нейронов) и многопеременным выходным слоем из 10 нейронов:

```
import tensorflow as tf

feature_cols =
    tf.contrib.learn.infer_real_valued_columns_from_input(X_train)
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100],
                                         n_classes=10, feature_columns=feature_cols)

dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # если TensorFlow >= 1.1
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

Код сначала создает набор столбцов вещественных значений из обучающего набора (доступны и другие типы столбцов вроде категориальных столбцов). Затем создается экземпляр `DNNClassifier`, который помещается внутрь вспомогательного экземпляра, обеспечивающего совместимость с библиотекой Scikit-Learn. Наконец, прогоняются 40 000 итераций обучения с применением пакетов из 50 образцов.

Если вы запустите этот код на наборе данных MNIST (после его масштабирования, скажем, с использованием класса `StandardScaler` из Scikit-Learn), тогда фактически получите модель, которая достигает примерно 98.2%-ной правильности на испытательном наборе! Она превосходит наилучшую модель, которая обучалась в главе 3:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = dnn_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred['classes'])
0.982500000000000004
```



Пакет `tensorflow.contrib` содержит много полезных функций, но представляет собой место для хранения экспериментального кода, который пока еще не созрел для того, чтобы стать частью основного API-интерфейса TensorFlow. Таким образом, в будущем класс `DNNClassifier` (и любой другой код в пакете `contrib`) может измениться без какого-либо уведомления.

Внутренне класс `DNNClassifier` создает все слои нейронов, основываясь на функции активации ReLU (мы можем изменить функцию активации путем установки гиперпараметра `activation_fn`). Выходной слой опирается на многопеременную функцию, а функцией издержек является перекрестная энтропия (см. главу 4).

Обучение глубокой нейронной сети с использованием только TensorFlow

Если вам нужен больший контроль над архитектурой сети, тогда вы можете отдать предпочтение низкоуровневому API-интерфейсу для Python в TensorFlow (см. главу 9). В настоящем разделе с применением этого API-интерфейса мы построим ту же модель, что и ранее, и реализуем мини-пакетный градиентный спуск для ее обучения на наборе данных MNIST. Первый шаг — стадия построения, на которой создается граф TensorFlow. Второй шаг — стадия выполнения, где граф действительно прогоняется для обучения модели.

Стадия построения

Итак, начнем. Прежде всего, нам необходимо импортировать библиотеку `tensorflow`. Затем понадобится указать количество входов и выходов, а также установить количество скрытых нейронов внутри каждого слоя:

```
import tensorflow as tf

n_inputs = 28*28 # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Затем, как и в главе 9, вы можете использовать узлы-заполнители для представления обучающих данных и целей. Форма узла `X` определена только частично. Нам известно, что он будет двумерным тензором (т.е. матрицей) с образцами по первому измерению и признаками по второму измерению, а также, что количество признаков составит 28×28 (один признак на пиксель), но мы пока не знаем, сколько образцов будет содержать каждый обучающий пакет. Значит, формой `X` является `(None, n_inputs)`. Подобным образом нам известно, что узел `y` будет одномерным тензором с одним элементом на образец, но в этот момент мы опять не знаем размер обучающего пакета, а потому формой является `(None)`.

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

А теперь создадим нейронную сеть. Узел-заполнитель `X` будет действовать как входной слой; во время стадии выполнения он будет заменяться одним обучающим пакетом за раз (обратите внимание, что все образцы в обучаю-

щем пакете будут обрабатываться нейронной сетью одновременно). Далее необходимо создать два скрытых слоя и выходной слой. Два скрытых слоя почти идентичны: они отличаются только связанными входами и количеством содержащихся нейронов. Выходной слой также очень похож, но вместо функции активации ReLU он применяет многопеременную функцию активации. Таким образом, мы напишем функцию `neuron_layer()`, которая будет использоваться для создания одного слоя за раз. Ей понадобятся параметры для указания входов, количества нейронов, функции активации и имени слоя:

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs + n_neurons)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

Разберем приведенный код строка за строкой.

1. Сначала с применением имени слоя мы создаем пространство имени: оно будет содержать все вычислительные узлы для данного слоя нейронов. Это необязательно, но граф будет гораздо лучше выглядеть в TensorBoard, если узлы хорошо организованы.
2. Затем мы получаем количество входов, просматривая форму входной матрицы и извлекая размер второго измерения (первое измерение предназначено для образцов).
3. Следующие три строки создают переменную `W`, которая будет хранить матрицу весов (часто называемую *ядром* (*kernel*) слоя). Она будет двумерным тензором, содержащим веса всех связей между каждым входом и каждым нейроном; таким образом, ее форма имеет вид `(n_inputs, n_neurons)`. Переменная `W` будет инициализироваться случайно с использованием усеченного¹⁰ нормального (гауссова) распределения со стан-

¹⁰ Использование усеченного нормального распределения вместо обычного нормального распределения гарантирует отсутствие крупных весов, которые могли бы замедлить обучение.

дартным отклонением $2/\sqrt{n_{\text{inputs}} + n_{\text{neurons}}}$. Применение такого специфического стандартного отклонения помогает алгоритму сходиться намного быстрее (мы обсудим данную тему более подробно в главе 11; это одна из тех небольших подстроек нейронных сетей, которые оказали громадное воздействие на их эффективность). Важно инициализировать веса связей для всех скрытых слоев случайным образом, чтобы избежать любых симметрий, которые алгоритм градиентного спуска не смог бы преодолеть¹¹.

4. Следующая строка создает переменную `b` для смещений, инициализированную значением 0 (проблема симметрии в этом случае не возникает), с одним параметром смещения на нейрон.
5. Далее мы создаем подграф для вычисления $Z = X \cdot W + b$. Такая векторизованная реализация будет эффективно подсчитывать взвешенные суммы входов плюс член смещения для каждого нейрона в слое и для всех образцов в пакете за одно действие. Обратите внимание, что добавление одномерного массива (`b`) к двумерной матрице с тем же самым количеством столбцов ($X \cdot W$) в результате приводит к добавлению одномерного массива к каждой строке матрицы: это называется *ретрансляцией* (*broadcasting*).
6. Наконец, если предоставлен параметр `activation`, такой как `tf.nn.relu` (т.е. $\max(0, Z)$), то код возвращает `activation(Z)`, а иначе `Z`.

Хорошо, теперь у нас есть подходящая функция для создания слоя нейронов. Давайте воспользуемся ею для создания глубокой нейронной сети! Первый скрытый слой на входе принимает `X`, а второй скрытый слой — выход первого скрытого слоя. Выходной слой на входе принимает выход второго скрытого слоя.

```
with tf.name_scope("dnn") :  
    hidden1 = neuron_layer(X, n_hidden1, name="hidden1",  
                           activation=tf.nn.relu)
```

¹¹ Например, если вы установите все веса в 0, тогда все нейроны будут выдавать 0, и градиент ошибок окажется одинаковым для всех нейронов в заданном скрытом слое. Затем шаг градиентного спуска обновит все веса в каждом слое совершенно одинаково, так что все они останутся равными. Другими словами, несмотря на наличие сотен нейронов на слой, ваша модель будет себя вести так, как будто бы есть только один нейрон на слой. Она не собирается развиваться.

```
hidden2 = neuron_layer(hidden1, n_hidden2, name="hidden2",
                        activation=tf.nn.relu)
logits = neuron_layer(hidden2, n_outputs, name="outputs")
```

Обратите внимание, что мы снова ради ясности применяем пространство имени. Кроме того, имейте в виду, что `logits` представляет собой выход нейронной сети *до* прохождения многопеременной функции активации: по причинам, связанным с оптимизацией, мы будем выполнять многопеременный расчет позже.

Как вы и могли ожидать, TensorFlow поступает с множеством удобных функций, предназначенных для создания слоев нейронных сетей, поэтому зачастую нет нужды определять собственную функцию `neuron_layer()` вроде определенной только что. Например, функция `tf.layers.dense()` из TensorFlow (ранее называемая `tf.contrib.layers.fully_connected()`) создает полносвязный слой, где все входы связаны со всеми нейронами, находящимися внутри слоя. Она позаботится о создании переменных весов и смещений, именуемых `kernel` и `bias` соответственно, с использованием подходящей стратегии инициализации, а с применением аргумента `activation` может быть установлена функция активации. В главе 11 вы увидите, что функция `dense()` также поддерживает параметры регуляризации. Давайте скорректируем предыдущий код с целью использования функции `dense()` вместо `neuron_layer()`. Просто заменим раздел создания `dnn` следующим кодом:

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                             activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                             activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Теперь, когда мы располагаем готовой моделью нейронной сети, необходимо определить функцию издержек, которая будет применяться при обучении модели. Подобно тому, как делалось для многопеременной логистической регрессии в главе 4, мы будем использовать перекрестную энтропию. Ранее уже упоминалось, что перекрестная энтропия будет штрафовать модели, которые подсчитывают низкую вероятность для целевого класса. Библиотека TensorFlow предоставляет несколько функций для вычисления перекрестной энтропии. Мы будем применять функцию `sparse_softmax_cross_entropy_with_logits()`: она подсчитывает перекрестную энтропию,

основываясь на “логитах” (т.е. выход сети *до* прохождения через многопеременную функцию активации), и ожидает метки в форме целых чисел из диапазона от 0 до количества классов минус 1 (в нашем случае от 0 до 9). В результате мы получим одномерный тензор, содержащий перекрестную энтропию для каждого экземпляра. Затем мы можем использовать функцию `reduce_mean()` из TensorFlow для вычисления средней перекрестной энтропии по всем образцам.

```
with tf.name_scope("loss") :  
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(  
                                                labels=y, logits=logits)  
    loss = tf.reduce_mean(xentropy, name="loss")
```



Вызов функции `sparse_softmax_cross_entropy_with_logits()` является эквивалентом применения многопеременной функции активации и последующего расчета перекрестной энтропии, но она более эффективна и прекрасно справляется с граничными случаями: когда логиты большие, ошибки округления чисел с плавающей точкой могут стать причиной того, что многопеременный выход окажется равным в точности 0 или 1, и тогда уравнение перекрестной энтропии будет содержать член $\log(0)$, равный отрицательной бесконечности. Функция `sparse_softmax_cross_entropy_with_logits()` решает эту проблему, вычисляя взамен $\log(\epsilon)$, где ϵ — крошечное положительное число. Вот почему мы не применяли многопеременную функцию активации ранее. Существует также еще одна функция по имени `softmax_cross_entropy_with_logits()`, которая принимает метки в форме унитарно кодированных векторов (вместо целых чисел от 0 до количества классов минус 1).

Мы имеем модель нейронной сети, а также функцию издержек, и теперь должны определить оптимизатор `GradientDescentOptimizer`, который будет подстраивать параметры модели для сведения к минимуму функции издержек. Ничего нового; что-то похожее мы делали в главе 9:

```
learning_rate = 0.01  
with tf.name_scope("train") :  
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)  
    training_op = optimizer.minimize(loss)
```

Последний важный шаг на стадии построения связан с указанием, каким образом оценивать модель. В качестве меры производительности мы будем использовать правильность. Первым делом для каждого образца выясним, корректен ли прогноз сети, путем проверки, соответствует или нет самый высокий логит целевому классу. Для этого можно задействовать функцию `in_top_k()`. Она возвращает одномерный тензор, наполненный булевскими значениями, а потому нам понадобится привести такие булевские значения к значениям с плавающей точкой и затем подсчитать среднее. Это даст общую правильность сети.

```
with tf.name_scope("eval"):  
    correct = tf.nn.in_top_k(logits, y, 1)  
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Вдобавок, как обычно, нам необходимо создать узел для инициализации всех переменных, и мы также создадим экземпляр `Saver`, чтобы сохранить параметры обученной модели на диск:

```
init = tf.global_variables_initializer()  
saver = tf.train.Saver()
```

Уф! Стадия построения закончена. Всего получилось меньше 40 строк кода, но они были довольно насыщенными: мы создали заполнители для входов и целей, написали функцию для построения слоя нейронов, применили ее для создания сети DNN, определили функцию издержек, создали оптимизатор и в заключение определили меру производительности. Наступило время стадии выполнения.

Стадия выполнения

Стадия выполнения намного короче и проще. Прежде всего, загрузим набор данных MNIST. Мы могли бы использовать для этого библиотеку Scikit-Learn, как поступали в предшествующих главах, но TensorFlow предлагает собственный вспомогательный класс, который извлекает данные, масштабирует их (между 0 и 1), тасует и предоставляет простую функцию для загрузки по одному мини-пакету за раз. Сверх того данные уже расщеплены на обучающий набор (55 000 образцов), проверочный набор (5 000 образцов) и испытательный набор (10 000 образцов). Задействуем упомянутый вспомогательный класс:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
```

Далее мы определяем количество эпох, которые нужно прогнать, а также размер мини-пакетов:

```
n_epochs = 40
batch_size = 50
```

А теперь мы можем обучить модель:

```
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
            acc_val = accuracy.eval(feed_dict={X: mnist.validation.images,
                                                y: mnist.validation.labels})
            print(epoch, "Правильность при обучении:", acc_train,
                  "Правильность при проверке:", acc_val)
    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Код открывает сеанс TensorFlow, который запускает узел `init`, инициализирующий все переменные. Затем он начинает главный цикл обучения: в каждой эпохе код выполняет итерацию по количеству мини-пакетов, которое соответствует размеру обучающего набора. Каждый мини-пакет извлекается через метод `next_batch()`, после чего код просто выполняет операцию обучения, передавая ей текущий мини-пакет входных данных и цели. В конце каждой эпохи код оценивает модель на последнем мини-пакете и на полном обучающем наборе и выдает результат. В заключение параметры модели сохраняются на диск.

Использование нейронной сети

Теперь, когда нейронная сеть обучена, вы можете ее применять для выработки прогнозов. Для этого можно повторно использовать ту же самую стадию построения, но изменить стадию выполнения примерно так:

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = [...] # ряд новых изображений
                        # (масштабированных между 0 и 1)
```

```
Z = logits.eval(feed_dict={X: X_new_scaled})  
y_pred = np.argmax(Z, axis=1)
```

Код сначала загружает параметры модели из диска. Затем он загружает ряд новых изображений, которые вы хотите классифицировать. Не забудьте применить то же самое масштабирование признаков, как и для обучающих данных (в этом случае от 0 до 1). Далее код оценивает узел `logits`. Если вы желаете знать все оценочные вероятности классов, тогда должны применить функцию `softmax()` к логитам, но если вы хотите выработать прогноз класса, то можете просто выбрать класс, который имеет самое высокое значение логита (цель достигается с использованием функции `argmax()`).

Точная настройка гиперпараметров нейронной сети

Гибкость нейронных сетей является также и одним из главных недостатков: существует много гиперпараметров для подстройки. Мало того, что можно применять любую воображимую *топологию сети* (способ связывания нейронов друг с другом), но даже в простом многослойном персептроне допускается изменять число слоев, количество нейронов на слой, тип функции активации, используемой в каждом слое, логику инициализации весов и многое другое. Как узнать, какая комбинация гиперпараметров будет наилучшей для имеющейся задачи?

Разумеется, для нахождения правильных значений гиперпараметров вы можете применять решетчатый поиск с перекрестной проверкой, как делалось в предшествующих главах, но поскольку гиперпараметров, подлежащих настройке, очень много, а обучение нейронной сети на крупном наборе данных занимает долгое время, за приемлемый промежуток времени вам удастся исследовать лишь крошечную часть пространства гиперпараметров. Гораздо лучше воспользоваться рандомизированным поиском (<https://goo.gl/QFjMKu>), как обсуждалось в главе 2.

Другой вариант предусматривает применение инструмента наподобие Oscar (<http://oscar.calldesk.ai/>), который реализует более сложные алгоритмы, чтобы помочь вам быстро отыскать хороший набор значений гиперпараметров.

Это помогает получить представление о том, какие значения будут подходящими для гиперпараметров, чтобы можно было ограничить пространство поиска. Давайте начнем с количества скрытых слоев.

При решении многих задач вы можете начать с единственного скрытого слоя и получить приемлемые результаты. В действительности было показано, что многослойный персептрон с только одним скрытым слоем способен моделировать даже самые сложные функции при условии, что он имеет достаточное число нейронов. В течение долгого времени такое положение дел убеждало исследователей, что нет никакой нужды изучать более глубокие нейронные сети. Но исследователи упускали из виду тот факт, что глубокие сети имеют гораздо более высокую *эффективность параметров*, чем сети с меньшей глубиной: они могут моделировать сложные функции с использованием экспоненциально меньшего количества нейронов, чем неглубокие сети, намного ускоряя их обучение.

Чтобы понять причину, представьте, что вам предложено нарисовать лес с применением какого-то программного обеспечения для рисования, но использовать копирование/вставку запрещено. Вам придется рисовать каждое дерево по отдельности, ветвь за ветвью, листик за листиком. Если бы взамен вы могли нарисовать один листик, скопировать/вставить его для рисования ветви, затем скопировать/вставить готовую ветвь для рисования дерева и в заключение скопировать/вставить полученное дерево для образования леса, то закончили бы работу в кратчайший срок.

Реальные данные часто структурированы в соответствие с какой-то иерархией, и глубокие нейронные сети автоматически извлекают преимущество из этого факта: скрытые слои, расположенные в самом низу, моделируют низкоуровневые структуры (например, линейные сегменты разнообразных форм и ориентаций), промежуточные скрытые слои объединяют такие низкоуровневые структуры для моделирования структур промежуточных уровней (скажем, квадратов и окружностей), а высокоуровневые скрытые слои объединяют промежуточные структуры для моделирования высокоуровневых структур (например, лиц).

Иерархическая архитектура не только помогает сетям DNN быстрее сходиться в хорошее решение, она также улучшает их способность к обобщению на новые наборы данных. Скажем, если модель уже обучена распознавать лица на фотографиях и теперь ее необходимо обучить распознаванию причесок, тогда вы можете ускорить обучение, повторно задействовав нижние слои первой сети. Вместо инициализации весов и смещений первых нескольких слоев в новой нейронной сети случайным образом вы можете инициализи-

ровать их значениями весов и смещений нижних слоев первой сети. В таком случае сети не придется обучаться с нуля всем низкоуровневым структурам, которые встречаются на большинстве фотографий; она должна будет обучиться только структурам более высоких уровней (например, прическам).

Итак, для многих задач вы можете начать с одного или двух скрытых слоев, и они будут прекрасно работать. (Скажем, на наборе данных MNIST можно довольно легко достичь правильности свыше 97%, используя только один скрытый слой с несколькими сотнями нейронов, и свыше 98%, применяя два скрытых слоя с тем же самым общим количеством нейронов, приблизительно за одинаковое время обучения.) Для более сложных задач вы можете постепенно увеличивать число слоев до тех пор, пока не начнется переобучение обучающим набором. Очень сложные задачи, такие как классификация крупных изображений или распознавание речи, обычно требуют сетей с десятками слоев (либо даже с сотнями, но не полностью связанных слоев, что будет показано в главе 13), и они нуждаются в обучающих данных огромных объемов. Однако обучать такие сети с нуля придется редко: гораздо чаще будут повторно использоваться части заранее обученной современной сети, которая выполняет похожую задачу. Обучение пройдет намного быстрее и потребует меньшего объема данных (мы обсудим это в главе 11).

Количество нейронов на скрытый слой

Очевидно, что количество нейронов внутри входного и выходного слоев определяется требуемым для задачи типом входа и выхода. Например, задача с набором данных MNIST требует $28 \times 28 = 784$ входных нейронов и 10 выходных нейронов. Устоявшаяся практика в отношении скрытых слоев предусматривает установление их размеров для образования воронки, с уменьшением числа нейронов на каждом слое — логическое обоснование того, что многие низкоуровневые средства могут объединяться в намного меньшее количество высокоуровневых средств. Скажем, типовая нейронная сеть для MNIST может иметь два скрытых слоя, первый из которых содержит 300 нейронов, а второй — 100 нейронов. Тем не менее, теперь подобная практика не настолько распространена, и вы можете просто применять для всех скрытых слоев один и тот же размер — например, 150 нейронов: есть лишь один гиперпараметр, подлежащий настройке, вместо одного гиперпараметра на слой. Как и для количества слоев, вы можете попробовать постепенно увеличивать число нейронов до тех пор, пока не начнется переобучение

сети. В общем случае вы получите лучшее соотношение затрат и прибыли, увеличивая количество слоев, а не число нейронов на слой. К сожалению, вы увидите, что нахождение идеального количества нейронов по-прежнему является в какой-то мере “черной магией”.

Более простой подход предполагает выбор модели с большим числом слоев и нейронов, чем фактически необходимо, и раннее прекращение в целях предотвращения ее переобучения (а также использование других приемов регуляризации, особенно *отключения (dropout)*, как будет показано в главе 11). Это было скопировано с подхода “растягивающихся брюк”¹²: вместо того, чтобы тратить время на поиск штанов, идеально подходящих вам по размеру, просто возьмите растягивающиеся штаны, которые будут ужиматься до правильного размера.

Функции активации

В большинстве случаев в скрытых слоях вы можете применять функцию активации ReLU (или один из ее вариантов, как демонстрируется в главе 11). Она немного быстрее в вычислении, чем другие функции активации, и благодаря тому факту, что функция не насыщается крупными входными значениями, градиентный спуск не застревает настолько сильно на плато (в противоположность логистической функции или функции гиперболического тангенса, которая насыщается при 1).

Для выходного слоя многопеременная функция активации, как правило, является хорошим вариантом для задач классификации с взаимно исключающими классами. Когда они не взаимно исключающие (или классов только два), вы обычно хотите использовать логистическую функцию. В случае задач регрессии вы можете вообще не применять функций активации для выходного слоя.

На этом введение в искусственные нейронные сети завершено. В последующих главах мы обсудим приемы обучения очень глубоких сетей и распределения обучения среди множества серверов и графических процессоров. Затем мы исследуем ряд других популярных архитектур нейронных сетей: сверточные нейронные сети, рекуррентные нейронные сети и автокодировщики¹³.

¹² От Винсента Ванхаука в его учебном курсе по глубокому обучению (<https://goo.gl/Y5TFqz>) на [Udacity.com](https://www.udacity.com).

¹³ В приложении Д представлено несколько дополнительных архитектур искусственных нейронных сетей.

Упражнения

1. Используя исходные искусственные нейроны (вроде показанных на рис. 10.3), изобразите сеть ANN, которая вычисляет $A \oplus B$ (где \oplus представляет операцию исключающего “ИЛИ”).

Подсказка: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.

2. Почему обычно предпочтительнее применять классификатор на основе логистической регрессии, а не классический персептрон (т.е. единственный слой линейных пороговых элементов, обученных с использованием алгоритма обучения персептронов)? Каким образом вы могли бы подстроить персептрон, чтобы сделать его эквивалентом классификатора на основе логистической регрессии?
3. Почему логистическая функция активации является ключевым ингредиентом при обучении первых многослойных персептронов?
4. Назовите три популярных функции активации. Можете ли вы их представить?
5. Предположим, что у вас есть многослойный персептрон, состоящий из одного входного слоя с 10 сквозными нейронами, за которым следует один скрытый слой с 50 искусственными нейронами и один выходной слой с 3 искусственными нейронами. Все искусственные нейроны применяют функцию активации ReLU.
 - Какова форма входной матрицы X ?
 - Что можно сказать о форме вектора весов W_h скрытого слоя и форме его вектора смещений b_h ?
 - Какова форма вектора весов W_o выходного слоя и форма его вектора смещений b_o ?
 - Какова форма выходной матрицы Y сети?
 - Напишите уравнение, которое вычисляет выходную матрицу Y сети как функцию X , W_h , b_h , W_o и b_o .
6. Сколько нейронов в выходном слое вам понадобится, если нужно классифицировать почтовые сообщения на спам и не спам? Какую функцию активации вы должны использовать в выходном слое? Если взамен вы захотите взяться за набор данных MNIST, то сколько нейронов потребуется иметь в выходном слое и какую функцию активации применять?

Дайте ответы на аналогичные вопросы, когда необходимо настроить сеть на выработку прогнозов цен на дома (см. главу 2).

7. Что такое обратное распространение и как оно работает? В чем разница между обратным распространением и автоматическим дифференцированием в обратном режиме?
8. Можете ли вы перечислить все гиперпараметры, которые допускают подстройку в многослойном персептроне? Если многослойный персептрон переобучается обучающими данными, тогда каким образом вы могли бы подстроить эти гиперпараметры, чтобы попытаться устранить проблему?
9. Обучите глубокий многослойный персептрон на наборе данных MNIST и посмотрите, можете ли вы получить точность выше 98%. Как и в последнем упражнении главы 9, попробуйте добавить все украшения (т.е. сохранение контрольных точек, восстановление последней контрольной точки в случае прерывания, отображение сводок, вычерчивание кривых обучения с использованием TensorBoard и т.д.).

Решения приведенных упражнений доступны в приложении А.

Обучение глубоких нейронных сетей

В главе 10 мы представили искусственные нейронные сети и обучили первую глубокую нейронную сеть. Но мы имели дело с совсем мелкой сетью DNN, имеющей всего лишь два скрытых слоя. Что, если вам нужно заняться очень сложной задачей, такой как обнаружение сотен типов объектов в изображениях с высоким разрешением? Вам может понадобиться обучить гораздо более глубокую сеть DNN, вероятно с (предположим) 10 слоями, каждый из которых содержит сотни нейронов, соединенных сотнями тысяч связей. Это не будет простой прогулкой по парку.

- Во-первых, вы столкнетесь с коварной проблемой *исчезновения градиентов* (*vanishing gradients*) или связанной с ней проблемой *взрывного роста градиентов* (*exploding gradients*), которая оказывает воздействие на глубокие нейронные сети и делает низкоуровневые слои весьма трудными в обучении.
- Во-вторых, обучение такой крупной сети может стать крайне медленным.
- В-третьих, модель с миллионами параметров подвержена высокому риску переобучения на обучающем наборе.

В настоящей главе мы по очереди разберем каждую из упомянутых проблем и покажем приемы для их разрешения. Мы начнем с объяснения проблемы исчезновения градиентов и исследуем некоторые из самых популярных ее решений. Затем мы рассмотрим разнообразные оптимизаторы, которые могут чрезвычайно ускорить обучение крупных моделей в сравнении с обыкновенным градиентным спуском. Наконец, мы обратимся к нескольким популярным приемам регуляризации для больших нейронных сетей.

С помощью таких инструментов вы будете в состоянии обучать очень глубокие сети: добро пожаловать в глубокое обучение!

Проблемы исчезновения и взрывного роста градиентов

Как обсуждалось в главе 10, алгоритм с обратным распространением работает, двигаясь от выходного слоя к входному слою и попутно распространяя градиент ошибок. После того, как алгоритм вычислил градиент функции издержек относительно каждого параметра в сети, он использует эти градиенты для обновления каждого параметра посредством шага градиентного спуска.

К сожалению, с продвижением алгоритма к более низким слоям градиенты зачастую становятся все меньше и меньше. В результате градиентный спуск оставляет веса связей нижних слоев практически неизменными, а обучение никогда не сходится в хорошее решение. Это называется проблемой *исчезновения градиентов*. В ряде случаев может произойти противоположное: градиенты способны расти все больше и больше, из-за чего многие слои получают безумно высокие обновления весов и алгоритм расходится. Мы имеем проблему *взрывного роста градиентов*, которая главным образом встречается в рекуррентных нейронных сетях (глава 14). В целом глубокие нейронные сети страдают от изменчивых градиентов; разные слои могут обучаться с широко варьирующимися скоростями.

Несмотря на то что такое неподходящее поведение эмпирически наблюдалось довольно долго (и было одной из причин, по которым от нейронных сетей обычно отказывались на протяжении длительного времени), только примерно в 2010 году был совершен значительный прогресс в его понимании. В работе Ксавье Глоро и Йошуа Бенджи под названием “Understanding the Difficulty of Training Deep Feedforward Neural Networks” (“Осмысление сложности обучения глубоких нейронных сетей прямого распространения”)¹ обнаружился ряд подозреваемых виновников. В их число входило сочетание популярной логистической сигмоидальной функции активации и приема инициализации весов, который на то время был самым широко распространенным — случайной инициализации с применением нормального распределения со средним 0 и стандартным отклонением 1. Выражаясь кратко, авторы статьи показали, что с такой функцией активации и схемой инициализации дисперсия выходов каждого слоя намного больше дисперсии его

¹ “Understanding the Difficulty of Training Deep Feedforward Neural Networks” (“Осмысление сложности обучения глубоких нейронных сетей прямого распространения”), К. Глоро и Й. Бенджи (2010 год) (<http://goo.gl/1rhAef>).

входов. Продвигаясь вперед по сети, дисперсия продолжает увеличиваться после каждого слоя до тех пор, пока функция активации не насыщается на верхних слоях. В действительности ситуация усугубляется тем фактом, что логистическая функция имеет среднее 0.5, а не 0 (функция гиперболического тангенса имеет среднее 0 и ведет себя в глубоких сетях чуть лучше, чем логистическая функция).

Взглянув на логистическую функцию активации (рис. 11.1), вы можете заметить, что когда входы становятся большими (отрицательными или положительными), функция насыщается в точке 0 или 1 с производной, предельно близкой к 0. Таким образом, когда происходит обратное распространение, то у него практически нет градиента для передачи обратно через сеть, и существующий небольшой градиент продолжает понижаться с продвижением обратного распространения по верхним слоям, так что для нижних слоев фактически ничего не остается.

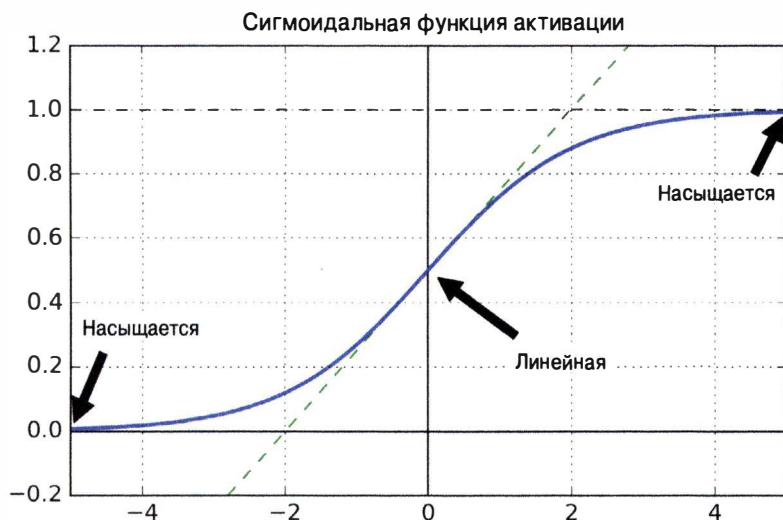


Рис. 11.1. Насыщение логистической функции активации

Инициализация Ксавье и Хе

Глоро и Бенджи в своей статье предложили способ значительного смягчения проблемы. Нам нужно, чтобы сигнал должным образом протекал в обоих направлениях: в прямом направлении при выработывании прогнозов и в обратном направлении при обратном распространении градиентов. Мы не хотим, чтобы сигнал затухал или взрывообразно возрастал и насыщался.

Авторы доказали, что для надлежащего протекания сигнала необходимо, чтобы дисперсия выходов каждого слоя была равна дисперсии его входов², и также нужно, чтобы градиенты имели равную дисперсию до и после протекания через слой в обратном направлении (в статье приведены математические детали). На самом деле гарантировать соблюдение обоих условий невозможно, если только речь не идет о слое с одинаковым количеством входных и выходных связей, но авторы представили хороший компромисс, который доказал свою эффективность на практике: веса связей должны быть инициализированы случайным образом, как описано в уравнении 11.1, где $n_{\text{входов}}$ и $n_{\text{выходов}}$ — количество входных и выходных связей для слоя, чьи веса инициализируются (также называемые *коэффициентом разветвления по входу (fan-in)* и *коэффициентом разветвления по выходу (fan-out)*). Такую стратегию инициализации часто называют *инициализацией Ксавье* (по имени ее автора) или иногда *инициализацией Глоро*.

Уравнение 11.1. Инициализация Ксавье (при использовании логистической функции активации)

Нормальное распределение со средним 0 и стандартным отклонением

$$\sigma = \sqrt{\frac{2}{n_{\text{входов}} + n_{\text{выходов}}}}.$$

Или равномерное распределение между $-r$ и $+r$, где $r = \sqrt{\frac{6}{n_{\text{входов}} + n_{\text{выходов}}}}$.

Когда количество входных связей приблизительно равно количеству выходных связей, получаются более простые уравнения³ (например, $\sigma = 1/\sqrt{n_{\text{входов}}}$ или $r = \sqrt{3}/\sqrt{n_{\text{входов}}}$).

Применение стратегии инициализации Ксавье может существенно ускорить обучение, и она является одним из трюков, которые привели к текуще-

² Рассмотрим аналогию: если вы установите регулятор микрофонного усилителя слишком близко к нулю, то люди не услышат вашего голоса, но в случае установки регулятора чересчур близко к максимуму ваш голос будет насыщаться и люди не смогут понять, что вы говорите. Теперь вообразите цепочку таких усилителей: их регуляторы должны быть надлежаще установлены, чтобы в конце цепочки ваш голос раздавался громко и четко. Ваш голос обязан выходить из каждого усилителя с той же самой амплитудой, с которой он входил.

³ В действительность эта упрощенная стратегия была предложена намного раньше — скажем, в книге 1998 года “Neural Networks: Tricks of the Trade” (“Нейронные сети: специфические приемы”) Женевьевы Опп и Клауса-Роберта Мюллера (Springer).

му успеху глубокого обучения. В нескольких более поздних работах⁴ представлены похожие стратегии для разных функций активации (табл. 11.1). Стратегию инициализации для функции активации ReLU (и ее разновидностей, включая активацию ELU, которая вскоре будет описана) временами называют *инициализацией Хе* (по фамилии ее автора). Именно такую стратегию мы использовали в главе 10.

Таблица 11.1. Параметры инициализации для функций активации разных типов

Функция активации	Равномерное распределение $[-r, r]$	Нормальное распределение
Логистическая	$r = \sqrt{\frac{6}{n_{\text{входов}} + n_{\text{выходов}}}}$	$\sigma = \sqrt{\frac{2}{n_{\text{входов}} + n_{\text{выходов}}}}$
Гиперболического тангенса	$r = 4\sqrt{\frac{6}{n_{\text{входов}} + n_{\text{выходов}}}}$	$\sigma = 4\sqrt{\frac{2}{n_{\text{входов}} + n_{\text{выходов}}}}$
ReLU (и разновидности)	$r = \sqrt{2}\sqrt{\frac{6}{n_{\text{входов}} + n_{\text{выходов}}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{\text{входов}} + n_{\text{выходов}}}}$

По умолчанию функция `tf.layers.dense()`, введенная в главе 10, применяет инициализацию Ксавье (с равномерным распределением). Вы можете использовать взамен инициализацию Хе, применяя функцию `variance_scaling_initializer()` следующим образом:

```
he_init = tf.contrib.layers.variance_scaling_initializer()
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                           kernel_initializer=he_init, name="hidden1")
```



Инициализация Хе учитывает только коэффициент разветвления по входу, а не среднее между коэффициентом разветвления по входу и коэффициентом разветвления по выходу, как в инициализации Ксавье. По умолчанию так принято и для функции `variance_scaling_initializer()`, но вы можете изменить это, установив аргумент `mode="FAN_AVG"`.

⁴ К примеру, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification” (“Погружение в выпрямители: превышение человеческой эффективности при классификации ImageNet”), К. Хе и др. (2015 год) (<http://goo.gl/VHP3pB>).

Ненасыщаемые функции активации

Одна из идей, выдвинутых Глоро и Бенджи в своей статье 2010 года, заключалась в том, что проблемы исчезновения/взрывного роста градиентов возникали отчасти из-за неудачно выбранной функции активации. До тех пор большинство людей предполагали, что если уж мать-природа избрала приблизительно сигмоидальные функции активации для использования в биологических нейронах, то они обязаны быть превосходным вариантом. Но выяснилось, что в глубоких нейронных сетях гораздо лучше ведут себя другие функции активации, в частности функция активации ReLU. Это объясняется главным образом тем, что она не насыщается для положительных значений (а также довольно высокой скоростью вычисления).

К сожалению, функция активации ReLU не идеальна. Она страдает от проблемы, известной как *угасающие элементы ReLU* (*dying ReLU*): во время обучения некоторые нейроны фактически отмирают, приводя к тому, что они перестают выдавать что-либо, отличающееся от 0. В ряде случаев вы можете обнаружить, что половина нейронов вашей сети погибла, особенно когда применяется большая скорость обучения. Если во время обучения веса нейрона обновляются так, что взвешенная сумма входов нейрона является отрицательной, тогда он начинает выдавать 0. После того как это произошло, возвращение к жизни нейрона маловероятно, поскольку градиент функции ReLU равен 0, когда его вход отрицателен.

Чтобы решить описанную проблему, вы можете воспользоваться разновидностью функции ReLU наподобие *ReLU с утечкой* (*leaky ReLU*). Такая функция определяется как $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$ и показана на рис. 11.2.

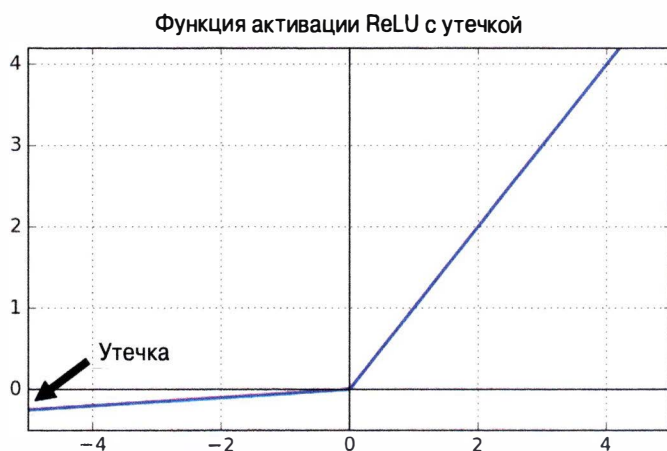


Рис. 11.2. ReLU с утечкой

Гиперпараметр α задает размер “утечки” функции: это наклон функции для $z < 0$, и он обычно устанавливается в 0.01. Такой небольшой наклон гарантирует, что ReLU с утечкой никогда не угасает; нейроны могут впасть в длительную кому, но у них есть шанс со временем очнуться. В недавно опубликованной статье⁵ сравнивались разновидности функции активации ReLU, и один из выводов заключался в том, что варианты с утечкой всегда превосходят строгую функцию активации ReLU.

На самом деле, как представляется, установка $\alpha = 0.2$ (огромная утечка) дает лучшую производительность, чем $\alpha = 0.01$ (небольшая утечка). Авторы также провели оценку *рандомизированного ReLU с утечкой* (*randomized leaky ReLU — RReLU*), где α выбирается случайно в заданном диапазоне во время обучения и фиксируется на среднем значении во время испытаний. Он также выполняется достаточно хорошо и, кажется, действует в качестве регуляризатора (снижая риск переобучения обучающим набором).

Наконец, авторы оценили *параметрический ReLU с утечкой* (*parametric leaky ReLU — PReLU*), где α разрешено учиться во время обучения (вместо того, чтобы быть гиперпараметром, α становится параметром, который может быть модифицирован обратным распространением как любой другой параметр). Это говорило о сильном превосходстве ReLU с утечкой над ReLU на крупных наборах данных с изображениями, но на меньших наборах данных возникает риск переобучения обучающим набором.

Последнее, но не менее важное: в статье 2015 года⁶, написанной Джорком-Арне Клевером и др., была предложена новая функция активации, названная *экспоненциальным линейным элементом* (*exponential linear unit — ELU*), которая в проведенных авторами экспериментах превзошла все разновидности ReLU: время обучения сокращалось, а нейронная сеть работала лучше на испытательном наборе. Эта функция изображена на рис. 11.3, а в уравнении 11.2 приведено ее определение.

⁵ “Empirical Evaluation of Rectified Activations in Convolution Network” (“Эмпирическая оценка выпрямленных активаций в сверточной сети”), Б. Ксу и др. (2015 год) (<https://goo.gl/BlxhKn>).

⁶ “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)” (“Быстрое и точное обучение глубоких сетей с помощью экспоненциальных линейных элементов (ELU)”), Д. Клевер, Т. Антерсинер, С. Хохрайтер (2015 год) (<http://goo.gl/Sdl2P7>).

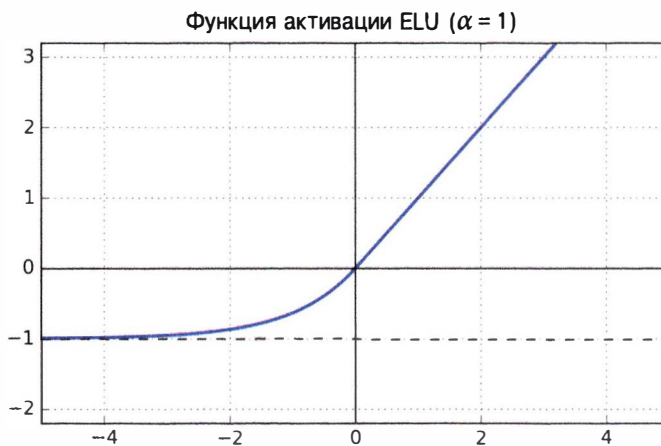


Рис. 11.3. Функция активации ELU

Уравнение 11.2. Функция активации ELU

$$\text{ELU}_{\alpha}(z) = \begin{cases} \alpha (\exp(z) - 1), & \text{если } z < 0 \\ z, & \text{если } z \geq 0 \end{cases}$$

Функция активации ELU во многом похожа на функцию ReLU, но обладает несколькими важными отличиями.

- Во-первых, она принимает отрицательные значения, когда $z < 0$, что позволяет элементу иметь средний вывод ближе к 0. Как обсуждалось ранее, это помогает смягчить проблему исчезновения градиентов. Гиперпараметр α определяет значение, к которому функция ELU приближается, когда z представляет собой большое отрицательное число. Обычно он устанавливается в 1, но при желании вы можете подстраивать его подобно любому другому гиперпараметру.
- Во-вторых, она имеет ненулевой градиент для $z < 0$, что позволяет избежать проблемы угасающих элементов.
- В-третьих, функция является повсеместно гладкой, включая точки поблизости $z = 0$, что помогает ускорить градиентный спуск, т.к. он не отскакивает настолько сильно слева и справа $z = 0$.

Главный недостаток функции активации ELU связан с тем, что она вычисляется медленнее, чем функция ReLU и ее разновидности (из-за применения экспоненциальной функции), но во время обучения это компенсируется более высокой скоростью схождения. Однако во время испытаний сеть ELU будет медленнее сети ReLU.



Итак, какую функцию активации вы должны использовать для скрытых слоев своих глубоких нейронных сетей? Хотя ваша ситуация может быть иной, в целом выбор выглядит так: функция ELU \Rightarrow функция ReLU с утечкой (и разновидности) \Rightarrow функция ReLU \Rightarrow функция гиперболического тангенса \Rightarrow логистическая функция. Если вас сильно заботит производительность во время выполнения, тогда вы можете отдать предпочтение функциям ReLU с утечкой перед ELU. Если вы не хотите подстраивать очередной гиперпараметр, то можете просто применять стандартные значения α , предложенные ранее (0.01 для ReLU с утечкой и 1 для ELU). При наличии свободного времени и вычислительной мощности вы можете воспользоваться перекрестной проверкой для оценки других функций активации, в частности RReLU, если сеть переобучается, или PReLU, если имеете дело с гигантским обучающим набором.

Библиотека TensorFlow предоставляет функцию `elu()`, которую вы можете применять для построения своей нейронной сети. При вызове функции `dense()` просто установите аргумент `activation`, как показано ниже:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.elu,
                           name="hidden1")
```

Заранее определенная функция для ReLU с утечкой в TensorFlow отсутствует, но определить ее довольно легко:

```
def leaky_relu(z, name=None):
    return tf.maximum(0.01 * z, z, name=name)

hidden1 = tf.layers.dense(X, n_hidden1, activation=leaky_relu,
                           name="hidden1")
```

Пакетная нормализация

Несмотря на то что использование инициализации Хе вместе с ELU (или любой разновидностью ReLU) может значительно уменьшить проблемы исчезновения/взрывного роста градиентов в начале обучения, оно вовсе не гарантирует, что во время обучения проблемы не возникнут снова.

В своей статье 2015 года⁷ Сергей Иоффе и Кристиан Сегеди предложили прием, называемый *пакетной нормализацией* (*Batch Normalization* — *BN*), который предназначен для решения проблем исчезновения/взрывного роста градиентов и более общей проблемы, заключающейся в том, что распределение входов каждого слоя изменяется во время обучения с изменением параметров предшествующих слоев (они назвали ее проблемой *внутреннего ковариационного сдвига* (*internal covariate shift*)).

Прием предусматривает добавление в модель операции прямо перед функцией активации каждого слоя, простое центрирование относительно нуля и нормализацию входов, а затем масштабирование и сдвиг результата с применением двух новых параметров на слой (один для масштабирования и еще один для сдвига). Другими словами, такая операция позволяет модели узнать оптимальный масштаб и среднюю величину по входам для каждого слоя.

Чтобы центрировать относительно нуля и нормализовать входы, алгоритму необходимо произвести оценку средней величины и стандартного отклонения по входам. Он делает это путем вычисления средней величины и стандартного отклонения входов по текущему мини-пакету (отсюда и название “Пакетная нормализация”). Полная операция представлена в уравнении 11.3.

Уравнение 11.3. Алгоритм пакетной нормализации

$$\begin{aligned}
 1. \quad \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)} \\
 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \mu_B)^2 \\
 3. \quad \hat{\mathbf{x}}^{(i)} &= \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
 4. \quad \mathbf{z}^{(i)} &= \gamma \hat{\mathbf{x}}^{(i)} + \beta
 \end{aligned}$$

⁷ “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift” (“Пакетная нормализация: ускорение обучения глубоких сетей путем сокращения внутреннего ковариационного сдвига”), С. Иоффе и К. Сегеди (2015 год) (<https://goo.gl/gA4GSP>).

- μ_B — эмпирическое среднее, вычисленное по всему мини-пакету B .
- σ_B — эмпирическое стандартное отклонение, также вычисленное по всему мини-пакету.
- m_B — количество образцов в мини-пакете.
- $\hat{\mathbf{x}}^{(i)}$ — центрированный относительно нуля и нормализованный вход.
- γ — параметр масштабирования для слоя.
- β — параметр сдвига (смещение) для слоя.
- ε — крошечное число, чтобы избежать деления на ноль (обычно 10^{-5}). Оно называется *сглаживающим членом* (*smoothing term*).
- $\mathbf{z}^{(i)}$ — выход операции BN: это масштабированная и сдвинутая версия входов.

Во время испытаний отсутствует мини-пакет, по которому можно было бы вычислить среднюю величину и стандартное отклонение, поэтому взамен просто используются средняя величина и стандартное отклонение целого обучающего набора. Обычно они эффективно вычисляются во время обучения с применением скользящего среднего. Таким образом, для каждого слоя, прошедшего пакетную нормализацию, в совокупности обучаются четыре параметра: γ (*масштаб*), β (*смещение*), μ (*средняя величина*) и σ (*стандартное отклонение*).

Авторы продемонстрировали, что такой прием значительно улучшил все глубокие нейронные сети, с которыми они экспериментировали. Проблема исчезновения градиентов настолько уменьшилась, что они смогли использовать насыщаемые функции активации, такие как функция гиперболического тангенса или даже логистическая функция активации. Сети также стали гораздо менее чувствительными к инициализации весов. У них была возможность выбирать намного более высокие скорости обучения, значительно ускоряющие процесс обучения. В частности, авторы отмечают: “Пакетная нормализация, примененная к современной модели классификации изображений, достигает такой же правильности за в 14 раз меньшее количество шагов обучения и значительно превосходит исходную модель. [...] За счет использования ансамбля сетей, прошедших пакетную нормализацию, мы улучшаем наилучший опубликованный результат в классификации ImageNet: достигая 4.9%-ной ошибки проверки top-5 (и 4.8%-ной ошибки испытаний), что превышает правильность, показываемую оценщиками-людьми”. Наконец, подобно подарку, не перестающему радовать, пакетная нормализация также ведет

себя как регуляризатор, снижая потребность в других приемах регуляризации (вроде **отключения** (*dropout*)), которое рассматривается далее в главе).

Тем не менее, пакетная нормализация приносит в модель некоторую сложность (хотя устраняет необходимость в нормализации входных данных, т.к. об этом позаботится первый скрытый слой при условии, что он подвергался пакетной нормализации). Кроме того, возникает проблема во время выполнения: нейронная сеть медленнее вырабатывает прогнозы из-за дополнительных вычислений, требующихся на каждом слое. Таким образом, если вам нужно вырабатывать прогнозы молниеносно, то перед экспериментированием с пакетной нормализацией имеет смысл проверить, насколько хорошо выполняется обычный ELU с инициализацией Хе.



Вы можете обнаружить, что поначалу обучение проходит довольно медленно, пока градиентный спуск ищет оптимальные масштабы и смещения для каждого слоя, но оно ускоряется после того, как найдены достаточно хорошие значения.

Реализация пакетной нормализации с помощью TensorFlow

Библиотека TensorFlow предоставляет функцию `tf.nn.batch_normalization()`, которая просто центрирует и нормализует входные данные, но вы должны самостоятельно подсчитать среднюю величину и стандартное отклонение (основываясь на данных мини-пакета во время обучения или полном наборе данных во время испытаний, как только что обсуждалось), после чего передать их указанной функции в качестве параметров. Вдобавок вы обязаны обеспечить создание параметров масштаба и смещения (и передать их функции). Задача выполнима, но подход не самый удобный. Взамен вы должны применять функцию `tf.layers.batch_normalization()`, которая все это делает автоматически, как показано в следующем коде:

```
import tensorflow as tf

n_inputs = 28 * 28
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
training = tf.placeholder_with_default(False, shape=(), name='training')
hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
```

```

bn1 = tf.layers.batch_normalization(hidden1, training=training,
                                     momentum=0.9)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = tf.layers.batch_normalization(hidden2, training=training,
                                     momentum=0.9)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = tf.layers.batch_normalization(logits_before_bn,
                                       training=training, momentum=0.9)

```

Давайте разберем приведенный код. Начальные строки понятны без пояснений вплоть до определения заполнителя `training`: во время обучения мы установим его в `True`, а иначе он будет иметь стандартное значение `False`. Заполнитель предназначен для сообщения функции `tf.layers.batch_normalization()` о том, должна ли она использовать среднюю величину и стандартное отклонение по текущему мини-пакету (во время обучения) или по полному обучающему набору (во время испытаний).

Затем мы чередуем полносвязные слои и слои пакетной нормализации: полносвязные слои создаются с помощью функции `tf.layers.dense()`, как делалось в главе 10. Обратите внимание, что для полносвязных слоев мы не указываем функцию активации, поскольку хотим применять ее после каждого слоя пакетной нормализации⁸. Слои пакетной нормализации создаются с использованием функции `tf.layers.batch_normalization()` с установкой ее параметров `training` и `momentum`. Для подсчета скользящих средних алгоритм BN применяет *экспоненциальный спад* (*exponential decay*), для чего и требуется параметр `momentum`: для имеющегося нового значения v скользящее среднее \hat{v} обновляется посредством такого уравнения:

$$\hat{v} \leftarrow \hat{v} \times \text{momentum} + v \times (1 - \text{momentum})$$

Хорошее значение `momentum` обычно близко к 1 — например, 0,9, 0,99 или 0,999 (для более крупных наборов данных и мелких мини-пакетов вы будете указывать больше девяток).

Вы наверняка заметили, что в коде есть немало повторений, когда те же самые параметры пакетной нормализации встречаются снова и снова.

⁸ Многие исследователи утверждают, что ничуть не хуже или даже лучше размещать слои пакетной нормализации после активаций (а не перед ними).

Во избежание таких повторений вы можете использовать функцию `partial()` из модуля `functools` (часть стандартной библиотеки Python). Она создает тонкую оболочку вокруг функции и позволяет определить стандартные значения для ряда параметров. Создание слоев сети в предыдущем коде можно модифицировать следующим образом:

```
from functools import partial

my_batch_norm_layer = partial(tf.layers.batch_normalization,
                              training=training, momentum=0.9)

hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1")
bn1 = my_batch_norm_layer(hidden1)
bn1_act = tf.nn.elu(bn1)
hidden2 = tf.layers.dense(bn1_act, n_hidden2, name="hidden2")
bn2 = my_batch_norm_layer(hidden2)
bn2_act = tf.nn.elu(bn2)
logits_before_bn = tf.layers.dense(bn2_act, n_outputs, name="outputs")
logits = my_batch_norm_layer(logits_before_bn)
```

В этом небольшом примере новый код может выглядеть не намного лучше, чем ранее, но при наличии 10 слоев и желании применять те же самые функцию активации, инициализатор, регуляризатор и т.п. такой трюк сделает ваш код гораздо более читабельным.

Остаток стадии построения такой же, как в главе 10: определение функции издержек, создание оптимизатора, сообщение ему о необходимости сведения к минимуму функции издержек, определение операций оценки, создание инициализатора переменных, создание экземпляра `Saver` и т.д.

Стадия выполнения тоже практически совпадает, но с двумя исключениями. Во-первых, всякий раз, когда во время обучения вы запускаете операцию, которая зависит от слоя `batch_normalization()`, вы обязаны устанавливать заполнитель `training` в `True`.

Во-вторых, функция `batch_normalization()` создает несколько операций, которые должны вычисляться на каждом шаге во время обучения, чтобы обновлять скользящие средние (вспомните, что эти скользящие средние нужны для вычисления средней величины и стандартного отклонения обучающего набора). Такие операции автоматически добавляются в коллекцию `UPDATE_OPS`, а потому остается лишь извлечь операции из указанной коллекции и выполнять их на каждой итерации обучения:

```

extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run([training_op, extra_update_ops],
                    feed_dict={training: True, X: X_batch, y: y_batch})
            accuracy_val = accuracy.eval(feed_dict={X: mnist.test.images,
                                                    y: mnist.test.labels})
            print(epoch, "Правильность испытаний:", accuracy_val)
        save_path = saver.save(sess, "./my_model_final.ckpt")

```

Вот и все! В рассмотренном крошечном примере со всего лишь двумя слоями вряд ли пакетная нормализация окажет значительное положительное воздействие, но в более глубоких сетях ее влияние может быть огромным.

Отсечение градиентов

Популярный прием уменьшения проблемы, связанной с взрывным ростом градиентов, предусматривает просто урезание градиентов во время обратного распространения, чтобы они никогда не превышали определенный порог (главным образом это полезно для рекуррентных нейронных сетей, как будет показано в главе 14). Прием называется *отсечением градиентов* (*Gradient Clipping*)⁹. Теперь люди в основном отдают предпочтение пакетной нормализации, но знать об отсечении градиентов и о том, как его реализовать, по-прежнему полезно.

Функция `minimize()` оптимизатора в TensorFlow позаботится о вычислении градиентов и их применении, поэтому вы должны сначала вызвать метод `compute_gradients()` оптимизатора, затем создать операцию для отсечения градиентов, используя функцию `clip_by_value()`, и последней создать операцию для применения отсеченных градиентов, используя метод `apply_gradients()` оптимизатора:

```

threshold = 1.0
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(loss)
capped_gvs = [(tf.clip_by_value(grad, -threshold, threshold), var)
              for grad, var in grads_and_vars]
training_op = optimizer.apply_gradients(capped_gvs)

```

⁹ "On the difficulty of training recurrent neural networks" ("О трудности обучения рекуррентных нейронных сетей"), Р. Паскану и др. (2013 год) (<http://goo.gl/dRDAaf>).

Далее вы будете запускать созданную операцию `training_op` на каждом шаге обучения, как обычно. Она вычислит градиенты, отсечет между -1.0 и 1.0 и применит их. Здесь `threshold` является гиперпараметром, который можно подстраивать.

Повторное использование заранее обученных слоев

В большинстве случаев обучать очень крупные сети DNN с нуля — не особенно удачная идея: взамен вы всегда должны пытаться отыскать существующую нейронную сеть, которая выполняет задачу, похожую на решаемую вами, и просто повторно задействовать нижние слои такой сети: прием называется *обучением передачей знаний* (*transfer learning*). Он не только значительно ускоряет обучение, но также требует гораздо меньше данных.

Например, пусть у вас есть доступ к сети DNN, обученной классификации изображений для 100 отличающихся категорий, в числе которых животные, растения, транспортные средства и повседневные предметы. Теперь вы хотите обучить сеть DNN классификации специфических видов транспортных средств. Задачи очень похожи, поэтому вы должны попробовать повторно задействовать части первой сети (рис. 11.4).



Если входные изображения в новой задаче имеют не такие же размеры, как в первой задаче, тогда вам понадобится добавить шаг предварительной обработки для приведения их размеров к тем, которые ожидает первая модель. Вообще говоря, обучение передачей знаний будет хорошо работать только при наличии у входов похожих низкоуровневых признаков.

Повторное использование модели TensorFlow

Если первая модель была обучена с применением TensorFlow, то вы можете просто восстановить ее и обучать на ней свою новую задачу. Как обсуждалось в главе 9, можно воспользоваться функцией `import_meta_graph()`, чтобы импортировать операции в стандартный граф. Она возвращает экземпляр `Saver`, который позже может применяться для загрузки состояния модели:

```
saver = tf.train.import_meta_graph("./my_model_final.ckpt.meta")
```

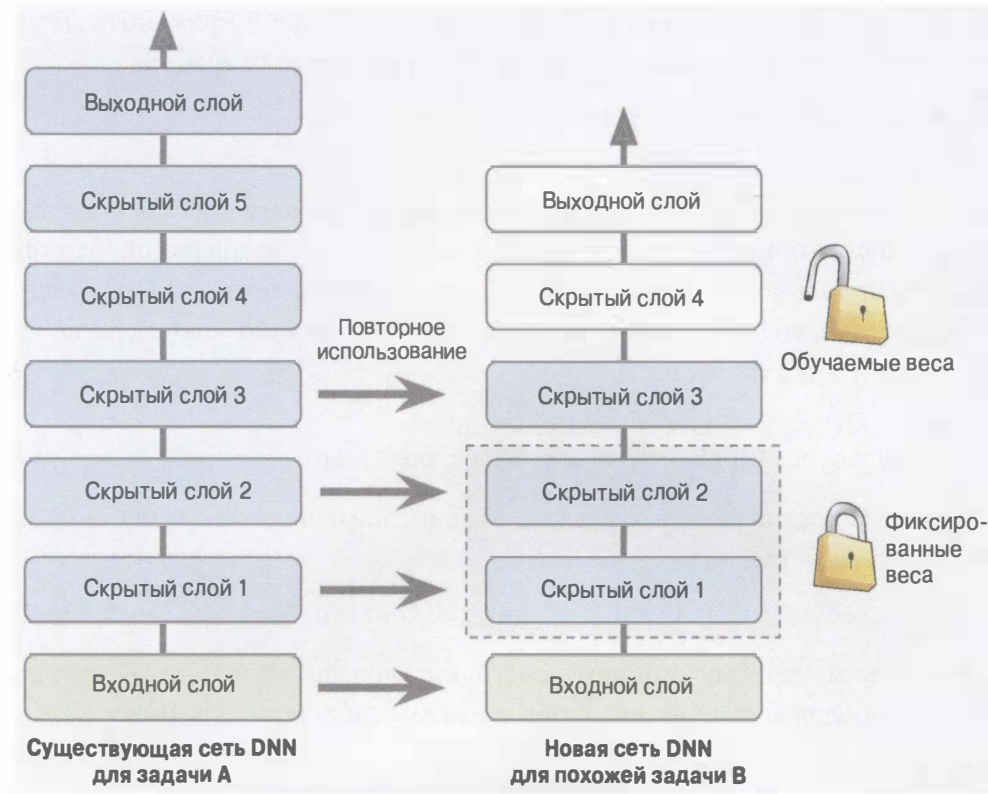


Рис. 11.4. Повторное использование заранее обученных слоев

Далее вы должны получить дескриптор операций и тензоров, который будет необходим при обучении, для чего можете использовать методы `get_operation_by_name()` и `get_tensor_by_name()` графа. Имя тензора представляет собой имя операции, которая выдает его, плюс `:0` (или `:1` для второго выхода, `:2` — для третьего и т.д.):

```
x = tf.get_default_graph().get_tensor_by_name("x:0")
y = tf.get_default_graph().get_tensor_by_name("y:0")
accuracy = tf.get_default_graph().get_tensor_by_name("eval/accuracy:0")
training_op =
    tf.get_default_graph().get_operation_by_name("GradientDescent")
```

Если заранее обученная модель не является хорошо документированной, тогда вам придется исследовать граф с целью поиска имен операций, которые будут нужны. В такой ситуации вы можете либо исследовать граф с применением TensorBoard (сначала необходимо экспортировать граф с исполь-

зованием `FileWriter`, как обсуждалось в главе 9), либо применить метод `get_operations()` графа для извлечения списка всех операций:

```
for op in tf.get_default_graph().get_operations():  
    print(op.name)
```

Если вы автор исходной модели, то могли бы упростить работу людям, которые будут повторно использовать вашу модель, назначая операциям очень ясные имена и документируя их. Другой подход заключается в создании коллекции, содержащей все важные операции, для которых люди пожелают получить дескриптор:

```
for op in (X, y, accuracy, training_op):  
    tf.add_to_collection("my_important_ops", op)
```

Такой подход позволит людям, повторно применяющим вашу модель, просто написать так:

```
X, y, accuracy, training_op = tf.get_collection("my_important_ops")
```

Затем вы можете восстановить состояние модели, используя экземпляр `Saver`, и продолжить обучение с применением собственных данных:

```
with tf.Session() as sess:  
    saver.restore(sess, "./my_model_final.ckpt")  
    [...] # обучение модели на собственных данных
```

В качестве альтернативы, если у вас есть доступ к коду Python, который строил первоначальный граф, тогда вы можете использовать его вместо `import_meta_graph()`.

В общем случае вас будет интересовать применение только части первоначальной модели, обычно низкоуровневых слоев. Если для восстановления графа вы используете функцию `import_meta_graph()`, то она загрузит полный исходный граф, но ничто не может помешать вам просто проигнорировать слои и не думать о них. Например, как показано на рис. 11.4, вы могли бы построить новые слои (скажем, один скрытый слой и один выходной слой) поверх заранее обученного слоя (к примеру, заранее обученного скрытого слоя 3). Вам также потребуется подсчитать потерю для этого нового выхода и создать оптимизатор, сводящий ее к минимуму.

При наличии доступа к коду Python заранее обученного графа вы можете просто повторно задействовать необходимые части и отбросить остальные. Однако в такой ситуации нужен экземпляр `Saver` для восстановления за-

ранее обученной модели (с указанием переменных, которые желательно восстановить; иначе TensorFlow сообщит о несовпадении графов), и еще один экземпляр `Saver` для сохранения новой модели. Например, следующий код восстанавливает только скрытые слои 1, 2 и 3:

```
[...] # построение новой модели с теми же скрытыми слоями 1-3,
      # как и прежде

reuse_vars = tf.get_collection(tf.GraphKeys.GLOBAL_VARIABLES,
                              scope="hidden[123]") # регулярное выражение
reuse_vars_dict = dict([(var.op.name, var) for var in reuse_vars])
restore_saver = tf.train.Saver(reuse_vars_dict) # для восстано-
                                              # вления слоев 1-3

init = tf.global_variables_initializer() # для инициализации всех
                                         # переменных, старых и новых
saver = tf.train.Saver() # для сохранения новой модели

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")
    [...] # обучение модели
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

Сначала мы строим модель, обеспечивая копирование скрытых слоев 1–3 исходной модели. Затем с применением регулярного выражения `"hidden[123]"` мы получаем список всех переменных в скрытых слоях 1–3. Далее мы создаем словарь, который отображает имя каждой переменной в исходной модели на ее имя в новой модели (как правило, вы будете стремиться сохранять имена одинаковыми). После этого мы создаем экземпляр `Saver`, который будет восстанавливать только такие выбранные переменные. Мы также создаем операцию для инициализации всех переменных (старых и новых) и второй экземпляр `Saver` для сохранения полной новой модели, а не только слоев 1–3. Мы запускаем сеанс и инициализируем все переменные в модели, после чего восстанавливаем значения переменных из слоев 1–3 исходной модели. Наконец, мы обучаем модель на новой задаче и сохраняем ее.



Чем больше похожи задачи, тем больше слоев вы пожелаете повторно задействовать (начиная с низкоуровневых слоев). Для очень похожих задач вы можете попробовать сохранить все скрытые слои и просто заменить выходной слой.

Повторное использование моделей из других фреймворков

Если модель обучалась с применением другого фреймворка, тогда вам придется загрузить параметры модели вручную (скажем, используя код Theano в случае ее обучения с помощью Theano) и присвоить их соответствующим переменным. Такие действия могут оказаться довольно утомительными. Например, в следующем коде показано, как можно было бы скопировать веса и смещения из первого скрытого слоя модели, обученной посредством другого фреймворка:

```
original_w = [...] # Загрузка весов из другого фреймворка
original_b = [...] # Загрузка смещений из другого фреймворка

X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                           name="hidden1")

[...] # Построение остатка модели

# Получение дескриптора узлов присваивания для переменных hidden1
graph = tf.get_default_graph()
assign_kernel = graph.get_operation_by_name("hidden1/kernel/Assign")
assign_bias = graph.get_operation_by_name("hidden1/bias/Assign")
init_kernel = assign_kernel.inputs[1]
init_bias = assign_bias.inputs[1]

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init,
              feed_dict={init_kernel: original_w, init_bias: original_b})
[...] # Обучение модели на новой задаче
```

В приведенной реализации мы загружаем модель, заранее обученную с помощью другого фреймворка (код здесь не показан), и извлекаем из нее параметры модели, подлежащие повторному применению. Далее мы строим нашу модель TensorFlow обычным образом. Затем наступает время для трюка: каждая переменная TensorFlow имеет ассоциированную операцию присваивания, которая используется для ее инициализации. Первым делом мы получаем дескриптор для этих операций присваивания (они имеют то же имя, что и переменная, плюс `"/Assign"`). Мы также получаем дескриптор для второго входа каждой операции присваивания: в случае операции присваивания второй вход соответствует значению, которое будет присваиваться переменной, так что в данном случае это инициализирующее значение переменной. После начала сеанса мы запускаем обычную операцию иници-

ализации, но на этот раз передаем ей желаемые значения, предназначенные для переменных, которые должны применяться повторно. В качестве альтернативы мы могли бы создать новые операции присваивания и заполнители и использовать их для установки значений переменных после инициализации. Но зачем создавать новые узлы в графе, если в нем уже есть все, что нам нужно?

Замораживание низкоуровневых слоев

Скорее всего, низкоуровневые слои первой сети DNN научились выявлять в изображениях низкоуровневые признаки, которые будут полезны в обеих задачах классификации изображений, поэтому вы можете просто повторно применять такие слои в том виде как есть. Обычно имеет смысл “заморозить” их веса при обучении новой сети DNN: если веса низкоуровневых слоев фиксированы, тогда будет легче обучать веса слоев более высоких уровней (поскольку им не придется иметь дело с движущейся целью). Чтобы заморозить низкоуровневые слои во время обучения, оптимизатору можно предоставить список обучаемых переменных, который не включает переменные из низкоуровневых слоев:

```
train_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                              scope="hidden[34]|outputs")
training_op = optimizer.minimize(loss, var_list=train_vars)
```

Первая строка получает список всех обучаемых переменных в скрытых слоях 3 и 4, а также в выходном слое. Она пропускает переменные в скрытых слоях 1 и 2. Далее мы предоставляем такой ограниченный список обучаемых переменных функции `minimize()` оптимизатора. Та-дам! Слои 1 и 2 теперь заморожены: во время обучения они не сдвинутся с места (и потому называются *замороженными слоями*).

Другой вариант предусматривает добавление в граф слоя `stop_gradient()`. Он будет замораживать любой слой, находящийся ниже:

```
with tf.name_scope("dnn") :
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                             name="hidden1") # повторно использованный, замороженный
    hidden2 = tf.layers.dense(hidden1, n_hidden2,
                             activation=tf.nn.relu,
                             name="hidden2") # повторно использованный, замороженный
    hidden2_stop = tf.stop_gradient(hidden2)
```

```
hidden3 = tf.layers.dense(hidden2_stop, n_hidden3,
    activation=tf.nn.relu,
    name="hidden3") # повторно использованный, не замороженный
hidden4 = tf.layers.dense(hidden3, n_hidden4,
    activation=tf.nn.relu, name="hidden4") # НОВЫЙ!
logits = tf.layers.dense(hidden4, n_outputs,
    name="outputs") # НОВЫЙ!
```

Кеширование замороженных слоев

Так как замороженные слои изменяться не будут, выход самого верхнего замороженного слоя можно кешировать для каждого обучающего образца. Учитывая, что процесс обучения проходит через полный набор данных много раз, это даст огромный скачок скорости, поскольку проходить через замороженные слои придется только один раз на обучающий образец (а не раз на эпоху). Например, вы могли бы сначала прогнать полный обучающий набор через низкоуровневые слои (при наличии достаточного объема ОЗУ), а во время обучения вместо построения пакетов обучающих образцов строить пакеты выходов из скрытого слоя 2 и передавать их операции обучения:

```
import numpy as np

n_batches = mnist.train.num_examples // batch_size

with tf.Session() as sess:
    init.run()
    restore_saver.restore(sess, "./my_model_final.ckpt")
    h2_cache = sess.run(hidden2, feed_dict={X: mnist.train.images})
    for epoch in range(n_epochs):
        shuffled_idx = np.random.permutation(mnist.train.num_examples)
        hidden2_batches =
            np.array_split(h2_cache[shuffled_idx], n_batches)
        y_batches =
            np.array_split(mnist.train.labels[shuffled_idx], n_batches)
        for hidden2_batch, y_batch in zip(hidden2_batches, y_batches):
            sess.run(training_op,
                feed_dict={hidden2:hidden2_batch, y:y_batch})
    save_path = saver.save(sess, "./my_new_model_final.ckpt")
```

Последняя строка цикла обучения запускает определенную ранее операцию обучения (которая не затрагивает слои 1 и 2), передавая ей пакет выходов из второго скрытого слоя (а также цели для этого пакета). Так как мы предоставляем библиотеке TensorFlow выход скрытого слоя 2, она не пытается его оценивать (как и любого узла, от которого он зависит).

Подстройка, отбрасывание или замена слоев верхних уровней

Выходной слой исходной модели всегда должен заменяться, поскольку он почти наверняка совершенно бесполезен для новой задачи и может даже не иметь корректного количества выходов для новой задачи.

Аналогично верхние скрытые слои исходной модели с меньшей вероятностью будут столь же полезны, как низкоуровневые слои, потому что высокоуровневые признаки, наиболее полезные для новой задачи, могут значительно отличаться от признаков, которые были таковыми для исходной задачи. Вы хотите найти правильное количество слоев, пригодных для повторного использования.

Попробуйте сначала заморозить все скопированные слои, затем обучите модель и посмотрите, как она себя ведет. Далее разморозьте один или два верхних скрытых слоя, чтобы позволить обратному распространению подстроить их, и выясните, улучшилась ли производительность. Чем больше обучающих данных вы имеете, тем больше слоев вы можете размораживать.

Если вы все еще не смогли добиться хорошей производительности и располагаете небольшим объемом обучающих данных, тогда попробуйте отбросить верхний скрытый слой (или слои) и снова заморозить все оставшиеся скрытые слои. Вы можете повторять такие действия до тех пор, пока не найдете правильное количество слоев для повторного использования. Если обучающих данных много, тогда вместо отбрасывания можете попытаться заменить верхние скрытые слои и даже добавить дополнительные скрытые слои.

Зоопарки моделей

Где можно найти нейронную сеть, обученную для задачи, которая похожа на ту, что вы собираетесь решать? Первым очевидным местом, куда следует заглянуть, будет ваш собственный каталог моделей. Вот почему рекомендуется сохранять все свои модели и организовывать их так, чтобы позже их можно было без труда извлекать. Другой вариант предусматривает поиск в *зоопарке моделей* (*model zoo*). Многие люди обучают модели МО для разнообразных задач и любезно предоставляют заранее обученные модели широкой общественности.

Библиотека TensorFlow имеет собственный зоопарк моделей, доступный по ссылке <https://github.com/tensorflow/models>. В частности, он содержит большинство современных сетей классификации изображений, таких как VGG, Inception и ResNet (за деталями обращайтесь в главу 13,

а также просмотрите каталог [models/slim](#)), включая код, заранее обученные модели и инструменты для загрузки популярных наборов данных с изображениями.

Еще одним популярным зоопарком моделей является Caffe's Model Zoo (<https://goo.gl/XI02X3>). Он также содержит много моделей компьютерного зрения (скажем, LeNet, AlexNet, ZFNet, GoogLeNet, VGGNet, inception), обученных на различных наборах данных (например, ImageNet, Places Database, CIFAR10 и т.д.). Сумитро Дасгупта написал преобразователь, который доступен по ссылке <https://github.com/ethereon/caffe-tensorflow>.

Предварительное обучение без учителя

Предположим, что вы хотите заняться сложной задачей, для которой не располагаете большим объемом помеченных обучающих данных, но, к сожалению, не можете найти модель, обученную на похожей задаче. Не отчаивайтесь! Прежде всего, конечно же, вы должны попытаться собрать больше помеченных обучающих данных, но если это слишком трудно или дорого, тогда вы по-прежнему в состоянии провести *предварительное обучение без учителя* (*unsupervised pretraining*), которое проиллюстрировано на рис. 11.5. Другими словами, при наличии множества непомеченных обучающих данных вы можете попробовать обучить слои один за другим. Вы начинаете с самого нижнего слоя и двигаетесь вверх с применением некоторого алгоритма обнаружения признаков без учителя, такого как *ограниченные машины Больцмана* (*Restricted Boltzmann Machine — RBM*), описанные в приложении Д, или автокодировщики, рассматриваемые в главе 15. Каждый слой обучается на выходе из предшествующих обученных слоев (все слои кроме одного обучаемого замораживаются). После того как все слои были обучены указанным способом, вы можете точно настроить сеть, используя обучение с учителем (т.е. с обратным распространением).

Процесс довольно долгий и утомительный, но часто он работает хорошо; на самом деле это тот самый прием, который Джеффри Хинтон и его команда применяли в 2006 году, что привело к возрождению нейронных сетей и успеху глубокого обучения. До 2010 года нормой для глубоких сетей было предварительное обучение без учителя (как правило, с использованием машин RBM), и только после смягчения проблемы исчезновения градиентов гораздо более распространенным стало обучение сетей DNN исключительно с помощью обратного распространения. Тем не менее, предварительное обу-

чение без учителя (в наши дни обычно с применением автокодировщиков, а не машин RBM) все еще представляет собой хороший вариант в ситуации, когда есть сложная в решении задача, отсутствуют похожие модели, которыми можно было бы воспользоваться, и доступно мало помеченных, но масса непомеченных обучающих данных.

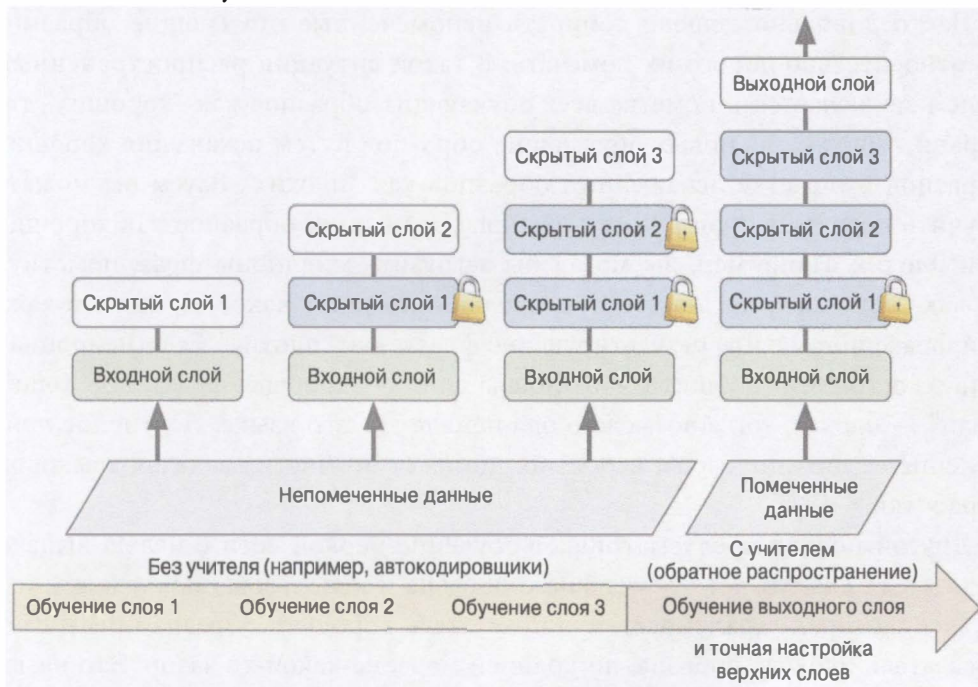


Рис. 11.5. Предварительное обучение без учителя

Предварительное обучение на вспомогательной задаче

Последний вариант предусматривает обучение первой нейронной сети на вспомогательной задаче, для которой можно легко получить или сгенерировать помеченные обучающие данные, и затем повторно применить нижние слои этой сети для фактической задачи. Нижние слои первой нейронной сети научатся обнаруживать признаки и, скорее всего, будут повторно использоваться второй нейронной сетью.

Например, при построении системы для распознавания лиц у вас может быть лишь несколько фотографий каждого индивидуума — явно недостаточно для обучения эффективного классификатора. Накопление сотен фотографий каждой персоны не будет практичным подходом. Однако вы могли бы собрать множество фотографий случайных людей в Интернете и обучить

первую нейронную сеть для обнаружения, представляют ли две разных фотографии одну и ту же персону. Такая сеть стала бы хорошим детектором признаков для лиц, а потому повторное применение ее нижних слоев позволило бы обучить эффективный классификатор лиц с использованием обучающих данных небольшого объема.

Часто довольно дешево собирать непомеченные обучающие образцы, но относительно дорого их пометить. В такой ситуации распространенный прием заключается в пометке всех обучающих образцов как “хороших”, генерации множества новых обучающих образцов путем искажения хороших образцов и пометке искаженных образцов как “плохих”. Затем вы можете обучить первую нейронную сеть для классификации образцов как хороших или плохих. Например, вы могли бы загрузить миллионы фраз, пометить их как “хорошие”, после чего произвольно изменить какое-то слово в каждой фразе и пометить результирующие фразы как “плохие”. Если нейронная сеть в состоянии сообщить, что “Кошка спит” — хорошая фраза, но “Кошка люди” — плохая, тогда возможно она немало знает о языке. Повторное применение ее нижних слоев, вероятно, поможет во многих задачах языковой обработки.

Другой подход предусматривает обучение первой сети с целью выдачи показателя для каждого обучающего образца и использование функции издержек, которая гарантирует, что показатель хорошего образца превышает показатель плохого образца, по крайней мере, на какой-то зазор. Это называется *обучением с максимальным зазором* (*max margin learning*).

Более быстрые оптимизаторы

Обучение очень больших глубоких нейронных сетей может быть мучительно медленным. До сих пор мы видели четыре способа ускорения обучения (и достижения лучшего решения): применение хорошей стратегии инициализации для весов связей, использование хорошей функции активации, применение пакетной нормализации и повторное использование заранее обученной сети. Еще один крупнейший скачок скорости обеспечивает применение более быстрого оптимизатора, чем обыкновенный оптимизатор на основе градиентного спуска. В настоящем разделе мы представим самые популярные из них: *моментная оптимизация* (*momentum optimization*), *ускоренный градиент Нестерова* (*Nesterov accelerated gradient*), AdaGrad, RMSProp и *оптимизация Adam* (*Adam optimization*).

Моментная оптимизация

Вообразите большой мяч, катящийся по гладкой поверхности с пологим уклоном: он начинает движение медленно, но быстро накопит кинетическую энергию, пока в итоге не достигнет конечной скорости (если есть некоторое трение или сопротивление воздуха). Такая совершенно простая идея лежит в основе *моментной оптимизации*, предложенной Борисом Поляком в 1964 году¹⁰. По контрасту обыкновенный градиентный спуск будет делать небольшие постоянные шаги вдоль уклона, поэтому для достижения низа ему потребуется гораздо больше времени.

Вспомните, что градиентный спуск просто обновляет веса θ путем прямого вычитания градиента функции издержек $J(\theta)$ относительно весов ($\nabla_{\theta} J(\theta)$), умноженного на скорость обучения η . Вот уравнение: $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$. Он не заботится о том, какими были начальные градиенты. Если локальный градиент очень маленький, то процесс движется крайне медленно.

Моментная оптимизация принимает во внимание предыдущие градиенты: на каждой итерации локальные градиенты вычитаются из *вектора момента* (*momentum vector*) \mathbf{m} , умноженного на скорость обучения η , а веса обновляются добавлением этого вектора момента (уравнение 11.4). Другими словами, градиенты используются как ускорение, а не как скорость. Для эмуляции механизма трения какого-то вида и предотвращения слишком сильного роста момента алгоритм вводит новый гиперпараметр β , называемый *моментом* (*momentum*), который должен быть установлен между 0 (высокое трение) и 1 (отсутствие трения). Типичное значение момента составляет 0.9.

Уравнение 11.4. Алгоритм моментной оптимизации

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta + \mathbf{m}$

Вы можете легко удостовериться в том, что если градиент остается постоянным, то конечная скорость (т.е. максимальный размер обновлений весов) равна произведению данного градиента и скорости обучения η , умноженной на $\frac{1}{1-\beta}$ (с игнорированием знака). Например, если $\beta = 0.9$, тогда конечная скорость получается умножением 10 на градиент и на скорость обучения,

¹⁰ “Some methods of speeding up the convergence of iteration methods” (“Некоторые методы ускорения сходимости методов итерации”), Б. Поляк (1964 год) (<https://goo.gl/FlSE8c>).

а потому моментная оптимизация заканчивается в 10 раз быстрее градиентного спуска! Это позволяет моментной оптимизации покидать плато намного скорее, чем градиентный спуск. В частности, в главе 4 было показано, что когда входы имеют сильно отличающиеся масштабы, функция издержек будет выглядеть как вытянутая чаша (см. рис. 4.7). Градиентный спуск довольно быстро продвигается вниз по крутому уклону, но затем тратит очень много времени на движение к низу впадины. В противоположность этому моментная оптимизация будет скатываться во впадину все быстрее и быстрее, пока не достигнет низа (оптимума). В глубоких нейронных сетях, которые не применяют пакетную оптимизацию, нижние слои часто будут иметь входы с очень разными масштабами, так что использование моментной оптимизации оказывает существенную помощь. Она также может содействовать в проскакивании мимо локальных оптимумов.



Вследствие момента оптимизатор может слегка промахнуться, возвратиться обратно, снова промахнуться и колебаться подобным образом много раз, прежде чем стабилизироваться в точке минимума. Это одна из причин иметь в системе небольшое трение: оно устраняет такие колебания и тем самым ускоряет сходжение.

Реализовать моментную оптимизацию TensorFlow очень просто: просто поменяйте `GradientDescentOptimizer` на `MomentumOptimizer` и наслаждайтесь полученными выгодами!

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,  
                                         momentum=0.9)
```

Недостаток моментной оптимизации связан с тем, что она добавляет еще один гиперпараметр, подлежащий настройке. Тем не менее, на практике значение момента 0.9 обычно работает хорошо и почти всегда обеспечивает большую скорость, чем градиентный спуск.

Ускоренный градиент Нестерова

Небольшая вариация моментной оптимизации, предложенная Юрием Нестеровым в 1983 году¹¹, почти всегда быстрее обыкновенной моментной

¹¹ “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$ ” (“Метод для решения задачи неограниченной выпуклой минимизации со скоростью схождения $O(1/k^2)$ ”), Юрий Нестеров (1983 год) (<https://goo.gl/V011vD>).

Легко заметить, что обновление моментной оптимизации Нестерова оказывается чуть ближе к оптимуму. Через некоторое время эти мелкие улучшения накапливаются и алгоритм NAG становится значительно быстрее обыкновенной моментной оптимизации. Кроме того, обратите внимание, что когда момент продвигает веса через впадину, ∇_1 продолжает двигаться дальше через впадину, тогда как ∇_2 движется в направлении нижней точки впадины. Это помогает сократить колебания и в результате сделать сходжение более быстрым.

Алгоритм NAG почти всегда будет ускорять обучение в сравнении с обыкновенной моментной оптимизацией. Чтобы применить его, при создании `MomentumOptimizer` просто установите `use_nesterov=True`:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,  
                                         momentum=0.9, use_nesterov=True)
```

AdaGrad

Снова рассмотрим проблему вытянутой чаши: градиентный спуск начинает с быстрого движения вниз по самому крутому уклону и затем медленно перемещается в нижнюю точку впадины. Было бы неплохо, если бы алгоритм мог своевременно выявить это и скорректировать свое направление к точке, более близкой к глобальному оптимуму.

Алгоритм *AdaGrad*¹² достигает такой цели, пропорционально уменьшая вектор-градиент вдоль самых крутых направлений (уравнение 11.6).

Уравнение 11.6. Алгоритм AdaGrad

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

Первый шаг накапливает квадраты градиентов в векторе \mathbf{s} (символ \otimes представляет поэлементное умножение). Такая векторизованная форма эквивалентна вычислению $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$ для каждого элемента s_i вектора \mathbf{s} ; другими словами, каждый s_i накапливает квадраты частной производной функции издержек относительно параметра θ_i . Если функция издержек является крутой вдоль i -того измерения, тогда s_i будет становиться все больше и больше на каждой итерации.

¹² "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization" ("Адаптивные субградиентные методы для динамического обучения и стохастической оптимизации"), Д. Дучи и др. (2011 год) (<http://goo.gl/4Tyd4j>).

Второй шаг почти идентичен градиентному спуску, но с одним большим отличием: вектор-градиент пропорционально уменьшается на коэффициент $\sqrt{s + \varepsilon}$ (символ \oslash представляет поэлементное деление, а ε — сглаживающий член, позволяющий избежать деления на ноль и обычно устанавливаемый в 10^{-10}). Такая векторизованная форма эквивалентна вычислению $\theta_i \leftarrow \theta_i - \eta \partial J(\theta) / \partial \theta_i / \sqrt{s_i + \varepsilon}$ для всех параметров θ_i (одновременно).

Выражаясь кратко, алгоритм ослабляет скорость обучения, но делает это быстрее для крутых измерений, чем для измерений с более пологими уклонами. Получается то, что носит название *адаптивная скорость обучения* (*adaptive learning rate*). Она помогает направлять результирующие обновления в большей степени к глобальному оптимуму (рис. 11.7). Дополнительное преимущество связано с тем, что требуется гораздо меньший объем подстройки гиперпараметра скорости обучения η .

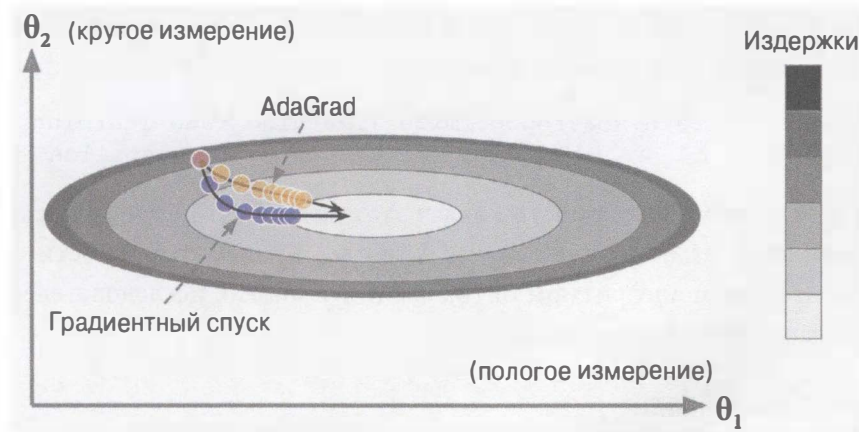


Рис. 11.7. Сравнение AdaGrad и градиентного спуска

Алгоритм AdaGrad часто работает лучше для простых квадратичных задач, но, к сожалению, при обучении нейронных сетей нередко останавливается слишком рано. Скорость обучения сокращается настолько, что алгоритм полностью прекращает работу до достижения глобального оптимума. Таким образом, хотя в TensorFlow есть класс `AdagradOptimizer`, вы не должны его использовать для обучения глубоких нейронных сетей (однако он может оказаться эффективным для более простых задач вроде линейной регрессии).

RMSProp

Несмотря на то что алгоритм AdaGrad замедляется слишком быстро и в итоге никогда не сходится в глобальном оптимуме, алгоритм *RMSProp*¹³ исправляет ситуацию, накапливая только градиенты из самых последних итераций (в противоположность накоплению всех градиентов с начала обучения). Он делает это путем применения экспоненциального ослабления на первом шаге (уравнение 11.7).

Уравнение 11.7. Алгоритм RMSProp

1. $\mathbf{s} \leftarrow \beta \mathbf{s} + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2. $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{\mathbf{s} + \varepsilon}$

Коэффициент ослабления β обычно устанавливается в 0.9. Да, снова имеем новый гиперпараметр, но такое стандартное значение часто хорошо работает, так что потребность в его подстройке может вообще не возникать. Как вы и могли ожидать, в TensorFlow имеется класс `RMSPropOptimizer`:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate,  
                                       momentum=0.9, decay=0.9, epsilon=1e-10)
```

За исключением очень простых задач этот оптимизатор практически всегда выполняется намного лучше, чем AdaGrad. В действительности он был предпочтительным алгоритмом оптимизации у многих исследователей, пока не появилась оптимизация Adam.

Оптимизация Adam

Оптимизация *Adam* (*adaptive moment estimation* — адаптивная оценка момента)¹⁴ объединяет идеи моментной оптимизации и RMSProp: как и моментная оптимизация, она отслеживает экспоненциально ослабляемое среднее арифметическое прошедших градиентов, и подобно RMSProp она отсле-

¹³ Данный алгоритм был создан Тийменом Тилеманом и Джеффри Хинтоном в 2012 году и презентован Джеффри Хинтоном в его курсе Coursera по нейронным сетям (слайды: <http://goo.gl/RsQeis>; видео: <https://goo.gl/XUbIyJ>). Интересно отметить, что поскольку авторы не написали статью, которая описывала бы его, исследователи в своих работах часто ссылаются на “слайд 29 в лекции 6”.

¹⁴ “Adam: A Method for Stochastic Optimization” (“Adam: метод стохастической оптимизации”), Д. Кингма, Д. Ба (2015 год) (<https://goo.gl/Un8Axa>).

живает экспоненциально ослабляемое среднее арифметическое квадратов прошедших градиентов (уравнение 11.8)¹⁵.

Уравнение 11.8. Алгоритм Adam

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\theta} J(\theta)$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
3. $\mathbf{m} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4. $\mathbf{s} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5. $\theta \leftarrow \theta + \eta \mathbf{m} \oslash \sqrt{\mathbf{s} + \varepsilon}$

t представляет номер итерации (начиная с 1).

Взглянув на шаги 1, 2 и 5, вы заметите близкое сходство оптимизации Adam с моментной оптимизацией и RMSProp. Единственное отличие в том, что на шаге 1 вычисляется экспоненциально ослабляемое среднее арифметическое, а не экспоненциально ослабляемая сумма, но фактически они эквивалентны за исключением постоянного множителя (ослабляемое среднее арифметическое — это просто $1 - \beta_1$ раз ослабляемой суммы). Шаги 3 и 4 представляют собой в некотором роде техническую деталь: поскольку \mathbf{m} и \mathbf{s} инициализируются в 0, они будут смещены в сторону 0 в начале обучения, так что указанные два шага помогут поднять \mathbf{m} и \mathbf{s} в начале обучения.

Гиперпараметр ослабления момента β_1 , как правило, инициализируется значением 0.9, а гиперпараметр ослабления масштабирования β_2 часто инициализируется значением 0.999. Как и ранее, сглаживающий член ε обычно инициализируется крошечным числом, скажем, 10^{-8} . Таковы стандартные значения для класса `AdamOptimizer` из TensorFlow, поэтому вы можете поступить следующим образом:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
```

На самом деле, поскольку Adam является алгоритмом с адаптивной скоростью обучения (подобно AdaGrad и RMSProp), он требует меньшей подстройки гиперпараметра скорости обучения η . Часто можно использовать

¹⁵ Они являются оценками средней величины и (неотцентрированной) дисперсии градиентов. Среднюю величину часто именуют *первым моментом*, а дисперсию — *вторым моментом*, отсюда и название алгоритма.

стандартное значение $\eta = 0.001$, делая применение алгоритма Adam даже более легким, чем градиентного спуска.



Первоначально в настоящей книге рекомендовалось применять оптимизацию Adam, потому что обычно она считалась быстрее и лучше других методов. Тем не менее, в статье 2017 года¹⁶ Аши Вилсон и др. показано, что адаптивные методы оптимизации (т.е. AdaGrad, RMSProp и Adam) могут привести к решениям, которые плохо обобщаются на определенные наборы данных. Таким образом, в настоящее время имеет смысл придерживаться моментной оптимизации или ускоренного градиента Нестерова, пока исследователи не обретут лучшее понимание этой проблемы.

Все рассмотренные до сих пор приемы оптимизации полагались только на *частные производные первого порядка (якобианы (jacobian))*. В литературе по оптимизации описаны изумительные алгоритмы, основанные на *частных производных второго порядка (гессианах (hessian))*. К сожалению, эти алгоритмы очень трудно применять к глубоким нейронным сетям, т.к. они предполагают вычисление n^2 гессианов на выход (где n — количество параметров) в противоположность только n якобианов на выход. Из-за того, что сети DNN обычно имеют десятки тысяч параметров, алгоритмы оптимизации второго порядка часто даже не умещаются в память, но если и умещаются, то все равно вычисляют гессианы крайне медленно.

Обучение разреженных моделей

Все представленные алгоритмы оптимизации выпускают плотные модели, т.е. большинство параметров будут ненулевыми. Если вам необходима невероятно быстрая модель во время выполнения или нужно, чтобы модель занимала меньше памяти, тогда вы можете отдать предпочтение разреженной модели.

Очевидный способ достичь цели предусматривает обучение модели обычным образом с последующим избавлением от очень маленьких весов (путем установки их в 0).

¹⁶ “The Marginal Value of Adaptive Gradient Methods in Machine Learning” (“Маргинальное значение адаптивных градиентных методов в машинном обучении”), А. С. Вилсон и др. (2017 год) (<https://goo.gl/NAkWIa>).

Другой вариант предполагает применение во время обучения сильной регуляризации ℓ_1 , потому что она вынуждает оптимизатор обнулять столько весов, сколько возможно (как обсуждалось в главе 4 относительно лассо-регрессии).

Однако в ряде случаев такие приемы могут оставаться неэффективными. Последний вариант заключается в применении *двойного усреднения* (*dual averaging*), часто называемого *следованием за регуляризованным лидером* (*Follow The Regularized Leader* — *FTRL*), которое представляет собой прием, предложенный Юрием Нестеровым¹⁷. Когда этот прием используется с регуляризацией ℓ_1 , он нередко приводит к очень разреженным моделям. Библиотека TensorFlow реализует разновидность FTRL под названием *FTRL-Proximal*¹⁸ в классе `FTRLOptimizer`.

Планирование скорости обучения

Найти хорошую скорость обучения может быть нелегко. Если вы установите ее крайне высокой, тогда обучение фактически может расходиться (как обсуждалось в главе 4). Если вы установите скорость чересчур низкой, то обучение в итоге сойдется в оптимуме, но через очень длительный период времени. Если вы установите ее близкой к слишком высокой, тогда обучение поначалу будет продвигаться очень быстро, но в конце станет совершать прыжки вокруг оптимума, никогда не попадая в него (при условии, что вы не применяете алгоритм оптимизации с адаптивной скоростью обучения, такой как AdaGrad, RMSProp или Adam, но даже тогда ему понадобится время, чтобы добраться до оптимума). В случае ограниченных вычислительных ресурсов может возникнуть необходимость в прерывании обучения до того, как оно сойдется надлежащим образом, с выдачей субоптимального решения (рис. 11.8).

Вы можете найти довольно хорошую скорость обучения, обучая сеть несколько раз на протяжении небольшого числа эпох с использованием различных скоростей обучения и последующим сравнением кривых обучения. Идеальная скорость обучения обеспечит быстрое обучение и схождение в хорошее решение.

¹⁷ “Primal-Dual Subgradient Methods for Convex Problems” (“Прямые-двойные субградиентные методы для выпуклых задач”), Юрий Нестеров (2005 год) (<https://goo.gl/xSQD4C>).

¹⁸ “Ad Click Prediction: a View from the Trenches” (“Прогноз кликов на рекламных объявлениях: взгляд из-за кулис”), Г. Мак-Махан и др. (2013 год) (<https://goo.gl/bxme2B>).

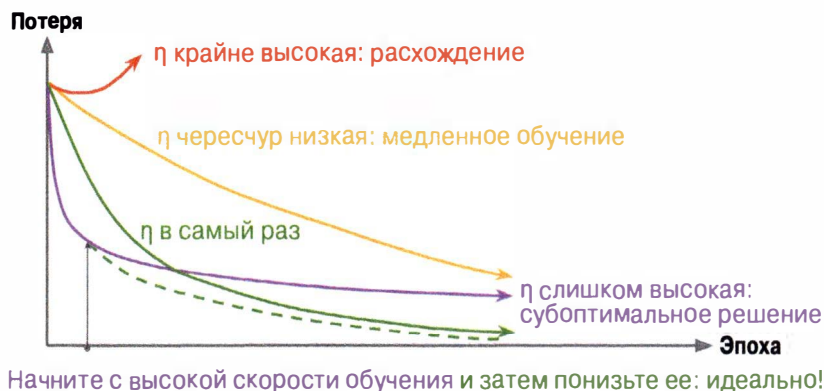


Рис. 11.8. Кривые обучения для различных скоростей обучения η

Тем не менее, вы можете поступить лучше, чем применять постоянную скорость обучения: если начать с высокой скорости обучения и затем понизить ее, как только она перестанет приводить к быстрому продвижению, то достичь хорошего решения удастся быстрее, нежели с помощью оптимальной постоянной скоростью обучения. Существует множество стратегий понижения скорости во время обучения. Такие стратегии называются *графиками обучения* (мы кратко представляли эту концепцию в главе 4); ниже перечислены наиболее распространенные из них.

- **Предопределенная кусочно-линейная постоянная скорость обучения.**
Например, установите сначала скорость обучения в $\eta_0 = 0.1$, а после завершения 50 эпох — в $\eta_1 = 0.001$. Хотя такое решение может работать очень хорошо, часто требуется время на выяснение правильных скоростей обучения и моментов, когда их использовать.
- **График производительности.**
Измеряйте ошибку проверки каждые N шагов (как для раннего прекращения) и понижайте скорость обучения на коэффициент λ , когда ошибка перестает уменьшаться.
- **Экспоненциальный график.**
Установите скорость обучения в функцию номера итерации t : $\eta(t) = \eta_0 10^{-t/r}$. Стратегия работает замечательно, но требует подстройки η_0 и r . Скорость обучения будет падать на показатель 10 каждые r шагов.
- **График мощности.**
Установите скорость обучения в $\eta(t) = \eta_0 (1 + t/r)^{-c}$. Гиперпараметр c обычно устанавливается в 1. Это похоже на экспоненциальный график, но скорость обучения падает гораздо медленнее.

В статье Эндрю Сеньора и др., опубликованной в 2013 году¹⁹, сравнивается производительность ряда самых популярных графиков обучения, когда глубокие нейронные сети обучаются распознаванию речи с применением моментной оптимизации. Авторы пришли к выводу, что в такой среде хорошо работают график производительности и экспоненциальный график, но они отдали предпочтение экспоненциальному графику, поскольку он проще для реализации, легче для подстройки и чуть быстрее сходится в оптимальное решение.

Реализация графика обучения с помощью TensorFlow довольно прямолинейна:

```
initial_learning_rate = 0.1
decay_steps = 10000
decay_rate = 1/10
global_step = tf.Variable(0, trainable=False, name="global_step")
learning_rate = tf.train.exponential_decay(initial_learning_rate,
                                           global_step, decay_steps, decay_rate)
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum=0.9)
training_op = optimizer.minimize(loss, global_step=global_step)
```

После установки значений гиперпараметров мы создаем необучаемую переменную `global_step` (инициализированную 0), предназначенную для хранения номера текущей итерации обучения. Затем мы определяем экспоненциально ослабляемую скорость обучения (с $\eta_0 = 0.1$ и $r = 10\,000$), используя функцию `exponential_decay()` из TensorFlow. Далее мы создаем оптимизатор (в данном примере `MomentumOptimizer`) с применением этой ослабляемой скорости обучения. Наконец, мы создаем операцию обучения, вызывая метод `minimize()` оптимизатора; т.к. мы передаем ему переменную `global_step`, он позаботится о ее инкрементировании. Вот и все!

Поскольку алгоритмы AdaGrad, RMSProp и Adam автоматически понижают скорость обучения в процессе обучения, в добавлении дополнительного графика обучения нет необходимости. Для других алгоритмов оптимизации использование экспоненциального ослабления или графика производительности может значительно ускорить схождение.

¹⁹ “An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition” (“Эмпирическое исследование скоростей обучения в глубоких нейронных сетях для распознавания речи”), Э. Сеньор и др. (2013 год) (<http://goo.gl/Hu6Zyq>).

Избегание переобучения посредством регуляризации

*С помощью четырех параметров я могу описать слона,
а с помощью пяти — заставить его махать хоботом.*

— Перевод высказывания Джона фон Неймана, оригинал которого был процитирован Фрименом Дайсоном в эссе “A meeting with Enrico Fermi” (“Встреча с Энрико Ферми”), журнал *Nature*, том 427 (2004 год), стр. 297

Глубокие нейронные сети обычно имеют десятки тысяч параметров, а иногда параметров миллионы. Благодаря настолько большому количеству параметров сеть обладает невероятной степенью свободы и может подогнаться к широкому многообразию сложных наборов данных. Но такая огромная гибкость означает и предрасположенность к переобучению обучающим набором.

С миллионами параметров вы можете подогнать сеть к целому зоопарку. В настоящем разделе мы представим некоторые из самых популярных приемов регуляризации для нейронных сетей и покажем, как их реализовать с помощью TensorFlow: раннее прекращение, регуляризация ℓ_1 и ℓ_2 , отключение, регуляризация на основе max-нормы (max-norm regularization) и дополнение данных (data augmentation).

Раннее прекращение

Замечательное решение, позволяющее избежать переобучения обучающим набором, заключается в раннем прекращении (введенном в главе 4): просто прервите обучение, когда его производительность на проверочном наборе начинает падать.

Один из способов реализации раннего прекращения посредством TensorFlow предусматривает оценку модели на проверочном наборе через регулярные интервалы (скажем, каждые 50 шагов) и сохранение “выигравшего” снимка, если он превосходит предшествующие “выигравшие” снимки. Подсчитайте количество шагов с момента сохранения последнего “выигравшего” снимка и прервите обучение, когда это число достигло некоторого предела (например, 2 000 шагов). Затем восстановите последний “выигравший” снимок.

Хотя раннее прекращение на практике работает очень хорошо, вы обычно можете добиться гораздо большей производительности от сети, сочетая его с другими приемами регуляризации.

Регуляризация ℓ_1 и ℓ_2

Подобно тому, как мы поступали в отношении простых линейных моделей в главе 4, вы можете применять регуляризацию ℓ_1 и ℓ_2 для ограничения весов связей нейронной сети (но обычно не ее смещений).

Реализовать это с использованием TensorFlow можно путем добавления подходящих членов регуляризации в функцию издержек. Например, предполагая наличие одного скрытого слоя с весами W_1 и одного выходного слоя с весами W_2 , вот как можно применить регуляризацию ℓ_1 :

```
[...] # построение нейронной сети
W1 = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
W2 = tf.get_default_graph().get_tensor_by_name("outputs/kernel:0")
scale = 0.001 # гиперпараметр регуляризации l1
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(
                                                labels=y, logits=logits)
    base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
    reg_losses = tf.reduce_sum(tf.abs(W1)) + tf.reduce_sum(tf.abs(W2))
    loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

Однако если слоев много, тогда такой подход не особенно удобен. К счастью, TensorFlow предлагает лучший вариант. Многие функции, которые создают переменные (такие как `get_variable()` или `tf.layers.dense()`), принимают для каждой создаваемой переменной аргумент `*_regularizer` (скажем, `kernel_regularizer`). Вы можете передавать любую функцию, которая принимает веса в качестве аргументов и возвращает соответствующую потерю регуляризации. Функции `l1_regularizer()`, `l2_regularizer()` и `l1_l2_regularizer()` возвращают такие функции. В следующем коде все сказанное собрано вместе:

```
my_dense_layer = partial(
    tf.layers.dense, activation=tf.nn.relu,
    kernel_regularizer=tf.contrib.layers.l1_regularizer(scale))
with tf.name_scope("dnn"):
    hidden1 = my_dense_layer(X, n_hidden1, name="hidden1")
    hidden2 = my_dense_layer(hidden1, n_hidden2, name="hidden2")
    logits = my_dense_layer(hidden2, n_outputs, activation=None,
                             name="outputs")
```

Код создает нейронную сеть с двумя скрытыми слоями и одним выходным слоем, а также узлы в графе для вычисления потери регуляризации ℓ_1 , соответствующей весам каждого слоя. Библиотека TensorFlow автоматически добавляет эти узлы в специальную коллекцию, которая содержит все потери регуляризации. Вам понадобится лишь добавить такие потери регуляризации к общей потере:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```



Не забывайте о добавлении потерь регуляризации к общей потере, иначе они попросту будут игнорироваться.

Отключение

Вероятно, наиболее популярным приемом регуляризации для глубоких нейронных сетей является *отключение* (*dropout*). Оно было предложено²⁰ Джеффри Хинтоном в 2012 году и дополнительно детализировано в статье²¹ Нитиша Шриваставы и др. Отключение доказало свою высокую успешность: из-за его добавления даже современные нейронные сети получают рост правильности в 1–2%. Рост может не выглядеть большим, но когда модуль уже имеет 95%-ную правильность, то рост правильности в 2% означает уменьшение частоты ошибок почти на 40% (от 5%-ной ошибки до приблизительно 3%-ной).

Алгоритм довольно прост: на каждом шаге обучения каждый нейрон (в том числе входные, но не выходные нейроны) имеет вероятность p быть временно “отключенным”, так что он будет полностью игнорироваться в течение этого шага обучения, но оказаться активным во время следующего шага (рис. 11.9). Гиперпараметр p называется *долей отключения* (*dropout rate*) и обычно устанавливается в 50%. После обучения нейроны больше не отключаются. На этом все (кроме технических деталей, которые мы вскоре обсудим).

²⁰ “Improving neural networks by preventing co-adaptation of feature detectors” (“Улучшение нейронных сетей путем предотвращения совместной адаптации обнаружителей признаков”), Д. Хинтон и др. (2012 год) (<https://goo.gl/PMjVnG>).

²¹ “Dropout: A Simple Way to Prevent Neural Networks from Overfitting” (“Отключение: простой способ предотвращения переобучения нейронных сетей”), Н. Шривастава и др. (2014 год) (<http://goo.gl/DNKZol>).

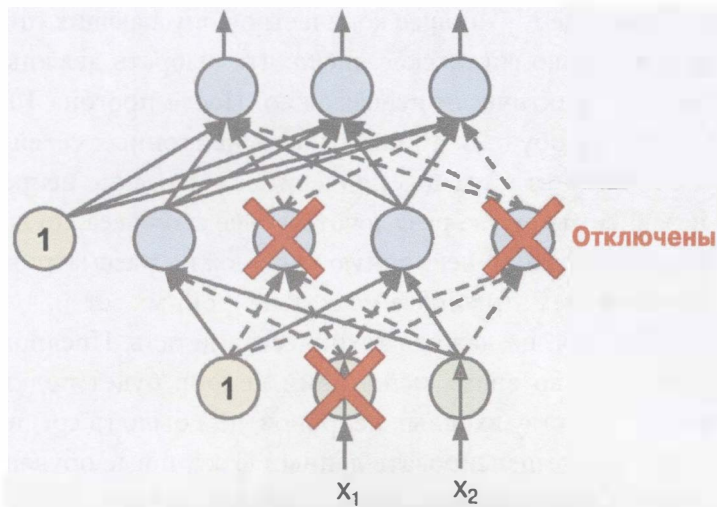


Рис. 11.9. Регуляризация на основе отключения

Поначалу несколько удивляет, что такой весьма жестокий прием вообще работает. Станет ли компания работать лучше, если ее сотрудникам придется по утрам бросать монету, чтобы решать, идти на работу или нет? Как знать; может быть, и станет! Очевидно, компании придется приспособить свою организацию; при заправке кофеварки или выполнении любых других критически важных задач она теперь не может полагаться на какого-то конкретного человека, поэтому такой опыт должен быть распространен на нескольких людей. Служащим придется научиться сотрудничать со многими коллегами, а не только с горсткой из них. Компания станет намного гибче. Если один человек уходит, то это не будет иметь большого значения. Неясно, действительно ли такая идея будет работать для компаний, но она определенно работает для нейронных сетей. Нейроны, обученные с отключением, не могут совместно адаптироваться с соседними нейронами; они должны быть максимально полезными сами по себе. Нейроны, обученные с отключением, также не могут чрезмерно полагаться лишь на несколько входных нейронов; они обязаны обращать внимание на каждый из своих входных нейронов. В итоге они становятся менее чувствительными к незначительным изменениям во входах. В конечном счете, вы получаете более надежную сеть, которая лучше обобщается.

Еще один способ понять мощь отключения — осознать, что на каждом шаге обучения генерируется уникальная нейронная сеть. Поскольку каждый нейрон может либо присутствовать, либо отсутствовать, то существует всего

2^N возможных сетей (где N — общее количество допускающих отбрасывание нейронов). Это настолько гигантское число, что выбрать дважды одну и ту же нейронную сеть практически невозможно. После прогона 10 000 шагов обучения вы, по сути, обучили 10 000 разных нейронных сетей (каждую с только одним обучающим образцом). Очевидно, что такие нейронные сети не являются независимыми, т.к. разделяют многие свои веса, но все-таки они все разные. Результирующую нейронную сеть можно рассматривать как усредненный ансамбль всех этих более мелких нейронных сетей.

Есть одна небольшая, но важная техническая деталь. Предположим, что $p = 50\%$, в случае чего во время испытаний нейрон будет подключаться к вдвое большему количеству входных нейронов, чем было (в среднем) во время обучения. Чтобы скомпенсировать данный факт, после обучения нам необходимо умножить веса входных связей каждого нейрона на 0.5. Если этого не сделать, тогда каждый нейрон получит приблизительно вдвое больший совокупный входной сигнал, чем при обучении сети, и вряд ли будет работать хорошо. В общем случае после обучения мы должны умножить вес каждой входной связи на *вероятность сохранения* ($1 - p$). В качестве альтернативы мы можем разделить выход каждого нейрона на вероятность сохранения во время обучения (такие альтернативы не полностью эквивалентны, но работают одинаково хорошо).

Чтобы реализовать отключение, используя TensorFlow, вы можете просто применить функцию `tf.layers.dropout()` к входному слою и/или к выходу любого желаемого скрытого слоя. Во время обучения указанная функция случайным образом отключает некоторые элементы (устанавливая их в 0) и разделяет оставшиеся элементы на вероятность сохранения. После обучения эта функция вообще ничего не делает. Следующий код применяет регуляризацию на основе отключения к нашей нейронной сети с тремя слоями:

```
[...]
training=tf.placeholder_with_default(False,shape=(),name='training')
dropout_rate = 0.5 # == 1 - keep_prob
X_drop = tf.layers.dropout(X, dropout_rate, training=training)
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X_drop, n_hidden1,
                              activation=tf.nn.relu, name="hidden1")
    hidden1_drop = tf.layers.dropout(hidden1, dropout_rate,
                                     training=training)
```

```
hidden2 = tf.layers.dense(hidden1_drop, n_hidden2,
                           activation=tf.nn.relu, name="hidden2")
hidden2_drop = tf.layers.dropout(hidden2, dropout_rate,
                                  training=training)
logits = tf.layers.dense(hidden2_drop, n_outputs,
                          name="outputs")
```



Вам нужно использовать функцию `tf.layers.dropout()`, а не `tf.nn.dropout()`. Первая функция отключается (ничего не делает), когда не выполняется обучение, что вам и требуется, тогда как вторая — нет.

Конечно, как это делалось при пакетной нормализации, во время обучения `training` необходимо установить в `True`, а во время испытаний оставить стандартное значение `False`.

Если вы обнаруживаете, что модель переобучается, тогда можете увеличить долю отключения. И наоборот, если модель недообучается на обучающем наборе, то долю отключения потребуется уменьшить. Также может помочь увеличение доли отключения для крупных слоев и ее уменьшение для небольших слоев.

Отключение имеет тенденцию значительно замедлять сходжение, но при надлежащей подстройке оно обычно приводит к гораздо лучшей модели. Таким образом, в большинстве ситуаций отключение стоит потраченного времени и усилий.



Развязывание (*dropconnect*) представляет собой разновидность отключения, когда случайным образом отключаются индивидуальные связи, а не целые нейроны. В общем случае отключение работает лучше.

Регуляризация на основе тах-нормы

Еще один прием регуляризации, довольно популярный для нейронных сетей, называется регуляризацией на основе тах-нормы: для каждого нейрона она ограничивает веса \mathbf{w} входящих связей, так что $\|\mathbf{w}\|_2 \leq r$, где r — гиперпараметр тах-нормы, а $\|\cdot\|_2$ — норма ℓ_2 .

Мы обычно реализуем такое ограничение, подсчитывая $\|\mathbf{w}\|_2$ после каждого шага обучения и при необходимости отсекая \mathbf{w} ($\mathbf{w} \leftarrow \mathbf{w} \frac{r}{\|\mathbf{w}\|_2}$).

Сокращение *r* увеличивает величину регуляризации и помогает ослабить переобучение. Регуляризация на основе *max*-нормы также может содействовать в смягчении проблем исчезновения/взрывного роста градиентов (если вы не применяете пакетную нормализацию).

Библиотека TensorFlow не предоставляет готовый регуляризатор на основе *max*-нормы, но реализовать его не слишком трудно. Показанный ниже код получает дескриптор для весов первого скрытого слоя и затем использует функцию `clip_by_norm()`, чтобы создать операцию, которая будет отсекать веса вдоль второй оси, так что каждый вектор-строка имеет *max*-норму 1.0. Последняя строка создает операцию присваивания, которая будет присваивать отсеченные веса переменной `weights`:

```
threshold = 1.0
weights = tf.get_default_graph().get_tensor_by_name("hidden1/kernel:0")
clipped_weights = tf.clip_by_norm(weights, clip_norm=threshold, axes=1)
clip_weights = tf.assign(weights, clipped_weights)
```

Далее созданная операция просто применяется после каждого шага обучения:

```
sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
clip_weights.eval()
```

Обычно это делалось бы для каждого скрытого слоя. Хотя такое решение должно работать хорошо, оно слегка беспорядочно. Более чистое решение предусматривает создание функции `max_norm_regularizer()`, которая используется точно как функция `l1_regularizer()`:

```
def max_norm_regularizer(threshold, axes=1,
                        name="max_norm",
                        collection="max_norm"):

    def max_norm(weights):
        clipped = tf.clip_by_norm(weights, clip_norm=threshold, axes=axes)
        clip_weights = tf.assign(weights, clipped, name=name)
        tf.add_to_collection(collection, clip_weights)
        return None      # член потери регуляризации отсутствует

    return max_norm
```

Функция `max_norm_regularizer()` возвращает параметризованную функцию `max_norm()`, которую можно применять подобно любому другому регуляризатору:

```

max_norm_reg = max_norm_regularizer(threshold=1.0)
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg,
                              name="hidden1")
    hidden2 = tf.layers.dense(hidden1, n_hidden2,
                              activation=tf.nn.relu,
                              kernel_regularizer=max_norm_reg,
                              name="hidden2")
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")

```

Обратите внимание, что регуляризация на основе max-нормы вовсе не требует добавления потери регуляризации к общей функции потерь, из-за чего функция `max_norm()` возвращает `None`. Но вам по-прежнему нужна возможность запускать операции `clip_weights` после каждого шага обучения, а потому вы должны быть в состоянии получать дескриптор для них. Вот почему функция `max_norm()` добавляет операцию `clip_weights` в коллекцию операций отсечения `max_norm`. Вам необходимо извлекать эти операции отсечения и выполнять их после каждого шага обучения:

```

clip_all_weights = tf.get_collection("max_norm")
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            sess.run(clip_all_weights)

```

Намного более чистый код, не так ли?

Дополнение данных

Последний прием регуляризации, дополнение данных, предусматривает генерирование новых обучающих образцов на основе существующих, что искусственно повышает размер обучающего набора. Такое действие будет уменьшать переобучение и это делает его приемом регуляризации. Хитрость в том, чтобы генерировать реалистичные обучающие образцы; в идеале человек не должен быть способен сказать, какие образцы были сгенерированы, а какие нет. Кроме того, простое добавление белого шума не поможет; применяемые вами модификации обязаны быть поддающимися изучению (белый шум не поддается).

Например, если ваша модель предназначена для классификации фотографий грибов, тогда вы можете слегка сдвинуть, повернуть и изменить размер каждой фотографии из обучающего набора на различные величины, после чего добавить результирующие фотографии в обучающий набор (рис. 11.10). Это вынудит модель стать более толерантной к позиции, ориентации и размеру грибов на фотографии. Если вы хотите, чтобы модель была более толерантной к условиям освещения, тогда можете похожим образом сгенерировать множество изображений с разными контрастами. Предполагая, что грибы симметричны, вы также можете переворачивать фотографии по горизонтали. За счет сочетания указанных трансформаций вы в состоянии значительно увеличить свой обучающий набор.

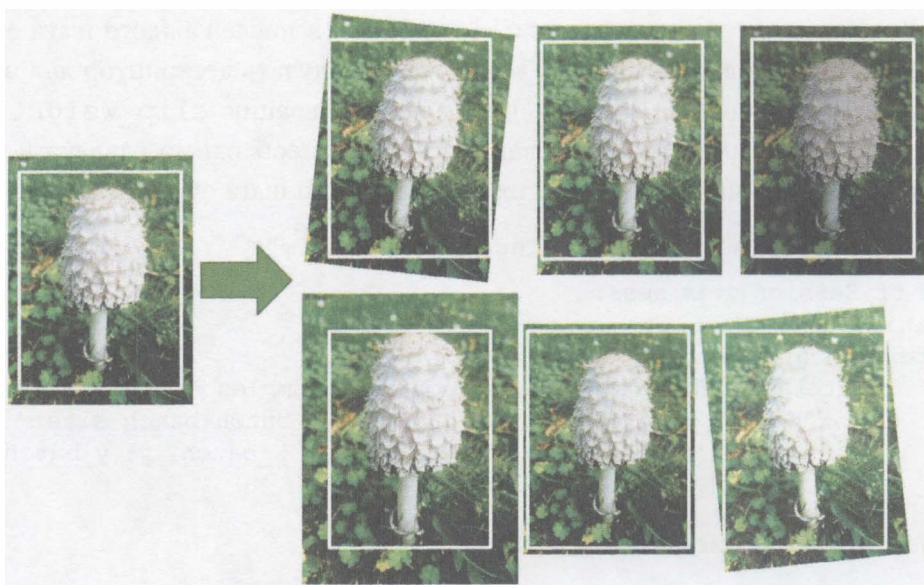


Рис. 11.10. Генерирование новых обучающих образцов на основе существующих

Часто предпочтительнее генерировать обучающие образцы на лету вместо того, чтобы растрачивать пространство памяти и полосу пропускания сети. Библиотека TensorFlow предлагает несколько операций манипулирования изображениями, в том числе перемещение (сдвиг), поворот, изменение размера, переворачивание и обрезка, а также регулирование яркости, контрастности, насыщенности и оттенка (за дополнительными сведениями обращайтесь в документацию по API-интерфейсу). Они облегчают реализацию дополнения данных для наборов данных с изображениями.



Другой мощный прием для обучения очень глубоких нейронных сетей заключается во введении *обходящих связей* (*skip connection*). Обходящая связь относится к добавлению входа слоя к выходу более высокого слоя. Мы исследуем эту идею в главе 13, когда речь пойдет о глубоких остаточных сетях.

Практические рекомендации

В настоящей главе мы раскрыли широкий диапазон приемов, и вас может интересовать, какие из них должны использоваться. В большинстве случаев будет работать конфигурация, приведенная в табл. 11.2.

Таблица 11.2. Стандартная конфигурация сетей DNN

Инициализация	Инициализация Хе
Функция активации	ELU
Нормализация	Пакетная нормализация
Регуляризация	Отключение
Оптимизатор	Ускоренный градиент Нестерова
График обучения	Нет

Разумеется, вы должны пытаться повторно использовать части заранее обученной нейронной сети, если удалось отыскать сеть, которая решает похожую задачу.

Такая стандартная конфигурация может потребовать подстройки.

- Если вы не можете найти хорошую скорость обучения (схождение было очень медленным, поэтому вы увеличили скорость обучения, и теперь схождение стало быстрым, но правильность сети оказалась субоптимальной), тогда попробуйте добавить график обучения, такой как экспоненциальный спад.
- Если ваш обучающий набор несколько мал, то вы можете реализовать дополнение данных.
- Если вам нужна разреженная модель, тогда можете добавить к смеси регуляризацию ℓ_1 (и дополнительно обнулить крошечные веса после обучения). Когда необходима даже более разреженная модель, то вместо оптимизации Adam можете попробовать применить FTRL вместе с регуляризацией ℓ_1 .

- Если вас интересует молниеносно работающая модель во время выполнения, тогда вы можете решить отказаться от пакетной нормализации и возможно заменить функцию активации ELU вариантом ReLU с утечкой. Также поможет наличие разреженной модели.

Располагая такими рекомендациями, вы теперь готовы обучать очень глубокие сети — но только если будете крайне терпеливы! При наличии единственной машины окончания процесса обучения, может быть, придется ждать дни или даже месяцы. В следующей главе мы обсудим, как использовать распределенную библиотеку TensorFlow для обучения и запуска моделей на множестве серверов и графических процессоров.

Упражнения

1. Можно ли инициализировать все веса одним и тем же значением при условии, что оно выбрано случайно с применением инициализации Хе?
2. Можно ли инициализировать члены смещения значением 0?
3. Назовите три преимущества функции активации ELU по сравнению с ReLU.
4. В каких случаях вы бы использовали следующие функции активации: ELU, ReLU с утечкой (и разновидности), ReLU, гиперболического тангенса, логистическую и многопеременную?
5. Что может произойти, если во время применения `MomentumOptimizer` вы установите гиперпараметр `momentum` в значение, слишком близкое к 1 (например, 0.99999)?
6. Назовите три способа получения разреженной модели.
7. Замедляет ли отключение процесс обучения? Замедляет ли оно выведение (т.е. выработку прогнозов на новых образцах)?
8. Глубокое обучение.
 - а) Постройте сеть DNN с пятью скрытыми слоями по 100 нейронов в каждом, инициализацией Хе и функцией активации ELU.
 - б) Используя оптимизацию Adam и раннее прекращение, попробуйте обучить сеть на наборе данных MNIST, но только для цифр 0–4, т.к. в следующем упражнении мы будем применять обучение передачей

знаний для цифр 5–9. Вам понадобится многопеременный выходной слой с пятью нейронами; вдобавок, как всегда, обеспечьте сохранение контрольных точек через регулярные интервалы и сохраните финальную модель, чтобы впоследствии ее можно было повторно использовать.

- в) Подстройте гиперпараметры с применением перекрестной проверки и посмотрите, какой точности вы смогли достичь.
- г) Теперь попробуйте добавить пакетную нормализацию и сравните кривые обучения: происходит ли схождение быстрее, чем раньше? Дает ли это лучшую модель?
- д) Переучивается ли модель обучающим набором? Попробуйте добавить к каждому слою отключение и повторите обучение. Помогло ли это?

9. Обучение передач знаний.

- а) Создайте новую сеть DNN, которая повторно использует все заранее обученные скрытые слои предыдущей модели, заморозьте их и замените многопеременный выходной слой новым слоем.
- б) Обучите новую сеть DNN на цифрах 5–9, применяя только 100 изображений на цифру, и зафиксируйте время, сколько это заняло. Несмотря на такое небольшое количество образцов, смогли ли вы добиться высокой точности?
- в) Попробуйте кешировать замороженные слои и обучите модель снова: насколько быстрее это происходит теперь?
- г) Попробуйте повторно использовать только четыре скрытых слоя вместо пяти. Можете ли вы добиться более высокой точности?
- д) Разморозьте верхние два скрытых слоя и продолжите обучение: удалось ли вам заставить модель работать еще лучше?

10. Предварительное обучение на вспомогательной задаче.

- а) В этом упражнении вы построите сеть DNN, которая сравнивает два изображения цифр MNIST и вырабатывает прогноз, представляют ли они одну и ту же цифру. Затем вы будете повторно применять нижние слои полученной сети для обучения классификатора MNIST, используя обучающие данные совсем небольшого объема. Начните с

создания двух сетей DNN (назовите их DNN A и DNN B), которые похожи на построенную ранее сеть, но без выходного слоя: каждая сеть DNN должна содержать пять скрытых слоев по 100 нейронов, применять инициализацию Хе и использовать функцию активации ELU. Далее добавьте еще один скрытый слой с 10 элементами поверх обеих сетей DNN. Для этого вы должны применить функцию `concat()` из TensorFlow с `axis=1`, чтобы объединить выходы обеих сетей DNN для каждого образца. Затем передайте результат скрытому слою. Наконец, добавьте выходной слой с единственным нейроном, использующим логистическую функцию активации.

- б) Разделите обучающий набор MNIST на две части: часть #1 должна содержать 55 000 изображений, а часть #2 — 5 000 изображений. Создайте функцию, которая генерирует обучающий пакет, где каждый образец представляет собой пару изображений MNIST, выбранных из части #1. Одна половина обучающих образцов должна быть парами изображений, которые принадлежат тому же самому классу, а другая половина должна быть изображениями из разных классов. Для каждой пары метка обучения должна быть равна 0, если изображения относятся к одному и тому же классу, или 1, если к разным классам.
- в) Обучите сеть DNN на полученном обучающем наборе. Для каждой пары изображений вы можете передавать первое изображение в сеть DNN A, а второе — в сеть DNN B одновременно. Полная сеть постепенно научится сообщать, принадлежат ли два изображения к тому же самому классу.
- г) Теперь создайте новую сеть DNN за счет повторного использования и замораживания скрытых слоев сети DNN A и добавьте поверх многопеременный выходной слой с 10 нейронами. Обучите эту сеть на части #2 и посмотрите, можете ли вы добиться высокой производительности, невзирая на наличие всего лишь 500 изображений на класс.

Решения приведенных упражнений доступны в приложении А.

Использование TensorFlow для распределения вычислений между устройствами и серверами

В главе 11 обсуждалось несколько приемов, которые могут значительно ускорить обучение: более подходящая инициализация весов, пакетная нормализация, передовые оптимизаторы и т.д. Однако даже со всеми этими приемами обучение крупной нейронной сети на единственной машине с одним центральным процессором может занимать дни, а то и недели.

В настоящей главе мы посмотрим, как использовать библиотеку TensorFlow для распределения вычислений среди множества устройств (центральных процессоров (ЦП) и графических процессоров (ГП)) и их параллельного выполнения (рис. 12.1). Сначала мы распределим вычисления между устройствами только на одной машине, а затем между многими устройствами на множестве машин.

Поддержка распределенных вычислений библиотекой TensorFlow является одним из ее главных достоинств в сравнении с другими фреймворками для работы с нейронными сетями. Она предоставляет полный контроль над тем, как расщеплять (или реплицировать) вычислительный граф среди устройств и серверов, а также позволяет распараллеливать и синхронизировать операции гибкими способами, так что есть возможность выбирать любой подход к распараллеливанию.

Мы взглянем на ряд самых популярных подходов к распараллеливанию выполнения и обучения нейронной сети. Вместо ожидания неделями, пока алгоритм обучения закончит свою работу, вы можете сократить ожидание до всего нескольких часов. Это означает не только экономию огромного количества времени, но также гораздо более простую возможность экспериментирования с разнообразными моделями и частое повторное обучение моделей на свежих данных.

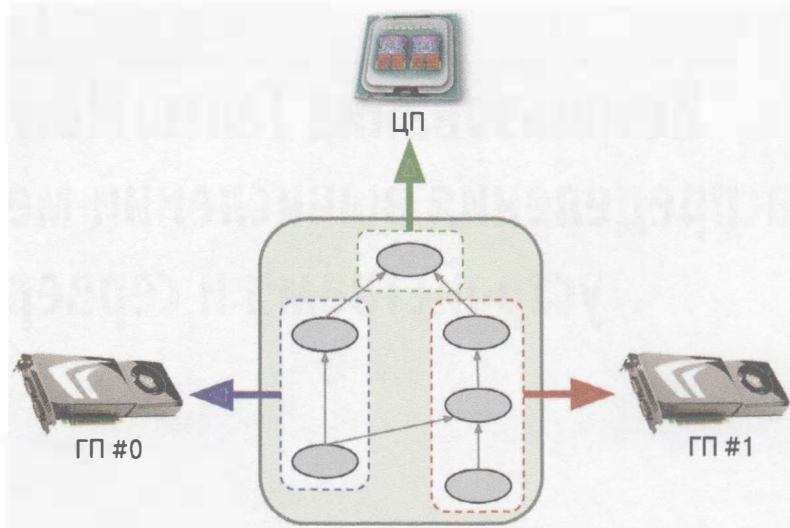


Рис. 12.1. Выполнение графа TensorFlow на множестве устройств параллельно

В число других замечательных сценариев использования распараллеливания входят исследование намного большего пространства гиперпараметров при точной подстройке модели и эффективный прогон крупных ансамблей нейронных сетей.

Но до того как мы сможем бегать, мы должны научиться ходить. Давайте начнем с распараллеливания простых графов между несколькими ГП на одной машине.

Множество устройств на единственной машине

Часто можно получить значительный прирост производительности, просто подключая к машине платы ГП. На самом деле во многих случаях этого окажется достаточно; вам вообще не придется задействовать множество машин. Например, обычно вы сможете обучить нейронную сеть быстрее с применением 8 ГП на единственной машине, чем 16 ГП на нескольких машинах (из-за дополнительной задержки, связанной с взаимодействием через сеть в многомашинной среде).

В этом разделе мы выясним, как настроить среду, чтобы библиотека TensorFlow могла использовать множество плат ГП на одной машине. Затем мы посмотрим, каким образом распределять операции между свободными устройствами и выполнять их в параллельном режиме.

Установка

Чтобы запускать TensorFlow на множестве плат ГП, сначала вы должны удостовериться в том, что имеющиеся платы ГП обладают вычислительной возможностью NVidia (NVidia Compute Capability), большей или равной 3.0. К таким платам относятся Nvidia Titan, Titan X, K20 и K40 (совместимость других плат можно проверить по ссылке <https://developer.nvidia.com/cuda-gpus>).



Если у вас нет плат ГП, тогда можете воспользоваться службой хостинга с возможностью ГП, такой как Amazon AWS. Подробные инструкции по установке TensorFlow 0.9 с Python 3.5 на экземпляре AWS GPU доступны в блоге Žiga Avsec (<http://goo.gl/kbge5b>). Распространить их на последнюю версию TensorFlow должно быть не слишком трудно. Кроме того, компания Google выпустила облачную службу под названием *Cloud Machine Learning* (<https://cloud.google.com/ml>), предназначенную для прогона графов. В мае 2016 года было анонсировано, что платформа Google теперь включает серверы, оснащенные *элементами обработки тензоров* (*Tensor Processing Unit — TPU*), т.е. процессорами, которые специализированы для машинного обучения и работают гораздо быстрее графических процессоров при решении многих задач МО. Разумеется, еще один вариант предполагает приобретение собственной платы ГП. Тим Деттмерс сделал великолепную публикацию в своем блоге (<https://goo.gl/pCtSAn>), которая поможет с выбором, и он достаточно регулярно ее обновляет.

Затем вы должны загрузить и установить подходящую версию библиотек CUDA и cuDNN (CUDA 8.0 и cuDNN v6, если вы применяете двоичную установку TensorFlow 1.3), а также настроить несколько переменных среды, чтобы библиотека TensorFlow знала, где находятся CUDA и cuDNN. Детали установки, вероятно, изменятся довольно скоро, поэтому лучше следовать инструкциям на веб-сайте TensorFlow.

Библиотека Nvidia CUDA (*Compute Unified Device Architecture — архитектура устройств для унифицированных вычислений*) позволяет разработчикам использовать ГП, поддерживающие CUDA, для всех видов вычислений (т.е. не только для ускорения обработки графики). Библиотека Nvidia cuDNN (*CUDA deep neural network — глубокая нейронная сеть CUDA*) представляет собой ускоренную с помощью ГП библиотеку примитивов для

сетей DNN. Она предлагает оптимизированные реализации общих вычислений DNN, таких как слои активации, нормализация, прямые и обратные свертки и организация пула (глава 13). Библиотека входит в состав комплекта Nvidia Deep Learning SDK (имейте в виду, что для ее загрузки потребуется создать учетную запись разработчика Nvidia). Библиотека TensorFlow применяет CUDA и cuDNN для управления платами ГП и ускорения вычислений (рис. 12.2).

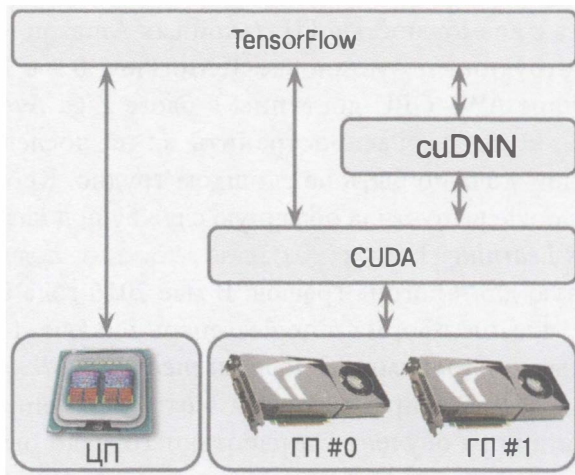


Рис. 12.2. Библиотека TensorFlow использует CUDA и cuDNN для управления платами ГП и ускорения сетей DNN

Для проверки правильности установки CUDA можно применять команду `nvidia-smi`. Она выводит список доступных плат ГП, а также процессов, выполняющихся на каждой плате:

```

$ nvidia-smi
Wed Sep 16 09:50:03 2016
+-----+
| NVIDIA-SMI 352.63      Driver Version: 352.63          |
+-----+-----+
| GPU  Name            Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf    Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|====+=====+====+=====+=====+=====+=====+=====+
|   0   GRID K520           Off    | 0000:00:03.0  Off    |             N/A     |
| N/A   27C    P8     17W / 125W | 11MiB /  4095MiB |         0%      Default |
+-----+-----+-----+-----+-----+-----+-----+

```

Processes:					GPU Memory
GPU	PID	Type	Process name		Usage
No running processes found					
<i>Выполняющиеся процессы не обнаружены</i>					

Наконец, вы должны установить TensorFlow с поддержкой ГП. Если вы создавали изолированную среду с использованием `virtualenv`, тогда активируйте ее:

```
$ cd $ML_PATH # Ваш рабочий каталог МО (например, $HOME/ml)
$ source env/bin/activate
```

Затем установите подходящую версию TensorFlow с поддержкой ГП:

```
$ pip3 install --upgrade tensorflow-gpu
```

Теперь вы можете открыть командную оболочку Python и удостовериться, что TensorFlow обнаруживает и применяет библиотеки CUDA и cuDNN должным образом, для чего импортировать TensorFlow и создать сеанс:

```
>>> import tensorflow as tf
I [...]/dso_loader.cc:108] successfully opened CUDA library
libcublas.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library
libcudnn.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library
libcufft.so locally
I [...]/dso_loader.cc:108] successfully opened CUDA library
libcuda.so.1 locally
I [...]/dso_loader.cc:108] successfully opened CUDA library
libcurand.so locally
>>> sess = tf.Session()
[...]
I [...]/gpu_init.cc:102] Found device 0 with properties:
name: GRID K520
major: 3 minor: 0 memoryClockRate (GHz) 0.797
pciBusID 0000:00:03.0
Total memory: 4.00GiB
Free memory: 3.95GiB
I [...]/gpu_init.cc:126] DMA: 0
I [...]/gpu_init.cc:136] 0: Y
I [...]/gpu_device.cc:839] Creating TensorFlow device
(/gpu:0) -> (device: 0, name: GRID K520, pci bus id: 0000:00:03.0)
```


Выглядит неплохо! Библиотека TensorFlow нашла CUDA и cuDNN и с помощью библиотеки CUDA обнаружила плату ГП (в данном случае Nvidia Grid K520).

Управление оперативной памятью графического процессора

По умолчанию при прогоне графа в первый раз TensorFlow автоматически захватывает всю оперативную память во всех доступных ГП, так что вы не сможете запустить вторую программу TensorFlow, пока выполняется первая программа. Если вы все же попытаетесь, то получите следующее сообщение об ошибке:

```
E [...]/cuda_driver.cc:965] failed to allocate 3.66G (3928915968 bytes)
from device: CUDA_ERROR_OUT_OF_MEMORY
```

Решение может заключаться в запуске каждого процесса на своей плате ГП. Для этого проще всего так установить переменную среды `CUDA_VISIBLE_DEVICES`, чтобы каждый процесс видел только разрешенные платы ГП. Например, вот как вы могли бы запустить две программы:

```
$ CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# и в другом терминале:
$ CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Программа #1 будет видеть только платы ГП 0 и 1 (пронумерованные соответственно 0 и 1), а программа #2 — только платы ГП 2 и 3 (пронумерованные соответственно 1 и 0). Все будет работать нормально (рис. 12.3).

Другой вариант заключается в том, чтобы сообщить TensorFlow о необходимости захвата только доли памяти.

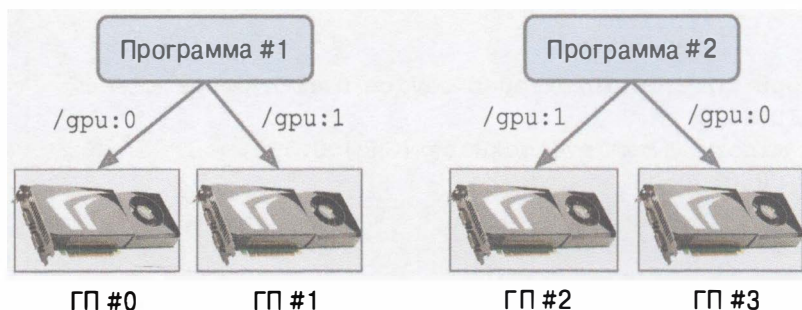


Рис. 12.3. Каждая программа получает в свое распоряжение два ГП

Например, чтобы заставить TensorFlow захватывать только 40% памяти каждого ГП, потребуется создать объект `ConfigProto`, установить его параметр `gpu_options.per_process_gpu_memory_fraction` в `0.4` и создать сеанс, используя такую конфигурацию:

```
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.4
session = tf.Session(config=config)
```

Теперь две программы вроде показанной выше могут выполняться параллельно с применением тех же самых плат ГП (но не три программы, т.к. $3 \times 0.4 > 1$). Взгляните на рис. 12.4.

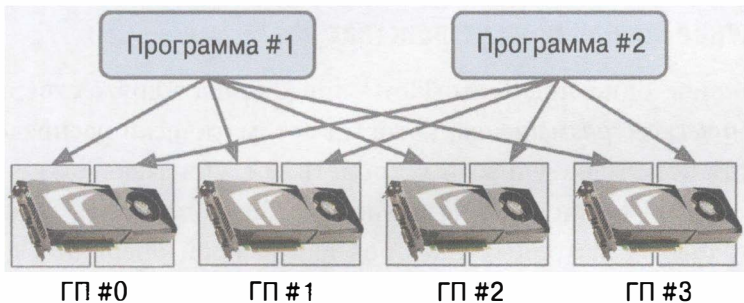


Рис. 12.4. Каждая программа получает в свое распоряжение все четыре ГП, но только 40% оперативной памяти в каждом из них

Если во время выполнения обеих программ вы введете команду `nvidia-smi`, то должны увидеть, что каждый процесс удерживает приблизительно 40% общей оперативной памяти каждой платы ГП:

```
$ nvidia-smi
[...]
```

Processes:					GPU Memory
GPU	PID	Type	Process name	Usage	
0	5231	C	python	1677MiB	
0	5262	C	python	1677MiB	
1	5231	C	python	1677MiB	
1	5262	C	python	1677MiB	

```
[...]
```

Еще один вариант предусматривает сообщение TensorFlow о том, что память нужно захватывать только по мере необходимости. Для этого понадобится установить `config.gpu_options.allow_growth` в `True`. Тем не менее, библиотека TensorFlow никогда не освобождает память после того, как она ее захватила (во избежание фрагментации памяти), а потому через некоторое время по-прежнему может возникнуть нехватка памяти. При таком подходе труднее гарантировать детерминированное поведение, поэтому в целом имеет смысл придерживаться одного из предшествующих вариантов.

Итак, вы располагаете работающей установленной копией TensorFlow с поддержкой графических процессоров. Давайте посмотрим, как пользоваться ею!

Размещение операций на устройствах

Официальное описание TensorFlow¹ представляет дружелюбный алгоритм *динамического размещения*, который автоматически распределяет операции между всеми доступными устройствами, учитывая такие вещи, как измеренное время вычисления в предшествующих прогонах графа, оценки размера входных и выходных тензоров для каждой операции, доступный объем оперативной памяти в каждом устройстве, коммуникационная задержка при передаче данных в устройства и из них, подсказки и ограничения со стороны пользователя и многие другие. К сожалению, этот сложно устроенный алгоритм применяется внутри компании Google; он не был выпущен в версии TensorFlow с открытым кодом. Причина его исключения связана с тем, что на практике небольшой набор правил размещения, указанный пользователем, обеспечивает более эффективный результат, нежели тот, который способно дать динамическое размещение. Однако команда разработчиков TensorFlow трудится над улучшением алгоритма динамического размещения и, может быть, когда-нибудь он станет достаточно хорошим для выпуска.

До тех пор TensorFlow полагается на простое *средство размещения*, которое (как подсказывает его название) является совершенно элементарным.

¹ “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems” (“TensorFlow: крупномасштабное машинное обучение на гетерогенных распределенных системах”), Google Research (2015 год) (<http://goo.gl/vSjA14>).

Простое размещение

Всякий раз, когда вы прогоняете граф, если библиотека TensorFlow нуждается в оценке узла, который пока еще не размещен на устройстве, то она использует простое средство размещения, чтобы разместить его наряду со всеми до сих пор не размещенными узлами. Простое средство размещения подчиняется перечисленным ниже правилам.

- Если узел уже был размещен на устройстве при предыдущем прогоне графа, тогда он остается на этом устройстве.
- Иначе, если пользователь прикрепил узел к устройству (как вскоре будет показано), то средство размещения размещает его на данном устройстве.
- Иначе узел по умолчанию размещается в ГП #0 или в ЦП, если графические процессоры отсутствуют.

Как видите, размещение операций на подходящем устройстве зависит в основном от вас. Если вы ничего не делаете, тогда граф целиком будет помещен в стандартное устройство. Чтобы прикрепить узлы к какому-то устройству, вы обязаны создать блок устройства с применением функции `device()`. Например, следующий код прикрепляет переменную `a` и константу `b` к ЦП, но узел умножения `c` не прикреплен к какому-либо устройству, а потому он будет размещен на стандартном устройстве:

```
with tf.device("/cpu:0") :  
    a = tf.Variable(3.0)  
    b = tf.constant(4.0)  
c = a * b
```



Устройство `"/cpu:0"` агрегирует все ЦП в многопроцессорной системе. В настоящий момент нет способов прикрепления узлов к специфическому ЦП либо использования только подмножества всех ЦП.

Регистрация размещений

Давайте проверим, что простое средство размещения соблюдает все определенные выше ограничения размещения. Для этого вы можете установить параметр `log_device_placement` в `True`, что сообщает средству размещения о необходимости записывать в журнал сообщение всякий раз, когда оно размещает узел.

Вот пример:

```
>>> config = tf.ConfigProto()
>>> config.log_device_placement = True
>>> sess = tf.Session(config=config)
I [...] Creating TensorFlow device (/gpu:0) -> (device: 0,
name: GRID K520, pci bus id: 0000:00:03.0)
[...]
>>> a.initializer.run(session=sess)
I [...] a: /job:localhost/replica:0/task:0/cpu:0
I [...] a/read: /job:localhost/replica:0/task:0/cpu:0
I [...] mul: /job:localhost/replica:0/task:0/gpu:0
I [...] a/Assign: /job:localhost/replica:0/task:0/cpu:0
I [...] b: /job:localhost/replica:0/task:0/cpu:0
I [...] a/initial_value: /job:localhost/replica:0/task:0/cpu:0
>>> sess.run(c)
12
```

Строки, начинающиеся с "I" (от Info — информация), являются журнальными сообщениями. Когда мы создаем сеанс, библиотека TensorFlow записывает в журнал сообщение, указывая на то, что она нашла плату ГП (в данном случае Grid K520). Затем при первом прогоне графа (в этом случае при инициализации переменной `a`) простое средство размещения запускается и размещает каждый узел на устройстве, которое было ему назначено.

Как и ожидалось, журнальные сообщения показывают, что все узлы размещены на устройстве `/cpu:0` кроме узла умножения, который в итоге оказывается на стандартном устройстве `/gpu:0` (пока вы можете благополучно проигнорировать префикс `/job:localhost/replica:0/task:0`; речь о нем пойдет очень скоро).

Обратите внимание, что при прогоне графа во второй раз (с целью вычисления `c`) средство размещения не применяется, поскольку все узлы, необходимые TensorFlow для вычисления `c`, уже размещены.

Функция динамического размещения

При создании блока устройства вместо имени устройства вы можете указывать функцию. Библиотека TensorFlow будет вызывать эту функцию для каждой операции, которую нужно поместить в блок устройства, и функция должна возвращать имя устройства для прикрепления операции. Следующий код прикрепляет все узлы переменных к `/cpu:0` (в данном случае только переменную `a`), а все остальные узлы — к `/gpu:0`:

```
def variables_on_cpu(op):
    if op.type == "Variable":
        return "/cpu:0"
    else:
        return "/gpu:0"

with tf.device(variables_on_cpu):
    a = tf.Variable(3.0)
    b = tf.constant(4.0)
    c = a * b
```

Вы можете без труда реализовать более сложные алгоритмы, такие как прикреплённые переменные к графическим процессорам в циклической манере.

Операции и ядра

Операция TensorFlow, подлежащая выполнению на устройстве, должна иметь реализацию для данного устройства, которая называется *ядром* (*kernel*). Многие операции располагают ядрами для ЦП и ГП, но не все. Например, в TensorFlow не предусмотрено ядро ГП для целочисленных переменных, поэтому приведенный далее код потерпит неудачу, когда TensorFlow попытается разместить переменную *i* на устройстве ГП #0:

```
>>> with tf.device("/gpu:0"):
... i = tf.Variable(3)
[...]
>>> sess.run(i.initializer)
Traceback (most recent call last):
[...]
tensorflow.python.framework.errors.InvalidArgumentError:
Cannot assign a device to node 'Variable': Could not satisfy
explicit device specification

tensorflow.python.framework.errors.InvalidArgumentError:
не удалось назначить устройство узлу 'Variable': не удовлетворяет
явной спецификации устройства
```

Обратите внимание, что TensorFlow делает вывод о том, что переменная должна относиться к типу `int32`, т.к. инициализирующее значение является целочисленным. Если вы измените инициализирующее значение на `3.0` вместо `3` или явно установите `dtype=tf.float32` при создании переменной, тогда код будет работать успешно.

Мягкое размещение

По умолчанию если вы попытаете прикрепить операцию к устройству, для которого у операции отсутствует ядро, то получите показанное ранее исключение, когда TensorFlow попытается разместить операцию на устройстве. Если вы предпочитаете, чтобы библиотека TensorFlow в таком случае размещала операцию на устройстве ЦП, тогда можете установить конфигурационный параметр `allow_soft_placement` в `True`:

```
with tf.device("/gpu:0") :  
    i = tf.Variable(3)  
  
config = tf.ConfigProto()  
config.allow_soft_placement = True  
sess = tf.Session(config=config)  
sess.run(i.initializer) # средство размещения запускается  
                        # и размещает на /cpu:0
```

До сих пор мы обсуждали, каким образом размещать узлы на различных устройствах. Давайте теперь посмотрим, как TensorFlow будет прогонять эти узлы в параллельном режиме.

Параллельное выполнение

Когда библиотека TensorFlow прогоняет граф, она сначала строит список операций, нуждающихся в оценке, и подсчитывает количество зависимостей для каждого узла. Затем TensorFlow добавляет каждую операцию с нулевыми зависимостями (т.е. каждую исходную операцию) в очередь оценки устройства для данной операции (рис. 12.5). После того, как оценка операции произведена, счетчики зависимостей всех операций, которые от нее зависят, декрементируются. Как только счетчик зависимостей какой-то операции достигает нуля, она помещается в очередь оценки своего устройства. После оценки всех необходимых узлов библиотека TensorFlow возвращает их выходы.

Операции в очереди оценки ЦП координируются пулом потоков, который называется *межоперационным пулом потоков* (*inter-op thread pool*). Если ЦП имеет множество ядер, тогда эти операции будут оцениваться параллельно. Некоторые операции располагают многопоточными ядрами ЦП: такие ядра расщепляют свои задачи на множество подопераций. Подоперации помещаются в другую очередь оценки и координируются дополнительным пулом потоков, который называется *внутриоперационным пулом потоков* (*intra-op*

thread pool) и разделяется всеми многопоточными ядрами ЦП. Короче говоря, множество операций и подопераций могут оцениваться параллельно на разных ядрах ЦП.

Для ГП ситуация несколько проще: операции в очереди оценки ГП оцениваются последовательно. Однако многие операции имеют многопоточные ядра ГП, обычно реализованные библиотеками, от которых зависит TensorFlow, например, CUDA и cuDNN. Такие реализации поддерживают собственные пулы потоков и, как правило, эксплуатируют столько потоков ГП, сколько могут (что вероятно является причиной, по которой необходимость в межоперационном пуле потоков для ГП отсутствует, поскольку каждая операция уже использует наибольшее число потоков ГП).

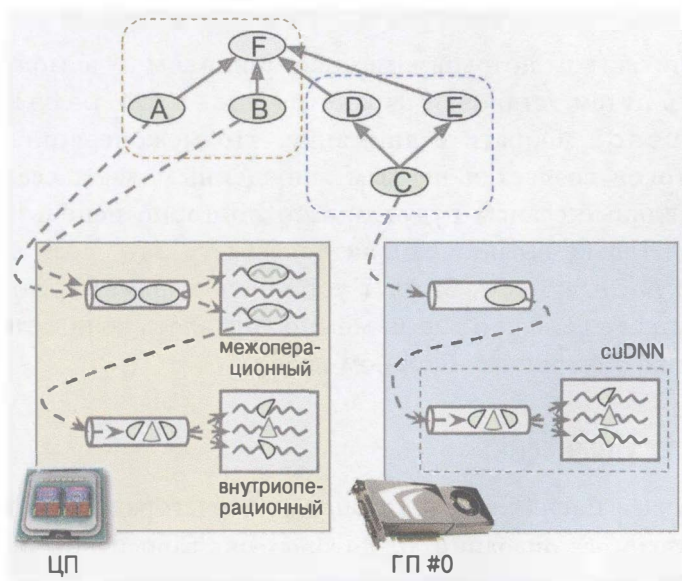


Рис. 12.5. Параллельное выполнение графа TensorFlow

Например, на рис. 12.5 операции A, B и C являются исходными, так что могут быть оценены немедленно. Операции A и B размещены в ЦП, поэтому они отправляются в очередь оценки ЦП, затем координируются межоперационным пулом потоков и незамедлительно оцениваются в параллельном режиме. Так случилось, что операция A имеет многопоточное ядро; ее вычисления расщепляются на три части, которые выполняются параллельно внутриоперационным пулом потоков. Операция C поступает в очередь оценки ГП #0 и в рассматриваемом примере ее ядро ГП задействует биб-

лиотеку cuDNN, которая поддерживает собственный внутриоперационный пул потоков и выполняет операцию во множестве потоков ГП параллельно. Предполагая, что операция С завершилась первой, счетчики зависимостей операций D и E декрементируются и достигают нуля, поэтому обе операции помещаются в очередь оценки ГП #0 и выполняются последовательно. Обратите внимание, что операция С оценивается только раз, хотя от нее зависят две операции, D и E. Предположив, что операция В завершилась следующей, счетчик зависимостей операции F декрементируется с 4 до 3, а поскольку он не достиг нуля, операция F пока не выполняется. После завершения операций A, D и E счетчик зависимостей операции F становится нулевым, так что операция F помещается в очередь оценки ЦП и выполняется. Наконец, библиотека TensorFlow возвращает все запрошенные выходы.



Количеством потоков в межоперационном пуле можно управлять путем установки параметра `inter_op_parallelism_threads`. Обратите внимание, что межоперационный пул потоков создается первым запущенным вами сеансом. Все остальные сеансы будут просто повторно использовать его, если только вы не установите параметр `use_per_session_threads` в `True`. За счет установки параметра `intra_op_parallelism_threads` можно управлять количеством потоков во внутриоперационном пуле.

Зависимости управления

В ряде случаев оценку какой-то операции благоразумно отложить, несмотря на то, что все операции, от которых она зависит, были выполнены. Например, если операция задействует много памяти, но ее значение требуется гораздо позже в графе, тогда будет лучше оценить ее в последний момент, избежав бесполезного расхода оперативной памяти, в которой могут нуждаться другие операции. Еще одним примером является набор операций, которые полагаются на данные, находящиеся вне устройства. Если выполнять их все одновременно, то они могут переполнить коммуникационную полосу пропускания устройства и в итоге ожидать освобождения каналов ввода-вывода. Другие операции, которым необходим обмен данными, также окажутся заблокированными. Более предпочтительно выполнять такие операции с интенсивными коммуникациями последовательно, давая устройству возможность выполнять остальные операции в параллельном режиме.

Чтобы отложить оценку, проще всего добавить *зависимости управления* (*control dependency*).

Например, следующий код сообщает TensorFlow о необходимости оценивать *x* и *y* лишь после того, как оценены *a* и *b*:

```
a = tf.constant(1.0)
b = a + 2.0

with tf.control_dependencies([a, b]):
    x = tf.constant(3.0)
    y = tf.constant(4.0)

z = x + y
```

Очевидно, поскольку *z* зависит от *x* и *y*, оценка *z* также подразумевает ожидание оценки *a* и *b*, хотя *z* явно не находится в блоке `control_dependencies()`. Кроме того, из-за того, что *b* зависит от *a*, мы могли бы упростить предыдущий код, создав зависимость управления на *[b]* вместо *[a, b]*, но в ряде случаев “явно лучше, чем неявно”.

Замечательно! Теперь вы знаете:

- как размещать операции на множестве устройств любым желаемым способом;
- как выполнять эти операции в параллельном режиме;
- как создавать зависимости управления для оптимизации параллельного выполнения.

Наступило время распределить вычисления между множеством серверов!

Множество устройств на множестве серверов

Для прогона графа на множестве серверов первым делом понадобится определить *кластер* (*cluster*). Кластер состоит из одного или большего числа серверов TensorFlow, называемых *задачами* (*task*), которые обычно охватывают несколько машин (рис. 12.6). Каждая задача принадлежит какому-то *заданию* (*job*). Задание представляет собой просто именованную группу задач, как правило, разделяющих общую роль наподобие отслеживания параметров модели (такое задание обычно именуется “ps” от “parameter server” — сервер параметров) или выполнения вычислений (такое задание обычно именуется “worker” — исполнитель).

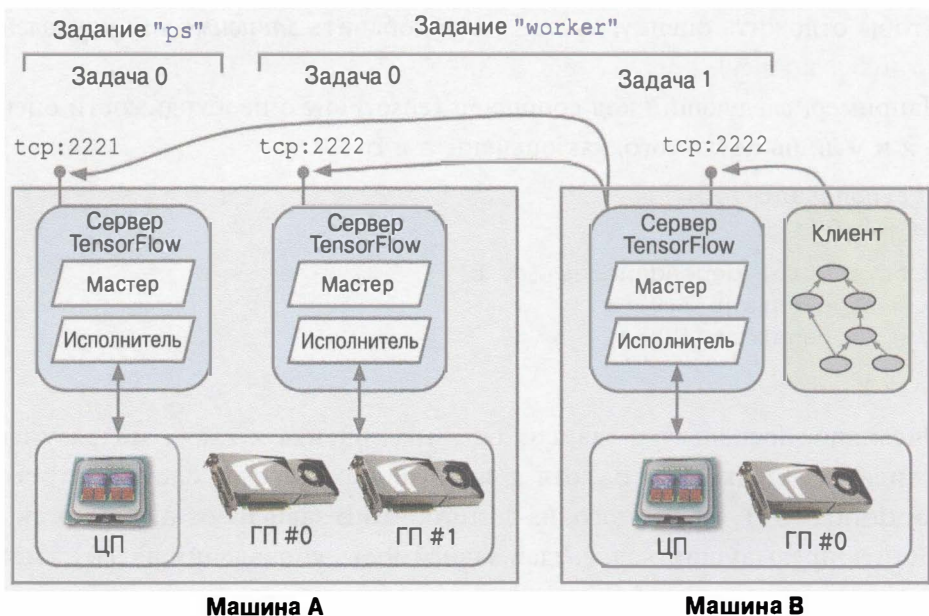


Рис. 12.6. Кластер TensorFlow

Приведенная далее *спецификация кластера* определяет два задания, "ps" и "worker", которые содержат соответственно одну задачу и две задачи. В рассматриваемом примере на машине А размещены два сервера TensorFlow (т.е. задачи), прослушивающие разные порты: первый является частью задания "ps", а второй — частью задания "worker". На машине В размещен только один сервер TensorFlow, представляющий собой часть задания "worker".

```
cluster_spec = tf.train.ClusterSpec({
    "ps": [
        "machine-a.example.com:2221", # /job:ps/task:0
    ],
    "worker": [
        "machine-a.example.com:2222", # /job:worker/task:0
        "machine-b.example.com:2222", # /job:worker/task:1
    ]
})
```

Чтобы запустить сервер TensorFlow, вы должны создать объект `Server`, передав ему спецификацию кластера (так что он сможет взаимодействовать с другими серверами), его собственное имя задания и номер задачи. Например, для запуска первой задачи исполнителя потребуется выполнить следующий код на машине А:


```
server = tf.train.Server(cluster_spec, job_name="worker",
                        task_index=0)
```

Как правило, проще запускать только по одной задаче на машину, но в предыдущем примере было продемонстрировано, что при желании TensorFlow позволяет запускать множество задач на той же самой машине².

При наличии на одной машине нескольких серверов вам придется гарантировать, что они не будут захватывать всю оперативную память каждого ГП, как объяснялось ранее. Скажем, на рис. 12.6 задача "ps" не видит устройства ГП, т.к. ее процесс был запущен с переменной среды `CUDA_VISIBLE_DEVICES=""`. Обратите внимание, что ЦП разделяется всеми задачами, находящимися на той же самой машине.

Если вы хотите, чтобы процесс не делал ничего кроме выполнения сервера TensorFlow, тогда можете заблокировать главный поток, указав ему ожидать завершения работы сервера с применением метода `join()` (иначе сервер будет уничтожен, как только закончится главный поток). Поскольку в настоящее время нет способа остановить сервер, показанный код в действительности будет блокироваться навсегда:

```
server.join()  # блокируется до тех пор, пока сервер
              # не остановится (т.е. навсегда)
```

Открытие сеанса

После того как все задачи подготовлены и работают (пока еще ничего не делаю), вы можете открыть сеанс на любом из серверов из клиента, находящегося в любом процессе на любой машине (даже из процесса, выполняющего одну из задач), и использовать данный сеанс подобно обыкновенному локальному сеансу. Вот пример:

```
a = tf.constant(1.0)
b = a + 2
c = a * 3
with tf.Session("grpc://machine-b.example.com:2222") as sess:
    print(c.eval())  # 9.0
```

² Допускается даже запускать множество задач в том же самом процессе. Такой подход может быть удобен при проведении проверок, но в производственной среде он не рекомендуется.

Клиентский код сначала создает простой граф, затем открывает сеанс на сервере TensorFlow, находящемся на машине В (который мы будем называть *мастером* (*master*)), и инструктирует его оценить *c*. Мастер начинает с размещения операций на подходящих устройствах. Поскольку в этом примере мы не прикрепляем операции к каким-либо устройствам, мастер просто размещает их на собственном стандартном устройстве — в данном случае ГП машины В. Далее согласно инструкции клиента он оценивает *c* и возвращает результат.

Службы мастера и исполнителя

Для взаимодействия с сервером клиент применяет *протокол gRPC* (*Google Remote Procedure Call* — удаленный вызов процедур Google). Он предлагает эффективный фреймворк для вызова удаленных функций и получения их выходных данных среди разнообразных платформ и языков³. Протокол gRPC основан на протоколе HTTP2, который открывает подключение и оставляет его открытым на протяжении всего сеанса, делая возможными двунаправленные коммуникации после того, как подключение установлено. Данные передаются в форме *протокольных буферов* (*protocol buffer*) — еще одной технологии Google с открытым кодом. Это легковесный формат обмена двоичными данными.



Каждый сервер в кластере TensorFlow может взаимодействовать с любыми другими серверами внутри кластера, поэтому удостоверьтесь в том, что открыли соответствующие порты в своем брандмауэре.

Каждый сервер TensorFlow предоставляет две службы: *службу мастера* (*master service*) и *службу исполнителя* (*worker service*). Служба мастера позволяет клиентам открывать сеансы и использовать их для прогона графов. Она координирует вычисления между задачами, полагаясь на службу исполнителя, которая фактически выполняет вычисления в других задачах и получает их результаты.

Такой архитектуре присуща огромная гибкость. Один клиент может подключаться к множеству серверов, открывая многочисленные сеансы в разных

³ В следующей версии внутренней службы Google под названием *Stubby*, которая успешно эксплуатируется более 10 лет. За дополнительными сведениями обращайтесь по адресу <http://grpc.io/>.

потоках. Один сервер способен одновременно обрабатывать множество подключений от одного или большего количества клиентов. Вы можете запускать по одному клиенту на задачу (обычно внутри того же самого процесса) или только один клиент для управления всеми задачами. Доступны все варианты.

Прикрепление операций между задачами

Вы можете применять блоки устройств для прикрепления операций к любому устройству, управляемому любой задачей, за счет указания имени задания, индекса задачи, типа устройства и индекса устройства. Например, приведенный ниже код прикрепляет `a` к ЦП первой задачи в задании `"ps"` (это ЦП на машине A) и `b` — ко второму ГП, управляемому первой задачей задания `"worker"` (это ГП #1 на машине A). Наконец, `c` не прикрепляется к какому-либо устройству, так что мастер размещает `c` на собственном стандартном устройстве (устройство ГП #0 машины B).

```
with tf.device("/job:ps/task:0/cpu:0") :  
    a = tf.constant(1.0)  
  
with tf.device("/job:worker/task:0/gpu:1") :  
    b = a + 2  
  
c = a + b
```

Как и ранее, если вы опустите тип и индекс устройства, то TensorFlow будет использовать стандартное устройство задачи; скажем, прикрепление операции к `"/job:ps/task:0"` разместит ее на стандартном устройстве первой задачи задания `"ps"` (ЦП машины A). Если вы также опустите индекс задачи (к примеру, `"/job:ps"`), тогда TensorFlow применит `"/task:0"`. Если вы опустите имя задания и индекс задачи, то TensorFlow будет использовать задачу мастера сеанса.

Фрагментация переменных среди множества серверов параметров

Как мы вскоре увидим, распространенный подход при обучении нейронной сети в распределенной конфигурации предусматривает хранение параметров модели на наборе серверов параметров (т.е. задач в задании `"ps"`), в то время как остальные задачи фокусируются на вычислениях (т.е. задачи в задании `"worker"`). Для крупных моделей с миллионами параметров эти параметры полезно фрагментировать среди множества серверов параметров, чтобы снизить риск насыщения сетевой платы единственного сервера параметров. Если бы пришлось вручную прикреплять каждую переменную к от-

личающемуся серверу параметров, то такая работа оказалась бы довольно утомительной.

К счастью, TensorFlow предлагает функцию `replica_device_setter()`, которая распределяет переменные между всеми задачами "ps" в циклической манере. Например, следующий код прикрепляет пять переменных к двум серверам параметров:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):  
    v1 = tf.Variable(1.0) # прикрепляется к /job:ps/task:0  
    v2 = tf.Variable(2.0) # прикрепляется к /job:ps/task:1  
    v3 = tf.Variable(3.0) # прикрепляется к /job:ps/task:0  
    v4 = tf.Variable(4.0) # прикрепляется к /job:ps/task:1  
    v5 = tf.Variable(5.0) # прикрепляется к /job:ps/task:0
```

Вместо передачи числа `ps_tasks` можно передать спецификацию кластера `cluster=cluster_spec` и TensorFlow просто подсчитает количество задач в задании "ps".

Если помимо только переменных вы создаете другие операции в блоке, тогда TensorFlow автоматически прикрепит их к устройству `"/job:worker"`, которое будет стандартным для первого устройства, управляемого первой задачей в задании "worker". Вы можете прикрепить их к другому устройству, устанавливая параметр `worker_device`, но более удачный подход заключается в применении встроенных блоков устройств. Внутренний блок устройства может переопределять задание, задачу или устройство, определенные во внешнем блоке.

Вот пример:

```
with tf.device(tf.train.replica_device_setter(ps_tasks=2)):  
    v1 = tf.Variable(1.0) # прикрепляется к /job:ps/task:0  
                        # (+ по умолчанию к /cpu:0)  
    v2 = tf.Variable(2.0) # прикрепляется к /job:ps/task:1  
                        # (+по умолчанию к /cpu:0)  
    v3 = tf.Variable(3.0) # прикрепляется к /job:ps/task:0  
                        # (+по умолчанию к /cpu:0)  
  
    [...]  
    s = v1 + v2          # прикрепляется к /job:worker  
                        # (+по умолчанию к task:0/gpu:0)  
  
    with tf.device("/gpu:1"):  
        p1 = 2 * s       # прикрепляется к /job:worker/gpu:1  
                        # (+по умолчанию к /task:0)  
  
        with tf.device("/task:1"):  
            p2 = 3 * s    # прикрепляется к /job:worker/task:1/gpu:1
```



В примере предполагается, что серверы параметров имеют только ЦП, и обычно это так, поскольку они должны лишь хранить и сообщать параметры, а не выполнять интенсивные вычисления.

Разделение состояния между сеансами с использованием контейнеров ресурсов

Когда вы применяете простой *локальный сеанс* (не распределенный), состояние каждой переменной поддерживается самим сеансом; после завершения сеанса все переменные утрачиваются. Кроме того, два локальных сеанса не способны разделять любое состояние, даже если они оба прогоняют тот же самый граф; каждый сеанс имеет собственные копии всех переменных (как обсуждалось в главе 9). По контрасту при использовании *распределенных сеансов* состояние переменных поддерживается *контейнерами ресурсов* (*resource container*), находящимися в самом кластере, а не сеансами. Таким образом, если вы создаете переменную по имени *x* с применением одного клиентского сеанса, то она автоматически будет доступной любому другому сеансу в том же кластере (даже когда оба сеанса подключены к разным серверам). Взгляните на показанный далее код:

```
# simple_client.py
import tensorflow as tf
import sys

x = tf.Variable(0.0, name="x")
increment_x = tf.assign(x, x + 1)

with tf.Session(sys.argv[1]) as sess:
    if sys.argv[2:] == ["init"]:
        sess.run(x.initializer)
    sess.run(increment_x)
    print(x.eval())
```

Представим, что вы имеете готовый кластер TensorFlow, функционирующий на машинах А и В, порт 2222. С помощью следующей команды вы могли бы запустить клиент, заставить его открыть сеанс с сервером на машине А и сообщить ему о необходимости инициализировать, инкрементировать и вывести значение переменной:

```
$ python3 simple_client.py grpc://machine-a.example.com:2222 init
1.0
```

Если теперь вы запустите клиент посредством приведенной ниже команды, тогда он подключится к серверу на машине В и магически повторно использует ту же самую переменную `x` (на этот раз без запрашивания у сервера инициализации переменной):

```
$ python3 simple_client.py grpc://machine-b.example.com:2222
2.0
```

Такое средство — палка о двух концах: оно великолепно, если нужно разделять переменные между множеством сеансов, но когда необходимо выполнять полностью независимые вычисления в одном кластере, придется проявлять осторожность, чтобы случайно не применить одинаковые имена переменных. Один из способов гарантировать отсутствие конфликтов имен предусматривает помещение всей стадии построения внутрь пространства переменной с уникальным именем для каждого вычисления:

```
with tf.variable_scope("my_problem_1"):
    [...] # Стадия построения задачи 1
```

Более удачный вариант заключается в использовании контейнерного блока (`container`):

```
with tf.container("my_problem_1"):
    [...] # Стадия построения задачи 1
```

В результате будет применяться контейнер, выделенный для задачи #1, вместо стандартного контейнера (с именем в виде пустой строки ""). Преимущество в том, что имена переменных остаются аккуратными и короткими. Еще одно преимущество в том, что именованный контейнер можно легко сбрасывать. Например, следующая команда подключится к серверу на машине А и запросит сброс контейнера по имени `"my_problem_1"`, что освободит все ресурсы, используемые данным контейнером (и также закроет все сеансы, открытые на сервере). Любую переменную, управляемую сброшенным контейнером, потребуются инициализировать перед тем, как ее можно будет применять снова:

```
tf.Session.reset("grpc://machine-a.example.com:2222", ["my_problem_1"])
```

Контейнеры ресурсов облегчают разделение переменных между сеансами гибкими способами.

На рис. 12.7 показаны четыре клиента, выполняющие разные графы в одном кластере, но разделяющие некоторые переменные. Клиенты А и В разделяют ту же самую переменную **x**, поддерживаемую стандартным контейнером, в то время как клиенты С и D разделяют другую переменную по имени **x**, управляемую контейнером "my_problem_1". Обратите внимание, что клиент С использует переменные из обоих контейнеров.



Рис. 12.7. Контейнеры ресурсов

Контейнеры ресурсов также позаботятся о сохранении состояния других операций, поддерживающих состояние, в частности очередей и считывателей. Первыми мы рассмотрим очереди.

Асинхронное взаимодействие с использованием очередей TensorFlow

Очереди предлагают еще один замечательный способ обмена данными между множеством сеансов; например, распространенный сценарий применения предусматривает наличие клиента, который создает граф, загружающий обучающие данные и помещающий их в очередь, и другого клиента, который создает граф, извлекающий данные из очереди и обучающий модель (рис. 12.8). Такой подход значительно ускоряет обучение, потому что операциям обучения не придется ожидать следующего мини-пакета на каждом шаге.

Библиотека TensorFlow предоставляет разнообразные виды очередей. Простейшим видом является очередь *FIFO* (*first-in first-out* — “первым пришел — первым обслужен”).

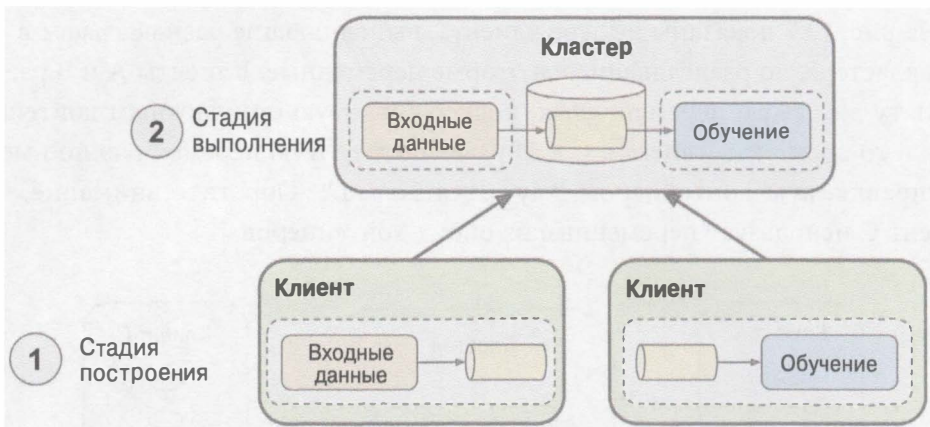


Рис. 12.8. Использование очередей для асинхронной загрузки обучающих данных

Например, приведенный далее код создает очередь FIFO, способную хранить до 10 тензоров, каждый из которых содержит два значения с плавающей точкой:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.float32], shapes=[[2]],
               name="q", shared_name="shared_q")
```



Для разделения переменных между сеансами понадобится лишь указать одно и то же имя и контейнер на обеих сторонах. В случае очередей TensorFlow не применяет атрибут `name`, а взамен использует `shared_name`, поэтому важно указать его (даже если он такой же, как `name`). И, разумеется, нужно применять тот же самый контейнер.

Помещение данных в очередь

Чтобы поместить данные в очередь, вы должны создать операцию `enqueue`. Скажем, следующий код помещает в очередь три обучающих образца:

```
# training_data_loader.py
import tensorflow as tf

q = tf.FIFOQueue(capacity=10, [...], shared_name="shared_q")
training_instance = tf.placeholder(tf.float32, shape=[2])
enqueue = q.enqueue([training_instance])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue, feed_dict={training_instance: [1., 2.]})
    sess.run(enqueue, feed_dict={training_instance: [3., 4.]})
    sess.run(enqueue, feed_dict={training_instance: [5., 6.]})
```

Вместо того чтобы помещать обучающие образцы в очередь по одному, с помощью операции `enqueue_many` вы можете поместить в очередь сразу несколько обучающих образцов:

```
[...]
training_instances = tf.placeholder(tf.float32, shape=(None, 2))
enqueue_many = q.enqueue_many([training_instances])

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    sess.run(enqueue_many,
              feed_dict={training_instances: [[1., 2.], [3., 4.], [5., 6.]]})
```

Оба фрагмента кода помещают в очередь те же самые три тензора.

Извлечение данных из очереди

Для извлечения обучающих образцов из очереди необходимо использовать операцию `dequeue`:

```
# trainer.py
import tensorflow as tf

q = tf.FIFOQueue(capacity=10, [...], shared_name="shared_q")
dequeue = q.dequeue()

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue)) # [1., 2.]
    print(sess.run(dequeue)) # [3., 4.]
    print(sess.run(dequeue)) # [5., 6.]
```

В общем случае вас будет интересовать извлечение за раз целого мини-пакета, а не только одного образца. Для этого придется применять операцию `dequeue_many`, указывая ей размер мини-пакета:

```
[...]
batch_size = 2
dequeue_mini_batch = q.dequeue_many(batch_size)

with tf.Session("grpc://machine-a.example.com:2222") as sess:
    print(sess.run(dequeue_mini_batch)) # [[1., 2.], [4., 5.]]
    print(sess.run(dequeue_mini_batch)) # блокируется в ожидании
                                         # еще одного образца
```

Когда очередь полна, операция помещения в очередь блокируется до тех пор, пока элементы не будут извлечены с помощью соответствующей операции. Аналогично, когда очередь пуста (либо используется `dequeue_many()`, но элементов меньше, чем размер мини-пакета), операция извлечения блоки-

руется до тех пор, пока в очередь не будет помещено достаточное количество элементов.

Очереди кортежей

Каждый элемент в очереди может быть не просто одиночным тензором, но кортежем тензоров (различных типов и форм). Скажем, следующая очередь хранит пары тензоров, в которых один тензор имеет тип `int32` и форму `()`, а другой — тип `float32` и форму `[3, 2]`:

```
q = tf.FIFOQueue(capacity=10, dtypes=[tf.int32, tf.float32],
                shapes=[[], [3, 2]], name="q", shared_name="shared_q")
```

Операции помещения в очередь должны передаваться пары тензоров (обратите внимание, что каждая пара представляет только один элемент в очереди):

```
a = tf.placeholder(tf.int32, shape=())
b = tf.placeholder(tf.float32, shape=(3, 2))
enqueue = q.enqueue((a, b))

with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={a: 10, b: [[1., 2.], [3., 4.], [5., 6.]]})
    sess.run(enqueue, feed_dict={a: 11, b: [[2., 4.], [6., 8.], [0., 2.]]})
    sess.run(enqueue, feed_dict={a: 12, b: [[3., 6.], [9., 2.], [5., 8.]]})
```

С другой стороны, функция `dequeue()` теперь создает пару операций извлечения из очереди:

```
dequeue_a, dequeue_b = q.dequeue()
```

В общем случае вы должны запускать такие операции вместе:

```
with tf.Session([...]) as sess:
    a_val, b_val = sess.run([dequeue_a, dequeue_b])
    print(a_val) # 10
    print(b_val) # [[1., 2.], [3., 4.], [5., 6.]]
```



Если вы запустите лишь операцию `dequeue_a`, то она извлечет пару и возвратит только первый элемент; второй элемент будет утрачен (если вы запустите лишь `dequeue_b`, тогда потеряется первый элемент).

Функция `dequeue_many()` также возвращает пару операций:

```
batch_size = 2
dequeue_as, dequeue_bs = q.dequeue_many(batch_size)
```

Вы можете применять ее вполне ожидаемым образом:

```
with tf.Session([...]) as sess:
    a, b = sess.run([dequeue_a, dequeue_b])
    print(a) # [10, 11]
    print(b) # [[[1., 2.], [3., 4.], [5., 6.]], [[2., 4.], [6., 8.], [0., 2.]]]
    a, b = sess.run([dequeue_a, dequeue_b]) # блокируется в ожидании
                                             # еще одной пары
```

Заккрытие очереди

Очередь можно закрыть, чтобы сообщить другим сеансам о том, что данные больше не будут помещаться в очередь:

```
close_q = q.close()
with tf.Session([...]) as sess:
    [...]
    sess.run(close_q)
```

Последующий запуск операций `enqueue` или `enqueue_many` сгенерирует исключение. По умолчанию любой ожидающий запрос будет обработан, если только вы не вызовете `q.close(cancel_pending_enqueues=True)`.

Последующие операции `dequeue` или `dequeue_many` будут успешно выполняться при условии наличия элементов в очереди, но когда элементов окажется недостаточно, операции потерпят неудачу. Если вы используете операцию `dequeue_many`, а в очереди остается меньше образцов, чем размер мини-пакета, тогда они будут утрачены. Вы можете взамен отдать предпочтение операции `dequeue_up_to`; она ведет себя в точности как `dequeue_many`, но когда очередь закрыта и в ней остается меньше чем `batch_size` образцов, `dequeue_up_to` просто возвратит их.

RandomShuffleQueue

Библиотека TensorFlow также поддерживает еще два типа очередей, в том числе очередь `RandomShuffleQueue`, которую можно применять подобно `FIFOQueue` за исключением того, что элементы извлекаются в случайном порядке. Она может быть удобной для тасования образцов на каждой эпохе во время обучения. Для начала создадим очередь:

```
q = tf.RandomShuffleQueue(capacity=50, min_after_dequeue=10,
                        dtypes=[tf.float32], shapes=[()],
                        name="q", shared_name="shared_q")
```

В `min_after_dequeue` указывается минимальное количество элементов, которые должны оставаться в очереди после операции извлечения. Это гарантирует, что число образцов в очереди будет достаточным для того, чтобы обеспечить нужную степень случайности (после закрытия очереди предел `min_after_dequeue` игнорируется). Теперь предположим, что в созданную очередь помещено 22 элемента (числа с плавающей точкой от 1. до 22.). Вот как их можно было бы извлечь:

```
dequeue = q.dequeue_many(5)
with tf.Session([...]) as sess:
    print(sess.run(dequeue)) # [ 20. 15. 11. 12. 4.]
                                # (осталось 17 элементов)
    print(sess.run(dequeue)) # [ 5. 13. 6. 0. 17.] (осталось 12 элементов)
    print(sess.run(dequeue)) # 12 - 5 < 10: блокируется в ожидании
                                # еще 3 образцов
```

PaddingFifoQueue

Очередь `PaddingFIFOQueue` также может использоваться подобно `FIFOQueue`, исключая то, что она принимает тензоры переменных размеров по любому измерению (но с фиксированным рангом). Когда вы извлекаете их из очереди посредством операции `dequeue_many` или `dequeue_up_to`, каждый тензор дополняется нулями по любому переменному измерению, чтобы сделать его размер таким же, как у самого крупного тензора в минипакете. Например, вы могли бы поместить в очередь двумерные тензоры (матрицы) произвольных размеров:

```
q = tf.PaddingFIFOQueue(capacity=50,
                        dtypes=[tf.float32], shapes=[(None, None)],
                        name="q", shared_name="shared_q")
v = tf.placeholder(tf.float32, shape=(None, None))
enqueue = q.enqueue([v])
with tf.Session([...]) as sess:
    sess.run(enqueue, feed_dict={v: [[1., 2.], [3., 4.], [5., 6.]]}) # 3x2
    sess.run(enqueue, feed_dict={v: [[1.]]})                         # 1x1
    sess.run(enqueue,
              feed_dict={v: [[7., 8., 9., 5.], [6., 7., 8., 9.]]}) # 2x4
```

Если просто извлекать по одному элементу за раз, то будут получаться в точности те же тензоры, которые ранее помещались в очередь. Но в случае извлечения сразу нескольких элементов (с применением `dequeue_many()` или `dequeue_up_to()`) очередь автоматически дополняет тензоры надле-

жащим образом. Скажем, если мы извлечем все три элемента за раз, тогда все тензоры дополнятся нулями, чтобы стать тензорами 3×4 , т.к. максимальный размер для первого измерения составляет 3 (первый элемент), а максимальный размер для второго измерения — 4 (третий элемент):

```
>>> q = [...]
>>> dequeue = q.dequeue_many(3)
>>> with tf.Session([...]) as sess:
...     print(sess.run(dequeue))
...
[[[ 1.  2.  0.  0.]
   [ 3.  4.  0.  0.]
   [ 5.  6.  0.  0.]]
 [[ 1.  0.  0.  0.]
   [ 0.  0.  0.  0.]
   [ 0.  0.  0.  0.]]
 [[ 7.  8.  9.  5.]
   [ 6.  7.  8.  9.]
   [ 0.  0.  0.  0.]]]
```

Очередь этого типа может быть удобна, когда приходится иметь дело с входными данными переменной длины, такими как последовательности слов (глава 14).

Итак, давайте остановимся на секунду: к настоящему времени вы научились распределять вычисления между множеством устройств и серверов, разделять переменные между сеансами и асинхронно взаимодействовать с помощью очередей. Тем не менее, прежде чем приступить к обучению нейронных сетей, необходимо обсудить последнюю тему: как эффективно загружать обучающие данные.

Загрузка данных напрямую из графа

До сих пор мы предполагали, что клиенты загружают обучающие данные и передают их кластеру с использованием заполнителей. Прием прост и неплохо работает в случае несложных конфигураций, но он довольно неэффективен, поскольку перемещает данные несколько раз.

1. Из файловой системы в клиент.
2. Из клиента в задачу мастера.
3. Возможно из задачи мастера в другие задачи, где требуются данные.

Ситуация становится хуже при наличии нескольких клиентов, обучающих различные нейронные сети с применением тех же самых обучающих данных (например, для подстройки гиперпараметров): если все клиенты загружают данные одновременно, тогда может даже произойти насыщение полосы пропускания файлового сервера или сети.

Предварительная загрузка данных в переменную

Для наборов данных, способных уместиться в памяти, лучшим вариантом будет однократная загрузка обучающих данных, присваивание их переменной и использование этой переменной в графе. Прием называется *предварительной загрузкой* обучающего набора. В итоге данные передаются только один раз от клиента в кластер (но по-прежнему могут нуждаться в перемещении между задачами в зависимости от того, каким операциям они необходимы). В следующем коде показано, как загрузить полный обучающий набор в переменную:

```
training_set_init = tf.placeholder(tf.float32, shape=(None, n_features))
training_set = tf.Variable(training_set_init, trainable=False,
                           collections=[], name="training_set")

with tf.Session([...]) as sess:
    data = [...] # загрузка обучающих данных из хранилища
    sess.run(training_set.initializer,
              feed_dict={training_set_init: data})
```

Вы обязаны установить `trainable=False`, чтобы оптимизаторы не пытались подстраивать эту переменную.

Вы также должны установить `collections=[]` для гарантии, что переменная не будет добавлена в коллекцию `GraphKeys.GLOBAL_VARIABLES`, которая применяется при сохранении и восстановлении контрольных точек.



В примере предполагается, что весь обучающий набор (включая метки) состоит только из значений `float32`. В противном случае вам понадобится одна переменная на тип.

Чтение обучающих данных напрямую из графа

Если обучающий набор не уместается в память, тогда хорошим решением будет использование *операций считывателя*, которые представляют собой операции, способные читать данные прямо из файловой системы. В итоге обу-

чающим данным вообще не придется протекать через клиентов. Библиотека TensorFlow предлагает считыватели для различных файловых форматов:

- CSV;
- двоичные записи фиксированной длины;
- собственный формат `TFRecords` библиотеки TensorFlow, основанный на протокольных буферах.

Давайте рассмотрим простой пример чтения из файла CSV (чтение из файлов других форматов подробно описано в документации по API-интерфейсу). Пусть есть файл по имени `my_test.csv`, который содержит обучающие экземпляры, и необходимо создать операции для его чтения. Предположим, что файл имеет представленное ниже содержимое, с двумя признаками типа с плавающей точкой `x1` и `x2` и одной целочисленной целью `target`, представляющей двоичный класс:

```
x1, x2, target
1. , 2. , 0
4. , 5 , 1
7. , , 0
```

Первым делом мы создадим объект `TextLineReader` для чтения файла. Объект `TextLineReader` открывает файл (после того, как ему сообщено, какой файл открывать) и читает его строка за строкой. Подобно переменным и очередям это операция, поддерживающая состояние: она сохраняет состояние между множеством прогонов графа, отслеживая файл, читаемый в текущий момент, и текущую позицию внутри файла.

```
reader = tf.TextLineReader(skip_header_lines=1)
```

Далее мы создаем очередь, из которой считыватель будет извлекать информацию о том, какой файл читать следующим. Мы также создаем операцию помещения и заполнитель для помещения в очередь любого желаемого имени файла, а также операцию закрытия очереди, после того как исчерпаны все файлы, подлежащие чтению:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string],
                             shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
```

Теперь мы готовы создать операцию `read`, которая будет читать по одной записи (т.е. строке) за раз и возвращать пару “ключ-значение”. Ключом является уникальный идентификатор записи (строка, состоящая из имени файла, двоеточия (`:`) и номера строки файла), а значением — строка с содержимым строки файла:

```
key, value = reader.read(filename_queue)
```

У нас есть все необходимое для того, чтобы читать файл строка за строкой! Но сделано пока еще не все — нужно произвести разбор этой строки для получения признаков и цели:

```
x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.], [-1.], [-1]])  
features = tf.stack([x1, x2])
```

Для извлечения значений из текущей строки в коде применяется анализатор CSV из TensorFlow. Когда поля опущены (в данном примере признак `x2` третьего обучающего образца), используются стандартные значения, которые также применяются для определения типа каждого поля (здесь два значения с плавающей точкой и одно целочисленное значение).

Наконец, мы можем поместить обучающий экземпляр и его цель в очередь `RandomShuffleQueue`, которая будет совместно использоваться с обучающим графом (так что он может извлекать из очереди мини-пакеты), а также создать операцию для закрытия очереди, когда помещение в нее образцов завершится:

```
instance_queue = tf.RandomShuffleQueue(  
    capacity=10, min_after_dequeue=2,  
    dtypes=[tf.float32, tf.int32], shapes=[[2], []],  
    name="instance_q", shared_name="shared_instance_q")  
enqueue_instance = instance_queue.enqueue([features, target])  
close_instance_queue = instance_queue.close()
```

Ух! Пришлось проделать немало работы, чтобы всего лишь прочитать файл. Вдобавок мы только создали граф, а его еще необходимо прогнать:

```
with tf.Session([...]) as sess:  
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})  
    sess.run(close_filename_queue)  
    try:  
        while True:  
            sess.run(enqueue_instance)
```

```
except tf.errors.OutOfRangeError as ex:
    pass # в текущем файле не осталось записей,
        # и больше нет файлов для чтения
sess.run(close_instance_queue)
```

Сначала мы открываем сеанс, помещаем в очередь имя файла "my_test.csv" и немедленно закрываем очередь, т.к. не будем помещать в нее другие имена файлов. Затем мы запускаем бесконечный цикл для помещения в очередь образцов друг за другом.

В чтении следующей строки объект `enqueue_instance` полагается на считыватель, так что на каждой итерации читается новая запись, пока не будет достигнут конец файла. В этой точке предпринимается попытка извлечения из очереди имен файлов, чтобы узнать, какой файл должен читаться следующим. Но поскольку упомянутая очередь закрыта, генерируется исключение `OutOfRangeError` (если мы не закроем очередь, произойдет блокировка до тех пор, пока мы не поместим в очередь еще одно имя файла или не закроем ее). В конце мы закрываем очередь образцов, чтобы операции обучения, извлекающие из нее данные, не оказались заблокированными навсегда. На рис. 12.9 подведены итоги того, что было изучено; на нем представлен типовой граф для чтения обучающих образцов из набора файлов CSV.

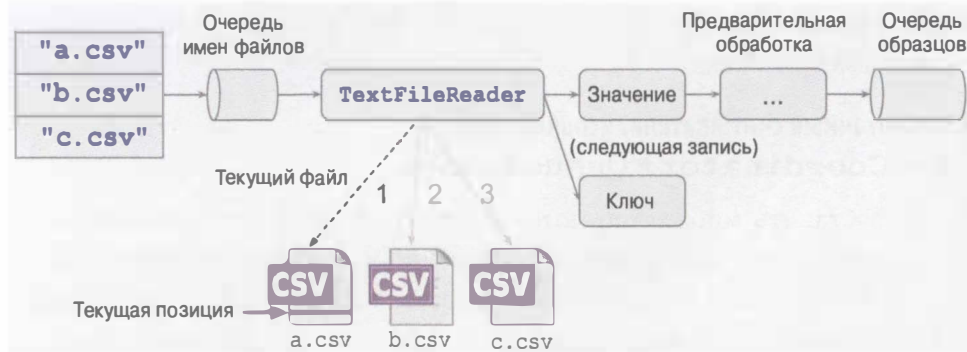


Рис. 12.9. Граф, предназначенный для чтения обучающих образцов из файлов CSV

В обучающем графе понадобится создать разделяемую очередь образцов и просто извлекать из нее мини-пакеты:

```
instance_queue = tf.RandomShuffleQueue(...,
                                     shared_name="shared_instance_q")
mini_batch_instances,
mini_batch_targets = instance_queue.dequeue_up_to(2)
```

```
[...] # использование образцов и целей мини-пакета
      # для построения обучающего графа
training_op = [...]
with tf.Session([...]) as sess:
    try:
        for step in range(max_steps):
            sess.run(training_op)
    except tf.errors.OutOfRangeError as ex:
        pass # обучающих образцов больше нет
```

В рассмотренном примере первый мини-пакет будет содержать первые два образца из файла CSV, а второй мини-пакет — последний образец.



Очереди TensorFlow не особенно хорошо обрабатывают разреженные тензоры, поэтому если ваши обучающие образцы являются разреженными, то вы должны производить разбор записей после очереди образцов.

Такая архитектура будет применять только один поток для чтения записей и их помещения в очередь образцов. С использованием множества считывателей вы можете добиться гораздо более высокой пропускной способности, заставив множество потоков одновременно читать из множества файлов. Давайте посмотрим как.

Многопоточные считыватели, использующие классы `Coordinator` и `QueueRunner`

Чтобы заставить множество потоков читать образцы одновременно, вы могли бы создать потоки Python (с применением модуля `threading`) и управлять ими самостоятельно. Однако TensorFlow предоставляет ряд инструментов, чтобы облегчить работу: классы `Coordinator` и `QueueRunner`.

Единственной целью объекта `Coordinator` является координация остановки множества потоков. Сначала создается объект `Coordinator`:

```
coord = tf.train.Coordinator()
```

Затем ему передаются все потоки, которые нужно остановить сообща, и их главный цикл выглядит примерно так:

```
while not coord.should_stop():
    [...] # делать что-то
```

Любой поток может запросить остановку каждого потока, вызывая метод `request_stop()` объекта `Coordinator`:

```
coord.request_stop()
```

Каждый поток остановится, как только завершит свою текущую итерацию. Вызывая метод `join()` объекта `Coordinator` и передавая ему список потоков, можно организовать ожидание момента, когда все потоки завершат свои текущие итерации:

```
coord.join(list_of_threads)
```

Объект `QueueRunner` запускает множество потоков, каждый из которых многократно выполняет операцию помещения в очередь, как можно быстрее заполняя очередь. После закрытия очереди следующий поток, который попытается поместить в нее элемент, получит исключение `OutOfRangeError`; он перехватит это исключение и с помощью объекта `Coordinator` незамедлительно сообщит остальным потокам о необходимости остановиться. В следующем коде показано, как с использованием объекта `QueueRunner` заставить пять потоков одновременно читать образцы и помещать их в очередь образцов:

```
[...] # та же самая стадия построения, что и ранее
queue_runner =
    tf.train.QueueRunner(instance_queue, [enqueue_instance] * 5)
with tf.Session() as sess:
    sess.run(enqueue_filename, feed_dict={filename: "my_test.csv"})
    sess.run(close_filename_queue)
    coord = tf.train.Coordinator()
    enqueue_threads =
        queue_runner.create_threads(sess, coord=coord, start=True)
```

Первая строка кода создает объект `QueueRunner` и указывает ему о том, что нужно запустить пять потоков, которые все многократно выполняют ту же самую операцию `enqueue_instance`. Затем мы начинаем сеанс и помещаем в очередь имена файлов, подлежащих чтению (здесь только `"my_test.csv"`). Далее мы создаем объект `Coordinator`, который `QueueRunner` будет применять для элегантной остановки, как объяснялось выше. Наконец, мы сообщаем `QueueRunner` о необходимости создания и запуска потоков. Потоки прочитают все обучающие образцы и поместят их в очередь образцов, после чего будут элегантно остановлены.

Решение будет чуть более эффективным, чем ранее, но мы способны сделать его еще лучше. В настоящий момент все потоки выполняют чтение из одного файла. Мы можем взамен организовать одновременное чтение ими из отдельных файлов (при условии, что обучающие данные фрагментируются среди множества файлов CSV), создав несколько считывателей (рис. 12.10).

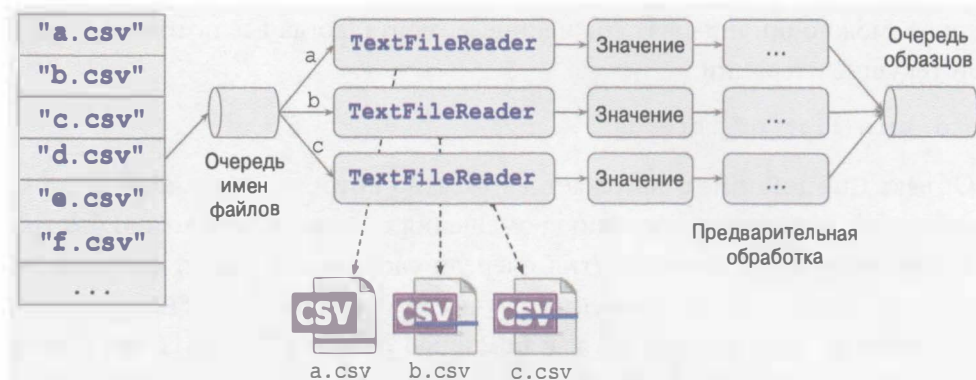


Рис. 12.10. Одновременное чтение из множества файлов

В таком случае нам понадобится написать небольшую функцию для создания считывателя и узлов, которая будет читать один образец и помещать его в очередь образцов:

```
def read_and_push_instance(filename_queue, instance_queue):
    reader = tf.TextLineReader(skip_header_lines=1)
    key, value = reader.read(filename_queue)
    x1, x2, target = tf.decode_csv(value, record_defaults=[[-1.],
                                                            [-1.], [-1.]])
    features = tf.stack([x1, x2])
    enqueue_instance = instance_queue.enqueue([features, target])
    return enqueue_instance
```

Затем мы определяем очереди:

```
filename_queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string],
                             shapes=[()])
filename = tf.placeholder(tf.string)
enqueue_filename = filename_queue.enqueue([filename])
close_filename_queue = filename_queue.close()
instance_queue = tf.RandomShuffleQueue(...)
```

И, наконец, мы создаем объект `QueueRunner`, но на этот раз передаем ему список отличающихся операций помещения в очередь.

Каждая операция использует другой считыватель, так что потоки будут одновременно читать из разных файлов:

```
read_and_enqueue_ops = [  
    read_and_push_instance(filename_queue, instance_queue)  
    for i in range(5)]  
queue_runner = tf.train.QueueRunner(instance_queue,  
                                    read_and_enqueue_ops)
```

Стадия выполнения остается такой же, как ранее: в очередь помещаются имена файлов, подлежащих чтению, создается объект `Coordinator`, а затем создаются и запускаются потоки `QueueRunner`. Теперь все потоки будут читать из разных файлов одновременно, пока все файлы не будут полностью считаны, после чего `QueueRunner` закроет очередь образцов, чтобы остальные операции извлечения из этой очереди не блокировались.

Другие удобные функции

Библиотека TensorFlow также предлагает удобные функции, позволяющие упростить выполнение ряда общих задач при чтении обучающих образцов. Мы кратко рассмотрим лишь некоторые из них (полный список доступен в документации по API-интерфейсу).

Функция `string_input_producer()` принимает одномерный тензор, содержащий список имен файлов, создает поток, который помещает в очередь имен файлов по одному имени файла за раз, и затем закрывает очередь. Если указать число эпох, она будет циклически проходить по именам файлов раз за эпоху, прежде чем закрывать очередь. По умолчанию функция тасует имена файлов на каждой эпохе. Она создает объект `QueueRunner` для управления своим потоком и добавляет его в коллекцию `GraphKeys.QUEUE_RUNNERS`. Чтобы запустить каждый объект `QueueRunner` в этой коллекции, можно вызвать функцию `tf.train.start_queue_runners()`. Обратите внимание, что если вы забудете запустить `QueueRunner`, то очередь имен файлов будет открытой и пустой, а считыватели — заблокированными навсегда.

Существует несколько других *функций-генераторов* (*producer*), которые похожим образом создают очередь и соответствующий объект `QueueRunner` для выполнения операции помещения в очередь (например, `input_producer()`, `range_input_producer()` и `slice_input_producer()`).

Функция `shuffle_batch()` принимает список тензоров (скажем, `[features, target]`) и создает:

- объект `RandomShuffleQueue`;
- объект `QueueRunner` для помещения тензоров в очередь (добавленный в коллекцию `GraphKeys.QUEUE_RUNNERS`);
- операцию `dequeue_many` для извлечения мини-пакета из очереди.

Функция `shuffle_batch()` облегчает управление внутри одиночного процесса многопоточным конвейером ввода, наполняющим очередь, и конвейером обучения, читающим мини-пакеты из этой очереди. Взгляните также на функции `batch()`, `batch_join()` и `shuffle_batch_join()`, которые предоставляют аналогичную функциональность.

Хорошо! У вас есть все инструменты, необходимые для того, чтобы начать эффективное обучение и запуск нейронных сетей на множестве устройств и серверов в кластере TensorFlow. Давайте посмотрим, чему вы научились:

- применять устройства ГП;
- настраивать и запускать кластер TensorFlow;
- распределять вычисления между множеством устройств и серверов;
- разделять переменные (и другие операции, поддерживающие состояние, такие как очереди и считыватели) между сеансами с использованием контейнеров;
- координировать асинхронную работу множества графов с применением очередей;
- эффективно читать входные данные с использованием считывателей, класса `QueueRunner` и класса `Coordinator`.

А теперь давайте применим все это для распараллеливания нейронных сетей!

Распараллеливание нейронных сетей в кластере TensorFlow

В настоящем разделе мы сначала выясним, как распараллелить несколько нейронных сетей, просто помещая каждую сеть в отдельное устройство. Затем мы рассмотрим гораздо более сложную задачу обучения единственной нейронной сети на множестве устройств и серверов.

Одна нейронная сеть на устройство

Самый тривиальный способ обучения и запуска нейронных сетей в кластере TensorFlow предусматривает использование того же самого кода, который применялся бы для единственного устройства на единственной машине, и указание адреса сервера мастера при создании сеанса. Вот и все — работа сделана! Ваш код будет выполняться на стандартном устройстве этого сервера. Вы можете изменить устройство, которое будет прогонять граф, просто помещая стадию построения вашего кода внутрь блока устройства.

За счет запуска нескольких клиентских сеансов параллельно (в разных потоках или процессах), подключения их к разным серверам и настройки на использование разных устройств вы можете довольно легко обучать и запускать много нейронных сетей в параллельном режиме на всех устройствах и машинах в кластере (рис. 12.11). Ускорение будет почти линейным⁴. Обучение 100 нейронных сетей на 50 серверах с 2 графическими процессорами у каждого будет не намного дольше, чем обучение лишь одной нейронной сети на одном ГП.

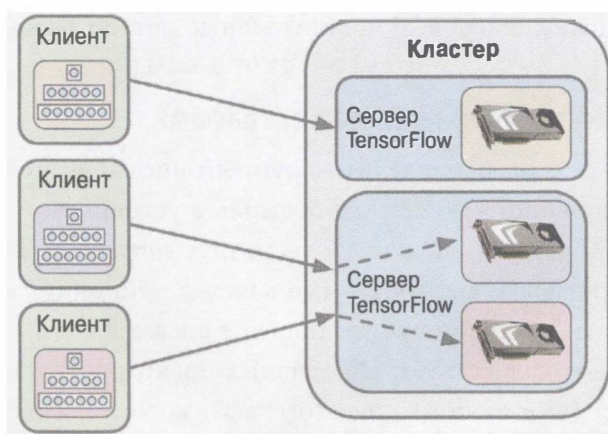


Рис. 12.11. Обучение одной нейронной сети на устройство

Такое решение идеально подходит для подстройки гиперпараметров: каждое устройство в кластере будет обучать отличающуюся модель с собственным набором гиперпараметров. Чем большая вычислительная мощность вам доступна, тем большее пространство гиперпараметров вы сможете исследовать.

⁴ Не на 100% линейное, если вы ожидаете завершения всех устройств, т.к. суммарным временем будет время работы самого медленного устройства.

Решение также прекрасно работает в ситуации, когда у вас есть веб-служба, которая принимает большое количество *запросов в секунду* (*Queries Per Second* — *QPS*), и вы хотите, чтобы нейронная сеть вырабатывала прогноз для каждого запроса. Просто реплицируйте нейронную сеть на всех устройствах в кластере и распределяйте запросы между всеми устройствами. Добавляя дополнительные серверы, вы сможете обрабатывать неограниченное количество QPS (тем не менее, это не сократит время, необходимое для обработки одиночного запроса, т.к. он по-прежнему должен ожидать выработки прогноза нейронной сетью).



Другой вариант заключается в обслуживании нейронной сети с применением *TensorFlow Serving* — системы с открытым кодом, выпущенной Google в феврале 2016 года, которая предназначена для обслуживания большого объема запросов к моделям МО (обычно построенным с помощью TensorFlow). Она поддерживает контроль версий моделей, поэтому вы можете легко развертывать новую версию сети в производственной среде либо экспериментировать с различными алгоритмами, не прерывая обслуживание. Кроме того, система TensorFlow Serving способна выдерживать значительную нагрузку за счет увеличения числа серверов. Дополнительные детали ищите по адресу <https://www.tensorflow.org/serving/>.

Репликация внутри графа или между графами

Вы можете также распараллелить крупный ансамбль нейронных сетей, помещая каждую нейронную сеть на отдельное устройство (ансамбли были введены в главе 7). Однако поскольку вы хотите *запускать* ансамбль, то индивидуальные прогнозы, выработанные каждой нейронной сетью, понадобится агрегировать, чтобы получить прогноз ансамбля, а это требует некоторой координации. Существуют два главных подхода к поддержке ансамбля нейронных сетей (или любого другого графа, который содержит крупные порции независимых вычислений).

- Можно создать один большой граф, содержащий все нейронные сети, каждая из которых прикреплена к своему устройству, плюс все вычисления, необходимые для агрегирования индивидуальных прогнозов от всех нейронных сетей (рис. 12.12). Затем нужно просто создать один сеанс на любом сервере в кластере и позволить ему позаботиться обо всем остальном (включая ожидание доступности индивидуальных прогнозов перед их агрегированием). Такой подход называется *репликацией внутри графа* (*in-graph replication*).

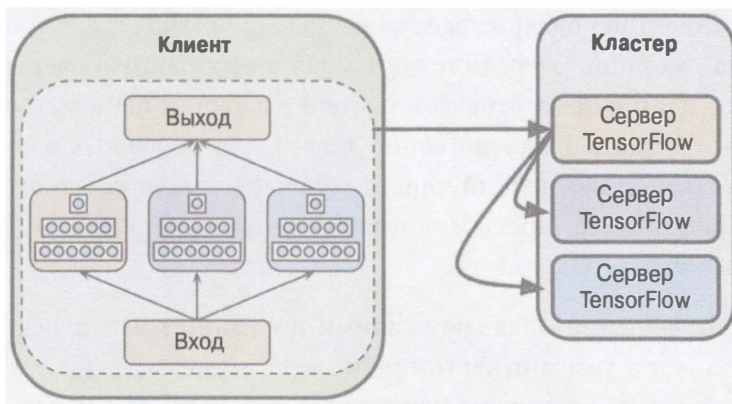


Рис. 12.12. Репликация внутри графа

- В качестве альтернативы можно создать по одному отдельному графу для каждой нейронной сети и самостоятельно поддерживать синхронизацию между такими графами. Такой подход называется *репликацией между графами* (*between-graph replication*). Типовая реализация предусматривает координирование прогона этих графов с использованием очередей (рис. 12.13).

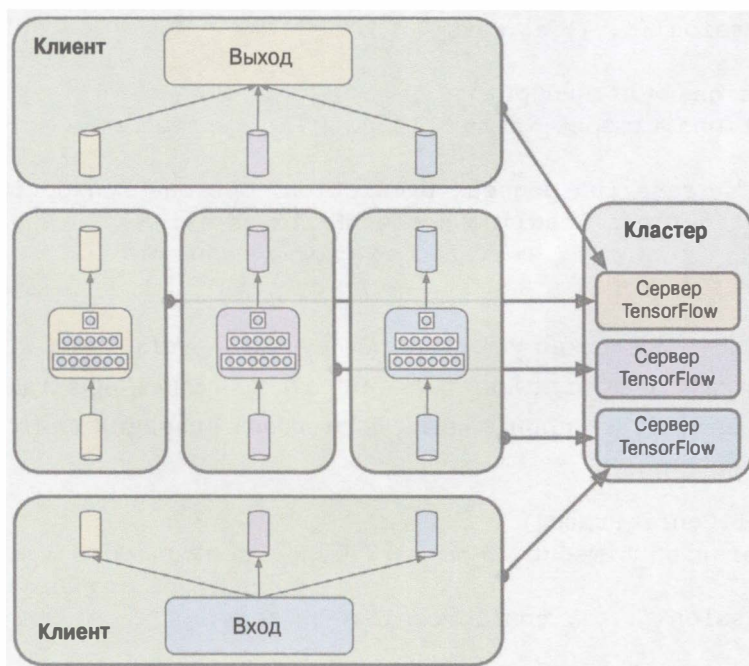


Рис. 12.13. Репликация между графами

Набор клиентов поддерживает каждую нейронную сеть, читая из ее выделенной входной очереди и записывая в выделенную очередь прогнозов. Еще один клиент отвечает за чтение входных данных и помещение их во все входные очереди (копируя все входные данные в каждую очередь). Наконец, последний клиент отвечает за чтение одного прогноза из каждой очереди прогнозов и их агрегирование для получения прогноза ансамбля.

Описанные решения обладают своими достоинствами и недостатками. Репликация внутри графа отчасти проще в реализации, потому что не приходится управлять множеством клиентов и очередей. Тем не менее, репликацию между графами несколько легче организовать в хорошо связанные и простые в тестировании модули.

Кроме того, она обеспечивает более высокую гибкость. Например, к клиенту агрегирования можно было бы добавить тайм-аут операции извлечения из очереди, чтобы ансамбль не терпел неудачу, когда одна из нейронных сетей выходит из строя или требует слишком много времени на выработку своего прогноза. Библиотека TensorFlow позволяет указывать тайм-аут при вызове функции `run()`, передавая объект `RunOptions` с `timeout_in_ms`:

```
with tf.Session([...]) as sess:
    [...]
    run_options = tf.RunOptions()
    run_options.timeout_in_ms = 1000 # 1-секундный тайм-аут
    try:
        pred = sess.run(dequeue_prediction, options=run_options)
    except tf.errors.DeadlineExceededError as ex:
        [...] # спустя 1 секунду происходит тайм-аут
               # операции извлечения из очереди
```

Время тайм-аута можно указать и по-другому — установить конфигурационный параметр `operation_timeout_in_ms` сеанса, но тогда тайм-аут функции `run()` будет происходить, если *любая* операция длится дольше времени тайм-аута:

```
config = tf.ConfigProto()
config.operation_timeout_in_ms = 1000 # 1-секундный тайм-аут
                                     # для любой операции
with tf.Session([...], config=config) as sess:
    [...]
    try:
        pred = sess.run(dequeue_prediction)
```

```
except tf.errors.DeadlineExceededError as ex:
    [...] # спустя 1 секунду происходит тайм-аут
           # операции извлечения из очереди
```

Параллелизм модели

До сих пор мы запускали каждую нейронную сеть на одиночном устройстве. А что, если вы хотите запускать единственную нейронную сеть на множестве устройств? В таком случае модель потребуется разбить на отдельные порции и запускать каждую порцию на отдельном устройстве. Это называется *параллелизмом модели* (*model parallelism*). К сожалению, параллелизм модели оказывается очень сложным и на самом деле зависящим от архитектуры имеющейся нейронной сети. Для полносвязных сетей такой подход обычно дает немного (рис. 12.14).

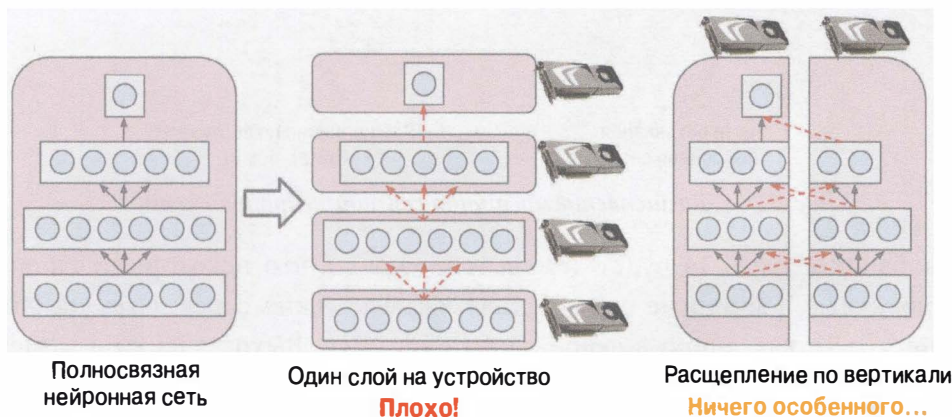


Рис. 12.14. Расщепление полносвязной нейронной сети

Может показаться, что легкий способ расщепления модели заключается в размещении каждого слоя на отдельном устройстве, но он не работает, поскольку каждому слою придется ожидать выходных данных предыдущего слоя, прежде чем он сможет делать что-либо. Так может быть модель можно расщепить по вертикали, скажем, размещая левую половину каждого слоя на одном устройстве, а правую — на другом? Чуть лучше, т.к. обе половины каждого слоя действительно могут работать параллельно, но проблема в том, что каждая половина следующего слоя требует выходных данных обеих половин текущего слоя, а потому будет возникать много коммуникаций между устройствами (представлены пунктирными линиями со стрелками). Скорее всего, это полностью сведет на нет выгоду от параллельного вычисления,

потому что коммуникации между устройствами являются медленными (особенно если они происходят между отдельными машинами).

Однако, как будет показано в главе 13, ряд архитектур нейронных сетей, такие как сверточные нейронные сети, содержат слои, которые только частично связаны с более низкими слоями, поэтому эффективно распределять порции между устройствами гораздо легче (рис. 12.15).

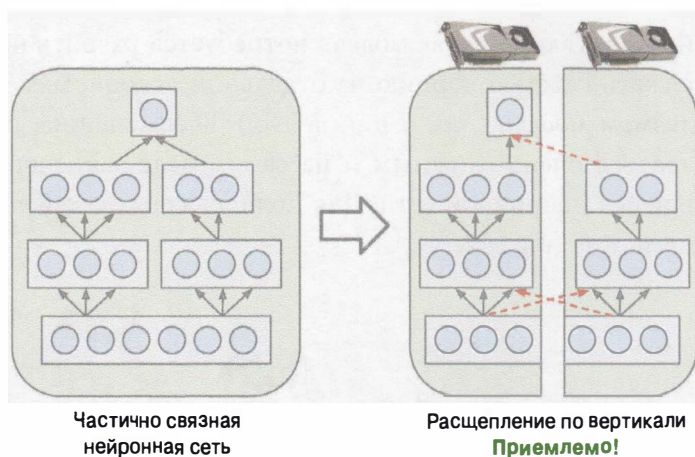


Рис. 12.15. Расщепление частично связанной нейронной сети

Вдобавок в главе 14 будет демонстрироваться, что некоторые глубокие рекуррентные нейронные сети состоят из нескольких слоев *ячеек памяти* (*memory cell*), как видно в левой части рис. 12.16. Выход ячейки в момент времени t передается обратно на ее вход в момент времени $t + 1$ (что можно заметить в правой части рис. 12.16).

Если расщепить такую сеть горизонтально, размещая каждый слой на своем устройстве, тогда на первом шаге активным будет только одно устройство, на втором шаге — два устройства, а ко времени распространения сигнала до выходного слоя все устройства будут активными одновременно. По-прежнему происходит много коммуникаций между устройствами, но из-за того, что каждая ячейка может быть довольно сложной, выгода от выполнения множества ячеек в параллельном режиме часто перевешивает штраф в виде коммуникаций.

Короче говоря, параллелизм модели способен ускорить выполнение или обучение нейронных сетей определенных типов, но не всех. Он требует специального внимания и подстройки, скажем, чтобы большинство устройств, которые нуждаются в коммуникациях, функционировало на той же самой машине.

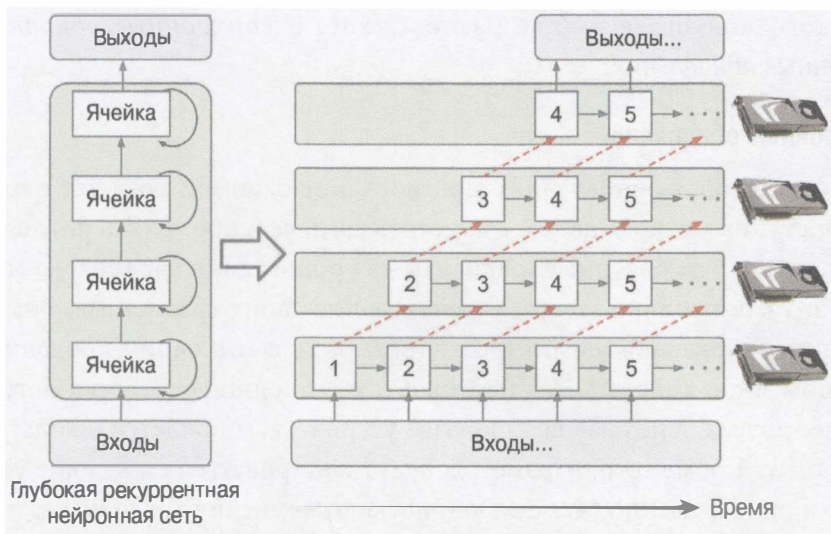


Рис. 12.16. Расщепление глубокой рекуррентной нейронной сети

Параллелизм данных

Существует еще один способ распараллеливания обучения нейронной сети. Он предполагает репликацию сети на каждое устройство, запуск шага обучения одновременно на всех репликах, применяя для каждой отличающийся мини-пакет, и агрегирование градиентов с целью обновления параметров модели. Подход называется *параллелизмом данных* (*data parallelism*) и представлен на рис. 12.17.

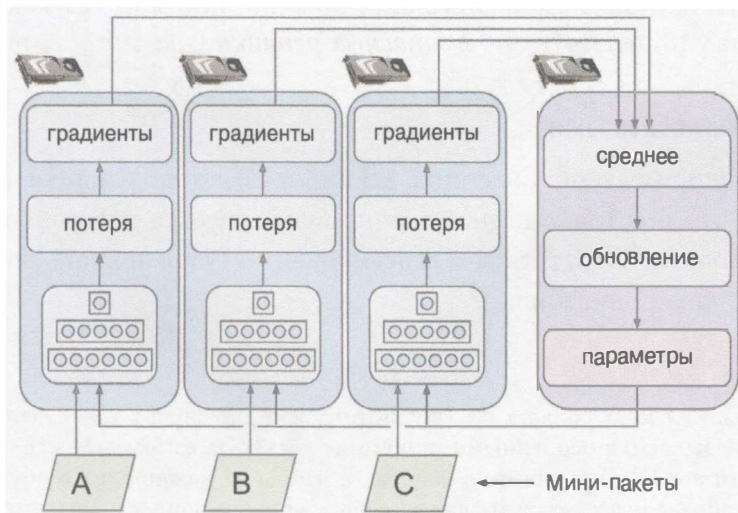


Рис. 12.17. Параллелизм данных

У такого подхода есть две разновидности: *синхронные обновления* и *асинхронные обновления*.

Синхронные обновления

При синхронных обновлениях агрегирование ожидает, пока все градиенты станут доступными, прежде чем вычислить среднее и применить результат (т.е. использовать агрегированные градиенты для обновления параметров модели). После того как реплика завершит вычисление своих градиентов, она обязана ожидать обновления параметров, чтобы быть в состоянии продолжить со следующим мини-пакетом. Недостаток в том, что одни устройства могут быть медленнее других, а потому всем другим устройствам придется ожидать их на каждом шаге. Кроме того, параметры будут копироваться на каждое устройство почти одновременно (немедленно после применения градиентов), что может привести к насыщению полосы пропускания серверов параметрами.



Чтобы сократить время ожидания на каждом шаге, вы могли бы проигнорировать градиенты из нескольких самых медленных реплик (как правило, ~10%). Например, вы могли бы запускать 20 реплик, но на каждом шаге агрегировать градиенты только из 18 наиболее быстрых реплик и проигнорировать градиенты из оставшихся 2.

Как только параметры обновлены, первые 18 реплик могут возобновить работу немедленно, не ожидая 2 самых медленных реплик. Такая конфигурация обычно описывается как имеющая 18 реплик плюс 2 *запасных реплики* (*spare replica*)⁵.

Асинхронные обновления

При асинхронных обновлениях всякий раз, когда реплика завершает вычисление градиентов, они немедленно используются для обновления параметров модели. Отсутствуют агрегирование (удален шаг “среднее” на рис. 12.17) и синхронизация.

⁵ Такое название слегка сбивает с толку, потому что оно звучит так, будто некоторые реплики оказываются особенными, ничего не делая. На самом деле все реплики эквивалентны: все они напряженно работают, чтобы на каждом шаге обучения быть в числе наиболее быстрых, и на каждом шаге проигравшие меняются (если только некоторые устройства не являются по-настоящему медленнее остальных).

Реплики просто работают независимо друг от друга. Поскольку нет никакого ожидания других реплик, такой подход позволяет выполнять больше шагов обучения в минуту. Кроме того, хотя параметры по-прежнему необходимо копировать на все устройства на каждом шаге, для каждой реплики это происходит в разное время, а потому риск насыщения полосы пропускания сокращается.

Параллелизм данных с асинхронными обновлениями — привлекательный вариант из-за своей простоты, отсутствия задержки синхронизации и лучшего потребления полосы пропускания. Тем не менее, несмотря на достаточно хорошую работу на практике, больше всего удивляет то, что он вообще работает!

Действительно, к тому времени, когда реплика завершает вычисление градиентов на основе некоторых значений параметров, эти параметры несколько раз обновятся другими репликами (в среднем $N - 1$ раз при наличии N реплик), и нет никакой гарантии, что вычисленные градиенты все еще будут указывать в правильном направлении (рис. 12.18).

Сильно просроченные градиенты называют *устаревшими градиентами* (*stale gradient*): они могут замедлить сходжение, привносить эффекты шума и раскачивания (кривая обучения может содержать временные колебания) или даже делать алгоритм обучения расходящимся.

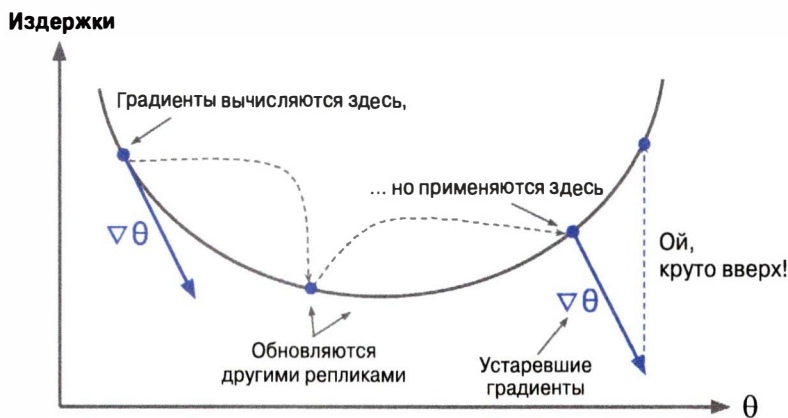


Рис. 12.18. Устаревшие градиенты во время применения асинхронных обновлений

Сократить влияние устаревших градиентов можно несколькими способами.

- Уменьшить скорость обучения.
- Отбросить устаревшие градиенты или понизить их масштаб.
- Скорректировать размер мини-пакетов.
- Начать первые несколько эпох с использованием только одной реплики (называется *стадией разогрева* (*warmup phase*)). Устаревшие градиенты имеют склонность быть более разрушительными в начале обучения, когда градиенты обычно большие и параметры еще не попали во впадину функции издержек, поэтому разные реплики могут продвигать параметры в совершенно разных направлениях.

В работе, опубликованной командой Google Brain в апреле 2016 года (<http://goo.gl/9GCiPb>), сравнивались различные подходы. В ней сделан вывод о том, что подход с параллелизмом данных и синхронными обновлениями, применяющий несколько запасных реплик, был самым эффективным, не только сходясь быстрее, но давая в результате лучшую модель. Однако это по-прежнему область активных исследований, так что полностью исключать асинхронные обновления пока не стоит.

Насыщение полосы пропускания

Независимо от того, какие обновления используются, синхронные или асинхронные, параллелизм данных все же требует передачи параметров модели из серверов параметров в каждую реплику в начале каждого шага обучения и градиентов в другом направлении в конце каждого шага обучения. К сожалению, это означает, что всегда наступает момент, когда добавление дополнительного ГП вообще не улучшит производительность, т.к. время, затрачиваемое на перемещение данных в и из оперативной памяти ГП (и возможно через сеть), будет перевешивать ускорение, достигнутое расщеплением вычислительной нагрузки. В такой точке добавление дополнительных ГП просто усилит насыщение и замедлит обучение.



Некоторые модели, обычно относительно небольшие и обученные на очень крупном обучающем наборе, часто лучше обучать на единственной машине с одним ГП.

Насыщение является более серьезным для больших плотных моделей, поскольку они имеют много параметров и градиентов, подлежащих пере-

мещению. Оно менее серьезно для мелких моделей (но выигрыш от распараллеливания мал) и также для крупных разреженных моделей, потому что обычно градиенты по большей части нулевые, поэтому могут передаваться эффективным образом. Джефф Дин, инициатор и руководитель проекта Google Brain, рассказал (<http://goo.gl/E4ypxo>) о типичном ускорении в 25–40 раз при распределении вычислений между 50 ГП для плотных моделей и в 300 раз для более разреженных моделей, обучаемых на 500 ГП. Как видите, разреженные модели действительно масштабируются лучше. Ниже приведено несколько конкретных примеров.

- Neural Machine Translation (нейронный машинный перевод): 6-кратное ускорение на 8 ГП.
- Inception/ImageNet: 32-кратное ускорение на 50 ГП.
- RankBrain: 300-кратное ускорение на 500 ГП.

Указанные числа представляют положение дел на первый квартал 2016 года. За пределами нескольких десятков ГП для плотной модели или нескольких сотен ГП для разреженной модели наступает насыщение и производительность ухудшается. В поисках решения данной проблемы ведется много изысканий (исследование возможности использования децентрализованных архитектур вместо централизованных серверов параметров, применение сжатия моделей с потерями, оптимизация того, когда и для каких реплик необходимо обмениваться информацией, и т.д.), поэтому за ближайшие несколько лет, скорее всего, будет сделан большой прогресс в области распараллеливания нейронных сетей.

Между тем есть ряд простых шагов, которые можно предпринять для ослабления проблемы насыщения.

- Группирование графических процессоров на нескольких серверах, а не их рассредоточение по множеству серверов. Это устранил нежелательные повторные передачи через сеть.
- Фрагментация параметров среди множества серверов параметров (как обсуждалось ранее).
- Понижение точности значений с плавающей точкой для параметров модели с 32 битов (`tf.float32`) до 16 битов (`tf.bfloat16`). Это вдвое сократит объем передаваемых данных, не сильно влияя на скорость схождения или производительность модели.



Хотя 16-битовая точность представляет собой минимум для обучения нейронной сети, в действительности после обучения вы можете понизить точность до 8-битовой, чтобы сократить размер модели и ускорить вычисления. Прием называется *квантованием* (*quantizing*) нейронной сети. Он особенно полезен для развертывания и запуска заранее обученных моделей на мобильных телефонах. Ознакомьтесь с великолепной публикацией в блоге Пита Уордена (<http://goo.gl/09Cb6v>), посвященной этой теме.

Реализация с помощью TensorFlow

Чтобы реализовать параллелизм данных, используя TensorFlow, сначала понадобится выбрать нужный вид репликации (внутри графа или между графами) и обновлений (синхронные или асинхронные). Давайте посмотрим, как можно было бы реализовать каждую комбинацию (завершенные примеры кода ищите в упражнениях и тетрадах Jupyter).

В случае репликации внутри графа и синхронных обновлений вы строите один большой граф, который содержит все реплики модели (размещенные на разных устройствах), а также несколько узлов для агрегирования всех градиентов и их передачи оптимизатору. Ваш код открывает сеанс в кластере и просто многократно запускает операцию обучения.

В случае репликации внутри графа и асинхронных обновлений вы также строите один большой граф, но с одним оптимизатором на реплику, и открываете по одному потоку на реплику, который многократно запускает оптимизатор реплики.

В случае репликации между графами и асинхронных обновлений вы запускаете множество независимых клиентов (обычно в отдельных процессах), каждый из которых обучает реплику модели, как если бы она была единственной в мире, но параметры фактически разделяются с другими репликами (с применением контейнера ресурсов).

В случае репликации между графами и синхронных обновлений вы снова запускаете множество клиентов, каждый из которых обучает реплику модели на основе разделяемых параметров, но на этот раз оптимизатор (скажем, `MomentumOptimizer`) помещается внутрь оболочки `SyncReplicasOptimizer`. Каждая реплика использует такой оптимизатор, как любой другой, но внутренне данный оптимизатор управляет градиен-

ты набору очередей (по одной на переменную), которые читаются оптимизатором `SyncReplicasOptimizer` одной из реплик, называемым *старшим* (*chief*). Старший агрегирует градиенты и применяет их, после чего записывает маркер в *очередь маркеров* (*token queue*) для каждой реплики, сигнализируя о том, что она может двигаться дальше и вычислять следующие градиенты. Такой подход поддерживает наличие *запасных реплик* (*spare replica*).

Если вы займетесь упражнениями, то реализуете все четыре решения. Вы легко сможете воспользоваться обретенными знаниями для обучения крупных глубоких нейронных сетей на десятках серверов и графических процессоров! В последующих главах мы рассмотрим несколько более важных архитектур нейронных сетей и затем приступим к обучению с подкреплением.

Упражнения

1. Если при запуске программы TensorFlow вы получили ошибку `CUDA_ERROR_OUT_OF_MEMORY`, то что, скорее всего, произошло? Что вы можете с этим поделать?
2. В чем разница между прикреплением операции к устройству и размещением операции на устройстве?
3. Если вы работаете с установленной копией TensorFlow, поддерживающей графические процессоры, и применяете стандартное размещение, то разместятся ли все операции на первом устройстве ГП?
4. Если вы прикрепляете переменную к `"/gpu:0"`, то может ли она использоваться операциями, размещенными на `/gpu:1`? Операциями, размещенными на `"/cpu:0"`? Операциями, прикрепленными к устройствам, которые находятся на других серверах?
5. Могут ли две операции, размещенные на одном устройстве, выполняться в параллельном режиме?
6. Что такое зависимость управления и когда ее необходимо применять?
7. Пусть вы целыми днями обучали сеть DNN в кластере TensorFlow и сразу после окончания программы обучения обнаружили, что забыли сохранить модель с использованием `Saver`. Утрачена ли ваша обученная модель?

8. Обучите несколько сетей DNN параллельно в кластере TensorFlow, применяя разные значения гиперпараметров. Это могут быть сети DNN для классификации MNIST или любой другой задачи, которая вас интересует. Простейший вариант предусматривает написание единственной клиентской программы, которая обучает только одну сеть DNN, и запуск этой программы во множестве процессов параллельно с отличающимися значениями гиперпараметров для каждого клиента. Программа обязана иметь параметры командной строки для управления тем, на какой сервер и устройство должна размещаться сеть DNN, а также какой контейнер ресурсов и значения гиперпараметров использовать (удостоверьтесь, что для каждой сети DNN применяется свой контейнер ресурсов). С помощью проверочного набора или перекрестной проверки выберите лучшие три модели.
9. Создайте ансамбль, используя лучшие три модели из предыдущего упражнения. Определите его в одиночном графе, обеспечив запуск каждой сети DNN на своем устройстве. Оцените ансамбль на проверочном наборе: работает ли ансамбль лучше индивидуальных сетей DNN?
10. Обучите сеть DNN с применением репликации между графами и параллелизма данных с асинхронными обновлениями, измеряя время достижения ею удовлетворительной производительности. Затем проведите обучение с использованием синхронных обновлений. Позволили ли синхронные обновления получить лучшую модель? Стало ли обучение быстрее? Расщепите сеть DNN по вертикали, разместите каждый вертикальный срез на своем устройстве и снова обучите модель. Стало ли обучение сколько-нибудь быстрее? Заметны ли какие-то изменения в производительности?

Решения приведенных упражнений доступны в приложении А.

Сверточные нейронные сети

Хотя суперкомпьютер IBM Deep Blue победил чемпиона мира по шахматам Гарри Каспарова еще в 1996 году, до недавнего времени компьютеры не были способны надежно выполнять на вид простые задачи, такие как обнаружение щенка на фотографии или распознавание сказанных слов. Почему эти задачи настолько легки для нас, людей? Ответ заключается в том, что восприятие в основном происходит за пределами нашего сознания, внутри специализированных зрительных, слуховых и других сенсорных модулей в наших мозгах. К тому времени, когда сенсорная информация достигает нашего сознания, она уже оснащена высокоуровневыми признаками; например, глядя на фотографию забавного щенка, вы не в состоянии сделать выбор *не* видеть щенка или *не* заметить, что он забавный. Вы также не можете объяснить, *как* вы распознали забавного щенка; для вас это просто очевидно. Таким образом, мы не можем доверять своему субъективному опыту: восприятие вообще не является тривиальным и для его понимания мы должны посмотреть, как работают сенсорные модули.

Сверточные нейронные сети (*Convolutional Neural Network — CNN*) появились в результате изучения зрительной коры головного мозга и применялись в распознавании изображений, начиная с 1980-х годов. Благодаря росту вычислительной мощности в последние несколько лет, увеличению объема доступных обучающих данных и появлению трюков для обучения глубоких сетей, которые были представлены в главе 11, сетям CNN удалось достичь сверхчеловеческой производительности при решении ряда сложных зрительных задач. Они приводят в действие мощные службы поиска изображений, беспилотные автомобили, системы автоматической классификации видеороликов и т.п. Кроме того, сети CNN не ограничиваются зрительным восприятием: они также успешно решают другие задачи вроде *распознавания речи* (*voice recognition*) или *обработки естественного языка* (*Natural Language Processing — NLP*); однако пока что мы сосредоточимся на зрительных применениях.

В настоящей главе мы расскажем, откуда произошли сети CNN, как выглядят их строительные блоки и каким образом реализовать их с использованием TensorFlow. Затем мы представим несколько наилучших архитектур сетей CNN.

Строение зрительной коры головного мозга

Дэвид Х. Хьюбел и Торстен Визель провели серию экспериментов на кошках в 1958¹ и 1959² годах (а спустя несколько лет также на обезьянах³), выявив важнейшие сведения о структуре зрительной коры головного мозга (в 1981 году авторы получили за свою работу Нобелевскую премию в области физиологии или медицины). В частности, они показали, что многие нейроны в зрительной коре имеют небольшое *локальное рецепторное поле* (*local receptive field*), а потому реагируют на зрительные раздражители, находящиеся в ограниченной области поля зрения (на рис. 13.1 локальные рецепторные поля пяти нейронов представлены пунктирными окружностями). Рецепторные поля разных нейронов могут перекрываться и вместе они охватывают все поле зрения. Вдобавок авторы продемонстрировали, что некоторые нейроны реагируют только на изображения горизонтальных линий, в то время как другие — на линии в разных направлениях (два нейрона могут иметь одно и то же рецепторное поле, но реагировать на линии с отличающимися направлениями). Они также заметили, что некоторые нейроны обладают большими рецепторными полями и реагируют на более сложные образы, являющиеся комбинациями низкоуровневых образов. Эти наблюдения привели к мысли о том, что нейроны более высокого уровня основываются на выходах соседствующих нейронов более низкого уровня (обратите внимание на рис. 13.1, что каждый нейрон связан только с несколькими нейронами из предыдущего уровня). Такая мощная архитектура способна обнаруживать все виды сложных образов в любой области поля зрения.

¹ “Single Unit Activity in Striate Cortex of Unrestrained Cats” (“Единичная активность в полосатой коре естественных кошек”), Д. Хьюбел и Т. Визель (1958 год) (<http://goo.gl/VLxXf9>).

² “Receptive Fields of Single Neurones in the Cat’s Striate Cortex” (“Рецепторные поля одиночных нейронов в полосатой коре кошки”), Д. Хьюбел и Т. Визель (1959 год) (<http://goo.gl/OYuFUZ>).

³ “Receptive Fields and Functional Architecture of Monkey Striate Cortex” (“Рецепторные поля и функциональная архитектура полосатой коры обезьян”), Д. Хьюбел и Т. Визель (1968 год) (<http://goo.gl/95F7QH>).

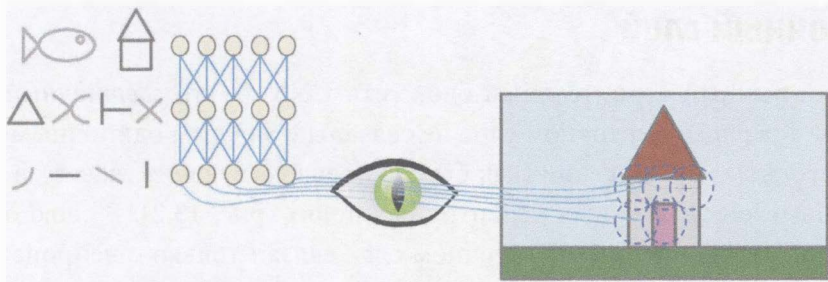


Рис. 13.1. Локальные рецепторные поля в зрительной коре

Проведенные исследования зрительной коры вдохновили на создание в 1980 году *неокогнитрона* (*neocognitron*)⁴, который постепенно развился в то, что сейчас мы называем *сверточными нейронными сетями*. Важной вехой стала работа 1998 года⁵ Яна Лекуна, Леона Ботту, Йошуа Бенджи и Патрика Хаффнера, в которой была введена знаменитая архитектура *LeNet-5*, широко применяемая для распознавания рукописных чисел на чеках. Эта архитектура содержит ряд строительных блоков, которые вам уже известны, в том числе полносвязные слои и сигмоидальные функции активации, но она также представляет два новых строительных блока: *сверточные слои* (*convolutional layer*) и *объединяющие слои* (*pooling layer*). Давайте рассмотрим их.



Почему для решения задач распознавания изображений просто не воспользоваться обыкновенной глубокой нейронной сетью с полносвязными слоями? К сожалению, хотя она хорошо работает для небольших изображений (скажем, MNIST), в случае более крупных изображений ее работа нарушается из-за гигантского количества требующихся параметров. Например, изображение 100×100 содержит 10 000 пикселей, и если первый слой имеет лишь 1 000 нейронов (что уже серьезно ограничивает объем информации, передаваемой следующему слою), то в итоге получится 10 миллионов связей. И это только первый слой. Сети CNN решают такую проблему с применением частично связных слоев.

⁴ “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position” (“Неокогнитрон: самоорганизующаяся модель нейронной сети для механизма распознавания образов, не подверженная влиянию сдвига в позиции”), К. Фукушима (1980 год) (<http://goo.gl/XwiXs9>).

⁵ “Gradient-Based Learning Applied to Document Recognition” (“Основанное на градиентах обучение, примененное к распознаванию документов”), Я. Лекун и др. (1998 год) (<http://goo.gl/A347S4>).

Сверточный слой

Самый важный строительный блок сети CNN — это *сверточный слой*⁶: нейроны в первом сверточном слое не связаны с каждым одиночным пикселем во входном изображении (как было в предшествующих главах), а только с пикселями в собственных рецепторных полях (рис. 13.2). В свою очередь каждый нейрон во втором сверточном слое связан только с нейронами, находящимися внутри небольшого прямоугольника в первом слое. Такая архитектура позволяет сети сосредоточиться на низкоуровневых признаках в первом скрытом слое, затем скомпоновать их в признаки более высокого уровня в следующем скрытом слое и т.д. Подобная иерархическая структура распространена в реальных изображениях, что и является одной из причин, почему сети CNN настолько хорошо работают при распознавании изображений.

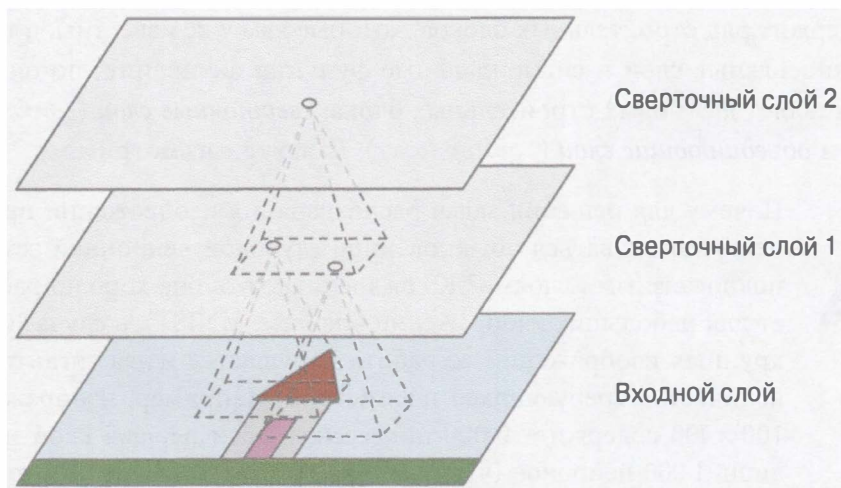


Рис. 13.2. Слои сети CNN с прямоугольными локальными рецепторными полями

⁶ *Свертка* — это математическая операция, которая плавно перемещает одну функцию по другой и измеряет интеграл их точечного умножения. Она имеет глубинные связи с преобразованием Фурье и преобразованием Лапласа и интенсивно используется в обработке сигналов. Сверточные слои на самом деле применяют взаимную корреляцию, которая очень похожа на свертку (за дополнительными сведениями обращайтесь по адресу <http://goo.gl/NAfxXd>).



Все рассмотренные до сих пор многослойные нейронные сети имели слои, состоящие из длинной линейки нейронов, и нам приходилось разглаживать входные изображения до одного измерения, прежде чем передавать их нейронной сети. Теперь каждый слой представлен в двух измерениях, что облегчает сопоставление нейронов с соответствующими им входами.

Нейрон, расположенный в строке i и столбце j заданного слоя, связывается с выходами нейронов предыдущего слоя, которые находятся в строках с i по $i + f_h - 1$ и столбцах с j по $j + f_w - 1$, где f_h и f_w — высота и ширина рецепторного поля (рис. 13.3). Для того чтобы слой имел такую же высоту и ширину, как у предыдущего слоя, вокруг входов обычно добавляют нули, что и видно на рис. 13.3. Это называется *дополнением нулями* (*zero padding*).

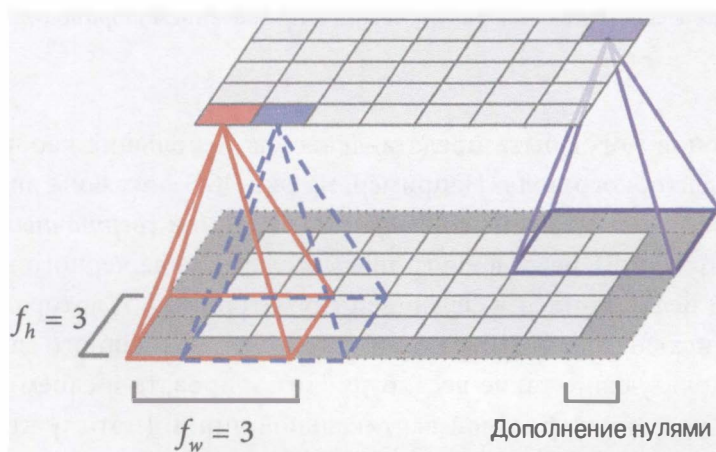


Рис. 13.3. Связи между слоями и дополнение нулями

Также можно связать крупный входной слой с гораздо меньшим слоем, растягивая рецепторные поля (рис. 13.4). Расстояние между двумя последовательными рецепторными полями называется *страйдом* (*stride*), или большим шагом. На рис. 13.4 входной слой 5×7 (плюс дополнение нулями) связывается со слоем 3×4 , используя рецепторные поля 3×3 и страйд 2 (в данном примере страйд одинаков в двух направлениях, но он вовсе не обязан быть таким). Нейрон, расположенный в строке i и столбце j более высокого слоя, связывается с выходами нейронов предыдущего слоя, которые находятся в строках с $i \times s_h$ по $i \times s_h + f_h - 1$ и столбцах с $j \times s_w$ по $j \times s_w + f_w - 1$, где s_h и s_w — вертикальный и горизонтальный страйды.

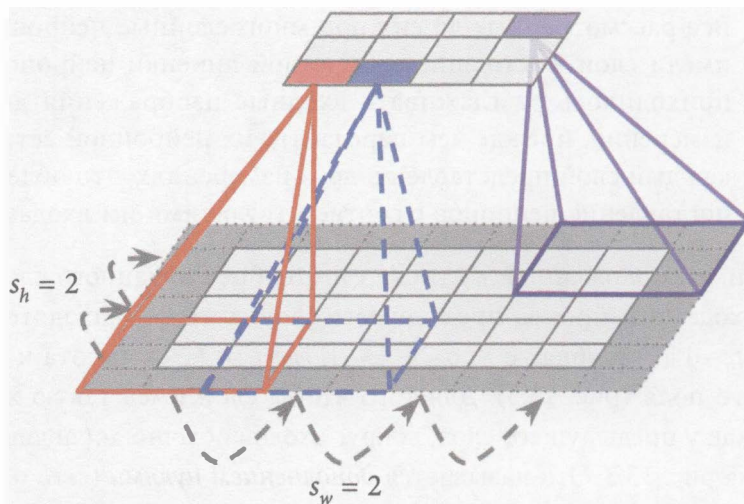


Рис. 13.4. Понижение размерности с применением страйда 2

Фильтры

Веса нейронов могут быть представлены как небольшие изображения с размером рецепторного поля. Например, на рис. 13.5 показаны два возможных набора весов, называемые *фильтрами* (*filter*) или *сверточными ядрами* (*convolution kernel*). Первый фильтр представлен в виде черного квадрата с вертикальной белой линией в середине (это матрица 7×7 , которая заполнена нулями за исключением центрального столбца, заполненного единицами); нейроны, использующие такие веса, будут игнорировать в своем рецепторном поле все кроме центральной вертикальной линии (потому что все входы будут умножаться на 0 за исключением входов, расположенных на центральной вертикальной линии). Второй фильтр имеет вид черного квадрата с горизонтальной белой линией в середине. И снова нейроны, использующие такие веса, будут игнорировать в своем рецепторном поле все кроме центральной горизонтальной линии.

Теперь если все нейроны в слое используют один и тот же фильтр с вертикальной линией (и тот же самый член смещения), и вы передаете сети входное изображение, приведенное на рис. 13.5 (нижнее изображение), то слой выдаст левое верхнее изображение. Обратите внимание, что вертикальные белые линии усилились, в то время как остальное стало размытым. Аналогично правое верхнее изображение представляет собой то, что будет получено, если все нейроны применяют фильтр с горизонтальной линией;

заметно, что горизонтальные белые линии усилились, а остальное стало размытым. Таким образом, слой с нейронами, использующими один и тот же фильтр, выдает *карту признаков* (*feature map*), которая выделяет области изображения, наиболее сходные с фильтром. Во время обучения сеть CNN находит самые пригодные фильтры для своей задачи и учится комбинировать их в более сложные образы (скажем, крест является областью, где активны оба фильтра — с вертикальной линией и с горизонтальной линией).

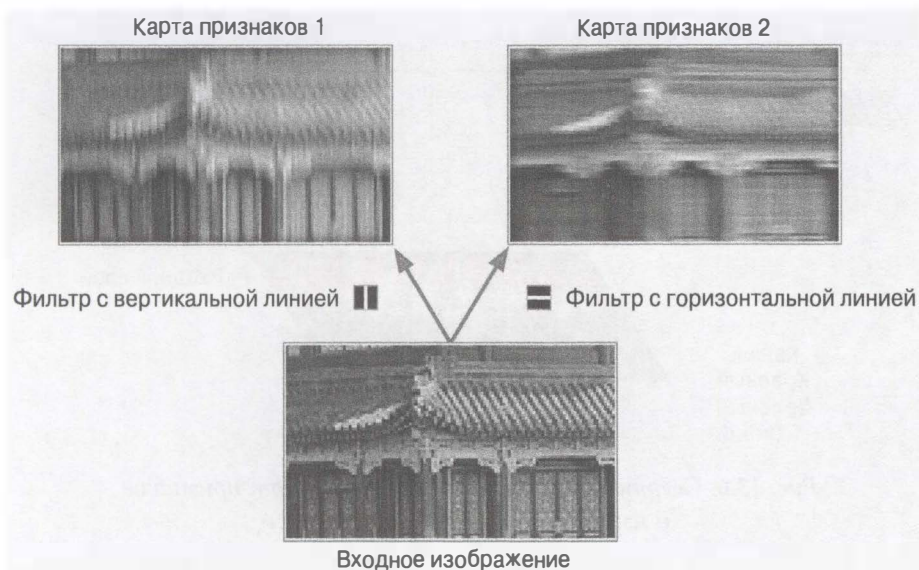


Рис. 13.5. Применение двух разных фильтров для получения карт признаков

Наложение множества карт признаков

До сих пор ради простоты мы представляли каждый сверточный слой как тонкий двумерный слой, но в реальности он состоит из нескольких карт признаков равных размеров, поэтому более точно представляется в трех измерениях (рис. 13.6). Внутри одной карты признаков все нейроны разделяют те же самые параметры (веса и член смещения), но разные карты признаков могут иметь отличающиеся параметры. Рецепторное поле нейрона такое же, как описано ранее, но оно распространяется на все карты признаков предшествующих слоев. Короче говоря, сверточный слой одновременно применяет множество фильтров к своим входам, становясь способным обнаруживать множество признаков повсюду в своих входах.

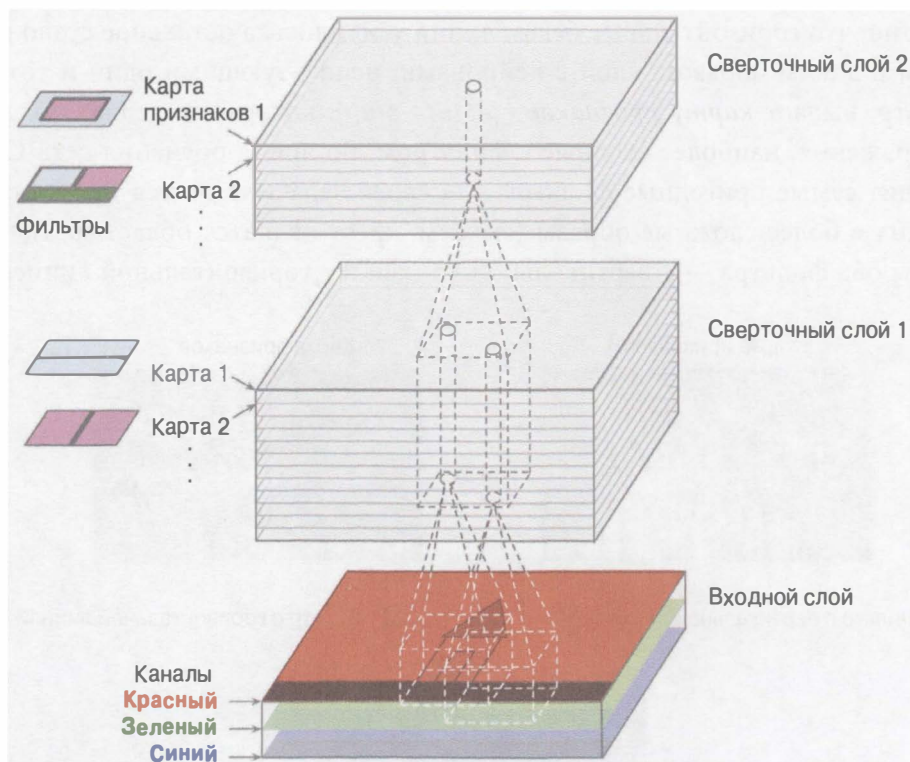


Рис. 13.6. Сверточные слои с множеством карт признаков и изображения с тремя каналами



Тот факт, что все нейроны в карте признаков разделяют те же самые параметры, значительно сокращает количество параметров в модели, но более важно то, что после обучения сети CNN распознаванию образа в одном положении, она способна распознавать его в любом другом положении. Напротив, после того, как обыкновенная сеть DNN обучена распознаванию образа в одном положении, она может распознавать его только в этом конкретном положении.

Кроме того, входные изображения также состоят из множества подслов: по одному на **цветовой канал**. Их обычно три: красный, зеленый и синий (red, green, blue — RGB). Полутонные изображения имеют только один канал, но некоторые изображения могут иметь гораздо больше каналов — например, изображения со спутников, фиксирующие свет дополнительных частот (такой как инфракрасный).

В частности, нейрон, расположенный в строке i и столбце j карты признаков k в заданном сверточном слое l , связывается с выходами нейронов в предыдущем слое $l-1$, которые находятся в строках с $i \times s_h$ по $i \times s_h + f_h - 1$ и столбцах с $j \times s_w$ по $j \times s_w + f_w - 1$ через все карты признаков (в слое $l-1$). Обратите внимание, что все нейроны, расположенные в той же самой строке i и столбце j , но в разных картах признаков, связываются с выходами точно тех же нейронов в предыдущем слое.

В уравнении 13.1 приведенные объяснения подытоживаются в одно большое математическое уравнение: оно показывает, как вычислять выход заданного нейрона в сверточном слое. Из-за обилия разных индексов уравнение выглядит несколько неуклюжим, но все, что оно делает — вычисляет взвешенную сумму всех входов плюс член смещения.

Уравнение 13.1. Вычисление выхода нейрона в сверточном слое

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{с} \quad \begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

- $z_{i,j,k}$ — выход нейрона, расположенного в строке i и столбце j в карте признаков k сверточного слоя (слоя l).
- Как объяснялось ранее, s_h и s_w — вертикальный и горизонтальный страйды, f_h и f_w — высота и ширина рецепторного поля, а $f_{n'}$ — количество карт признаков в предыдущем слое (слое $l-1$).
- $x_{i',j',k'}$ — выход нейрона, расположенного в слое $l-1$, строка i' , столбец j' , карта признаков k' (или канал k' , если предыдущий слой является входным).
- b_k — член смещения для карты признаков k (в слое l). Вы можете думать об этом как о ручке управления, которая регулирует общую яркость карты признаков k .
- $w_{u,v,k',k}$ — вес связи между любым нейроном в карте признаков k слоя l и его входом, расположенным в строке u , столбце v (относительно рецепторного поля нейрона) и карте признаков k' .

Реализация с помощью TensorFlow

В TensorFlow каждое входное изображение обычно представляется как трехмерный тензор в форме [высота, ширина, каналы]. Мини-пакет представляется как четырехмерный тензор в форме [размер мини-пакета, высота, ширина, каналы]. Веса сверточного слоя представляются как четырехмерный тензор в форме $[f_h, f_w, f_n', f_n]$. Члены смещения сверточного слоя представляются просто как одномерный тензор в форме $[f_n]$.

Давайте рассмотрим простой пример. Приведенный ниже код загружает два примера изображений, используя функцию `load_sample_images()` из Scikit-Learn (которая загружает два цветных изображения, одно с китайским храмом и одно с цветком). Затем код создает два фильтра 7×7 (один с вертикальной белой линией в середине, а второй с горизонтальной белой линией в середине) и применяет их к обоим изображениям с использованием сверточного слоя, построенного посредством функции `tf.nn.conv2d()` из TensorFlow (с дополнением нулями и страйдом 2). Наконец, код вычерчивает одну из результирующих карт признаков (подобную правому верхнему изображению на рис. 13.5).

```
import numpy as np
from sklearn.datasets import load_sample_images

# Загрузка примеров изображений
china = load_sample_image("china.jpg")
flower = load_sample_image("flower.jpg")
dataset = np.array([china, flower], dtype=np.float32)
batch_size, height, width, channels = dataset.shape

# Создание двух фильтров
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1          # вертикальная линия
filters[3, :, :, 1] = 1          # горизонтальная линия

# Создание графа с входным X и сверточным слоем, применяющим два фильтра
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
convolution = tf.nn.conv2d(X, filters, strides=[1, 2, 2, 1], padding="SAME")

with tf.Session() as sess:
    output = sess.run(convolution, feed_dict={X: dataset})

plt.imshow(output[0, :, :, 1], cmap="gray") # вычерчивание второй
                                             # карты признаков первого изображения
plt.show()
```

Большая часть кода самоочевидна, но строка `tf.nn.conv2d()` заслуживает некоторых пояснений.

- `X` — входной мини-пакет (четырёхмерный тензор, как объяснялось ранее).
- `filters` — набор фильтров, подлежащих применению (также четырёхмерный тензор, как объяснялось ранее).
- `strides` — четырёхэлементный одномерный массив, где два центральных элемента представляют собой вертикальный и горизонтальный страйды (s_h и s_w). Первый и последний элементы в текущий момент должны быть равны 1. В один прекрасный день они могут использоваться для указания страйда пакета (чтобы пропускать некоторые образцы) и страйда канала (чтобы пропускать некоторые карты признаков или каналы предыдущего слоя).
- `padding` должен быть установлен либо в "VALID", либо в "SAME".
 - Если `padding` установлен в "VALID", то сверточный слой не задействует дополнение нулями и может игнорировать некоторые строки и столбцы внизу и справа входного изображения в зависимости от страйда, как демонстрируется на рис. 13.7 (для простоты здесь показано только горизонтальное измерение, но, само собой разумеется, та же самая логика применима и к вертикальному измерению).

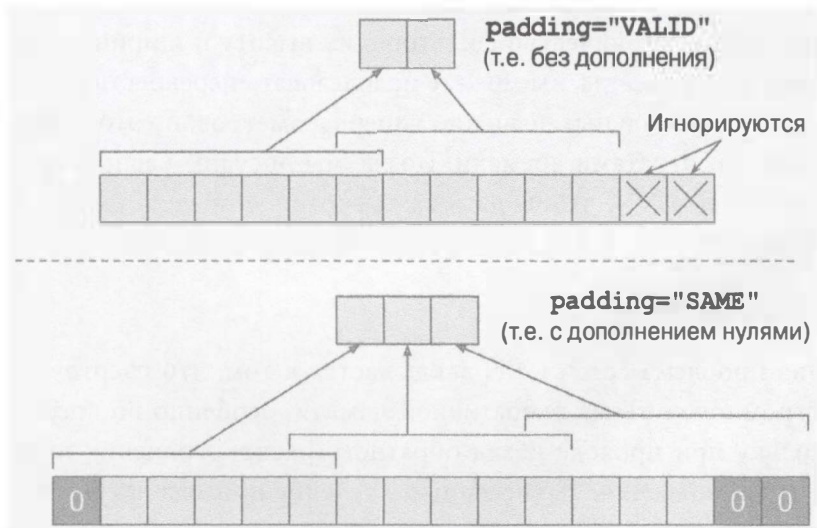


Рис. 13.7. Варианты дополнения — ширина входа: 13, ширина фильдра: 6, страйд: 5

- Если `padding` установлен в `"SAME"`, то сверточный слой при необходимости использует дополнение нулями. В таком случае количество выходных нейронов равно числу входных нейронов, деленному на страйд, с округлением в большую сторону (в данном примере, $\text{ceil}(13 / 5) = 3$). Затем вокруг входов как можно более равномерно добавляются нули.

В этом простом примере мы создаем фильтры вручную, но в реальной сети CNN вы позволите алгоритму обучения выявить наилучшие фильтры автоматически. Библиотека TensorFlow имеет функцию `tf.layers.conv2d()`, которая создает переменную фильтров (по имени `kernel`) и инициализирует ее случайным образом. Она также создает переменную смещения (по имени `bias`) и инициализирует ее нулями. Например, следующий код создает входной заполнитель, за которым следует сверточный слой с двумя картами признаков 7×7 , применяя страйды 2×2 (обратите внимание, что данная функция ожидает только вертикального и горизонтального страйдов) и дополнение `"SAME"`:

```
X = tf.placeholder(shape=(None, height, width, channels),
                    dtype=tf.float32)
conv = tf.layers.conv2d(X, filters=2, kernel_size=7, strides=[2,2],
                        padding="SAME")
```

К сожалению, сверточные слои имеют порядочное число гиперпараметров: вы обязаны выбрать количество фильтров, их высоту и ширину, страйды и тип дополнения. Как всегда, вы можете использовать перекрестную проверку для нахождения правильных значений гиперпараметров, но это сопряжено с очень большими затратами времени. Позже мы обсудим распространенные архитектуры сетей CNN, чтобы дать вам некоторое представление о значениях гиперпараметров, лучше всего работающих на практике.

Требования к памяти

Еще одна проблема сетей CNN заключается в том, что сверточные слои требуют огромного объема оперативной памяти, особенно во время обучения, поскольку при проходе назад обратного распространения нужны все промежуточные значения, вычисленные в течение прохода вперед.

Например, возьмем сверточный слой с фильтрами 5×5 , выдающий 200 карт признаков размером 150×100 , со страйдом 1 и дополнением `"SAME"`.

Если на вход передается RGB-изображение 150×100 (три канала), тогда количество параметров составляет $(5 \times 5 \times 3 + 1) \times 200 = 15\,200$ (+1 соответствует членам смещения), что достаточно немного в сравнении с полносвязным слоем⁷. Тем не менее, каждая из 200 карт признаков содержит 150×100 нейронов, а каждый нейрон нуждается в вычислении взвешенной суммы своих $5 \times 5 \times 3 = 75$ входов: всего 225 миллионов операций умножения с плавающей точкой. Не настолько плохо, как у полносвязного слоя, но все-таки достаточно интенсивно в вычислительном отношении. Кроме того, если карты признаков представлены с применением 32-битовых значений с плавающей точкой, то выход сверточного слоя будет занимать $200 \times 150 \times 100 \times 32 = 96$ миллионов битов (около 11.4 Мбайт) оперативной памяти⁸. И это лишь для одного образца! Если обучающий пакет содержит 100 образцов, тогда данный слой будет использовать свыше 1 Гбайт оперативной памяти!

Во время выведения (т.е. выработки прогноза для нового образца) оперативная память, занятая одним слоем, может быть освобождена, как только вычислен следующий слой, поэтому оперативной памяти необходимо иметь столько, сколько требуется для двух последовательных слоев. Но во время обучения все, что вычисляется в течение прохода вперед, должно быть сохранено для прохода назад, а потому оперативной памяти нужно (по крайней мере) столько, сколько ее требуется для всех слоев.



Если обучение терпит неудачу из-за ошибки, вызванной нехваткой памяти, тогда вы можете попробовать сократить размер мини-пакета. В качестве альтернативы вы можете попытаться понизить размерность с применением страйда или удалить несколько слоев. Либо вы можете использовать 16-битовые значения с плавающей точкой вместо 32-битовых. Или же можете распределить сеть CNN между множеством устройств.

А теперь давайте посмотрим на второй общий строительный блок в сетях CNN: *объединяющий слой*.

⁷ Полносвязный слой с 150×100 нейронами, каждый из которых связан со всеми $150 \times 100 \times 3$ входами, имел бы $150^2 \times 100^2 \times 3 = 675$ миллионов параметров!

⁸ 1 Мбайт = 1 024 Кбайт = $1\,024 \times 1\,024$ байтов = $1\,024 \times 1\,024 \times 8$ битов.

Объединяющий слой

Разобравшись в том, как работают сверточные слои, понять назначение объединяющих слоев (называемых также субдискретизирующими слоями или слоями подвыборки — *примеч. пер.*) довольно легко. Их цель заключается в том, чтобы *проредить* (т.е. сжать) входное изображение для сокращения вычислительной нагрузки, расхода памяти и количества параметров (тем самым ограничивая риск переобучения).

Как и в сверточных слоях, каждый нейрон в объединяющем слое связан с выходами ограниченного числа нейронов из предыдущего слоя, которые расположены внутри небольшого прямоугольного рецепторного поля. Вы должны определить его размер, страйд и тип дополнения, точно как раньше. Однако объединяющий нейрон не имеет весов; он лишь агрегирует входы с применением функции агрегирования, такой как максимум или среднее. На рис. 13.8 показан *слой объединения по максимуму* (*max pooling layer*), который является самым распространенным типом объединяющего слоя. В данном примере мы используем *объединяющее ядро* (*pooling kernel*) 2×2 , страйд 2, без дополнения. Обратите внимание, что на следующий слой попадает только максимальное входное значение в каждом ядре. Остальные входы отбрасываются.

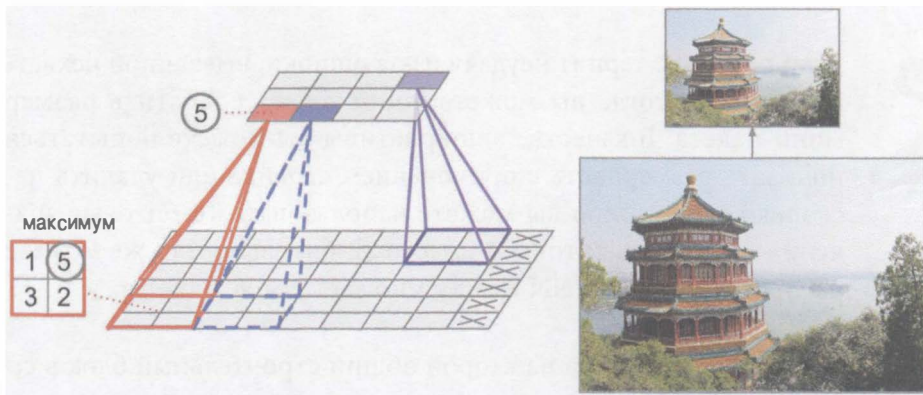


Рис. 13.8. Слой объединения по максимуму
(объединяющее ядро 2×2 , страйд 2, без дополнения)

Очевидно, это крайне деструктивный вид слоя: даже с крошечным ядром 2×2 и страйдом 2 выход будет в два раза меньше в обоих направлениях (так что его площадь окажется в четыре раза меньше) из-за простого отбрасывания 75% входных значений.

Объединяющий слой обычно работает с каждым каналом входного изображения независимо, поэтому выходная глубина является такой же, как входная. В качестве альтернативы вы можете объединять по измерению глубины, как будет показано далее; в таком случае пространственные измерения изображения (высота и ширина) остаются прежними, но количество каналов сокращается.

Реализовать слой объединения по максимуму в TensorFlow довольно легко. Следующий код создает слой объединения по максимуму, используя ядро 2×2 , страйд 2, без дополнения, и затем применяет его ко всем изображениям в наборе данных:

```
[...] # загрузка набора данных с изображениями, как делалось ранее
# Создание графа с входным X и слоем объединения по максимуму
X = tf.placeholder(tf.float32, shape=(None, height, width, channels))
max_pool = tf.nn.max_pool(X, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding="VALID")

with tf.Session() as sess:
    output = sess.run(max_pool, feed_dict={X: dataset})
plt.imshow(output[0].astype(np.uint8)) # вычерчивание выхода для
                                         # первого изображения
plt.show()
```

Аргумент `ksize` содержит форму ядра наряду со всеми четырьмя измерениями входного тензора: [размер пакета, высота, ширина, каналы]. В настоящее время TensorFlow не поддерживает объединение по множеству образцов, так что первый элемент `ksize` обязан быть равен 1. Кроме того, TensorFlow не поддерживает объединение по пространственным измерениям (высота и ширина) и по измерению глубины, поэтому либо `ksize[1]` и `ksize[2]` должны быть равны 1, либо `ksize[3]` должен быть равен 1.

Для создания *слоя объединения по среднему* (*average pooling layer*) просто используйте функцию `avg_pool()` вместо `max_pool()`.

Теперь вы знаете о строительных блоках все, чтобы создать сеть CNN. Давайте выясним, как их компоновать.

Архитектуры сверточных нейронных сетей

Типовая архитектура сети CNN укладывает стопкой несколько сверточных слоев (за каждым из которых следует слой ReLU), объединяющий слой, еще несколько сверточных слоев (плюс слои ReLU), снова объединяющий

слой и т.д. По мере прохождения через сеть изображение становится все меньше и меньше, но обычно также глубже и глубже (т.е. с большим числом карт признаков) благодаря сверточным слоям (рис. 13.9). На верхушку стопки добавляется обыкновенная нейронная сеть прямого распространения, состоящая из нескольких полносвязных слоев (плюс слоев ReLU), и финальный слой выдает прогноз (например, многопеременный слой, который выдает оценочные вероятности классов).



Рис. 13.9. Типовая архитектура сети CNN



Общее заблуждение связано с применением слишком крупных сверточных ядер. Того же самого результата, который дает один сверточный слой с ядром 9×9 , часто можно достичь, укладывая стопкой два слоя с ядрами 3×3 и имея дело с гораздо меньшим объемом вычислений и параметров.

На протяжении многих лет были разработаны варианты этой фундаментальной архитектуры, что привело к удивительным успехам в данной области. Эффективной мерой развития является частота ошибок в состязаниях наподобие задачи ILSVRC ImageNet (<http://image-net.org/>). В рамках данного состязания частота ошибок топ-5 для классификации изображений упала с более чем 26% до менее 3% всего лишь за шесть лет. Частота ошибок топ-5 представляет собой количество испытательных изображений, для которых лучшие 5 прогнозов не включают корректный ответ. Изображения крупные (высотой 256 пикселей), есть 1 000 классов и некоторые из них действительно едва различимы (попробуйте найти отличия между 120 породами собак). Просмотр эволюции победивших записей — хороший способ понять, как работают сети CNN.

Сначала мы рассмотрим классическую архитектуру LeNet-5 (1998 года), а затем три победивших решения задачи ILSVRC: AlexNet (2012 года), GoogLeNet (2014 года) и ResNet (2015 года).

Другие зрительные задачи

Ошеломляющее развитие затронуло также и другие зрительные задачи, такие как обнаружение и установление местонахождения объектов и сегментация изображений. В задаче обнаружения и установления местонахождения объектов нейронная сеть обычно выдает последовательность ограничивающих прямоугольников вокруг разнообразных объектов в изображении. Скажем, в работе Максима Окаба и др., написанной в 2015 году (<https://goo.gl/ZKuDtv>), для каждого класса объектов выдается тепловая карта. В работе Стюарта Рассела и др., написанной тоже в 2015 году (<https://goo.gl/upuHl2>), используется комбинация сети CNN для обнаружения лиц и рекуррентной нейронной сети для выдачи последовательности ограничивающих прямоугольников вокруг них. В задаче сегментации изображений сеть выдает изображение (как правило, того же размера, что и входное), где каждый пиксель указывает класс объекта, к которому принадлежит соответствующий пиксель входного изображения. Например, почитайте работу Эвана Шелхамера и др. (<https://goo.gl/7ReZql>), написанную в 2016 году.

LeNet-5

Архитектура LeNet-5 является, возможно, самой широко известной архитектурой сетей CNN. Как упоминалось ранее, она была создана Яном Лекуном в 1998 году и масштабно применялась для распознавания рукописных цифр (MNIST). Она состоит из слоев, приведенных в табл. 13.1.

Таблица 13.1. Архитектура LeNet-5

Слой	Тип	Карты	Размер	Размер ядра	Страйд	Активация
Out (выходной)	Полносвязный	—	10	—	—	RBF
F6	Полносвязный	—	84	—	—	tanh
C5	Сверточный	120	1 × 1	5 × 5	1	tanh
S4	Объединение по среднему	16	5 × 5	2 × 2	2	tanh
C3	Сверточный	16	10 × 10	5 × 5	1	tanh
S2	Объединение по среднему	6	14 × 14	2 × 2	2	tanh
C1	Сверточный	6	28 × 28	5 × 5	1	tanh
In (входной)	Входной	1	32 × 32	—	—	—

Ниже отмечено несколько добавочных деталей.

- Изображения MNIST имеют 28×28 пикселей, но перед передачей в сеть они дополняются нулями до 32×32 пикселей и нормализуются. В оставшейся части сети никакое дополнение не используется, а потому по мере прохождения через сеть размер продолжает уменьшаться.
- Слои объединения по среднему чуть сложнее, чем обычно: каждый нейрон вычисляет среднее своих входов, умножает результат на обучаемый коэффициент (один на карту) и добавляет обучаемый член смещения (тоже один на карту), после чего применяет функцию активации.
- Большинство нейронов в картах S_3 связываются с нейронами лишь из трех или четырех карт S_2 (вместо всех шести карт S_2). За деталями обращайтесь к таблице 1 в исходной работе.
- Выходной слой несколько специфичен: вместо вычисления скалярного произведения входов и вектора весов каждый нейрон выдает возведенное в квадрат евклидово расстояние между своим вектором входов и своим вектором весов. Каждый выход измеряет, в какой степени изображение принадлежит отдельному классу цифр. Теперь отдается предпочтение функции издержек на основе перекрестной энтропии, т.к. она намного больше штрафует неправильные прогнозы, порождая крупные градиенты и в итоге ускоряя сходжение.

На веб-сайте Яна Лекуна (<http://yann.lecun.com/>) в разделе “LENET” предлагаются замечательные демонстрации классификации цифр LeNet-5.

AlexNet

В 2012 году архитектура сетей CNN под названием *AlexNet*⁹ с большим отрывом победила в решении задачи ImageNet ILSVRC: она достигла 17%-ной частоты ошибок топ-5, тогда как второй результат давал только 26%! Архитектура AlexNet была разработана Алексом Крижевским (отсюда ее название), Ильей Сатскевером и Джефффри Хинтоном. Она довольно похожа на LeNet-5, но гораздо крупнее и глубже, к тому же, в ней сверточные слои впервые укладывались непосредственно друг на друга вместо помещения

⁹ “ImageNet Classification with Deep Convolutional Neural Networks” (“Классификация ImageNet с помощью глубоких сверточных нейронных сетей”), А. Крижевский и др. (2012 год) (<http://goo.gl/mWRBRp>).

объединяющего слоя поверх каждого сверточного слоя. Архитектура AlexNet представлена в табл. 13.2.

Таблица 13.2. Архитектура AlexNet

Слой	Тип	Карты	Размер	Размер ядра	Страйд	Дополнение	Активация
Out (выходной)	Полносвязный	—	1 000	—	—	—	Многопеременная
F9	Полносвязный	—	4 096	—	—	—	ReLU
F8	Полносвязный	—	4 096	—	—	—	ReLU
C7	Сверточный	256	13 × 13	3 × 3	1	SAME	ReLU
C6	Сверточный	384	13 × 13	3 × 3	1	SAME	ReLU
C5	Сверточный	384	13 × 13	3 × 3	1	SAME	ReLU
S4	Объединение по максимуму	256	13 × 13	3 × 3	2	VALID	—
C3	Сверточный	256	27 × 27	5 × 5	1	SAME	ReLU
S2	Объединение по максимуму	96	27 × 27	3 × 3	2	VALID	—
C1	Сверточный	96	55 × 55	11 × 11	4	SAME	ReLU
In (входной)	Входной	3 (RGB)	224×224	—	—	—	—

Чтобы сократить риск переобучения, авторы использовали два приема регуляризации, которые обсуждались в предшествующих главах. Во-первых, во время обучения они применяли отключение (с долей отключения 50%) к выходам слоев F8 и F9. Во-вторых, они реализовали дополнение данных, случайно сдвигая обучающие изображения на различные смещения, переворачивая их по горизонтали и изменяя условия освещения.

Немедленно после шага ReLU слоев C1 и C3 в AlexNet также используется шаг состязательной нормализации (competitive normalization), называемый *локальной нормализацией ответа* (*Local Response Normalization* — *LRN*). Эта форма нормализации заставляет нейроны, которые наиболее сильно активируются, подавлять нейроны в том же самом местоположении, но в соседствующих картах признаков (такая состязательная активация наблюдалась в биологических нейронах). Данный прием стимулирует разные карты признаков специализироваться за счет их отделения и принуждения к исследованию более широкого диапазона признаков, что в итоге улучшает обобщение. В уравнении 13.2 видно, как применять LRN.

Уравнение 13.2. Локальная нормализация ответа

$$b_i = a_i \left(k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{с} \quad \begin{cases} j_{\text{high}} = \min \left(i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left(0, i - \frac{r}{2} \right) \end{cases}$$

- b_i — нормализованный выход нейрона, расположенного внутри карты признаков i , в какой-то строке u и столбце v (обратите внимание, что в этом уравнении мы учитываем только нейроны, находящиеся в данной строке и столбце, а потому u и v не показаны).
- a_i — активация этого нейрона после шага ReLU, но перед нормализацией.
- k , α , β и r — гиперпараметры. k называется *смещением* (*bias*), а r — *радиусом глубины* (*depth radius*).
- f_n — количество карт признаков.

Например, если $r = 2$ и нейрон имеет сильную активацию, тогда он подавит активацию нейронов, расположенных в картах признаков непосредственно выше и ниже своей собственной.

В AlexNet гиперпараметры устанавливаются следующим образом: $r = 2$, $\alpha = 0.00002$, $\beta = 0.75$ и $k = 1$. Такой шаг может быть реализован с помощью операции `tf.nn.local_response_normalization()` из TensorFlow.

Вариант AlexNet под названием *ZF Net* был разработан Мэттью Зайлером и Робом Фергюсом; эта сеть победила в решении задачи ILSVRC в 2013 году. По существу она представляет собой AlexNet с несколькими подстроеными гиперпараметрами (количество карт признаков, размер ядра, страйд и т.д.).

GoogLeNet

Архитектура GoogLeNet была разработана Кристианом Сегеди и др. из Google Research¹⁰ и в 2014 году победила в решении задачи ILSVRC, уронив частоту ошибок топ-5 ниже 7%. Столь высокая производительность по большей части проистекала из того факта, что сеть была намного глубже предшествующих сетей CNN (рис. 13.10). Это стало возможным благодаря подсетям, называемым *модулями начала* (*inception module*)¹¹, которые позволяют

¹⁰ “Going Deeper with Convolutions” (“Углубление с помощью сверток”), К. Сегеди и др. (2015 год) (<http://goo.gl/tCFzVs>).

¹¹ В фильме “Inception” (“Начало”) 2010 года действующие лица погружались все глубже и глубже в многочисленные слои сновидений, отсюда и такое название у модулей.

GoogLeNet использовать параметры гораздо эффективнее, чем предыдущие архитектуры: GoogLeNet фактически имеет в 10 раз меньше параметров, чем AlexNet (приблизительно 6 миллионов вместо 60 миллионов).

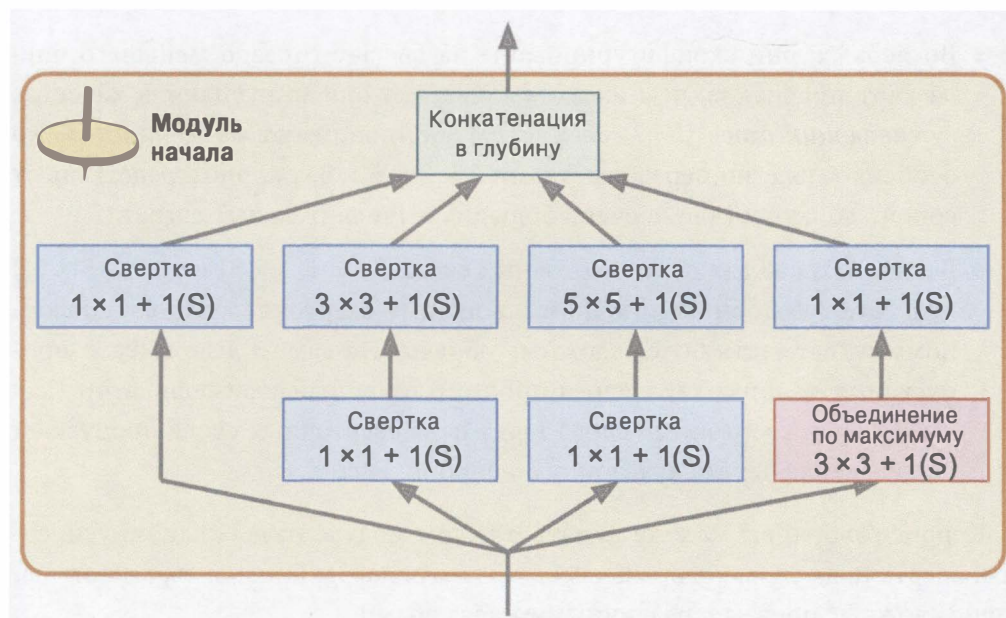


Рис. 13.10. Модуль начала

На рис. 13.10 показана архитектура модуля начала. Запись “ $3 \times 3 + 1(S)$ ” означает, что слой применяет ядро 3×3 , страйд 1 и дополнение SAME. Входной сигнал сначала копируется и передается четырем разным слоям. Все сверточные слои используют функцию активации ReLU. Обратите внимание, что второй набор сверточных слоев применяет ядра отличающихся размеров (1×1 , 3×3 и 5×5), позволяя им захватывать образы с разными масштабами.

Вдобавок отметьте, что каждый одиночный слой использует страйд 1 и дополнение SAME (даже слой объединения по максимуму), а потому все их выходы имеют такую же высоту и ширину, как у их входов. Это делает возможным конкатенацию всех выходов по измерению глубины в финальном *слое конкатенации в глубину* (*depth concat layer*), т.е. укладывание друг на друга карт признаков из всех четырех верхних сверточных слоев. В TensorFlow такой слой конкатенации может быть реализован посредством операции `tf.concat()` с `axis=3` (ось 3 — глубина).

Вас может интересовать, почему модули начала имеют сверточные слои с ядрами 1×1 . Действительно ли эти слои не могут захватывать любые признаки, поскольку видят только один пиксель за раз? В сущности, такие слои служат двум целям.

- Во-первых, они сконфигурированы на выдачу гораздо меньшего числа карт признаков, чем их входы, так что они выступают в качестве *суживающих слоев* (*bottleneck layer*), т.е. понижают размерность. Это особенно полезно перед свертками 3×3 и 5×5 , т.к. они представляют собой слои, требующие очень больших вычислительных затрат.
- Во-вторых, каждая пара сверточных слоев ($[1 \times 1, 3 \times 3]$ и $[1 \times 1, 5 \times 5]$) действуют подобно одиночному, мощному сверточному слою, способному захватывать более сложные образы. На самом деле вместо пропуска изображения через простой линейный классификатор (как делает один сверточный слой) такая пара сверточных слоев пропускает изображение через двухслойную нейронную сеть.

Короче говоря, вы можете думать о целом модуле начала как об усиленном сверточном слое, который умеет выдавать карты признаков, захватывающие сложные образы с различными масштабами.



Количество сверточных ядер для каждого сверточного слоя является гиперпараметром. К сожалению, это означает, что с каждым добавленным модулем начала у вас появляется шесть дополнительных гиперпараметров, которые придется подстраивать.

Теперь давайте рассмотрим архитектуру сверточных нейронных сетей GoogLeNet (рис. 13.11). Она настолько глубока, что мы решили представить ее в трех колонках, но GoogLeNet фактически выглядит как одна высокая стопка, включающая девять модулей начала (прямоугольники с волчками), каждый из которых в действительности содержит три слоя. Перед размером ядра указано количество карт признаков, выдаваемых каждым сверточным слоем и каждым объединяющим слоем. Шесть чисел в модулях начала представляют количество карт признаков, выдаваемых каждым сверточным слоем в модуле (в том же порядке, как на рис. 13.10). Обратите внимание, что все сверточные слои применяют функцию активации ReLU.

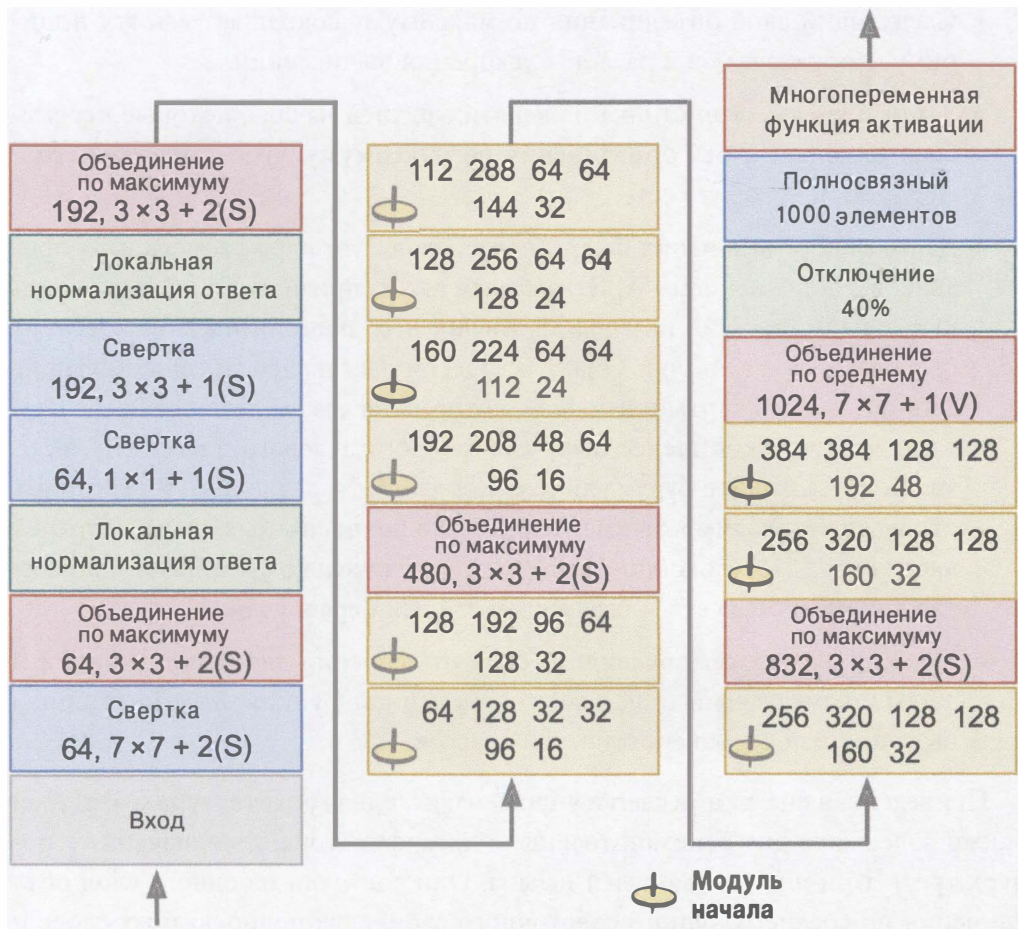


Рис. 13.11. Архитектура GoogleNet

Пройдемся по представленной сети.

- Первые два слоя делят высоту и ширину изображения на 4 (так что его область делится на 16) для снижения вычислительной нагрузки.
- Затем слой локальной нормализации ответа удостоверяется, что предшествующие слои обучились широкому разнообразию признаков (как обсуждалось ранее).
- Далее идут два сверточных слоя, первый из которых действует подобно *суживающему слою*. Как объяснялось выше, вы можете думать об этой паре как об одном более интеллектуальном сверточном слое.
- И снова слой локальной нормализации ответа удостоверяется в том, что предшествующие слои захватили широкое разнообразие образов.

- Следующий слой объединения по максимуму сокращает высоту и ширину изображения в 2 раза для ускорения вычислений.
- Затем идет высокая стопка из девяти модулей начала, которые перемежаются парой слоев объединения по максимуму, чтобы понизить размерность и ускорить сеть.
- Далее слой объединения по среднему использует ядро размера карт признаков с дополнением VALID, выдавая карты признаков 1×1 : такая удивительная стратегия называется *глобальным объединением по среднему* (*global average pooling*). Она фактически заставляет предшествующие слои выпускать карты признаков, которые на самом деле являются картами доверия (confidence map) для каждого целевого класса (т.к. признаки других видов будут уничтожены шагом усреднения). В результате отпадает необходимость иметь несколько полносвязных слоев в верхней части сети CNN (подобно AlexNet), что значительно уменьшает количество параметров в сети и ограничивает риск переобучения.
- Последние слои самоочевидны: слой отключения для регуляризации и затем полносвязный слой с многопеременной функцией активации для выдачи оценочных вероятностей классов.

Приведенная диаграмма слегка упрощена: исходная архитектура GoogLeNet также содержала два вспомогательных классификатора, подключенные поверх третьего и шестого модулей начала. Они состояли из одного слоя объединения по среднему, одного сверточного слоя, двух полносвязных слоев и слоя многопеременной активации. Во время обучения их потеря (уменьшенная на 70%) добавлялась к общей потере. Целью была борьба с проблемой исчезновения градиентов и регуляризация сети. Тем не менее, они показали, что эффект от них оказался относительно малым.

ResNet

Последнее по счету, но не по значению: победителем в решении задачи ILSVRC в 2015 году стала *остаточная сеть* (*Residual Network*, или *ResNet*), разработанная Кеймингом Хе и др.¹², которая давала поразительную частоту ошибок топ-5 ниже 3.6%, применяя крайне глубокую сеть CNN из 152 слоев.

¹² “Deep Residual Learning for Image Recognition” (“Глубокое остаточное обучение для распознавания изображений”), К. Хе (2015 год) (<http://goo.gl/4puHU5>).

Ключом к возможности обучения настолько глубокой сети является использование *обходящих связей* (*skip connection*), также называемых *сокращенными связями* (*shortcut connection*): сигнал, передаваемый в слой, также добавляется к выходу слоя, расположенного чуть выше в стопке. Давайте посмотрим, чем это полезно.

При обучении нейронной сети преследуется задача заставить ее моделировать целевую функцию $h(x)$. Если добавить вход x к выходу сети (т.е. добавить обходящую связь), тогда сеть будет принудительно моделировать $f(x) = h(x) - x$, а не $h(x)$. Процесс называется *остаточным обучением* (*residual learning*) и демонстрируется на рис. 13.12.

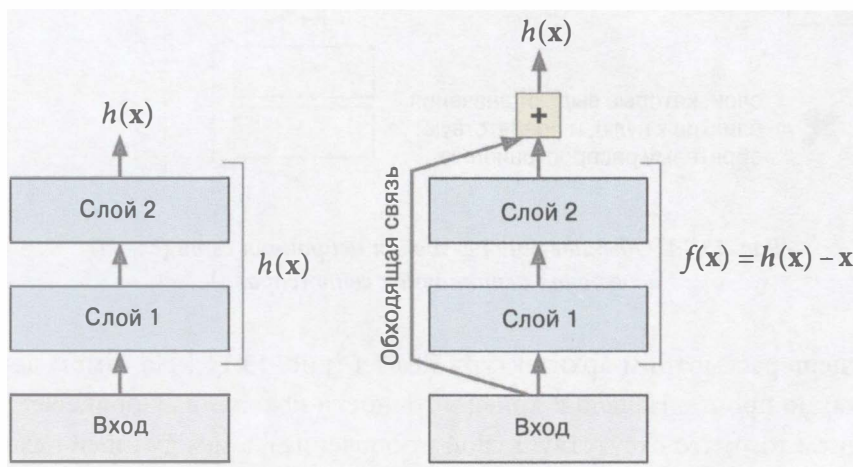


Рис. 13.12. Остаточное обучение

После инициализации обыкновенной нейронной сети ее веса близки к нулю, поэтому сеть просто выдает значения, близкие к нулю. Если добавить обходящую связь, то результирующая сеть выдаст копию своих входов; другими словами, первоначально она моделирует функцию тождественности. Когда целевая функция довольно близка к функции тождественности (что случается часто), это значительно ускорит обучение.

Кроме того, если добавить много обходящих связей, то сеть может делать успехи, даже когда некоторые слои еще не начали обучаться (рис. 13.13). Благодаря обходящим связям сигнал может легко отыскать свой путь через всю сеть. Глубокую остаточную сеть можно рассматривать как стопку *остаточных элементов* (*residual unit*), где каждый остаточный элемент представляет собой небольшую нейронную сеть с обходящей связью.

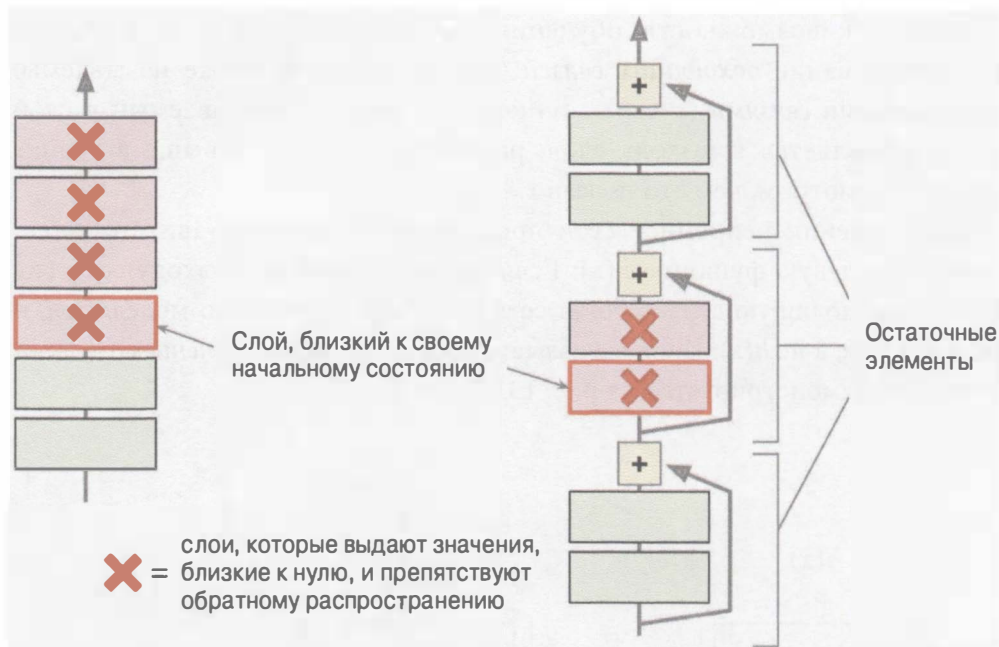


Рис. 13.13. Обыкновенная глубокая нейронная сеть (слева) и глубокая остаточная сеть (справа)

А теперь рассмотрим архитектуру ResNet (рис. 13.14). На самом деле она неожиданно проста. Начало и конец в точности похожи на GoogLeNet (за исключением того, что отсутствует слой отключения), а между ними находится очень глубокая стопка простых остаточных элементов. Каждый остаточный элемент состоит из двух сверточных слоев с пакетной нормализацией (BN) и активацией ReLU, применяющие ядра 3×3 и предохраняющие пространственные измерения (страйд 1, дополнение SAME).

Обратите внимание, что каждые несколько остаточных элементов удваивают количество карт признаков, одновременно деля пополам их высоты и ширины (с использованием сверточного слоя со страйдом 2). Когда такое происходит, входы не могут добавляться напрямую к выходам остаточного элемента, поскольку они не имеют ту же самую форму (например, эта проблема влияет на обходящую связь, представленную на рис. 13.14 пунктирной линией со стрелкой). Чтобы решить проблему, входы пропускаются через сверточный слой 1×1 со страйдом 2 и правильным количеством выходных карт признаков (рис. 13.15).

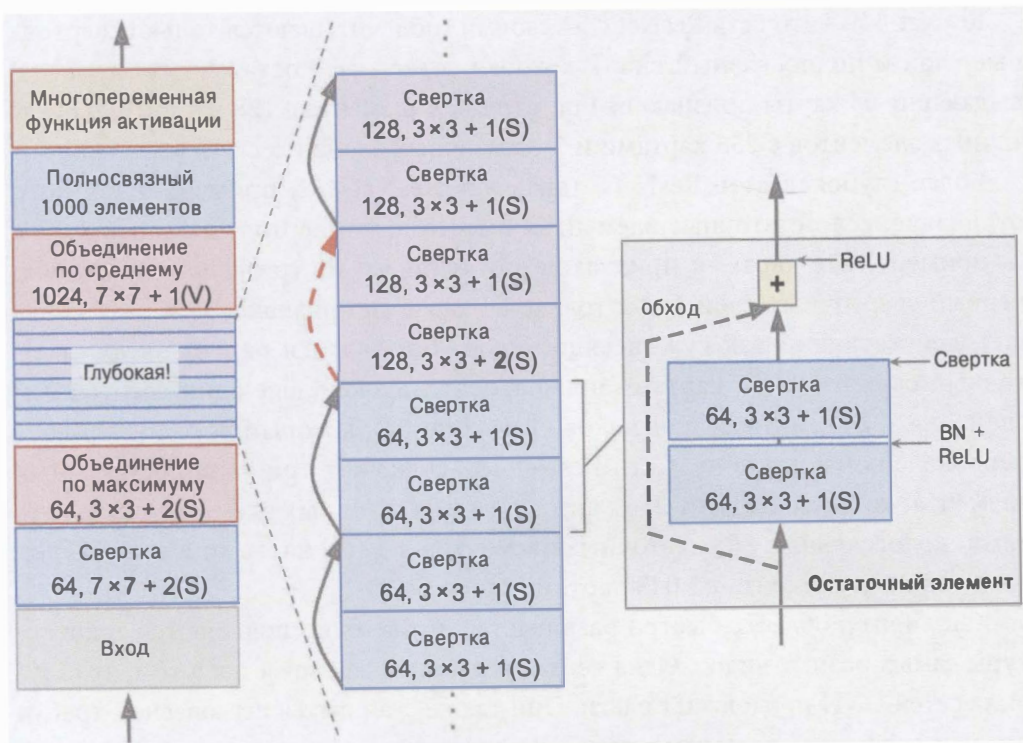


Рис. 13.14. Архитектура ResNet

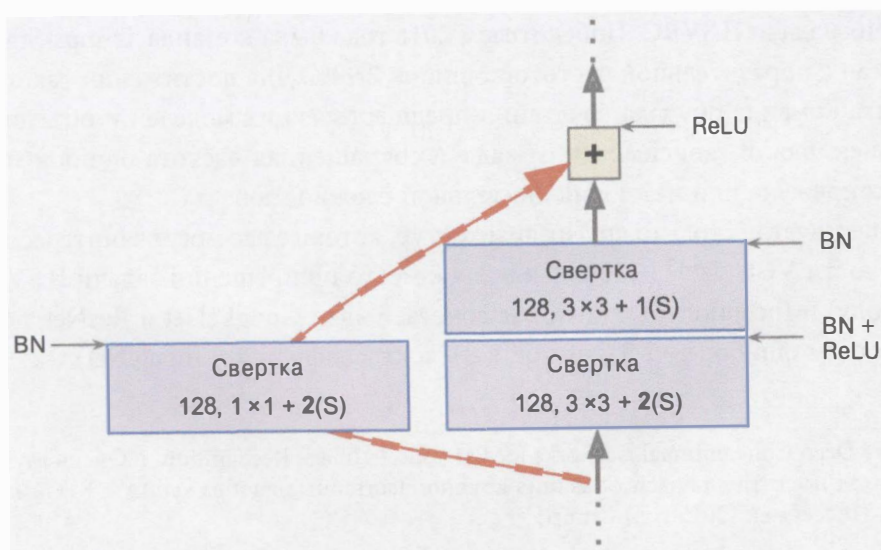


Рис. 13.15. Обходящая связь при изменении размера и глубины карты признаков

ResNet-34 — это сеть ResNet с 34 слоями (подсчитываются только сверточные слои и полносвязный слой), которая содержит 3 остаточных элемента, выдающие 64 карты признаков, 4 остаточных элемента с 128 картами, 6 остаточных элементов с 256 картами и 3 остаточных элемента с 512 картами.

Более глубокая сеть ResNets, такая как ResNet-152, применяет немного отличающиеся остаточные элементы. Вместо двух сверточных слоев 3×3 с (к примеру) 256 картами признаков они используют три сверточных слоя: первый сверточный слой 1×1 с только 64 картами признаков (в 4 раза меньше), действующий как суживающий слой (обсуждался ранее), затем сверточный слой 3×3 с 64 картами признаков и, наконец, еще один сверточный слой 1×1 с 256 картами признаков (4 раза по 64), который восстанавливает первоначальную глубину. Сеть ResNet-152 содержит три таких остаточных элемента, которые выдают 256 карт, затем 8 остаточных элементов с 512 картами, колоссальные 36 остаточных элементов с 1 024 картами и напоследок 3 остаточных элемента с 2 048 картами.

Как видите, область быстро развивается, и ежегодно появляются архитектуры самых разных видов. Одна четкая тенденция заключается в том, что глубина сетей CNN продолжает расти. Они также становятся легче, требуя все меньше и меньше параметров. В настоящее время архитектура ResNet является самой мощной и возможно наиболее простой, поэтому сейчас вероятно имеет смысл применять именно ее, но продолжайте отслеживать ежегодные решения задачи ILSVRC. Победителем 2016 года была команда Trimps-Soushen из Китая с поразительной частотой ошибок 2.99%. Для достижения такого результата команда обучала сочетания предшествующих моделей и объединяла их в ансамбль. В зависимости от задачи сокращенная частота ошибок может быть сопряжена или нет с дополнительной сложностью.

Существует несколько других архитектур, которые вас могут заинтересовать, в частности VGGNet¹³ (занявшая второе место при решении задачи ILSVRC в 2014 году) и Inception-v4¹⁴ (которая сочетает идеи GoogLeNet и ResNet, достигая частоты ошибок топ-5 близкой к 3% в классификации ImageNet).

¹³ “Very Deep Convolutional Networks for Large-Scale Image Recognition” (“Очень глубокие сверточные сети для распознавания крупномасштабных изображений”), К. Симоньян и Э. Циссерман (2015 год) (<http://goo.gl/QcMjXQ>).

¹⁴ “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning” (“Inception-v4, Inception-ResNet и влияние остаточных связей на обучение”), К. Сегеди и др. (2016 год) (<http://goo.gl/Ak2vBp>).



В реализации только что обсужденных архитектур сетей CNN нет ничего особенного. Ранее было показано, как создавать отдельные строительные блоки, и потому вам останется лишь скомпоновать их для получения желаемой архитектуры. Построению полной сети CNN посвящены упражнения в конце главы, а работающий код доступен в тетрадах Jupyter.

Операции свертки TensorFlow

Библиотека TensorFlow предлагает также и другие виды сверточных слоев.

- `tf.layers.conv1d()` создает сверточный слой для одномерных входов. Это полезно, скажем, при обработке естественного языка, когда фраза может быть представлена как одномерный массив слов, а рецепторное поле покрывает несколько соседствующих слов.
- `tf.layers.conv3d()` создает сверточный слой для трехмерных входов, таких как трехмерное изображение в позитронно-эмиссионной томографии.
- `tf.nn.atrous_conv2d()` создает *сверточный слой “à trous”* (“à trous” (фр.) — “с отверстиями”). Это эквивалентно использованию обыкновенного сверточного слоя с фильтром, расширенным за счет вставки строк и столбцов нулей (т.е. отверстий). Например, фильтр 1×3 , равный $\begin{bmatrix} 1, 2, 3 \end{bmatrix}$, может быть расширен с *коэффициентом расширения* (*dilation rate*) 4, в результате давая *расширенный фильтр* $\begin{bmatrix} 1, 0, 0, 0, 2, 0, 0, 0, 3 \end{bmatrix}$. Он позволяет сверточному слою иметь большее рецепторное поле без вычислительных затрат и дополнительных параметров.
- `tf.layers.conv2d_transpose()` создает *транспонированный сверточный слой*, иногда называемый *слоем обращения свертки* (*deconvolutional layer*)¹⁵, который *повышает дискретизацию* (*upsample*) изображения. Он делает это, вставляя нули между входами, так что можете думать о нем как об обыкновенном сверточном слое с дробным страйдом. Повышение дискретизации полезно, к примеру, при сегментации изображений: по мере продвижения через типовую сеть CNN карты признаков становятся меньше и меньше, так что когда нужно выдать изображение такого же размера, как на входе, необходим слой повышения дискретизации.

¹⁵ Такое название несколько сбивает с толку, потому что этот слой вовсе *не* выполняет обращение свертки, представляющее собой четко определенную математическую операцию (противоположность свертки).

- `tf.nn.depthwise_conv2d()` создает *слой свертки по глубине* (*depthwise convolutional layer*), который независимо применяет каждый фильтр к каждому индивидуальному входному каналу. Таким образом, если есть фильтры f_n и входные каналы $f_{n'}$, тогда он будет выдавать $f_n \times f_{n'}$ карт признаков.
- `tf.layers.separable_conv2d()` создает *сепарабельный сверточный слой* (*separable convolutional layer*), который сначала действует подобно слою свертки по глубине, а затем применяет к результирующим картам признаков сверточный слой 1×1 . В итоге появляется возможность применения фильтров к произвольным наборам входных каналов.

Упражнения

1. Каковы преимущества сети CNN в сравнении с полносвязной сетью DNN для классификации изображений?
2. Рассмотрим сеть CNN, состоящую из трех сверточных слоев, каждый с ядрами 3×3 , страйдом 2 и дополнением SAME. Самый нижний слой выдает 100 карт признаков, средний слой — 200 карт признаков, а верхний слой — 400 карт признаков. На вход поступают изображения RGB размером 200×300 пикселей. Сколько всего будет параметров в сети CNN? Если используются 32-битовые значения с плавающей точкой, то какой минимальный объем оперативной памяти потребуется этой сети при выработке прогноза для одиночного образца? Что можно сказать насчет обучения на мини-пакете из 50 изображений?
3. Если во время обучения сети CNN в вашем графическом процессоре случается нехватка памяти, тогда какие пять действий вы могли бы предпринять, чтобы решить проблему?
4. Почему может понадобиться добавление слоя объединения по максимуму, а не сверточного слоя с тем же самым страйдом?
5. Когда может потребоваться добавление слоя локальной нормализации ответа?

6. Можете ли вы назвать главные новшества AlexNet в сравнении с LeNet-5? Каковы основные нововведения GoogLeNet и ResNet?
7. Постройте собственную сеть CNN и попробуйте достичь наивысшей возможной правильности на наборе MNIST.
8. Классификация крупных изображений с применением Inception v3.
 - а) Загрузите изображения различных животных. Загружайте их в Python, используя функцию `matplotlib.image.imread()` или `scipy.misc.imread()`. Измените размер и/или обрежьте изображения до 299×299 пикселей и удостоверьтесь в том, что они имеют только три канала (RGB) без канала прозрачности. Изображения, на которых обучалась модель Inception, были предварительно обработаны так, что их значения находятся в диапазоне от -1.0 до 1.0 , поэтому вы обязаны обеспечить то же самое для своих изображений.
 - б) Загрузите самую последнюю заранее обученную модель Inception v3: контрольная точка доступна по ссылке <https://goo.gl/25uDF7>. Список имен классов находится по ссылке <https://goo.gl/brXRtZ>, но вы должны вставить в начало “фондовый” класс.
 - в) Создайте модель Inception v3, вызвав функцию `inception_v3()`, как показано ниже. Это должно делаться внутри пространства аргумента, созданного функцией `inception_v3_arg_scope()`. Вдобавок нужно установить `is_training=False` и `num_classes=1001`:

```
from tensorflow.contrib.slim.nets import inception
import tensorflow.contrib.slim as slim

X = tf.placeholder(tf.float32, shape=[None, 299, 299, 3],
                  name="X")

with slim.arg_scope([inception.inception_v3_arg_scope()]:
    logits, end_points = inception.inception_v3(
        X, num_classes=1001, is_training=False)
predictions = end_points["Predictions"]
saver = tf.train.Saver()
```
 - г) Откройте сеанс и с помощью `Saver` восстановите загруженную ранее контрольную точку заранее обученной модели.
 - д) Запустите модель для классификации подготовленных изображений. Для каждого изображения отобразите лучшие пять прогнозов и оценочную вероятность. Насколько правильной является модель?

9. Обучение передач знаний для классификации крупных изображений.
- а) Создайте обучающий набор, содержащий минимум 100 изображений на класс. Например, вы могли бы классифицировать собственные фотографии на основе места съемки (пляж, горы, город и т.д.) или в качестве альтернативы применить существующий набор данных, такой как набор данных с фотографиями цветков (<https://goo.gl/EgJVXZ>) или набор данных с фотографиями мест из сайта Массачусетского технологического института (<http://places.csail.mit.edu/>; требует регистрации, и он гигантский).
 - б) Реализуйте шаг предварительной обработки, который будет изменять размер и обрезать изображение до 299×299 пикселей, с долей случайности для дополнения данных.
 - в) Используя заранее обученную модель Inception v3 из предыдущего упражнения, заморозьте все слои вплоть до суживающего слоя (т.е. последнего слоя перед выходным слоем) и замените выходной слой подходящим количеством выходов для новой задачи классификации (скажем, набор данных с фотографиями цветков содержит пять взаимно исключающих классов, поэтому выходной слой должен иметь пять нейронов и применять многопеременную функцию активации).
 - г) Расщепите набор данных на обучающий набор и испытательный набор. Обучите модель на обучающем наборе и оцените ее на испытательном наборе.
10. Просмотрите руководство по DeepDream (<https://goo.gl/4b2s6g>) в TensorFlow. Это интересный метод ознакомления с разнообразными способами визуализации образов, выученных сетью CNN, и генерации графических материалов с использованием глубокого обучения.

Решения приведенных упражнений доступны в приложении А.

Рекуррентные нейронные сети

Бэттер отбивает мяч. Вы немедленно начинаете бежать, предугадывая траекторию мяча. Вы следите за ним, подстраиваете свое движение и, наконец, ловите его (под гром аплодисментов). Прогнозирование будущего — вот то, что вы делаете постоянно, заканчиваете ли фразу друга или предвкушаете запах кофе на завтрак. В настоящей главе мы обсудим *рекуррентные нейронные сети* (*Recurrent Neural Network* — *RNN*), класс сетей, которые способны прогнозировать будущее (разумеется, до определенного момента). Они могут анализировать данные *временных рядов* (*time series*), такие как курсы акций, и сообщать вам, когда покупать или продавать. В системах автопилотов они способны предугадывать траектории автомобилей и препятствовать авариям. В более общем плане сети RNN могут работать с *последовательностями* (*sequence*) произвольной длины, а не с входными данными фиксированного размера, как в случае всех сетей, рассмотренных до сих пор. Например, они в состоянии принимать на входе предложения, документы или аудио-образцы, что делает их чрезвычайно полезными для систем обработки естественного языка (NLP) вроде автоматического перевода, преобразования речи в текст или *смыслового анализа* (скажем, чтения рецензий на фильм и извлечения впечатлений рецензента о фильме).

Кроме того, способность сетей RNN предвидеть также делает их удивительно креативными. Вы можете предложить им спрогнозировать, какая нота вероятнее всего окажется в мелодии следующей, после чего случайным образом выбрать одну из таких нот и воспроизвести ее. Затем запросите у сети следующую наиболее вероятную ноту, воспроизведите ее и повторяйте процесс снова и снова. Прежде чем вы об этом узнаете, ваша сеть сочинит мелодию, подобную той, что доступна по ссылке <http://goo.gl/IxILlV>, которая произведена проектом Google Magenta (<https://magenta.tensorflow.org/>). Аналогично сети RNN могут генерировать предложения (<http://goo.gl/onkPNd>), подписывать изображения

(<http://goo.gl/Nwx7Kh>) и делать многое другое. Результатам пока еще далеко до Шекспира или Моцарта, но кто знает, что они будут производить спустя несколько лет?

В главе мы рассмотрим фундаментальные концепции, лежащие в основе сетей RNN, главную проблему, с которой они сталкиваются (а именно — исчезновение/взрывной рост градиентов, обсуждаемые в главе 11), и решения, широко используемые для борьбы с ней: ячейки LSTM и GRU. Как всегда, попутно будет показано, каким образом реализовывать сети RNN с применением TensorFlow. Наконец, мы взглянем на архитектуру системы машинного перевода.

Рекуррентные нейроны

До сих пор мы видели по большей части нейронные сети прямого распространения, где активации протекали только в одном направлении, от входного слоя до выходного слоя (за исключением нескольких сетей в приложении Д). Рекуррентная нейронная сеть выглядит очень похожей на нейронную сеть прямого распространения, но также имеет связи, указывающие в обратном направлении. Давайте посмотрим на простейшую сеть RNN, которая состоит из одного нейрона, получающего входы, производящего выход и передающего этот выход обратно самому себе (рис. 14.1 (слева)). На каждом *временном шаге* (*time step*) t (также называется *фреймом* (*frame*)) такой *рекуррентный нейрон* (*recurrent neuron*) получает входы $x_{(t)}$, а также собственный выход из предыдущего временного шага, $y_{(t-1)}$. Мы можем представить эту крошечную сеть по оси времени, как показано на рис. 14.1 (справа). Процесс называется *развертыванием сети во времени*.

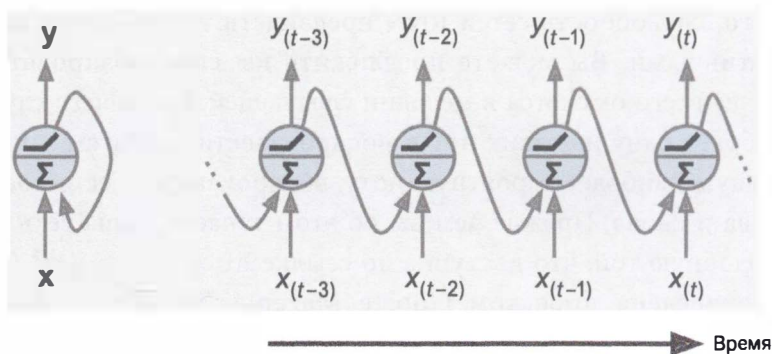


Рис. 14.1. Рекуррентный нейрон (слева), развернутый во времени (справа)

Можно легко создать слой рекуррентных нейронов. На каждом временном шаге t каждый нейрон получает входной вектор $\mathbf{x}_{(t)}$ и выходной вектор из предыдущего временного шага $\mathbf{y}_{(t-1)}$, что продемонстрировано на рис. 14.2. Как видите, теперь и входы, и выходы являются векторами (когда существовал только один нейрон, выход был скаляром).

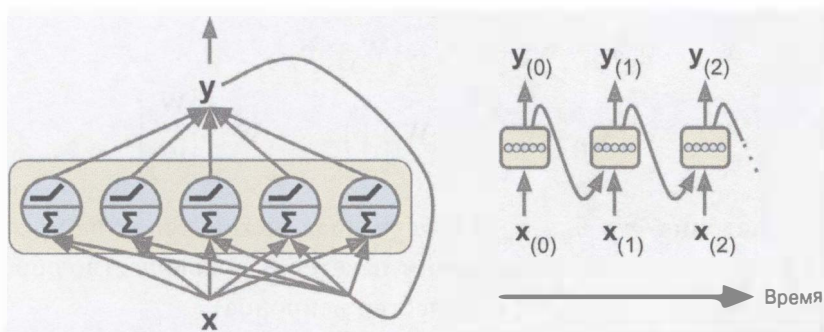


Рис. 14.2. Слой рекуррентных нейронов (слева), развернутый во времени (справа)

Каждый рекуррентный нейрон имеет два набора весов: один для входов, $\mathbf{x}_{(t)}$, и один для выходов предыдущего временного шага, $\mathbf{y}_{(t-1)}$. Назовем эти весовые векторы \mathbf{w}_x и \mathbf{w}_y . Если мы примем во внимание целый рекуррентный слой, а не лишь один рекуррентный нейрон, тогда можем поместить все весовые векторы в две весовые матрицы, \mathbf{W}_x и \mathbf{W}_y . Затем выходной вектор целого рекуррентного слоя можно вычислить вполне ожидаемым образом, как показано в уравнении 14.1 (здесь \mathbf{b} — вектор смещений, а $\phi(\cdot)$ — функция активации, скажем, ReLU^1).

Уравнение 14.1. Выход рекуррентного слоя для одиночного образца

$$\mathbf{y}_{(t)} = \phi(\mathbf{W}_x^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_y^T \cdot \mathbf{y}_{(t-1)} + \mathbf{b})$$

¹ Обратите внимание, что многие исследователи предпочитают использовать в сетях RNN функцию активации в виде гиперболического тангенса (\tanh), а не функцию активации ReLU. В качестве примера ознакомьтесь с работой Ву Фама и др. “Dropout Improves Recurrent Neural Networks for Handwriting Recognition” (“Отключение улучшает рекуррентные нейронные сети для распознавания почерка”) (<https://goo.gl/2WSnaj>). Однако сети RNN на основе ReLU также возможны, как поясняют Кюок В. Ле и др. в своей работе “A Simple Way to Initialize Recurrent Networks of Rectified Linear Units” (“Простой способ инициализации рекуррентных сетей выпрямленных линейных элементов”) (<https://goo.gl/NrKAP0>).

Подобно нейронным сетям прямого распространения мы можем вычислить выход рекуррентного слоя сразу для целого мини-пакета, помещая все входы на временном шаге t во входную матрицу $X_{(t)}$ (уравнение 14.2).

Уравнение 14.2. Выходы слоя рекуррентных нейронов для всех образцов в мини-пакете

$$\begin{aligned} Y_{(t)} &= \phi(X_{(t)} \cdot W_x + Y_{(t-1)} \cdot W_y + b) \\ &= \phi\left(\begin{bmatrix} X_{(t)} & Y_{(t-1)} \end{bmatrix} \cdot W + b\right) \quad \text{с} \quad W = \begin{bmatrix} W_x \\ W_y \end{bmatrix} \end{aligned}$$

- $Y_{(t)}$ — матрица $m \times n_{\text{нейронов}}$, содержащая выходы слоя на временном шаге t для каждого образца в мини-пакете (m — количество образцов в мини-пакете, а $n_{\text{нейронов}}$ — количество нейронов).
- $X_{(t)}$ — матрица $m \times n_{\text{входов}}$, содержащая входы для всех образцов ($n_{\text{входов}}$ — количество входных признаков).
- W_x — матрица $n_{\text{входов}} \times n_{\text{нейронов}}$, содержащая веса связей для входов текущего временного шага.
- W_y — матрица $n_{\text{нейронов}} \times n_{\text{нейронов}}$, содержащая веса связей для выходов предыдущего временного шага.
- b — вектор размера $n_{\text{нейронов}}$, содержащий член смещения каждого нейрона.
- Матрицы весов W_x и W_y часто вертикально объединяются в единственную матрицу весов W формы $(n_{\text{входов}} + n_{\text{нейронов}}) \times n_{\text{нейронов}}$ (см. вторую строку в уравнении 14.2).
- Обозначение $[X_{(t)} \ Y_{(t-1)}]$ представляет горизонтальное объединение матриц $X_{(t)}$ и $Y_{(t-1)}$.

Обратите внимание, что $Y_{(t)}$ — функция от $X_{(t)}$ и $Y_{(t-1)}$, которая является функцией от $X_{(t-1)}$ и $Y_{(t-2)}$, являющейся функцией от $X_{(t-2)}$ и $Y_{(t-3)}$, и т.д. Это делает $Y_{(t)}$ функцией от всех входов с временного шага $t = 0$ (т.е. $X_{(0)}$, $X_{(1)}$, ..., $X_{(t)}$). На первом временном шаге, $t = 0$, предшествующие выходы отсутствуют, потому что обычно предполагается, что все они нулевые.

Ячейки памяти

Поскольку выход рекуррентного нейрона на временном шаге t — это функция от всех входов из предшествующих временных шагов, можно было бы сказать, что он обладает некоторой формой *памяти*. Часть нейронной сети, которая сохраняет состояние через временные шаги, называется *ячейкой памяти* (*memory cell*) или просто *ячейкой*. Одиночный рекуррентный нейрон или слой рекуррентных нейронов представляет собой самую *базовую ячейку*, но позже в главе мы рассмотрим ряд более сложных и мощных типов ячеек.

В общем случае состояние ячейки на временном шаге t , обозначаемое $\mathbf{h}_{(t)}$ (“h” означает “hidden” — “скрытый”), является функцией от некоторых входов на данном временном шаге и ее состояния на предыдущем временном шаге: $\mathbf{h}_{(t)} = f(\mathbf{h}_{(t-1)}, \mathbf{x}_{(t)})$. Выход ячейки на временном шаге t , обозначаемый $\mathbf{y}_{(t)}$, также представляет собой функцию от предыдущего состояния и текущих входов. Для базовых ячеек, которые обсуждались до сих пор, выход просто равен состоянию, но в более сложных ячейках это не всегда так (рис. 14.3).

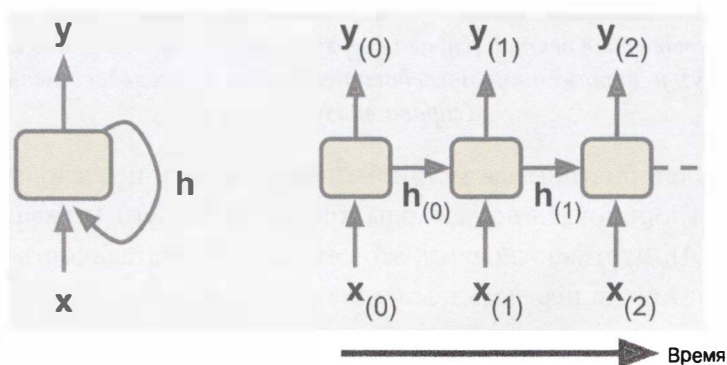


Рис. 14.3. Скрытое состояние ячейки и ее выход могут отличаться

Входные и выходные последовательности

Сеть RNN может одновременно принимать последовательность входов и порождать последовательность выходов (левая верхняя сеть на рис. 14.4). Сеть такого типа полезна, например, для прогнозирования временных рядов, подобных курсам акций: вы передаете сети цены за последние N дней, а она должна выдать цены, сдвинутые на один день в будущее (т.е. от $N - 1$ дней тому назад до завтрашнего дня).

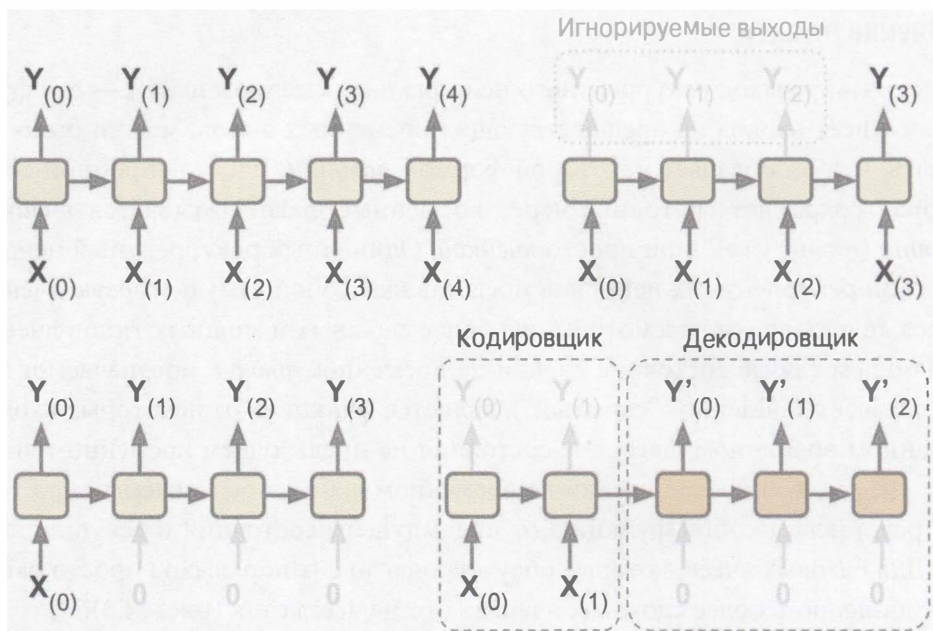


Рис. 14.4. Последовательность в последовательность (слева сверху), последовательность в вектор (справа сверху), вектор в последовательность (слева снизу) и отложенная последовательность в последовательность (справа снизу)

В качестве альтернативы вы могли бы передать сети последовательность входов и проигнорировать все выходы кроме последнего (правая верхняя сеть на рис. 14.4). Другими словами, это сеть “последовательность в вектор”. Например, сети можно передать последовательность слов, соответствующих рецензии на фильм, и она выдаст оценку отношения (скажем, от -1 (не понравилось) до $+1$ (понравилось)).

И наоборот, вы могли бы передать сети одиночный вход на первом временном шаге (и нули для всех остальных временных шагов) и позволить ей выдать последовательность (левая нижняя сеть на рис. 14.4). Это сеть “вектор в последовательность”. Например, входом может быть изображение, а выходом — подпись изображения.

Наконец, вы могли бы иметь сеть “последовательность в вектор”, которая называется *кодировщиком* (*encoder*), а за ней сеть “вектор в последовательность”, называемую *декодировщиком* (*decoder*), что показано справа внизу на рис. 14.4. Такую конфигурацию можно применять для перевода предложения с одного языка на другой. Сети передается предложение на одном языке,

кодировщик преобразует его в представление, имеющее форму одиночного вектора, после чего декодировщик превратит вектор в предложение на другом языке. Эта двухшаговая модель, называемая “кодировщик–декодировщик”, работает намного лучше, чем попытка перевода на лету с помощью единственной сети RNN типа “последовательность в последовательность” (вроде приведенной слева вверху на рис. 14.4). Дело в том, что последние слова предложения могут повлиять на первые слова перевода (в случае английского языка — *прим.пер.*), а потому необходимо подождать завершения предложения, прежде чем переводить его.

Звучит многообещающе, так что давайте приступим к написанию кода!

Базовые рекуррентные нейронные сети в TensorFlow

Сначала мы реализуем очень простую модель RNN, не используя операции RNN из TensorFlow, чтобы лучше понять происходящее внутри. Мы создадим сеть RNN, содержащую слой из пяти рекуррентных нейронов (похожую на сеть RNN, представленную на рис. 14.2), которые применяют функцию активации в виде гиперболического тангенса. Мы будем считать, что сеть RNN проходит только через два временных шага, принимая на каждом из них входные векторы размера 3. Следующий код строит такую сеть RNN, развернутую в два временных шага:

```
n_inputs = 3
n_neurons = 5

X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

Wx = tf.Variable(tf.random_normal(shape=[n_inputs, n_neurons],
                                   dtype=tf.float32))
Wy = tf.Variable(tf.random_normal(shape=[n_neurons, n_neurons],
                                   dtype=tf.float32))
b = tf.Variable(tf.zeros([1, n_neurons], dtype=tf.float32))

Y0 = tf.tanh(tf.matmul(X0, Wx) + b)
Y1 = tf.tanh(tf.matmul(Y0, Wy) + tf.matmul(X1, Wx) + b)

init = tf.global_variables_initializer()
```

Эта сеть выглядит очень похожей на двухслойную нейронную сеть прямого распространения с несколькими ухищрениями: во-первых, оба слоя разделяют те же самые веса и члены смещения и, во-вторых, каждому слою передаются входы и из каждого слоя получаются выходы.

Чтобы прогнать модель, необходимо передать ей входы на обоих временных шагах:

```
import numpy as np
```

```
# Мини-пакет:           образец 0, образец 1, образец 2, образец 3
X0_batch = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 0, 1]]) # t=0
X1_batch = np.array([[9, 8, 7], [0, 0, 0], [6, 5, 4], [3, 2, 1]]) # t=1

with tf.Session() as sess:
    init.run()
    Y0_val, Y1_val = sess.run([Y0, Y1],
                               feed_dict={X0: X0_batch, X1: X1_batch})
```

Мини-пакет содержит четыре образца, каждый с входной последовательностью, состоящей из двух входов. В конце `Y0_val` и `Y1_val` содержат выходы сети на обоих временных шагах для всех нейронов и всех образцов в мини-пакете:

```
>>> print(Y0_val) # ВЫХОД в t = 0
[[-0.0664006  0.96257669 0.68105787 0.70918542 -0.89821595] #образец 0
 [ 0.9977755 -0.71978885 -0.99657625 0.9673925 -0.99989718] #образец 1
 [ 0.99999774 -0.99898815 -0.99999893 0.99677622 -0.99999988] #образец 2
 [ 1.          -1.          -1.          -0.99818915 0.99950868]] #образец 3

>>> print(Y1_val) # ВЫХОД в t = 1
[[ 1.          -1.          -1.          0.40200216 -1.          ] #образец 0
 [-0.12210433 0.62805319 0.96718419 -0.99371207 -0.25839335] #образец 1
 [ 0.99999827 -0.9999994 -0.9999975 -0.85943311 -0.9999879 ] #образец 2
 [ 0.99928284 -0.99999815 -0.99990582 0.98579615 -0.92205751]] #образец 3
```

Работа была не слишком трудной, но если нужно иметь возможность прогона сети RNN через 100 временных шагов, тогда граф станет довольно большим. Теперь давайте посмотрим, как создать ту же самую модель с использованием операций RNN из TensorFlow.

Статическое развертывание во времени

Функция `static_rnn()` создает развернутую сеть RNN путем формирования цепочки ячеек. Приведенный ниже код строит точно такую же модель, как и предыдущий код:

```
X0 = tf.placeholder(tf.float32, [None, n_inputs])
X1 = tf.placeholder(tf.float32, [None, n_inputs])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, [X0, X1],
                                                dtype=tf.float32)

Y0, Y1 = output_seqs
```

Прежде всего, мы создаем входные заполнители, как и ранее. Затем мы создаем объект `BasicRNNCell`, о котором можно думать как о фабрике, создающей копии ячейки для построения развернутой сети RNN (по одной для каждого временного шага). Затем мы вызываем функцию `static_rnn()`, передавая ей фабрику ячеек и входные тензоры, а также сообщая тип данных входов (это применяется для создания начальной матрицы состояния, которая по умолчанию заполнена нулями).

Функция `static_rnn()` вызывает функцию `__call__()` фабрики ячеек по одному разу на вход, создавая две копии ячейки (каждая содержит слой из пяти рекуррентных нейронов), с разделяемыми весами и членами смещения, и формирует из них цепочку, как делалось ранее. Функция `static_rnn()` возвращает два объекта. Первый — список Python с выходными тензорами для каждого временного шага. Второй — тензор, содержащий финальные состояния сети. Когда используются базовые ячейки, финальное состояние просто равно последнему выводу.

Если бы временных шагов было 50, то определять 50 входных заполнителей и 50 выходных тензоров оказалось бы не особенно удобно. Кроме того, во время выполнения пришлось бы снабжать данными каждый из 50 заполнителей и манипулировать 50 выходами.

Давайте упростим положение дел. Показанный далее код строит ту же самую сеть RNN, но на этот раз она принимает единственный входной заполнитель с формой `[None, n_steps, n_inputs]`, где первое измерение представляет собой размер мини-пакета. Затем извлекается список входных последовательностей для каждого временного шага. Здесь `X_seqs` — список Python из `n_steps` тензоров с формой `[None, n_inputs]`, где первое измерение снова является размером мини-пакета. Для этого мы сначала переставляем первые два измерения с применением функции `transpose()`, так что временные шаги становятся первым измерением. Затем мы извлекаем список Python тензоров по первому измерению (т.е. один тензор на временной шаг), используя функцию `unstack()`. Следующие две строки остались такими же, как раньше. Наконец, мы объединяем все выходные тензоры в единственный тензор с применением функции `stack()` и переставляем первые два измерения, чтобы получить финальный тензор `outputs` с формой `[None, n_steps, n_neurons]` (где первое измерение опять представляет собой размер мини-пакета).

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
X_seqs = tf.unstack(tf.transpose(X, perm=[1, 0, 2]))
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
output_seqs, states = tf.contrib.rnn.static_rnn(basic_cell, X_seqs,
                                                dtype=tf.float32)
outputs = tf.transpose(tf.stack(output_seqs), perm=[1, 0, 2])
```

Далее мы можем прогнать сеть, передав ей одиночный тензор, который содержит все последовательности мини-пакетов:

```
X_batch = np.array([
    # t = 0      t = 1
    [[0, 1, 2], [9, 8, 7]], # образец 0
    [[3, 4, 5], [0, 0, 0]], # образец 1
    [[6, 7, 8], [6, 5, 4]], # образец 2
    [[9, 0, 1], [3, 2, 1]], # образец 3
])
with tf.Session() as sess:
    init.run()
    outputs_val = outputs.eval(feed_dict={X: X_batch})
```

Затем мы получаем единственный тензор `outputs_val` для всех образцов, всех временных шагов и всех нейронов:

```
>>> print(outputs_val)
[[[-0.91279727  0.83698678 -0.89277941  0.80308062 -0.5283336 ]
  [-1.          1.          -0.99794829  0.99985468 -0.99273592]]
 [[-0.99994391  0.99951613 -0.9946925   0.99030769 -0.94413054]
  [ 0.48733309  0.93389565 -0.31362072  0.88573611  0.2424476 ]]
 [[-1.          0.99999875 -0.99975014  0.99956584 -0.99466234]
  [-0.99994856  0.99999434 -0.96058172  0.99784708 -0.9099462 ]]
 [[-0.95972425  0.99951482  0.96938795 -0.969908   -0.67668229]
  [-0.84596014  0.96288228  0.96856463 -0.14777924 -0.9119423 ]]]
```

Тем не менее, такой подход по-прежнему строит граф, содержащий по одной ячейке на временной шаг. Если бы временных шагов было 50, тогда граф выглядел бы довольно неуклюже. Это немного напоминает написание программы без использования циклов (скажем, `Y0=f(0, X0); Y1=f(Y0, X1); Y2=f(Y1, X2); ...; Y50=f(Y49, X50)`). Настолько крупный граф даже способен привести к нехватке памяти при обратном распространении (особенно в случае ограниченной памяти плат ГП), т.к. в ней должны храниться значения всех тензоров в течение прохода вперед, чтобы их можно было задействовать при вычислении градиентов во время шага назад. К счастью, существует более удачное решение: функция `dynamic_rnn()`.

Динамическое развертывание во времени

Функция `dynamic_rnn()` применяет операцию `while_loop()` для прохода по ячейке подходящее количество раз. Если вы хотите, чтобы при обратном распространении она меняла местами память ГП и память ЦП во избежание ошибок, связанных с нехваткой памяти, то можете установить `swap_memory=True`. Удобство функции `dynamic_rnn()` в том, что на каждом временном шаге она также принимает единственный тензор для всех входов (с формой `[None, n_steps, n_inputs]`) и выдает единственный тензор для всех выходов (с формой `[None, n_steps, n_neurons]`); необходимость в операциях `stack()`, `unstack()` или `transpose()` отсутствует. Следующий код создает ту же самую сеть RNN, что и ранее, с использованием функции `dynamic_rnn()`. Насколько же он изящнее!

```
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)
```



При обратном распространении операция `while_loop()` делает соответствующую магию: она сохраняет значения тензоров для каждой итерации в течение прохода вперед, поэтому может их применять для вычисления градиентов во время шага назад.

Обработка входных последовательностей переменной длины

До настоящего времени мы использовали только входные последовательности фиксированного размера (все с длиной в точности два шага). Что, если входные последовательности имеют переменную длину (скажем, как у предложений)? В таком случае при вызове функции `dynamic_rnn()` (или `static_rnn()`) потребуется установить аргумент `sequence_length`; он должен быть одномерным тензором, указывающим длину входной последовательности для каждого образца. Вот пример:

```
seq_length = tf.placeholder(tf.int32, [None])
[...]
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32,
                                   sequence_length=seq_length)
```


Предположим, что вторая входная последовательность содержит не два, а только один вход. Чтобы приспособиться к входному тензору `X`, она должна быть дополнена нулевым вектором (поскольку вторым измерением входного тензора является размер самой длинной последовательности, т.е. 2).

```
X_batch = np.array([
    # шаг 0      шаг 1
    [[0, 1, 2], [9, 8, 7]], # образец 0
    [[3, 4, 5], [0, 0, 0]], # образец 1 (дополненный нулевым вектором)
    [[6, 7, 8], [6, 5, 4]], # образец 2
    [[9, 0, 1], [3, 2, 1]], # образец 3
])
seq_length_batch = np.array([2, 1, 2, 2])
```

Конечно, теперь вам нужно предоставить значения для заполнителей `X` и `seq_length`:

```
with tf.Session() as sess:
    init.run()
    outputs_val, states_val = sess.run(
        [outputs, states],
        feed_dict={X: X_batch, seq_length: seq_length_batch})
```

Сеть RNN выдает нулевые векторы для каждого временного шага за пределами длины последовательности (взгляните на выход второго образца для второго временного шага):

```
>>> print(outputs_val)
[[[-0.68579948 -0.25901747 -0.80249101 -0.18141513 -0.37491536]
  [-0.99996698 -0.94501185  0.98072106 -0.9689762  0.99966913]]
   # финальное состояние

[[-0.99099374 -0.64768541 -0.67801034 -0.7415446  0.7719509 ]
   # финальное состояние

 [ 0.          0.          0.          0.          0.          ]
   # нулевой вектор

[[-0.99978048 -0.85583007 -0.49696958 -0.93838578  0.98505187]
  [-0.99951065 -0.89148796  0.94170523 -0.38407657  0.97499216]]
   # финальное состояние

[[-0.02052618 -0.94588047  0.99935204  0.37283331  0.9998163 ]
  [-0.91052347  0.05769409  0.47446665 -0.44611037  0.89394671]]]
   # финальное состояние
```

Кроме того, тензор `states` содержит финальное состояние каждой ячейки (исключая нулевые векторы):

```
>>> print(states_val)
```

```
[[-0.99996698 -0.94501185  0.98072106 -0.9689762  0.99966913] # t=1  
[-0.99099374 -0.64768541 -0.67801034 -0.7415446  0.7719509 ] # t=0!!!  
[-0.99951065 -0.89148796  0.94170523 -0.38407657  0.97499216] # t=1  
[-0.91052347  0.05769409  0.47446665 -0.44611037  0.89394671]] # t=1
```

Обработка выходных последовательностей переменной длины

А что, если переменную длину имеют также и выходные последовательности? Когда вы знаете заранее, какую длину будет иметь каждая последовательность (например, если вам известно, что она будет такой же длины, как и входная последовательность), то можете установить параметр `sequence_length`, который был описан выше. К сожалению, в общем случае это будет невозможно: скажем, длина переведенного предложения обычно отличается от длины входного предложения. Самое распространенное решение в такой ситуации предусматривает определение специального выхода, называемого *маркером конца последовательности* (*End-Of-Sequence (EOS) token*). Любой выход после маркера EOS должен игнорироваться (мы обсудим это позже в главе).

Итак, вы знаете, каким образом строить сеть RNN (точнее сеть RNN, развернутую во времени). Но как вы будете ее обучать?

Обучение рекуррентных нейронных сетей

Чтобы обучить сеть RNN, ее понадобится развернуть во времени (как только что делалось) и просто применить обыкновенное обратное распространение (рис. 14.5). Такая стратегия называется *обратным распространением во времени* (*Backpropagation Through Time — BPTT*).

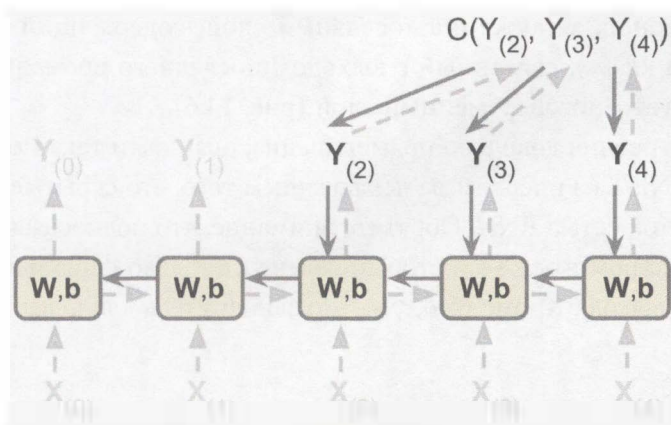


Рис. 14.5. Обратное распространение во времени

Подобно обыкновенному обратному распространению имеется первый проход вперед через развернутую сеть (представлен пунктирными линиями со стрелками). Затем выходная последовательность оценивается с использованием функции издержек $C(Y_{(t_{\min})}, Y_{(t_{\min} + 1)}, \dots, Y_{(t_{\max})})$ (где t_{\min} и t_{\max} — первый и последний временные шаги выходов, не считая игнорируемых выходов). Градиенты этой функции издержек распространяются в обратном направлении через развернутую сеть (что представлено сплошными линиями со стрелками). Наконец, параметры модели обновляются с применением градиентов, вычисленных во время ВРТТ. Обратите внимание, что градиенты протекают назад через все выходы, использованные функцией издержек, а не только через финальный выход (к примеру, на рис. 14.5 функция издержек вычисляется с применением последних трех выходов сети, $Y_{(2)}$, $Y_{(3)}$ и $Y_{(4)}$, поэтому градиенты протекают через указанные три выхода, но не через $Y_{(0)}$ и $Y_{(1)}$). Более того, поскольку на каждом временном шаге использовались те же самые параметры W и b , обратное распространение будет поступать правильно и суммировать по всем временным шагам.

Обучение классификатора последовательностей

Давайте обучим сеть RNN с целью классификации изображений MNIST. Для классификации изображений больше бы подошла сверточная нейронная сеть (см. главу 13), но мы решили взять простой пример, с которым вы уже знакомы. Мы будем трактовать каждое изображение как последовательность из 28 строк по 28 пикселей в каждой (потому что каждое изображение MNIST имеет 28×28 пикселей). Мы будем применять ячейки из 150 рекуррентных нейронов, а также полносвязный слой, содержащий 10 нейронов (по одному на класс), связанный с выходом последнего временного шага, за которым следует многопеременный слой (рис. 14.6).

Стадия построения довольно прямолинейна; она почти такая же, как у классификатора MNIST из главы 10, за исключением того, что скрытые слои заменяются развернутой сетью RNN. Обратите внимание, что полносвязный слой связывается с тензором `states`, который содержит только финальное состояние RNN (т.е. 28-й выход). Кроме того, `y` — это заполнитель для целевых классов.

```
n_steps = 28
n_inputs = 28
n_neurons = 150
n_outputs = 10
```

```

learning_rate = 0.001
X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.int32, [None])

basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
outputs, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)

logits = tf.layers.dense(states, n_outputs)
xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                           logits=logits)

loss = tf.reduce_mean(xentropy)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
correct = tf.nn.in_top_k(logits, y, 1)
accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))

init = tf.global_variables_initializer()

```

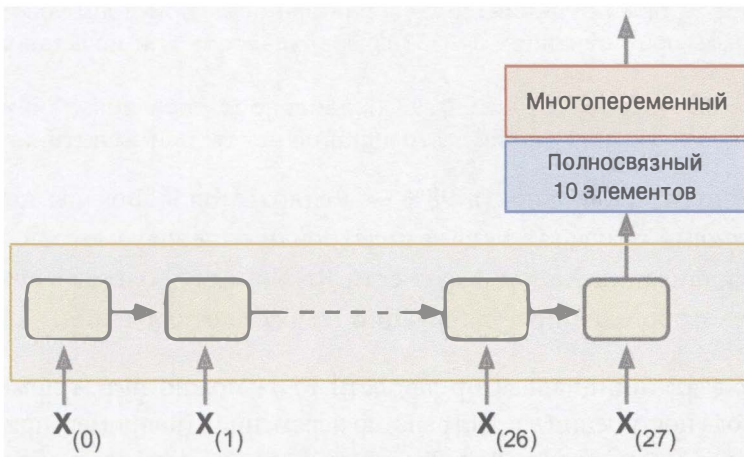


Рис. 14.6. Классификатор последовательностей

Теперь загрузим набор данных MNIST и изменим форму испытательных данных на `[batch_size, n_steps, n_inputs]`, которую ожидает сеть. Вскоре мы позаботимся об изменении формы также и обучающих данных.

```

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
X_test = mnist.test.images.reshape((-1, n_steps, n_inputs))
y_test = mnist.test.labels

```

Для обучения сети RNN все готово. Стадия выполнения является точно такой же, как у классификатора MNIST из главы 10, кроме того, что мы изменяем форму обучающего пакета до его передачи сети.

```

n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
            acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
            acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
            print(epoch, "Правильность при обучении:", acc_train,
                  "Правильность при испытании:", acc_test)

```

Вывод должен выглядеть так:

```

0 Правильность при обучении: 0.94 Правильность при испытании: 0.9308
1 Правильность при обучении: 0.933333 Правильность при испытании: 0.9431
[...]
98 Правильность при обучении: 0.98 Правильность при испытании: 0.9794
99 Правильность при обучении: 1.0 Правильность при испытании: 0.9804

```

Мы получили правильность 98% — неплохо! Вдобавок вы непременно получите лучший результат за счет подстройки гиперпараметров, использования инициализации Хе для весов сети RNN, более долгого обучения или введения некоторой доли регуляризации (скажем, отключения).



Указать инициализатор для сети RNN можно путем помещения кода построения в пространство переменной (например, применить `variable_scope("rnn", initializer=variance_scaling_initializer())` для использования инициализации Хе).

Обучение для прогнозирования временных рядов

Давайте теперь выясним, как обрабатывать временные ряды, подобные курсам акций, температуре воздуха, картине мозговых волн и т.д. В этом разделе мы будем обучать сеть RNN для прогнозирования следующего значения в сгенерированном временном ряде. Каждый обучающий образец представляет собой случайно выбранную из временного ряда последовательность 20 следующих друг за другом значений, а целевая последовательность — та же, что и входная, но сдвинутая на один временной шаг в будущее (рис. 14.7).

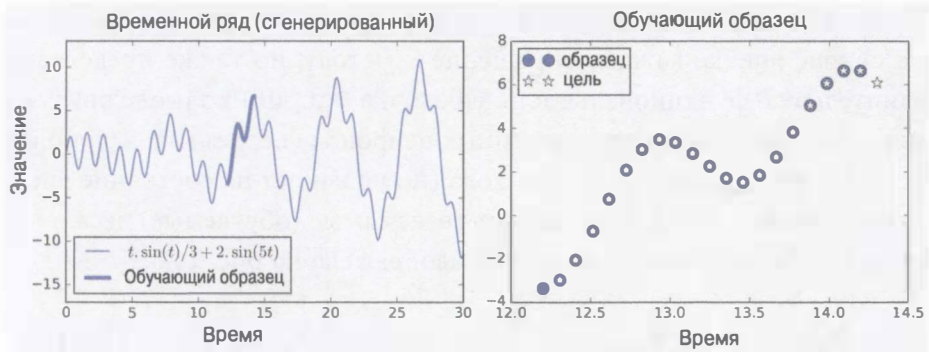


Рис. 14.7. Временной ряд (слева) и обучающий образец из этого ряда (справа)

Прежде всего, создадим сеть RNN. Она будет содержать 100 рекуррентных нейронов, и мы развернем ее на 20 временных шагов, т.к. каждый обучающий образец будет иметь длину в 20 входов. Каждый вход будет содержать только один признак (значение в тот момент времени). Цели также представляют собой последовательности из 20 входов с единственным значением каждая. Код в основном такой же, как ранее:

```
n_steps = 20
n_inputs = 1
n_neurons = 100
n_outputs = 1

X = tf.placeholder(tf.float32, [None, n_steps, n_inputs])
y = tf.placeholder(tf.float32, [None, n_steps, n_outputs])
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
                                   activation=tf.nn.relu)
outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```



В общем случае вы будете иметь более одного входного признака. Например, при прогнозировании курса акций на каждом временном шаге существовало бы много других входных признаков, таких как цены конкурирующих акций, рейтинги от аналитиков и прочие признаки, которые могут помочь системе вырабатывать прогнозы.

На каждом временном шаге теперь имеется выходной вектор с размером 100. Но в действительности на каждом временном шаге нас интересует единственное выходное значение. Простейшее решение предусматривает помещение ячейки в оболочку `OutputProjectionWrapper`.

Оболочка ячейки действует подобно нормальной ячейке, передавая лежащей в основе ячейке каждое обращение к методу, но также предоставляет дополнительную функциональность. Оболочка `OutputProjectionWrapper` добавляет полносвязный слой линейных нейронов (т.е. без какой-либо функции активации) поверх каждого выхода (но не влияет на состояние ячейки). Все эти полносвязные слои разделяют те же самые (обучаемые) веса и члены смещения. Результирующая сеть RNN изображена на рис. 14.8.

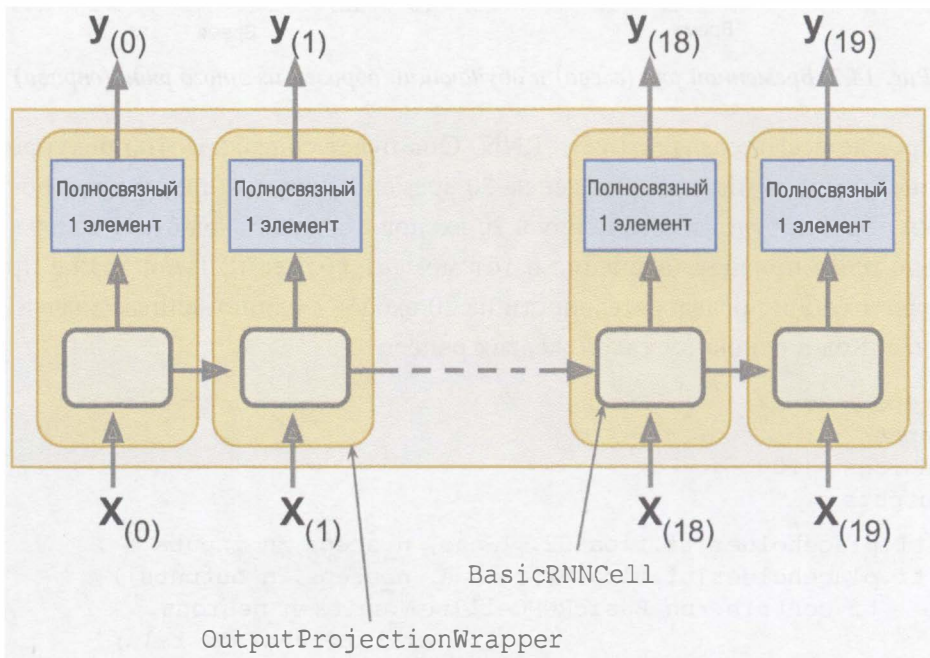


Рис. 14.8. Ячейки RNN, применяющие выходные проекции

Поместить ячейку в оболочку довольно легко. Скорректируем предыдущий код, поместив `BasicRNNCell` в оболочку `OutputProjectionWrapper`:

```
cell = tf.contrib.rnn.OutputProjectionWrapper(
    tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
                                activation=tf.nn.relu), output_size=n_outputs)
```

Пока все хорошо. Теперь необходимо определить функцию издержек. Мы будем использовать среднеквадратическую ошибку (MSE), как ранее поступали при решении задач регрессии. Затем, как обычно, мы создадим оптимизатор Adam, операцию обучения и операцию инициализации переменных:

```

learning_rate = 0.001
loss = tf.reduce_mean(tf.square(outputs - y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)
init = tf.global_variables_initializer()

```

А вот стадия выполнения:

```

n_iterations = 1500
batch_size = 50
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = [...] # извлечь следующий обучающий пакет
        sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        if iteration % 100 == 0:
            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

```

Вывод программы выглядит примерно так:

```

0      MSE: 13.6543
100    MSE: 0.538476
200    MSE: 0.168532
300    MSE: 0.0879579
400    MSE: 0.0633425
[...]

```

После обучения модели можно вырабатывать прогнозы:

```

X_new = [...] # Новые последовательности
y_pred = sess.run(outputs, feed_dict={X: X_new})

```

На рис. 14.9 показана прогнозируемая последовательность для образца, который мы видели ранее (на рис. 14.7), после всего лишь 1 000 итераций обучения.

Хотя применение оболочки `OutputProjectionWrapper` является простейшим решением для понижения размерности выходных последовательностей сети RNN до только одного значения на временной шаг (на образец), оно не самое эффективное. Существует более сложное, но и эффективное решение: поменять форму выходов RNN с `[batch_size, n_steps, n_neurons]` на `[batch_size * n_steps, n_neurons]`, после чего применить одиночный полносвязный слой подходящего размера (в нашем случае 1), который даст в результате выходной тензор с формой `[batch_size * n_steps, n_outputs]`, и затем изменить форму данного тензора на `[batch_size, n_steps, n_outputs]`. Такие операции представлены на рис. 14.10.

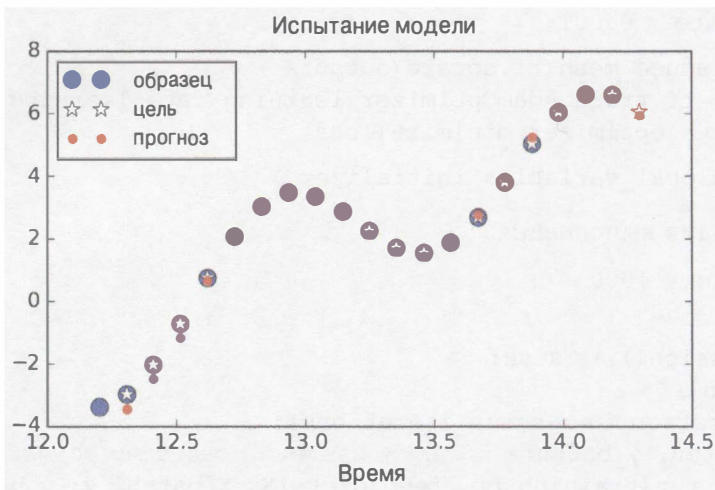


Рис. 14.9. Прогнозирование временного ряда

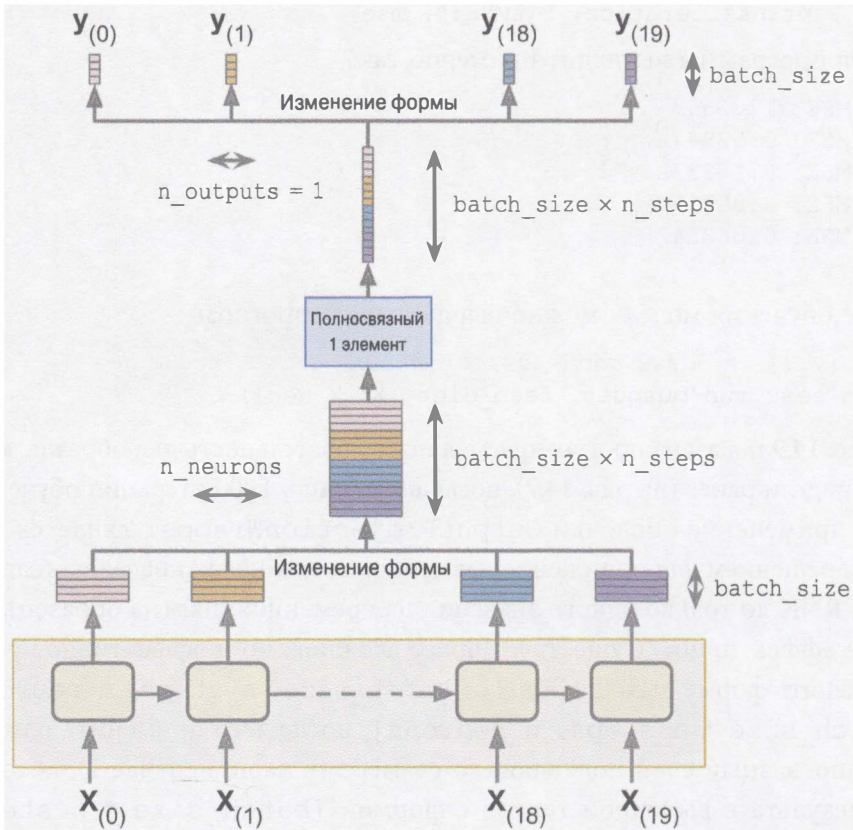


Рис. 14.10. Укладывание в стопку всех выходов, применение проекции и восстановление результата

Чтобы реализовать описанное решение, мы сначала возвратимся к базовой ячейке без оболочки `OutputProjectionWrapper`:

```
cell = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,  
                                   activation=tf.nn.relu)  
rnn_outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

Затем мы укладываем в стопку все выходы, используя операцию `reshape()`, применяем полносвязный линейный слой (без какой-либо функции активации; это просто проекция) и в конце восстанавливаем все выходы, снова используя `reshape()`:

```
stacked_rnn_outputs = tf.reshape(rnn_outputs, [-1, n_neurons])  
stacked_outputs = tf.layers.dense(stacked_rnn_outputs, n_outputs)  
outputs = tf.reshape(stacked_outputs, [-1, n_steps, n_outputs])
```

Остальной код остался таким же, как ранее. В итоге может быть обеспечен значительный прирост скорости, поскольку есть всего лишь один полносвязный слой вместо одного такого слоя на временной шаг.

Креативная рекуррентная нейронная сеть

Располагая моделью, которая способна прогнозировать будущее, мы можем применять ее для генерации ряда креативных последовательностей, как упоминалось в начале главы. Все, что понадобится — предоставить модели начальную последовательность, содержащую `n_steps` значений (например, все нули), использовать модель для прогнозирования следующего значения, присоединить спрогнозированное значение к последовательности, передать модели заключительные `n_steps` значений для прогнозирования следующего значения и т.д. Такой процесс генерирует новую последовательность, которая имеет определенное сходство с начальным временным рядом (рис. 14.11).

```
sequence = [0.] * n_steps  
for iteration in range(300):  
    X_batch = np.array(sequence[-n_steps:]).reshape(1, n_steps, 1)  
    y_pred = sess.run(outputs, feed_dict={X: X_batch})  
    sequence.append(y_pred[0, -1, 0])
```

Далее вы можете передать сети RNN все альбомы Джона Леннона и посмотреть, сумеет ли она сгенерировать песню “Imagine” (“Представь себе”). Однако вероятно вам потребуется намного более мощная сеть RNN, с большим числом нейронов и гораздо большей глубиной. Давайте рассмотрим глубокие сети RNN.

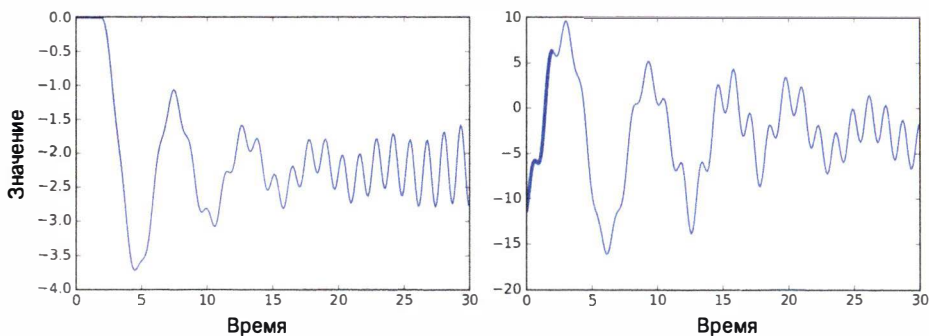


Рис. 14.11. Креативные последовательности с начальными нулями (слева) или с образцом (справа)

Глубокие рекуррентные нейронные сети

Множество слоев ячеек довольно часто укладывают друг на друга, как показано на рис. 14.12. В результате получается *глубокая сеть RNN*.

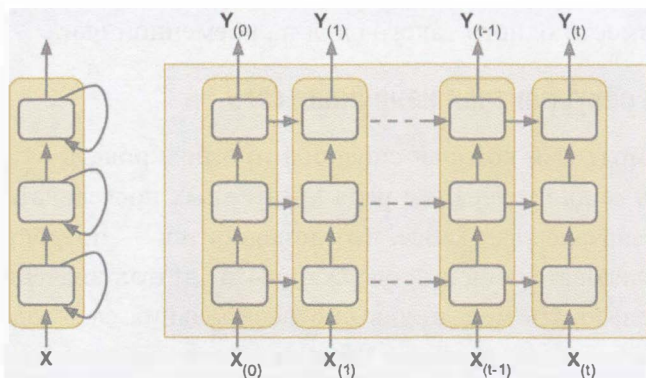


Рис. 14.12. Глубокая сеть RNN (слева), развернутая во времени (справа)

Чтобы реализовать глубокую сеть RNN в TensorFlow, вы можете создать несколько ячеек и уложить их стопкой в `MultiRNNCell`. В следующем коде мы укладываем три идентичных ячейки (но вы прекрасно могли бы применять разнообразные виды ячеек с разным количеством нейронов):

```
n_neurons = 100
n_layers = 3

layers = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons,
                                       activation=tf.nn.relu)

    for layer in range(n_layers)]

multi_layer_cell = tf.contrib.rnn.MultiRNNCell(layers)
outputs, states = tf.nn.dynamic_rnn(multi_layer_cell, X,
                                     dtype=tf.float32)
```


Вот и все! Переменная `states` — это кортеж, в котором для каждого слоя содержится один тензор, представляющий финальное состояние ячейки данного слоя (с формой `[batch_size, n_neurons]`). Если при создании `MultiRNNCell` установить `state_is_tuple=False`, тогда `states` становится одиночным тензором, который содержит состояния из каждого слоя, объединенные по оси столбцов (т.е. его форма выглядит как `[batch_size, n_layers * n_neurons]`). Обратите внимание, что до появления версии TensorFlow 0.11.0 такое поведение было стандартным.

Распределение глубокой рекуррентной нейронной сети между множеством графических процессоров

В главе 12 было показано, что мы можем эффективно распределять глубокие сети RNN между множеством ГП, прикрепляя каждый слой к своему ГП (см. рис. 12.16). Тем не менее, если вы попытаетесь создавать каждую ячейку в отличающемся блоке `device()`, то код работать не будет:

```
with tf.device("/gpu:0"): # НЕПРАВИЛЬНО! Игнорируется.
    layer1 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
with tf.device("/gpu:1"): # НЕПРАВИЛЬНО! Также игнорируется.
    layer2 = tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
```

Код отказывается функционировать, потому что `BasicRNNCell` — это фабрика ячеек, а не сама ячейка (как упоминалось ранее); при создании фабрики никакие ячейки не создаются, соответственно нет и переменных. Блок устройства попросту игнорируется. Фактически ячейки создаются позже. Когда вызывается функция `dynamic_rnn()`, она обращается к объекту `MultiRNNCell`, обращающемуся к каждому индивидуальному объекту `BasicRNNCell`, который создает действительные ячейки (включая их переменные). К сожалению, ни один из указанных классов не предоставляет какого-то способа управления устройствами, на которых создаются переменные. Если вы поместите вызов `dynamic_rnn()` внутрь блока устройства, тогда сеть RNN целиком прикрепится к единственному устройству. Значит, вы застряли? К счастью, нет! Фокус в том, чтобы создать собственную оболочку ячеек (либо воспользоваться классом `tf.contrib.rnn.DeviceWrapper`, который был добавлен в TensorFlow 1.1):

```
import tensorflow as tf
class DeviceCellWrapper(tf.contrib.rnn.RNNCell):
```



```

def __init__(self, device, cell):
    self._cell = cell
    self._device = device

@property
def state_size(self):
    return self._cell.state_size

@property
def output_size(self):
    return self._cell.output_size

def __call__(self, inputs, state, scope=None):
    with tf.device(self._device):
        return self._cell(inputs, state, scope)

```

Такая оболочка просто переадресует каждый вызов метода другой ячейке, но помещает функцию `__call__()` внутрь блока устройства². Теперь можно распределять каждый слой на свой ГП:

```

devices = ["/gpu:0", "/gpu:1", "/gpu:2"]
cells = [DeviceCellWrapper(dev, tf.contrib.rnn.BasicRNNCell(
    num_units=n_neurons))
    for dev in devices]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells)
outputs, states =
    tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)

```



Не устанавливайте `state_is_tuple=False`, или же `MultiRNNCell` будет объединять все состояния ячеек в единственный тензор на одном ГП.

Применение отключения

Если вы строите очень глубокую сеть RNN, то можете столкнуться с переобучением обучающим набором. Распространенный прием, предотвращающий такую проблему, предусматривает применение отключения (введенного в главе 11). Вы можете просто добавить слой отключения до или после сети RNN обычным образом, но если вы хотите применять отключение также между слоями RNN, тогда придется использовать `DropoutWrapper`. Следующий код применяет отключение к входам каждого слоя в сети RNN:

² Здесь применяется паттерн проектирования “Декоратор” (Decorator).

```

keep_prob = tf.placeholder_with_default(1.0, shape=())
cells = [tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
         for layer in range(n_layers)]
cells_drop =
    [tf.contrib.rnn.DropoutWrapper(cell, input_keep_prob=keep_prob)
     for cell in cells]
multi_layer_cell = tf.contrib.rnn.MultiRNNCell(cells_drop)
rnn_outputs,
    states = tf.nn.dynamic_rnn(multi_layer_cell, X, dtype=tf.float32)
# Остаток стадии построения похож на приведенные ранее.

```

Во время обучения вы можете передать в заполнитель `keep_prob` любое желаемое значение (обычно, 0.5):

```

n_iterations = 1500
batch_size = 50
train_keep_prob = 0.5
with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        X_batch, y_batch = next_batch(batch_size, n_steps)
        _, mse = sess.run([training_op, loss],
                          feed_dict={X: X_batch, y: y_batch,
                                      keep_prob: train_keep_prob})
    saver.save(sess, "./my_dropout_time_series_model")

```

Во время испытаний вы обязаны дать возможность использовать стандартное значение `keep_prob`, фактически запрещая отключение (вспомните, что оно должно быть активным только во время обучения):

```

with tf.Session() as sess:
    saver.restore(sess, "./my_dropout_time_series_model")
    X_new = [...] # испытательные данные
    y_pred = sess.run(outputs, feed_dict={X: X_new})

```

Обратите внимание, что применять отключение допускается и к выходам за счет установки `output_keep_prob`, а начиная с версии TensorFlow 1.1, отключение также можно применять к состоянию ячейки, используя `state_keep_prob`.

С помощью перечисленных средств вы должны быть способны обучать все виды сетей RNN! К сожалению, если сеть RNN нужно обучать на длинных последовательностях, то все становится несколько труднее. Давайте выясним причины и что в их отношении можно предпринять.

Трудность обучения в течение многих временных шагов

Чтобы обучить сеть RNN на длинных последовательностях, вам необходимо прогонять ее через множество временных шагов, делая развернутую сеть RNN очень глубокой. Подобно любой глубокой нейронной сети сеть RNN может страдать от проблемы исчезновения/взрывного роста градиентов (см. главу 11) и обучаться бесконечно долго. К глубоким развернутым сетям RNN могут также применяться многие обсуждаемые ранее трюки, целью которых было смягчение данной проблемы: хорошая инициализация параметров, не-насыщаемые функции активации (например, ReLU), пакетная нормализация, отсечение градиентов и более быстрые оптимизаторы. Однако если сеть RNN нуждается в обработке даже умеренно длинных последовательностей (скажем, 100 входов), тогда ее обучение по-прежнему будет очень медленным.

Простейшее и самое распространенное решение проблемы заключается в том, чтобы во время обучения развертывать сеть RNN только на ограниченное число временных шагов. Прием называется *укороченным обратным распространением во времени* (*truncated backpropagation through time*). Реализовать его в TensorFlow можно за счет укорачивания входных последовательностей. Например, в случае задачи прогнозирования временных рядов вы просто сокращаете `n_steps` во время обучения. Разумеется, возникает проблема, связанная с тем, что модель не будет иметь возможности изучить долговременные образы. В качестве обходного пути можно было бы удостовериться, что укороченные последовательности содержат и старые, и последние данные, чтобы модель могла научиться их использовать (к примеру, в последовательность могли бы входить месячные данные для последних пяти месяцев, недельные данные для последних пяти недель и дневные данные за последние пять дней). Но такой обходной путь имеет свои ограничения: что, если детализированные данные из последнего года на самом деле полезны? Что, если было короткое, но значительное событие, которое, безусловно, должно приниматься во внимание даже спустя годы (скажем, результаты выборов)?

Кроме долгого времени обучения, длительно работающие сети RNN сталкиваются и со второй проблемой — фактом постепенного исчезновения памяти первых входов. Действительно, из-за трансформаций, которым подвергаются данные при проходе через сеть RNN, после каждого временного шага определенная информация утрачивается. Через некоторое время состояние сети RNN практически не содержит следов первых входов. Может возникнуть накладка. Например, пусть вы хотите провести смысловой анализ длин-

ной рецензии, которая начинается четырьмя словами “Мне понравился этот фильм”, но в остатке рецензии перечисляются многие вещи, которые могли бы сделать фильм еще лучше. Если сеть RNN постепенно забывает первые четыре слова, тогда она будет интерпретировать рецензию совершенно ошибочно. Для решения проблемы были введены разнообразные типы ячеек с долговременной памятью. Они оказались настолько успешными, что базовые ячейки больше широко не применяются. Давайте сначала рассмотрим самую популярную из ячеек долговременной памяти: ячейку LSTM.

Ячейка LSTM

Ячейка *долгой краткосрочной памяти* (*Long Short-Term Memory* — *LSTM*) была предложена в 1997 году³ Сеппом Хохрайтером и Юргеном Шмидхубером, а с годами понемногу совершенствовалась такими исследователями, как Алекс Грейвз, Хасим Сак⁴, Войцех Заремба⁵ и многими другими. Если трактовать ячейку LSTM как черный ящик, то ее можно использовать очень похоже на базовую ячейку, но она будет функционировать гораздо лучше; обучение будет сходиться быстрее и обнаруживать установившиеся зависимости в данных. В TensorFlow можно просто применять `BasicLSTMCell` вместо `BasicRNNCell`:

```
lstm_cell = tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
```

Ячейки LSTM поддерживают два вектора состояния, которые по умолчанию хранятся обособленно по причинам, связанным с производительностью. Стандартное поведение можно изменить, установив `state_is_tuple=False` при создании `BasicLSTMCell`.

Каким же образом работает ячейка LSTM? Архитектура элементарной ячейки LSTM показана на рис. 14.13.

Если не смотреть на то, что находится внутри ящика, то ячейка LSTM выглядит в точности как обыкновенная ячейка, но с расщеплением своего

³ “Long Short-Term Memory” (“Долгая краткосрочная память”), С. Хохрайтер и Ю. Шмидхубер (1997 год) (<https://goo.gl/j39AGv>).

⁴ “Long Short-Term Memory Recurrent Neural Network Architectures for Large Scale Acoustic Modeling” (“Архитектуры рекуррентных нейронных сетей с долгой краткосрочной памятью для крупномасштабного акустического моделирования”), Х. Сак и др. (2014 год) (<https://goo.gl/6BHh81>).

⁵ “Recurrent Neural Network Regularization” (“Регуляризация рекуррентных нейронных сетей”), В. Заремба и др. (2015 год) (<https://goo.gl/SZ9kzB>).

состояния на два вектора: $\mathbf{h}_{(t)}$ и $\mathbf{c}_{(t)}$ (“с” обозначает “cell” (“ячейка”)). Можете считать $\mathbf{h}_{(t)}$ краткосрочным состоянием, а $\mathbf{c}_{(t)}$ — долгосрочным состоянием.

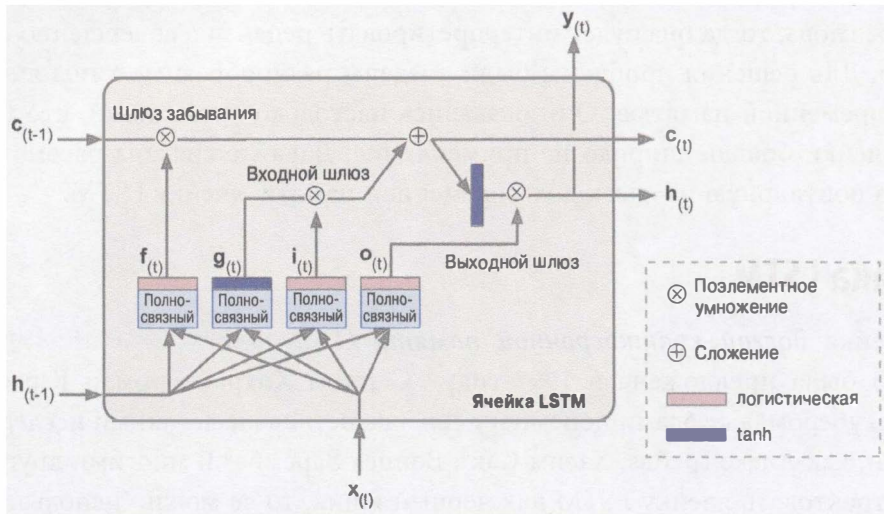


Рис. 14.13. Ячейка LSTM

Давайте приоткроем ящик! Основная идея в том, что сеть может узнать, что хранить в долгосрочном состоянии, что отбрасывать и из чего читать. Во время пересечения сети слева направо долгосрочное состояние $\mathbf{c}_{(t-1)}$ сначала проходит через *шлюз забывания* (*forget gate*) с отбрасыванием некоторых воспоминаний и затем посредством операции сложения к нему добавляется ряд новых воспоминаний (выбранных *входным шлюзом* (*input gate*)). Результат $\mathbf{c}_{(t)}$ отправляется прямо на выход без какой-либо дальнейшей трансформации. Следовательно, на каждом временном шаге одни воспоминания отбрасываются, а другие добавляются. Кроме того, после операции сложения долгосрочное состояние копируется и пропускается через функцию \tanh , а результат фильтруется *выходным шлюзом* (*output gate*). Итогом будет краткосрочное состояние $\mathbf{h}_{(t)}$ (которое равно выходу ячейки для данного временного шага $\mathbf{y}_{(t)}$). Теперь посмотрим, откуда поступают новые воспоминания, и каким образом работают шлюзы.

Первым делом текущий входной вектор $\mathbf{x}_{(t)}$ и предыдущее краткосрочное состояние $\mathbf{h}_{(t-1)}$ передаются четырем полносвязным слоям. Все они служат разным целям.

- Главный слой выводит $\mathbf{g}_{(t)}$. Он выполняет обычную роль, анализируя текущие входы $\mathbf{x}_{(t)}$ и предыдущее (краткосрочное) состояние $\mathbf{h}_{(t-1)}$.

В базовой ячейке нет ничего другого кроме этого слоя, и ее выход поступает прямо в $y_{(t)}$ и $h_{(t)}$. Напротив, в ячейке LSTM выход данного слоя прямо не выдается, а частично сохраняется в долгосрочном состоянии.

- Остальные три слоя являются *контроллерами шлюзов (gate controller)*. Поскольку они используют логистическую функцию активации, их выходы находятся в диапазоне от 0 до 1. Как видите, их выходы передаются операциям поэлементного умножения, так что если они выдают нули, то закрывают шлюз, а если единицы, то открывают его. Более точно:
 - *шлюз забывания* (контролируемый $f_{(t)}$) управляет тем, какие части долгосрочного состояния должны быть разрушены;
 - *входной шлюз* (контролируемый $i_{(t)}$) управляет тем, какие части $g_{(t)}$ должны быть добавлены к долгосрочному состоянию (именно потому речь идет только о “частичном сохранении”);
 - *выходной шлюз* (контролируемый $o_{(t)}$) управляет тем, какие части долгосрочного состояния должны быть прочитаны и выданы на данном временном шаге (в $h_{(t)}$ и $y_{(t)}$).

Короче говоря, ячейка LSTM способна научиться распознавать важный вход (роль входного шлюза), записывать его в долгосрочное состояние, сохранять его столько, сколько нужно (роль шлюза забывания), и извлекать его по мере необходимости. Это объясняет, почему ячейки LSTM были поразительно успешными в сборе долгосрочных образов из временных рядов, длинных текстов, аудиозаписей и многого другого.

В уравнении 14.3 показано, как вычислять долгосрочное состояние ячейки и ее выход на каждом временном шаге для одиночного образца (уравнения для целого мини-пакета очень похожи).

Уравнение 14.3. Вычисления, связанные с ячейкой LSTM

$$\begin{aligned}
 i_{(t)} &= \sigma(W_{xi}^T \cdot x_{(t)} + W_{hi}^T \cdot h_{(t-1)} + b_i) \\
 f_{(t)} &= \sigma(W_{xf}^T \cdot x_{(t)} + W_{hf}^T \cdot h_{(t-1)} + b_f) \\
 o_{(t)} &= \sigma(W_{xo}^T \cdot x_{(t)} + W_{ho}^T \cdot h_{(t-1)} + b_o) \\
 g_{(t)} &= \tanh(W_{xg}^T \cdot x_{(t)} + W_{hg}^T \cdot h_{(t-1)} + b_g) \\
 c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\
 y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)})
 \end{aligned}$$

- W_{xi} , W_{xf} , W_{xo} и W_{xg} — матрицы весов каждого из четырех слоев для их связи с входным вектором $\mathbf{x}_{(t)}$.
- W_{hi} , W_{hf} , W_{ho} и W_{hg} — матрицы весов каждого из четырех слоев для их связи с предыдущим краткосрочным состоянием $\mathbf{h}_{(t-1)}$.
- \mathbf{b}_i , \mathbf{b}_f , \mathbf{b}_o и \mathbf{b}_g — члены смещения для каждого из четырех слоев. Обратите внимание, что TensorFlow инициализирует \mathbf{b}_f вектором, заполненным единицами, а не нулями. Это препятствует забыванию чего-либо в начале обучения.

Смотровые связи

В элементарной ячейке LSTM контроллеры шлюзов могут смотреть только на вход $\mathbf{x}_{(t)}$ и предыдущее краткосрочное состояние $\mathbf{h}_{(t-1)}$. Иногда полезно предоставить им чуть больше контекста, позволив заглядывать также в долгосрочное состояние. Такая идея была выдвинута Феликсом Герсом и Юргеном Шмидгубером в 2000 году⁶. Они предложили вариант ячейки LSTM с дополнительными связями, называемыми *смотровыми связями* (*peephole connection*): к контроллерам шлюза забывания и входного шлюза в качестве входа добавляется предыдущее долгосрочное состояние $\mathbf{c}_{(t-1)}$, а к контроллеру выходного шлюза — текущее долгосрочное состояние $\mathbf{c}_{(t)}$. Чтобы реализовать смотровые связи в TensorFlow, потребуется применить класс `LSTMCell` вместо `BasicLSTMCell` и установить `use_peepholes=True`:

```
lstm_cell = tf.contrib.rnn.LSTMCell(num_units=n_neurons,
                                    use_peepholes=True)
```

Существует много других вариантов ячейки LSTM. Особенно популярной среди вариантов является ячейка GRU, которую мы сейчас рассмотрим.

Ячейка GRU

Ячейка *управляемого рекуррентного блока* (*Gated Recurrent Unit — GRU*), показанная на рис. 14.14, предложена Кьенгьеном Чо и др. в работе 2014 года⁷, где также была представлена упомянутая ранее сеть “кодировщик–декодировщик”.

⁶ “Recurrent Nets that Time and Count” (“Рекуррентные сети, которые распределяют время и подсчитывают”), Ф. Герс и Ю. Шмидгубер (2000 год) (<https://goo.gl/ch8xz3>).

⁷ “Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation” (“Изучение представлений фраз с использованием рекуррентной нейронной сети ‘кодировщик–декодировщик’ для статистического машинного перевода”), К. Чо и др. (2014 год) (<https://goo.gl/ZnAE0Z>).

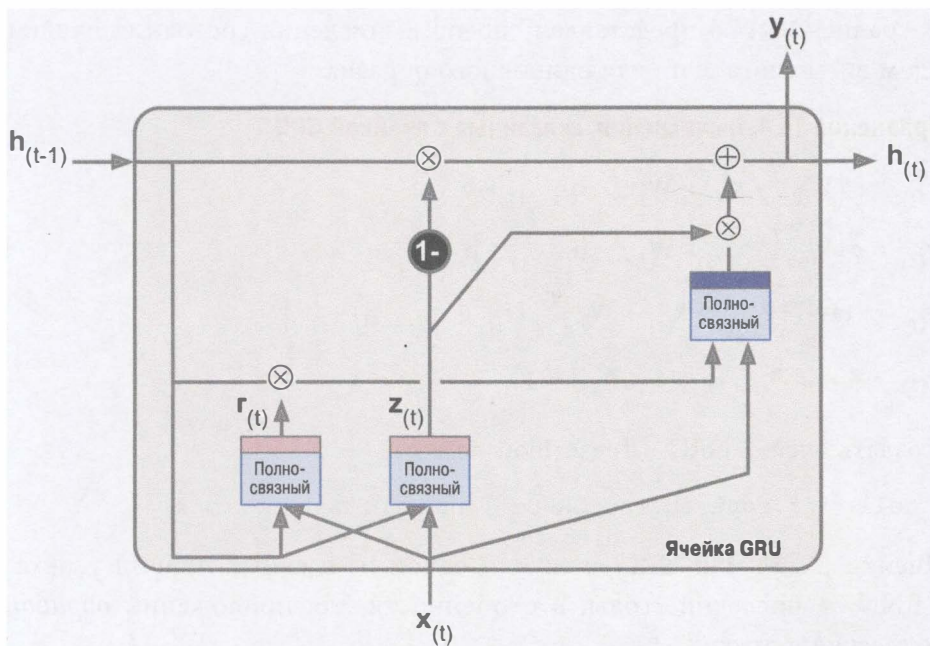


Рис. 14.14. Ячейка GRU

Ячейка GRU — это упрощенная версия ячейки LSTM, которая, по-видимому, работает в равной мере хорошо⁸ (чем и объясняется ее растущая популярность). Ниже перечислены главные упрощения.

- Два вектора состояния объединены в единственный вектор $\mathbf{h}_{(t)}$.
- Один контроллер шлюза управляет шлюзом забывания и входным шлюзом. Если контроллер шлюза выдает 1, то шлюз забывания открывается, а входной шлюз закрывается. Если контроллер шлюза выдает 0, тогда происходит противоположное. Другими словами, всякий раз, когда должно сохраняться воспоминание, сначала очищается место, куда оно будет сохранено. На самом деле это частый вариант и для самой ячейки LSTM.
- Выходной шлюз отсутствует; на каждом временном шаге выдается полный вектор состояния. Тем не менее, имеется новый контроллер шлюза, который управляет тем, какие части предыдущего состояния будут показаны главному слою.

⁸ В работе Клауса Греффа и др. “LSTM: A Search Space Odyssey” (“LSTM: скитания пространства поиска”), опубликованной в 2015 году (<https://goo.gl/hZB4KW>), показано, что все варианты LSTM работают примерно одинаково.

В уравнении 14.4 представлен способ вычисления состояния ячейки на каждом временном шаге для одиночного образца.

Уравнение 14.4. Вычисления, связанные с ячейкой GRU

$$\begin{aligned} \mathbf{z}_{(t)} &= \sigma(\mathbf{W}_{xz}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_z) \\ \mathbf{r}_{(t)} &= \sigma(\mathbf{W}_{xr}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \cdot \mathbf{h}_{(t-1)} + \mathbf{b}_r) \\ \mathbf{g}_{(t)} &= \tanh(\mathbf{W}_{xg}^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \cdot (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)} \end{aligned}$$

Создать ячейку GRU в TensorFlow легко:

```
gru_cell = tf.contrib.rnn.GRUCell(num_units=n_neurons)
```

Ячейки LSTM или GRU являются одной из главных причин успеха сетей RNN за последние годы, в особенности для приложений *обработки естественного языка (NLP)*.

Обработка естественного языка

Большинство современных приложений NLP, таких как машинный перевод, автоматическое резюмирование, грамматический разбор, смысловой анализ и многие другие, теперь основаны (по крайней мере, частично) на сетях RNN. В последнем разделе главы мы бегло взглянем на модель машинного перевода. Данная тема очень хорошо раскрыта в потрясающих руководствах Word2Vec (<https://goo.gl/edArdi>) и Seq2Seq (<https://goo.gl/L82gvS>) из документации по TensorFlow, поэтому вы определенно должны посмотреть их.

Векторные представления слов

Прежде чем начать, нам необходимо выбрать представление для слов. Один из вариантов мог бы предусматривать представление каждого слова с применением вектора в унитарном коде. Пусть ваш словарь содержит 50 000 слов, тогда n -е слово можно представить в виде вектора с 50 000 измерений, заполненного нулями за исключением единицы в позиции n . Однако с настолько крупным словарем подобное разреженное представление вообще неэффективно. В идеале вы хотите, чтобы похожие слова имели похожие пред-

ставления, облегчая модели обобщение своих знаний о слове на все похожие слова. Например, если модели было сообщено, что “I drink milk” (“Я пью молоко”) — допустимое предложение, и ей известно, что слово “milk” (“молоко”) близко к слову “water” (“вода”), но далеко от слова “shoes” (“туфли”), тогда модель будет знать, что предложение “I drink water” (“Я пью воду”), возможно, также является допустимым предложением, но “I drink shoes” (“Я пью туфли”) — вероятно нет. Но как придумать содержательное представление такого рода?

Самое распространенное решение состоит в том, чтобы представлять каждое слово в словаре с использованием довольно небольшого и плотного вектора (скажем, со 150 измерениями), называемого *векторным представлением* (*embedding*), и просто позволить нейронной сети во время обучения узнать хорошее представление для каждого слова. В начале обучения векторные представления выбираются случайным образом, но в процессе обучения обратное распространение автоматически перемещает векторные представления так, чтобы помочь нейронной сети выполнять свою задачу. Обычно это означает, что похожие слова будут понемногу группироваться ближе друг к другу и даже в итоге организуются довольно содержательным образом. Например, векторные представления могут разместиться вдоль различных осей, которые соответствуют роду, единственному/множественному числу, прилагательному/существительному и т.д. Результат может оказаться по-настоящему удивительным⁹.

В TensorFlow сначала понадобится создать переменную для векторного представления каждого слова из словаря (инициализированную случайным образом):

```
vocabulary_size = 50000
embedding_size = 150
init_embeddings =
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0)
embeddings = tf.Variable(init_embeddings)
```

Теперь предположим, что вы хотите передать своей нейронной сети предложение “I drink milk”. Вы должны сначала предварительно обработать предложение и разбить его на список известных слов. Например, вы можете уда-

⁹ Дополнительную информацию ищите в великолепной записи блога Кристофера Ола (<https://goo.gl/5rLNTj>) и в серии записей Себастьяна Рудера (<https://goo.gl/ojJjiE>).

лить ненужные символы, заменить неизвестные слова заранее определенным маркером вроде “[UNK]”, заменить числовые значения маркером “[NUM]”, заменить URL-адреса маркером “[URL]” и т.д. После получения списка известных слов вы можете найти в словаре целочисленные идентификаторы для каждого слова (от 0 до 49999), скажем, [72, 3335, 288]. В этот момент вы готовы передать такие идентификаторы слов библиотеке TensorFlow с применением заполнителя и воспользоваться функцией `embedding_lookup()` для получения соответствующих векторных представлений:

```
# из идентификаторов...
train_inputs = tf.placeholder(tf.int32, shape=[None])

# ...в векторные представления
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

Как только ваша модель узнает хорошие векторные представления слов, они фактически могут довольно эффективно применяться в любом приложении NLP: в конце концов, независимо от приложения слово “milk” по-прежнему близко к слову “water” и далеко от слова “shoes”.

На самом деле вместо обучения с целью получения собственных векторных представлений слов вы можете загрузить готовые векторные представления слов. Подобно тому, как делалось при повторном использовании заранее обученных слоев (см. главу 11), вы можете заморозить заранее обученные векторные представления (например, создавая переменную `embeddings` с `trainable=False`) или позволить обратному распространению подстраивать их для вашего приложения. Первый вариант ускорит обучение, но второй может привести к чуть более высокой производительности.



Векторные представления также удобны для категориальных атрибутов, которые могут принимать большое количество разных значений, особенно когда между значениями имеются запутанные сходства. В качестве примера подумайте о профессиях, увлечениях, блюдах, видах, брендах и т.п.

Теперь вы располагаете почти всеми инструментами, необходимыми для реализации системы машинного перевода. Итак, приступим.

Сеть “кодировщик–декодировщик” для машинного перевода

Давайте рассмотрим простую модель машинного перевода¹⁰, которая будет переводить английские предложения на французский язык (рис. 14.15).

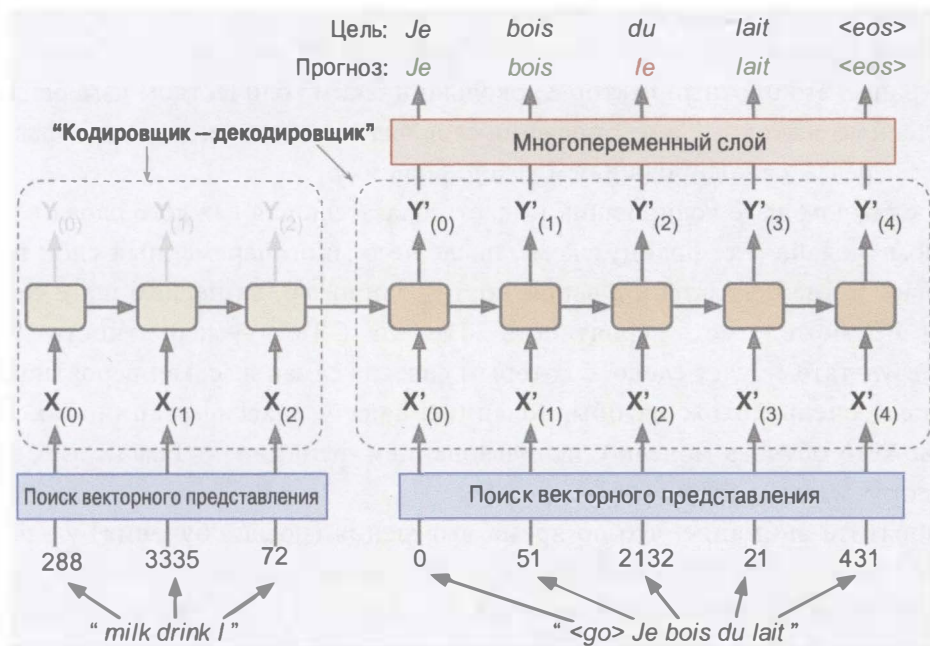


Рис. 14.15. Простая модель машинного перевода

Английское предложение передается кодировщику, а декодировщик выдаст его перевод на французский язык. Обратите внимание, что французский перевод также применяется как вход в декодировщик, но отодвинут на один шаг. Другими словами, декодировщику в качестве входа предоставляется слово, которое он *должен* был выдать на предыдущем шаге (независимо от того, что он выдал в действительности). Для самого первого слова ему дается маркер, представляющий начало предложения (скажем, "<go>"). Декодировщик ожидает, что предложение завершится маркером конца последовательности (EOS), таким как "<eos>".

Как видите, перед передачей кодировщику английские предложения выворачиваются наизнанку. Например, "I drink milk" превращается в "milk drink I".

¹⁰ "Sequence to Sequence learning with Neural Networks" ("Обучение типа 'последовательность в последовательность' с помощью нейронных сетей"), И. Суцкевер и др. (2014 год) (<https://goo.gl/0g9zWP>).

Такое действие гарантирует, что начало английского предложения будет передано кодировщику последним, и это удобно, т.к. обычно декодировщику приходится переводить его первым.

В начальной стадии каждое слово представлено простым целочисленным идентификатором (скажем, 288 для слова “milk”). Затем поиск векторного представления возвращает векторное представление слова (как объяснялось ранее, это плотный вектор с довольно низким количеством измерений). Найденные векторные представления слов являются именно тем, что фактически передается кодировщику и декодировщику.

На каждом шаге кодировщик выдает показатель для каждого слова в выходном словаре (т.е. французском), после чего многопеременный слой превращает такие показатели в вероятности. Например, на первом шаге слово “Je” (“Я”) может иметь вероятность 20%, “Tu” (“Ты”) — вероятность 1% и т.д. Результатом будет слово, с которым связана самая высокая вероятность. Процесс очень похож на обыкновенную задачу классификации, так что вы можете обучать модель с использованием функции `softmax_cross_entropy_with_logits()`.

Обратите внимание, что во время вывода (после обучения) у вас не будет целевого предложения для передачи декодировщику. Взамен просто передайте ему слово, выданное им на предыдущем шаге, как иллюстрируется на рис. 14.16 (это потребует поиска векторного представления, что на диаграмме не показано).

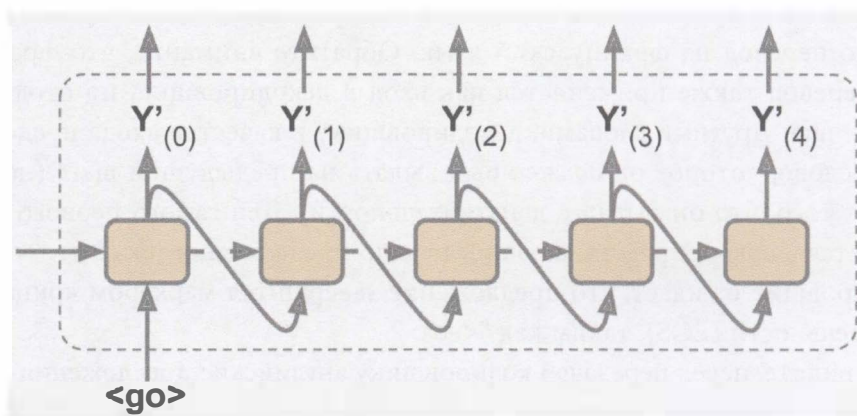


Рис. 14.16. Передача предыдущего выходного слова в качестве входа во время вывода

Итак, теперь у вас есть общая картина. Тем не менее, если вы просмотрите руководство Seq2Seq в документации по TensorFlow и ознакомитесь с кодом в `rnn/translate/seq2seq_model.py` (<https://github.com/tensorflow/models>), то заметите несколько важных отличий.

- Во-первых, до сих пор мы полагали, что все входные последовательности (передаваемые в кодировщик и декодировщик) имели постоянную длину. Но вполне очевидно длина предложений может варьироваться. Обработать это можно несколькими способами — скажем, передавая функциям `static_rnn()` или `dynamic_rnn()` аргумент `sequence_length` для указания длины каждого предложения (как обсуждалось ранее). Однако в руководстве применен другой подход (предположительно по причинам, связанным с производительностью): предложения объединяются в группы похожей длины (например, есть группа для предложений с числом слов от 1 до 6, еще одна группа для предложений с числом слов от 7 до 12 и т.д.¹¹), а более короткие предложения дополняются с использованием специального маркера дополнения (вроде “<pad>”). Скажем, предложение “I drink milk” превращается в “<pad> <pad> <pad> milk drink I”, а его переводом становится “Je bois du lait <eos> <pad>”. Разумеется, мы хотим игнорировать все, что выдается после маркера EOS. Для этого в реализации из руководства применяется вектор `target_weights`. Например, для целевого предложения “Je bois du lait <eos> <pad>” веса были бы установлены в `[1.0, 1.0, 1.0, 1.0, 1.0, 0.0]` (вес 0.0 соответствует маркеру дополнения в целевом предложении). Простое умножение потерь на целевые веса обнулит потери, которые соответствуют словам, находящимся за маркерами EOS.
- Во-вторых, в случае крупного выходного словаря (как здесь) выдача вероятности для каждого возможного слова была бы ужасающе медленной. Если целевой словарь содержит, скажем, 50 000 французских слов, тогда декодировщик должен выдавать векторы, имеющие 50 000 измерений, и последующее вычисление многопеременной функции на таком большом векторе потребует очень интенсивных вычислений. Чтобы избежать этого, можно позволить декодировщику выдавать гораздо меньшие векторы, такие как векторы с 1 000 измерений, и затем

¹¹ В руководстве используется другой размер группы.

использовать какой-то прием выборки для оценки потери, не проводя вычисления над каждым одиночным словом в целевом словаре. Такой *многопеременный прием с выборкой* (*sampled softmax technique*) был введен Себастьяном Жаном и др. в 2015 году¹². В TensorFlow можно применять функцию `sampled_softmax_loss()`.

- В-третьих, в реализации из руководства используется *механизм внимания* (*attention mechanism*), который позволяет декодирующему заглядывать внутрь входной последовательности. Рассмотрение дополненных вниманием сетей RNN (attention augmented RNN) выходит за рамки настоящей книги, но если вы заинтересовались, то можете ознакомиться с полезными статьями о машинном переводе¹³, машинном чтении¹⁴ и подписании изображений¹⁵ с применением внимания.
- В-четвертых, в реализации из руководства задействован модуль `tf.nn.legacy_seq2seq`, который предоставляет инструменты для легкого построения разнообразных моделей “кодировщик–декодировщик”. Например, функция `embedding_rnn_seq2seq()` создает простую модель “кодировщик–декодировщик”, которая автоматически позаботится о векторных представлениях слов, как и модель, приведенная на рис. 14.15. Вероятнее всего, код будет вскоре обновлен с целью использования нового модуля `tf.nn.seq2seq`.

Итак, у вас есть все инструменты, необходимые для понимания реализации сети RNN типа “последовательность в последовательность” из руководства. Опробуйте их и обучите собственный переводчик с английского языка на французский!

¹² “On Using Very Large Target Vocabulary for Neural Machine Translation” (“Об использовании очень крупного целевого словаря для нейронного машинного перевода”), С. Жан и др. (2015 год) (<https://goo.gl/u0GR8k>).

¹³ “Neural Machine Translation by Jointly Learning to Align and Translate” (“Нейронный машинный перевод путем обучения согласовывать и переводить совместно”), Д. Багданау и др. (2014 год) (<https://goo.gl/8RCous>).

¹⁴ “Long Short-Term Memory-Networks for Machine Reading” (“Сети с долгой краткосрочной памятью для машинного чтения”), Я. Ченг (2016 год) (<https://goo.gl/X0Nau8>).

¹⁵ “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention” (“Показать, уделить внимание и сообщить: нейронная генерация подписей для изображений с помощью зрительного внимания”), К. Суй и др. (2015 год) (<https://goo.gl/xmhvfK>).

Упражнения

1. Можете ли вы придумать несколько приложений для сети RNN типа “последовательность в последовательность”? Как насчет сети RNN типа “последовательность в вектор”? А сети RNN типа “вектор в последовательность”?
2. Почему для автоматического перевода люди применяют сети RNN типа “кодировщик–декодировщик”, а не простые сети RNN типа “последовательность в последовательность”?
3. Как вы могли бы объединить сверточную нейронную сеть с сетью RNN, чтобы классифицировать видеоролики?
4. Каковы преимущества построения сети RNN с использованием функции `dynamic_rnn()` вместо `static_rnn()`?
5. Как бы вы справились с входными последовательностями переменной длины? А что насчет выходных последовательностей переменной длины?
6. Каков распространенный способ распределения обучения и выполнения глубокой сети RNN между множеством ГП?
7. В своей работе, посвященной ячейкам LSTM, Хохрайтер и Шмидхубер применяли *вложенные грамматики Ребера* (*embedded Reber grammar*). Они представляют собой искусственные грамматики, которые производят строки наподобие “BPBTSXXVPSEPE”. Ознакомьтесь с хорошим введением в эту тему, написанным Дженни Опп (<https://goo.gl/7CkNRn>). Выберите отдельную вложенную грамматику Ребера (одну из приведенных на странице Дженни Опп) и обучите сеть RNN идентифицировать то, соблюдает ли строка правила выбранной грамматики. Сначала вам необходимо написать функцию, способную генерировать обучающий пакет с примерно 50% строк, которые соблюдают правила грамматики, и еще примерно 50% строк, которые правила не соблюдают.
8. Займитесь задачей о прогнозах почасовых осадков (“How much did it rain? II”; <https://goo.gl/0DS5Xe>) в состязаниях Kaggle. Это задача прогнозирования временных рядов: вам даются снимки значений из поляриметрических радаров, и предлагается выработать прогноз о почасовом показании дождемера. Луис Андре Дутра э Сильва в своем интервью (<https://goo.gl/fTA90W>) дает несколько интересных под-

сказок по приемам, которые он использовал для того, чтобы достичь второго места в состязании. В частности, он использовал сеть RNN, состоящую из двух слоев ячеек LSTM.

9. Проработайте руководство Word2Vec (<https://goo.gl/edArdi>) из документации по TensorFlow, чтобы создать векторные представления слов, и затем руководство Seq2Seq (<https://goo.gl/L82gvS>), чтобы обучить систему перевода с английского языка на французский.

Решения приведенных упражнений доступны в приложении А.

Автокодировщики

Автокодировщики — это искусственные нейронные сети, которые способны узнавать эффективные представления входных данных, называемые *кодировками* (*coding*), без какого-либо учителя (т.е. обучающий набор является непомеченным). Такие кодировки обычно имеют гораздо меньшую размерность, чем входные данные, делая автокодировщики удобным средством понижения размерности (см. главу 8). Более важно то, что автокодировщики действуют как мощные обнаружители признаков и могут использоваться для предварительного обучения без учителя глубоких нейронных сетей (что обсуждалось в главе 11). Наконец, они способны случайным образом генерировать новые данные, которые выглядят очень похожими на обучающие данные; это называется *порождающей моделью* (*generative model*). Например, вы могли бы обучить автокодировщик на фотографиях лиц, и он был бы в состоянии генерировать новые лица.

На удивление автокодировщики работают, просто обучаясь копировать свои входы в выходы. Это может показаться тривиальной задачей, но мы увидим, что ограничение сети разнообразными путями может сделать ее довольно трудной. Скажем, вы можете ограничить размер внутреннего представления или добавить к входам шум и учить сеть восстанавливать исходные входы. Такие ограничения не позволяют автокодировщику просто копировать входы прямо в выходы, вынуждая его обучаться эффективным способам представления данных. Короче говоря, кодировки являются побочными продуктами попытки автокодировщика узнать тождественную функцию под некоторыми ограничениями.

В настоящей главе объясняется работа автокодировщиков, типы накладываемых ограничений и их реализация с применением TensorFlow, будь то понижение размерности, выделение признаков, предварительное обучение без учителя или порождающие модели.

Эффективные представления данных

Какую из указанных ниже последовательностей вы сочтете более легкой для запоминания?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 25, 76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20

На первый взгляд может показаться, что первую последовательность должно быть легче запомнить, т.к. она намного короче. Однако если вы посмотрите на вторую последовательность более внимательно, то можете заметить, что в ней соблюдаются два простых правила: за четными числами следуют их половинные величины, а за нечетными — их утроенные величины плюс единица (это известная *последовательность чисел-градир* (*hailstone sequence*)). После выявления описанного шаблона вторая последовательность становится гораздо проще для запоминания, нежели первая, поскольку вам необходимо запомнить только два правила, первое число и длину последовательности. Обратите внимание, что если бы вы могли быстро и легко запоминать очень длинные последовательности, тогда существование какого-то шаблона во второй последовательности не было бы особенно важным. Вы всего лишь заучивали бы каждое число наизусть и ничего более. Именно этот факт объясняет трудность запоминания длинных последовательностей, что делает полезным распознавание шаблонов и вероятно проливает свет на то, почему ограничение автокодировщика во время обучения заставляет его выявлять и эксплуатировать шаблоны в данных.

Взаимоотношение между памятью, восприятием и сопоставлением с образцами было превосходно изучено Уильямом Чейзом и Гербертом Симоном в начале 1970-х годов¹. Они заметили, что опытные шахматисты способны запомнить позиции всех фигур в игре, просто глядя на доску в течение каких-то 5 секунд — задача, справиться с которой большинство людей сочло бы невозможным. Тем не менее, так происходило только в случае, когда фигуры находились в реалистичных позициях (из реальных игр), но не при их размещении произвольным образом. Знатоки шахмат не обладают лучшей памятью, чем вы и я, они лишь гораздо легче замечают шахматные шаблоны благодаря своему опыту игры. Выявление шаблонов помогает им эффективно запоминать информацию.

¹ “Perception in chess” (“Восприятие в шахматах”), У. Чейз и Г. Симон (1973 год) (<https://goo.gl/kSNcX0>).

Подобно шахматистам из упомянутого эксперимента с запоминанием автокодировщик просматривает входы, преобразует их в эффективное внутреннее представление и выдает то, что (надо надеяться) выглядит очень близким к входам. Автокодировщик всегда состоит из двух частей: кодировщика (или *распознающей сети* (*recognition network*)), который преобразует входы во внутреннее представление, и следующего за ним декодировщика (или *порождающей сети* (*generative network*)), который преобразует внутреннее представление в выходы (рис. 15.1).

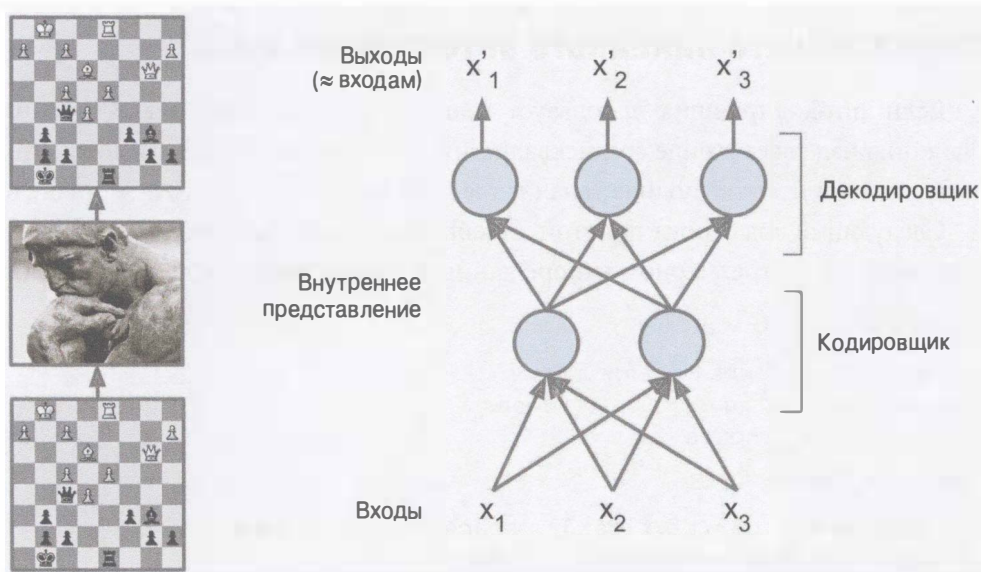


Рис. 15.1. Эксперимент с запоминанием позиций шахматных фигур (слева) и простой автокодировщик (справа)

Как видите, автокодировщик обычно имеет ту же самую архитектуру, что и многослойный персептрон (см. главу 10), но только количество нейронов в выходном слое должно быть равно количеству входов. В приведенном примере есть лишь один скрытый слой, состоящий из двух нейронов (кодировщик), и один выходной слой, включающий три нейрона (декодировщик). Выходы часто называют *реконструкциями* (*reconstruction*), т.к. автокодировщик пытается реконструировать входы, а функция издержек содержит *потерю из-за реконструкции* (*reconstruction loss*), которая штрафует модель, когда реконструкции отличаются от входов.

Поскольку внутреннее представление имеет меньшую размерность, чем входные данные (оно двумерное, а не трехмерное), говорят, что автокоди-

ровщик является *понижающим* (*undercomplete*). Понижающий автокодировщик не может просто скопировать свои входы в кодировки, он обязан еще и отыскать способ выдачи копии своих входов. Понижающий автокодировщик вынужден узнать самые важные признаки во входных данных (и отбросить несущественные признаки).

Давайте посмотрим, как реализовать очень простой понижающий автокодировщик для уменьшения размерности.

Выполнение анализа главных компонент с помощью понижающего линейного автокодировщика

Если автокодировщик использует только линейные функции активации и функцию издержек в виде среднеквадратической ошибки (MSE), то можно показать, что он в итоге выполняет анализ главных компонент (PCA; см. главу 8).

Следующий код строит простой линейный автокодировщик для выполнения анализа PCA на трехмерном наборе данных, проецируя его в два измерения:

```
import tensorflow as tf

n_inputs = 3 # трехмерные входы
n_hidden = 2 # двумерные кодировки
n_outputs = n_inputs

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden)
outputs = tf.layers.dense(hidden, n_outputs)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(reconstruction_loss)

init = tf.global_variables_initializer()
```

Код на самом деле не очень сильно отличается от всех многослойных персептронов, которые мы строили в предшествующих главах. Отметим два основных момента.

- Количество выходов равно количеству входов.
- Для выполнения простого анализа PCA мы не применяем какую-то функцию активации (т.е. все нейроны линейны), а функция издержек основана на MSE. Вскоре мы рассмотрим более сложные автокодировщики.

Теперь загрузим набор данных, обучим модель на обучающем наборе и используем ее для кодирования испытательного набора (т.е. его проецирования в два измерения):

```
X_train, X_test = [...] # загрузка набора данных
n_iterations = 1000
codings = hidden # выход скрытого слоя предоставляет кодировки

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        # метки отсутствуют (без учителя)
        training_op.run(feed_dict={X: X_train})
        codings_val = codings.eval(feed_dict={X: X_test})
```

Слева на рис. 15.2 показан исходный трехмерный набор данных, а справа — выход скрытого слоя автокодировщика (т.е. кодирующего слоя). Как видите, автокодировщик нашел наилучшую двумерную плоскость для проецирования на нее данных, предохраняя как можно больше дисперсии (подобно PCA).

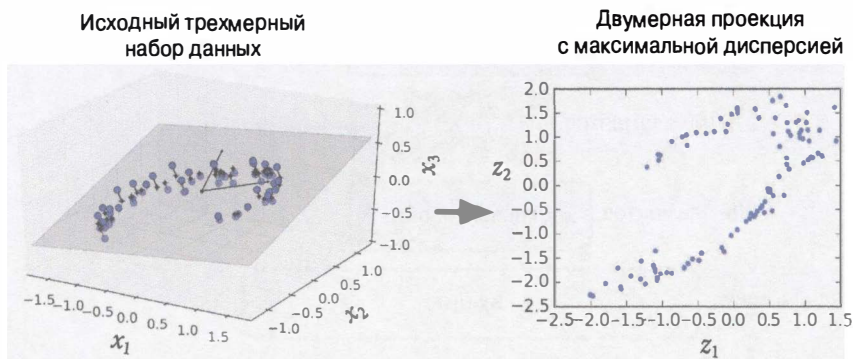


Рис. 15.2. Анализ PCA, выполненный понижающим линейным автокодировщиком

Многослойные автокодировщики

Подобно другим нейронным сетям, которые мы обсуждали, автокодировщики могут иметь множество скрытых слоев. В таком случае они называются *многослойными автокодировщиками* (*stacked autoencoder*) или *глубокими автокодировщиками* (*deep autoencoder*). Добавление дополнительных слоев помогает автокодировщику узнавать более сложные кодировки. Однако

нужно соблюдать осторожность, чтобы не сделать автокодировщик чересчур мощным. Представьте себе кодировщик, который является настолько мощным, что он просто научится сопоставлять каждый вход с единственным произвольным числом (а декодировщик научится обратному сопоставлению). Очевидно, такой автокодировщик будет идеально реконструировать обучающие данные, но в процессе не узнает ни одного полезного представления данных (и вряд ли хорошо обобщится на новые образцы).

Архитектура многослойного автокодировщика обычно симметрична относительно центрального скрытого слоя (кодирующего слоя). Попросту говоря, она похожа на бутерброд. Например, автокодировщик для MNIST (введенный в главе 3) может иметь 784 входа, за которыми следует скрытый слой из 300 нейронов, центральный скрытый слой из 150 нейронов, еще один скрытый слой из 300 нейронов и выходной слой из 784 нейронов. Такой многослойный автокодировщик показан на рис. 15.3.

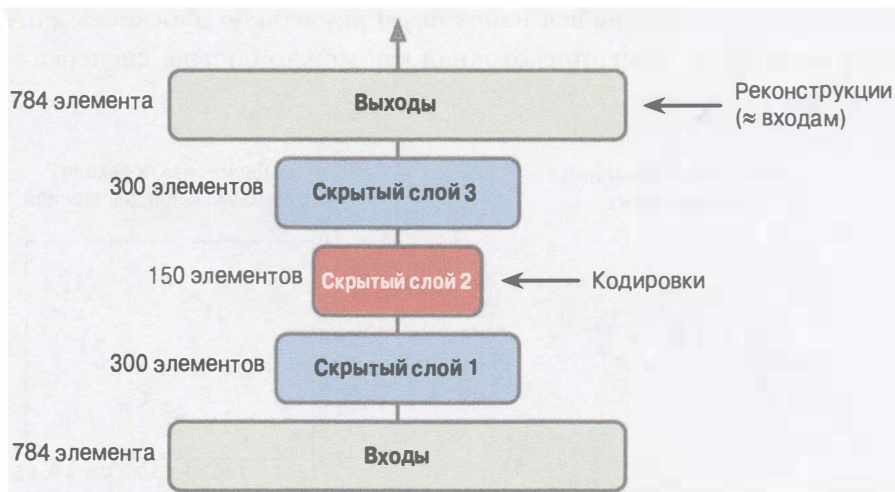


Рис. 15.3. Многослойный автокодировщик

Реализация с помощью TensorFlow

Многослойный автокодировщик можно реализовать способом, очень похожим на реализацию обыкновенного глубокого многослойного персептрона. В частности, могут применяться те же самые приемы, которые использовались для обучения глубоких сетей в главе 11. Скажем, приведенный ниже код строит многослойный автокодировщик для MNIST с применением инициализации Хе, функции активации ELU и регуляризации ℓ_2 . Код должен выглядеть знакомым, но только отсутствуют метки (нет y):

```

from functools import partial

n_inputs = 28 * 28 # для MNIST
n_hidden1 = 300
n_hidden2 = 150 # кодировки
n_hidden3 = n_hidden1
n_outputs = n_inputs

learning_rate = 0.01
l2_reg = 0.0001

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

he_init = tf.contrib.layers.variance_scaling_initializer()
l2_regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
my_dense_layer = partial(tf.layers.dense,
                          activation=tf.nn.elu,
                          kernel_initializer=he_init,
                          kernel_regularizer=l2_regularizer)

hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2) # кодировки
hidden3 = my_dense_layer(hidden2, n_hidden3)
outputs = my_dense_layer(hidden3, n_outputs, activation=None)

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([reconstruction_loss] + reg_losses)

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()

```

Затем модель можно обучить обычным образом. Обратите внимание, что метки цифр (`y_batch`) не используются:

```

n_epochs = 5
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

```


Когда автокодировщик четко симметричен, как только что построенный, распространенный прием предусматривает связывание весов декодирующих слоев с весами кодирующих слоев. В результате наполовину уменьшается количество весов в модели, ускоряя обучение и ограничивая риск переобучения. В частности, если автокодировщик имеет всего N слоев (не считая входного), и W_L представляет веса связей слоя L (например, слой 1 — первый скрытый слой, слой $\frac{N}{2}$ — кодирующий слой, а слой N — выходной слой), тогда веса декодирующего слоя могут быть определены так: $W_{N-L+1} = W_L^T$ (с $L = 1, 2, \dots, \frac{N}{2}$).

К сожалению, реализация связанных весов в TensorFlow с применением функции `dense()` несколько громоздка; на самом деле легче определять слои вручную. Код становится значительно многословнее:

```
activation = tf.nn.elu
regularizer = tf.contrib.layers.l2_regularizer(l2_reg)
initializer = tf.contrib.layers.variance_scaling_initializer()

X = tf.placeholder(tf.float32, shape=[None, n_inputs])

weights1_init = initializer([n_inputs, n_hidden1])
weights2_init = initializer([n_hidden1, n_hidden2])

weights1 = tf.Variable(weights1_init, dtype=tf.float32, name="weights1")
weights2 = tf.Variable(weights2_init, dtype=tf.float32, name="weights2")
weights3 = tf.transpose(weights2, name="weights3") # связанные веса
weights4 = tf.transpose(weights1, name="weights4") # связанные веса

biases1 = tf.Variable(tf.zeros(n_hidden1), name="biases1")
biases2 = tf.Variable(tf.zeros(n_hidden2), name="biases2")
biases3 = tf.Variable(tf.zeros(n_hidden3), name="biases3")
biases4 = tf.Variable(tf.zeros(n_outputs), name="biases4")

hidden1 = activation(tf.matmul(X, weights1) + biases1)
hidden2 = activation(tf.matmul(hidden1, weights2) + biases2)
hidden3 = activation(tf.matmul(hidden2, weights3) + biases3)
outputs = tf.matmul(hidden3, weights4) + biases4

reconstruction_loss = tf.reduce_mean(tf.square(outputs - X))
reg_loss = regularizer(weights1) + regularizer(weights2)
loss = reconstruction_loss + reg_loss

optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
```

Код довольно прямолинеен, но необходимо отметить несколько важных моментов.

- Во-первых, `weight3` и `weights4` не являются переменными, а представляют собой транспонированные `weights2` и `weights1` соответственно (они “связываются” с ними).
- Во-вторых, поскольку `weight3` и `weights4` — не переменные, они не требуют регуляризации: мы регуляризируем только `weights1` и `weights2`.
- В-третьих, смещения никогда не связываются и никогда не регуляризуются.

Обучение по одному автокодировщику за раз

Вместо обучения сразу целого многослойного автокодировщика, как мы только что делали, часто намного быстрее обучать по одному неглубокому автокодировщику за раз, после чего уложить всех их в один многослойный автокодировщик, как показано на рис. 15.4. Поступать так особенно удобно в случае очень глубоких автокодировщиков.

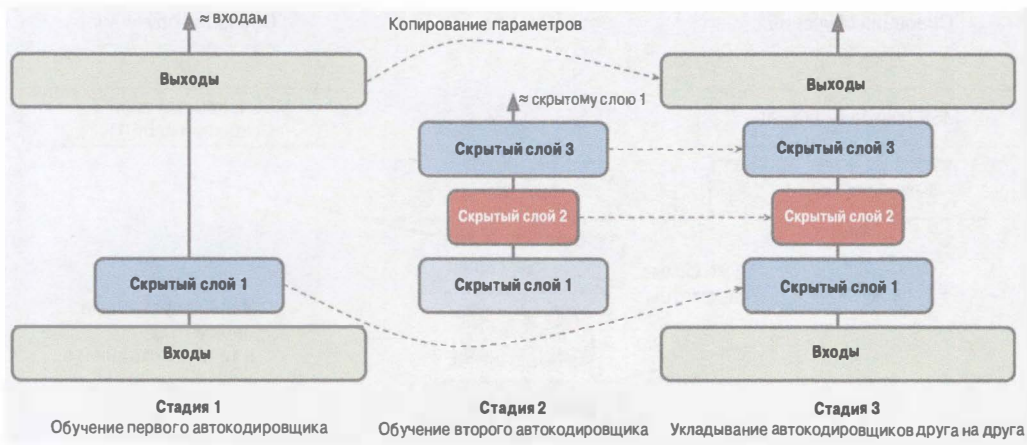


Рис. 15.4. Обучение по одному автокодировщику за раз

В течение первой стадии обучения первый автокодировщик учится реконструировать входы. Во время второй стадии второй автокодировщик учится реконструировать выход скрытого слоя первого автокодировщика. Наконец, используя все имеющиеся автокодировщики, вы формируете большой бутерброд, как продемонстрировано на рис. 15.4 (т.е. сначала уклады-

ваете скрытые слои каждого автокодировщика, а затем выходные слои в обратном порядке). В итоге получается финальный многослойный автокодировщик. Подобным образом вы могли бы легко обучить больше автокодировщиков, формируя очень глубокий многослойный автокодировщик.

Чтобы реализовать такой многостадийный алгоритм обучения, проще всего применять для каждой стадии разный граф TensorFlow. После обучения автокодировщика вы просто прогоняете через него обучающий набор и захватываете выход каждого скрытого слоя. Затем этот выход служит обучающим набором для следующего автокодировщика. Как только все автокодировщики будут обучены, вы копируете веса и смещения из каждого автокодировщика и используете их при построении многослойного автокодировщика. Реализация этого подхода достаточно прямолинейна, так что подробности здесь не приводятся, но просмотрите код в тетрадях Jupyter (<https://github.com/ageron/handson-ml>) для данного примера.

Другой подход предусматривает применение единственного графа, содержащего полный многослойный автокодировщик, а также ряд добавочных операций для выполнения каждой стадии обучения (рис. 15.5).

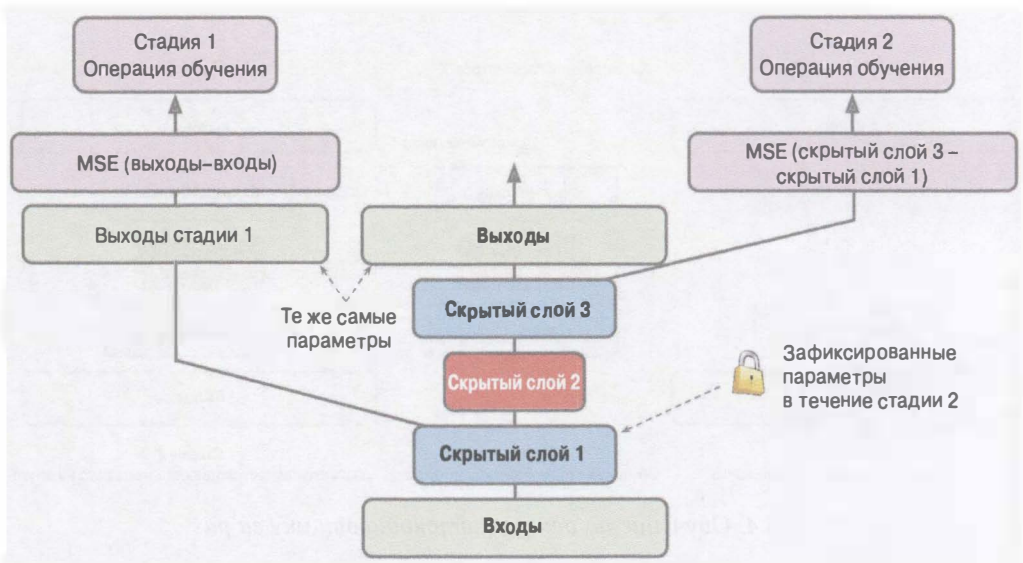


Рис. 15.5. Единственный граф для обучения многослойного автокодировщика

Подход заслуживает некоторых пояснений.

- Центральная колонка в графе представляет собой многослойный автокодировщик. Эта часть может использоваться после обучения.

- Левая колонка — набор операций, необходимых для выполнения первой стадии обучения. Он создает выходной слой, который обходит скрытые слои 2 и 3. Этот выходной слой разделяет те же самые веса и смещения, что и выходной слой многослойного автокодировщика. Следом идут операции обучения, которые будут нацелены на то, чтобы сделать выход как можно более близким к входам. Таким образом, данная стадия подготовит веса и смещения для скрытого слоя 1 и выходного слоя (т.е. первого автокодировщика).
- Правая колонка — набор операций, требующихся для выполнения второй стадии обучения. Он добавляет операцию, которая будет нацелена на то, чтобы сделать выход скрытого слоя 3 как можно более близким к выходу скрытого слоя 1. Обратите внимание, что на время прохождения стадии 2 мы должны заморозить скрытый слой 1. На этой стадии будут подготовлены веса и смещения для скрытых слоев 2 и 3 (т.е. второго автокодировщика).

Код TensorFlow выглядит следующим образом:

```
[...] # Построение целого многослойного автокодировщика
      # обычным образом. В этом примере веса не связываются.

optimizer = tf.train.AdamOptimizer(learning_rate)

with tf.name_scope("phase1"):
    phase1_outputs = tf.matmul(hidden1, weights4) + biases4
    phase1_reconstruction_loss =
        tf.reduce_mean(tf.square(phase1_outputs - X))
    phase1_reg_loss = regularizer(weights1) + regularizer(weights4)
    phase1_loss = phase1_reconstruction_loss + phase1_reg_loss
    phase1_training_op = optimizer.minimize(phase1_loss)

with tf.name_scope("phase2"):
    phase2_reconstruction_loss =
        tf.reduce_mean(tf.square(hidden3 - hidden1))
    phase2_reg_loss = regularizer(weights2) + regularizer(weights3)
    phase2_loss = phase2_reconstruction_loss + phase2_reg_loss
    train_vars = [weights2, biases2, weights3, biases3]
    phase2_training_op =
        optimizer.minimize(phase2_loss, var_list=train_vars)
```

Первая стадия довольно прямолинейна: мы просто создаем выходной слой, который обходит скрытые слои 2 и 3, а затем строим операции обучения для сведения к минимуму расстояния между выходами и входами (плюс определенная доля регуляризации).

Вторая стадия лишь добавляет операции, необходимые для сведения к минимуму расстояния между выходом скрытого слоя 3 и скрытым слоем 1 (также с некоторой регуляризацией). Что важнее, мы предоставляем методу `minimize()` список обучаемых переменных, не включая в него `weights1` и `biases1`; это эффективно замораживает скрытый слой 1 в течение стадии 2.

Во время стадии выполнения понадобится лишь запустить операцию обучения стадии 1 для некоторого числа эпох и затем операцию обучения стадии 2 для дополнительных эпох.



Поскольку скрытый слой 1 на стадии 2 замораживается, его выход будет всегда одинаковым для любого обучающего образца. Во избежание повторного вычисления выхода скрытого слоя 1 на каждой отдельной эпохе вы можете вычислить его для всего обучающего набора в конце стадии 1, после чего напрямую передавать кешированный выход скрытого слоя 1 во время стадии 2. Такой прием может дать неплохой прирост производительности.

Визуализация реконструкций

Удостовериться в том, что автокодировщик надлежащим образом обучен, можно путем сравнения входов и выходов. Они обязаны быть довольно похожими, а отличия должны касаться лишь неважных деталей. Давайте вычертим две случайные цифры и их реконструкции:

```
n_test_digits = 2
X_test = mnist.test.images[:n_test_digits]

with tf.Session() as sess:
    [...] # Обучение автокодировщика
    outputs_val = outputs.eval(feed_dict={X: X_test})

def plot_image(image, shape=[28, 28]):
    plt.imshow(image.reshape(shape), cmap="Greys",
               interpolation="nearest")
    plt.axis("off")

for digit_index in range(n_test_digits):
    plt.subplot(n_test_digits, 2, digit_index * 2 + 1)
    plot_image(X_test[digit_index])
    plt.subplot(n_test_digits, 2, digit_index * 2 + 2)
    plot_image(outputs_val[digit_index])
```

Результирующие изображения показаны на рис. 15.6.

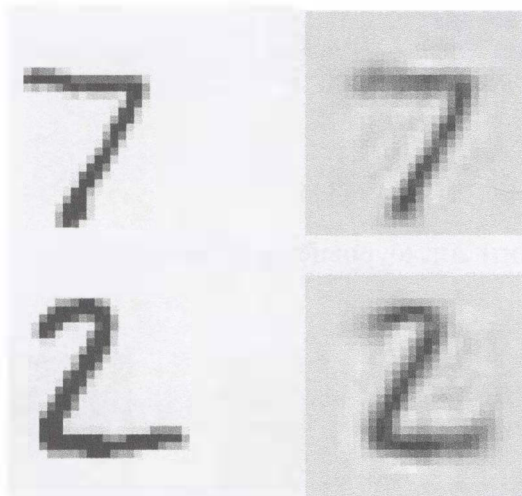


Рис. 15.6. Исходные цифры (слева) и их реконструкции (справа)

Выглядят достаточно близкими. Значит, автокодировщик надлежащим образом научился воспроизводить свои входы, но узнал ли он полезные признаки? Давайте выясним.

Визуализация признаков

После того как автокодировщик выучил определенные признаки, у вас может возникнуть желание взглянуть на них. Это можно сделать с помощью разнообразных приемов. Вероятно, самый простой прием заключается в том, чтобы просмотреть каждый нейрон в каждом скрытом слое и найти обучающие образцы, которые активируют его больше всего. Поступать подобным образом особенно полезно для верхних скрытых слоев, т.к. они часто захватывают относительно крупные признаки, легко обнаруживаемые в группе обучающих образцов, которые их содержат. Например, если нейрон интенсивно активируется, когда он видит кошку на фотографии, то будет вполне очевидным, что большинство фотографий, которые его активировали, содержали кошек. Тем не менее, для нижних слоев такой прием работает не настолько хорошо, поскольку признаки меньше и абстрактнее, поэтому зачастую трудно понять, что именно возбудило нейрон.

Рассмотрим другой прием. Для каждого нейрона в первом скрытом слое вы можете создать изображение, в котором яркость пикселя соответствует весу связи с заданным нейроном. Например, следующий код вычерчивает признаки, найденные пятью нейронами из первого скрытого слоя:


```
with tf.Session() as sess:
    [...] # обучение автокодировщика
    weights1_val = weights1.eval()

for i in range(5):
    plt.subplot(1, 5, i + 1)
    plot_image(weights1_val.T[i])
```

Вы можете получить низкоуровневые признаки вроде представленных на рис. 15.7.

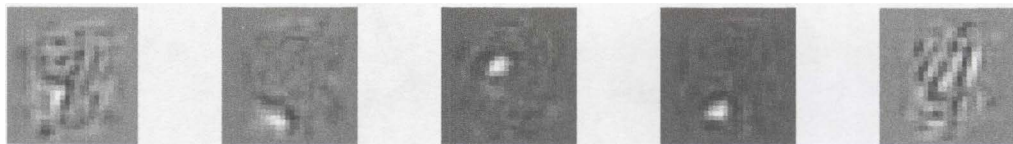


Рис. 15.7. Признаки, найденные пятью нейронами из первого скрытого слоя

Первые четыре признака, похоже, соответствуют небольшим пятнам, в то время как пятый признак, кажется, ищет вертикальные штрихи (обратите внимание, что эти признаки поступают из многослойного автокодировщика, устраняющего шум, который мы обсудим позже).

Еще один прием предусматривает передачу автокодировщику случайного входного изображения, измерение степени активации интересующего нейрона и выполнение обратного распространения так, чтобы этот нейрон активировался даже сильнее. Если вы повторите указанные действия несколько раз (делая градиентный подъем (*gradient ascent*)), тогда изображение будет постепенно превращаться в самое возбуждающее изображение (для нейрона). Такой прием удобен для визуализации разновидностей входов, которые ищет нейрон.

Наконец, если вы применяете автокодировщик для проведения предварительного обучения без учителя (скажем, для задачи классификации), то простой способ проверки, что найденные автокодировщиком признаки полезны, заключается в измерении производительности классификатора.

Предварительное обучение без учителя с использованием многослойных автокодировщиков

Как обсуждалось в главе 11, если вы занимаетесь сложной задачей обучения с учителем, но не располагаете большим объемом помеченных обучающих

данных, то решением может быть нахождение нейронной сети, рассчитанной на похожую задачу, и повторное использование ее самых нижних слоев. Это позволяет обучать высокопроизводительную модель с применением только малого объема обучающих данных, поскольку ваша нейронная сеть не нуждается в изучении всех низкоуровневых признаков; она просто повторно использует обнаружители признаков, обученные существующей сетью.

Аналогично, если вы имеете крупный набор данных, но большинство из них не помечено, тогда можете сначала обучить многослойный автокодировщик с применением всех данных, затем повторно использовать самые нижние слои для создания нейронной сети, ориентированной на фактическую задачу, и обучить ее с применением помеченных данных.

На рис. 15.8 показано, как с использованием многослойного автокодировщика провести предварительное обучение без учителя для классификационной нейронной сети. Как обсуждалось ранее, сам многослойный автокодировщик обычно обучает по одному автокодировщику за раз. Когда обучается классификатор, а помеченных обучающих данных не особенно много, может возникнуть желание заморозить заранее обученные слои (во всяком случае, самые нижние из них).



Рис. 15.8. Предварительное обучение без учителя с использованием автокодировщиков



В действительности такая ситуация довольно распространена, потому что построение крупных непомеченных наборов данных часто обходится недорого (скажем, простой сценарий способен загрузить миллионы изображений из Интернета), но их надежная пометка может производиться только людьми (например, классификация изображений как привлекательных или нет). Пометка образцов является отнимающей много времени и дорогостоящей процедурой, поэтому нередко имеется только несколько тысяч помеченных образцов.

Как уже обсуждалось, одним из инициаторов текущего всплеска интереса к глубокому обучению стало открытие, что глубокие нейронные сети могут быть предварительно обучены в стиле без учителя, совершенное в 2006 году Джефффри Хинтоном и др. Для этого они применяли ограниченные машины Больцмана (см. приложение Д), но в 2007 году Йошуа Бенджи и др. показали², что в равной степени хорошо работают и автокодировщики.

В реализации TensorFlow нет ничего особенного: сначала с использованием всех обучающих данных производится обучение автокодировщика и затем его кодирующие слои применяются для создания новой нейронной сети (дополнительные сведения о повторном использовании заранее обученных слоев ищите в главе 11 или просмотрите примеры кода в тетрадях Jupyter).

До сих пор для того, чтобы заставить автокодировщик узнать интересные признаки, мы ограничивали размер его кодирующего слоя, делая автокодировщик понижающим. В действительности есть много других видов ограничений, которые можно применять, включая те, что позволяют кодирующему слою быть таким же большим, как входы, или даже больше, давая в результате *повышающий автокодировщик* (*overcomplete autoencoder*). Давайте рассмотрим некоторые подходы такого рода.

Шумоподавляющие автокодировщики

Еще один способ заставить автокодировщик выявлять интересные признаки предусматривает добавление шума к входам, обучая его восстанавливать исходные, свободные от шума входы. Такой прием удерживает автокодировщик от простого копирования своих входов в выходы, поэтому он вынужден искать шаблоны в данных.

² “Greedy Layer-Wise Training of Deep Networks” (“Жадное послойное обучение глубоких сетей”), Й. Бенджи и др. (2007 год) (<https://goo.gl/y2Kqin>).

Идея использования автокодировщиков для подавления шума существует с 1980-х годов (скажем, в 1987 году Ян Лекун упомянул о ней в своей кандидатской диссертации). В работе Паскаля Винсента и др., опубликованной в 2008 году³, было показано, что автокодировщики могли бы применяться также для выделения признаков. В работе Винсента и др., вышедшей в 2010 году⁴, были представлены *многослойные шумоподавляющие автокодировщики* (*stacked denoising autoencoder*).

Шум может быть чистым гауссовым шумом, добавленным к входам, или случайным выключением входов подобно отключению (введенному в главе 11). На рис. 15.9 приведены оба варианта.



Рис. 15.9. Шумоподавляющие автокодировщики с гауссовым шумом (слева) или отключением (справа)

Реализация с помощью TensorFlow

Реализовать шумоподавляющие автокодировщики в TensorFlow не слишком трудно. Давайте начнем с гауссова шума. На самом деле это похоже на обучение обыкновенного автокодировщика, но только к входам добавляется шум, а потеря из-за реконструкции подсчитывается на основе исходных входов:

³ “Extracting and Composing Robust Features with Denoising Autoencoders” (“Выделение и компоновка устойчивых признаков с помощью шумоподавляющих автокодировщиков”), П. Винсент и др. (2008 год) (<https://goo.gl/K9pqcx>).

⁴ “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion” (“Многослойные шумоподавляющие автокодировщики: обучение глубокой сети выявлению полезных представлений с помощью локального критерия устранения шумов”), П. Винсент и др. (2010 год) (<https://goo.gl/HgCDIA>).

```
noise_level = 1.0
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_noisy = X + noise_level * tf.random_normal(tf.shape(X))
hidden1 = tf.layers.dense(X_noisy, n_hidden1, activation=tf.nn.relu,
                           name="hidden1")

[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```



Поскольку на стадии построения форма `X` определяется только частично, мы не знаем заранее форму шума, который должен быть добавлен к `X`. Мы не можем вызывать `X.get_shape()`, потому что в результате возвратилась бы только частично определенная форма `X` (`[None, n_inputs]`), а функция `random_normal()` ожидает полностью определенной формы и потому сгенерирует исключение. Взамен мы вызываем функцию `tf.shape(X)`, создающую операцию, которая будет возвращать форму `X` во время выполнения, полностью определенную на тот момент.

Реализовать версию с отключением, которая более распространена, не намного труднее:

```
dropout_rate = 0.3
training=tf.placeholder_with_default(False, shape=(),name='training')
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
X_drop = tf.layers.dropout(X, dropout_rate, training=training)
hidden1 = tf.layers.dense(X_drop, n_hidden1, activation=tf.nn.relu,
                           name="hidden1")

[...]
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
[...]
```

Во время обучения мы обязаны установить `training` в `True` (как объяснялось в главе 11), используя `feed_dict`:

```
sess.run(training_op, feed_dict={X: X_batch, training: True})
```

Во время испытаний устанавливать `training` в `False` необязательно, т.к. мы делаем это по умолчанию в вызове функции `placeholder_with_default()`.

Разреженные автокодировщики

Другим видом ограничения, который часто приводит к хорошему выделению признаков, является *разреженность* (*sparsity*): за счет добавления подходящего члена к функции издержек автокодировщик принуждается к сокращению количества активных нейронов в кодирующем слое. Например, его можно заставить иметь в среднем лишь 5% заметно активных нейронов в кодирующем слое. Это заставит автокодировщик представлять каждый вход как комбинацию небольшого числа активаций. В результате каждый нейрон внутри кодирующего слоя обычно представляет полезный признак (если бы вы могли произносить лишь несколько слов в месяц, то наверняка постарались бы обеспечить, чтобы их слышали).

Для поддержки разреженных моделей нам придется сначала измерить фактическую разреженность кодирующего слоя на каждой итерации обучения. Мы делаем это путем вычисления средней активации каждого нейрона в кодирующем слое по целому обучающему пакету. Размер пакета не должен быть слишком маленьким, иначе среднее не будет точным.

Имея среднюю активацию на нейрон, мы хотим штрафовать чересчур активные нейроны, добавляя к функции издержек *потерю из-за разреженности* (*sparsity loss*). Скажем, если в результате измерений выяснилось, что нейрон имеет среднюю активацию 0.3, но целевая разреженность составляет 0.1, то он должен быть оштрафован, чтобы активироваться меньше. Один из подходов мог бы заключаться в добавлении к функции издержек квадратичной ошибки $(0.3 - 0.1)^2$, но на практике лучший подход предусматривает применение расстояния Кульбака–Лейблера (кратко упомянутого в главе 4), которое имеет намного более сильные градиенты, чем среднеквадратическая ошибка (рис. 15.10).

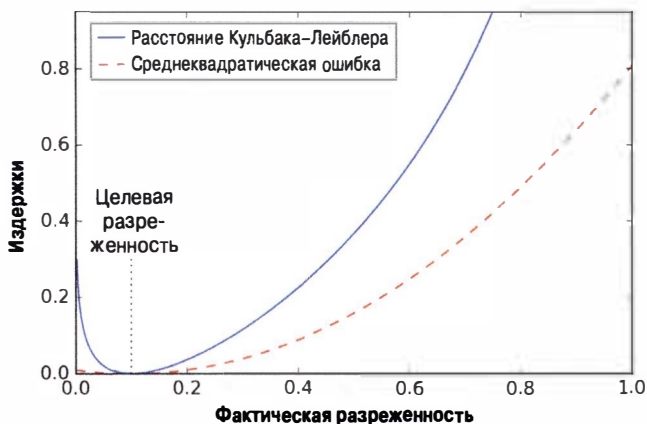


Рис. 15.10. Потеря из-за разреженности

Имея два дискретных распределения вероятности P и Q , расстояние Кульбака–Лейблера между этими распределениями, обозначаемое $D_{KL}(P \parallel Q)$, может быть вычислено с использованием уравнения 15.1.

Уравнение 15.1. Расстояние Кульбака–Лейблера

$$D_{KL}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

В данном случае нам нужно измерить расстояние между целевой вероятностью p , что нейрон в кодирующем слое будет активирован, и фактической вероятностью q (т.е. среднюю активацию по обучающему пакету). Таким образом, расстояние Кульбака–Лейблера упрощается до уравнения 15.2.

Уравнение 15.2. Расстояние Кульбака–Лейблера между целевой разреженностью p и фактической разреженностью q

$$D_{KL}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

Вычислив потерю из-за разреженности для каждого нейрона в кодирующем слое, мы просто суммируем полученные потери и добавляем результат к функции издержек. Чтобы контролировать относительную важность потери из-за разреженности и потери из-за реконструкции, мы можем умножить потерю из-за разреженности на гиперпараметр веса разреженности. Если вес очень высок, то модель будет держаться близко к целевой разреженности, но она может не реконструировать входы надлежащим образом, делая модель бесполезной. И наоборот, если вес слишком низок, тогда модель будет по большей части игнорировать цель разреженности и не узнает никаких интересных признаков.

Реализация с помощью TensorFlow

Теперь мы располагаем всем необходимым для реализации разреженного автокодировщика с применением TensorFlow:

```
def kl_divergence(p, q):  
    return p * tf.log(p / q) + (1 - p) * tf.log((1 - p) / (1 - q))  
  
learning_rate = 0.01  
sparsity_target = 0.1  
sparsity_weight = 0.2  
  
[...] # Построение нормального автокодировщика  
      # (в данном примере кодирующим слоем является hidden1)
```

```
hidden1_mean = tf.reduce_mean(hidden1, axis=0) # среднее по пакету
sparsity_loss = tf.reduce_sum(kl_divergence(sparsity_target,
                                           hidden1_mean))
reconstruction_loss = tf.reduce_mean(tf.square(outputs - X)) # MSE
loss = reconstruction_loss + sparsity_weight * sparsity_loss
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(loss)
```

Важной деталью является тот факт, что активации кодирующего слоя должны быть между 0 и 1 (но не равны 0 или 1), иначе функция подсчета расстояния Кульбака–Лейблера возвратит NaN (Not a Number — не число). Простое решение предполагает использование логистической функции активации для кодирующего слоя:

```
hidden1 = tf.layers.dense(X, n_hidden1, activation=tf.nn.sigmoid)
```

Ускорить сходжение позволит несложный трюк: вместо применения MSE мы можем выбрать потерю из-за реконструкции, которая будет иметь более высокие градиенты. Часто хорошим вариантом является перекрестная энтропия. Для ее использования мы должны нормализовать входы, чтобы сделать их получающими значения от 0 до 1, и задействовать в выходном слое логистическую функцию активации, обеспечив на выходах значения от 0 до 1. Функция `sigmoid_cross_entropy_with_logits()` из TensorFlow позаботится об эффективном применении логистической (сигмоидальной) функции активации к выходам и о вычислении перекрестной энтропии:

```
[...]
logits = tf.layers.dense(hidden1, n_outputs)
outputs = tf.nn.sigmoid(logits)
xentropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=X,
                                                    logits=logits)
reconstruction_loss = tf.reduce_sum(xentropy)
```

Обратите внимание, что операция `outputs` во время обучения не нужна (мы используем ее только тогда, когда хотим взглянуть на реконструкции).

Вариационные автокодировщики

Еще одна важная категория автокодировщиков была введена в 2014 году Дидериком Кингма и Максом Веллингом⁵, быстро став одним из самых по-

⁵ “Auto-Encoding Variational Bayes” (“Автокодирование на основе вариационного байесовского подхода”), Д. Кингма и М. Веллинг (2014 год) (<https://goo.gl/NZq7r2>).

пулярных типов автокодировщиков: *вариационные автокодировщики* (*variational autoencoder*). Вариационные автокодировщики совершенно отличаются от всех автокодировщиков, которые обсуждались до сих пор, по перечисленным ниже причинам.

- Они являются *вероятностными автокодировщиками*, т.е. их выходы отчасти определяются случайно, даже после обучения (в противоположность шумоподавляющим автокодировщикам, которые задействуют случайность только во время обучения).
- Более того, они являются *порождающими автокодировщиками*, т.е. могут генерировать новые образцы, которые выглядят так, будто они были выбраны из обучающего набора.

Обе характеристики делают вариационные автокодировщики похожими на ограниченные машины Больцмана (см. приложение Д), но их проще обучать, а процесс выборки намного быстрее (в случае ограниченных машин Больцмана необходимо ждать, пока сеть придет в “тепловое равновесие”, прежде чем можно будет выбрать новый образец).

Давайте посмотрим, как они работают. На рис. 15.11 (слева) показан вариационный автокодировщик. Конечно, вы можете узнать в нем базовую структуру, присущую всем автокодировщикам, с кодировщиком, за которым следует декодировщик (в этом примере они оба имеют два скрытых слоя), но здесь есть одна уловка: вместо выпуска кодировки для заданного входа напрямую кодировщик выдает *среднюю кодировку μ* и стандартное отклонение σ . Действительная кодировка затем выбирается случайным образом из гауссова распределения со средним μ и стандартным отклонением σ . Далее декодировщик просто декодирует выбранную кодировку как обычно. В правой части рис. 15.11 представлен образец, проходящий через данный автокодировщик. Сначала кодировщик вырабатывает μ и σ , затем случайным образом выбирается кодировка (обратите внимание, что она расположена не точно в позиции μ), а в заключение эта кодировка декодируется, и финальный выход напоминает обучающий образец.

На диаграмме видно, что хотя входы могут иметь весьма спиралевидное распределение, вариационный автокодировщик стремится выдавать кодировки, которые выглядят так, будто они были выбраны из простого гауссова распределения⁶:

⁶ Вариационные автокодировщики на самом деле являются более универсальными; кодировки не ограничиваются гауссовым распределением.

во время обучения функция издержек (обсуждается ниже) вынуждает кодировки постепенно мигрировать внутри пространства кодировок (также называемого *латентным пространством* (*latent space*)), чтобы занять примерно (гипер)сферическую область, похожую на облако гауссовых точек.

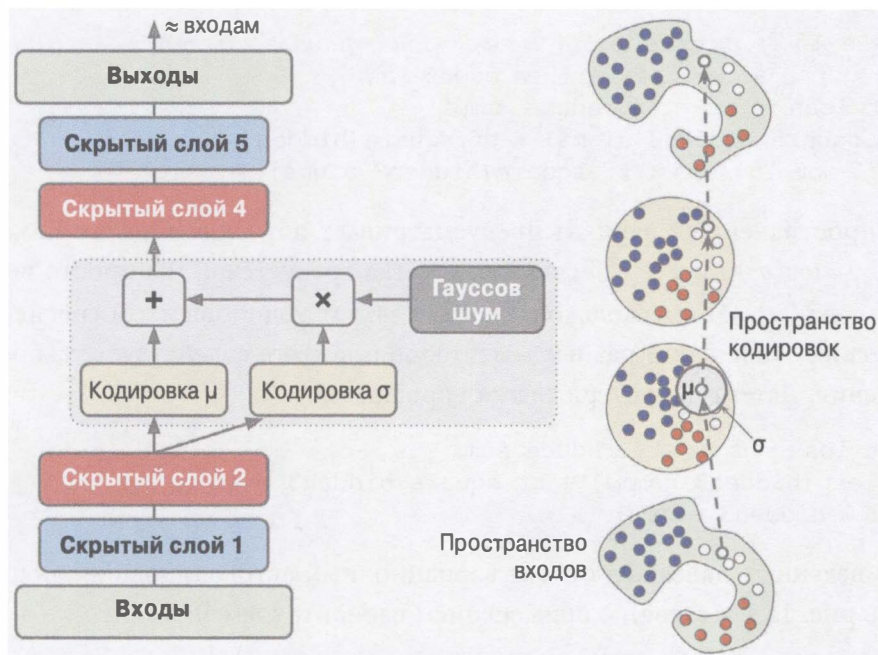


Рис. 15.11. Вариационный автокодировщик (слева) и проходящий через него образец (справа)

Важное следствие заключается в том, что после обучения вариационного автокодировщика можно очень легко сгенерировать новый образец: нужно всего лишь выбрать из гауссова распределения случайную кодировку, декодировать ее — и готово!

Итак, рассмотрим функцию издержек. Она состоит из двух частей. Первая часть представляет собой обычную потерю из-за реконструкции, которая заставляет автокодировщик воспроизводить свои входы (как обсуждалось ранее, для этого мы можем применять перекрестную энтропию). Вторая часть — *латентная потеря* (*latent loss*), вынуждающая автокодировщик иметь кодировки, которые выглядят так, будто они были выбраны из простого гауссова распределения, для чего мы используем расстояние Кульбака–Лейблера между целевым (гауссовым) распределением и фактическим распределением кодировок.

Математические выкладки чуть сложнее, чем ранее, в частности из-за гауссова шума, ограничивающего объем информации, которая может быть передана кодирующему слою (тем самым заставляя автокодировщик узнавать полезные признаки). К счастью, уравнения упрощаются до следующего кода для латентной потери⁷:

```
eps = 1e-10 # сглаживающий член во избежание вычисления log(0),
            # представляющего собой NaN
latent_loss = 0.5 * tf.reduce_sum(
    tf.square(hidden3_sigma) + tf.square(hidden3_mean)
    - 1 - tf.log(eps + tf.square(hidden3_sigma)))
```

Распространенный вариант предусматривает обучение кодировщика выдавать $\gamma = \log(\sigma^2)$ вместо σ . Всякий раз, когда требуется σ , мы просто вычисляем $\sigma = \exp(\frac{\gamma}{2})$. Это несколько облегчает захват кодировщиком среднеквадратических отклонений разных масштабов и в итоге содействует ускорению схождения. Латентная потеря слегка упрощается:

```
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean)
    - 1 - hidden3_gamma)
```

Приведенный далее код строит вариационный автокодировщик, показанный на рис. 15.11 (слева), с применением варианта $\log(\sigma^2)$:

```
from functools import partial

n_inputs = 28 * 28
n_hidden1 = 500
n_hidden2 = 500
n_hidden3 = 20 # кодировки
n_hidden4 = n_hidden2
n_hidden5 = n_hidden1
n_outputs = n_inputs
learning_rate = 0.001

initializer = tf.contrib.layers.variance_scaling_initializer()
my_dense_layer = partial(
    tf.layers.dense,
    activation=tf.nn.elu,
    kernel_initializer=initializer)

X = tf.placeholder(tf.float32, [None, n_inputs])
```

⁷ За математическими выкладками обращайтесь к исходной работе или к замечательному руководству Карла Доерша (2016 год) (<https://goo.gl/ViiAzQ>).

```

hidden1 = my_dense_layer(X, n_hidden1)
hidden2 = my_dense_layer(hidden1, n_hidden2)
hidden3_mean = my_dense_layer(hidden2, n_hidden3, activation=None)
hidden3_gamma = my_dense_layer(hidden2, n_hidden3, activation=None)
noise = tf.random_normal(tf.shape(hidden3_gamma), dtype=tf.float32)
hidden3 = hidden3_mean + tf.exp(0.5 * hidden3_gamma) * noise
hidden4 = my_dense_layer(hidden3, n_hidden4)
hidden5 = my_dense_layer(hidden4, n_hidden5)
logits = my_dense_layer(hidden5, n_outputs, activation=None)
outputs = tf.sigmoid(logits)
xentropy =
    tf.nn.sigmoid_cross_entropy_with_logits(labels=X, logits=logits)
reconstruction_loss = tf.reduce_sum(xentropy)
latent_loss = 0.5 * tf.reduce_sum(
    tf.exp(hidden3_gamma) + tf.square(hidden3_mean)
    - 1 - hidden3_gamma)
loss = reconstruction_loss + latent_loss

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(loss)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

Генерирование цифр

Давайте воспользуемся созданным вариационным автокодировщиком, чтобы сгенерировать изображения, которые выглядят как рукописные цифры. Для этого понадобится лишь обучить модель, случайным образом выбрать кодировки из гауссова распределения и декодировать их.

```

import numpy as np

n_digits = 60
n_epochs = 50
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        n_batches = mnist.train.num_examples // batch_size
        for iteration in range(n_batches):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch})

        codings_rnd = np.random.normal(size=[n_digits, n_hidden3])
        outputs_val = outputs.eval(feed_dict={hidden3: codings_rnd})

```


Вот и все. Теперь мы можем посмотреть, на что похожи “рукописные” цифры, выпущенные автокодировщиком (рис. 15.12):

```
for iteration in range(n_digits):  
    plt.subplot(n_digits, 10, iteration + 1)  
    plot_image(outputs_val[iteration])
```

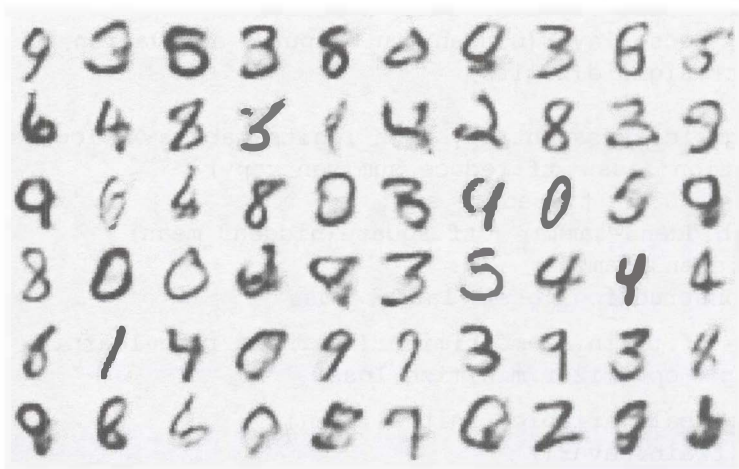


Рис. 15.12. Изображения рукописных цифр, сгенерированные вариационным автокодировщиком

Большинство цифр выглядят достаточно убедительно, в то время как некоторые являются скорее “креативными”. Но не будьте слишком строги в отношении автокодировщика — он начал учиться менее часа тому назад. Дайте ему чуть больше времени на обучение, и генерируемые цифры будут все лучше и лучше.

Другие автокодировщики

Поразительные успехи обучения с учителем в распознавании изображений, распознавании речи, переводе текста и многом другом отчасти затмевают обучение без учителя, но на самом деле оно бурно развивается. Регулярно изобретаемых новых архитектур для автокодировщиков и других алгоритмов обучения без учителя настолько много, что раскрыть их все в данной книге попросту невозможно.

Ниже представлен краткий (и далекий от полноты) обзор ряда дополнительных типов автокодировщиков, которые могут вызвать у вас интерес.

Сжимающий автокодировщик (contractive autoencoder (CAE))⁸

Автокодировщик ограничивается во время обучения так, что производные кодировок относительно входов являются небольшими. Другими словами, два похожих входа должны иметь похожие кодировки.

Многослойные сверточные автокодировщики (stacked convolutional autoencoder)⁹

Автокодировщики, которые обучаются выделению визуальных признаков путем восстановления изображений, обработанных сверточными слоями.

Порождающая стохастическая сеть (generative stochastic network (GSN))¹⁰

Обобщение шумоподавляющих автокодировщиков с добавленной возможностью генерации данных.

Автокодировщик типа “победитель получает все” (winner-take-all (WTA) autoencoder)¹¹

Во время обучения после вычисления активаций всех нейронов в кодирующем слое для каждого нейрона на обучающем пакете сохраняются только верхние $k\%$ активаций, а остальные устанавливаются в нули. Это естественным образом приводит к разреженным кодировкам. Кроме того, похожий подход WTA может использоваться для получения разреженных сверточных автокодировщиков.

Порождающая состязательная сеть (Generative Adversarial Network (GAN))¹²

Одна сеть, называемая “дискриминатором”, обучается отличать реальные данные от поддельных данных, произведенных второй сетью, которая называется “генератором”. Генератор обучается обманывать диск-

⁸ “Contractive Auto-Encoders: Explicit Invariance During Feature Extraction” (“Сжимающие автокодировщики: явная инвариантность во время выделения признаков”), С. Рифаи и др. (2011 год) (<https://goo.gl/U5t9Ux>).

⁹ “Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction” (“Многослойные сверточные автокодировщики для выделения иерархических признаков”), Д. Маски и др. (2011 год) (<https://goo.gl/PTwsol>).

¹⁰ “GSNs: Generative Stochastic Networks” (“GSN: порождающие стохастические сети”), Г. Алэн и др. (2015 год) (<https://goo.gl/HjONlm>).

¹¹ “Winner-Take-All Autoencoders” (“Автокодировщики типа ‘победитель получает все’”), А. Махзани и Б. Фрей (2015 год) (<https://goo.gl/1lLvzL>).

¹² “Generative Adversarial Networks” (“Порождающие состязательные сети”), Я. Гудфеллоу и др. (2014 год) (<https://goo.gl/qd4RhN>).

риминатор, а дискриминатор учится уклоняться от обмана генератора. Такое состязание приводит к появлению все более реалистичных поддельных данных и полностью надежных кодировок. Состязательное обучение — очень мощная идея, в настоящее время набирающая обороты. Ян Лекун даже назвал ее “лучшей вещью в мире”.

Упражнения

1. Каковы главные задачи, для которых применяются автокодировщики?
2. Предположим, вы хотите обучить классификатор и имеете очень много непомеченных обучающих данных, но лишь несколько тысяч помеченных образцов. Чем могут помочь автокодировщики? Как вы поступите?
3. Если автокодировщик идеально восстанавливает входы, то обязательно ли он является хорошим автокодировщиком? Как можно оценить производительность автокодировщика?
4. Что собой представляют понижающий и повышающий автокодировщики? Каков главный риск чрезмерно понижающего автокодировщика? А каков главный риск повышающего автокодировщика?
5. Как связывать веса в многослойном автокодировщике? Какой смысл делать это?
6. Как выглядит распространенный прием визуализации признаков, которые узнал самый нижний слой многослойного автокодировщика? Что насчет более высоких слоев?
7. Что такое порождающая модель? Можете ли вы назвать тип порождающего автокодировщика?
8. Воспользуйтесь шумоподавляющим автокодировщиком для предварительного обучения классификатора изображений.
 - Если вас интересует более сложная задача, тогда можете взять набор MNIST (самый простой) или другой крупный набор изображений, такой как CIFAR10 (<https://goo.gl/VbsmxG>). В случае выбора CIFAR10 вам понадобится написать код для загрузки пакетов изображений для обучения. Если вы хотите пропустить эту часть, то зоопарк моделей TensorFlow содержит подходящие инструменты (<https://goo.gl/GEDqG8>).

- Расщепите набор данных на обучающий набор и испытательный набор. Обучите глубокий шумоподавляющий автокодировщик на полном обучающем наборе.
- Удостоверьтесь в том, что изображения достаточно хорошо реконструируются, и визуализируйте низкоуровневые признаки. Визуализируйте изображения, которые больше других активируют каждый нейрон в кодирующем слое.
- Постройте глубокую нейронную сеть для классификации с применением самых нижних слоев автокодировщика. Обучите ее, используя только 10% обучающего набора. Можете ли вы заставить ее работать в той же мере хорошо, как и аналогичный классификатор, обученный на полном обучающем наборе?

9. *Семантическое хеширование* (*semantic hashing*), введенное в 2008 году Русланом Салахутдиновым и Джеффри Хинтоном¹³, представляет собой прием, применяемый для эффективного *информационного поиска* (*information retrieval*): документ (скажем, изображение) пропускается через систему, обычно нейронную сеть, которая выдает двоичный вектор с довольно малым числом измерений (например, 30 битов). Два похожих документа, вероятно, будут иметь идентичные или очень похожие хеши. Индексирование каждого документа с использованием его хеша делает возможным извлечение многих документов, похожих на имеющийся документ, почти мгновенно, даже если документов миллиарды: нужно лишь вычислить хеш документа и поискать все документы с таким же хешем (либо с хешами, отличающимися только одним или двумя битами). Реализуйте семантическое хеширование с применением слегка подстроенного многослойного автокодировщика.

- Создайте многослойный автокодировщик, содержащий два скрытых слоя ниже кодирующего слоя, и обучите его на наборе данных с изображениями, который использовался в предыдущем упражнении. Кодирующий слой должен содержать 30 нейронов и применять логистическую функцию активации для выдачи значений между 0 и 1. Чтобы получить хеш изображения после обучения, вы можете просто прогнать его через автокодировщик, взять выход кодирующего слоя и округлить каждое значение до ближайшего целого (0 или 1).

¹³ “Semantic Hashing” (“Семантическое хеширование”), Р. Салахутдинов и Д. Хинтон (2008 год) (<https://goo.gl/LXzFX6>).

- Ловкий трюк, предложенный Салахутдиновым и Хинтоном, предусматривает добавление гауссова шума (с нулевым средним) к входам кодирующего слоя только во время обучения. Чтобы сохранить отношение “сигнал-шум” высоким, автокодировщик будет учиться подавать крупные значения кодирующему слою (так что шум становится пренебрежимо малым). В свою очередь это означает, что логистическая функция кодирующего слоя, скорее всего, будет насыщаться в точке 0 или 1. В результате округление кодировок к 0 или 1 не слишком сильно исказит их, и это повысит надежность хешей.
- Вычислите хеш каждого изображения и посмотрите, выглядят ли одинаковыми изображения с идентичными хешами. Поскольку наборы MNIST и CIFAR10 являются помеченными, более объективный способ измерения производительности автокодировщика для семантического хеширования заключается в том, чтобы удостовериться, что изображения с тем же самым хешем в общем случае имеют один и тот же класс. Сделать это можно, измерив среднюю загрязненность Джини (введенную в главе 6) наборов изображений с идентичными (или очень похожими) хешами.
- Попробуйте точно подстроить гиперпараметры, используя перекрестную проверку.
- Обратите внимание, что в случае помеченного набора данных еще один подход предусматривает обучение сверточной нейронной сети (см. главу 13) для классификации с последующим применением выходного слоя для выпуска хешей. Ознакомьтесь с работой Йиньма Гуа и Яньмина Ли, написанной в 2015 году¹⁴. Посмотрим, работает ли это лучше.

10. Обучите вариационный автокодировщик на наборе данных с изображениями, который использовался в предшествующих упражнениях (MNIST или CIFAR10), и заставьте его генерировать изображения. В качестве альтернативы попробуйте найти непомеченный набор данных, который вас заинтересует, и посмотреть, сумеете ли вы генерировать новые образцы.

Решения приведенных упражнений доступны в приложении А.

¹⁴ “CNN Based Hashing for Image Retrieval” (“Хеширование на основе сверточной нейронной сети для поиска изображений”), Й. Гуа и Я. Ли (2015 год) (<https://goo.gl/i9FTln>).

Обучение с подкреплением

Обучение с подкреплением (*Reinforcement Learning* — *RL*) на сегодняшний день является одной из наиболее захватывающих областей машинного обучения, а также одним из самых старых подходов. Оно появилось в 1950-х годах и за годы произвело много интересных приложений¹, особенно в игровой отрасли (например, *TD-Gammon*, программу для игры в *нард*) и в управлении станками, но нечасто попадало в заголовки новостей. Однако в 2013 году произошел крутой перелом, когда исследователи из английского стартапа под названием DeepMind продемонстрировали систему, которая могла научиться играть практически в любую игру Atari с нуля² и со временем превосходить людей³ в большинстве игр, используя только низкоуровневые пиксели в качестве входов и не располагая предварительными знаниями правил игр⁴. Это было первым в серии удивительных трюков, кульминацией которых стала победа в мае 2017 года системы AlphaGo над Кэ Цзе, чемпионом мира по игре *го*. Ни одна программа даже близко не подбиралась к выигрышу у мастера в данной игре, не говоря уже о самом чемпионе мира. Сегодня вся область обучения с подкреплением кипит новыми идеями с широким спектром приложений. В 2014 году стартап DeepMind был приобретен компанией Google за сумму, превышающую 500 миллионов долларов.

¹ Дополнительные детали ищите в книге Ричарда Саттона и Эндрю Барто *Reinforcement Learning: An Introduction* (MIT Press) (<https://goo.gl/ebVg6i>) или в бесплатном курсе лекций по обучению с подкреплением Дэвида Силвера (<https://goo.gl/AWcMFw>) из Университетского колледжа Лондона.

² “Playing Atari with Deep Reinforcement Learning” (“Игра в игры Atari с помощью глубокого обучения с подкреплением”), В. Мних и др. (2013 год) (<https://goo.gl/hceDs5>).

³ “Human-level control through deep reinforcement learning” (“Контроль человеческого уровня посредством глубокого обучения с подкреплением”), В. Мних и др. (2015 год) (<https://goo.gl/hgpvz7>).

⁴ Просмотрите видеоролики с демонстрацией обучения системы DeepMind играм *Space Invaders*, *Breakout* и другим, которые доступны по ссылке <https://goo.gl/yTsH6X>.

Как же исследователи этого добились? С оглядкой назад все кажется довольно простым: они применили мощь глубокого обучения к области обучения с подкреплением, что превзошло их самые смелые мечты. В настоящей главе мы сначала выясним, что собой представляет обучение с подкреплением и для чего оно хорошо подходит, а затем опишем два наиболее важных приема в глубоком обучении с подкреплением: *градиенты политики* (*policy gradient*) и *глубокие Q-сети* (*Deep Q-Network — DQN*), включая обсуждение *марковских процессов принятия решений* (*Markov Decision Process — MDP*). Мы будем использовать такие приемы, чтобы научить одну модель балансировать дышло в двигающейся телеге, а другую модель — играть в игры Atari. Те же самые приемы могут применяться для широкого разнообразия задач, от шагающих роботов до беспилотных автомобилей.

Обучение для оптимизации наград

При обучении с подкреплением программный *агент* (*agent*) делает *наблюдения* (*observation*) и предпринимает *действия* (*action*) внутри *среды* (*environment*), получая в ответ *награды* (*reward*). Его цель — научиться действовать так, чтобы довести до максимума ожидаемые долгосрочные награды. Если вы не противник некоторой доли антропоморфизма, то можете считать положительные награды удовольствием, а отрицательные — страданием (в данном случае термин “награда” несколько дезориентирует). Короче говоря, агент действует в среде и учится методом проб и ошибок максимально увеличивать удовольствие и сводить к минимуму страдание.

Это довольно обширная установка, которая может применяться к широкому выбору задач. Ниже перечислено несколько примеров (рис. 16.1).

- а) Агент может быть программой, которая управляет шагающим роботом. В таком случае средой является реальный мир, агент наблюдает за средой через набор *датчиков* вроде камер и датчиков касания, а действия представляют собой отправку сигналов, запускающих моторы. Робот может быть запрограммирован на получение положительных наград, когда он приближается к целевому месту назначения, и отрицательных наград, когда попусту тратит время, двигается в ошибочном направлении или падает.
- б) Агент может быть программой, управляющей мисс Пэкмен (Ms. Pac-Man). В данном случае среда — это симуляция игры Atari, действия — девять возможных положений джойстика (влево вверх, вниз, центральное и т.д.), наблюдения — экранные снимки, а награды — очки в игре.

- в) Аналогично агент может быть программой, играющей в настольную игру, такую как *го*.
- г) Агент вовсе не обязан управлять двигающейся вещью физически (или виртуально). Скажем, агент может быть интеллектуальным термостатом, получая положительные награды всякий раз, когда он близок к целевой температуре и экономит энергию, и отрицательные награды, если людям приходится подстраивать температуру, а потому агент должен предугадывать человеческие потребности.
- д) Агент может наблюдать за биржевыми курсами и каждую секунду решать, сколько акций покупать или продавать. Наградами вполне очевидно являются денежные доходы и убытки.

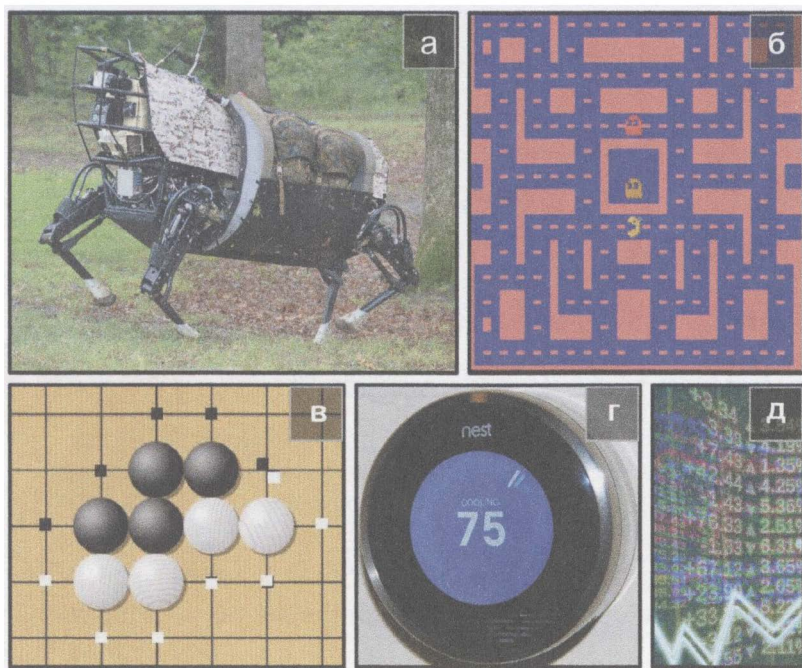


Рис. 16.1. Примеры обучения с подкреплением: (а) шагающий робот, (б) мисс Пэкмен, (в) игрок в го, (г) термостат, (д) автоматический биржевой маклер⁵

⁵ Изображения (а), (в) и (г) воспроизведены из Википедии. Изображения (а) и (г) находятся в публичной собственности. Изображение (в) было создано пользователем Stevertigo и выпущено под лицензией Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>). Изображение (б) — экранный снимок из игры Ms. Pac-Man, принадлежащей Atari (автор полагает, что в главе оно используется законно). Изображение (д) воспроизведено из Pixabay и выпущено под лицензией Creative Commons CC0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

Обратите внимание, что иногда положительные награды вообще не существуют; например, агент может перемещаться по лабиринту, получая на каждом временном шаге отрицательную награду, поэтому ему лучше максимально быстро найти выход! Есть много других примеров задач, для которых хорошо подходит обучение с подкреплением, такие как беспилотные автомобили, размещение рекламы на веб-странице или управление тем, куда должна сосредоточивать свое внимание система классификации изображений.

Поиск политики

Алгоритм, применяемый программным агентом для определения своих действий, называется *политикой* (*policy*). Скажем, политикой могла бы быть нейронная сеть, которая на входе принимает наблюдения и выдает действие, подлежащее выполнению (рис. 16.2).



Рис. 16.2. Обучение с подкреплением, использующее политику в форме нейронной сети

Политикой может быть любой алгоритм, какой только вы можете придумать, и он даже не обязан быть детерминированным. Например, возьмем роботизированный пылесос, наградой которого является объем пыли, собранной за 30 минут. Его политика могла бы предусматривать ежесекундное движение вперед с вероятностью p либо случайный поворот влево или вправо с вероятностью $1 - p$. Угол поворота был бы случайным числом между $-r$ и $+r$. Поскольку эта политика включает в себя некоторую случайность, она называется *стохастической политикой*. Робот будет иметь непредсказуемую траекторию, что гарантирует его попадание со временем в любое достижимое место и сбор всей пыли. Вопрос в том, сколько пыли он соберет за 30 минут?

Как бы вы обучали такого робота? Есть только два *параметра политики* (*policy parameter*), которые можно подстраивать: вероятность p и диапазон углов поворота r . Один возможный алгоритм обучения мог бы предполагать опробование множества разных значений для указанных параметров и выбор комбинации, которая работает наилучшим образом (рис. 16.3). Это пример *поиска политики* (*policy search*), в данном случае с применением подхода грубой силы. Тем не менее, когда *пространство политик* (*policy space*) слишком велико (как обычно случается), то нахождение хорошего набора параметров подобным способом будет похоже на поиск иголки в гигантском стоге сена.

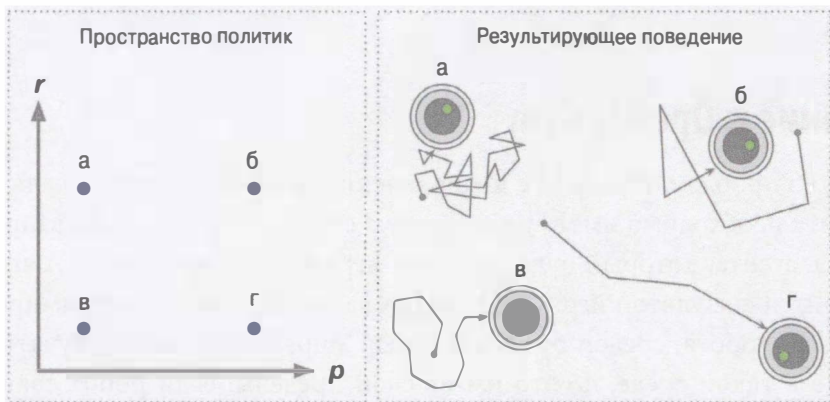


Рис. 16.3. Четыре точки в пространстве политик и соответствующее поведение агента

Другой способ исследования пространства политик связан с использованием *генетических алгоритмов* (*genetic algorithm*). Например, вы могли бы случайным образом создать первое поколение 100 политик и испытать их, затем “уничтожить” 80 наихудших политик⁶ и заставить каждую из 20 выживших политик произвести по 4 потомка. Потомок представляет собой просто копию своего родителя⁷, дополненную случайной вариацией. Выжившие политики вместе со своими потомками образуют второе поколение. Проход по поколениям в подобном стиле можно продолжать до тех пор, пока не будет найдена подходящая политика.

⁶ Часто лучше предоставить плохим исполнителям небольшой шанс выжить, чтобы сохранить некоторое разнообразие в “генофонде”.

⁷ При наличии единственного родителя это называется *вегетативным воспроизведением*. С двумя (и более) родителями получается *половое размножение*. Геном потомка (в данном случае набор параметров политики) произвольно образован из частей геномов своих родителей.

Еще один подход предусматривает применение приемов оптимизации путем оценки градиентов наград относительно параметров политики и подстройки этих параметров, следуя градиенту в направлении более высоких наград (*градиентный подъем* (*gradient ascent*)). Такой подход называется *градиентами политики* (*Policy Gradient* — *PG*) и более подробно обсуждается далее в главе. В случае робота-пылесоса вы могли бы чуть повысить p и выяснить, привело ли это к увеличению объема пыли, собранной им за 30 минут; если привело, тогда еще больше повысить p или в противном случае снизить p . Мы реализуем популярный алгоритм PG с использованием TensorFlow, но сначала нужно создать среду для агента, так что самое время представить OpenAI Gym.

Введение в OpenAI Gym

Одна из проблем обучения с подкреплением связана с тем, что для обучения агента необходимо иметь работающую среду. Если вы хотите запрограммировать агента, который будет учиться играть в какую-то игру Atari, тогда понадобится симулятор игры Atari. Когда вас интересует программирование шагающего робота, средой будет реальный мир, и вы можете обучать робота прямо в такой среде, но это имеет свои пределы: если робот свалится с отвесной скалы, то вы не в состоянии просто нажать на “кнопку отмены”. Вы также не можете ускорить время, а добавление дополнительной вычислительной мощности не заставит робот перемещаться хоть как-то быстрее. К тому же обучать параллельно сразу 1 000 роботов обычно слишком дорого. Короче говоря, обучение в реальном мире является трудным и медленным, так что в большинстве случаев нужна имитированная среда, по крайней мере, чтобы запустить обучение.

OpenAI Gym (<https://gym.openai.com/>)⁸ — это комплект инструментов, который предоставляет широкое разнообразие имитированных сред (игры Atari, настольные игры, двумерные и трехмерные физические симуляции и т.д.), давая возможность обучать агентов, сравнивать их или разрабатывать новые алгоритмы RL.

⁸ OpenAI — некоммерческая компания, занимающаяся исследованиями в области искусственного интеллекта, которую частично финансирует Илон Маск. Ее заявленная цель — продвигать и разрабатывать дружественные искусственные интеллекты, которые будут приносить пользу человечеству (а не истребят его).

Давайте установим OpenAI Gym. Если вы создавали изолированную среду, то должны сначала ее активировать:

```
$ cd $ML_PATH # Ваш рабочий каталог МО (например, $HOME/ml)
$ source env/bin/activate
```

Далее установите OpenAI Gym (если вы не применяете `virtualenv`, тогда потребуются права администратора, или же добавьте параметр `--user`):

```
$ pip3 install --upgrade gym
```

Затем откройте командную оболочку Python или тетрадь Jupyter и создайте свою первую среду:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2017-08-27 11:08:05,742] Making new env: CartPole-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115,  0.02337094,  0.00720711])
>>> env.render()
```

Функция `make()` создает среду, в данном случае `CartPole`. Это двумерная симуляция, в которой телега может быть ускорена влево или вправо, чтобы балансировать дышло, размещенное сверху (рис. 16.4). После создания среды мы должны ее инициализировать с использованием метода `reset()`. В результате возвращается первое наблюдение. Наблюдения зависят от типа среды. Для среды `CartPole` каждое наблюдение является одномерным массивом NumPy, содержащим четыре числа с плавающей точкой, которые представляют горизонтальное положение телеги (0.0 = центр), скорость телеги, угол дышла (0.0 = вертикально) и угловую скорость дышла. Наконец, метод `render()` отображает среду, как показано на рис. 16.4.

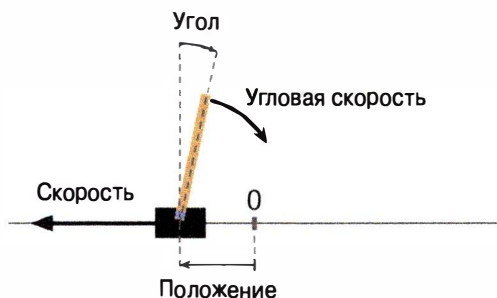


Рис. 16.4. Среда `CartPole`

Если вы хотите, чтобы метод `render()` возвращал визуализированное изображение в виде массива NumPy, тогда можете установить параметр режима `mode` в `rgb_array` (обратите внимание, что другие среды могут поддерживать отличающиеся режимы):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # высота, ширина, каналы (3=RGB)
(400, 600, 3)
```



К сожалению, среда CartPole (и несколько других сред) визуализирует изображение на экране, даже если вы установите режим в `"rgb_array"`. Единственный способ избежать этого предусматривает применение поддельного сервера X, такого как Xvfb или Xdummy. Скажем, вы можете установить Xvfb и запустить Python, используя следующую команду: `xvfb-run -s "-screen 0 1400x900x24" python`. Либо вы можете применять пакет `xvfbwrapper` (<https://goo.gl/wRloJl>).

Давайте выясним у среды, какие действия возможны:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` означает, что возможными действиями являются целые числа 0 и 1, которые представляют ускорение влево (0) или вправо (1). Другие среды могут иметь больше дискретных действий либо отличающиеся типы действий (например, непрерывные). Поскольку дышло кренится вправо, ускорим телегу вправо:

```
>>> action = 1 # ускорение вправо
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608, 0.16189797, 0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

Метод `step()` выполняет заданное действие и возвращает четыре значения, которые описаны ниже.

obs

Новое наблюдение. Телега теперь движется вправо (`obs[1]>0`). Дышло по-прежнему дает крен вправо (`obs[2]>0`), но его угловая скорость уже отрицательная (`obs[3]<0`), так что после следующего шага оно, вероятно, будет крениться влево.

reward

В данной среде вы получаете награду 1.0 на каждом шаге вне зависимости от того, что делаете, а потому цель в том, чтобы как можно дольше продолжать движение.

done

Значением будет `True`, когда *эпизод* (*episode*) закончился. Такое произойдет, когда дышло даст слишком большой крен. После этого среда должна быть сброшена, прежде чем ее можно будет использовать снова.

info

Словарь, который может предоставлять дополнительную отладочную информацию в других средах. Такие данные не должны применяться для обучения (это было бы обманом).

Давайте жестко закодируем простую политику, которая ускоряет влево, когда дышло дает крен влево, и ускоряет вправо, когда дышло кренится вправо. Мы будем проводить эту политику, чтобы увидеть средние награды, получаемые ею за 500 эпизодов:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # максимум 1000 шагов, нам не нужно
                            # бесконечное выполнение
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

Приведенный выше код не должен требовать объяснений. Взглянет на результат:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(42.125999999999998, 9.1237121830974033, 24.0, 68.0)
```

Даже при 500 попытках данная политика так и не смогла удержать дышло прямо больше чем для 68 последовательных шагов. Не особо впечатляет. Если вы просмотрите симуляцию в тетрадах (<https://github.com/ageron/handson-ml>), то заметите, что телега энергично раскачивается влево и вправо все больше и больше до тех пор, пока дышло не даст слишком большой крен. Теперь выясним, сможет ли нейронная сеть найти лучшую политику.

Политики в форме нейронных сетей

Давайте создадим политику в форме нейронной сети. Подобно политике, жестко закодированной ранее, эта нейронная сеть будет в качестве входа получать наблюдение и выдавать действие, подлежащее выполнению. Точнее говоря, она оценит вероятность для каждого действия, после чего мы выберем действие случайным образом согласно оценочным вероятностям (рис. 16.5).

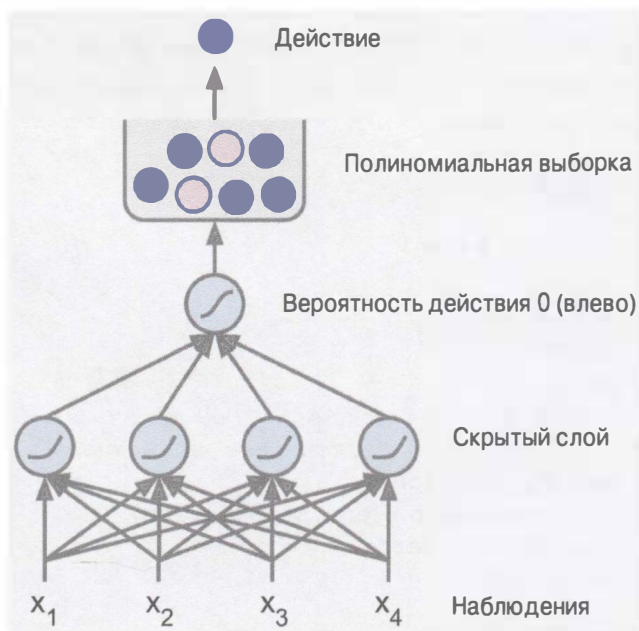


Рис. 16.5. Политика в форме нейронной сети

В случае среды CartPole есть всего два возможных действия (влево или вправо), так что нам необходим выходной нейрон. Он будет выдавать вероятность p действия 0 (влево), а вероятность действия 1 (вправо) составит, конечно же, $1 - p$. Например, если выходной нейрон выдаст 0.7, тогда мы выберем действие 0 с вероятностью 70% и действие 1 вероятностью 30%.

Вас может интересовать, почему мы выбираем случайное действие на основе вероятности, выданной нейронной сетью, а не действие с самой высокой суммой оценки. Такой подход позволяет агенту находить правильный баланс между *исследованием* новых действий и *эксплуатацией* действий, о которых известно, что они работают хорошо. Вот аналогия: предположим, вы зашли в какой-то ресторан впервые, и все блюда выглядят одинаково привлекательными, так что вы выбираете случайным образом одно из них. Если оно вам понравится, тогда вы можете повысить вероятность его заказа в следующий раз. Однако вы не должны увеличивать эту вероятность до 100%, иначе никогда не попробуете другие блюда, часть которых может оказаться даже лучше, чем то, что вы попробовали при первом посещении ресторана.

Вдобавок обратите внимание, что в этой конкретной среде прошедшие действия и наблюдения можно безопасно игнорировать, т.к. каждое наблюдение содержит полное состояние среды. Если бы существовало какое-то скрытое состояние, то мог бы потребоваться учет также прошедших действий и наблюдений. Скажем, если среда обнаруживает положение телеги, но не ее скорость, тогда для оценки текущей скорости пришлось бы учитывать не только текущее, но и предыдущее наблюдение. Другой пример касается ситуации, когда наблюдения зашумлены; в таком случае обычно возникает необходимость использовать несколько прошедших наблюдений, чтобы определить наиболее правдоподобное текущее состояние. Таким образом, задача CartPole крайне проста; наблюдения свободны от шума и содержат полное состояние среды.

Ниже приведен код, который строит описанную нейронную сеть с применением TensorFlow:

```
import tensorflow as tf

# 1. Определение архитектуры нейронной сети
n_inputs = 4      # == env.observation_space.shape[0]
n_hidden = 4      # задача проста, поэтому нет нужды
                  # в большем числе скрытых нейронов
n_outputs = 1     # выдает только вероятность ускорения влево
initializer = tf.contrib.layers.variance_scaling_initializer()
```

```
# 2. Построение нейронной сети
```

```
X = tf.placeholder(tf.float32, shape=[None, n_inputs])  
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,  
                           kernel_initializer=initializer)  
logits = tf.layers.dense(hidden, n_outputs,  
                           kernel_initializer=initializer)  
outputs = tf.nn.sigmoid(logits)
```

```
# 3. Выбор случайного действия на основе оценочных вероятностей
```

```
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])  
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)  
init = tf.global_variables_initializer()
```

Давайте пройдемся по коду.

1. После импорта мы определяем архитектуру нейронной сети. Количество входов соответствует размеру пространства наблюдений (которое в случае CartPole равно четырем), мы просто имеем четыре скрытых элемента, не нуждаясь в большем их числе, и один выходной элемент для выдачи вероятности (вероятности движения влево).
2. Далее мы строим нейронную сеть. В рассматриваемом примере это обычный многослойный персептрон с единственным выходом. Обратите внимание, что выходной слой использует логистическую (сигмоидальную) функцию активации, чтобы выдавать вероятность от 0.0 до 1.0. Если бы возможных действий было больше, тогда потребовалось бы предусмотреть по одному выходному нейрону на действие и взамен применять многопеременную функцию активации.
3. Наконец мы вызываем функцию `multinomial()` для выбора случайного действия. Указанная функция независимо выбирает одно целое число (или больше) с учетом логарифмической вероятности каждого целого числа. Например, ее вызов с массивом `[np.log(0.5), np.log(0.2), np.log(0.3)]` и `num_samples=5` приводит к выдаче пяти целых чисел, каждое из которых имеет вероятность 50% быть 0, вероятность 20% быть 1 и вероятность 30% быть 2. В данном случае нам нужно лишь одно целое число, представляющее предпринимаемое действие. Поскольку тензор `outputs` содержит только вероятность движения влево, мы должны сначала присоединить к нему `1 - outputs`, чтобы получить тензор, содержащий вероятности действий влево и вправо. Обратите внимание, что когда возможных действий больше двух, то нейронная сеть выдает по одной вероятности на действие, а потому необходимость в шаге присоединения отсутствует.

Итак, мы располагаем политикой в форме нейронной сети, которая будет принимать наблюдения и выдавать действия. Но как ее обучить?

Оценка действий: проблема присваивания коэффициентов доверия

Если бы мы знали, каким было наилучшее действие на каждом шаге, то могли бы обучать нейронную сеть традиционным способом, сводя к минимуму перекрестную энтропию между оценочной и целевой вероятностью. Тогда мы имели бы обыкновенное обучение с учителем. Тем не менее, при обучении с подкреплением единственным руководством, которое получает агент, являются награды, а награды обычно назначаются нечасто и с задержкой. Например, если агенту удастся сбалансировать дышло за 100 шагов, тогда как он может узнать, какие из предпринятых им 100 действий были хорошими, а какие плохими? Ему лишь известно, что дышло опустилось после последнего действия, но, несомненно, последнее действие не несет полной ответственности за такой результат. Это называется *проблемой присваивания коэффициентов доверия* (*credit assignment problem*): когда агент получает награду, то ему трудно узнать, какие действия следует благодарить (или винить) за нее. Подумайте о собаке, которая получает награду спустя несколько часов после хорошего поведения; поймет ли она, за что была награждена?

Общая стратегия решения упомянутой задачи заключается в оценке действия на основе суммы всех наград, которые поступили после него, обычно с применением *дисконтной ставки* (*discount rate*) r на каждом шаге. Скажем (рис. 16.6), если агент решает двигаться вправо три раза подряд и получает награду +10 после первого шага, 0 после второго и в заключение -50 после третьего шага, тогда при дисконтной ставке $r = 0.8$ первое действие будет иметь общую оценку $10 + r \times 0 + r^2 \times (-50) = -22$. Если дисконтная ставка близка к 0, то будущие награды окажутся менее значимыми в сравнении с недавними наградами. И наоборот, если дисконтная ставка близка к 1, тогда награды далеко в будущем станут почти такими же, как недавние награды. Типичная дисконтная ставка составляет 0.95 или 0.99. При дисконтной ставке 0.95 награды 13 шагов в направлении будущего представляют собой приблизительно половину недавних наград (т.к. $0.95^{13} \approx 0.5$), а при дисконтной ставке 0.99 награды 69 шагов в направлении будущего дадут половину недавних наград. В среде CartPole действия оказывают относительно краткосрочное влияние, поэтому выбор дисконтной ставки 0.95 выглядит разумным.

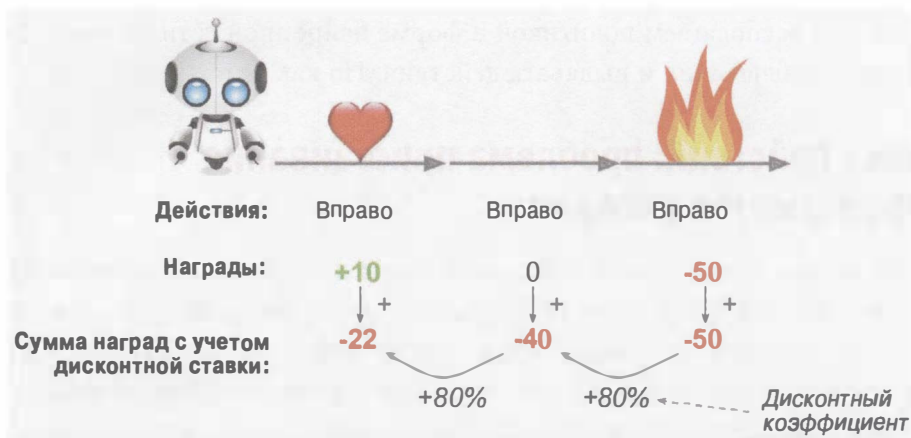


Рис. 16.6. Награды с учетом дисконтной ставки

Разумеется, за хорошим действием может следовать несколько плохих действий, которые приведут к быстрому опусканию дышла, в результате чего хорошее действие получит низкую оценку (подобно тому, как хороший актер может сыграть главную роль в отвратительном фильме). Однако если мы играем в игру достаточное количество раз, то в среднем хорошие действия будут иметь лучшие оценки, чем плохие действия. Итак, чтобы получить довольно надежные оценки для действий, мы должны прогнать много эпизодов и нормализовать все оценки действий (путем вычитания среднего и деления на стандартное отклонение). После этого мы можем корректно предполагать, что действия с отрицательными оценками были плохими, а действия с положительными оценками — хорошими. Прекрасно — теперь у нас есть способ оценки каждого действия, и мы готовы обучить первого агента, используя градиенты политики. Давайте посмотрим, как.

Градиенты политики

Как обсуждалось ранее, алгоритмы PG оптимизируют параметры политики, следуя по градиентам в сторону более высоких наград. Популярный класс алгоритмов PG, называемый *алгоритмами REINFORCE* (REward Increment = non-negative Factor × Offset Reinforcement × Characteristic Eligibility — прирост наград = неотрицательный множитель × подкрепление вознаграждения × характерный возможный выбор), был введен еще в 1992 году⁹ Рональдом Уильямсом.

⁹ “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning” (“Простые статистические алгоритмы, следующие по градиентам, для коннекционного обучения с подкреплением”), R. Williams (1992 год) (<https://goo.gl/tUe4Sh>).

Ниже описан его распространенный вариант.

1. Дайте возможность политике в форме нейронной сети сыграть в игру несколько раз и на каждом шаге вычисляйте градиенты, которые сделают выбранное действие даже более вероятным, но пока не применяйте градиенты.
2. После прогона нескольких эпизодов подсчитайте оценки каждого действия (используя метод, который описан в предыдущем разделе).
3. Если оценка действия положительная, то это означает, что действие было хорошим, и вы хотите применить вычисленные ранее градиенты, чтобы сделать выбор данного действия в будущем еще более вероятным. Если же оценка действия отрицательная, то это значит, что действие было плохим, и вы хотите применить противоположные градиенты, чтобы сделать его выбор в будущем чуть *менее* вероятным. Решение заключается просто в умножении каждого вектора-градиента на соответствующую оценку действия.
4. Вычислите все результирующие векторы-градиенты и воспользуйтесь ими для выполнения шага градиентного спуска.

Реализуем представленный выше алгоритм с применением TensorFlow. Мы обучим созданную ранее политику в форме нейронной сети, чтобы она научилась балансировать дышло в телеге. Начнем с завершения написанной в начале главы стадии построения, добавив целевую вероятность, функцию издержек и операцию обучения. Поскольку мы полагаем, что выбранное действие является наилучшим из возможных, целевая вероятность должна составлять 1.0, если выбранное действие представляет собой действие 0 (влево), и 0.0, если действие 1 (вправо):

```
y = 1. - tf.to_float(action)
```

Имея целевую вероятность, мы можем определить функцию издержек (перекрестную энтропию) и вычислить градиенты:

```
learning_rate = 0.01
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(labels=y,
                                                         logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

Обратите внимание, что мы вызываем метод оптимизатора `compute_gradients()`, а не `minimize()`. Причина в том, что мы хотим подстроить градиенты перед их применением¹⁰. Метод `compute_gradients()` возвращает список пар “вектор-градиент/переменная” (по одной паре на обучаемую переменную). Давайте поместим все градиенты в список, чтобы было более удобно получать их значения:

```
gradients = [grad for grad, variable in grads_and_vars]
```

Далее идет сложная часть. Во время стадии выполнения алгоритм будет прогонять политику и на каждом шаге оценивать такие тензоры градиентов, а также сохранять их значения. После некоторого количества эпизодов он будет подстраивать эти градиенты, как объяснялось ранее (т.е. умножать на оценки действий и нормализовать их), и вычислять среднее подстроенных градиентов. Затем ему понадобится передать результирующие градиенты оптимизатору, чтобы тот смог провести шаг оптимизации. Таким образом, нам необходим заполнитель на каждый вектор-градиент. Кроме того, мы должны создать операцию, которая применит обновленные градиенты. Для этого мы вызовем метод `apply_gradients()` оптимизатора, который принимает список пар “вектор-градиент/переменная”. Вместо исходных векторов-градиентов мы предоставим ему список, содержащий обновленные градиенты (т.е. те, которые передаются через заполнители градиентов):

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32,
                                         shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

Давайте отвлечемся и посмотрим на полную стадию построения:

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()
learning_rate = 0.01
```

¹⁰ Что-то подобное мы уже делали в главе 11, когда обсуждалось отсечение градиентов: сначала вычисляли градиенты, затем отсекали их и напоследок применяли отсеченные градиенты.

```

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = tf.layers.dense(X, n_hidden, activation=tf.nn.elu,
                          kernel_initializer=initializer)
logits = tf.layers.dense(hidden, n_outputs,
                          kernel_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32,
                                         shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()

```

Мы снова на стадии выполнения! Нам понадобится пара функции для вычисления итоговых наград с учетом дисконтной ставки, имея необработанные награды, и для нормализации результатов по множеству эпизодов:

```

def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards =
            rewards[step] + cumulative_rewards * discount_rate
        discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards, discount_rate)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std
            for discounted_rewards in all_discounted_rewards]

```

Проверим, все ли работает:

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
                                   discount_rate=0.8)
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([ 1.26665318,  1.0727777 ])]
```

Вызов `discount_rewards()` возвращает именно то, что мы ожидали (рис. 16.6). Вы можете удостовериться в том, что функция `discount_and_normalize_rewards()` действительно возвращает нормализованные оценки для каждого действия в обоих эпизодах. Обратите внимание, что первый эпизод был гораздо хуже второго, поэтому все его оценки отрицательные; все действия из первого эпизода будут считаться плохими и наоборот, все действия из второго эпизода будут считаться хорошими.

Теперь мы располагаем всем необходимым для того, чтобы обучить политику:

```
n_iterations = 250          # количество итераций обучения
n_max_steps = 1000         # максимальное число шагов на эпизод
n_games_per_update = 10    # обучение политики каждые 10 эпизодов
save_iterations = 10       # сохранение модели каждые 10 итераций обучения
discount_rate = 0.95

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []    # все последовательности необработанных
                             # наград для каждого эпизода
        all_gradients = []  # градиенты, сохраняемые на каждом шаге
                             # каждого эпизода
        for game in range(n_games_per_update):
            current_rewards = [] # все необработанные награды из
                                 # текущего эпизода
            current_gradients = [] # все градиенты из текущего эпизода
            obs = env.reset()
            for step in range(n_max_steps):
                action_val, gradients_val = sess.run(
                    [action, gradients],
                    feed_dict={X: obs.reshape(1, n_inputs)}) # одно
                                                                # наблюдение
                obs, reward, done, info = env.step(action_val[0][0])
                current_rewards.append(reward)
                current_gradients.append(gradients_val)
```

```

        if done:
            break
    all_rewards.append(current_rewards)
    all_gradients.append(current_gradients)

# В этой точке мы прогнали политику для 10 эпизодов и готовы
# к обновлению политики, используя описанный ранее алгоритм.
    all_rewards = discount_and_normalize_rewards(all_rewards,
                                                discount_rate)

    feed_dict = {}
    for var_index, grad_placeholder in enumerate(
        gradient_placeholders):
        # умножение градиентов на оценки действий и вычисление среднего
        mean_gradients = np.mean(
            [reward * all_gradients[game_index][step][var_index]
             for game_index, rewards in enumerate(all_rewards)
             for step, reward in enumerate(rewards)],
            axis=0)
        feed_dict[grad_placeholder] = mean_gradients
    sess.run(training_op, feed_dict=feed_dict)
    if iteration % save_iterations == 0:
        saver.save(sess, "./my_policy_net_pg.ckpt")

```

Каждая итерация обучения начинается с прогона политики для 10 эпизодов (с максимум 1 000 шагов на эпизод во избежание бесконечного выполнения). На каждом шаге мы также вычисляем градиенты, делая вид, что выбранное действие было наилучшим. После завершения 10 эпизодов мы вычисляем оценки действий с использованием функции `discount_and_normalize_rewards()`. Мы проходим через каждую обучаемую переменную по всем эпизодам и всем шагам, чтобы умножить каждый вектор-градиент на соответствующую ему оценку действия, и вычисляем среднее результирующих градиентов. Наконец, мы запускаем операцию обучения, передав ей усредненные градиенты (по одному на обучаемую переменную). Мы также сохраняем модель каждые 10 операций обучения.

Все готово! Приведенный выше код будет обучать политику в форме нейронной сети, и она успешно научится балансировать дышло в телеге (можете испытать ее в тетрадах Jupyter). Обратите внимание, что на самом деле существуют два пути к проигрышу игры агентом: либо дышло может дать слишком большой крен, либо телега может полностью исчезнуть с экрана. В случае 250 итераций политика обучается совсем неплохо балансировать дышло в телеге, но недостаточно хорошо избегать исчезновения с экрана. Проблему исправят дополнительные несколько сот итераций обучения.



Исследователи стараются отыскать алгоритмы, которые работают хорошо, даже когда агенту первоначально ничего не известно о среде. Тем не менее, если только вы не готовите статью, то должны снабдить агента как можно большим объемом априорных знаний, т.к. они значительно ускорят обучение. Например, вы могли бы добавить отрицательные награды, пропорциональные расстоянию от центра экрана и углу дышла. К тому же, если вы уже имеете достаточно успешную политику (скажем, жестко закодированную), тогда может возникнуть желание обучить нейронную сеть с целью ее имитации, прежде чем применять градиенты политики для ее улучшения.

Несмотря на относительную простоту, это довольно мощный алгоритм. Его можно использовать для решения более трудных задач, нежели балансирование дышла в телеге. В действительности система AlphaGo была основана на похожем алгоритме PG (с дополнительным *поиском по дереву методом Монте-Карло (Monte Carlo tree search)*, обсуждение которого выходит за рамки настоящей книги).

Существует еще одно популярное семейство алгоритмов. В то время как алгоритмы PG пытаются непосредственно оптимизировать политику с целью увеличения наград, рассматриваемые далее алгоритмы менее прямолинейны. Агент учится оценивать ожидаемую сумму будущих наград с учетом дисконтной ставки для каждого состояния или для каждого действия в каждом состоянии и затем на основе полученных знаний принимает решение о том, каким образом действовать. Чтобы понять эти алгоритмы, мы сначала должны представить *марковские процессы принятия решений (Markov Decision Process — MDP)*.

Марковские процессы принятия решений

В начале двадцатого века математик Андрей Марков изучал стохастические процессы без памяти, называемые цепями Маркова (Markov chain). Такой процесс имеет фиксированное число состояний и на каждом шаге случайным образом переходит из одного состояния в другое. Вероятность его перехода из состояния s в состояние s' является фиксированной и зависит только от пары (s, s') , но не от прошлых состояний (система не имеет памяти).

На рис. 16.7 показан пример цепи Маркова с четырьмя состояниями. Пусть процесс начинается в состоянии s_0 и есть 70%-ный шанс, что он останется в данном состоянии на следующем шаге. Со временем он обязательно покинет

это состояние и никогда не вернется обратно, поскольку ни одно из других состояний не указывает на s_0 . Если процесс перейдет в состояние s_1 , то затем, скорее всего, попадет в состояние s_2 (вероятность 90%) и немедленно возвратится в состояние s_1 (с вероятностью 100%). Он может несколько раз перемещаться между указанными двумя состояниями, но в итоге попадет в состояние s_3 и останется там навсегда (это *заключительное состояние*). Цепи Маркова могут обладать очень разными динамическими свойствами и широко применяются в термодинамике, химии, статистике и многих других областях.

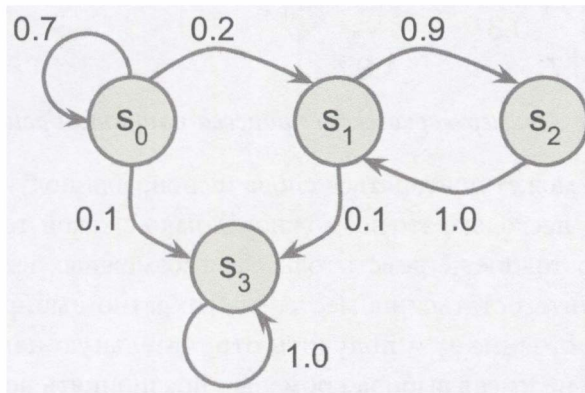


Рис. 16.7. Пример цепи Маркова

Марковские процессы принятия решений впервые были описаны в 1950-годах Ричардом Беллманом¹¹. Они напоминают цепи Маркова, но с одной уловкой: на каждом шаге агент может выбирать одно из нескольких возможных действий, а вероятности переходов зависят от выбранного действия. Кроме того, определенные переходы между состояниями возвращают награду (положительную или отрицательную), и цель агента в том, чтобы отыскать политику, которая будет доводить до максимума награды с течением времени.

Например, представленный на рис. 16.8 процесс MDP имеет три состояния и до трех возможных дискретных действий на каждом шаге. Он начинается с состояния s_0 , и агент может выбирать одно из действий a_0 , a_1 или a_2 . Если агент выберет действие a_1 , тогда процесс с уверенностью остается в состоянии s_0 без какой-либо награды. Затем при желании он может решить остаться там навсегда. Но в случае выбора действия a_0 есть вероятность 70% получить награду +10 и остаться в состоянии s_0 .

¹¹ “A Markovian Decision Process” (“Марковский процесс принятия решений”), Р. Беллман (1957 год) (<https://goo.gl/wZTVIN>).

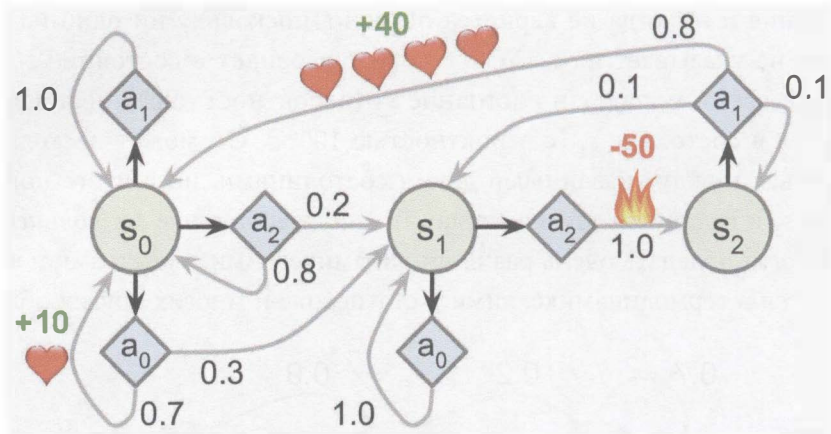


Рис. 16.8. Пример марковского процесса принятия решений

Далее попытка может повторяться снова и снова для получения настолько большой награды, насколько это возможно. Однако в какой-то момент он взамен переходит в состояние s_1 , где есть только два возможных действия: a_0 или a_2 . Агент может решить остаться на месте, многократно выбирая действие a_0 , или перейти в состояние s_2 и получить отрицательную награду -50 (ой!). В состоянии s_2 он не имеет выбора кроме как предпринять действие a_1 , которое с высокой вероятностью приведет его обратно в состояние s_0 с наградой $+40$ при переходе. Суть вы уловили. Глядя на этот процесс MDP, можете ли вы догадаться, какая стратегия со временем обеспечит наибольшую награду? В состоянии s_0 ясно, что лучшим выбором является действие a_0 , а в состоянии s_2 у агента нет выбора, кроме как принять действие a_1 , но в состоянии s_1 не очевидно, должен агент остаться на месте (a_0) либо пройти сквозь огонь (a_2).

Беллман нашел способ оценки *оптимальной ценности состояния* (*optimal state value*) для любого состояния s , обозначаемой $V^*(s)$. Она представляет собой сумму всех будущих наград с учетом дисконтной ставки, которые агент может ожидать в среднем после достижения состояния s , при условии, что он действует оптимально. Беллман показал, что если агент действует оптимально, то применимо *уравнение оптимальности Беллмана* (*Bellman optimality equation*), приведенное в уравнении 16.1. Это рекурсивное уравнение говорит о том, что если агент действует оптимально, тогда оптимальная ценность текущего состояния равна награде, которую он получит в среднем после принятия одного оптимального действия, плюс ожидаемая оптимальная ценность всех возможных следующих состояний, куда может привести данное действие.

Уравнение 16.1. Уравнение оптимальности Беллмана

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{для всех } s$$

- $T(s, a, s')$ — вероятность перехода из состояния s в состояние s' при условии, что агент выбрал действие a .
- $R(s, a, s')$ — награда, которую агент получает, когда переходит из состояния s в состояние s' при условии, что он выбрал действие a .
- γ — дисконтная ставка.

Уравнение 16.1 напрямую приводит к алгоритму, который может точно оценить оптимальную ценность каждого возможного состояния: сначала все оценки ценностей состояний инициализируются нулями, после чего они многократно обновляются с использованием алгоритма *итерации по ценностям* (*value iteration*), показанного в уравнении 16.2. Замечательный результат заключается в том, что при достаточном времени оценки гарантированно сойдутся в оптимальные ценности состояний, соответствующие оптимальной политике.

Уравнение 16.2. Алгоритм итерации по ценностям

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{для всех } s$$

- $V_k(s)$ — оценочная ценность состояния s на k -той итерации алгоритма.



Алгоритм итерации по ценностям является примером *динамического программирования*, которое разбивает задачу (в данном случае оценку потенциально бесконечной суммы будущих наград с учетом дисконтной ставки) на легко поддающиеся обработке подзадачи, допускающие итеративное решение (здесь нахождение действия, доводящего до максимума среднюю награду плюс ценность следующего состояния с учетом дисконтной ставки).

Знание оптимальных ценностей состояний может быть полезным, в частности для оценки политики, но оно не говорит агенту явно, что делать. К счастью, Беллман отыскал очень похожий алгоритм для оценки оптимальных *ценностей пар “состояние-действие”* (*state-action value*), обычно называемых *Q-ценностями* (*Q-value*). Оптимальная Q-ценность пары “состояние-

действие” (s, a), обозначаемая $Q^*(s, a)$, представляет собой сумму будущих наград с учетом дисконтной ставки, которые агент может ожидать в среднем после того, как он достигнет состояния s и выберет действие a , но перед тем, как увидит исход этого действия, при условии его оптимального поведения после действия.

Вот как работает прием: все оценки Q-ценностей изначально инициализируются нулями, а затем обновляются с применением алгоритма *итерации по Q-ценностям* (*Q-value iteration*), приведенного в уравнении 16.3.

Уравнение 16.3. Алгоритм итерации по Q-ценностям

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a') \right] \quad \text{для всех } (s, a)$$

После нахождения всех оптимальных Q-ценностей определение оптимальной политики, обозначаемой $\pi^*(s)$, становится тривиальным: когда агент пребывает в состоянии s , он должен выбрать действие с наивысшей Q-ценностью для этого состояния: $\pi^*(s) = \operatorname{argmax}_a Q^*(s, a)$.

Давайте применим такой алгоритм к процессу MDP, представленному на рис. 16.8. Прежде всего, необходимо определить процесс MDP:

```
nan = np.nan # представляет невозможные действия
T = np.array([ # форма = [s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
    [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
])
R = np.array([ # форма = [s, a, s']
    [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
    [[0.0, 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
    [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Теперь запустим алгоритм итерации по Q-ценностям:

```
Q = np.full((3, 3), -np.inf) # -inf для невозможных действий
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Начальная ценность = 0.0
                           # для всех возможных действий

discount_rate = 0.95
n_iterations = 100
```

```

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp]
                    + discount_rate * np.max(Q_prev[sp]))
                for sp in range(3)
            ])

```

Результирующие Q-ценности выглядят так:

```

>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
       [  1.11669335,        -inf,   1.17573546],
       [        -inf,  53.86946068,        -inf]])
>>> np.argmax(Q, axis=1) # оптимальное действие для каждого состояния
array([0, 2, 1])

```

В итоге мы получаем оптимальную политику для данного процесса MDP, когда используется дисконтная ставка 0.95: в состоянии s_0 выбрать действие a_0 , в состоянии s_1 — действие a_2 (пройти сквозь огонь!) и в состоянии s_2 — действие a_1 (единственное возможное действие). Интересно отметить, что если уменьшить дисконтную ставку до 0.9, то оптимальная политика изменится: в состоянии s_1 наилучшим действием становится a_0 (остаться на месте; не проходить сквозь огонь). Это имеет смысл, потому что если настоящее ценится гораздо больше будущего, тогда перспектива будущих наград не стоит немедленных страданий.

Обучение методом временных разностей и Q-обучение

Задачи обучения с подкреплением с дискретными действиями часто могут моделироваться как марковские процессы принятия решений, но агент изначально не имеет представления о вероятностях переходов (не знает $T(s, a, s')$) и о возможных наградах (не знает $R(s, a, s')$). Он должен испытать каждое состояние и каждый переход хотя бы раз, чтобы выяснить награды, и несколько раз, чтобы получить приемлемую оценку вероятностей переходов.

Алгоритм *обучения методом временных разностей* (*Temporal Difference (TD) learning*) очень похож на алгоритм итерации по ценностям, но он скорректирован для учета того факта, что агент располагает лишь частичным знанием процесса MDP. В общем случае мы предполагаем, что первоначаль-

но агенту известны только возможные состояния и действия. Агент применяет политику исследования — скажем, совершенно случайную политику — для изучения процесса MDP, и по мере его продвижения алгоритм обучения TD обновляет оценки ценностей состояний на основе переходов и наград, фактически полученных путем наблюдений (уравнение 16.4).

Уравнение 16.4. Алгоритм обучения TD

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

- α — скорость обучения (например, 0.01).



Обучение TD во многом похоже на стохастический градиентный спуск (SGD), в частности тем, что обрабатывает по одному образцу за раз. Подобно SGD алгоритм обучения TD может действительно сходиться только при постепенном снижении скорости обучения (иначе он продолжит совершать прыжки возле оптимума).

Для каждого состояния s алгоритм обучения TD просто отслеживает скользящее среднее непосредственных наград, которые агент получает при покидании данного состояния, плюс наград, которые он ожидает получить позже (при условии оптимального поведения).

Аналогично алгоритм *Q-обучения* (*Q-learning*) является адаптацией алгоритма итерации по Q-ценностям к ситуации, когда вероятности переходов и награды изначально не известны (уравнение 16.5).

Уравнение 16.5. Алгоритм Q-обучения

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha \left(r + \gamma \cdot \max_{a'} Q_k(s', a') \right)$$

Для каждой пары “состояние-переход” (s, a) этот алгоритм отслеживает скользящее среднее наград r , которые агент получает при покидании состояния s посредством действия a , плюс наград, которые он ожидает получить позже. Поскольку целевая политика должна действовать оптимально, мы получаем максимум оценок Q-ценностей для следующего состояния.

Вот как можно реализовать алгоритм Q-обучения:

```
learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000
```

```

s = 0 # начало в состоянии 0
Q = np.full((3, 3), -np.inf) # -inf для невозможных действий
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0 # Начальная ценность = 0.0
                           # для всех возможных действий

for iteration in range(n_iterations):
    a = np.random.choice(possible_actions[s]) # выбор действия
                                              # (случайным образом)
    sp = np.random.choice(range(3), p=T[s, a]) # выбор следующего
                                                # состояния, используя T[s, a]

    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = ((1 - learning_rate) * Q[s, a] +
               learning_rate * (reward + discount_rate * np.max(Q[sp])))
    s = sp # переход в следующее состояние

```

При достаточном числе итераций алгоритм Q-обучения будет сходиться к оптимальным Q-ценностям. Он называется алгоритмом *вне политики* (*off-policy*), т.к. политика, применяемая при обучении, не является той, которая используется во время выполнения. Слегка удивляет, что этот алгоритм способен узнать оптимальную политику, просто наблюдая за случайными действиями агента (вообразите себе обучение игре в гольф, когда наставником является нетрезвая обезьяна). Можем ли мы добиться лучшего?

Политики исследования

Разумеется, Q-обучение может работать, только если политика исследования анализирует процесс MDP в достаточной степени тщательно. Хотя чисто случайная политика в итоге гарантированно посетит каждое состояние и пройдет по каждому переходу много раз, это может потребовать чрезвычайно долгого времени. Следовательно, более удачным вариантом будет *ε-жадная политика* (*ε-greedy policy*): на каждом шаге она ведет себя случайным образом с вероятностью ϵ или жадным образом (выбирая действие с наивысшей Q-ценностью) с вероятностью $1 - \epsilon$. Преимущество ϵ -жадной политики (в сравнении с полностью случайной политикой) заключается в том, что она будет тратить все больше и больше времени на исследование интересных частей среды, поскольку оценки Q-ценностей становятся лучше и лучше, по-прежнему выделяя некоторое время на посещение неизвестных областей процесса MDP. Довольно часто начинают с высокого значения ϵ (скажем, 1.0) и постепенно его понижают (например, до 0.05).

В качестве альтернативы вместо того, чтобы полагаться на шанс исследования, другой подход предусматривает стимулирование политики исследования к испытанию действий, которые ранее не были опробованы часто. Как показано в уравнении 16.6, это можно реализовать в виде премии, добавляемой к оценкам Q-ценностей.

Уравнение 16.6. Q-обучение с применением функции исследования

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left(r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a')) \right)$$

- $N(s', a')$ подсчитывает, сколько раз действие a' было выбрано в состоянии s' .
- $f(q, n)$ — *функция исследования (exploration function)*, такая как $f(q, n) = q + K / (1 + n)$, где K — гиперпараметр любопытства, который измеряет, в какой степени агент увлекается неизвестным.

Приближенное Q-обучение и глубокое Q-обучение

Главная проблема с Q-обучением в том, что оно не масштабируется хорошо к крупным (или даже средним) процессам MDP, содержащим много состояний и действий. Обсудим попытку использования Q-обучения для того, чтобы научить агента играть в игру Ms. Pac-Man. Существует свыше 250 зерен, которые мисс Пэкмен может съесть. Каждое зерно может присутствовать или отсутствовать (т.е. уже быть съеденным). Таким образом, количество возможных состояний больше чем $2^{250} \approx 10^{75}$ (и это возможные состояния только зерен). Результирующее число превышает количество атомов в нашей галактике, так что абсолютно нет никакого способа учесть оценку для каждой отдельно взятой Q-ценности.

Решение заключается в том, чтобы отыскать функцию $Q_\theta(s, a)$, которая приближенно вычисляет Q-ценность пары “состояние-действие” (s, a) с применением управляемого количества параметров (заданных вектором параметров θ). Подход называется *приближенным Q-обучением (approximate Q-learning)*. В течение многих лет для оценки Q-ценностей рекомендовалось использовать линейные комбинации созданных вручную признаков, выделенных из состояния (скажем, расстояние до ближайших призраков, направления их передвижения и т.д.). Тем не менее, система DeepMind показала, что применение глубоких нейронных сетей может дать гораздо лучшие результаты, особенно в случае сложных задач, и какое-либо конструирование

признаков не требуется. Сеть DNN, используемая для оценки Q-ценностей, называется *глубокой Q-сетью (DQN)*, а применение сети DQN для приближенного Q-обучения — *глубоким Q-обучением (deep Q-learning)*.

Но как можно обучить сеть DQN? Рассмотрим приближенную Q-ценность, вычисленную сетью DQN для заданной пары “состояние-действие” (s, a). Благодаря Беллману нам известно, что эта приближенная Q-ценность должна быть как можно ближе к награде r , которую мы фактически наблюдаем после выполнения действия a в состоянии s , плюс ценность с учетом дисконтной ставки от оптимальной игры, начиная с данного момента. Чтобы оценить такую будущую ценность с учетом дисконтной ставки, мы можем просто прогнать сеть DQN на следующем состоянии s' для всех возможных действий a' . Мы получим приближенную будущую Q-ценность для каждого возможного действия. Затем мы выбираем самую высокую (т.к. предполагаем оптимальную игру), учитываем дисконтную ставку и в результате имеем оценку будущей ценности с учетом дисконтной ставки. Суммируя награду r и оценку будущей ценности с учетом дисконтной ставки, мы получаем целевую Q-ценность $y(s, a)$ для пары “состояние-действие” (s, a), как показано в уравнении 16.7.

Уравнение 16.7. Целевая Q-ценность

$$y(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s', a')$$

Располагая целевой Q-ценностью, мы можем запустить итерацию обучения с использованием любого алгоритма градиентного спуска. В частности, обычно мы будем пытаться свести к минимуму квадратичную ошибку между оценочной Q-ценностью и целевой Q-ценностью. Вот и весь базовый алгоритм глубокого Q-обучения!

Однако в алгоритм DQN системы DeepMind были внесены две ключевых модификации.

- Вместо обучения сети DQN на основе самого последнего опыта алгоритм системы DeepMind хранит данные опыта в крупной *памяти воспроизведения* и на каждой итерации обучения выбирает из нее случайный обучающий пакет. Это содействует сокращению корреляции между опытами в обучающем пакете, что чрезвычайно помогает обучению.
- Алгоритм системы DeepMind применяет не одну, а две сети DQN: первая, называемая *динамической сетью DQN*, занимается игрой и изуче-

нием на каждой итерации обучения. Вторая, называемая *целевой сетью DQN*, используется только для вычисления целевых Q-ценностей (см. уравнение 16.7). Через регулярные промежутки времени веса динамической сети копируются в целевую сеть. Система DeepMind продемонстрировала, что такое изменение поразительно улучшает производительность алгоритма. На самом деле без него была бы единственная сеть, которая устанавливает собственные цели и пытается достичь их, что несколько напоминает собаку, преследующую свой хвост. В итоге могут образоваться петли обратной связи, которые сделают сеть нестабильной (она может расходиться, колебаться, замораживаться и т.д.). Наличие двух сетей помогает ослабить такие петли обратной связи, приводя к стабилизации процесса обучения.

В оставшейся части главы мы будем применять алгоритм сети DQN из DeepMind для того, чтобы научить агента играть в игру Ms. Pac-Man, почти как это делала система DeepMind в 2013 году. Код можно легко подстроить для обучения довольно хорошо играть в большинство игр Atari при условии, что он обучается достаточно долго (дни или недели в зависимости от имеющегося оборудования). В большинстве игр-боевиков код способен достичь сверхчеловеческого искусства, но он не так хорош в играх с длинными сюжетными линиями.

Обучение играть в игру Ms. Pac-Man с использованием алгоритма сети DQN

Поскольку мы будем применять среду Atari, сначала потребуется установить зависимости Atari из OpenAI Gym. Одновременно мы также установим зависимости для других сред OpenAI Gym, с которыми может возникнуть желание поработать. На машине с macOS (при условии предшествующей установки Homebrew (<http://brew.sh/>)) необходимо ввести такую команду:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

На машине с Ubuntu введите следующую команду (заменяя `python3` на `python`, если используется Python 2):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev  
libjpeg-dev xvfb libav-tools xorg-dev python3-opengl  
libboost-all-dev libsdl2-dev swig
```

Затем установите дополнительные модули Python (если вы применяете среду `virtualenv`, то сначала активируйте ее):

```
$ pip3 install --upgrade 'gym[all]'
```

Если все прошло успешно, тогда вы должны быть в состоянии создать среду игры `Ms. Pac-Man`:

```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape # [высота, ширина, каналы]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

Как видите, доступно девять дискретных действий, которые соответствуют девяти возможным положениям джойстика (центральное, вверх, вправо, влево, вниз, вправо вверх и т.д.), а наблюдениями являются просто снимки экрана Atari (рис. 16.9, слева), представленные как трехмерные массивы `NumPy`. Поскольку изображения довольно велики, мы создадим небольшую функцию предварительной обработки, которая будет обрезать изображение и уменьшать его до 88×80 пикселей, преобразовывать в оттенки серого и усиливать контрастность значка мисс Пэкмен. Такой прием сократит объем вычислений, требуемых сетью DQN, и ускорит обучение.

```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # обрезать и уменьшить размер
    img = img.mean(axis=2)   # преобразовать в оттенки серого
    img[img==mspacman_color] = 0 # усилить контрастность
    img = (img - 128) / 128 - 1 # нормализовать от -1. до 1.
    return img.reshape(88, 80, 1)
```

Результат предварительной обработки показан на рис. 16.9 (справа).

Далее мы создадим сеть DQN. Она могла бы получать на входе пару “состояние-действие” (s, a) и выдавать оценку соответствующей Q -ценности $Q(s, a)$, но из-за того, что действия дискретны, более удобно и эффективно использовать нейронную сеть, которая принимает на входе только состояние s и выдает по одной оценке Q -ценности на действие. Сеть DQN будет состоять из трех сверточных слоев, за которыми следуют два полносвязных слоя, включая выходной слой (рис. 16.10).

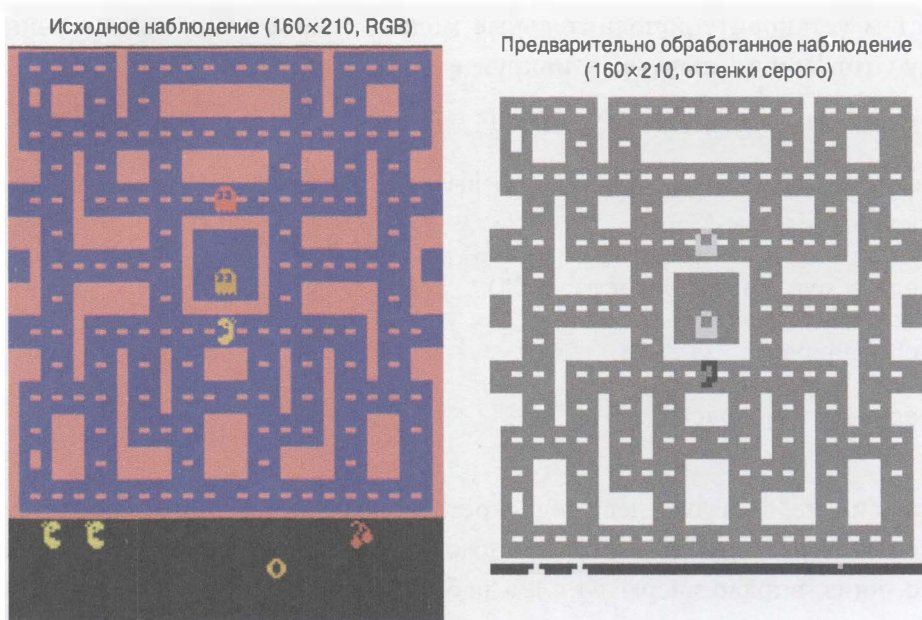


Рис. 16.9. Наблюдение в игре Ms. Pac-Мэн, исходное (слева) и после предварительной обработки (справа)

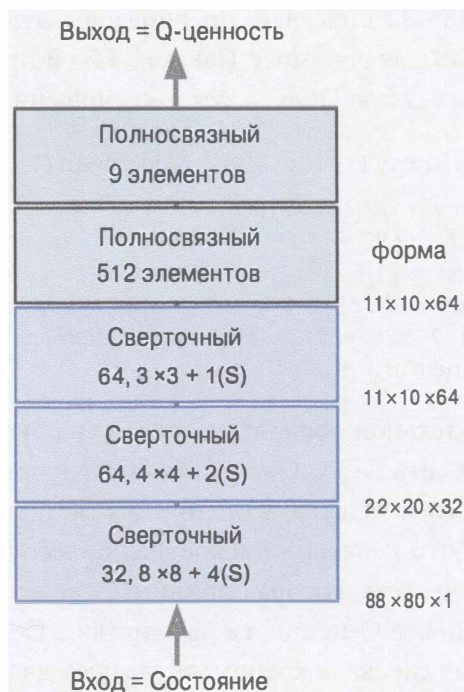


Рис. 16.10. Глубокая Q-сеть для игры в Ms. Pac-Мэн

Как обсуждалось ранее, алгоритм обучения DQN, разработанный DeepMind, требует двух сетей DQN с одной и той же архитектурой (но разными параметрами): динамическая сеть DQN будет учиться управлять мисс Пэкмен, а с применением целевой сети DQN будут вычисляться целевые Q-ценности, предназначенные для обучения динамической сети DQN. Через регулярные промежутки времени мы будем копировать динамическую сеть DQN в целевую сеть DQN, заменяя ее параметры. Поскольку нам нужны две сети DQN с той же самой архитектурой, для их построения мы создадим функцию `q_network()`:

```
input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"] * 3
conv_activation = [tf.nn.relu] * 3
n_hidden_in = 64 * 11 * 10 # conv3 имеет 64 карты, каждая 11x10
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n # доступны 9 дискретных действий
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, name):
    prev_layer = X_state
    with tf.variable_scope(name) as scope:
        for n_maps, kernel_size, strides, padding, activation in zip(
            conv_n_maps, conv_kernel_sizes, conv_strides,
            conv_paddings, conv_activation):
            prev_layer = tf.layers.conv2d(
                prev_layer, filters=n_maps, kernel_size=kernel_size,
                strides=strides, padding=padding, activation=activation,
                kernel_initializer=initializer)
        last_conv_layer_flat = tf.reshape(prev_layer,
                                           shape=[-1, n_hidden_in])
        hidden = tf.layers.dense(last_conv_layer_flat, n_hidden,
                                  activation=hidden_activation,
                                  kernel_initializer=initializer)
        outputs = tf.layers.dense(hidden, n_outputs,
                                    kernel_initializer=initializer)
    trainable_vars =
        tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                           scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                              for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

В первой части кода определяются гиперпараметры архитектуры DQN. Затем определяется функция `q_network()`, предназначенная для создания сетей DQN, которая принимает на входе состояние среды `X_state` и имя пространства переменной. Обратите внимание, что для представления состояния среды мы будем использовать только одно наблюдение, т.к. скрытого состояния почти нет (кроме мерцающих объектов и направлений движения призраков).



Игры вроде Pong или Breakout содержат перемещающийся мяч, направление движения и скорость которого не могут быть определены с помощью единственного наблюдения, поэтому они потребуют комбинирования последних нескольких наблюдений в состояние среды. Для этого можно создать изображение с одним каналом для каждого из последних нескольких наблюдений. В качестве альтернативы можно было бы объединить последние несколько наблюдений в единственное одноканальное изображение, скажем, путем вычисления максимума по упомянутым наблюдениям (после затемнения более старых наблюдений, чтобы на финальном изображении ясно просматривалось направление хода времени).

Словарь `trainable_vars_by_name` накапливает все обучаемые переменные данной сети DQN. Он будет полезен очень скоро, когда мы создадим операции для копирования динамической сети DQN в целевую сеть DQN. Ключами словаря являются имена переменных без части префикса, которая просто соответствует имени пространства. Вот как он выглядит:

```
>>> trainable_vars_by_name
{'/conv2d/bias:0': <tf.Variable... shape=(32,) dtype=float32_ref>,
 '/conv2d/kernel:0': <tf.Variable... shape=(8, 8, 1, 32)
                        dtype=float32_ref>,
 '/conv2d_1/bias:0': <tf.Variable... shape=(64,) dtype=float32_ref>,
 '/conv2d_1/kernel:0': <tf.Variable... shape=(4, 4, 32, 64)
                        dtype=float32_ref>,
 '/conv2d_2/bias:0': <tf.Variable... shape=(64,) dtype=float32_ref>,
 '/conv2d_2/kernel:0': <tf.Variable... shape=(3, 3, 64, 64)
                        dtype=float32_ref>,
 '/dense/bias:0': <tf.Variable... shape=(512,) dtype=float32_ref>,
 '/dense/kernel:0': <tf.Variable... shape=(7040, 512) dtype=float32_ref>,
 '/dense_1/bias:0': <tf.Variable... shape=(9,) dtype=float32_ref>,
 '/dense_1/kernel:0': <tf.Variable... shape=(512, 9) dtype=float32_ref>}
```

Теперь создадим входной заполнитель, две сети и операцию для копирования динамической сети DQN в целевую сеть DQN:

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height,
                                           input_width, input_channels])
online_q_values, online_vars = q_network(X_state,
                                         name="q_networks/online")
target_q_values, target_vars = q_network(X_state,
                                         name="q_networks/target")

copy_ops = [target_var.assign(online_vars[var_name])
            for var_name, target_var in target_vars.items()]
copy_online_to_target = tf.group(*copy_ops)
```

Давайте ненадолго отвлечемся: мы располагаем двумя сетями DQN, которые обе способны принимать на входе состояние среды (в данном примере одиночное предварительно обработанное наблюдение) и выдавать оценочную Q-ценность для каждого возможного действия в этом состоянии. Вдобавок у нас есть операция по имени `copy_online_to_target`, предназначенная для копирования значений всех обучаемых переменных динамической сети DQN в соответствующие переменные целевой сети DQN. Мы применяем функцию `tf.group()` из TensorFlow для группирования всех операций присваивания в единственную удобную операцию.

Теперь добавим операции обучения динамической сети DQN. Первым делом нам необходимо иметь возможность вычисления ее прогнозированной Q-ценности для каждой пары “состояние-действие” в пакете из памяти. Поскольку сеть DQN выдает по одной Q-ценности для каждого возможного действия, нам нужно сохранить только Q-ценность, которая соответствует фактически выполненному действию. Для этого мы преобразуем действие в вектор в унитарном коде (вспомните, что такой вектор заполнен нулями кроме единицы по *i*-тому индексу) и умножим его на Q-ценности: в результате обнулится все Q-ценности за исключением одной, которая соответствует запомненному действию. Затем мы просто суммируем по первой оси, чтобы получить только желаемый прогноз Q-ценности для каждого запомненного действия.

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(target_q_values * tf.one_hot(X_action,
                                                    n_outputs), axis=1, keep_dims=True)
```

Далее мы создаем заполнитель `y`, который будем использовать для представления целевых Q-ценностей, и вычисляем потерю: мы применяем квадратичную ошибку, когда она меньше 1.0, и удвоенную абсолютную ошибку, когда квадратичная ошибка превышает 1.0. Другими словами, потеря является квадратичной для небольших ошибок и линейной для крупных ошибок. Такой прием уменьшает влияние крупных ошибок и помогает стабилизировать обучение.

```
y = tf.placeholder(tf.float32, shape=[None, 1])
error = tf.abs(y - q_value)
clipped_error = tf.clip_by_value(error, 0.0, 1.0)
linear_error = 2 * (error - clipped_error)
loss = tf.reduce_mean(tf.square(clipped_error) + linear_error)
```

Наконец, мы создаем оптимизатор на основе ускоренного градиента Нестерова, чтобы свести к минимуму потерю. Мы также создаем необучаемую переменную по имени `global_step` для отслеживания шага обучения. О ее инкрементировании позаботится операция обучения. Напоследок мы создаем обычную операцию `init` и объект `Saver`.

```
learning_rate = 0.001
momentum = 0.95

global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum,
                                         use_nesterov=True)
training_op = optimizer.minimize(loss, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Так выглядит стадия построения. Прежде чем переходить к стадии выполнения нам необходимо реализовать пару инструментов. Начнем с реализации памяти воспроизведения. Мы будем использовать список `deque`, т.к. он очень эффективен в плане заталкивания элементов в очередь и выталкивания самого старого элемента, когда достигнут максимальный размер памяти. Мы также напишем небольшую функцию для случайной выборки пакета опытов из памяти воспроизведения. Каждый опыт будет 5-элементным кортежем (состояние, действие, награда, следующее состояние, продолжение), где элемент “продолжение” будет равен 0.0, когда игра завершена, или 1.0 в противном случае.


```

from collections import deque

replay_memory_size = 500000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = np.random.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # состояние, действие, награда,
                                # следующее состояние, продолжение

    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],
            cols[4].reshape(-1, 1))

```

Следующим нам понадобится агент для наблюдения за игрой. Мы будем применять ϵ -жадную политику и постепенно уменьшать ϵ с 1.0 до 0.1 за два миллиона шагов обучения:

```

eps_min = 0.1
eps_max = 1.0
eps_decay_steps = 2000000

def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max - eps_min)
                  * step / eps_decay_steps)

    if np.random.rand() < epsilon:
        return np.random.randint(n_outputs) # случайное действие
    else:
        return np.argmax(q_values)           # оптимальное действие

```

Итак, у нас есть все необходимое, чтобы начать обучение. Стадия выполнения не содержит ничего особо сложного, но она довольно велика, поэтому сделайте глубокий вдох. Готовы? Тогда поехали! Первым делом установим несколько параметров:

```

n_steps = 4000000 # общее количество шагов обучения
training_start = 10000 # начать обучение после 10 000 итераций игры
training_interval = 4 # запускать шаг обучения каждые 4 итерации игры
save_steps = 1000 # сохранять модель каждую 1 000 шагов обучения
copy_steps = 10000 # копировать динамическую сеть DQN в целевую
                  # сеть DQN каждые 10 000 шагов обучения

discount_rate = 0.99
skip_start = 90 # пропускать начало каждой игры
               # (это просто время ожидания)

```



```

batch_size = 50
iteration = 0 # итерации игры
checkpoint_path = "./my_dqn.ckpt"
done = True # среда нуждается в сбросе

```

Затем откроем сеанс и запустим главный цикл обучения:

```

with tf.Session() as sess:
    if os.path.isfile(checkpoint_path + ".index"):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
        copy_online_to_target.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        if done: # игра окончена, начать заново
            obs = env.reset()
            for skip in range(skip_start): #пропускать начало каждой игры
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

            # Динамическая сеть DQN оценивает, что делать
            q_values = online_q_values.eval(feed_dict={X_state: [state]})
            action = epsilon_greedy(q_values, step)

            # Динамическая сеть DQN играет
            obs, reward, done, info = env.step(action)
            next_state = preprocess_observation(obs)

            # Запомнить происшедшее только что
            replay_memory.append((state, action, reward, next_state,
                                1.0 - done))

            state = next_state

        if iteration < training_start
        or iteration % training_interval != 0:
            continue # обучать только после стадии разогрева
                    # и через регулярные интервалы

        # Выбрать запомненные данные и использовать
        # целевую сеть DQN для выработки целевой Q-ценности
        X_state_val, X_action_val, rewards, X_next_state_val,
        continues = (sample_memories(batch_size))
        next_q_values = target_q_values.eval(
            feed_dict={X_state: X_next_state_val})
        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)

```

```

y_val=rewards + continues * discount_rate * max_next_q_values
# Обучить динамическую сеть DQN
training_op.run(feed_dict={X_state: X_state_val,
                           X_action: X_action_val, y: y_val})

# Регулярно копировать динамическую сеть DQN в целевую сеть DQN
if step % copy_steps == 0:
    copy_online_to_target.run()

# И регулярно сохранять
if step % save_steps == 0:
    saver.save(sess, checkpoint_path)

```

Прежде всего, мы восстанавливаем модель, если существует файл контрольной точки, или в противном случае просто инициализируем переменные обычным образом, после чего копируем динамическую сеть DQN в целевую сеть DQN. Затем запускается главный цикл, где переменная `iteration` подсчитывает общее количество шагов игры, которые прошли с момента начала программы, а переменная `step` подсчитывает общее количество шагов обучения с момента запуска процесса обучения (если восстанавливалась контрольная точка, тогда благодаря переменной `global_step` также было восстановлено общее количество шагов обучения).

Далее код сбрасывает игру и пропускает первые неинтересные шаги игры, в течение которых ничего не происходит. Потом динамическая сеть DQN оценивает, что делать, играет в игру и ее опыт запоминается в памяти воспроизведения. Через регулярные промежутки времени (после стадии разогрева) динамическая сеть DQN проходит шаг обучения. Сначала мы выбираем пакет из памяти и запрашиваем у целевой сети DQN оценку Q-ценностей всех возможных действий для “следующего состояния” каждого запомненного опыта, после чего применяем уравнение 16.7, чтобы вычислить переменную `y_val`, содержащую целевую Q-ценность для каждой пары “состояние-действие”.

Единственная сложность здесь в том, что мы должны умножить вектор `max_next_q_values` на вектор `continues`, чтобы обнулить будущие Q-ценности, которые соответствуют запомненным опытам, где игра закончилась. Далее мы запускаем операцию обучения, чтобы усовершенствовать способность динамической сети DQN прогнозировать Q-ценности. Наконец, через регулярные интервалы мы копируем динамическую сеть DQN в целевую сеть DQN и сохраняем модель.



К сожалению, обучение проходит очень медленно: в случае использования для обучения ноутбука пройдет несколько дней, прежде чем мисс Пэкмен сумеет делать хоть какие-то успехи. Вы можете вычерчивать кривые обучения, которые, скажем, измеряют средние награды на игру, или вычислять максимальную Q-ценность, оцененную динамической сетью DQN на каждом шаге игры, и отслеживать среднее этих максимальных Q-ценностей для каждой игры. Вы заметите, что эти кривые крайне зашумленные. В некоторых точках видимый прогресс может отсутствовать очень долгое время, пока неожиданно агент не научится выживать в течение разумного временного промежутка. Как упоминалось ранее, одно из решений предусматривает ввод в модель как можно большего объема априорных знаний (например, через предварительную обработку, награды и т.д.), и вы также можете попробовать ускорить модель, сначала обучив ее имитировать базовую стратегию. В любом случае обучение с подкреплением все еще требует довольно большого терпения, но конечный результат будет весьма захватывающим.

Упражнения

1. Каким образом вы определили бы обучение с подкреплением? Чем оно отличается от обыкновенного обучения с учителем или без учителя?
2. Можете ли вы придумать три вероятных приложения RL, которые не были упомянуты в настоящей главе? Что является средой в каждом из них? Что собой представляет агент? Что собой представляют возможные действия? Что собой представляют награды?
3. Что такое дисконтная ставка? Может ли оптимальная политика измениться, если модифицировать дисконтную ставку?
4. Как бы вы измеряли производительность агента обучения с подкреплением?
5. В чем заключается проблема присваивания коэффициентов доверия? Когда она возникает? Как ее можно смягчить?
6. В чем смысл применения памяти воспроизведения?
7. Что собой представляет алгоритм RL вне политики?
8. Воспользуйтесь градиентами политики, чтобы заняться средой `BipedalWalker-v2` из `OpenAI Gym`.

9. Примените алгоритм DQN, чтобы обучить агента играть в известную игру Atari под названием *Pong* (*Pong-v0* в OpenAI Gym). Предостережение: для выяснения направления и скорости мяча индивидуального наблюдения недостаточно.
10. При наличии свободной суммы около \$100 вы можете купить Raspberry Pi 3 и ряд недорогих робототехнических компонентов, установить TensorFlow на Pi и немного позабавиться! Например, почитайте веселую статью Лукаса Бивальда (<https://goo.gl/Eu5u28>) либо взгляните на GoPiGo или BrickPi. Почему бы не попытаться построить реалистичное дышло телеги, обучив робота с использованием градиентов политики? Или построить роботизированного паука, который учится ходить; давайте ему награды всякий раз, когда он приближается к какой-то цели (вам понадобятся датчики для измерения расстояния к цели). Здесь вы ограничены лишь собственным воображением.

Решения приведенных упражнений доступны в приложении А.

Спасибо!

Прежде чем закончить последнюю главу этой книги, я хотел бы поблагодарить вас за то, что вы дочитали ее вплоть до последнего абзаца. Я искренне надеюсь, что вы получили столько же удовольствия от чтения книги, сколько было у меня во время ее написания, и что она принесет пользу вашим проектам, большую или малую.

Если вы обнаружите какие-то ошибки, пожалуйста, уведомьте о них. В целом мне нравится знать, что вы думаете, поэтому не стесняйтесь связываться со мной через издательство O'Reilly, проект [ageron/handson-ml](#) на GitHub или посредством Твиттера [@aureliengeron](#).

На пути вперед мой лучший совет вам — постоянно совершенствуйте свои навыки: попытайтесь выполнить все упражнения, если вы этого еще сделали, поработайте с тетрадами Jupyter, присоединитесь к Kaggle.com или какому-то другому сообществу ML, пройдите курсы обучения ML, читайте статьи, участвуйте в конференциях, встречайтесь с экспертами. Можете также изучить темы, которые в настоящей книге не раскрывались, в том числе системы рекомендаций, алгоритмы кластеризации, алгоритмы обнаружения аномалий и генетические алгоритмы.

Моя огромная надежда заключается в том, что книга вдохновит вас на создание замечательного приложения ML, которое принесет пользу всем нам! Что это будет?

Орельен Жерон, 26 ноября 2016 года

Решения упражнений



Решения упражнений, связанных с написанием кода, доступны в онлайн-овых тетрадах Jupyter по адресу <https://github.com/ageron/handson-ml>.

Глава 1. Введение в машинное обучение

1. Машинное обучение — это построение систем, которые могут учиться на основе данных. Обучение означает изменение работы в лучшую сторону на некоторой задаче при наличии определенной меры производительности.
2. Машинное обучение прекрасно себя ведет в сложных задачах, для которых мы не имеем алгоритмического решения, чтобы заменить длинные списки вручную подстраиваемых правил, строить системы, адаптирующиеся к меняющимся средам, и в итоге помочь учиться людям (примером может служить интеллектуальный или глубинный анализ данных).
3. Помеченный обучающий набор — это обучающий набор, который содержит желательное решение (также известное как метка) для каждого образца.
4. Двумя наиболее распространенными задачами обучения с учителем являются регрессия и классификация.
5. Распространенные задачи обучения без учителя включают кластеризацию, визуализацию, понижение размерности и обучение ассоциативным правилам.

6. Если вы хотите, чтобы робот проходил по разнообразным неизведанным территориям, то вероятно наилучшим будет обучение с подкреплением, поскольку именно такой тип задач оно обычно решает. Возможно, задачу удалось бы выразить как задачу обучения с учителем или задачу частичного обучения, но это было бы менее естественно.
7. Если вы не знаете, как определять группы, тогда можете воспользоваться каким-то алгоритмом кластеризации (обучение без учителя) для сегментирования своих заказчиков в кластеры похожих заказчиков. Однако если вам известно, какие группы желательно иметь, то можете передать множество образцов каждой группы алгоритму классификации (обучение с учителем), который классифицирует заказчиков в эти группы.
8. Выявление спама является типичной задачей обучения с учителем: алгоритму передается множество почтовых сообщений вместе с их метками (спам или не спам).
9. Система динамического обучения может обучать последовательно в противоположность системе пакетного обучения. Это делает ее способной к быстрой адаптации к изменяющимся данным и к автономным системам, а также к обучению на сверхбольших объемах данных.
10. Внешние алгоритмы могут обрабатывать громадные количества данных, которые не умещаются в основную память компьютера. Алгоритм внешнего обучения расщепляет данные на мини-пакеты и применяет приемы динамического обучения, чтобы учиться на таких мини-пакетах.
11. Система обучения на основе образцов заучивает обучающие данные на память; затем при получении нового образца она использует измерение сходства, чтобы отыскать наиболее похожие изученные образцы и применять их для выработки прогнозов.
12. Модель имеет один или более параметров модели, которые определяют, *что* она будет прогнозировать, получив новый образец (скажем, наклон линейной модели). Алгоритм обучения пытается найти оптимальные значения для этих параметров, чтобы модель хорошо обобщалась на новые образцы. Гиперпараметр — это параметр самого алгоритма обучения, а не модели (например, величина регуляризации, подлежащей применению).

13. Алгоритмы обучения на основе моделей ищут оптимальные значения для параметров модели, обеспечивающие хорошее обобщение модели на новые образцы. Обычно мы обучаем такие системы, сводя к минимуму функцию издержек, которая измеряет, насколько плохо система вырабатывает прогнозы на обучающих данных, плюс штраф за сложность модели, если модель регуляризирована. Для выработки прогнозов мы передаем признаки нового образца в прогнозирующую функцию модели, которая использует значения параметров, найденные алгоритмом обучения.
14. Основные проблемы в машинном обучении — это нехватка данных, плохое качество данных, нерепрезентативные данные, неинформативные признаки, чересчур простые модели, которые недообучены на обучающих данных, и чрезмерно сложные модели, которые переобучены обучающими данными.
15. Если модель прекрасно работает с обучающими данными, но плохо обобщается на новые образцы, тогда вероятно она переобучена обучающими данными (или вам исключительно повезло при обучении на обучающих данных). Возможными решениями проблемы переобучения являются получение большего количества данных, упрощение модели (выбор более простого алгоритма, уменьшение количества применяемых параметров или признаков либо регуляризация модели) или сокращение уровня шума в обучающих данных.
16. Испытательный набор используется для оценки ошибки обобщения, которую модель будет допускать на новых образцах, перед передачей модели в производственную среду.
17. Проверочный набор применяется для сравнения моделей. Он делает возможным выбор лучшей модели и подстройку гиперпараметров.
18. При подстройке гиперпараметров с использованием испытательного набора возникает риск переобучения испытательным набором, и измеряемая ошибка обобщения будет оптимистичной (вы можете получить модель, которая работает хуже, чем ожидалось).
19. Перекрестная проверка представляет собой прием, который дает возможность сравнивать модели (для выбора лучшей модели и подстройки гиперпараметров), не требуя отдельного проверочного набора. Это экономит драгоценные обучающие данные.

Глава 2. Полный проект машинного обучения

См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml>.

Глава 3. Классификация

См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml>.

Глава 4. Обучение моделей

1. При наличии обучающего набора со многими миллионами признаков вы можете применять стохастический градиентный спуск или мини-пакетный градиентный спуск и возможно пакетный градиентный спуск, если обучающий набор умещается в памяти. Но вы не можете использовать нормальное уравнение, потому что с увеличением количества признаков вычислительная сложность быстро растет (более чем квадратично).
2. Если признаки в вашем обучающем наборе имеют очень разные масштабы, тогда функция издержек будет иметь форму вытянутой чаши, поэтому алгоритмы градиентного спуска потребуют длительного времени на сходжение. Чтобы решить проблему, перед обучением модели вы должны масштабировать данные. Обратите внимание, что нормальное уравнение будет хорошо работать и без масштабирования. Кроме того, если признаки не масштабированы, то регуляризированные модели могут сходиться в субоптимальное решение: на самом деле, поскольку регуляризация штрафует крупные веса, признаки с меньшими значениями будут игнорироваться в сравнении с признаками, имеющими более высокие значения.
3. Градиентный спуск не может застрять в локальном минимуме при обучении логистической регрессионной модели, потому что функция издержек является выпуклой¹.

¹ Прямая линия между любыми двумя точками, выбранными на кривой, никогда не пересечет кривую.

4. Если задача оптимизации выпуклая (такая как линейная регрессия или логистическая регрессия) и скорость обучения не очень высока, тогда все алгоритмы градиентного спуска приведут в глобальный оптимум и в итоге выработают довольно похожие модели. Тем не менее, если только вы постепенно не снижаете скорость обучения, то стохастический градиентный спуск и мини-пакетный градиентный спуск никогда по-настоящему не сойдутся; взамен они продолжают прыжки вперед и назад возле глобального оптимума. Это означает, что даже если вы позволите им выполняться в течение очень долгого времени, упомянутые алгоритмы градиентного спуска будут производить слегка отличающиеся модели.
5. Если ошибка проверки последовательно растет после каждой эпохи, то возможно причина в том, что скорость обучения слишком высока и алгоритм расходится. Если ошибка обучения также увеличивается, тогда это ясно указывает на проблему, и вы должны снизить скорость обучения. Однако если ошибка обучения не растет, тогда модель переобучается обучающим набором и вы должны остановить обучение.
6. Из-за своей случайной природы ни стохастический градиентный спуск, ни мини-пакетный градиентный спуск не гарантируют продвижения на каждой отдельно взятой итерации обучения. Следовательно, если вы немедленно прекратите обучение, когда ошибка проверки возрастает, то можете остановиться слишком рано, до достижения оптимума. Более удачное решение предусматривает сохранение модели через регулярные интервалы, и если она не улучшается в течение долгого времени (это означает, что модель возможно никогда не побьет рекорд), тогда вы можете возвратиться к наилучшей сохраненной модели.
7. Стохастический градиентный спуск имеет самую быструю итерацию обучения, поскольку он рассматривает только один обучающий образец за раз, так что в общем случае он первым достигнет окрестностей глобального оптимума (или мини-пакетный градиентный спуск с очень малым размером мини-пакета). Тем не менее, лишь пакетный градиентный спуск действительно сойдется при условии достаточно долгого времени обучения. Как упоминалось, если постепенно не снижать скорость обучения, то стохастический градиентный спуск и мини-пакетный градиентный спуск будут прыгать вокруг оптимума.

8. Если ошибка проверки намного превышает ошибку обучения, то причина, скорее всего, в том, что ваша модель переобучается обучающим набором. Одна попытка устранить проблему заключается в снижении полиномиальной степени: модель с меньшим числом степеней свободы менее склонна к переобучению. Другая попытка предусматривает регуляризацию модели — скажем, путем добавления к функции издержек штрафа ℓ_2 (гребневая регрессия) или штрафа ℓ_1 (лассо-регрессия). Это также сократит количество степеней свободы модели. Наконец, можно попробовать увеличить размер обучающего набора.
9. Если ошибка обучения и ошибка проверки почти одинаковы и довольно высоки, то модель, вероятно, недообучается на обучающем наборе, что означает наличие у нее высокого смещения. Вы должны попробовать снизить гиперпараметр регуляризации α .
10. Давайте посмотрим.
- Модель с определенной регуляризацией обычно работает лучше, чем модель без регуляризации, поэтому в общем случае вы должны отдавать предпочтение гребневой регрессии перед обыкновенной линейной регрессией².
 - Лассо-регрессия применяет штраф ℓ_1 , который стремится довести веса до полных нулей. Итогом будут разреженные модели, где все веса за исключением наиболее важных являются нулевыми. Это способ автоматического выполнения выбора признаков, который хорош, если вы полагаете, что в действительности только немногие признаки существенны. Когда такой уверенности нет, вы должны использовать гребневую регрессию.
 - Эластичной сети обычно отдают предпочтение перед лассо-регрессией, т.к. в некоторых случаях лассо-регрессия может работать с перебоями (когда несколько признаков сильно связаны или признаков больше, чем обучающих образцов). Однако эластичная сеть добавляет дополнительный гиперпараметр, подлежащий подстройке. Если вы просто хотите получить лассо-регрессию, работающую без перебоев, тогда можете применять эластичную сеть со значением `l1_ratio`, близким к 1.

² Кроме того, нормальное уравнение требует получения обратной матрицы, но его матрица не всегда обратима. Напротив, матрица для гребневой регрессии обратима всегда.

11. Если вы хотите классифицировать фотографии как сделанные снаружи/внутри и днем/ночью, то поскольку такие классы не являются взаимоисключающими (т.е. возможны все четыре комбинации), потребуется обучить два классификатора, основанные на логистической регрессии.
12. См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml>.

Глава 5. Методы опорных векторов

1. Фундаментальная идея, лежащая в основе методов опорных векторов, состоит в том, чтобы обеспечить самую широкую, какую только возможно, полосу между классами. Другими словами, целью является наличие как можно более широкого зазора между границей решений, которая разделяет два класса и обучающие образцы. При выполнении классификации с мягким зазором классификатора SVM ищет компромисс между идеальным разделением двух классов и самой широкой из возможных полосой (т.е. несколько образцов могут оказаться на самой полосе). Еще одна ключевая идея в том, чтобы использовать ядра при обучении на нелинейных наборах данных.
2. После обучения классификатора SVM опорный вектор — это любой образец, расположенный на полосе (см. предыдущий ответ), включая границу. Граница решений полностью определяется опорными векторами. Образцы, которые не являются опорными векторами (т.е. находятся вне полосы), не оказывают никакого влияния; вы могли бы удалить такие образцы, добавить дополнительные образцы или переместить их, и до тех пор, пока образцы остаются вне полосы, они не будут влиять на границу решений. При вычислении прогнозов задействуются только опорные векторы, а не полный обучающий набор.
3. Методы SVM пытаются обеспечить самую широкую, какую только возможно, полосу между классами (см. первый ответ), так что если обучающий набор не масштабирован, то методы SVM будут иметь тенденцию игнорировать небольшие признаки (см. рис. 5.2).
4. Классификатор SVM способен выдавать расстояние между испытательным образцом и границей решений, которое вы можете применять в качестве меры доверия. Тем не менее, эту меру невозможно

прямо преобразовать в оценку вероятности класса. Если вы установите `probability=True` при создании классификатора SVM в Scikit-Learn, тогда после обучения он будет калибровать вероятности с использованием логистической регрессии по мерам SVM (обученных посредством дополнительной перекрестной проверки с контролем по пяти блокам на обучающих данных). Это добавит к SVM методы `predict_proba()` и `predict_log_proba()`.

5. Данный вопрос применим только к линейным SVM, поскольку ядерные SVM могут применять только двойственную форму. Вычислительная сложность прямой формы задачи SVM пропорциональна количеству обучающих образцов m , в то время как вычислительная сложность двойственной формы пропорциональна числу между m^2 и m^3 . Таким образом, при наличии миллионов образцов вы определенно должны использовать прямую форму, потому что двойственная форма будет гораздо более медленной.
6. Если классификатор SVM с ядром RBF недообучается на обучающем наборе, то возможно регуляризации слишком много. Чтобы уменьшить ее, вам понадобится увеличить гиперпараметр `gamma` либо `C` (или оба).
7. Давайте назовем параметры QP для задачи классификации с жестким зазором \mathbf{H}' , \mathbf{f}' , \mathbf{A}' и \mathbf{b}' (см. раздел “Квадратичное программирование” в главе 5). Параметры QP для задачи классификации с мягким зазором имеют m дополнительных параметров ($n_p = n + 1 + m$) и m дополнительных ограничений ($n_c = 2m$). Они могут быть определены следующим образом.
 - \mathbf{H} равно \mathbf{H}' плюс m столбцов нулей справа и m строк нулей внизу:

$$\mathbf{H} = \begin{pmatrix} \mathbf{H}' & 0 & \dots \\ 0 & 0 & \\ \vdots & & \ddots \end{pmatrix}.$$

- \mathbf{f} равно \mathbf{f}' с m дополнительными элементами, которые все равны значению гиперпараметра C .
- \mathbf{b} равно \mathbf{b}' с m дополнительными элементами, которые все равны нулю.

- A равно A' с добавочной единичной матрицей I_m размера $m \times m$, добавленной справа, $-I_m$ чуть ниже ее и остатком, заполненным нулями:

$$A = \begin{pmatrix} A' & I_m \\ 0 & -I_m \end{pmatrix}.$$

Решения упражнений 8, 9 и 10 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 6. Деревья принятия решений

1. Глубина хорошо сбалансированного двоичного дерева, содержащего m листьев, равна $\log_2(m)^3$ с округлением в большую сторону. Двоичное дерево принятия решений (дерево, которое принимает только двоичные решения, как в случае всех деревьев внутри библиотеки Scikit-Learn) к концу обучения будет более или менее сбалансированным, имея по одному листу на обучающий образец, если оно обучалось без ограничений. Таким образом, если обучающий набор содержал один миллион образцов, то дерево принятия решений будет иметь глубину $\log_2(10^6) \approx 20$ (фактически чуть больше, т.к. дерево обычно не является идеально сбалансированным).
2. Загрязненность Джини узла обычно ниже, чем у его родителя. Причиной является функция издержек алгоритма обучения CART, которая расщепляет каждый узел способом, сводящим к минимуму взвешенную сумму загрязненностей Джини его дочерних узлов. Однако узел может иметь более высокую загрязненность Джини, чем у его родителя, пока такое увеличение с лихвой компенсируется уменьшением загрязненности другого дочернего узла. Например, возьмем узел, содержащий четыре образца класса A и один образец класса B. Вот его загрязненность Джини: $1 - \frac{1^2}{5} - \frac{4^2}{5} = 0.32$. Теперь предположим, что набор данных является одномерным, а образцы выстроены в следующем порядке: A, B, A, A, A. Вы можете проверить, будет ли алгоритм расщеплять данный узел после второго образца, производя один дочерний узел с образцами A, B и еще один дочерний узел с образцами A, A, A. Загрязненность Джини первого дочернего узла выглядит как

³ \log_2 — это двоичный логарифм, $\log_2(m) = \log(m) / \log(2)$.

$1 - \frac{1^2}{2} - \frac{1^2}{2} = 0.5$, что выше, чем у его родителя. Это уравнивается тем фактом, что другой узел является чистым, а потому общая взвешенная загрязненность Джини составляет $0.5 + \frac{3}{5} \times 0 = 0.2$, что ниже, чем загрязненность Джини родителя.

3. Если дерево принятия решений переобучается обучающим набором, то уменьшение `max_depth` может быть хорошей идеей, т.к. это ограничит модель, регулируя ее.
4. Деревья принятия решений не заботятся о том, масштабированы или центрированы обучающие данные; это один из связанных с ними приятных моментов. Следовательно, если дерево принятия решений недообучается на обучающем наборе, тогда масштабирование входных признаков будет просто пустой тратой времени.
5. Вычислительная сложность обучения дерева принятия решений составляет $O(n \times m \log(m))$. Таким образом, если вы умножите размер обучающего набора на 10, то время обучения будет умножено на $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. Если $m = 10^6$, тогда $K \approx 11.7$, поэтому вы можете ожидать, что обучение займет примерно 11.7 часов.
6. Предварительная сортировка обучающего набора ускоряет обучение, только если набор данных содержит менее нескольких тысяч образцов. В случае 100 000 образцов установка `presort=True` значительно замедлит обучение.

Решения упражнений 7 и 8 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 7. Ансамблевое обучение и случайные леса

1. Если вы обучили пять разных моделей, и все они достигают точности 95%, то можете попытаться скомбинировать их в ансамбль с голосованием, который часто будет давать даже лучшие результаты. Он лучше работает, если модели сильно отличаются (например, классификатор SVM, классификатор на базе дерева принятия решений, классификатор на основе логистической регрессии и т.д.). Еще лучше, когда модели обучаются на разных обучающих образцах (в этом заключается весь смысл

ансамблей с бэггингом и вставкой), но если это не так, то ансамбль по-прежнему будет работать при условии, что модели сильно отличаются.

2. Классификатор с жестким голосованием просто подсчитывает голоса каждого классификатора в ансамбле и выбирает класс, получивший большинство голосов. Классификатор с мягким голосованием вычисляет среднюю оценочную вероятность для каждого класса и выбирает класс с наивысшей вероятностью. Это придает голосам с высоким доверием больший вес и часто работает лучше, но только в случае, если каждый классификатор способен оценивать вероятности классов (например, для классификаторов SVM в Scikit-Learn потребуется установить `probability = True`).
3. Обучение ансамбля с бэггингом вполне можно ускорить, распределив его между множеством серверов, т.к. каждый прогнозатор в ансамбле не зависит от остальных. То же самое касается ансамблей с вставкой и случайных лесов, по той же причине. Тем не менее, каждый прогнозатор в ансамбле с бустингом построен на основе предыдущего прогнозатора, поэтому обучение обязано быть последовательным, и вы не получите какой-либо выгоды, распределив обучение между несколькими серверами. Что касается ансамблей со стекингом, то все прогнозаторы в заданном слое не зависят друг от друга, а потому их можно обучать параллельно на множестве серверов. Однако прогнозаторы в одном слое могут обучаться только после того, как обучены все прогнозаторы из предыдущего слоя.
4. При оценке с помощью неиспользуемых образцов каждый прогнозатор в ансамбле с бэггингом оценивается с применением образцов, на которых он не обучался (они удерживались в стороне). Это позволяет получить достаточно объективную оценку ансамбля без необходимости в наличии дополнительного проверочного набора. Таким образом, для обучения доступно больше образцов и ансамбль может работать чуть лучше.
5. При выращивании дерева в случайном лесе для каждого узла, подлежащего расщеплению, рассматривается только случайный поднабор признаков. Это справедливо также для особо случайных деревьев, но они продвигаются на шаг дальше: вместо поиска наилучшего возможного порога, как поступает обыкновенное дерево принятия решений, они используют для каждого признака случайные пороги. Такая дополни-

тельная случайность действует подобно форме регуляризации: если случайный лес переобучается обучающими данными, то особо случайные деревья могут работать лучше. Кроме того, поскольку особо случайные деревья не ищут наилучшие возможные пороги, в обучении они гораздо быстрее, чем случайные леса. Тем не менее, при выработке прогнозов особо случайные деревья ни быстрее, ни медленнее случайных лесов.

6. Если ваш ансамбль AdaBoost недообучается на обучающих данных, тогда можете попробовать увеличить количество оценщиков или уменьшить гиперпараметры регуляризации базового оценщика. Можете также попробовать слегка повысить скорость обучения.
7. Если ваш ансамбль с градиентным бустингом переобучается обучающим набором, то вы должны попробовать уменьшить скорость обучения. Вы также можете воспользоваться ранним прекращением, чтобы найти правильное количество прогнозаторов (по всей видимости, их слишком много).

Решения упражнений 8 и 9 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 8. Понижение размерности

1. Мотивы и недостатки.

- Ниже перечислены главные мотивы для понижения размерности.
 - Чтобы ускорить последующий алгоритм обучения (в некоторых случаях понижение размерности может даже устранить шум и избыточные признаки, улучшая работу алгоритма обучения).
 - Чтобы визуализировать данные и получить представление о самых важных признаках.
 - Просто чтобы сберечь пространство (сжатие).
- Далее указаны основные недостатки понижения размерности.
 - Некоторая информация утрачивается, из-за чего может ухудшиться производительность последующих алгоритмов обучения.
 - Понижение размерности может быть сопряжено с большим объемом вычислений.

- Оно привносит некоторую сложность в конвейеры машинного обучения.
 - Трансформированные признаки зачастую трудно интерпретировать.
2. Под “проклятием размерности” понимается тот факт, что многие проблемы, которых нет в пространстве с низким числом измерений, появляются в пространстве с высоким числом измерений. В машинном обучении его распространенным олицетворением является то, что случайно выбранные векторы с высоким числом измерений обычно оказываются сильно разреженными, а это увеличивает риск переобучения и весьма затрудняет идентификацию шаблонов в данных без наличия большого объема обучающих данных.
 3. После того как размерность набора данных была понижена с применением одного из алгоритмов, рассмотренных в главе, полностью обратить операцию почти всегда невозможно, потому что во время понижения размерности некоторая информация была утрачена. Кроме того, в то время как одни алгоритмы (такие как PCA) имеют простую процедуру обратной трансформации, которая может восстановить набор данных в виде, относительно похожем на первоначальный набор, другие алгоритмы (вроде T-SNE) такой процедурой не располагают.
 4. Алгоритм PCA можно использовать для значительного понижения размерности большинства наборов данных, даже если они крайне нелинейные, потому что алгоритм, по меньшей мере, способен избавиться от бесполезных измерений. Однако если бесполезных измерений нет (скажем, как в наборе данных Swiss roll), тогда понижение размерности посредством PCA приведет к утрате слишком большого объема информации. Ведь вы хотите развернуть швейцарский рулет, а не сплющить его.
 5. Сложный вопрос: все зависит от набора данных. Давайте рассмотрим два предельных примера. Предположим, что набор данных состоит из точек, которые почти полностью выровнены. В таком случае алгоритм PCA может сократить набор данных до только одного измерения, одновременно предохраняя 95% дисперсии. Теперь представим, что набор данных состоит из совершенно случайных точек, разбросанных по 1 000 измерений. В этом случае для предохранения 95% дисперсии требуется приблизительно 950 измерений. Следовательно, ответом будет — в зависимости от набора данных, и количеством измерений

может быть любое число между 1 и 950. Вычерчивание объясненной дисперсии как функции количества измерений позволяет получить примерное представление о присущей набору данных размерности.

6. Простой алгоритм PCA принимается по умолчанию, но он работает, только когда набор данных умещается в памяти. Инкрементный алгоритм PCA удобен для крупных наборов данных, которые в память не умещаются, но он медленнее простого PCA, поэтому если набор данных умещается в памяти, тогда вы должны отдавать предпочтение простому алгоритму PCA. Инкрементный алгоритм PCA также удобен для задач динамического обучения, в которых необходимо применять PCA на лету при каждом поступлении нового образца. Рандомизированный алгоритм PCA удобен, когда нужно значительно понизить размерность и набор данных умещается в памяти; в таком случае он намного быстрее простого PCA. Наконец, ядерный алгоритм PCA удобен для нелинейных наборов данных.
7. По идее алгоритм понижения размерности выполняется хорошо, если он устраняет из набора данных большое количество измерений, не утрачивая слишком много информации. Измерить это можно за счет применения обратной трансформации и подсчета ошибки восстановления. Тем не менее, не все алгоритмы понижения размерности предоставляют обратную трансформацию. В качестве альтернативы, когда понижение размерности используется как шаг предварительной обработки перед выполнением другого алгоритма машинного обучения (скажем, классификатора на основе случайного леса), вы можете просто измерить производительность второго алгоритма. Если понижение размерности не вызывает утрату слишком большого объема информации, то алгоритм должен выполняться в той же степени хорошо, как и на первоначальном наборе данных.
8. Соединение в цепочку двух разных алгоритмов понижения размерности, безусловно, может иметь смысл. Распространенным примером является использование алгоритма PCA для быстрого избавления от большого количества бесполезных измерений, после чего применение еще одного более медленного алгоритма понижения размерности, такого как LLE. Такой двухэтапный подход обеспечит ту же самую производительность модели, как и в случае использования только LLE, но за меньшее время.

Глава 9. Подготовка к работе с TensorFlow

1. Ниже перечислены основные преимущества и недостатки создания вычислительного графа вместо выполнения вычислений напрямую.
 - Основные преимущества.
 - Библиотека TensorFlow может автоматически вычислять градиенты (с применением дифференцирования в обратном режиме).
 - Она может позаботиться о параллельном выполнении операций в разных потоках.
 - Она облегчает прогон той же самой модели на разных устройствах.
 - Она упрощает самоанализ — например, для просмотра модели в TensorBoard.
 - Основные недостатки.
 - Библиотека TensorFlow делает кривую обучения более крутой.
 - Она затрудняет пошаговую отладку.
2. Да, оператор `a_val = a.eval(session=sess)` действительно эквивалентен оператору `a_val = sess.run(a)`.
3. Нет, оператор `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` не эквивалентен оператору `a_val, b_val = sess.run([a, b])`. На самом деле первый оператор запускает граф дважды (раз для вычисления `a` и еще раз для вычисления `b`), в то время как второй оператор запускает граф только один раз. Если любая из этих операций (или операций, от которых они зависят) имеет побочные эффекты (например, модифицируется переменная, в очередь помещается элемент, объект чтения читает файл), тогда результаты окажутся разными. Если побочные эффекты отсутствуют, то оба оператора возвратят тот же самый результат, но второй оператор будет быстрее первого.
4. Нет, запускать два вычислительных графа в одном сеансе нельзя. Графы сначала понадобится объединить в единственный граф.

5. В локальной версии TensorFlow значениями переменных управляют сеансы, поэтому если вы создадите граф `g`, содержащий переменную `w`, затем запустите два потока и откроете в каждом потоке сеанс с использованием того же самого графа `g`, то каждый сеанс будет иметь собственную копию переменной `w`. Однако в распределенной версии TensorFlow значения переменных хранятся в контейнерах, управляемых кластером, так что если оба сеанса подключатся к тому же самому кластеру и задействуют тот же самый контейнер, то они будут разделять то же самое значение переменной `w`.
6. Переменная инициализируется, когда вызывается ее инициализатор, и уничтожается при окончании сеанса. В распределенной версии TensorFlow переменные находятся внутри контейнеров в кластере, поэтому закрытие сеанса не приводит к уничтожению переменных. Чтобы уничтожить переменную, необходимо очистить ее контейнер.
7. Переменные и заполнители совершенно отличаются друг от друга, но начинающие часто путают их.
 - Переменная представляет собой операцию, которая содержит в себе значение. Если вы запускаете переменную, тогда она возвращает это значение. Прежде чем переменную можно будет запустить, ее потребуется инициализировать. Значение переменной можно изменять (например, с применением операции присваивания). Переменная поддерживает состояние: она сохраняет значение между последовательными проходами графа. Переменная обычно используется для хранения параметров модели, но может применяться и для других целей (скажем, для подсчета глобальных шагов обучения).
 - Формально заполнители делают немного: они всего лишь содержат в себе информацию о типе и форме тензоров, которые представляют, но не имеют значения. В действительности, чтобы вычислить операцию, которая зависит от заполнителя, потребуется предоставить TensorFlow значение этого заполнителя (используя аргумент `feed_dict`), иначе возникнет исключение. Заполнители обычно применяются для снабжения TensorFlow обучающими либо испытательными данными во время стадии выполнения. Они также удобны при передаче значения узлу присваивания, чтобы изменить значение переменной (например, веса модели).

8. Если вы запустите граф для оценки операции, которая зависит от заполнителя, но не передадите его значение, тогда получите исключение. Если операция не зависит от заполнителя, то исключение не генерируется.
9. При запуске графа вы можете передавать выходное значение любой операции, а не только значения заполнителей. Тем не менее, на практике подобное случается довольно редко (оно может быть полезно, скажем, когда вы кешируете выход замороженных слоев; см. главу 11).
10. Вы можете указать начальное значение переменной при построении графа, и она будет инициализироваться позже, когда во время стадии выполнения запускается ее инициализатор. Если вы хотите изменить значение переменной на стадии выполнения, тогда проще всего для этого создать узел присваивания (на стадии построения графа), используя функцию `tf.assign()` с передачей ей переменной и заполнителя в качестве параметров. Во время стадии выполнения вы можете запустить операцию присваивания и предоставить ей новое значение переменной с применением заполнителя.

```
import tensorflow as tf

x=tf.Variable(tf.random_uniform(shape=(),minval=0.0,maxval=1.0))
x_new_val = tf.placeholder(shape=(), dtype=tf.float32)
x_assign = tf.assign(x, x_new_val)

with tf.Session():
    x.initializer.run() #случайное число,выбранное *прямо сейчас*
    print(x.eval()) # 0.646157 (какое-то случайное число)
    x_assign.eval(feed_dict={x_new_val: 5.0})
    print(x.eval()) # 5.0
```

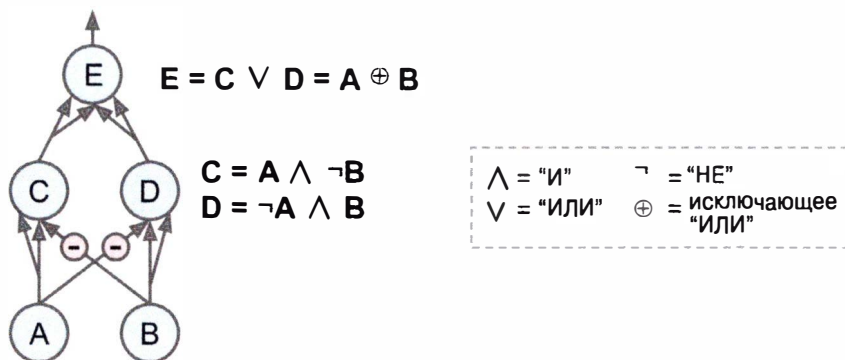
11. Автоматическому дифференцированию в обратном режиме (реализованному TensorFlow) понадобится обойти граф только дважды, чтобы вычислить градиенты функции издержек относительно любого количества переменных. С другой стороны, автоматическому дифференцированию в прямом режиме необходим один обход для каждой переменной (поэтому если нужны градиенты относительно 10 разных переменных, тогда потребуется обойти граф 10 раз). Что касается символического дифференцирования, то для вычисления градиентов оно построит другой граф, а потому вообще не будет обходить исходный граф (кроме случая, когда строится новый граф градиентов). Высоко

оптимизированная система символического дифференцирования потенциально могла бы запускать новый граф градиентов только раз для вычисления градиентов относительно всех переменных, но такой новый граф может быть чрезвычайно сложным и неэффективным в сравнении с исходным графом.

12. См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml>.

Глава 10. Введение в искусственные нейронные сети

1. Ниже показана нейронная сеть на основе исходных искусственных нейронов, которая вычисляет $A \oplus B$ (где \oplus представляет исключающее “ИЛИ”), пользуясь тем фактом, что $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. Существуют и другие решения, учитывающие тот факт, что $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$, или то, что $A \oplus B = (A \vee B) \wedge (\neg A \vee \neg B)$, и т.д.



2. Классический персептрон будет сходиться, только если набор данных является линейно сепарабельным, и он не в состоянии оценивать вероятности классов. В противоположность этому классификатор на основе логистической регрессии будет сходиться в хорошее решение, даже когда набор данных не линейно сепарабельный, и выдавать вероятности классов. Если вы измените функцию активации персептрона на логистическую функцию активации (или многопеременную функцию активации при наличии множества нейронов) и обучите его с применением градиентного спуска (либо какого-то другого алгоритма оптимизации, сводящего к минимуму функцию издержек, обычно перекрестную энтропию), тогда персептрон станет эквивалентным классификатору на основе логистической регрессии.

3. Логистическая функция активации была ключевым ингредиентом при обучении первых многослойных персептронов потому, что ее производная всегда ненулевая, поэтому градиентный спуск мог неизменно скатываться по уклону. Когда функция активации является ступенчатой, градиентный спуск не в состоянии перемещаться, т.к. уклон вообще отсутствует.
4. Ступенчатая функция, логистическая функция, функция гиперболического тангенса, выпрямленный линейный элемент (см. рис. 10.8). В главе 11 приводятся другие примеры, такие как ELU и варианты ReLU.
5. Обсудим многослойный персептрон, описанный в вопросе: он состоит из одного входного слоя с 10 сквозными нейронами, за которым следует один скрытый слой с 50 искусственными нейронами и один выходной слой с 3 искусственными нейронами. Все искусственные нейроны используют функцию активации ReLU.
 - Формой входной матрицы \mathbf{X} является $m \times 10$, где m представляет размер обучающего пакета.
 - Формой вектора весов скрытого слоя \mathbf{W}_h является 10×50 , а длина его вектора смещений \mathbf{b}_h составляет 50.
 - Формой вектора весов выходного слоя \mathbf{W}_o является 50×3 , а длина его вектора смещений \mathbf{b}_o составляет 3.
 - Формой входной матрицы \mathbf{Y} сети является $m \times 3$.
 - $\mathbf{Y} = \text{ReLU}(\text{ReLU}(\mathbf{X} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_o + \mathbf{b}_o)$. Вспомните, что функция ReLU просто устанавливает в ноль каждое отрицательное число внутри матрицы. Также обратите внимание, что при добавлении вектора смещений к матрице он добавляется к каждой строке в матрице, что называется ретрансляцией.
6. Чтобы классифицировать почтовые сообщения на спам и не спам, в выходном слое нейронной сети нужен только один нейрон, к примеру, указывающий вероятность того, что почтовое сообщение представляет собой спам. Обычно при оценке вероятности вы будете применять в выходном слое логистическую функцию активации. Если взамен вы хотите заняться набором данных MNIST, тогда понадобится иметь в выходном слое 10 нейронов и заменить логистическую функцию мно-

гопеременной функцией активации, которая способна обрабатывать множество классов, выдавая по одной вероятности на класс. Наконец, при желании, чтобы нейронная сеть прогнозировала цены на дома (см. главу 2), то необходим один выходной нейрон, при этом в выходном слое функция активации вообще не используется⁴.

7. Обратное распространение — это прием, применяемый для обучения искусственных нейронных сетей. Сначала вычисляются градиенты функции издержек относительно каждого параметра модели (все веса и смещения), после чего с использованием этих градиентов выполняется шаг градиентного спуска. Такой шаг обратного распространения обычно делается тысячи или миллионы раз с применением многих обучающих пакетов, пока параметры модели не сойдутся в значения, которые (надо надеяться) минимизируют функцию издержек. Для вычисления градиентов обратное распространение использует автоматическое дифференцирование в обратном режиме (хотя на момент создания обратного распространения оно так не называлось и вдобавок изобреталось заново несколько раз). Автоматическое дифференцирование в обратном режиме выполняет прямой проход через вычислительный граф, вычисляя значение каждого узла для текущего обучающего пакета, и затем осуществляет обратный проход, вычисляя все градиенты за один раз (дополнительные сведения ищите в приложении Г). Итак, в чем разница? Дело в том, что обратное распространение относится к полному процессу обучения искусственной нейронной сети с применением множества шагов обратного распространения, каждый из которых вычисляет градиенты и использует их для выполнения шага градиентного спуска. Напротив, автоматическое дифференцирование в обратном режиме представляет собой просто прием эффективного вычисления градиентов, который по стечению обстоятельств применяется обратным распространением.
8. Вот список гиперпараметров, которые можно подстраивать в базовом многослойном персептроне: количество скрытых слоев, число нейронов в каждом скрытом слое, а также функция активации, используемая

⁴ Когда прогнозируемые значения по своей величине могут варьироваться в пределах многих порядков, тогда вы можете предпочесть прогнозирование логарифма целевого значения, а не самого целевого значения. Простое вычисление экспоненты выхода нейронной сети даст оценочное значение (т.к. $\exp(\log v) = v$).

в каждом скрытом слое и в выходном слое⁵. Обычно хорошим стандартным выбором для скрытых слоев является функция активации ReLU (или один из ее вариантов; см. главу 11). В выходном слое, как правило, будет применяться логистическая функция активации для двоичной классификации, многопеременная функция активации для многоклассовой классификации или вообще никакой функции активации для регрессии.

9. Если многослойный персептрон переобучается обучающими данными, тогда можно попробовать уменьшить количество скрытых слоев и число нейронов на скрытый слой.
10. См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml>.

Глава 11. Обучение глубоких нейронных сетей

1. Нет, все веса обязаны выбираться независимо; они не должны все иметь одно и то же начальное значение. Важной целью случайного выбора весов является нарушение симметрии: если все веса имеют то же самое начальное значение, даже отличное от нуля, тогда симметрия не нарушается (т.е. все нейроны в заданном слое эквивалентны) и обратное распространение неспособно ее нарушить. Более конкретно, все нейроны в любом заданном слое всегда будут иметь одинаковые веса. Это похоже на то, как если бы существовал только один нейрон на слой, но гораздо медленнее. Такой конфигурации практически нереально сойтись в хорошее решение.
2. Инициализировать члены смещения нулями совершенно допустимо. Некоторым людям нравится инициализировать их, как и веса, что также нормально; большой разницы в подходах нет.
3. Ниже указано несколько преимуществ функции активации ELU по сравнению с ReLU.

⁵ В главе 11 мы обсуждали много приемов, которые вводят дополнительные гиперпараметры: тип инициализации весов, гиперпараметры функции активации (скажем, величина утечки в ReLU с утечкой), порог отсечения градиентов, тип оптимизатора и его гиперпараметры (например, гиперпараметр момента в случае использования `MomentumOptimizer`), тип регуляризации для каждого слоя и гиперпараметры регуляризации (к примеру, доля отключения, когда применяется отключение) и т.д.

- Функция активации ELU может принимать отрицательные значения, а потому усредненный выход нейронов в любом заданном слое обычно ближе к 0, чем в случае использования функции активации ReLU (которая никогда не выдает отрицательные значения). Это помогает смягчить проблему исчезновения градиентов.
 - Она всегда имеет ненулевую производную, избегая проблемы угасающих элементов, которой могут подвергаться элементы ReLU.
 - Функция активации ELU повсюду гладкая, тогда как наклон функции активации ReLU в точке $z = 0$ скачкообразно изменяется с 0 на 1. Такое резкое изменение может замедлить градиентный спуск, поскольку он будет прыгать возле $z = 0$.
4. Функция активации ELU является хорошим выбором по умолчанию. Если необходимо, чтобы нейронная сеть была как можно более быстрой, тогда применяйте взамен варианты ReLU с утечкой (скажем, простой элемент ReLU с утечкой, использующий стандартное значение гиперпараметра). Простота функции активации ReLU делает ее предпочтительным выбором для многих людей, несмотря на тот факт, что в целом ELU и ReLU с утечкой ее превосходят. Однако способность функции активации ReLU выдавать в точности ноль в некоторых случаях может быть полезной (примеры ищите в главе 15). Функция гиперболического тангенса (\tanh) может быть удобной в выходном слое, если нужно выдавать число между -1 и 1 , но в настоящее время в скрытых слоях она применяется нечасто. Логистическая функция активации также полезна в выходном слое, когда необходимо оценивать вероятность (скажем, для двоичной классификации), но она редко используется в скрытых слоях (есть исключения — например, для кодирующего слоя вариационного автокодировщика; см. главу 15). Наконец, многопеременная функция активации удобна в выходном слое при выдаче вероятностей для взаимоисключающих классов, но помимо этого она редко (если вообще когда-либо) применяется в скрытых слоях.
5. Если при использовании `MomentumOptimizer` вы установите значение гиперпараметра `momentum` слишком близким к 1 (скажем, 0.99999), тогда алгоритм, вероятно, наберет высокую скорость, с надеждой добраться примерно до глобального минимума, но затем из-за своего момента он промахнется, попав прямо за минимум. Далее он замедлится

и возвратится, снова ускорится, опять промахнется и т.д. Алгоритм может раскачиваться подобным образом много раз до того, как сойтись, поэтому в итоге его сходжение потребует гораздо большего времени, чем при меньшем значении `momentum`.

6. Первый способ получения разреженной модели (т.е. модели с большинством нулевых весов) предусматривает обучение модели обычным образом с последующим обнулением очень маленьких весов. Второй способ, обеспечивающий более высокую разреженность, предполагает применение регуляризации ℓ_1 во время обучения, что подталкивает оптимизатор к разреженности. Третий способ заключается в том, чтобы скомбинировать регуляризацию ℓ_1 с двойным усреднением, используя класс `FTRLOptimizer` из TensorFlow.
7. Да, отключение замедляет процесс обучения, в целом приблизительно в два раза. Тем не менее, оно не влияет на выводение, поскольку включается лишь во время обучения.

Решения упражнений 8, 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 12. Использование TensorFlow для распределения вычислений между устройствами и серверами

1. Процесс TensorFlow в самом начале захватывает всю доступную память во всех графических процессорах, которые ему видны. Таким образом, если при запуске программы TensorFlow вы получили ошибку `CUDA_ERROR_OUT_OF_MEMORY`, то вероятная причина связана с тем, что другие выполняющиеся процессы уже заняли всю память, по крайней мере, на одном видимом ГП (скорее всего, это еще один процесс TensorFlow). Тривиальное решение проблемы предусматривает останов других процессов и повторный запуск программы TensorFlow. Однако если все процессы должны выполняться одновременно, тогда простым вариантом будет выделение каждому процессу своего устройства путем соответствующей установки переменной среды `CUDA_VISIBLE_DEVICES` для каждого устройства. Другой вариант — сконфигурировать TensorFlow на захватывание только части памяти ГП, для чего создать объект `ConfigProto`, указать в его

параметре `gpu_options.per_process_gpu_memory_fraction` долю всей памяти, которую он должен захватывать (например, 0.4), и применять этот объект `ConfigProto` при открытии сеанса. Последний вариант заключается в том, чтобы сообщить TensorFlow о необходимости захвата памяти только по мере надобности, установив `gpu_options.allow_growth` в `True`. Тем не менее, использовать такой подход обычно не рекомендуется, поскольку любая память, которую TensorFlow захватывает, никогда не освобождается, и повторяемое поведение гарантировать трудно (могут возникать состязания в зависимости от того, какой процесс начинается первым, сколько памяти нужно процессам во время обучения и т.д.).

2. Прикрепляя операцию к устройству, вы сообщаете TensorFlow о том, где желательно разместить эту операцию. Однако ряд ограничений могут помешать TensorFlow удовлетворить ваш запрос. Например, у операции может отсутствовать реализация (называемая ядром) для конкретного типа устройства. В таком случае по умолчанию библиотека TensorFlow будет генерировать исключение, но вы можете сконфигурировать ее так, чтобы она взамен помещала операцию в центральный процессор (это называется мягким размещением). Другим примером является операция, которая может модифицировать переменную; такая операция и переменная должны располагаться рядом. Таким образом, разница между прикреплением и размещением операции состоит в том, что прикрепление является просьбой к TensorFlow (“Пожалуйста, разместите данную операцию на ГП #1”), а размещение представляет собой действие, которое TensorFlow в итоге выполняет (“Жаль, но операция помещена в ЦП”).
3. Если вы работаете с установленной копией TensorFlow, поддерживающей графические процессоры, и применяете стандартное размещение, тогда если все операции имеют ядро ГП (т.е. реализацию для ГП), то все они действительно будут размещены на первом устройстве ГП. Тем не менее, если одна или большее число операций не имеют ядра ГП, тогда по умолчанию TensorFlow будет генерировать исключение. Если вы сконфигурируете библиотеку TensorFlow так, чтобы она взамен помещала операции в ЦП (мягкое размещение), то на первом устройстве ГП будут размещены все операции кроме тех, у которых нет ядра ГП, и операции, которые должны размещаться рядом с ними (см. предыдущий ответ).

4. Да, если вы прикрепляете переменную к `/gpu:0`, то она может использовать операциями, размещенными на `/gpu:1`. Библиотека TensorFlow позаботится о добавлении подходящих операций для передачи значения переменной между устройствами. То же самое касается устройств, находящихся на разных серверах (при условии, что они являются частью одного кластера).
5. Да, две операции, размещенные на одном устройстве, могут выполняться параллельно: TensorFlow позаботится о запуске операций в параллельном режиме (на разных ядрах ЦП или в разных потоках ГП) при условии, что никакие операции не зависят от результатов других операций. Кроме того, вы можете запускать множество сеансов в параллельных потоках (или процессах) и в каждом потоке выполнять операции. Поскольку сеансы независимы, у TensorFlow будет возможность выполнять операцию из одного сеанса параллельно с операцией из другого сеанса.
6. Зависимости управления применяются, когда оценку операции X желательно отложить до тех пор, пока не будут выполнены какие-то другие операции, даже если они не обязательны для вычисления X . Это особенно полезно, когда операция X занимает большой объем памяти и нужна только позже в вычислительном графе, либо если X использует много операций ввода-вывода (скажем, требует значения крупной переменной, расположенной на другом устройстве или сервере), и вы не хотите выполнять ее одновременно с остальными операциями с интенсивным вводом-выводом, чтобы избежать насыщения полосы пропускания.
7. Вам повезло! В распределенной версии TensorFlow значения переменных находятся в контейнерах, управляемых кластером, так что даже если вы закроете сеанс и завершите работу клиентской программы, параметры модели по-прежнему будут благополучно существовать в кластере. Вам просто необходимо открыть новый сеанс в кластере и сохранить модель (удостоверьтесь, что не вызываете инициализаторы переменных и не восстанавливаете предыдущую модель, т.к. это уничтожит новую более точную модель!).

Решения упражнений 8, 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 13. Сверточные нейронные сети

1. Ниже перечислены главные преимущества сети CNN в сравнении с полносвязной сетью DNN для классификации изображений.
 - Поскольку следующие друг за другом слои связаны только частично и интенсивно повторно используют свои веса, сеть CNN имеет намного меньше параметров, чем полносвязная сеть DNN, что делает ее гораздо более быстрой в обучении, снижает риск переобучения и требует значительно меньшего объема обучающих данных.
 - Когда сеть CNN узнала ядро, которое может обнаруживать определенный признак, она способна выявить этот признак где угодно в изображении. Напротив, когда сеть DNN узнала признак в одном месте, она может обнаруживать его только в этом конкретном месте. Так как изображения обычно имеют очень повторяющиеся признаки, сети CNN способны обобщаться гораздо лучше сетей DNN для задач обработки изображений, подобных классификации, с применением меньшего количества обучающих образцов.
 - Наконец, сеть DNN не обладает априорными знаниями о том, как организованы пиксели; она не знает, что соседние пиксели близки. В архитектуре сетей CNN такие априорные знания являются встроенными. Более низкие слои обычно идентифицируют признаки в небольших областях изображений, а более высокие слои объединяют низкоуровневые признаки в признаки покрупнее. Такой подход хорошо работает с большинством естественных изображений, с самого начала давая сетям CNN решающее преимущество в сравнении с сетями DNN.
2. Давайте подсчитаем, сколько параметров имеет такая сеть CNN. Поскольку ее первый сверточный слой содержит 3×3 ядер, а вход имеет три канала (красный, зеленый и синий), то каждая карта признаков обладает $3 \times 3 \times 3$ весами плюс член смещения. Получается 28 параметров на карту признаков. Так как в первом сверточном слое есть 100 карт признаков, суммарно он имеет 2 800 параметров. Второй сверточный слой содержит 3×3 ядер, а его входом является набор из 100 карт признаков предыдущего слоя, поэтому каждая карта признаков имеет $3 \times 3 \times 100 = 900$ весов плюс член смещения. Поскольку данный слой включает 200 карт признаков, то он содержит $901 \times 200 = 180\,200$ параметров.

Наконец, третий (и последний) сверточный слой также имеет 3×3 ядра и его вход представляет собой набор из 200 карт признаков предыдущего слоя, а потому каждая карта признаков содержит $3 \times 3 \times 200 = 1\,800$ весов плюс член смещения. Так как этот слой включает 400 карт признаков, он обладает $1\,801 \times 400 = 720\,400$ параметрами. В общем и целом сеть CNN имеет $2\,800 + 180\,200 + 720\,400 = 903\,400$ параметров.

А теперь выясним, сколько памяти (по меньшей мере) потребует такая нейронная сеть при выработке прогноза для одиночного образца. Первым делом вычислим размеры карт признаков для всех слоев. Поскольку мы используем страйд 2 и дополнение SAME, в каждом слое горизонтальный и вертикальный размеры карт признаков делятся на 2 (с округлением в большую сторону, если необходимо), поэтому при входных каналах 200×300 пикселей картами признаков первого слоя будут 100×150 , второго слоя — 50×75 и третьего слоя — 25×38 . Так как 32 бита соответствуют 4 байтам и первый сверточный слой имеет 100 карт признаков, данный слой займет $4 \times 100 \times 150 \times 100 = 6$ миллионов байтов (около 5.7 Мбайт с учетом того, что 1 Мбайт = 1 024 Кбайт и 1 Кбайт = 1 024 байта). Второй слой потребует $4 \times 50 \times 75 \times 200 = 3$ миллионов байтов (около 2.9 Мбайт). Третий слой займет $4 \times 25 \times 38 \times 400 = 1\,520\,000$ байтов (около 1.4 Мбайт). Однако после расчета слоя память, занимаемая предыдущим слоем, может быть освобождена, поэтому если все хорошо оптимизировано, то потребуются лишь $6 + 3 = 9$ миллионов байтов (около 8.6 Мбайт) памяти (когда второй слой только что рассчитан, но память, занятая первым слоем, пока еще не освобождена). Но подождите, необходимо также добавить память, занимаемую параметрами сети CNN. Ранее мы выяснили, что сеть имеет 903 400 параметров, каждый из которых действует до 4 байтов, а потому добавляется 3 613 600 байтов (около 3.4 Мбайт). Общий объем требуемой памяти составляет (по меньшей мере) 12 613 600 байтов (около 12.0 Мбайт).

В заключение подсчитаем минимальный объем памяти, который требуется при обучении сети CNN на мини-пакете из 50 изображений. Во время обучения TensorFlow применяет обратное распространение, которое требует сохранения всех значений, вычисленных в течение прямого шага, до тех пор, пока не начнется обратный шаг. Следовательно, мы должны подсчитать общий объем памяти, который требуют

все слои для одиночного образца, и умножить его на 50! С этой точки давайте начнем считать в мегабайтах, а не в байтах. Ранее мы вычислили, что три слоя требуют соответственно 5.7, 2.9 и 1.4 Мбайт для каждого образца. В итоге имеем 10.0 Мбайт на образец. Таким образом, для 50 образцов общий объем памяти составляет 500 Мбайт. Добавим к нему объем памяти, необходимой для хранения входных изображений, который равен $50 \times 4 \times 200 \times 300 \times 3 = 36$ миллионов байтов (около 34.3 Мбайт), плюс объем памяти, требуемой для параметров модели, который составляет примерно 3.4 Мбайт (вычислен ранее), плюс объем памяти для градиентов (мы проигнорируем эту память, т.к. она будет постепенно освобождаться по мере продвижения обратного распространения вниз по слоям во время обратного прохода). В общей сложности мы имеем приблизительно $500.0 + 34.3 + 3.4 = 537.7$ Мбайт. И это на самом деле оптимистичный абсолютный минимум.

3. Если во время обучения сети CNN в вашем графическом процессоре случается нехватка памяти, то есть пять действий, которые вы могли бы предпринять, чтобы попытаться решить проблему (кроме покупки ГП с большим объемом памяти):
 - уменьшить размер мини-пакета;
 - понизить размерность, используя более высокий страйд в одном или большем числе слоев;
 - удалить один или больше слоев;
 - применить 16-битовые значения с плавающей точкой вместо 32-битовых;
 - распределить сеть CNN между множеством устройств.
4. Слой объединения по максимуму вообще не имеет параметров, тогда как у сверточного слоя их довольно много (см. предыдущие ответы).
5. Слой локальной нормализации ответа заставляет нейроны, которые наиболее сильно активируются, подавлять нейроны в том же самом местоположении, но в соседствующих картах признаков, что стимулирует разные карты признаков специализироваться за счет их отделения и принуждения к исследованию более широкого диапазона признаков. Прием обычно используется в нижних слоях, чтобы иметь более крупное объединение низкоуровневых признаков, на основе которых могут строиться верхние слои.

6. Главные новшества AlexNet в сравнении LeNet-5 были связаны с тем, что (1) AlexNet гораздо крупнее и глубже и (2) AlexNet укладывает сверточные слои прямо друг на друга, а не помещает поверх каждого сверточного слоя объединяющий слой. Основным новшеством GoogLeNet является ввод модулей начала, которые позволяют иметь намного более глубокую сеть, чем предшествующие архитектуры сетей CNN, с меньшим количеством параметров. Наконец, главное новшество ResNet — ввод обходящих связей, которые сделали возможным выход далеко за пределы 100 слоев. Вероятно, простоту и согласованность ResNet также можно считать новаторскими.

Решения упражнений 7, 8, 9 и 10 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 14. Рекуррентные нейронные сети

1. Ниже описано несколько приложений для сетей RNN.

- Для сети RNN типа “последовательность в последовательность”: прогнозирование погоды (или любого другого временного ряда), машинный перевод (с применением архитектуры “кодировщик–декодировщик”), снабжение субтитрами видеороликов, преобразование речи в текст, генерация музыки (или других последовательностей), идентификация аккордов в песне.
- Для сети RNN типа “последовательность в вектор”: классификация музыкальных фрагментов по жанрам, смысловой анализ рецензий на книги, прогнозирование слова, о котором думает пациент, страдающий афазией (нарушением речи), на основе сигналов от вживленных в мозг имплантатов, прогнозирование вероятности, что пользователь захочет смотреть фильм, на основе его хронологии просмотров (одна из множества возможных реализаций совместного фильтрования).
- Для сети RNN типа “вектор в последовательность”: подписание изображений, создание списка музыкальных произведений на основе встраивания текущего исполнителя, генерация мелодии на основе набора параметров, определение местоположения пешеходов на фотографии (например, на видеокадре из камеры беспилотного автомобиля).

2. Вообще говоря, если переводить предложение по одному слову за раз, то результат будет ужасным. Например, французское предложение “Je vous en prie” на английском языке выглядит как “You are welcome” (“Добро пожаловать”), но в результате перевода по одному слову за раз получится предложение “I you in pray” (“Я тебя в молитве”). Что-что? Гораздо лучше первоначально прочитав все предложение и затем переводить его. Простая сеть RNN типа “последовательность в последовательность” начала бы перевод предложения немедленно после чтения первого слова, тогда как сеть RNN типа “кодировщик–декодировщик” сначала прочитает все предложение и лишь затем его переведет. Тем не менее, можно было бы вообразить простую сеть RNN типа “последовательность в последовательность”, которая выдавала бы молчание всякий раз, когда не уверена в том, что сказать следующим (в точности как поступают люди-переводчики при переводе прямой передачи).
3. Для классифицирования видеороликов на основе визуального содержания возможная архитектура могла бы принимать (скажем) по одному кадру в секунду, прогонять каждый кадр через сверточную нейронную сеть, передавать выход сети CNN в сеть RNN типа “последовательность в вектор” и в заключение прогонять ее выход через многопеременный слой, предоставляя все вероятности классов. Для обучения вы могли бы просто использовать в качестве функции издержек перекрестную энтропию. Если вы хотите применять для классификации также и аудиодорожку, то могли бы преобразовать каждую секунду аудиодорожки в спектрограмму, передать эту спектрограмму в сеть CNN, а выход сети CNN передать в сеть RNN (наряду с соответствующим выходом другой сети CNN).
4. Построение сети RNN с использованием функции `dynamic_rnn()`, а не `static_rnn()`, дает несколько преимуществ.
 - Функция `dynamic_rnn()` основана на операции `while_loop()`, которая во время обратного распространения способна менять местами память ГП и память ЦП, избегая ошибок нехватки памяти.
 - Она вероятно легче в применении, поскольку может непосредственно брать одиночный тензор как вход и выход (покрывая все временные шаги), а не список тензоров (по одному на временной шаг). Нет нужды в укладывании, восстановлении или транспонировании.
 - Она генерирует меньший граф, который легче визуализировать в TensorBoard.

5. Обрабатывать входные последовательности переменной длины проще всего, устанавливая параметр `sequence_length` при вызове функций `static_rnn()` или `dynamic_rnn()`. Другой вариант предусматривает дополнение меньших входов (например, нулями), чтобы сделать их размер таким же, как у самого большого входа (данный вариант может оказаться быстрее, чем первый, если все входные последовательности имеют очень похожие длины). Если заранее известна длина каждой выходной последовательности, то для обработки выходных последовательностей переменной длины можно использовать параметр `sequence_length`. (Например, подумайте о сети RNN типа “последовательность в последовательность”, которая помечает каждый кадр в видеоролике мерой жестокости: выходная последовательность будет иметь ту же самую длину, что и входная последовательность.) Если вы не знаете заранее длину выходной последовательности, то можете применить трюк с дополнением: всегда выдавать последовательность того же самого размера, но игнорировать любые выходы, которые поступают после маркера конца последовательности (игнорируя их при вычислении функции издержек).
6. Распространенный способ распределения обучения и выполнения глубокой сети RNN между множеством ГП предусматривает размещение каждого слоя на своем ГП (см. главу 12).

Решения упражнений 7, 8 и 9 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 15. Автокодировщики

1. Ниже перечислено несколько главных задач, для которых применяются автокодировщики:
 - выделение признаков;
 - предварительное обучение без учителя;
 - понижение размерности;
 - порождающие модели;
 - обнаружение аномалий (автокодировщик, как правило, плох при восстановлении выбросов).

2. Если вы хотите обучить классификатор и имеете очень много помеченных обучающих данных, но лишь несколько тысяч помеченных образцов, тогда можете сначала обучить глубокий автокодировщик на полном наборе данных (помеченных и помеченных), затем повторно использовать нижнюю половину слоев для классификатора (т.е. слои вплоть до кодирующего включительно) и обучить классификатор с применением помеченных данных. Если помеченных данных мало, тогда при обучении классификатора, возможно, вы пожелаете заморозить повторно использованные слои.
3. Факт идеального восстановления входов вовсе не обязательно означает, что автокодировщик является хорошим; может быть, он просто является повышающим автокодировщиком, который научился копировать свои входы в кодирующий слой и затем в выходы. На самом деле, даже если кодирующий слой содержит единственный нейрон, то очень глубокий автокодировщик вполне возможно обучить сопоставлению каждого обучающего образца с отличающейся кодировкой (скажем, первый образец мог бы сопоставляться с 0.001, второй — с 0.002, третий — с 0.003 и т.д.). Этот автокодировщик мог бы выучить “наизусть”, каким образом восстанавливать правильный обучающий образец для каждой кодировки. Он бы идеально восстанавливал свои входы, фактически не узнавая какие-то полезные шаблоны в данных. На практике такое сопоставление вряд ли случится, но оно иллюстрирует то, что идеальные реконструкции совершенно не гарантируют узнавание автокодировщиком чего-нибудь полезного. Тем не менее, если он производит очень плохие реконструкции, тогда он практически обязательно является плохим автокодировщиком. Чтобы оценить производительность автокодировщика, можно измерить потерю из-за реконструкции (например, подсчитать MSE, средний квадрат разницы между выходами и входами). И снова высокая потеря из-за реконструкции сигнализирует о том, что автокодировщик плох, но низкая потеря из-за реконструкции не гарантирует, что он хорош. Вы должны также оценивать автокодировщик относительно того, для чего он будет применяться. Например, если он будет использоваться для предварительного обучения классификатора без учителя, тогда понадобится также оценить производительность классификатора.

4. Понижающий автокодировщик — это такой автокодировщик, у которого кодирующий слой меньше, чем входной и выходной слои. Если кодирующий слой больше, то автокодировщик становится повышающим. Основной риск чрезмерно понижающего автокодировщика заключается в том, что он может потерпеть неудачу с восстановлением входов. Главный риск повышающего автокодировщика связан с тем, что он может просто копировать входы в выходы, не узнавая о каком-либо полезном признаке.
5. Чтобы связать веса кодирующего слоя с соответствующими весами декодирующего слоя, нужно сделать веса декодировщика равными транспонированным весам кодировщика. Это наполовину уменьшает количество весов в модели, часто обеспечивая более быстрое схождение с меньшим объемом обучающих данных и снижая риск переобучения обучающим набором.
6. Распространенный прием визуализации признаков, которые узнал самый нижний слой многослойного автокодировщика, предусматривает вычерчивание весов каждого нейрона путем изменения формы каждого весового вектора для приведения к размеру входного изображения (скажем, в случае набора данных MNIST изменение формы весового вектора с [784] на [28, 28]). Чтобы визуализировать признаки, изученные более высокими слоями, необходимо отобразить обучающие образцы, которые в наибольшей степени активировали каждый нейрон.
7. Порождающая модель — это модель, способная случайным образом генерировать выходы, которые напоминают обучающие образцы. Например, после успешного обучения на наборе данных MNIST порождающая модель может применяться для случайного генерирования реалистичных изображений цифр. Выходное распределение обычно похоже на обучающие данные. Скажем, поскольку набор данных MNIST содержит много изображений каждой цифры, порождающая модель выдавала бы приблизительно такое же количество изображений каждой цифры. Некоторые порождающие модели могут быть параметризованы, генерируя только определенные виды выходов. Примером порождающего автокодировщика является вариационный автокодировщик.

Решения упражнений 8, 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Глава 16. Обучение с подкреплением

1. Обучение с подкреплением представляет собой область машинного обучения, направленную на создание агентов, которые способны предпринимать действия в среде способом, доводящим до максимума награды с течением времени. Существует много отличий между обучением с подкреплением, обыкновенным обучением с учителем и обучением без учителя. Вот некоторые отличия.
 - В обучении с учителем и без учителя целью обычно является нахождение шаблонов в данных и их использование для выработки прогнозов. В обучении с подкреплением цель заключается в нахождении хорошей политики.
 - В отличие от обучения с учителем агент не получает “правильный” ответ явным образом. Он должен обучаться методом проб и ошибок.
 - В отличие от обучения без учителя имеется определенная форма контроля через награды. Мы не указываем агенту, как выполнять работу, но уведомляем его, когда он делает успехи или терпит неудачу.
 - Агенту обучения с подкреплением нужно найти правильный баланс между исследованием среды в поисках новых способов получения наград и эксплуатацией источников наград, которые ему уже известны. В противоположность этому системы обучения с учителем и без учителя обычно не должны беспокоиться об исследовании; они просто потребляют предоставленные им обучающие данные.
 - В обучении с учителем и без учителя обучающие образцы, как правило, независимы (на самом деле они обычно перетасованы). В обучении с подкреплением последовательные наблюдения в общем случае независимыми *не* являются. Агент какое-то время может оставаться в той же самой области среды, прежде чем двигаться дальше, поэтому последовательные наблюдения будут сильно связанными друг с другом. В ряде случаев применяется память воспроизведения, чтобы гарантировать получение алгоритмом обучения достаточно независимых наблюдений.

2. Ниже описаны некоторые возможные приложения обучения с подкреплением, отличные от тех, что упоминались в главе 16.

Персонализация музыкальных произведений

Среда представляет собой персонализированное веб-радио пользователя. Агентом является программа, которая решает, какую песню воспроизводить следующей для данного пользователя. Возможными действиями будут воспроизведение любой песни из каталога (агент должен постараться выбрать песню, которая понравится пользователю) или воспроизведение рекламы (агент должен попытаться выбрать рекламу, которая заинтересует пользователя). Агент получает небольшую награду каждый раз, когда пользователь слушает песню, более крупную награду, когда пользователь слушает рекламу, отрицательную награду, когда пользователь пропускает песню или рекламу, и большую отрицательную награду, если пользователь покидает программу.

Маркетинг

Средой является отдел маркетинга вашей компании. Агент представляет собой программу, которая определяет, каким заказчикам следует отправлять рекламу по электронной почте, с учетом их профиля и истории покупок (для каждого заказчика есть два возможных действия: отправлять и не отправлять). Агент получает отрицательную награду за издержки рекламной кампании и положительную награду за оценочный доход, обусловленный этой кампанией.

Доставка товаров

Пусть агент управляет парком грузовиков доставки, принимая решения о том, какие товары должны загружаться в них со складов, куда они должны ехать, что с них нужно выгружать и т.д. Агент будет получать положительную награду за каждый доставленный вовремя товар и отрицательную награду за просроченную доставку.

3. При оценке ценности действия алгоритмы обучения с подкреплением, как правило, суммируют все награды, к которым привело это действие, придавая больший вес непосредственным наградам и меньший вес более поздним наградам (с учетом того, что действие оказывает большее влияние на ближайшее будущее, чем на отдаленное будущее). Чтобы

смоделировать это, на каждом временном шаге обычно применяется дисконтная ставка. Например, с дисконтной ставкой 0.9 при оценке ценности действия награда 100, полученная на два временных шага позже, учитывается только как $0.9^2 \times 100 = 81$. Можете думать о дисконтной ставке как о мере того, насколько будущее ценится в сравнении с настоящим. Если дисконтная ставка очень близка к 1, то будущее ценится практически одинаково с настоящим. Если она близка к 0, тогда имеют значение только непосредственные награды. Разумеется, это чрезвычайно влияет на оптимальную политику: если вы цените будущее, то можете быть готовы смириться с сильными немедленными страданиями ради шанса получить возможные награды, но если вы не цените будущее, тогда просто будете хватать любые непосредственные награды, которые сумеете найти, никогда не инвестируя в будущее.

4. Для измерения производительности агента обучения с подкреплением вы можете просто суммировать получаемые им награды. В симулированной среде вы будете прогонять множество эпизодов с просмотром итоговых наград, которые агент получает в среднем (и возможно минимум, максимум, стандартное отклонение и т.д.).
5. Проблема присваивания коэффициентов доверия заключается в том, что когда агент обучения с подкреплением получает награду, у него нет прямого способа узнать, какие из его предшествующих действий способствовали этой награде. Она обычно возникает при наличии большой задержки между действием и результирующими наградами (скажем, во время Atari-игры Pong с момента удара агента по мячу до момента выигрыша им очка может пройти несколько десятков временных шагов). Чтобы смягчить проблему, необходимо по возможности предоставлять агенту краткосрочные награды. Как правило, это требует априорных знаний о задаче. Например, если мы хотим построить агента, который будет учиться играть в шахматы, то вместо выдачи ему награды, когда он выигрывает игру, можно было бы давать награду при каждом взятии одной из фигур соперника.
6. Агент может часто оставаться в той же самой области среды на какое-то время, а потому весь его опыт за этот период будет крайне однообразным. Итогом может быть привнесение некоторого смещения в алгоритм обучения. Он может подстраивать свою политику для данной

области среды, но не будет выполняться хорошо после того, как покинет эту область. Для решения указанной проблемы можно использовать память воспроизведения. Вместо применения при обучении только большей части непосредственного опыта агент будет обучаться на основе буфера своего прошлого опыта, недавнего и не такого недавнего (возможно, поэтому нам по ночам снятся сны: чтобы воспроизвести опыт, накопленный за день, и лучше учиться на нем?).

7. Алгоритм RL вне политики узнает ценность оптимальной политики (т.е. сумму наград с учетом дисконтной ставки, которую можно ожидать для каждого состояния, если агент действует оптимально), пока агент следует другой политике. Хорошим примером такого алгоритма служит Q-обучение. В противоположность этому алгоритм внутри политики (on-policy) узнает ценность политики, которую агент фактически исполняет, включая исследование и эксплуатацию.

Решения упражнений 8, 9 и 10 ищите в тетрадах Jupyter, доступных по адресу <https://github.com/ageron/handson-ml>.

Контрольный перечень для проекта машинного обучения

Данный контрольный перечень помогает вести проекты машинного обучения. Существует восемь главных шагов.

1. Постановка задачи и выяснение общей картины.
2. Получение данных.
3. Исследование данных для понимания их сущности.
4. Подготовка данных с целью лучшего обнаружения лежащих в основе шаблонов для алгоритмов машинного обучения.
5. Исследование множества разных моделей и составление окончательного списка лучших из них.
6. Точная настройка моделей и их объединение в наилучшее решение.
7. Представление своего решения.
8. Запуск, наблюдение и сопровождение системы.

Вполне очевидно, вы вольны приспособлять этот контрольный перечень к имеющимся потребностям.

Постановка задачи и выяснение общей картины

1. Определите цель в бизнес-понятиях.
2. Как будет использоваться ваше решение?
3. Как выглядят текущие решения/обходные пути (если они есть)?
4. Как вы должны сформулировать задачу (обучение с учителем/без учителя, динамическое/автономное обучение и т.д.)?

5. Как вы должны измерять производительность?
6. Согласована ли мера производительности с бизнес-целью?
7. Какой будет минимальная производительность, необходимая для достижения бизнес-цели?
8. Каковы сопоставимые задачи? Можете ли вы воспользоваться имеющимся опытом или инструментами?
9. Доступен ли человеческий опыт?
10. Как бы вы решали задачу вручную?
11. Перечислите предположения, сделанные вами (или другими) до сих пор.
12. По возможности проверьте предположения.

Получение данных

Примечание: автоматизируйте как можно больше действий, чтобы легко получать свежие данные.

1. Перечислите нужные данные и укажите, сколько их необходимо.
2. Найдите и документируйте места, где можно получить данные.
3. Выясните объем пространства, которое займут данные.
4. Ознакомьтесь с правовыми обязательствами и при необходимости получите разрешение.
5. Получите права доступа.
6. Создайте рабочее пространство (с хранилищем достаточного объема).
7. Получите данные.
8. Представьте данные в формате, который позволяет легко манипулировать данными (не изменяя сами данные).
9. Удостоверьтесь в том, что конфиденциальная информация удалена или защищена (например, анонимизирована).
10. Выясните размер и тип данных (временной ряд, выборка, географические данные и т.д.).
11. Произведите выборку испытательного набора, отложите его в сторону и никогда не смотрите в него (никакого подглядывания за данными!).

Исследование данных

Примечание: для этих шагов попробуйте получить сведения от экспертов на местах.

1. Создайте копию данных для исследования (при необходимости производя выборку, чтобы получить поддающийся управлению размер).
2. Создайте тетрадь Jupyter для сохранения записи об исследовании данных.
3. Изучите каждый атрибут и его характеристики:
 - имя;
 - тип (категориальный, целочисленный/с плавающей точкой, ограниченный/неограниченный, текстовый, структурированный и т.д.);
 - процент отсутствующих значений;
 - зашумленность и тип шума (стохастический, выбросы, ошибки округления и т.д.);
 - возможная польза для задачи;
 - тип распределения (гауссово, равномерное, логарифмическое и т.д.).
4. Для задач обучения с учителем идентифицируйте целевой атрибут (атрибуты).
5. Визуализируйте данные.
6. Исследуйте взаимосвязи между атрибутами.
7. Подумайте, как бы вы решали задачу вручную.
8. Идентифицируйте перспективные трансформации, которые возможно захотите применить.
9. Идентифицируйте дополнительные данные, которые могут быть полезными (см. раздел “Получение данных” ранее в приложении).
10. Документируйте все, что вы узнали.

Подготовка данных

Примечания.

- Работайте с копиями данных (сохраните исходный набор данных неза-
тронутым).
- Напишите функции для всех применяемых трансформаций данных по
следующим пяти причинам.
 - Так вы сможете легко подготовить данные в следующий раз, когда
получите свежие данные.
 - Так вы сможете применить эти трансформации в будущих проектах.
 - Чтобы очистить и подготовить испытательный набор.
 - Чтобы очистить и подготовить новые образцы данных после того,
как решение начнет существовать.
 - Чтобы облегчить трактовку подготовительных вариантов как гипер-
параметров.
- 1. Очистите данные.
 - Исправьте или удалите выбросы (необязательно).
 - Заполните отсутствующие значения (например, нулевым, средним,
медианным или каким-то другим значением) или отбросьте их стро-
ки (либо столбцы).
- 2. Выберите признаки (необязательно).
 - Отбросьте атрибуты, которые не несут в себе никакой полезной ин-
формации для задачи.
- 3. Сконструируйте признаки, где это необходимо.
 - Дискретизируйте непрерывные признаки.
 - Разбейте признаки на составные части (например, категориальные,
дата/время и т.д.).
 - Добавьте перспективные трансформации признаков (например,
 $\log(x)$, \sqrt{x} , x^2 и т.д.).
 - Агрегируйте признаки в перспективные новые признаки.
- 4. Масштабируйте признаки: стандартизируйте или нормализуйте их.

Составление окончательного списка перспективных моделей

Примечания.

- В случае, когда данные гигантские, может возникнуть желание выбрать обучающие наборы меньших размеров, чтобы иметь возможность обучения множества разных моделей за разумное время (надо учесть, что это штрафует сложные модели, такие как крупные нейронные сети или случайные леса).
 - И снова постарайтесь максимально автоматизировать указанные шаги.
1. Обучите множество созданных на скорую руку моделей из разных категорий (например, линейную, наивную байесовскую, SVM, случайный лес, нейронную сеть и т.д.), используя стандартные параметры.
 2. Измерьте и сравните их производительность.
 - К каждой модели примените перекрестную проверку с контролем по N блокам и вычислите среднее значение и стандартное отклонение меры производительности для N блоков.
 3. Проанализируйте наиболее значимые переменные для каждого алгоритма.
 4. Проанализируйте типы ошибок, допускаемых моделями.
 - Какие данные использовал бы человек, чтобы избежать этих ошибок?
 5. Проведите быстрый цикл выбора и конструирования признаков.
 6. Проведите одну или две более быстрых итерации предшествующих пяти шагов.
 7. Составьте окончательный список из первых трех-пяти самых перспективных моделей, отдавая предпочтение моделям, которые допускают разные типы ошибок.

Точная настройка системы

Примечания.

- Для этого шага вы пожелаете задействовать как можно больше данных, особенно с приближением к концу точной настройки.
 - Как всегда, автоматизируйте все, что возможно.
1. Проведите точную настройку гиперпараметров с применением перекрестной проверки.
 - Трактуйте свои трансформации данных как гиперпараметры, в особенности, когда вы в них не уверены (например, чем должны заменяться отсутствующие значения — нулевым или медианным значением? Или же нужно просто отбрасывать строки?).
 - Если только значений гиперпараметров, которые подлежат исследованию, совсем немного, то отдавайте предпочтение случайному поиску перед решетчатым поиском. Когда обучение проходит очень долго, вы можете предпочесть подход с байесовской оптимизацией (например, априорно используя гауссов процесс, как описано в работе Джаспера Сноека, Хьюго Ларошеля и Райана Адамса¹).
 2. Испытайте ансамблевые методы. Комбинация лучших моделей часто будет работать более эффективно, чем индивидуальные модели.
 3. После обретения уверенности в отношении финальной модели измерьте ее производительность на испытательном наборе, чтобы оценить ошибку обобщения.



Не подстраивайте модель после измерения ошибки обобщения: вы просто начнете переобучать ее испытательным набором.

¹ “Practical Bayesian Optimization of Machine Learning Algorithms” (“Практическая байесовская оптимизация алгоритмов машинного обучения”), Д. Сноек, Х. Ларошель и Р. Адамс (2012 год) (<https://goo.gl/PEFfGr>).

Представление своего решения

1. Документируйте все, что вы сделали.
2. Создайте симпатичную презентацию.
 - Удостоверьтесь в том, что сначала прояснили общую картину.
3. Объясните, почему ваше решение достигает бизнес-цели.
4. Не забывайте представлять интересные моменты, которые вы попутно заметили.
 - Опишите, что работает, а что нет.
 - Перечислите свои предположения и ограничения системы.
5. Обеспечьте, чтобы ваши основные заключения передавались через привлекательные визуализации или простые для запоминания фразы (например, “медианный доход — это прогнозатор номер один цен на дома”).

Запуск!

1. Подготовьте свое решение для помещения в производственную среду (подключите к источникам производственных данных, напишите модульные тесты и т.д.).
2. Напишите код наблюдения, чтобы проверять реальную производительность системы через регулярные промежутки времени и подавать сигналы тревоги, когда она падает.
 - Остерегайтесь также и медленного снижения производительности: с развитием данных модели склонны к “разложению”.
 - Измерение производительности может требовать конвейера человеческой оценки (например, через службу краудсорсинга).
 - Наблюдайте также за качеством входных данных (например, неисправный датчик может посылать произвольные значения или выходные данные другой команды могут стать устаревшими). Это особенно важно для систем динамического обучения.
3. Повторно обучайте свои модели на регулярной основе с применением свежих данных (автоматизируйте все, что только можно).

Двойственная задача SVM

Чтобы понять *двойственность* (*duality*), сначала необходимо освоить метод *множителей Лагранжа* (*Lagrange multipliers method*). Общая идея заключается в трансформации цели условной оптимизации в безусловную цель путем перемещения ограничений в *целевую функцию* (*objective function*). Давайте рассмотрим простой пример. Предположим, что вы хотите найти значения x и y , которые сводят к минимуму функцию $f(x, y) = x^2 + 2y$ при наличии *ограничения в виде равенства* (*equality constraint*): $3x + 2y + 1 = 0$. Используя метод множителей Лагранжа, мы начинаем с определения новой функции, которая называется *лагранжианом* (*Lagrangian*) или *функцией Лагранжа* (*Lagrange function*): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Каждое ограничение (в этом случае только одно) умножается на новую переменную, называемую множителем Лагранжа, и вычитается из первоначальной цели.

Жозеф Луи Лагранж показал, что если (\hat{x}, \hat{y}) является решением задачи условной оптимизации, тогда должно существовать $\hat{\alpha}$, такое что $(\hat{x}, \hat{y}, \hat{\alpha})$ представляет собой *стационарную точку* (*stationary point*) лагранжиана (стационарная точка — это точка, где все частные производные равны нулю). Другими словами, мы можем вычислить частные производные $g(x, y, \alpha)$ в отношении x , y и α , найти точки, где все производные равны нулю, и решения задачи условной оптимизации (если они существуют) должны быть среди таких стационарных точек. В данном примере частные производные таковы:

$$\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$$

Когда все частные производные равны 0, мы обнаруживаем, что $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, откуда легко находим, что $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$ и $\hat{\alpha} = 1$.

Это единственная стационарная точка, и поскольку она соблюдает ограничение, то должна быть решением задачи условной оптимизации.

Однако такой метод применим только к ограничениям в виде равенства. К счастью, при некоторых условиях регулярности (соблюдаемых целями SVM) метод может быть обобщен также на *ограничения в виде неравенства* (*inequality constraint*), например, $3x + 2y + 1 \geq 0$. *Обобщенный лагранжиан* (*generalized Lagrangian*) для задачи классификации с жестким зазором приведен в уравнении В.1, где переменные $\alpha^{(i)}$ называются множителями *Каруша–Куна–Таккера* (*Karush–Kuhn–Tucker — KKT*), и они должны быть больше или равны нулю.

Уравнение С.1. Обобщенный лагранжиан для задачи классификации с жестким зазором

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2} \mathbf{w}^T \cdot \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} \left(t^{(i)} (\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) - 1 \right)$$

с $\alpha^{(i)} \geq 0$ для $i = 1, 2, \dots, m$.

Как и с методом множителей Лагранжа, вы можете вычислить частные производные и найти стационарные точки. Если решение есть, тогда оно непременно будет среди стационарных точек $(\hat{\mathbf{w}}, \hat{b}, \hat{\alpha})$, которые удовлетворяют *условиям KKT* (KKT condition):

- Соблюдают ограничения задачи: $t^{(i)} \left((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b} \right) \geq 1$ для $i = 1, 2, \dots, m$.
- Обеспечивают, что $\hat{\alpha}^{(i)} \geq 0$ для $i = 1, 2, \dots, m$.
- Либо $\hat{\alpha}^{(i)} = 0$, либо i -тое ограничение должно быть *активным ограничением*, т.е. удерживаться равенством: $t^{(i)} \left((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b} \right) = 1$. Такое условие называется условием *дополняющей нежесткости* (*complementary slackness*). Оно подразумевает, что либо $\hat{\alpha}^{(i)} = 0$, либо i -тый образец находится на границе (является опорным вектором).

Обратите внимание, что условия KKT — это необходимые условия для того, чтобы стационарная точка была решением задачи условной оптимизации. При определенных обстоятельствах они также являются достаточными условиями. К счастью, задача оптимизации SVM удовлетворяет этим условиям, так что любая стационарная точка, которая соответствует условиям KKT, гарантированно будет решением задачи условной оптимизации.

Мы можем вычислить частные производные обобщенного лагранжиана в отношении \mathbf{w} и b с помощью уравнения В.2.

Уравнение В.2. Частные производные обобщенного лагранжиана

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^m \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\frac{\partial}{\partial b} \mathcal{L}(\mathbf{w}, b, \alpha) = - \sum_{i=1}^m \alpha^{(i)} t^{(i)}$$

Когда эти частные производные равны 0, мы имеем уравнение В.3.

Уравнение В.3. Свойства стационарных точек

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} = 0$$

Если мы включим полученные результаты в определение обобщенного лагранжиана, то некоторые члены исчезнут, и получится уравнение В.4.

Уравнение В.4. Двойственная форма задачи SVM

$$\mathcal{L}(\hat{\mathbf{w}}, \hat{b}, \alpha) = \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

с $\alpha^{(i)} \geq 0$ для $i = 1, 2, \dots, m$.

Цель теперь в том, чтобы найти вектор $\hat{\mathbf{a}}$, который сводит к минимуму эту функцию с $\alpha^{(i)} \geq 0$ для всех образцов. Такая задача условной оптимизации является искомой двойственной задачей.

После нахождения оптимального $\hat{\mathbf{a}}$ можно вычислить $\hat{\mathbf{w}}$, используя первую строчку уравнения В.3. Чтобы вычислить \hat{b} , можно задействовать тот факт, что опорный вектор должен обеспечивать $t^{(i)}(\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b} = 1$, поэтому если опорным вектором является k -тый образец (т.е. $\hat{\alpha}^{(k)} > 0$), тогда его можно применять для вычисления $\hat{b} = t^{(k)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(k)}$. Тем не менее, предпочтение часто отдается вычислению среднего по всем опорным векторам, чтобы получить более устойчивое и точное значение, как показано в уравнении В.5.

Уравнение В.5. Оценка члена смещения с использованием двойственной формы

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m \left[t^{(i)} - \hat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)} \right]$$

Автоматическое дифференцирование

В этом приложении объясняется работа средства автоматического дифференцирования TensorFlow и приводятся сравнения с другими решениями.

Предположим, мы определили функцию $f(x, y) = x^2y + y + 2$, и нужны ее частные производные $\frac{\partial f}{\partial x}$ и $\frac{\partial f}{\partial y}$, обычно для выполнения градиентного спуска (или какого-то другого алгоритма оптимизации). Основными вариантами являются ручное дифференцирование, символическое дифференцирование, численное дифференцирование, автоматическое дифференцирование в прямом режиме и автоматическое дифференцирование в обратном режиме. Рассмотрим по очереди каждый вариант.

Ручное дифференцирование

Первый подход — взять карандаш и лист бумаги и задействовать свое знание исчисления с целью извлечения частных производных вручную. Для только что определенной функции $f(x, y)$ задача не слишком трудна; необходимо лишь воспользоваться следующими пятью правилами.

- Производная константы равна 0.
- Производная λx равна λ (где λ — константа).
- Производная x^λ равна $\lambda x^{\lambda-1}$, так что производной x^2 будет $2x$.
- Производная суммы функций равна сумме производных этих функций.
- Производная λ раз функции равна λ раз ее производной.

Учитывая указанные правила, мы можем получить уравнение Г.1.

Уравнение Г.1. Частные производные функции $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2 y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2 y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Такой подход может стать крайне утомительным для более сложных функций, увеличивая риск допустить ошибку. Хорошая новость заключается в том, что выведение математических уравнений для частных производных вроде показанных выше может быть автоматизировано посредством процесса, который называется символическим дифференцированием.

Символическое дифференцирование

На рис. Г.1 показано, как символическое дифференцирование работает для более простой функции $g(x, y) = 5 + xy$. Граф для этой функции приведен слева. После символического дифференцирования мы получаем граф справа, который представляет частную производную $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (частную производную относительно y можно получить похожим способом).

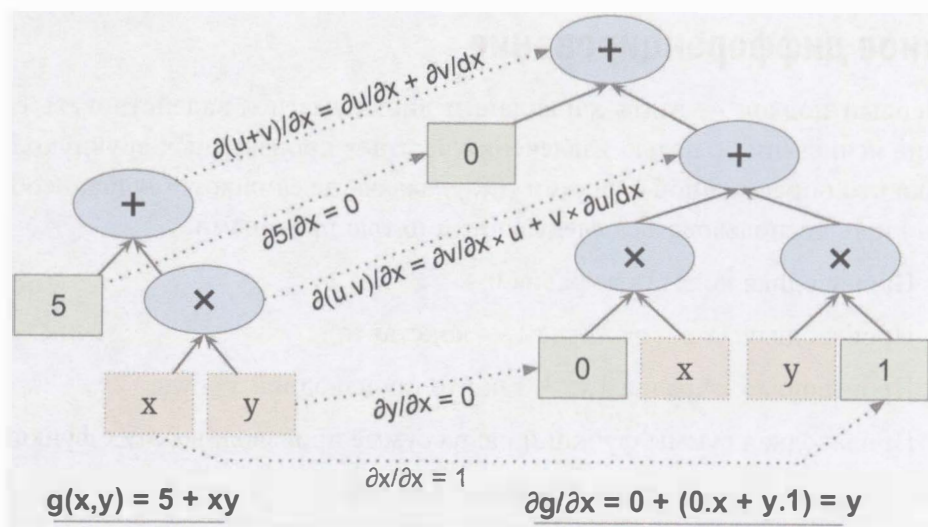


Рис. Г.1. Символическое дифференцирование

Алгоритм начинает с взятия частной производной листовых узлов. Узел константы (5) возвращает константу 0, т.к. производная константы всегда равна 0. Узел переменной x возвращает константу 1, поскольку $\frac{\partial x}{\partial x} = 1$, а узел переменной y возвращает константу 0, потому что $\frac{\partial y}{\partial x} = 0$ (если бы мы искали частную производную относительно y , то ситуация была бы обратной).

Теперь у нас есть все необходимое для перемещения графа в узел умножения внутри функции g . Исчисление говорит нам о том, что производной произведения двух функций u и v является $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + v \times \frac{\partial u}{\partial x}$. Следовательно, мы можем построить крупную часть графа справа, представляющую $0 \times x + y \times 1$.

Наконец, мы можем заняться узлом сложения в функции g . Как упоминалось ранее, производная суммы функций равна сумме производных этих функций. Таким образом, нам нужно лишь создать узел сложения и подключить его к готовым частям графа. Мы получаем надлежащую частную производную: $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

Однако граф может быть упрощен (и значительно). К нему можно применить несколько тривиальных действий отсечения, чтобы избавиться от всех излишних операций, и получить намного меньший граф с только одним узлом: $\frac{\partial g}{\partial x} = y$.

В данном случае добиться упрощения удалось довольно легко, но для более сложных функций символическое дифференцирование может давать гигантские графы, трудно поддающиеся упрощению, и приводить к субоптимальной производительности. Но важнее всего то, что символическое дифференцирование не может работать с функциями, определенными с помощью произвольного кода — например, со следующей функцией, которая обсуждалась в главе 9:

```
def my_func(a, b):  
    z = 0  
    for i in range(100):  
        z = a * np.cos(z + i) + z * np.sin(b - i)  
    return z
```

Численное дифференцирование

Простейшее решение предусматривает вычисление приближенных значений производных численным способом. Вспомните, что производная $h'(x_0)$

функции $h(x)$ в точке x_0 представляет собой наклон функции в этой точке, или более точно — уравнение Г.2.

Уравнение Г.2. Производная функции $h(x)$ в точке x_0

$$\begin{aligned} h'(x_0) &= \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0} \\ &= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon} \end{aligned}$$

Следовательно, для вычисления частной производной $f(x, y)$ относительно x в точке $x = 3$ и $y = 4$ мы просто подсчитываем $f(3 + \varepsilon, 4) - f(3, 4)$ и делим результат на ε , используя очень маленькое значение ε . Именно это делает показанный ниже код:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

К сожалению, результат не является точным (и он становится хуже для более сложных функций). Корректными результатами должны быть соответственно 24 и 10, но взамен мы получаем вот что:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Обратите внимание, что для вычисления обеих частных производных мы обязаны вызвать `f()`, по меньшей мере, три раза (в предыдущем коде мы вызываем ее четыре раза, но код можно было бы оптимизировать). Если бы было 1000 параметров, тогда пришлось бы вызывать `f()` минимум 1001 раз. При работе с крупными нейронными сетями метод численного дифференцирования становится слишком неэффективным.

Тем не менее, численное дифференцирование настолько просто реализовать, что оно становится прекрасным инструментом для проверки коррект-

ности реализации других методов. Например, если оно противоречит производной функции, полученной вручную, то функция, вероятно, содержит ошибку.

Автоматическое дифференцирование в прямом режиме

Автоматическое дифференцирование в прямом режиме — это ни численное дифференцирование, ни символическое дифференцирование, однако в некотором смысле оно является их любимым детищем. Автоматическое дифференцирование в прямом режиме полагается на **дуальные числа**, которые (странно, но притягательно) представляют собой числа в форме $a + b\epsilon$, где a и b — вещественные числа, а ϵ — бесконечно малое число, такое, что $\epsilon^2 = 0$ (но $\epsilon \neq 0$).

Вы можете представлять себе дуальное число $42 + 24\epsilon$ как что-то похожее на $42.0000\cdots 000024$ с бесконечным количеством нулей (разумеется, это упрощение предназначено лишь для того, чтобы вы уловили идею о том, чем являются дуальные числа). Дуальное число представляется в памяти в виде пары значений с плавающей точкой. Скажем, $42 + 24\epsilon$ представлено парой $(42.0, 24.0)$.

Дуальные числа можно суммировать, умножать и т.д., как демонстрируется в уравнении Г.3.

Уравнение Г.3. Несколько операций с дуальными числами

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Главное, можно показать, что $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, поэтому вычисление $h(a + \epsilon)$ дает сразу $h(a)$ и производную $h'(a)$. На рис. Г.2 видно, как автоматическое дифференцирование в прямом режиме вычисляет частную производную $f(x, y)$ относительно x в точке $x = 3$ и $y = 4$. Необходимо лишь вычислить $f(3 + \epsilon, 4)$; результатом будет дуальное число, первый компонент которого равен $f(3, 4)$, а второй — $\frac{\partial f}{\partial x}(3, 4)$.

Чтобы вычислить $\frac{\partial f}{\partial y}(3, 4)$, мы должны снова пройти через граф, но на этот раз с $x = 3$ и $y = 4 + \epsilon$.

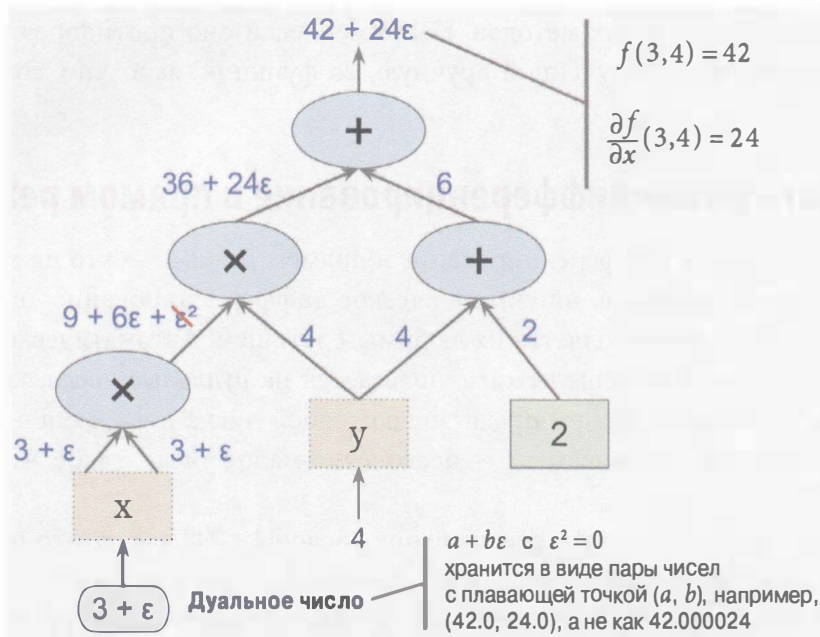


Рис. Г.2. Автоматическое дифференцирование в прямом режиме

Таким образом, автоматическое дифференцирование в прямом режиме гораздо точнее численного дифференцирования, но оно страдает от того же самого серьезного недостатка: если бы было 1000 параметров, тогда для вычисления всех частных производных потребовалось бы 1000 раз проходить через граф. Именно здесь блестяще себя показывает автоматическое дифференцирование в обратном режиме: оно способно вычислить все частные производные всего за два прохода через граф.

Автоматическое дифференцирование в обратном режиме

Автоматическое дифференцирование в обратном режиме — это решение, реализованное библиотекой TensorFlow. Оно сначала проходит через граф в прямом направлении (т.е. от входов к выходам), чтобы вычислить значение каждого узла. Затем оно осуществляет второй проход, но в противоположном направлении (т.е. от выходов к входам), чтобы вычислить все частные производные. На рис. Г.3 представлен второй проход. Во время первого прохода были вычислены значения всех узлов, начиная с $x = 3$ и $y = 4$. Эти значения показаны в правом нижнем углу каждого узла (например, $x \times x = 9$). Ради ясности узлы помечены от n_1 до n_7 . Выходным узлом является n_7 : $f(3, 4) = n_7 = 42$.

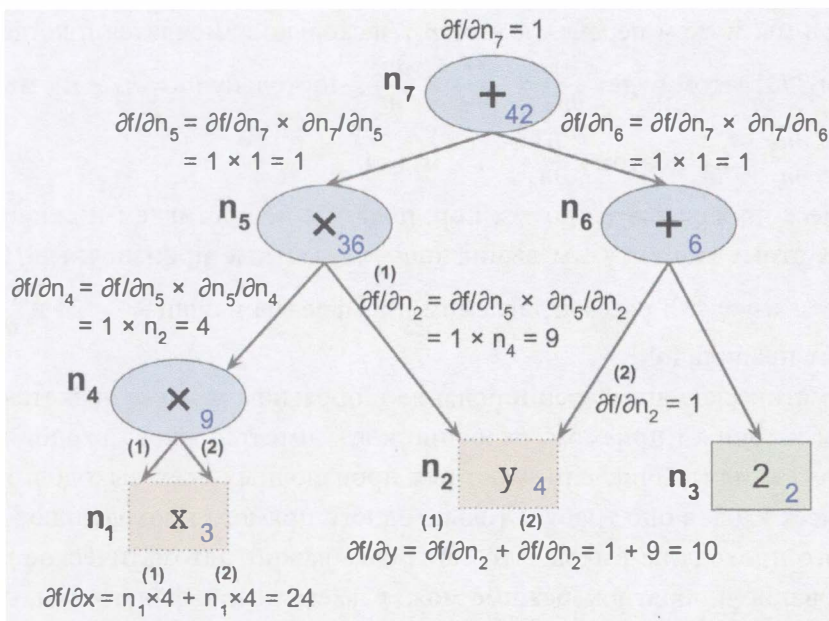


Рис. Г.3. Автоматическое дифференцирование в обратном режиме

Идея заключается в том, чтобы постепенно двигаться вниз по графу, вычисляя частную производную $f(x, y)$ относительно каждого последовательно узла, пока не будут достигнуты узлы переменных. Для этого автоматическое дифференцирование в обратном режиме в большой степени полагается на **цепное правило** (*chain rule*), приведенное в уравнении Г.4.

Уравнение Г.4. Цепное правило

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Поскольку n_7 является выходным узлом, $f = n_7$, то самоочевидно, что $\frac{\partial f}{\partial n_7} = 1$.

Продолжаем двигаться вниз по графу к n_5 : насколько изменяется f , когда изменяется n_5 ? Ответом будет $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. Мы уже знаем, что $\frac{\partial f}{\partial n_7} = 1$, поэтому нам необходимо лишь $\frac{\partial n_7}{\partial n_5}$. Так как n_7 просто дает сумму $n_5 + n_6$, мы находим, что $\frac{\partial n_7}{\partial n_5} = 1$, а потому $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Теперь мы можем перейти к узлу n_4 : насколько изменяется f , когда изменяется n_4 ? Ответом будет $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Поскольку $n_5 = n_4 \times n_2$, мы находим, что $\frac{\partial n_5}{\partial n_4} = n_2$, поэтому $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

Процесс продолжается до тех пор, пока мы не достигнем нижней части графа. К этому моменту мы вычислили все частные производные $f(x, y)$ в точке $x = 3$ и $y = 4$. В рассматриваемом примере мы нашли $\frac{\partial f}{\partial x} = 24$ и $\frac{\partial f}{\partial y} = 10$. Выглядит правильно!

Автоматическое дифференцирование в обратном режиме является очень мощным и точным приемом, особенно когда имеется много входов и мало выходов, т.к. для вычисления частных производных всех выходов относительно всех входов оно требует только одного прямого прохода плюс одного обратного прохода на выход. Что еще более важно, автоматическое дифференцирование в обратном режиме может иметь дело с функциями, определенными в произвольном коде. Оно также способно обрабатывать функции, которые не являются полностью дифференцируемыми, при условии, что вычисление частных производных запрашивается в точках, где функции дифференцируемы.



Если вы реализовали новый тип операции в TensorFlow и хотите сделать его совместимым с автоматическим дифференцированием, тогда вам потребуется предоставить функцию, которая строит подграф для вычисления своих частных производных относительно входов. Например, пусть вы реализовали функцию, вычисляющую квадрат своего входа, $f(x) = x^2$. В таком случае вы должны предоставить соответствующую производную функцию $f'(x) = 2x$. Обратите внимание, что эта функция вовсе не вычисляет численный результат, а взамен строит подграф, который будет (позже) вычислять результат. Прием очень полезен, потому что делает возможным вычисление градиентов от градиентов (для вычисления производных второго порядка или даже производных более высоких порядков).

Другие популярные архитектуры искусственных нейронных сетей

В этом приложении мы представим краткий обзор нескольких исторически важных архитектур нейронных сетей, которые в наши дни используются намного реже, чем глубокие многослойные перцептроны (глава 10), сверточные нейронные сети (глава 13), рекуррентные нейронные сети (глава 14) или автокодировщики (глава 15). Они часто упоминаются в литературе, и некоторые из них все еще применяются во многих приложениях, а потому такие архитектуры полезно знать. Кроме того, мы обсудим *глубокие сети доверия* (*Deep Belief Net* — *DBN*), которые отражали современное состояние глубокого обучения до начала 2010-х годов. Они по-прежнему являются предметом очень активных исследований, поэтому в ближайшем будущем вполне могут вернуться с удвоенной силой.

Сети Хопфилда

Сети Хопфилда (*Hopfield network*) впервые были введены У. Литтлом в 1974 году и затем популяризированы Дж. Хопфилдом в 1982 году. Они являются сетями *с ассоциативной памятью* (*associative memory network*): вы сначала обучаете сеть некоторым образцам, после чего при получении нового образца она (надо надеяться) выдаст наиболее близкий узнанный образец. Это сделало такие сети полезными в частности для распознавания образов до того, как их превзошли другие подходы. Первым делом вы обучаете сеть, показывая ей примеры изображений образа (каждый двоичный пиксель сопоставляется с одним нейроном), затем демонстрируете ей новое изображение образа, и после нескольких итераций сеть выдает ближайший узнанный образ.

Они представляют собой полносвязные графы (рис. Д.1), т.е. каждый нейрон связан со всеми остальными нейронами. Обратите внимание, что изображения имеют 6×6 пикселей, поэтому нейронная сеть слева должна содержать 36 нейронов (и 648 связей), но для зрительной ясности показана гораздо меньшая сеть.

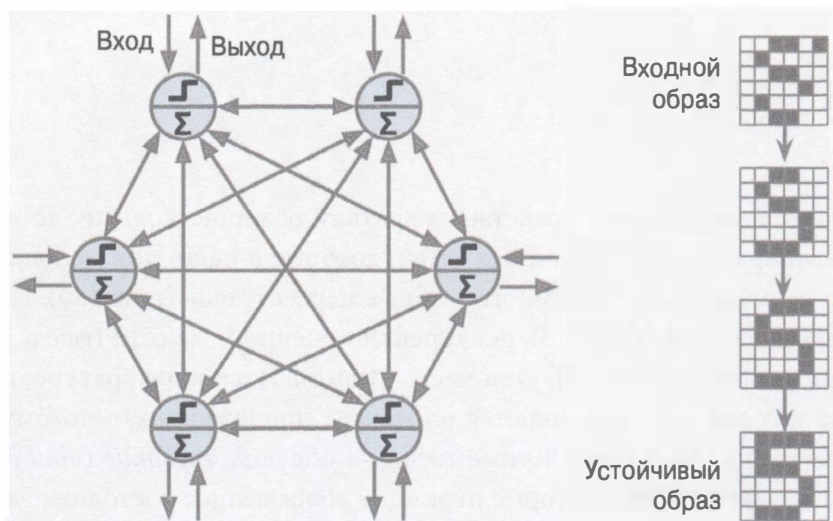


Рис. Д.1. Сеть Хопфилда

Алгоритм обучения работает за счет использования правила Хебба: для каждого обучающего изображения вес связи между двумя нейронами увеличивается, если соответствующие пиксели оба включены или оба выключены, но уменьшается, когда один пиксель включен, а другой выключен.

Чтобы показать сети новое изображение, вы просто активируете нейроны, которые соответствуют активным пикселям. Далее сеть рассчитывает выход каждого нейрона, что дает вам новое изображение. Затем вы берете это новое изображение и повторяете весь процесс. Через некоторое время сеть достигнет устойчивого состояния. Обычно оно соответствует обучающему изображению, которое больше всех напоминает входное изображение.

С сетями Хопфилда ассоциирована так называемая *энергетическая функция* (*energy function*). На каждой итерации энергия убывает, а потому сеть гарантированно со временем стабилизируется в низкоэнергетическом состоянии. Алгоритм обучения подстраивает веса способом, который снижает энергетический уровень обучающих образов, поэтому сеть, вероятно, стабилизируется в одной из таких низкоэнергетических конфигураций.

К сожалению, отдельные образы, отсутствовавшие в обучающем наборе, также заканчиваются с низкой энергией, поэтому сеть иногда стабилизируется в конфигурации, которая не изучалась. Такие образы называются *ложными образами* (*spurious pattern*).

Еще один серьезный недостаток сетей Хопфилда заключается в том, что они не особенно хорошо масштабируются — емкость их памяти составляет приблизительно 14% от количества нейронов. Например, чтобы классифицировать изображения 28×28 , понадобится сеть Хопфилда, имеющая 784 полносвязных нейрона и 306 936 весов. Итоговая сеть будет способна узнать около 110 разных образов (14% от 784). Слишком много параметров для такой небольшой памяти.

Машины Больцмана

Машины Больцмана (*Boltzmann machine*) были изобретены в 1985 году Джеффри Хинтоном и Терренсом Сейновски. Как и сети Хопфилда, они являются полносвязными искусственными нейронными сетями, но основаны на *стохастических нейронах* (*stochastic neuron*): вместо применения детерминированной ступенчатой функции при решении, какое значение выдавать, эти нейроны выдают 1 с некоторой вероятностью и 0 в противном случае. Вероятностная функция, используемая такими сетями ANN, базируется на распределении Больцмана (применяемом в статистической механике), отсюда и такое название. Уравнение Д.1 дает вероятность того, что нейрон будет выдавать 1.

Уравнение Д.1. Вероятность того, что i -тый нейрон будет выдавать 1

$$p(s_i^{\text{(следующий шаг)} = 1}) = \sigma \left(\frac{\sum_{j=1}^N w_{i,j} s_j + b_i}{T} \right)$$

- s_j — состояние j -того нейрона (0 или 1).
- $w_{i,j}$ — вес связи между i -тым и j -тым нейронами. Обратите внимание, что $w_{i,i} = 0$.
- b_i — член смещения i -того нейрона. Мы можем реализовать этот член, добавив в сеть нейрон смещения.
- N — количество нейронов в сети.

- T — число, называемое *температурой* сети; чем выше температура, тем более случайным будет выход (т.е. тем ближе вероятность к 50%).
- σ — логистическая функция.

Нейроны в машинах Больцмана разделены на две группы: *видимые элементы* и *скрытые элементы* (рис. Д.2). Все нейроны работают тем же самым стохастическим способом, но видимые элементы являются теми, которые получают входы и из которых читаются выходы.

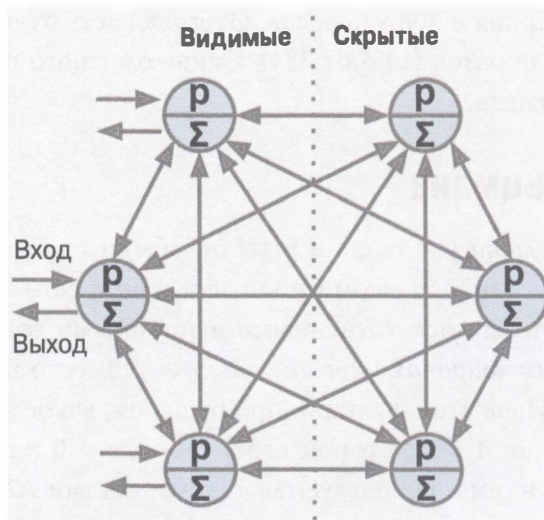


Рис. Д.2. Машина Больцмана

Из-за своей стохастической природы машина Больцмана никогда не стабилизируется в фиксированную конфигурацию, а вместо этого будет продолжать переключаться между множеством конфигураций. Если она работает достаточно долгое время, тогда вероятность наблюдения специфической конфигурации будет функцией только весов связей и членов смещения, а не первоначальной конфигурации (подобным же образом, после достаточно долгого тасования колоды карт конфигурация колоды не зависит от начального состояния). Когда сеть достигает этого состояния, в котором первоначальная конфигурация “забита”, то говорят, что она находится в *тепловом равновесии* (*thermal equilibrium*), хотя ее конфигурация продолжает все время изменяться. Устанавливая параметры сети надлежащим образом, позволяя сети достигнуть теплового равновесия и наблюдая ее состояние, мы можем моделировать широкий диапазон распределений вероятностей. Это называется *порождающей моделью*.

Обучение машины Больцмана означает поиск параметров, которые позволят сети приблизиться к распределению вероятностей обучающего набора. Например, если есть три видимых нейрона, а обучающий набор содержит 75% троек (0, 1, 1), 10% троек (0, 0, 1) и 15% троек (1, 1, 1), то после обучения машины Больцмана вы могли бы использовать ее для генерирования случайных двоичных троек с приблизительно таким же распределением вероятностей. Скажем, около 75% времени она выдавала бы тройку (0, 1, 1).

Такая порождающая модель может применяться различными способами. Например, если сеть обучается на изображениях, и вы предоставляете ей незавершенное или зашумленное изображение, тогда она автоматически “восстанавливает” его разумным образом. Вы также можете использовать порождающую модель для классификации. Просто добавьте несколько видимых нейронов для кодирования класса обучающего изображения (например, добавьте 10 видимых нейронов и включайте только пятый нейрон, когда обучающее изображение представляет пятерку). Затем при получении нового изображения сеть будет автоматически включать подходящие видимые нейроны, указывая класс этого изображения (скажем, она включит пятый видимый нейрон, если изображение представляет пятерку).

К сожалению, эффективных приемов обучения машин Больцмана не существует. Однако были разработаны довольно эффективные алгоритмы для обучения *ограниченных машин Больцмана* (*Restricted Boltzmann Machine* — *RBM*).

Ограниченные машины Больцмана

Ограниченная машина Больцмана — это просто машина Больцмана, в которой связи между видимыми элементами или между скрытыми элементами отсутствуют, а есть только связи между видимыми и скрытыми элементами. Например, на рис. Д.3 показана машина RBM с тремя видимыми и четырьмя скрытыми элементами.

В 2005 году Мигель Каррера-Перпиньян и Джеффри Хинтон предложили очень эффективный алгоритм обучения, названный *сопоставительным отклонением* (*contrastive divergence*)¹. Вот как он работает: для каждого обучающего образца x алгоритм начинает с того, что передает его сети, устанавливая состояние видимых элементов в x_1, x_2, \dots, x_n .

¹ “On Contrastive Divergence Learning” (“Об обучении методом сопоставительного отклонения”), М. Каррера-Перпиньян и Д. Хинтон (2005 год) (<http://goo.gl/ZCP6Ir>).

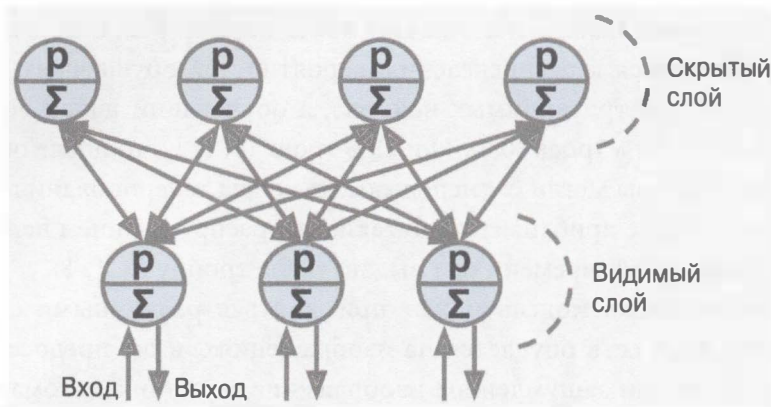


Рис. Д.3. Ограниченная машина Больцмана

Затем вычисляется состояние скрытых элементов путем применения статистического уравнения, описанного ранее (см. уравнение Д.1). Это дает скрытый вектор \mathbf{h} (где h_i — состояние i -того элемента). Далее вычисляется состояние видимых элементов с использованием того же самого статистического уравнения. Это дает вектор \mathbf{x}' . Потом еще раз вычисляется состояние скрытых элементов, что дает вектор \mathbf{h}' . Теперь можно обновить вес каждой связи, применив правило из уравнения Д.2, где η — скорость обучения.

Уравнение Д.2. Обновление весов методом сопоставительного отклонения

$$w_{i,j} \leftarrow w_{i,j} + \eta (\mathbf{x} \cdot \mathbf{h}^T - \mathbf{x}' \cdot \mathbf{h}'^T)$$

Огромное преимущество этого алгоритма заключается в том, что он не требует ожидания, пока сеть придет в тепловое равновесие: он просто проходит вперед, назад, снова вперед и все. Такая характеристика делает данный алгоритм несравненно более эффективным, чем предшествующие алгоритмы, и он был одной из главных составных частей первого успеха глубокого обучения, основанного на многослойных машинах RBM.

Глубокие сети доверия

Несколько слоев машин RBM могут быть уложены друг на друга; скрытые элементы машины RBM первого слоя служат видимыми элементами для машины RBM второго слоя и т.д. Такая стопка машин RBM называется *глубокой сетью доверия* (*Deep Belief Net* — DBN).

И-Вай Ти, один из студентов Джеффри Хинтона, опытным путем выяснил, что сети DBN можно обучать с помощью алгоритма сопоставительного отклонения по одному слою за раз, начиная с самых нижних слоев и постепенно перемещаясь к верхним слоям. Это вылилось в революционную статью, которая дала толчок началу цунами глубокого обучения в 2006 году².

Подобно машинам RBM сети DBN учатся воспроизводить распределение вероятностей своих входов без какого-либо надзора. Тем не менее, они намного лучше по той же самой причине, по которой глубокие нейронные сети являются более мощными, чем неглубокие: реальные данные часто систематизированы по иерархическим шаблонам и сети DBN извлекают из этого выгоду. Их нижние слои узнают низкоуровневые признаки во входных данных, тогда как верхние слои узнают высокоуровневые признаки.

Как и машины RBM, сети DBN по существу обучаются без учителя, но их также можно обучать с учителем, добавив несколько видимых элементов для представления меток. Кроме того, замечательное свойство сетей DBN состоит в том, что они способны обучаться в частичной манере. На рис. Д.4 представлена такая сеть DBN, сконфигурированная для частичного обучения.

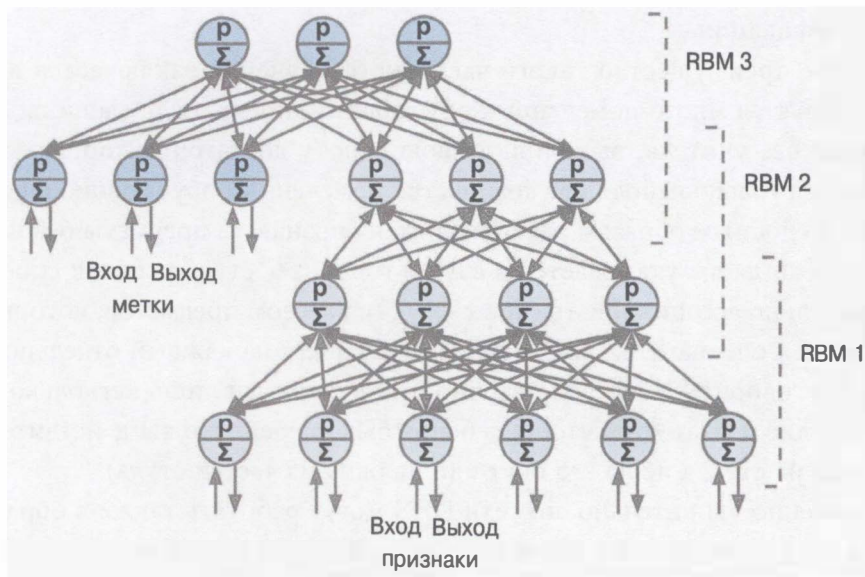


Рис. Д.4. Глубокая сеть доверия, сконфигурированная для частичного обучения

² “A Fast Learning Algorithm for Deep BeliefNets” (“Быстрый алгоритм обучения для глубоких сетей доверия”), Д. Хинтон, С. Осиндеро, И. Ти (2006 год) (<http://goo.gl/BcZQrH>).

Сначала машина RBM 1 обучается без учителя. Она узнает низкоуровневые признаки в обучающих данных. Затем машина RBM 2 обучается с использованием скрытых элементов машины RBM 1 в качестве входов, снова без учителя: она узнает высокоуровневые признаки (обратите внимание, что скрытые элементы машины RBM 2 включают только три крайних справа элемента, но не меточные элементы). Подобным образом можно было бы уложить большее количество машин RBM, но идея должна быть ясна. До сих пор обучение было полностью без учителя.

Наконец, машина RBM 3 обучается с применением в качестве входов скрытых элементов машины RBM 2, а также дополнительных видимых элементов, используемых для представления целевых меток (например, вектор в унитарном коде, который представляет класс образца). Она учится ассоциировать высокоуровневые признаки с обучающими метками. Это шаг обучения с учителем.

В конце обучения при передаче машине RBM 1 нового образца сигнал будет распространяться до машины RBM 2, далее до верхней части машины RBM 3 и затем обратно вниз до меточных элементов; надо надеяться, что высветится соответствующая метка. Именно так сеть DBN может применяться для классификации.

Большое преимущество такого частичного обучения заключается в том, что не требуется много помеченных обучающих данных. Если машины RBM, обученные без учителя, выполняют свою работу достаточно хорошо, тогда понадобится только небольшое количество помеченных обучающих образцов на класс. Подобным образом ребенок учится опознавать предметы без надзора, так что когда вы указываете на стул и говорите “стул”, ребенок способен самостоятельно ассоциировать слово “стул” с классом предметов, которые он уже научился опознавать. Вам не нужно указывать на каждый отдельно взятый стул и говорить “стул”; достаточно будет привести лишь несколько примеров (вполне достаточно, чтобы ребенок был уверен, что вы действительно ссылаетесь на стул, а не на его цвет или на одну из частей стула).

Совершенно удивительно, но сети DBN могут работать также в обратном направлении. Если вы активируете один из меточных элементов, то сигнал будет распространяться вверх до скрытых элементов машины RBM 3, затем вниз до машины RBM 2 и далее до машины RBM 1, а новый образец будет выдаваться видимыми элементами машины RBM 1. Этот новый образец обычно будет выглядеть похожим на обыкновенный образец класса,

чей меточный элемент вы активировали. Такая порождающая способность сетей DBN на самом деле обладает большой мощностью. Например, она использовалась с целью автоматической генерации надписей для изображений и наоборот: первая сеть DBN обучалась (без надзора) узнавать признаки в изображениях, а вторая сеть DBN обучалась (опять без надзора) узнавать признаки в наборах надписей (например, надписи “машина” часто сопутствует надписи “автомобиль”). После этого поверх обеих сетей DBN укладывалась машина RBM, которая обучалась с помощью набора изображений вместе с их надписями; она училась ассоциировать высокоуровневые признаки в изображениях с высокоуровневыми признаками в надписях. Затем в случае передачи сети DBN, работающей с изображениями, какого-то изображения автомобиля сигнал распространялся через сеть вверх до машины RBM верхнего уровня и обратно вниз до нижней части сети DBN, отвечающей за надписи, в итоге чего выдавалась надпись. Из-за стохастической природы машин RBM и сетей DBN надпись продолжала меняться случайным образом, но обычно она подходила к изображению. Если вы генерируете несколько сотен надписей, тогда наиболее часто генерируемые надписи, скорее всего, будут хорошим описанием изображения³.

Самоорганизующиеся карты

Самоорганизующиеся карты (*Self-Organizing Map* — *SOM*) совершенно отличаются от всех остальных типов нейронных сетей, которые обсуждались до сих пор. Они применяются при создании представления с низким числом измерений для набора данных, имеющего высокое число измерений, по большей части в целях визуализации, кластеризации или классификации. Нейроны распределяются по карте (как правило, двумерной для визуализации, но может существовать любое желаемое количество измерений), как демонстрируется на рис. Д.5. Каждый нейрон имеет взвешенную связь с каждым входом (на диаграмме показаны только два входа, но обычно их очень большое число, поскольку весь смысл карт SOM заключается в понижении размерности).

После того как сеть обучена, ей можно передавать новый образец и это активирует только один нейрон (т.е. одну точку на карте): нейрон, весовой вектор которого ближе всех к входному вектору.

³ Дополнительные детали и демонстрацию ищите в видеоролике Джеррри Хинтона: <http://goo.gl/7Z5QiS>.

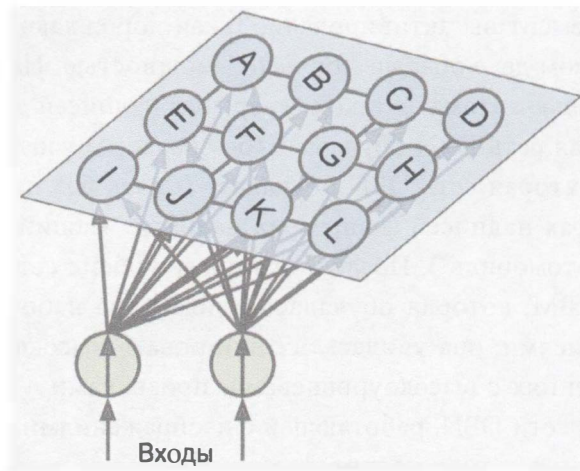


Рис. Д.5. Самоорганизующиеся карты

В общем случае образцы, находящиеся близко друг к другу в первоначальном входном пространстве, будут активировать нейроны, расположенные поблизости на карте. Такое свойство карт SOM делает их удобными для визуализации (в частности, можно легко идентифицировать кластеры на карте), но также и для приложений, подобных распознаванию речи. Например, если каждый образец представляет аудиозапись с произношением гласного звука человеком, тогда отличающиеся произношения гласного звука “а” будут активировать нейроны в одной и той же области карты, в то время как образцы с произношением гласного звука “е” будут активировать нейроны в другой области, а промежуточные звуки обычно приведут к активации промежуточных нейронов на карте.



Важное отличие от других приемов понижения размерности, которые обсуждались в главе 8, связано с тем, что все образцы отображаются на дискретное число точек в пространстве с низким количеством измерений (одна точка на нейрон). Когда нейронов совсем немного, такой прием лучше описывать как кластеризацию, а не понижение размерности.

Алгоритм обучения выполняется без учителя. Он работает, вынуждая все нейроны состязаться друг с другом. Сначала все веса инициализируются случайным образом. Затем случайно выбирается обучающий образец и передается сети. Все нейроны вычисляют расстояние между своим весовым вектором и входным вектором (что совершенно отличается от искусственных

нейронов, с которыми мы имели дело до сих пор). Нейрон с самым меньшим измеренным расстоянием выигрывает и его весовой вектор подстраивается, чтобы стать даже чуть ближе к входному вектору, что увеличивает вероятность выигрыша этим нейроном будущих состязаний для других входов, похожих на данный. Он также задействует соседние нейроны, которые также обновляют свой весовой вектор, чтобы слегка приблизится к входному вектору (но они не обновляют свои веса настолько же сильно, как выигравший нейрон). Далее алгоритм выбирает еще один обучающий образец и повторяет процесс, снова и снова. Как правило, алгоритм постепенно делает близлежащие нейроны специализированными на похожих входах⁴.

⁴ Вы можете представить себе класс маленьких детей с примерно одинаковыми навыками. Так случилось, что один ребенок показывает себя чуть лучше в баскетболе. Это мотивирует его больше тренироваться, особенно со своими друзьями. Через некоторое время группа друзей становится настолько хорошей в баскетболе, что другие дети не могут с ними соперничать. Но ситуация вполне нормальна, поскольку другие дети специализируются на других предметах. Со временем класс наполнится небольшими специализированными группами.

A

Accuracy, 29; 126
Action, 556
Activation function, 334
AdaBoost, 252
AdaGrad, 376; 380
Adam (adaptive moment estimation), 382
Adam optimization, 376
Adaptive learning rate, 381
Agent, 556
AlexNet, 470; 471
Anaconda, 72
Anomaly detection, 37
API
 Scikit-Learn, 97
 TF-slim, 295
Apriori, 35
Area Under the Curve (AUC), 136
Artificial Neural Network (ANN), 323
Artificial neuron, 326
Association rule learning, 35; 38
Associative memory network, 651
Autodiff, 304
Autoencoder, 34
Automatic differentiating (autodiff), 295
Average pooling layer, 467
Axon, 325

B

Backpropagation, 333
Backpropagation Through Time (BPTT), 497
Bagging, 240; 243
Batch GD (BGD), 154
Batch Gradient Descent, 164
Batch Normalization (BN), 360
Bellman optimality equation, 576
Between-graph replication, 441
Bias, 472
Bias neuron, 329
Bias term, 155
Binary classifier, 124

Biological Neural Networks (BNN), 326
Blender, 262
Boltzmann machine, 653
Boosting, 240; 251
Bootstrap aggregating, 244
Bootstrapping, 244
Bottleneck layer, 474

C

Classification And Regression Tree
 (CART), 228
Cloud machine learning, 403
Cluster, 415
Coding, 525
Comma-Separated Value (CSV), 75
Complementary slackness, 641
Computational complexity, 159
Confusion matrix, 127
Connectionism, 332
Continuous function, 162
Contractive Autoencoder (CAE), 551
Contrastive divergence, 655
Control dependency, 415
Convex function, 162
Convolutional layer, 455
Convolutional Neural Network (CNN), 453
Convolution kernel, 458
Cost function, 47
Credit assignment problem, 567
Cross entropy, 195
Cross-validation, 60
CUDA (Compute Unified Device
 Architecture), 403
CuDNN (CUDA Deep Neural Network), 403
Curse of dimensionality, 267

D

Data mining, 31
Data parallelism, 445
Datasets, 64
Data snooping bias, 81

- Decision boundary, 192
- Decision stump, 256
- Decision threshold, 131
- Decision Tree, 34; 108; 223
- Decoder, 490
- Deconvolutional layer, 481
- Deep autoencoder, 529
- Deep Belief Net (DBN), 39; 651; 656
- Deep Neural Network (DNN), 333
- Deep Q-learning, 583
- Deep Q-network (DQN), 556; 583
- Dendrite, 325
- Depth concat layer, 473
- Depth radius, 472
- Depthwise convolutional layer, 482
- Device, 297
- Dilation rate, 481
- Discount rate, 567
- Dropconnect, 393
- Dropout, 348; 362; 390
- Dropout rate, 390
- Duality, 640
- Dual problem, 216
- Dying ReLU, 356

E

- Early stopping, 185
- Eclat, 35
- Embedding, 101; 517
- Encoder, 490
- End-of-sequence (EOS), 497
- Energy function, 652
- Ensemble learning, 239
- Ensemble method, 239
- Episode, 563
- Epoch, 168
- Equality constraint, 640
- Estimator, 97
- Events file, 310
- Exclusive OR (XOR), 332
- Expectation maximization, 35
- Explained variance ratio, 278
- Exploding gradients, 351
- Exploration function, 582

- Exponential Linear Unit (ELU), 357
- Extremely randomized trees ensemble, 250

F

- False Negative (FN), 128
- False Positive Rate (FPR), 135
- Feature engineering, 54
- Feature extraction, 36; 54
- Feature map, 459
- Feature scaling, 103
- Feature selection, 54
- Feature space, 282
- Feedforward Neural Network (FNN), 336
- FIFO (first-in first-out), 423
- Filter, 458
- Fitness function, 47
- Fold, 109
- Follow The Regularized Leader (FTRL), 385
- Forget gate, 512
- Forward-mode autodiff, 305
- Frame, 486
- Function
 - activation, 334
 - softmax, 335

G

- Gate controller, 513
- Gated Recurrent Unit (GRU), 514
- Gaussian RBF kernel, 207
- Generalization error, 59
- Generalized lagrangian, 641
- Generative Adversarial Network (GAN), 551
- Generative model, 525
- Generative network, 527
- Genetic algorithm, 559
- GitHub, 296
- Google Remote Procedure Call (gRPC), 418
- GoogLeNet, 472; 474
- Gradient ascent, 560
- Gradient Boosted Regression Tree (GBRT), 257
- Gradient boosting, 252
- Gradient clipping, 365
- Gradient Descent (GD), 153

Gradient tree boosting, 256
Gradient vector, 164
Graphviz, 224
Grid search, 111

H

Hard margin classification, 200
Hard voting classifier, 240
Harmonic mean, 130
Heaviside step function, 329
Hebbian learning, 330
Hebb's rule, 330
Hessian, 384
Hidden layer, 333
Hierarchical Cluster Analysis (HCA), 35
Hinge loss, 221
Hopfield network, 651
Hypothesis, 69
Hypothesis boosting, 251

I

Impurity, 225
Inception module, 472
Incremental learning, 43
Incremental PCA (IPCA), 281
Inequality constraint, 641
Inference, 49
In-graph replication, 440
Input neuron, 329
Intercept term, 155
Internal covariate shift, 360
Inter-op thread pool, 412
Intra-op thread pool, 412
Isomap, 288

J

Jacobian, 384
Job, 415
Jupyter, 72; 74; 297; 298; 300

K

Karush-Kuhn-Tucker (KKT), 641
Keras, 295
Kernel, 218; 339; 411

Kernel PCA (kPCA), 35; 282
Kernel trick, 205
K-fold cross-validation, 108
K-means, 35
K-nearest neighbors, 34
Kullback–leibler divergence, 195

L

Label, 33
Lagrange function, 640
Lagrange multipliers method, 640
Lagrangian, 640
Landmark, 206
Large margin classification, 199
Lasso regression, 182
Latent loss, 547
Latent space, 547
Law of large numbers, 241
Leaky ReLU, 356
Learning curve, 154; 175
Learning rate, 43; 160
Learning schedule, 168
LeNet-5, 469
Levenshtein distance, 208
Liblinear, 209
Libsvm, 209
Linear Discriminant Analysis (LDA), 289
Linear regression, 34
Linear regression model, 153
Linear threshold unit (LTU), 328
Lipschitz continuous, 162
Locally-Linear Embedding (LLE), 35; 286
Local Response Normalization (LRN), 471
Logistic, 188
Logistic regression, 34; 154
Logit, 188
Logit regression, 187
Log loss, 179; 189
Long Short-Term Memory (LSTM), 511

M

Manifold, 272
Manifold assumption, 272
Manifold hypothesis, 272

- Manifold learning, 268; 272
- Mapping function, 217
- MapReduce, 68
- Markov Decision Process (MDP), 556; 574
- Master, 418
- Master service, 418
- Matplotlib, 72; 133; 136; 143
- Max margin learning, 376
- Max pooling layer, 466
- Mean Absolute Error (MAE), 70
- Mean Squared Error (MSE), 156
- Memory cell, 444; 489
- Mercer's theorem, 219
- Meta learner, 262
- Mini-batch, 170
- Mini-batch gradient descent, 154; 170
- Min-max scaling, 103
- MNIST (Mixed National Institute of Standards and Technology), 121
- Model parallelism, 443
- Model zoo, 373
- Momentum, 377
- Momentum optimization, 376
- Momentum optimizer, 306
- Momentum vector, 377
- Monte Carlo tree search, 574
- Multiclass classifier, 139
- Multidimensional Scaling (MDS), 288
- Multilabel classification, 147
- Multi-Layer Perceptron (MLP), 323; 332
- Multinomial classifier, 139
- Multinomial logistic regression, 193
- Multioutput classification, 148
- Multivariate regression, 67

N

- Name scope, 313
- Natural Language Processing (NLP), 453
- Negative class, 128
- Neocognitron, 455
- Nesterov Accelerated Gradient (NAG), 376; 379
- Neural network, 34

- No Free Lunch (NFL), 61
- Nonlinear Dimensionality Reduction (NLDR), 286
- Nonparametric model, 231
- Nonresponse bias, 53
- Normal equation, 156
- Normalization, 103
- Normalized exponential, 193
- NP-complete, 229
- Null hypothesis, 232
- Numerical differentiation, 305
- NumPy, 72; 96; 98; 148; 157; 281; 302
- Nvidia CUDA, 403
- Nvidia cuDNN, 403

O

- Objective function, 640
- Observation, 556
- Offline learning, 40
- Off-policy, 581
- One-hot encoding, 99
- One-versus-All (OvA), 139
- One-versus-One (OvO), 139
- OpenAI Gym, 560
- Optical Character Recognition (OCR), 27
- Optimal state value, 576
- Out-of-bag (oob), 247
- Out-of-core learning, 42
- Out-of-sample error, 59
- Output gate, 512
- Output layer, 333
- Overcomplete autoencoder, 540
- Overfitting, 55

P

- Pandas, 72; 76; 91; 96; 105
- Parametric leaky ReLU (PReLU), 357
- Parametric model, 231
- Partial derivative, 163
- Pasting, 244
- Peephole connection, 514
- Percentile, 79
- Perceptron, 328

- Perceptron convergence theorem, 331
- Placeholder node, 306
- Policy, 558
- Policy Gradient (PG), 556; 560
- Polynomial regression, 154; 172
- Polynomial regression model, 49
- Pooling kernel, 466
- Pooling layer, 455
- Positive class, 128
- Precision, 128
- Precision-Recall (PR) curve, 135
- Predictor, 33; 97
- Pre-image, 285
- Primal problem, 216
- Principal Component Analysis (PCA), 35; 235; 274
- Principal Component (PC), 275
- Producer, 437
- Projection, 268
- Protocol buffer, 418

Q

- Q-learning, 580
- Quadratic Programming (QP), 215
- Quantizing, 450
- Quartile, 79
- Queries per Second (QPS), 440
- Q-value, 577
- Q-value iteration, 578

R

- Radial Basis Function (RBF), 206
- Random forest, 34; 110; 239; 248
- Randomized leaky ReLU (RReLU), 357
- Randomized PCA, 282
- Randomized search, 114
- Random patches method, 248
- Random subspaces method, 248
- Recall, 129
- Receiver Operating Characteristic (ROC), 135
- Recognition network, 527
- Reconstruction, 527
- Reconstruction error, 280
- Rectified Linear Unit (ReLU), 314; 473

- Recurrent Neural Network (RNN), 485
- Recurrent neuron, 486
- Regression
 - Linear regression, 34
 - Logistic regression, 34
- Regularization, 56
- Regularization term, 179
- Reinforcement Learning (RL), 32; 555
- Residual error, 256
- Residual learning, 477
- Residual Network (ResNet), 476
- Residual unit, 477
- ResNet, 476; 478
- ResNet-34, 480
- Resource container, 421
- Restricted Boltzmann Machine (RBM), 34; 39; 374; 655
- Reverse-mode autodiff, 305
- Reward, 556
- RMSProp, 376; 382
- Root Mean Squared Error (RMSE), 68

S

- Sampling bias, 52
- Sampling noise, 52
- Scikit flow (skflow), 295
- Scikit-Learn, 48; 72; 96; 97; 99; 102; 104; 121; 125; 132; 137; 158; 171; 181; 196; 201; 209; 226; 229; 231; 242; 276; 277; 281; 288; 297; 336
 - средство перекрестной проверки
 - Scikit-Learn, 109
- SciPy, 98
- Self-Organizing Map (SOM), 659
- Semisupervised learning, 32
- Sensitivity, 129; 135
- Separable convolutional layer, 482
- Sequence, 485
- Session, 297
- Shortcut connection, 477
- Shrinkage, 259
- Signal, 325
- Sign function, 329

Simulated annealing, 168
 Singular Value Decomposition (SVD), 276
 Skip connection, 397; 477
 Slack variable, 214
 Soft margin classification, 201
 Softmax function, 193
 Softmax Regression, 154; 193
 Soft voting, 243
 Spare replica, 446; 450
 Sparsity, 543
 Sparsity loss, 543
 Specificity, 135
 Spurious pattern, 653
 Stacked autoencoder, 529
 Stacked denoising autoencoder, 541
 Stacking, 240; 261
 Stagewise Additive Modeling using a Multi-
 class Exponential (SAMME), 255
 Stale gradient, 447
 Standard correlation coefficient, 90
 Standard deviation, 78
 Standardization, 103
 State-action value, 577
 Stationary point, 640
 Statistical mode, 244
 StatLib, 64
 Step function, 328
 Stochastic average GD, 181
 Stochastic GD, 154
 Stochastic gradient boosting, 261
 Stochastic Gradient Descent (SGD), 124; 167
 Stochastic neuron, 653
 Strata, 84
 Stratified sampling, 84
 Stride, 457
 String kernel, 208
 Subgradient vector, 184
 Summary, 310
 Support vector, 200
 Support Vector Machine (SVM), 34; 199
 Swiss roll, 271
 Symbolic differentiation, 304; 305
 Synapse, 325

T

Tail heavy, 81
 Task, 415
 T-distributed Stochastic Neighbor
 Embedding (t-SNE), 35; 288
 Telodendria, 325
 Temporal Difference (TD), 579
 Tensor, 301
 TensorBoard, 295; 309; 312; 339
 TensorFlow, См. Библиотека, TensorFlow
 TensorFlow serving, 440
 Tensor Processing Unit (TPU), 403
 Test set, 59
 TFLearn, 295; 336
 TFLearn, 295
 TF-slim, 295
 Thermal equilibrium, 654
 Time series, 485
 Time step, 486
 Token, 497
 Token queue, 451
 Tolerance, 166
 Training data, 28
 Training instance, 28
 Training sample, 28
 Training set, 28; 59
 Transfer learning, 366
 Transformer, 97
 True Negative Rate (TNR), 135
 True Positive Rate (TPR), 129
 True Positive (TP), 128

U

Undercomplete, 528
 Underfitting, 57
 Univariate regression, 67
 Unsupervised pretraining, 374
 Upsample, 481
 Utility function, 47

V

Validation set, 60
 Value iteration, 577

Vanishing gradients, 351
Variance, 79
Variational autoencoder, 546
Voice recognition, 453

W

Wisdom of the crowd, 239
Worker service, 418

Z

Zero padding, 457
ZF Net, 472

A

Автокодировщик (autoencoder), 34, 525;
527; 626
вариационный, 545; 547
вероятностный, 546
глубокий, 529
линейный, 528
многослойный, 529; 530
предварительное обучение без учителя
с использованием многослойных
автокодировщиков, 538
сверточный, 551
шумоподавляющий, 541
повышающий, 540
понижающий, 628
порождающий, 546
разреженный, 543
реализация с помощью TensorFlow, 530
сжимающий (CAE), 551
шумоподавляющий, 540
реализация с помощью TensorFlow, 541
Агент, 39; 556
Агрегирование
бутстрэп, 244
прогнозов
с применением смешивающего
прогнозатора, 261
Аксон, 325
Алгоритм
AdaBoost, 254
AdaGrad, 380
Adam, 383

CART, 226; 228; 229
функция издержек для регрессии, 234
k ближайших соседей, 34; 49
LLE, 287
PCA, 275; 276; 277; 608
для сжатия, 279
обратная трансформация, 280
Q-обучения, 580
REINFORCE, 568
RL, 632
RMSProp, 382
SAMME, 255
вне политики, 581
генетический, 559
градиентного спуска, 160
динамического обучения, 42
динамического размещения, 408
для линейных методов SVM, 209
итерации
по Q-ценностям, 578
по ценностям, 577
линейной регрессии, 47
Apriori, 35
Eclat, 35
k-средние (k-means), 35
анализ главных компонент (PCA), 35
визуализации, 36
иерархический кластерный анализ
(HCA), 35
локальное линейное вложение (LLE), 35
максимизация ожиданий (expectation
maximization), 35
обнаружение аномалий (anomaly
detection), 37
обучение ассоциативным правилам
(association rule learning), 38
ограниченные машины Больцмана
(restricted Boltzmann machine), 34
понижение размерности, 35; 36
стохастическое вложение соседей
с t-распределением (t-SNE), 35
ядерный анализ главных компонент
(kernel PCA), 35
моментной оптимизации, 377
обновления весов методом сопостави-
тельного отклонения, 656

- обучения
 - TD, 580
 - без учителя, 35
 - методом временных разностей, 579
 - на основе моделей, 598
 - на основе образцов, 49
 - с обратным распространением, 333
 - с учителем, 34
 - к ближайших соседей, 34; 49
 - дерево принятия решений, 34
 - линейной регрессии, 34
 - логистической регрессии, 34
 - нейронных сетей, 34
 - случайного леса, 34
- пакетной нормализации, 360
- подготовка данных для алгоритмов
 - машинного обучения, 94
- понижения размерности, 37
- случайного леса, 34; 249
- сравнение алгоритмов для линейной
 - регрессии, 171
- стохастический, 236
- ускоренного градиента Нестерова, 379
- Анализ
 - главных компонент (PCA), 274; 281
 - инкрементный, 281
 - рандомизированный, 282
 - ядерный (kPCA), 282
 - иерархический кластерный (HCA), 35
 - линейный
 - дискриминантный (LDA), 289
 - смысловой, 485
- Ансамбль, 115; 239
- GBRT, 259
- Архитектура
 - AlexNet, 470; 471
 - BNN, 326
 - CUDA, 403
 - GoogLeNet, 472; 474; 475
 - LeNet-5, 455; 469
 - ResNet, 478
 - ZF Net, 472
 - сверточных нейронных сетей, 467
- Атрибут, 34
 - гистограммы для числовых атрибутов, 79
 - категориальный, 100; 101; 106
 - комбинации атрибутов, 93

- обработка текстовых и категориальных
 - атрибутов, 98
- сводка по числовым атрибутам, 78
- скомбинированный, 102
- числовой, 104; 106

Б

- Библиотека
 - Caffe, 296
 - Deeplearning4j, 296
 - H2O, 296
 - liblinear, 209
 - libsvm, 209
 - MXNet, 296
 - Nvidia CUDA, 403
 - Nvidia cuDNN, 403
 - Scikit-Learn, 121; 132; 141; 201; 209; 226; 228; 229; 245; 250; 285; 288; 336; 343
 - TensorFlow, 293; 296; 300; 301; 306; 315; 336; 338; 362; 373; 396; 401; 403; 462; 481; 610; 618; 643
 - использование CUDA и cuDNN, 404
 - кластер TensorFlow, 415
 - спецификация кластера, 415
 - операции свертки, 481
 - очередь TensorFlow, 423
 - размещение операций на устройствах, 408
 - считыватели для различных файловых
 - форматов, 430
 - установка, 297; 403
 - функции, 410; 437
- TFLearn, 295
- Theano, 296
- Torch, 296
 - с использованием TensorBoard, 309
- Блок (fold), 109
- Бустинг (boosting), 240; 251
 - адаптивный, 252
 - градиентный, 252; 256; 258
 - на основе деревьев, 256
 - стохастический, 261
- Бутстрэп-агрегирование, 244
- Бутстрэппинг, 244
- Буфер
 - протокольный, 418
- Бэггинг (bagging), 240; 243; 245

В

Вектор

- градиент, 164
- лассо-регрессии, 184
- опорный, 34; 200; 602
- субградиент, 184

Векторное представление, 517

Вероятность

- оценивание вероятностей, 227
- сохранения, 392

Вес

- обновление весов методом сопоставительного отклонения, 656
- связывание весов, 532

Визуализация, 35; 36

- алгоритмы визуализации, 36
- географических данных, 87
- улучшенная, 88
- графа, 309
- инструмент TensorBoard, 295
- признаков, 537
- пример визуализации t-SNE, 37
- реконструкций, 536

Вложения (embedding), 101

Временной ряд, 485; 501

Время

- обратное распространение во времени (BPTT), 497; 510
- обучение для прогнозирования временных рядов, 500

Вставка (или вклеивание), 244

- в Scikit-Learn, 245

Выборка

- смещение выборки (sampling bias), 52
- стратифицированная, 84
- шум выборки (sampling noise), 52

Выведение (inference), 49

Вычисления

- параллельные, 294

Вычислительная сложность, 159

Г

Гауссова функция RBF, 206

Гауссово ядро RBF, 207

Гессиан, 384

Гиперпараметр, 57; 98; 472; 597

С, 207

- coef0, 205
- criterion, 230
- gamma, 207
- penalty, 182
- допуска, 209
- подстройка гиперпараметров, 283
- регуляризации, 207; 230; 231
- скорости обучения (learning rate), 160

Гиперплоскость, 212

Гипотеза, 69

- нулевая (null hypothesis), 232

Гистограмма

- для медианного дохода, 85
- для числовых атрибутов, 79
- медленно убывающие хвосты, 81

Голосование

- мягкое, 243

Градиент

- взрывной рост градиентов, 351; 352
 - градиентный подъем, 560
 - градиентный спуск, 153, 160; 302; 599
 - мини-пакетный, 154; 170; 307
 - пакетный (BGD), 154; 164
 - стохастический (SGD), 154; 167; 181; 600
 - усредненный, 181
 - исчезновение градиентов, 351; 352
- ### Нестерова
- ускоренный (NAG), 376; 378
 - отсечение градиентов, 365
 - политики, 556; 568
 - расчет градиентов вручную, 303
 - устаревший, 447

Граница решений, 212; 192

- в дереве принятия решений, 226

Граф

TensorFlow

- параллельное выполнение, 413
- визуализация
- с использованием TensorBoard, 309
- для обучения многослойного автокодировщика, 534
- для чтения обучающих образцов, 433
- загрузка данных напрямую из графа, 429
- простой, 293

репликация
внутри графа, 439; 440
между графами, 439; 440; 441
управление графами, 299
График
мощности, 386
обучения, 168; 386
производительности, 386
рассеяния географических данных, 88
экспоненциальный, 386
Гребневая регрессия, 179; 180
решение в аналитическом виде, 181

Д

Данные
асинхронная загрузка, 424
асинхронные обновления, 446; 447
визуализация, 87
географических данных, 87
временных рядов, 485
глубинный анализ данных, 31
график рассеяния географических данных, 88
дополнение данных, 395
загрузка данных, 75
из графа
напрямую, 429
предварительная, 430
зашумление данных, 45
извлечение данных из очереди, 425
набор данных MNIST, 121
необоснованная эффективность данных, 50
нерепрезентативные, 52
обнаружение, 87
обучающие, 28
открытые хранилища данных, 63
Datasets, 64
StatLib, 64
метапорталы, 64
очистка данных, 95
параллелизм данных, 445–447; 450
переобучение обучающих данных, 55
подготовка данных для алгоритмов
машинного обучения, 94
получение данных, 71
помещение данных в очередь, 424

протокольный буфер, 417
синхронные обновления, 446
чтение
из графа, 430
из множества файлов, 435
эффективные представления данных, 526
Двойственность, 640
Действия, 556
Декодировщик, См. Сеть,
порождающая, 490; 527
Дендрит, 325
Дерево
двоичное, 226
принятия решений, 34; 108; 223; 232; 234;
604
визуализация, 223
границы решений, 226
двоичное, 604
для регрессии, 233
листовой узел, 225
прогнозы регрессионных моделей в виде
деревьев принятия решений, 234
Дисконтная ставка, 567
Дискретизация (upsample), 481
Дисперсия, 79; 178
как функция количества измерений, 279
коэффициент объясненной дисперсии, 278
предохранение дисперсии, 274
Дифференцирование
автоматическое, 295; 643
в обратном режиме, 305; 648
в прямом режиме, 305; 647
ручное, 643
символическое, 304; 305; 644
численное, 305; 645
Дополнение нулями, 457
Допуск (tolerance), 166
Допущения
проверка допущений, 71

З

Зависимости управления, 415
Загрузка данных, 75
Загрязненность (impurity), 225
Джини, 230; 604
мера загрязненности, 226; 230

Задание, 415

Задача, 415

NP-полная, 229

двойственная, 216

квадратичного программирования (QP), 215

классификации, 33

на основе исключяющего “ИЛИ”, 332

многомерной регрессии, 67

обнаружения аномалий, 37

обучения ассоциативным правилам, 38

одномерной регрессии, 67

понижения размерности, 36

прикрепление операций между

задачами, 419

прогноирования целевого числового

значения, 33

прямая, 216

регрессии, 33

квадратный корень из среднеквадратической ошибки (RMSE), 68

условной оптимизации, 214

Зазор

нарушения зазора, 214

Закон

больших чисел, 241

Запись

двоичная фиксированной длины, 430

Запрос

количество в секунду (QPS), 440

Запуск системы, 117

Зоопарки моделей, 373

И

Изображение

дискретизация изображения, 481

Изометрическое отображение, 288

Имитация отжига, 168

Инициализация

Глоро, 354

Ксавье, 353; 354

Хе, 355

Инспектирование, 98

Инструмент

TensorBoard, 295; 309

virtualenv, 73

Интерфейс

API

Keras, 295

TF.Learn, 295; 336

TF-slim, 295

Информация

прирост информации, См. Энтропия, 230

теория информации Шеннона, 230

Исключающее “ИЛИ”, 332

Исключение

OutOfRangeError, 433

Испытательный набор, 59; 116; 598

создание, 81

Итерации по ценностям, 577

К

Канал

цветовой, 460

Карта

признаков, 459

наложение множества карт признаков, 459

сверточные слои с множеством карт

признаков, 460

самоорганизующаяся (SOM), 659

Квадратный корень из среднеквадратической ошибки (RMSE), 68

Квантование нейронной сети, 450

Квартиль, 79

Кеширование замороженных слоев, 372

Класс

AdaBoostClassifier, 256

AdaBoostRegressor, 256

AdagradOptimizer, 381

AdamOptimizer, 383

BaggingClassifier, 245; 246; 248; 249

BaseEstimator, 102

CategoricalEncoder, 100

ColumnTransformer, 105

Coordinator, 434; 437

DataFrameMapper, 105

DataFrameSelector, 105

DecisionTreeClassifier, 231; 249

DecisionTreeRegressor, 232

DNNClassifier, 336; 337

ElasticNet, 185

ExtraTreesClassifier, 250

- ExtraTreesRegressor, 250
- FeatureUnion, 106
- FTRLOptimizer, 385
- GradientBoostingRegressor, 257; 260
- GridSearchCV, 111; 114; 237; 283
- IncrementalPCA, 281
- KernelPCA, 282
- Lasso, 184
- LinearSVC, 202; 203; 208; 209; 211
- LinearSVR, 210
- LocallyLinearEmbedding, 286
- OneVsOneClassifier, 141
- OneVsRestClassifier, 141
- PCA, 277
- Pipeline, 104
- PolynomialFeatures, 173; 174
- QueueRunner, 434; 437
- RandomForestClassifier, 137; 249; 250
- RandomForestRegressor, 112; 250; 257
- RandomizedSearchCV, 114
- ReLU, 317
- Ridge, 181
- SGDClassifier, 125; 131; 132; 146; 202; 209
- SGDRegressor, 169
- StandardScaler, 163; 200; 203; 303; 337
- StratifiedKFold, 126
- StratifiedShuffleSplit, 85
- SVC, 202; 205; 207; 208; 209; 243
- SVR, 211
 - линейно сепарабельный, 199
 - отрицательный, 128
 - спрогнозированный, 128
 - фактический, 128
- Классификатор
 - BaggingClassifier, 247
 - SVM, 205; 602
 - двойственная форма, 217
 - использующий ядро RBF, 208
 - линейный, 203; 204
 - прогноз, 212
 - с жестким зазором, 214
 - с мягким зазором, 215
 - с полиномиальным ядром, 205
 - двоичный, 124
 - многоклассовый, 139
 - полиномиальный, 139
 - последовательностей, 498
 - с голосованием, 240
- Классификация, 33; 121
- SVM, 209
 - линейная, 212
 - нелинейная, 203
 - с жестким зазором, 200
 - с мягким зазором, 200; 201
 - с широким зазором, 199; 200
- истинно отрицательная, 128
- истинно положительная, 129
- ложноотрицательная, 128
- многовыходовая, 148
- многозначная, 146
- многоклассовая, 139
- спама, 33
- Кластер, 415
 - TensorFlow, 416; 438
 - распараллеливание нейронных сетей, 438
 - спецификация кластера, 416
- Кластеризация, 35; 36
 - иерархическая, 36
- Ковариационный сдвиг, 360
- Кодирование
 - с одним активным состоянием, 99
 - унитарное, 99
- Кодировка, 525
- Кодировщик, См. Сеть, распознающая, 490
- Коллективный разум, 239
- Композиция, 98
- Компонент
 - главный (РС), 275
 - инкрементный анализ (IPCA), 281
 - рандомизированный анализ, 282
- Конвейер, 66
 - трансформации, 104
- Коннекционизм, 332
- Конструирование признаков, 54
- Контейнер
 - ресурсов, 421; 423
- Контроллер
 - шлюзов, 513
- Кора головного мозга
 - локальное рецепторное поле, 454
 - строение, 454

Корреляция
коэффициент
Пирсона, 90
стандартный, 90
Кортеж
очередь кортежей, 426
Коэффициент
доверия, 567
корреляции Пирсона, 90
объясненной дисперсии, 278
разветвления по входу (fan-in), 354
разветвления по выходу (fan-out), 354
расширения, 481
Кривая
ROC, 135; 136
площадь под кривой ROC, 136
обучения, 154; 174
для линейной модели, 176
для полиномиальной модели, 177
точности-полноты (PR), 135
Критерии качества работы, 68

Л

Лагранжиан, 640
обобщенный, 641
Лассо-регрессия, 182
вектор-субградиент лассо-регрессии, 184
Латентная потеря, 547
Лес
особо случайных деревьев, 250
случайный, 34; 110; 239; 248; 605
алгоритм случайного леса, 249
Линейный дискриминантный анализ
(LDA), 289
Линейный элемент
выпрямленный, 314
Логарифмическая потеря, 179; 189
Логистика, 188
Логистическая регрессия, 34; 187
Логит (logit), 188
Локальное линейное вложение (LLE), 286

М

Маркер конца последовательности
(EOS), 497

Марковские процессы принятия
решений (MDP), 556; 574
Массив
NumPy, 96; 98; 100
Мастер, 418
Масштабирование
по минимуму, 103
признаков, 103
Матрица
весов, 287
главных компонентов, 276
неточностей, 127; 129
разреженная SciPy, 98; 100
рассеяния, 92
Машина
Большмана, 34; 374; 653; 654
ограниченная (RBM), 39; 655
Машинное обучение (МО), 27; 28; 63;
153; 384; 439; 533; 538; 555; 556;
579; 582; 583; 596
Q-обучение, 579; 580; 582
автономное, 40
ансамблевое, 110; 239; 605
ассоциативным правилам, 35; 38
без учителя, 33; 35; 374
внешнее, 42
глубокое, 583
график обучения, 386
динамическое, 41; 42; 597
для оптимизации наград, 556
избегание переобучения посредством
регуляризации, 388
методом временных разностей, 579
модели, 47
на основе данных, 28
на основе многообразий, 272
на основе моделей, 44; 45
на основе образцов, 43; 44; 597
нейронных сетей, 351; 439; 616
основные проблемы машинного
обучения, 50
данные плохого качества, 54
недообучение обучающих данных, 57
недостаточный размер обучающих
данных, 50

- нерепрезентативные обучающие данные, 52
- чрезмерное обобщение, 55
- остаточное, 477
- пакетное, 40
- передачей знаний, 366
- планирование скорости обучения, 385
- по одному автокодировщику за раз, 533
- по Хеббу, 330
- полный проект машинного обучения, 63
- постепенное, 43
- предварительное, 374; 538
- приближенное, 582
- разреженных моделей, 384
- с максимальным зазором, 376
- с подкреплением, 32; 39; 40; 555; 556; 629
- с учителем, 33; 34
- скорость обучения, 43
- типы систем машинного обучения, 32
- частичное, 32; 38
- Мера**
 - F1, 130
 - загрязненности Джини, 230
- Метапорталы, 64**
- Мета-ученик, См. Смеситель, 262**
- Метки, 33**
- Метод**
 - apply_gradients(), 570
 - compute_gradients(), 365 ; 570
 - corr(), 90
 - decision_function(), 132; 137; 140
 - describe(), 78
 - dot(), 157
 - dropna(), 95
 - export_graphviz(), 224
 - factorize(), 99
 - fit(), 96; 102; 104; 106; 187; 260; 281
 - fit_transform(), 97; 100; 102; 104
 - get_operation_by_name(), 367
 - get_operations(), 368
 - get_params(), 102
 - get_tensor_by_name(), 367
 - head(), 77
 - hist(), 79; 80
 - info(), 77
 - inverse_transform(), 280
 - join(), 417; 435
 - k ближайших соседей, 49
 - minimize(), 387; 536
 - next_batch(), 344
 - partial_fit(), 281
 - predict(), 98; 132; 134; 192
 - predict_proba(), 137; 141; 243; 245; 256
 - render(), 561
 - reset(), 561
 - restore(), 308
 - save(), 308
 - score(), 98
 - set_params(), 102
 - staged_predict(), 259
 - step(), 562
 - toarray(), 100
 - transform(), 97; 102; 105; 106
 - value_counts(), 78
 - ансамблевый, 115; 242
 - множителей Лагранжа, 640
 - опорных векторов (SVM), 34; 199; 200; 602
 - динамический, 220
 - классификация SVM, 199
 - параметрически редуцированный, 217
 - разложения матрицы Андре-Луи Холецкого, 181
 - регуляризации полиномиальной модели, 179
 - случайных подпространств, 248
 - случайных участков, 248
 - сопоставительного отклонения, 656
 - стохастического градиентного спуска (SGD), 124
- Метрика**
 - мера F1, 130
 - полнота (recall), 129
 - точность (precision), 128
- Мини-пакет, 41; 170**
- Многомерное шкалирование (MDS), 288**
- Многообразие (manifold), 272**
- Множитель**
 - Каруша-Куна-Таккера (ККТ), 641
 - Лагранжа, 640
- Мода**
 - статистическая, 244
- Моделирование**
 - локальных связей
 - линейное, 287

Модель

SVM

параметрически редуцированная, 210

TensorFlow

повторное использование, 366

анализ моделей, 115

белого ящика, 227

восстановление, 308

зоопарки моделей, 373

кривые обучения

для линейной модели, 176

для полиномиальной модели, 177

линейная, 47

прогон с использованием Scikit-Learn, 48

простая, 46

регрессионная, 153; 155

прогнозы, 158; 234

функция издержек MSE, 156

регуляризированная, 179

машинного перевода

простая, 519

непараметрическая, 231

обучение моделей, 47; 106; 107; 153

параллелизм модели, 443

параметрическая, 231

повторное использование, 370

полиномиальная, 49

регрессионная, 49

порождающая, 525; 628; 654

разреженная, 183

сохранение, 308

точная настройка, 111

черного ящика, 227

член

свободный (intercept term), 155

смещения (bias term), 155

Модуль

Pandas, 91

pip, 72

Scikit-Learn, 84; 96; 99

начала, 472; 473

Модульность, 314

Момент, 377

вектора, 377

Н

Наблюдение, 556

Набор данных

MNIST, 121; 123

Swiss roll, 271

неразвернутый, 287

понижение размерности до двух измерений, 289

ассиметричный, 127

двумерный, 271

испытательный, 59; 82; 116; 598

линейно сепарабельный, 203

обучающий, 59

помеченный, 596

проецирование до d измерений, 277

понижение размерности, 289

проверочный, 60; 598

сжатие набора данных MNIST, 280

трехмерный, 270

Награда, 556

с учетом дисконтной ставки, 568

Недообучение (underfitting) обучающих данных, 57

Нейрон

биологический, 325

входной, 329

искусственный, 326

количество на скрытый слой, 347

развернутый во времени, 486

рекуррентный, 486

смещения, 329

стохастический, 653

Нейронные сети, 34; 323

архитектура

без учителя, 34

биологические (BNN)

архитектура, 326

глубокие, 478

обыкновенные, 478

искусственные (ANN), 323; 613

использование нейронной сети, 344

квантование нейронной сети, 450

обучение, 351

рекуррентные (RNN), 485; 624

базовые, 491

глубокие, 506

креативные, 505

обучение, 497

сверточные (CNN), 453; 621
архитектура, 467
точная настройка гиперпараметров, 345
частичные, 34
Неоднородность Джини, 230
Неокогнитрон, 455
Неустойчивость, 235
Норма, 70
евклидова, 70
Манхэттена, 70
Нормализация, 103
локальная ответа, 472
локальная ответа (LRN), 471
пакетная, 359

О

Обнаружение аномалий, 37
Образ
ложный, 653
Образец
неиспользуемый (oob), 247
Обратная связь, 333
Обратное распространение, 615
Обучающие данные, 28
Обучающий набор, 28; 59
для обучения с учителем, 33
помеченный, 596
проецирование до d измерений, 277
чувствительность к деталям обучающего набора, 236
чувствительность к поворотам обучающего набора, 235
Обучающий образец, 28
Обучение, См. Машинное обучение
Объект
Coordinator, 434
DataFrame, 76; 84; 95; 96
FileWriter, 310
Pandas, 96
DataFrame, 76; 96; 105
Python, 78
QueueRunner, 437
RandomShuffleQueue, 438
RunOptions, 442
Saver, 308; 309
TextLineReader, 431

Ограничение
активное, 641
в виде неравенства, 641
в виде равенства, 640
Оперативная память
графического процессора
управление, 406
Операция
dequeue, 425; 426
dequeue_many, 426
dequeue_up_to, 427
read, 432
TensorFlow, 411
источника, 301
прикрепление операций между задачами, 418
размещение операций на устройствах, 408
мягкое, 412
простое, 409
регистрация размещений, 409
функция динамического размещения, 410
с дуальными числами, 647
считывателя, 430
Опорный вектор, 34; 200; 602
Оптимизатор, 306
AdaGrad, 376
RMSProp, 376
SyncReplicasOptimizer, 451
моментный, 306
старший (chief), 451
Оптимизация
Adam, 376
моментная, 376; 377
условная, 214
Оптическое распознавание знаков
(OCR), 27
Ориентир (landmark), 206
Отклонение
сопоставительное, 655
среднее абсолютное, 70
стандартное, 78
Отключение (dropout), 348; 362; 390
доля отключения, 390
Отображение
изометрическое (Isomap), 288
Оценщик (estimator), 97

Очередь
FIFO (first-in first-out), 423
PaddingFIFOQueue, 428
RandomShuffleQueue, 427; 432
TensorFlow, 423; 434
заккрытие очереди, 427
извлечение данных из очереди, 424
кортежей, 426
маркеров, 451
помещение данных в очередь, 424

Очистка данных, 95

Ошибка
анализ ошибок, 142
восстановления, 280
выхода за пределы выборки, 59
обобщения, 59
остаточная, 256
прообраза восстановления, 284
среднеквадратическая (MSE), 156
средняя абсолютная (MAE), 70

П

Пакет
graphviz, 224
мини-пакет, 41; 170

Пакетная нормализация, 359
алгоритм пакетной нормализации, 360
с помощью TensorFlow, 362

Память
требования к памяти, 464
ячейка памяти, 444; 489

Параллелизм данных, 445; 450
с асинхронными обновлениями, 446; 447
с синхронными обновлениями, 446

Параллелизм модели, 443

Параллельные вычисления, 294

Параметр
гиперпараметр, 98; 472; 597
подстройка, 283
гиперпараметр coef0 , 205
гиперпараметр gamma , 207
гиперпараметр допуска, 209
гиперпараметр регуляризации, 207
гиперпараметру C , 207
пространство параметров модели, 163
скорости обучения (learning rate), 160

эффективность параметров, 346

Перекрестная проверка, 60

Scikit-Learn, 108

Переменная

совместное использование, 316

фиктивная, 214

фрагментация, 419

Переобучение (overfitting) обучающих
данных, 55

Персептрон, 328; 332

диаграмма персептрона, 330

классический, 613

многослойный, 323; 332; 333

задача классификации XOR, 332

обучение с помощью высокоуровневого

API-интерфейса TensorFlow, 336

правило обучения персептрона, 330

ступенчатая функция, применяемая
в персептронах, 329

теорема о сходимости персептрона, 331

Погрешность

вызванная неполучением ответов, 53

неустраняемая, 178

Подстройка гиперпараметров, 283

Поиск

по дереву методом Монте-Карло, 574

рандомизированный, 114

решетчатый, 111

связей, 90

Полиномиальное отображение второй
степени, 217

Политика, 39; 558

ϵ -жадная, 581

в форме нейронных сетей, 564

градиенты политики, 560; 568

исследования, 581

стохастическая, 558

Полнота (recall), 129

Полоса пропускания

насыщение, 448

Понижение размерности, 36; 267; 607

нелинейное, 286

приемы понижения размерности, 288

с одновременным предохранением свя-
зей, 288

- Порог принятия решения, 131
 - Последовательности, 485
 - Потеря
 - логарифмическая, 189
 - Поток
 - пулы потоков для распараллеливания операций, 412
 - внутриоперационный, 412
 - межоперационный, 412
 - Правило
 - обучения персептрона, 330
 - Хебба, 330; 652
 - цепное, 649
 - Правильность (accuracy), 126
 - Прием LLE, 286
 - Признак, 34
 - выбор признаков, 54
 - выделение признаков, 36; 54
 - конструирование признаков, 54
 - масштабирование признаков, 103
 - разреженный (sparse), 209
 - создание новых признаков, 54
 - Проверка
 - допущений, 71
 - перекрестная, 60; 108; 125; 598
 - Прогноз, 44; 212
 - AdaBoost, 255
 - агрегирование прогнозов с применением смешивающего прогнозатора, 261
 - выработка прогнозов, 225
 - с помощью параметрически редуцированного метода SVM, 220
 - классификатора с жестким голосованием, 241
 - линейного классификатора SVM, 212
 - линейной регрессионной модели, 158
 - регрессионной модели в виде деревьев принятия решений, 234
 - Прогнозатор (predictor), 33; 97
 - Прогнозирование целевого числового значения, 33
 - Программа
 - TensorFlow, 299
 - оптического распознавания знаков, 27
 - фильтр спама, 27
 - Программирование
 - квадратичное (QP), 215
 - подход с машинным обучением, 30
 - традиционное, 29
 - Проекция, 270
 - Проецирование до d измерений, 277
 - Производительность
 - показатели производительности, 125
 - Производная
 - функции, 644; 646
 - частная, 163
 - Проклятие размерности, 267; 268
 - Прообраз (pre-image), 285
 - Пространство
 - имен, 313
 - латентное, 547
 - параметров модели, 163
 - признаков, 282
 - Протокол
 - gRPC, 418
 - Процентиль (percentile), 79
 - Процессор
 - графический
 - управление оперативной памятью, 406
 - Пул потока
 - внутриоперационный, 412
 - для распараллеливания операций, 412
 - межоперационный, 412
- ## Р
- Рабочая характеристика приемника (ROC), 135
 - Радиус глубины, 472
 - Развязывание (dropconnect), 393
 - Размерность
 - понижение размерности, 267; 607
 - нелинейное (NLDR), 286
 - приемы понижения размерности, 288
 - с одновременным предохранением связей, 288
 - проклятие размерности, 267; 268; 608
 - Разреженность, 543
 - Разум
 - коллективный, 239

Раннее прекращение, 185
 Распознавание речи, 453
 Распределение
 нормальное (гауссово), 79; 339
 Распространение
 обратное, 615
 Расстояние (расхождение)
 Кульбака–Лейблера, 195; 544
 Регрессия, 33; 232
 SVM, 210
 линейная, 210
 применяющая полиномиальное ядро 2-го
 порядка, 211
 с узким зазором, 210
 с широким зазором, 210
 гребневая, 179; 180; 601
 решение в аналитическом виде, 181
 дерево принятия решений для регрес-
 сии, 233
 к среднему, 33
 лассо-, 182; 601
 вектор-субградиент лассо-регрессии, 184
 линейная, 34
 с помощью TensorFlow, 301
 сравнение алгоритмов для линейной
 регрессии, 171
 логистическая, 34; 154; 187
 глубокая, 34
 многопеременная, 154; 193
 полиномиальная, 193
 многомерная, 67
 одномерная, 67
 полиномиальная, 154; 172
 высокой степени, 174
 Регуляризация, 56
 на основе отключения, 391
 с ранним прекращением, 186
 Тихонова, См. Регрессия, гребневая, 179
 член регуляризации, 179
 Реконструкция, 527
 Рекуррентные нейроны, 486
 Реплика
 запасная, 446; 451
 Репликация
 внутри графа, 440
 между графами, 440; 441

Ресурсы
 контейнеры ресурсов, 421; 423

С

Свертка, 456
 сверточное ядро, 458
 Сводка (summary), 310
 Связь
 обходящая, 397; 477; 479
 смотровая, 514
 сокращенная, 477
 поиск связей, 90
 Сеанс, 297
 локальный, 421
 открытие сеанса, 417
 распределенный, 421
 Сервер
 Jupyter, 74
 TensorBoard, 311
 TensorFlow
 запуск, 416
 Сеть
 ANN, 323; 327
 CNN, 467; 453; 621
 DBN, 39; 651; 656
 DNN, 333
 стандартная конфигурация, 397
 DQN, 556; 583–585; 586
 глубокая, 556; 583; 586
 динамическая, 583
 целевая, 584
 FNN, 336
 ResNet-34, 476; 480
 RNN, 485; 497; 505; 506; 624
 архитектура биологических нейронных
 сетей, 326
 доверия
 глубокая (DBN), 39; 651; 656
 “кодировщик-декодировщик”, 519
 нейронная, 34; 323
 глубокая (DNN), 333; 478
 CUDA, 403
 обыкновенная, 478
 обучение с использованием TensorFlow, 338
 остаточная, 478
 рекуррентная, 445

- искусственная (ANN), 323; 613
- использование, 344
- обучение, 439; 351
- полносвязная
 - расщепление, 443
- прямого распространения (FNN), 336
- рекуррентная (RNN), 485; 624
 - глубокая, 506
 - креативная, 505
 - обучение, 497
- сверточная (CNN), 453; 621
 - архитектура, 467
- точная настройка гиперпараметров, 345
- частичная, 34
- квантование нейронной сети, 450
- остаточная (ResNet), 476; 480
- порождающая, 527
 - состязательная, 551
- развертывание сети во времени, 486
- распознающая, 527
- с ассоциативной памятью, 651
- температура сети, 654
- топология сети, 345
- Хопфилда, 651; 652
- целевая DQN, 584
- эластичная, 184
- Сжатие (shrinkage), 259
- Сигнал, 66; 325
- Синапс (synapse), 325
- Сингулярное разложение (SVD), 276
- Система
 - TensorFlow Serving, 439
- машинного обучения, 32
 - без учителя, 33
 - с подкреплением, 32
 - с учителем, 33
 - алгоритмы обучения с учителем, 34
 - частичное обучение, 32
- Скорость обучения, 43; 160
- адаптивная, 381
- Скорость сходимости, 166
- Слой
 - входной, 333
 - выходной, 333
 - дополнение нулями, 457
 - замороженный, 371
 - кеширование, 372
- конкатенации в глубину, 473
- обращения свертки, 481
- объединения по максимуму, 466
- объединения по среднему, 467
- объединяющий, 455; 465; 466
- свертки по глубине, 482
- сверточный, 455; 456; 481
 - вычисление выхода нейрона, 461
- сепарабельный, 482
- с множеством карт признаков, 460
- транспонированный, 481
- скрытый, 333; 346
 - количество нейронов на скрытый слой, 347
- суживающий, 474; 475
- Служба
 - Cloud Machine Learning, 403
 - для прогона графов TensorFlow, 296
 - исполнителя, 418
 - мастера, 418
- Смеситель (blender), 262
- Смещение (bias), 178; 472
 - выборки, 52; 53
 - из-за информационного просмотра данных, 81
- Согласованность, 97
- Специфичность, 135
- Среда
 - CartPole, 561
 - создание изолированной среды, 73
- Среднее гармоническое, 130
- Стандартизация, 103
- Стандартное отклонение, 78
- Стекинг (stacking), 240; 261
- Страйд (stride), 457
 - понижение размерности, 458
- Страта (strata), 84
- Субдифференциал, 221
- Считыватель
 - многопоточный, 434

Т

- Телодендрон (telodendria), 325
- Тензор (tensor), 301
 - элементы обработки тензоров (TPU), 403
- Теорема Мерсера, 219
 - об отсутствии бесплатных завтраков, 60
 - о сходимости перцептрона, 331
- Теория информации Шеннона, 66; 230
- Тепловое равновесие, 654
- Технология MapReduce, 68
- Топология сети, 345
- Точка
 - стационарная, 640
- Точность, 29; 128
- Трансформатор, 97
 - StandardScaler, 104
 - специальный, 102
- Трансформация PCA
 - обратная, 280
- Трюк
 - ядерный, 205
 - для полиномиального отображения второй степени, 218

У

- Узел
 - Saver, 308
 - заполнитель, 306
 - листовой, 225
 - оценка узла при помощи TensorFlow, 300
 - с множеством ребер, 313
- Управление
 - зависимости управления, 414
- Уравнение
 - нормальное, 156
 - оптимальности Беллмана, 576; 577
- Условие ККТ, 641
 - дополняющей нежесткости, 641
- Установка TensorFlow, 297
- Устройство (device), 297
 - множество устройств на единственной машине, 402

Ф

- Файл
 - событий, 310
- Фильтр, 458
 - расширенный, 481
 - спама, 27
- Формат CSV, 431
 - TFRecords, 431
 - двоичной записи фиксированной длины, 431
- Фрагментация переменных, 419
- Фрейм, 486
- Функция
 - argmax(), 345
 - array_split(), 281
 - assign(), 303
 - avg_pool(), 467
 - batch(), 438
 - batch_join(), 438
 - clip_by_norm(), 394
 - clip_by_value(), 365
 - confusion_matrix(), 128; 142
 - cross_val_predict(), 127; 132; 142
 - cross_val_score(), 125; 126; 142
 - dense(), 341; 532
 - dequeue_many(), 426; 428
 - dequeue_up_to(), 428
 - device(), 409
 - discount_and_normalize_rewards(), 572; 573
 - dropout(), 392
 - dynamic_rnn(), 495
 - f1_score(), 130
 - get_variable(), 317
 - gradients(), 305
 - import_meta_graph(), 366; 368
 - imread(), 483
 - imshow(), 122; 145
 - in_top_k(), 343
 - inv(), 157
 - make(), 561
 - matshow(), 143
 - max_norm(), 395
 - max_norm_regularizer(), 394
 - max_pool(), 467
 - mean_squared_error(), 107

minimize(), 365
 neuron_layer(), 341
 plot_digits(), 145
 precision_recall_curve(), 132
 reduce_mean(), 342
 ReLU, 335
 randint(), 148
 random_uniform(), 303
 relu(), 315; 316; 317; 318; 319
 reshape(), 100; 302
 roc_curve(), 135
 run(), 442
 scatter_matrix(), 91
 shuffle_batch(), 437
 shuffle_batch_join(), 438
 sigmoid_cross_entropy_with_logits(), 545
 sparse_softmax_cross_entropy_with_logits(), 341; 342
 split_train_test(), 82; 84
 start_queue_runners(), 437
 static_rnn(), 492
 string_input_producer(), 437
 tf.group(), 589
 tf.nn.batch_normalization(), 362
 train_test_split(), 84; 108
 unstack(), 493
 variance_scaling_initializer(), 355
 активации, 334; 335; 348
 ELU, 358; 617
 ReLU, 355; 473
 с уткой, 356
 гиперболического тангенса, 355
 логистическая, 355; 614
 ненасыщаемая, 356
 близости, 206
 выпуклая, 162
 гауссова RBF, 206
 линейного классификатора SVM, 221
 -генератор (producer), 437
 гиперболического тангенса, 335
 гипотезы, 155
 динамического размещения, 410
 издержек, 47; 69; 153; 160; 179; 189
 MSE, 156; 162
 алгоритма CART, 604
 для классификации, 228
 для регрессии, 234
 в форме перекрестной энтропии, 195
 для гребневой регрессии, 180
 для лассо-регрессии, 182
 логистической регрессии, 189
 одиночного обучающего образца, 189
 эластичной сети, 185
 издержек CART
 исследования, 582
 Лагранжа, 640
 обобщенная, 641
 логистическая, 188; 654
 многопеременная, 193; 194
 многопеременная, 335
 непрерывная, 162
 нормализованная экспоненциальная, 193
 петлевая, 202; 221
 полезности, 47
 полиномиальная, 217; 218
 приспособленности, 47
 прогнозирования системы, 69
 производная функции, 646
 радиальная базисная (RBF), 206
 решения, 212; 131
 oob, 247
 сигмоидальная, 188
 сигнум, 329
 стоимости, 47
 ступенчатая, 328
 Хевисайда, 329
 целевая, 640
 частная производная, 164; 644
 энергетическая, 652

X

Хебб Дональд, 330
 обучение по Хеббу, 330
 правило Хебба, 330

Ц

Цветовой канал, 460
 Цепи Маркова, 575
 Цепное правило, 649

Ч

Частичное обучение, 38
 Частная производная, 163

Число

дуальное, 647

Член регуляризации, 179

Чувствительность (sensitivity), 129; 135

Ш

Шеннон

теория информации Шеннона, 66

Шкалирование

многомерное (MDS), 288

Шлюз

входной, 512; 513

выходной, 513

забывания, 512; 513

контроллер шлюзов, 513

Шум выборки (sampling noise), 52

Э

Элемент

линейный пороговый, 328

обработки тензоров (TPU), 403

экспоненциальный линейный, 357

Энтропия, 230

перекрестная, 195

сокращение энтропии, 230

Эпизод, 563

Эпоха, 168

Эффективность параметров, 346

Я

Ядерный анализ

главных компонентов (kPCA), 282

Ядерный трюк, 205; 220

вычисление члена смещения с использованием ядерного трюка, 220

для полиномиального отображения второй степени, 218

Ядро, 218; 339; 411

гауссово RBF, 207; 208

на основе расстояния Левенштейна, 208

объединяющее, 466

полиномиальное, 204

распространенные ядра

гауссово RBF, 219

линейное, 219

полиномиальное, 219

сигмоидальное, 219

сверточное, 458

строковое, 208

Язык

обработка естественного языка, 516

Якобиан, 384

Ячейка

RNN, 502

долгой краткосрочной памяти (LSTM), 511

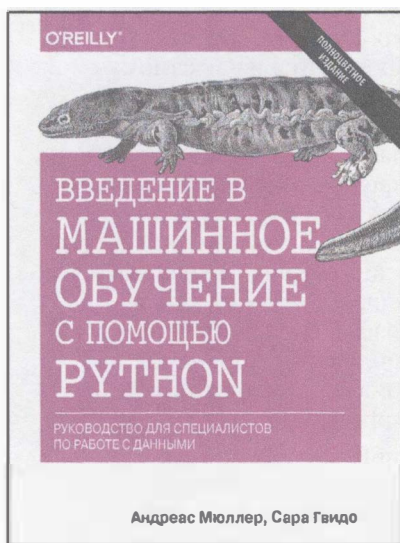
памяти, 444

базовая, 489

управляемого рекуррентного блока (GRU), 514

ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

**Андреас Мюллер
Сара Гвидо**



www.williamspublishing.com

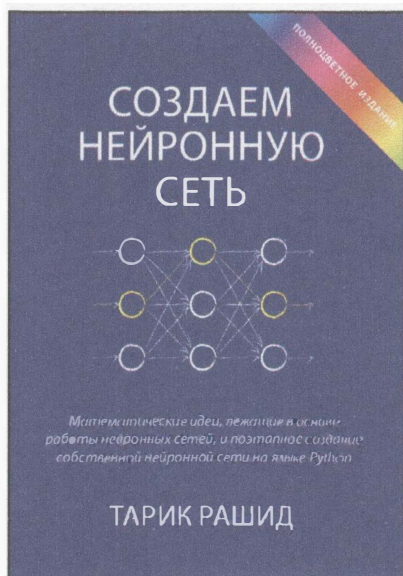
ISBN 978-5-9908910-8-1

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек `scikit-learn`, `NumPy` и `matplotlib`. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

в продаже

СОЗДАЕМ НЕЙРОННУЮ СЕТЬ

Тарик Рашид



www.williamspublishing.com

Эта книга представляет собой введение в теорию и практику создания нейронных сетей. Она предназначена для тех, кто хочет узнать, что такое нейронные сети, где они применяются и как самому создать такую сеть, не имея опыта работы в данной области. Изложение материала сопровождается подробным описанием процедуры поэтапного создания полностью функционального кода, который реализует нейронную сеть на языке Python и способен выполняться даже на таком миниатюрном компьютере, как Raspberry Pi Zero.

Основные темы книги:

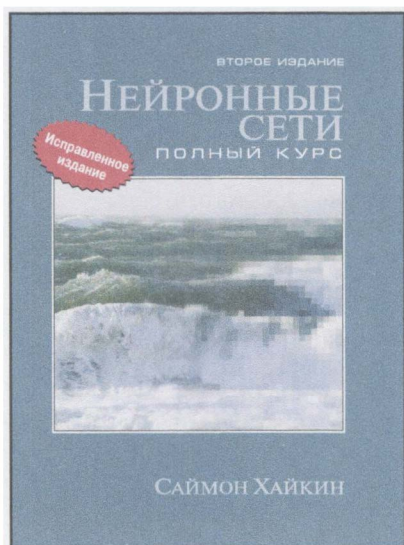
- нейронные сети и системы искусственного интеллекта;
- структура нейронных сетей;
- сглаживание сигналов, распространяющихся по нейронной сети, с помощью функции активации;
- тренировка и тестирование нейронных сетей;
- интерактивная среда программирования IPython;
- распознавание образов с помощью нейронных сетей.

ISBN 978-5-9909445-7-2 **в продаже**

НЕЙРОННЫЕ СЕТИ: ПОЛНЫЙ КУРС

2-Е ИЗДАНИЕ

Саймон Хайкин



www.williamspublishing.com

В книге рассматриваются основные парадигмы искусственных нейронных сетей. Представленный материал содержит строгое математическое обоснование всех нейросетевых парадигм, иллюстрируется примерами, описанием компьютерных экспериментов, содержит множество практических задач, а также обширную библиографию.

В книге также анализируется роль нейронных сетей при решении задач распознавания образов, управления и обработки сигналов. Структура книги очень удобна для разработки курсов обучения нейронным сетям и интеллектуальным вычислениям.

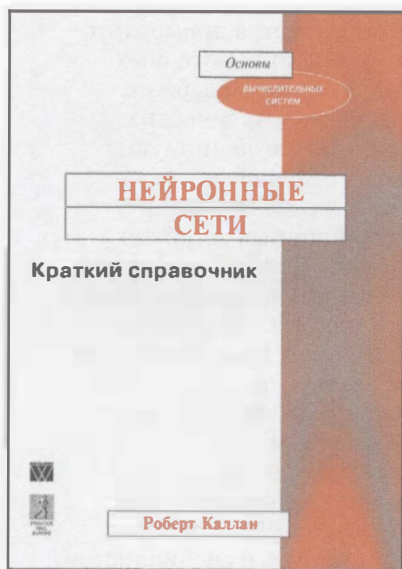
Книга будет полезна для инженеров, специалистов в области компьютерных наук, физиков и специалистов в других областях, а также для всех тех, кто интересуется искусственными нейронными сетями.

ISBN 978-5-8459-2069-0

в продаже

НЕЙРОННЫЕ СЕТИ. КРАТКИЙ СПРАВОЧНИК

Роберт Каллан



Эта книга является первой в полном курсе по нейронным сетям. Ее целью является раскрытие основных понятий и изучение основных моделей нейронных сетей с глубиной, достаточной для того, чтобы опытный программист мог реализовать такую сеть на том языке программирования, который покажется ему предпочтительнее. В книге рассматриваются основные модели нейронных сетей, важные для понимания основ изучаемого предмета, и обсуждаются связи между нейронными сетями и традиционными понятиями из области искусственного интеллекта.

www.williamspublishing.com

ISBN 978-5-8459-2131-4 в продаже

Прикладное машинное обучение с помощью Scikit-Learn и TensorFlow

Благодаря серии недавних достижений глубокое обучение значительно усилило всю область машинного обучения. В наше время даже программисты, почти ничего не знающие об этой технологии, могут использовать простые и эффективные инструменты для реализации программ, которые способны обучаться на основе данных. В настоящем практическом руководстве показано, что и как следует делать.

За счет применения конкретных примеров, минимума теории и двух фреймворков Python производственного уровня — Scikit-Learn и TensorFlow — автор книги Орельен Жерон поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем. Вы узнаете о ряде приемов, начав с простой линейной регрессии и постепенно добравшись до глубоких нейронных сетей. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования

- Исследуйте область машинного обучения, особенно нейронные сети
- Используйте Scikit-Learn для отслеживания проекта машинного обучения от начала до конца
- Исследуйте некоторые обучающие модели, включая методы опорных векторов, деревья принятия решений, случайные леса и ансамблевые методы
- Применяйте библиотеку TensorFlow для построения и обучения нейронных сетей
- Исследуйте архитектуры нейронных сетей, включая сверточные сети, рекуррентные сети и глубокое обучение с подкреплением
- Освойте приемы для обучения и масштабирования глубоких нейронных сетей
- Используйте практические примеры кода, не овладевая чрезмерно теорией машинного обучения или деталями алгоритмов

Орельен Жерон — консультант по машинному обучению. Бывший работник компании Google, с 2013 по 2016 год он руководил командой классификации видеороликов YouTube. С 2002 по 2012 год он также был основателем и руководителем технического отдела в компании Wifirst, ведущего поставщика услуг беспроводного доступа к Интернету во Франции, а в 2001 году — основателем и руководителем технического отдела в компании Polyconseil, которая сейчас управляет сервисом совместного пользования электромобилями Autolib'.

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей. Она охватывает ключевые моменты, необходимые для построения эффективных приложений, а также обеспечивает достаточную основу для понимания результатов новых исследований по мере их появления. Я рекомендую эту книгу всем, кто заинтересован в освоении практического машинного обучения”.

Пит Уорден,
технический руководитель
направления TensorFlow

Категория: данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

ISBN 978-5-9500296-2-2

