

O'REILLY®

2-е  
издание

# Прикладное машинае обучение с помощью Scikit-Learn, Keras и TensorFlow

Концепции, инструменты и техники  
для создания интеллектуальных  
систем

powered by



Орельен Жерон

2-е издание

---

**Прикладное  
машинное обучение  
с помощью  
Scikit-Learn, Keras  
и TensorFlow**

SECOND EDITION

---

# Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow

Concepts, Tools, and Techniques to Build  
Intelligent Systems

*Aurélien Géron*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

# Прикладное машинаное обучение с помощью **Scikit-Learn, Keras** и **TensorFlow**

Концепции, инструменты и техники для  
создания интеллектуальных систем

*Орельен Жерон*



Москва · Санкт-Петербург  
2020

ББК 32.973.26-018.2.75

Ж61

УДК 681.3.07

ООО "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@gmail.com](mailto:info.dialektika@gmail.com), <http://www.dialektika.com>

Жерон, Орельен.

Ж61 Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow: концепции, инструменты и техники для создания интеллектуальных систем , 2-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 1040 с. : ил. — Парал. тит. англ.

ISBN 978-5-907203-33-4 (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 Media, Inc.

Authorized Russian translation of the English edition of *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*, 2nd Edition (ISBN 978-1-492-03264-9) © 2019 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, Keras и TensorFlow:

концепции, инструменты и техники для создания

интеллектуальных систем

2-е издание

Подписано в печать 18.09.2020. Формат 70x100/16

Гарнитура Times

Усл. печ. л. 83,85. Уч.-изд. л. 56,2

Тираж 500 экз. Заказ № 6014.

Отпечатано в АО "Первая Образцовая типография"

Филиал "Чеховский Печатный Двор"

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

Сайт: [www.chpd.ru](http://www.chpd.ru), E-mail: [sales@chpd.ru](mailto:sales@chpd.ru), тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул.. д. 30, лит. А, пом. 848

ISBN 978-5-907203-33-4 (рус.)

© ООО "Диалектика", 2020,  
перевод, оформление, макетирование

© 2019 Aurélien Géron. All rights reserved

ISBN 978-1-492-03264-9 (англ.)

# Оглавление

<b>Часть I. Основы машинного обучения</b>	33
Глава 1. Введение в машинное обучение	35
Глава 2. Полный проект машинного обучения	77
Глава 3. Классификация	139
Глава 4. Обучение моделей	171
Глава 5. Методы опорных векторов	219
Глава 6. Деревья принятия решений	245
Глава 7. Аансамблевое обучение и случайные леса	261
Глава 8. Понижение размерности	289
Глава 9. Методики обучения без учителя	315
<b>Часть II. Нейронные сети и глубокое обучение</b>	367
Глава 10. Введение в искусственные нейронные сети с использованием Keras	369
Глава 11. Обучение глубоких нейронных сетей	435
Глава 12. Специальные модели и обучение с помощью TensorFlow	491
Глава 13. Загрузка и предварительная обработка данных с помощью TensorFlow	537
Глава 14. Глубокое компьютерное зрение с использованием сверточных нейронных сетей	579
Глава 15. Обработка последовательностей с использованием рекуррентных и сверточных нейронных сетей	645
Глава 16. Обработка естественного языка с помощью рекуррентных нейронных сетей и внимания	679
Глава 17. Обучение представлению и порождению с использованием автокодировщиков и порождающих состязательных сетей	733
Глава 18. Обучение с подкреплением	785
Глава 19. Широкомасштабное обучение и развертывание моделей TensorFlow	857
Приложение А. Решения упражнений	924
Приложение Б. Контрольный перечень для проекта машинного обучения	974
Приложение В. Двойственная задача SVM	981
Приложение Г. Автоматическое дифференцирование	984
Приложение Д. Другие популярные архитектуры искусственных нейронных сетей	992
Приложение Е. Специальные структуры данных	1003
Приложение Ж. Графы TensorFlow	1011
Предметный указатель	1021

# Содержание

Об авторе	18
Благодарности	19
<b>Предисловие</b>	22
Цунами машинного обучения	22
Машинное обучение в ваших проектах	23
Цель и подход	23
Предварительные требования	24
Дорожная карта	25
Изменения во втором издании	26
Другие ресурсы	27
Типографские соглашения, используемые в книге	29
Код примеров	30
Использование кода примеров	30
Об иллюстрации на обложке	31
Ждем ваших отзывов!	32
<b>Часть I. Основы машинного обучения</b>	33
<b>Глава 1. Введение в машинное обучение</b>	35
Что такое машинное обучение?	36
Для чего используют машинное обучение?	37
Примеры приложений	40
Типы систем машинного обучения	42
Обучение с учителем и без учителя	43
Пакетное и динамическое обучение	51
Обучение на основе образцов или на основе моделей	55
Основные проблемы машинного обучения	62
Недостаточный размер обучающих данных	62
Нерепрезентативные обучающие данные	64
Данные плохого качества	66
Несущественные признаки	66
Переобучение обучающими данными	67
Недообучение обучающими данными	69
Шаг назад	70
Испытание и проверка	71
Настройка гиперпараметра и подбор модели	71
Несоответствие данных	73
Упражнения	75

<b>Глава 2. Полный проект машинного обучения</b>	77
Работа с реальными данными	77
Выяснение общей картины	79
Постановка задачи	79
Выбор критерия качества работы	82
Проверка допущений	85
Получение данных	85
Создание рабочей области	85
Загрузка данных	90
Беглый взгляд на структуру данных	91
Создание испытательного набора	96
Обнаружение и визуализация данных для понимания их сущности	101
Визуализация географических данных	102
Поиск связей	104
Экспериментирование с комбинациями атрибутов	107
Подготовка данных для алгоритмов машинного обучения	109
Очистка данных	110
Обработка текстовых и категориальных атрибутов	113
Специальные трансформаторы	116
Масштабирование признаков	117
Конвейеры трансформации	119
Выбор и обучение модели	121
Обучение и оценка с помощью обучающего набора	121
Более подходящая оценка с использованием перекрестной проверки	123
Точная настройка модели	126
Решетчатый поиск	126
Рандомизированный поиск	129
Ансамблевые методы	130
Анализ лучших моделей и их ошибок	130
Оценка системы с помощью испытательного набора	131
Запуск, наблюдение и сопровождение системы	132
Пробуйте!	137
Упражнения	137
<b>Глава 3. Классификация</b>	139
MNIST	139
Обучение двоичного классификатора	142
Показатели эффективности	143
Измерение правильности с использованием перекрестной проверки	143
Матрица неточностей	145
Точность и полнота	147
Соотношение точность/полнота	149
Кривая ROC	153

<b>Многоклассовая классификация</b>	157
Анализ ошибок	160
<b>Многозначная классификация</b>	164
<b>Многовходовая классификация</b>	166
Упражнения	168
<b>Глава 4. Обучение моделей</b>	171
Линейная регрессия	172
Нормальное уравнение	174
Вычислительная сложность	178
Градиентный спуск	178
Пакетный градиентный спуск	182
Стochastic gradient descent	186
Мини-пакетный градиентный спуск	189
Полиномиальная регрессия	191
Кривые обучения	193
Регуляризованные линейные модели	198
Гребневая регрессия	198
Лассо-регрессия	201
Эластичная сеть	204
Раннее прекращение	205
Логистическая регрессия	207
Оценивание вероятностей	207
Обучение и функция издержек	208
Границы решений	210
Многопеременная логистическая регрессия	213
Упражнения	217
<b>Глава 5. Методы опорных векторов</b>	219
Линейная классификация SVM	219
Классификация с мягким зазором	220
Нелинейная классификация SVM	223
Полиномиальное ядро	224
Признаки близости	226
Гауссово ядро RBF	227
Вычислительная сложность	229
Регрессия SVM	230
Внутренняя кухня	232
Функция решения и прогнозы	232
Цель обучения	233
Квадратичное программирование	235
Двойственная задача	236
Параметрически редуцированные методы SVM	237
Динамические методы SVM	240
Упражнения	242

<b>Глава 6. Деревья принятия решений</b>	245
Обучение и визуализация дерева принятия решений	245
Вырабатывание прогнозов	247
Оценивание вероятностей классов	249
Алгоритм обучения CART	250
Вычислительная сложность	251
Загрязненность Джини или энтропия?	252
Гиперпараметры регуляризации	253
Регрессия	254
Неустойчивость	257
Упражнения	258
<b>Глава 7. Ансамблевое обучение и случайные леса</b>	261
Классификаторы с голосованием	262
Бэггинг и вставка	265
Бэггинг и вставка в Scikit-Learn	267
Оценка на неиспользуемых образцах	268
Методы случайных участков и случайных подпространств	270
Случайные леса	270
Особо случайные деревья	272
Значимость признаков	272
Бустинг	274
AdaBoost	274
Градиентный бустинг	278
Стекинг	283
Упражнения	287
<b>Глава 8. Понижение размерности</b>	289
“Проклятие размерности”	290
Основные подходы к понижению размерности	292
Проекция	292
Обучение на основе многообразий	294
PCA	296
Предохранение дисперсии	296
Главные компоненты	297
Проектирование до $d$ измерений	299
Использование Scikit-Learn	300
Коэффициент объясненной дисперсии	300
Выбор правильного количества измерений	300
Алгоритм PCA для сжатия	302
Рандомизированный анализ главных компонентов	303
Инкрементный анализ главных компонентов	303

<b>Ядерный анализ главных компонентов</b>	304
Выбор ядра и подстройка гиперпараметров	306
<b>LLE</b>	308
Другие методики понижения размерности	311
Упражнения	312
<b>Глава 9. Методики обучения без учителя</b>	315
Кластеризация	316
K-Means	319
Ограничения K-Means	331
Использование кластеризации для сегментирования изображений	332
Использование кластеризации для предварительной обработки	334
Использование кластеризации для частичного обучения	336
DBSCAN	340
Другие алгоритмы кластеризации	343
Смеси гауссовых распределений	346
Обнаружение аномалий с использованием смесей гауссовых распределений	352
Выбор количества кластеров	354
Байесовские модели со смесями гауссовых распределений	357
Другие алгоритмы для обнаружения аномалий и новизны	363
Упражнения	364
<b>Часть II. Нейронные сети и глубокое обучение</b>	367
<b>Глава 10. Введение в искусственные нейронные сети с использованием Keras</b>	369
От биологических нейронов к искусственным нейронам	370
Биологические нейроны	372
Логические вычисления с помощью нейронов	374
Персептрон	375
Многослойный персептрон и обратное распространение	380
Многослойные персептроны для регрессии	385
Многослойные персептроны для классификации	386
Реализация многослойных персептронов с помощью Keras	388
Установка TensorFlow 2	390
Построение классификатора изображений с использованием API-интерфейса Sequential	391
Построение многослойного персептрана для регрессии с использованием API-интерфейса Sequential	404
Построение сложных моделей с использованием API-интерфейса Functional	405
Использование API-интерфейса Subclassing для построения динамических моделей	411
Сохранение и восстановление модели	413
Использование обратных вызовов	414
Использование TensorBoard для визуализации	416

Точная настройка гиперпараметров нейронной сети	420
Количество скрытых слоев	424
Количество нейронов на скрытый слой	426
Скорость обучения, размер пакета и другие гиперпараметры	427
Упражнения	430
<b>Глава 11. Обучение глубоких нейронных сетей</b>	435
Проблемы исчезновения и взрывного роста градиентов	436
Инициализация Глоро и Хе	437
Ненасыщаемые функции активации	440
Пакетная нормализация	445
Отсечение градиентов	453
Повторное использование заранее обученных слоев	454
Обучение передачей знаний с помощью Keras	456
Предварительное обучение без учителя	459
Предварительное обучение на вспомогательной задаче	460
Более быстрые оптимизаторы	461
Моментная оптимизация	462
Ускоренный градиент Нестерова	464
AdaGrad	465
RMSProp	467
Оптимизация Adam и Nadam	467
Планирование скорости обучения	472
Избегание переобучения посредством регуляризации	477
Регуляризация $\ell_1$ и $\ell_2$	478
Отключение	479
Отключение Монте-Карло	483
Регуляризация на основе тах-нормы	486
Резюме и практические рекомендации	487
Упражнения	489
<b>Глава 12. Специальные модели и обучение с помощью TensorFlow</b>	491
Краткий тур по TensorFlow	492
Использование TensorFlow подобно NumPy	496
Тензоры и операции	496
Тензоры и NumPy	498
Преобразования типов	499
Переменные	500
Другие структуры данных	501
Настройка моделей и алгоритмов обучения	502
Специальные функции потерь	502
Сохранение и загрузка моделей, которые содержат специальные компоненты	503

Специальные функции активации, инициализаторы, регуляризаторы и ограничения	506
Специальные метрики	507
Специальные слои	511
Специальные модели	515
Потери и метрики, основанные на внутренностях модели	517
Вычисление градиентов с использованием автоматического дифференцирования	520
Специальные циклы обучения	524
Функции и графы TensorFlow	528
AutoGraph и трассировка	531
Правила TF Function	532
Упражнения	535
<b>Глава 13. Загрузка и предварительная обработка данных с помощью TensorFlow</b>	537
API-интерфейс Data	538
Формирование цепочки трансформаций	539
Тасование данных	541
Предварительная обработка данных	545
Собираем все вместе	547
Предварительная выборка	548
Использование набора данных с библиотекой <code>tf.keras</code>	550
Формат TFRecord	552
Сжатые файлы TFRecord	552
Краткое введение в протокольные буферы	553
Протобуферы TensorFlow	555
Загрузка и разбор протобуферов <code>Example</code>	556
Обработка списка списков с использованием протобуфера <code>SequenceExample</code>	558
Предварительная подготовка входных признаков	559
Кодирование категориальных признаков с использованием векторов в унитарном коде	561
Кодирование категориальных признаков с использованием вложений	564
Слои предварительной обработки Keras	569
TF Transform	572
Проект TensorFlow Datasets (TFDS)	574
Упражнения	576
<b>Глава 14. Глубокое компьютерное зрение с использованием сверточных нейронных сетей</b>	579
Строение зрительной коры головного мозга	580
Сверточные слои	582
Фильтры	585
Наложение множества карт признаков	586

Реализация с помощью TensorFlow	588
Требования к памяти	591
Объединяющие слои	592
Реализация в TensorFlow	595
Архитектуры сверточных нейронных сетей	597
LeNet-5	600
AlexNet	602
GoogLeNet	605
VGGNet	609
ResNet	609
Xception	613
SENet	616
Реализация сверточной нейронной сети ResNet-34 с использованием Keras	618
Использование заранее обученных моделей из Keras	620
Использование заранее обученных моделей для обучения передачей знаний	622
Классификация и установление местонахождения	626
Выявление объектов	628
Полностью сверточные сети	630
Вы просматриваете только раз (YOLO)	633
Семантическая сегментация	637
Упражнения	642
<b>Глава 15. Обработка последовательностей с использованием рекуррентных и сверточных нейронных сетей</b>	645
Рекуррентные нейроны и слои	646
Ячейки памяти	649
Входные и выходные последовательности	650
Обучение рекуррентных нейронных сетей	651
Прогнозирование временных рядов	652
Метрики базисного уровня	654
Реализация простой рекуррентной нейронной сети	655
Глубокие рекуррентные нейронные сети	657
Прогнозирование на несколько временных шагов вперед	659
Обработка длинных последовательностей	663
Борьба с проблемой нестабильных градиентов	664
Борьба с проблемой краткосрочной памяти	667
Упражнения	677
<b>Глава 16. Обработка естественного языка с помощью рекуррентных нейронных сетей и внимания</b>	679
Генерация шекспировского текста с использованием символьной сети RNN	681
Создание обучающего набора данных	681
Расщепление последовательного набора данных	683

Разрезание последовательного набора данных на множество окон	684
Построение и обучение модели Char-RNN	686
Использование модели Char-RNN	687
Генерирование поддельного шекспировского текста	688
Сеть RNN с запоминанием состояния	689
<b>Смысловой анализ</b>	692
Маскирование	697
Повторное использование заранее обученных вложений	700
<b>Сеть “кодировщик–декодировщик” для нейронного машинного перевода</b>	702
Двунаправленные рекуррентные нейронные сети	707
Лучевой поиск	708
<b>Механизмы внимания</b>	710
Зрительное внимание	714
Внимание — это все, что нужно: архитектура “Преобразователь”	716
Последние новшества в языковых моделях	727
Упражнения	731
<b>Глава 17. Обучение представлению и порождению с использованием автокодировщиков и порождающих состязательных сетей</b>	733
Эффективные представления данных	735
Выполнение анализа главных компонентов с помощью понижающего линейного автокодировщика	737
Многослойные автокодировщики	739
Реализация многослойного автокодировщика с использованием Keras	739
Визуализация реконструкций	741
Визуализация набора данных Fashion MNIST	742
Предварительное обучение без учителя с использованием многослойных автокодировщиков	744
Соединение весов	745
Обучение одного автокодировщика за раз	746
Сверточные автокодировщики	748
Рекуррентные автокодировщики	749
Шумоподавляющие автокодировщики	750
Разреженные автокодировщики	753
Вариационные автокодировщики	757
Генерирование изображений Fashion MNIST	762
Порождающие состязательные сети	764
Трудности обучения порождающих состязательных сетей	769
Глубокие сверточные порождающие состязательные сети	771
Прогрессивный рост порождающих состязательных сетей	775
Сети StyleGAN	779
Упражнения	782

<b>Глава 18. Обучение с подкреплением</b>	785
Обучение для оптимизации наград	786
Поиск политики	788
Введение в OpenAI Gym	790
Нейросетевые политики	795
Оценка действий: проблема присваивания коэффициентов доверия	797
Градиенты политики	799
Марковские процессы принятия решений	805
Обучение методом временных разностей	811
Q-обучение	812
Политики исследования	814
Приближенное Q-обучение и глубокое Q-обучение	815
Реализация глубокого Q-обучения	817
Варианты глубокого Q-обучения	822
Фиксированные цели Q-ценности	823
Двойная глубокая Q-сеть	824
Воспроизведение опытов по приоритетам	825
Соревнующаяся глубокая Q-сеть	826
Библиотека TF-Agents	827
Установка TF-Agents	828
Среды TF-Agents	829
Спецификации среды	830
Оболочки сред и предварительная обработка Atari	831
Структура обучения	835
Создание глубокой Q-сети	837
Создание агента DQN	840
Создание буфера воспроизведения и соответствующего наблюдателя	841
Создание метрик обучения	844
Создание драйвера сбора	845
Создание набора данных	847
Создание цикла обучения	850
Обзор ряда популярных алгоритмов обучения с подкреплением	852
Упражнения	855
<b>Глава 19. Широкомасштабное обучение и развертывание моделей TensorFlow</b>	857
Обслуживание модели TensorFlow	858
Использование TensorFlow Serving	859
Создание службы прогнозирования на облачной платформе Google	871
Использование службы прогнозирования	876
Развертывание модели на мобильном или встроенном устройстве	880
Использование графических процессоров для ускорения вычислений	886
Получение собственного графического процессора	887

Использование виртуальной машины, оснащенной графическим процессором	890
Среда Colaboratory	891
Управление оперативной памятью графического процессора	893
Размещение операций и переменных на устройствах	897
Параллельное выполнение на множестве устройств	899
Обучение моделей на множестве устройств	902
Параллелизм модели	902
Параллелизм данных	905
Обучение и масштабирование с использованием API-интерфейса Distribution Strategies	911
Обучение модели на кластере TensorFlow	913
Запуск крупных заданий обучения на платформе AI Platform инфраструктуры Google Cloud	917
Служба подстройки гиперпараметров типа “черный ящик” платформы AI Platform	920
Упражнения	921
Спасибо!	922
<b>Приложение А. Решения упражнений</b>	924
Глава 1. Введение в машинное обучение	924
Глава 2. Полный проект машинного обучения	927
Глава 3. Классификация	927
Глава 4. Обучение моделей	927
Глава 5. Методы опорных векторов	930
Глава 6. Деревья принятия решений	932
Глава 7. Ансамблевое обучение и случайные леса	934
Глава 8. Понижение размерности	935
Глава 9. Методики обучения без учителя	938
Глава 10. Введение в искусственные нейронные сети с использованием Keras	940
Глава 11. Обучение глубоких нейронных сетей	944
Глава 12. Специальные модели и обучение с помощью TensorFlow	946
Глава 13. Загрузка и предварительная обработка данных с помощью TensorFlow	949
Глава 14. Глубокое компьютерное зрение с использованием сверточных нейронных сетей	953
Глава 15. Обработка последовательностей с использованием рекуррентных и сверточных нейронных сетей	957
Глава 16. Обработка естественного языка с помощью рекуррентных нейронных сетей и внимания	961
Глава 17. Обучение представлению и порождению с использованием автокодировщиков и порождающих состязательных сетей	964
Глава 18. Обучение с подкреплением	966
Глава 19. Широкомасштабное обучение и развертывание моделей TensorFlow	970

<b>Приложение Б. Контрольный перечень для проекта машинного обучения</b>	974
Постановка задачи и выяснение общей картины	974
Получение данных	975
Исследование данных	976
Подготовка данных	977
Составление окончательного списка перспективных моделей	978
Точная настройка системы	978
Представление своего решения	979
Запуск!	980
<b>Приложение В. Двойственная задача SVM</b>	981
<b>Приложение Г. Автоматическое дифференцирование</b>	984
Ручное дифференцирование	984
Конечно-разностное приближение	985
Автоматическое дифференцирование в прямом режиме	986
Автоматическое дифференцирование в обратном режиме	989
<b>Приложение Д. Другие популярные архитектуры искусственных нейронных сетей</b>	992
Сети Хопфилда	992
Машины Больцмана	994
Ограниченные машины Больцмана	996
Глубокие сети доверия	998
Самоорганизующиеся карты	1000
<b>Приложение Е. Специальные структуры данных</b>	1003
Строки	1003
Зубчатые тензоры	1004
Разреженные тензоры	1006
Тензорные массивы	1006
Множества	1007
Очереди	1009
<b>Приложение Ж. Графы TensorFlow</b>	1011
Функции TF Function и конкретные функции	1011
Исследование определений и графов функций	1013
Более пристальный взгляд на трассировку	1015
Использование AutoGraph для захвата потока управления	1017
Обработка переменных и других ресурсов в функциях TF Function	1018
Использование функций TF Function с <code>tf.keras</code>	1020
<b>Предметный указатель</b>	1021

## Об авторе

**Орельен Жерон** — консультант и лектор по машинному обучению. Бывший работник компании Google, с 2013 по 2016 год он руководил командой классификации видеороликов YouTube. Он также был основателем и руководителем технического отдела в нескольких компаниях: Wifirst (ведущий поставщик услуг беспроводного доступа к Интернету во Франции), Polyconseil (консалтинговая фирма в области телекоммуникаций) и Kiwisoft (консалтинговая фирма в области машинного обучения и неприкословенности данных).

До этого Орельен работал инженером в различных областях: финансовой (JP Morgan и Société Générale), оборонной (Министерство национальной обороны Канады) и здравоохранения (переливание крови). Он также опубликовал несколько книг технического характера (по C++, WiFi и сетевым архитектурам) и читал лекции о компьютерных науках во французской инженерной школе.

Несколько забавных фактов: он учил своих трех детей считать в двоичной форме с помощью пальцев (до 1 023), до перехода в область разработки программного обеспечения изучал микробиологию и эволюционную генетику.

# Благодарности

Даже в своих самых смелых мечтах я не мог вообразить, что первое издание книги обретет настолько большую аудиторию. Я получил чрезвычайно много сообщений от читателей, многие задавали вопросы, некоторые дружелюбно указывали на ошибки, а большинство отправляли мне слова одобрения. Не могу выразить, насколько я признателен всем читателям за их потрясающую поддержку. Огромное спасибо вам всем! Пожалуйста, без колебаний протоколируйте проблемы на GitHub (<https://homl.info/issues2>), если обнаружите ошибки в коде примеров (или просто задавайте вопросы). Некоторые читатели также поделились тем, как эта книга помогла им получить первую работу или решить конкретную задачу, над которой они трудились. Я считаю такую обратную связь невероятно мотивирующей. Если вы найдете эту книгу полезной, я буду рад, если вы сможете поделиться со мной своей историей (например, через LinkedIn (<https://www.linkedin.com/in/aurelien-geron/>)).

Я также чрезвычайно признателен всем замечательным людям, находящим время в своей занятой жизни для критического анализа моей книги с такой тщательностью. В частности, я хочу поблагодарить Франсуа Шолле за рецензирование всех глав, основанных на Keras и TensorFlow, и представление мне большого исчерпывающего отзыва. Поскольку библиотека Keras является одним из главных дополнений второго издания, рецензирование книги автором данной библиотеки было поистине бесценным. Я настоятельно рекомендую прочитать книгу Франсуа *Deep Learning with Python* (Manning; <https://homl.info/cholletbook>): в ней отражена лаконичность, ясность и глубина самой библиотеки Keras. Особая благодарность также Анкуру Пателю, рецензировавшему каждую главу второго издания и предоставившему мне превосходный отзыв, в особенности на главу 9, в которой раскрываются методики обучения без учителя. Он мог бы написать целую книгу по данной теме... ох, подождите, он уже это сделал! Взгляните на книгу *Hands-On Unsupervised Learning Using Python: How to Build Applied Machine Learning Solutions from Unlabeled Data* (O'Reilly; <https://homl.info/patel>). Огромная благодарность Олжасу Акпамбетову, который пересмотрел все главы из второй части книги, протестировал большинство кода и дал множество ценных советов. Я признателен Марку Даусту, Джону

Крону, Доминику Монну и Джошу Паттерсону за скрупулезный пересмотр второй части книги и предоставление их экспертной оценки. Они перепахали буквально все и обеспечили удивительную обратную связь.

Во время написания второго издания мне посчастливилось получить основательную помощь от членов команды разработчиков TensorFlow — в частности, Мартина Вика, без устали отвечавшего на десятки моих вопросов и отправлявшего остальные сведущим людям, в список которых входят Кармел Эллисон, Пейдж Бейли, Юджин Бревдо, Уильям Чаргин, Дэниэл “Вольф” Добсон, Ник Фелт, Брюс Фонтейн, Голди Гадд, Сандип Гулта, Прайя Гупта, Кевин Хаас, Константинос Катсиапис, Вячеслав Ковалевский, Аллен Лавуа, Клеменс Мевалд, Дэн Молдован, Шон Морган, Том О’Молли, Александр Пассос, Андре Сюзано Пинто, Энтони Платаниос, Оскар Рамирез, Анна Ревинская, Саурабх Саксена, Райян Сепасси, Джири Симса, Сюдань Сун, Кристина Сорокина, Дастин Тран, Todd Ванг, Пит Уорден (он также рецензировал первое издание), Эдд Вилдер-Джеймс и Юэфэн Чжоу — все они были чрезвычайно любезными. Огромное спасибо вам и всем остальным членам команды разработчиков TensorFlow, не только за вашу помощь, но и за то, что вы сделали настолько великолепную библиотеку! Особая благодарность Ирэн Джанnumис и Роберту Кроуву из команды TFX за тщательный пересмотр глав 13 и 19.

Благодарю фантастических сотрудников издательства O'Reilly, в особенности Николь Таш, которая обеспечивала замечательную обратную связь и была всегда радостной, ободряющей и любезной: о таком редакторе можно мечтать. Большое спасибо Мишель Кронин за предупредительность (и терпение) в начале этого второго издания и Кристен Браун, технического редактора второго издания, за то, что провела его через все этапы (она также координировала внесение исправлений и обновлений для всех допечаток первого издания). Благодарю Рейчел Монеген и Аманду Кеси за их тщательное редактирование копий (в первом и втором изданиях), и Джонни О’Тула, который заведовал взаимодействием с Amazon и отвечал на множество моих вопросов. Спасибо Мари Богуро, Бену Лорику, Майку Лукидесу и Лорель Рюма за то, что они верили в проект и помогали мне определить его рамки. Благодарю Мэтта Хакера и всех из команды Atlas за ответы на мои вопросы о форматировании, AsciiDoc и LaTeX, а также Ника Адамса, Ребекку Демарест, Рейчел Хед, Джудит Макконвилл, Хелен Монро, Карен Монтгомери, Рейчел Румелиотис и всех остальных, кто привнес свой вклад в эту книгу.

Я также хочу поблагодарить моих бывших 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 первое издание. Особая благодарность моим персональным гуру по машинному обучению: Клемену Курбе, Жюльену Дюбуа, Матиасу Кенде, Дэниэлу Китачевски, Джеймсу Пэкку, Александеру Паку, Аношу Раджу, Витору Сессаку, Виктору Томчаку, Ингрид фон Глен, Ричу Уошингтону. И благодарю всех остальных, с кем я работал в YouTube и в изумительной исследовательской команде Google в Маунтин-Вью. Также большое спасибо Мартину Эндрюсу, Сэму Уиттевину и Джейсону Земену за то, что пригласили меня в свою группу Google Developer Experts в Сингапуре, за сердечную поддержку Сунсона Квона и за все наши продолжительные обсуждения глубокого обучения и TensorFlow. Любой, кто интересуется глубоким обучением, просто обязан побывать на встрече Deep Learning Singapore (<https://hml.info/meetupsg>). Джейсон заслуживает особой благодарности за то, что поделился своими знаниями TFLite для главы 19!

Я никогда не забуду замечательных людей, рецензировавших первое издание этой книги, в числе которых Дэвид Анджиевски, Лукас Бивальд, Джастин Фрэнсис, Венсан Гильбо, Эдди Хонг, Карим Матра, Грегуар Мениль, Салим Семаун, Иэн Смез, Мишель Тесье, Ингрид фон Глен, Пит Уорден и, конечно же, мой дорогой брат Сильвен. Выражаю особую благодарность Хассуну Парку, предоставившему мне массу отзывов и нашедшему серьезные ошибки, пока он переводил первое издание книги на корейский язык. Он также переводил на корейский язык тетради Jupyter, не говоря уже о документации по TensorFlow. Я не понимаю корейский, но судя по качеству отзывов, все его переводы должны быть действительно превосходными! Хассун также любезно предоставил несколько решений упражнений из второго издания.

И последнее, но не менее важное: я бесконечно признателен моей любимой жене Эмманюэль и трем замечательным детям, Александру, Реми и Габриель, за то, что они поддерживали меня в интенсивной работе над этой книгой. Я также благодарен им за ненасытное любопытство: объяснение моей жене и детям некоторых из самых сложных понятий в этой книге помогло мне прояснить мои мысли и напрямую улучшило многие ее части. И они продолжали приносить мне печенье и кофе! О чем еще можно было мечтать?

---

# Предисловие

## Цунами машинного обучения

В 2006 году Джекфри Хинтон и др. опубликовали статью (<https://homl.info/136>)<sup>1</sup>, 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*). Глубокая нейронная сеть является (весьма) упрощенной моделью коры нашего головного мозга, состоящей из стопки слоев искусственных нейронов. Обучение глубокой нейронной сети в то время считалось невозможным<sup>2</sup> 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*) не мог даже приблизиться к таким достижениям (при помощи гигантской вычислительной мощности и огромных объемов данных). Энтузиазм быстро распространился на многие другие области машинного обучения.

Спустя 10 лет или около того машинное обучение завоевало отрасль: оно лежит в основе большей части магии сегодняшних высокотехнологичных продуктов, упорядочивая результаты ваших поисковых запросов, приводя в действие распознавание речи в вашем смартфоне, рекомендуя к просмотру видеоролики и одержав победу над чемпионом мира по игре го. Вы не успеете оглянуться, как машинное обучение начнет вести ваш автомобиль.

---

<sup>1</sup> Джекфри Хинтон и др., *A Fast Learning Algorithm for Deep Belief Nets* (Быстрый алгоритм обучения для глубоких сетей доверия), *Neural Computation* 18 (2006 г.): с. 1527–1554.

<sup>2</sup> Несмотря на тот факт, что глубокие сверточные нейронные сети Яна Лекуна хорошо работали при распознавании изображений с 1990-х годов, они не были до такой степени универсальными.

# 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 вероятнее всего вы сумели бы найти там несколько скрытых жемчужин, просто зная, где искать. С помощью машинного обучения вы могли бы решить перечисленные ниже задачи, а также многие другие (<https://homl.info/usecases>):

- сегментировать заказчиков и установить наилучшую 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.

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, инструменты и идеи, которые необходимы для реализации программ, способных *учиться на основе данных*.

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

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

- Библиотека Scikit-Learn (<http://scikit-learn.org/>) очень проста в использовании, но эффективно реализует многие алгоритмы машин-

ного обучения, что делает ее великолепной отправной точкой при изучении машинного обучения.

- TensorFlow (<http://tensorflow.org/>) является более сложной библиотекой для распределенных численных расчетов. Она позволяет эффективно обучать и запускать очень большие нейронные сети, потенциально распределяя вычисления между сотнями серверов с множеством графических процессоров. Библиотека TensorFlow (TF) была создана в компании Google и поддерживает многие из их крупномасштабных приложений машинного обучения. В ноябре 2015 года она стала продуктом с открытым кодом.
- Библиотека Keras (<https://keras.io/>) представляет собой высокоуровневый API-интерфейс для глубокого обучения, который делает очень простым обучение и запуск нейронных сетей. Она может функционировать поверх библиотек TensorFlow, Theano либо инструментального комплекта Microsoft Cognitive Toolkit (ранее известного как CNTK). Библиотека TensorFlow поставляется с собственной реализацией такого API-интерфейса, называемой `tf.keras`, которая предлагает поддержку для ряда расширенных возможностей TensorFlow (скажем, способности эффективно загружать данные).

Предпочтение в книге отдается 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, я настоятельно рекомендую экспериментировать с кодом примеров, который доступен в виде тетрадей Jupyter по адресу <https://github.com/ageron/handson-ml2>.

## Предварительные требования

В книге предполагается, что вы обладаете некоторым опытом программирования на языке 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/>).

Вдобавок 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/>).

Если вы никогда не работали с Jupyter, то в главе 2 будет описан процесс 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 руководство по линейной алгебре.

## Дорожная карта

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

- Что собой представляет машинное обучение, задачи, которые оно пытается решать, а также основные категории и фундаментальные концепции его систем.
- 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.
- Проблемы машинного обучения, в особенности недообучение и переобучение (компромисс между смещением и дисперсией).
- 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.
- Понижение размерности обучающих данных в целях борьбы с “проклятием размерности”.
- Другие методики обучения без учителя, включая кластеризацию, оценку плотности и обнаружение аномалий.

В части II рассматриваются следующие темы.

- Что собой представляют нейронные сети и для чего они пригодны.
- Построение и обучение нейронных сетей с применением TensorFlow и Keras.
- Самые важные архитектуры нейронных сетей: нейронные сети прямого распространения для табличных данных, сверточные сети для компьютерного зрения, рекуррентные сети и сети с долгой краткосрочной памятью (LSTM) для обработки последовательностей, кодировщики/декодировщики и преобразователи для обработки естественного языка, автокодировщики и порождающие состязательные сети для генеративного обучения.
- Методики обучения глубоких нейронных сетей.
- Построение агента (например, бота в игре), который способен узнавать хорошие стратегии методом проб и ошибок с использованием обучения с подкреплением.
- Эффективная загрузка и предварительная обработка крупных объемов данных.
- Широкомасштабное обучение и развертывание моделей TensorFlow.

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



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 применением 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.

## Изменения во втором издании

Второе издание книги преследует шесть основных целей.

1. Раскрытие дополнительных тем о машинном обучении: больше методик для обучения без учителя (в числе которых кластеризация, обнаружение аномалий, оценка плотности и модели со смесями рас-

пределений); больше методик для обучения глубоких сетей (включая само нормализующиеся сети); дополнительные методики компьютерного зрения (в том числе Xception, SENet, выявление объектов с помощью YOLO и семантическая сегментация с использованием R-CNN); обработка последовательностей с применением сверточных нейронных сетей (включая WaveNet); обработка естественного языка с использованием рекуррентных нейронных сетей, сверточных нейронных сетей и преобразователей; порождающие состязательные сети.

2. Раскрытие дополнительных библиотек и API-интерфейсов (Keras, Data API, TF-Agents для обучения с подкреплением), а также широкомасштабное обучение и развертывание моделей TF с применением API-интерфейса Distribution Strategies, TF-Serving и платформы AI Platform инфраструктуры Google Cloud. Вдобавок предлагается краткое введение в TF Transform, TFLite, TF Addons/Seq2Seq и TensorFlow.js.
3. Обсуждение ряда последних важных результатов, полученных при исследовании глубокого обучения.
4. Переход с TensorFlow на TensorFlow 2 и повсеместное использование реализации API-интерфейса Keras библиотеки TensorFlow (`t.f.keras`).
5. Обновление кода примеров для применения последних версий Scikit-Learn, NumPy, pandas, Matplotlib и других библиотек.
6. Прояснение некоторых разделов и исправление ошибок благодаря многочисленным отзывам читателей.

В книгу добавлены новые главы, некоторые главы были переписаны и несколько глав переупорядочено. Дополнительные сведения об изменениях во втором издании доступны по ссылке <https://homl.info/changes2>.

## Другие ресурсы

Для освоения машинного обучения доступно много превосходных ресурсов. Учебные курсы по машинному обучению Эндрю Йна на Coursera (<https://homl.info/ngcourse>) 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 (<https://homl.info/skdoc>). Вам также может понравиться веб-сайт Dataquest (<https://www.dataquest.io/>), который

предлагает очень полезные интерактивные руководства, и блоги по машинному обучению вроде перечисленных на веб-сайте Quora (<https://homl.info/1>). Наконец, веб-сайт по глубокому обучению (<http://deeplearning.net/>) содержит хороший список ресурсов для дальнейшего изучения.

Доступно множество других вводных книг по машинному обучению, включая перечисленные ниже.

- В книге Джоэла Груса *Data Science from Scratch* (O'Reilly) представлены основы машинного обучения и реализации ряда основных алгоритмов с помощью чистого кода на Python (с нуля).
- Книга Стивена Марсленда *Machine Learning: An Algorithmic Perspective* (Chapman and Hall) представляет собой замечательное введение в машинное обучение и раскрывает широкий спектр вопросов с примерами кода на Python (также с нуля, но с использованием NumPy).
- Книга Себастьяна Рашки *Python Machine Learning* (Packt Publishing; *Python и машинное обучение: машинное и глубокое обучение с использованием Python, Scikit-learn и TensorFlow*, пер. с англ., Диалектика) также является великолепным введением в машинное обучение и задействует библиотеки Python с открытым кодом (Pylearn 2 и Theano).
- Книга Франсуа Шолле *Deep Learning with Python* (Manning) ориентирована на практику и раскрывает большой диапазон тем в ясной и лаконичной манере, чего и можно было ожидать от автора великолепной библиотеки Keras. Предпочтение в ней отдается примерам кода, а не математической теории.
- Книга Андрея Буркова *The Hundred-Page Machine Learning Book* отличается краткостью и посвящена впечатляющему спектру тем, представляя их доступными терминами и не избегая математических уравнений.
- Книга Ясера С. Абу-Мустафы, Малика Магдона-Исмаила и Сюань-Тянь Линя *Learning from Data* (MLBook) демонстрирует довольно теоретический подход к машинному обучению, который обеспечивает глубокое понимание многих аспектов, в частности компромисса между смещением и дисперсией (см. главу 4).
- Прекрасная (и большая) книга Стюарта Рассела и Питера Норвига *Artificial Intelligence: A Modern Approach, 3rd Edition* (Pearson) охватывает невероятный объем тем, в числе которых и машинное обучение. Она помогает уловить общую картину машинного обучения.

Наконец, присоединение к веб-сайтам состязаний по машинному обучению, таким как Kaggle.com, 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, определяемыми контекстом.



Этот элемент содержит совет или указание.



Этот элемент содержит замечание общего характера.



Этот элемент содержит предупреждение или предостережение.

## Код примеров

Имеется множество тетрадей Jupyter, заполненных дополнительными материалами, такими как код примеров и упражнения, которые доступны для загрузки по адресу <https://github.com/ageron/handson-m12>.

Код ряда примеров в книге не содержит повторяющихся разделов или деталей, которые очевидны или не имеют отношения к машинному обучению. Такой подход позволяет сосредоточить внимание на важных частях кода и экономит пространство для раскрытия большего числа тем. Полный код всех примеров находится в тетрадях Jupyter.

Имейте в виду, что когда код примеров отображает вывод, то он показан с приглашениями на ввод Python (`>>>` и `...`), как в командной оболочке Python, чтобы отделить код от вывода. Скажем, в следующем коде определяется функция `square()`, после чего она вызывается и отображается результат возведения 3 в квадрат:

```
>>> def square(x):
...     return x ** 2
...
>>> result = square(3)
>>> result
9
```

Если код ничего не отображает, тогда приглашения на ввод не используются. Однако временами результат может быть показан в виде комментария, например:

```
def square(x):
    return x ** 2

result = square(3)      # значение result равно 9
```

## Использование кода примеров

Настоящая книга призвана помочь вам выполнять свою работу. Обычно, если в книге предлагается код примеров, 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 код примера раз-

решения не требует. Для встраивания значительного объема кода примеров, 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, Keras, and TensorFlow*, 2nd Edition, by Aurélien Géron (O’Reilly). Copyright 2019 Aurélien Géron, 978-1-492-03264-9”. Если вам кажется, что способ использования вами кода примеров выходит за законные рамки или не соответствует упомянутым выше разрешениям, тогда свяжитесь с нами по следующему адресу электронной почты: [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Об иллюстрации на обложке

Животное, изображенное на обложке книги — это огненная, или пятнистая, или обыкновенная саламандра (*Salamandra salamandra*), амфибия, встречающаяся в большей части Европы. Ее черная лоснящаяся кожа отличается большими желтыми пятнами на голове и спине, которые сигнализируют о наличии алкалоидных токсинов. Они являются вероятным источником общего названия этой амфибии: контакт с алкалоидными токсинами (которые она также умеет разбрзгивать на короткие расстояния) вызывает конвульсии и гипервентиляцию. Болезненные яды или влажность кожи саламандры (либо то и другое) привели к ошибочному убеждению, что эти существа не только способны выжить, будучи помещенными в огонь, но также могут потушить его.

Огненные саламандры живут в прохладных лесах, прячась в сырых щелинах и под упавшими стволами деревьев вблизи заводей или других пресноводных водоемов, которые облегчают их размножение. Хотя большую часть своей жизни огненные саламандры проводят на земле, они рождают своих детенышей в воде. Огненные саламандры питаются в основном насекомыми, пауками, слизнями и червями. Они могут вырастать до 30 см в длину, а в неволе способны жить до 50 лет.

Численность огненной саламандры была уменьшена по причине уничтожения их лесного ареала и отлова особей с целью продажи в качестве домашних животных, но самая большая угроза, с которой они сталкиваются — это восприимчивость их влагопроницаемой кожи к загрязнителям и микробам. С 2014 года из-за привнесенного грибка они исчезли в некоторых частях Нидерландов и Бельгии.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой уничтожения; все они важны для нашего мира. Чтобы узнать больше о том, чем вы можете помочь, посетите веб-сайт [animals.oreilly.com](http://animals.oreilly.com).

Изображение на обложке взято из книги *Wood's Illustrated Natural History* (Иллюстрированное естествознание Вуда).

## 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 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@gmail.com](mailto:info.dialektika@gmail.com)

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



Все иллюстрации к книге в цветном варианте доступны по адресу  
<http://go.dialektika.com/mlearning>

# Основы машинного обучения



# Введение в машинное обучение

Когда большинство людей слышат словосочетание “машинальное обучение”, они представляют себе робота: надежного дворецкого или неумолимого Терминатора в зависимости от того, кого вы спросите. Но машинное обучение (МО) — не только футуристическая фантазия, оно уже здесь. На самом деле МО десятилетиями существовало в ряде специализированных приложений, таких как программы для *оптического распознавания знаков* (*Optical Character Recognition* — *OCR*). Но первое приложение МО, которое получило действительно широкое распространение, улучшив жизнь сотен миллионов людей, увидело свет еще в 1990-х годах: *фильтр спама*. Он не претендует быть вездесущим Скайнетом, но формально квалифицируется как приложение с МО (фактически фильтр обучается настолько хорошо, что вам очень редко 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 оно 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 *учиться 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 эффективности  $P$ , если ее эффективность на  $T$ , измеренная посредством  $P$ , улучшается благодаря опыту  $E$ .

Том Митчелл, 1997 год

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 эффективнос-

ти  $P$  нуждается в определении; скажем, вы можете применять коэффициент корректно классифицированных почтовых сообщений. Такая конкретная оценка эффективности 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 какую-нибудь задачу лучше, чем ранее. Таким образом, загрузка копии Википедии — не машинное обучение.

## Для чего используют машинное обучение?

Обдумайте, каким образом вы бы реализовали фильтр спама с применением приемов традиционного программирования (рис. 1.1).

1. Сначала вы бы посмотрели, как 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 остальных частях сообщения.
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 для запуска.



Рис. 1.1. Традиционный подход

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

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 внутри примеров спам-сообщений в сравнении с примерами нормальных сообщений (рис. 1.2). Программа оказывается гораздо более короткой, легкой в сопровождении и вполне вероятно более точной.

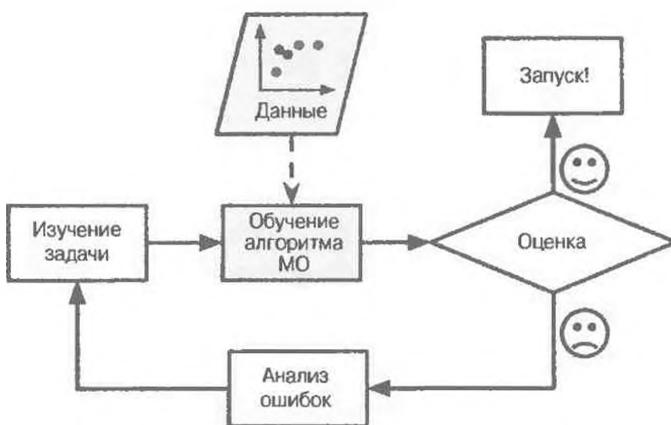


Рис. 1.2. Подход с машинным обучением

А если спамеры выяснят, что все их сообщения, которые содержат “4U”, блокируются? Взамен они могут начать писать “For U”. Фильтр спама, построенный с использованием приемов традиционного программирования, потребуется обновить, чтобы он маркировал сообщения с “For U”. Если спамеры продолжат 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.



Рис. 1.3. Автоматическая адаптация к изменениям

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

Наконец, машинное обучение способно помочь учиться людям (рис. 1.4).

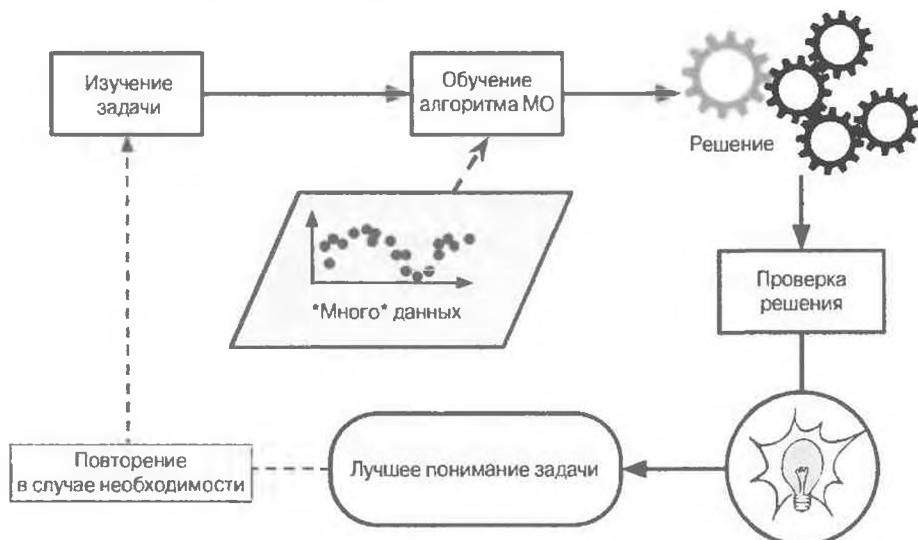


Рис. 1.4. Машинное обучение может помочь учиться людям

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, для которых существующие решения требуют большого объема тонкой настройки или длинных списков правил — один алгоритм МО часто способен упростить код и выполняться лучше, чем традиционный подход;
- сложных задач, для которых традиционный подход не дает пригодного решения — лучшие приемы МО вполне вероятно сумеют найти решение;
- изменяющихся сред — система МО способна адаптироваться к новым данным;
- получения сведений о сложных задачах и крупных объемах данных.

## Примеры приложений

Давайте посмотрим на ряд примеров задач МО наряду с методиками, которые могут справиться с ними.

*Анализ изображений товаров на производственной линии  
для их автоматической классификации*

Классификация изображений, обычно выполняемая с использованием сверточных нейронных сетей (см. главу 14).

*Обнаружение опухолей при сканировании мозга*

Семантическая сегментация, где каждый пиксель в изображении классифицируется (т.к. мы хотим определить точное местоположение и форму опухолей) обычно тоже с применением сверточных нейронных сетей.

## *Автоматическая классификация новых статей*

Обработка естественного языка и в особенности классификация текста, с которой можно справиться посредством рекуррентных нейронных сетей, сверточных нейронных сетей или преобразователей (см. главу 16).

## *Автоматическая пометка оскорбительных комментариев на дискуссионных форумах*

Также классификация текста, выполняемая с использованием инструментов для обработки естественного языка.

## *Автоматическое формирование резюме для длинных документов*

Ветвь обработки естественного языка, называемое реферирированием текста, которое действует те же самые инструменты.

## *Создание чатбота или личного секретаря*

Включает в себя многие компоненты обработки естественного языка, в том числе понимание естественного языка (*natural language understanding — NLU*) и модули ответов на вопросы.

## *Прогнозирование будущего годового дохода компании на основе множества показателей эффективности*

Задача регрессии (т.е. прогнозирования значений), которая может быть решена с применением любой регрессионной модели, такой как линейная или полиномиальная регрессионная модель (см. главу 4), регрессия методом опорных векторов (см. главу 5), регрессия на основе случайного леса (см. главу 7) или искусственная нейронная сеть (см. главу 10). Если желательно учесть последовательности прошлых показателей эффективности, тогда можно использовать рекуррентные нейронные сети, сверточные нейронные сети или преобразователи (см. главы 15 и 16).

## *Добавление к приложению реагирования на голосовые команды*

Распознавание речи, которое требует обработки аудиосэмплов: поскольку они являются длинными и сложными последовательностями, то обычно обрабатываются с применением рекуррентных нейронных сетей, сверточных нейронных сетей или преобразователей (см. главы 15 и 16).

## *Обнаружение мошенничества с кредитными картами*

Обнаружение аномалий (см. главу 9).

*Сегментирование клиентов на основе их покупок, чтобы для каждого сегмента можно было спланировать отличающуюся маркетинговую стратегию*

**Кластеризация** (см. главу 9).

*Представление сложного, многомерного набора данных на ясной диаграмме, отражающей суть*

**Визуализация** данных, часто вовлекающая методики понижения размерности (см. главу 8).

*Рекомендация товара, который может заинтересовать клиента, на основе прошлых покупок*

Рекомендательная система. Один из подходов заключается в том, чтобы передать прошлые покупки (и другую информацию о клиенте) искусственной нейронной сети (см. главу 10) и заставить ее выдать наиболее вероятную следующую покупку. Такая нейронная сеть обычно будет обучаться на последовательностях прошлых покупок для всех клиентов.

*Создание интеллектуального бота для игры*

Задача часто решается с использованием обучения с подкреплением (см. главу 18), являющегося ветвью МО, которая касается обучения агентов (подобных ботам) выбору действий, с течением времени доводящих их награды до максимума (например, бот может получать награду каждый раз, когда игрок теряет жизненные очки) внутри заданной среды (такой как игра). Знаменитая программа AlphaGo, победившая чемпиона мира в игре го, была создана с применением обучения с подкреплением.

Список можно было бы продолжать и продолжать, но надо надеяться, что он дал вам представление о невероятной широте и сложности задач, которые может решить МО, и разновидностях методик, используемых для каждой задачи.

## **Типы систем машинного обучения**

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

- 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 можно классифицировать согласно объему и типу контроля, которому 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* обучающий набор, поставляемый вами алгоритму, включает желательные решения, называемые *метками (label)*, как показано на рис. 1.5.

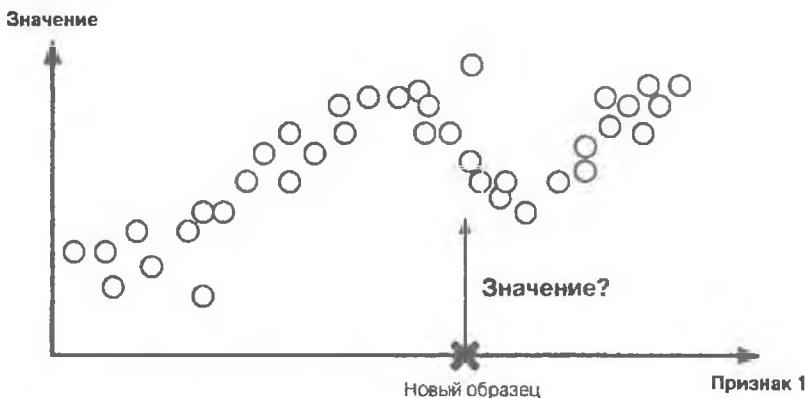
Типичной задачей обучения с учителем является *классификация*. Фильтр спама служит хорошим случаем классификации: он обучается на многих примерах сообщений с их *классом* (спам или не спам), а потому должен знать, каким образом классифицировать новые сообщения.

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)*.



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

Задачу подобного рода именуют *регрессией*<sup>1</sup> (рис. 1.6). Чтобы обучить систему, вам необходимо предоставить ей много примеров автомобилей, включая их прогнозаторы и метки (т.е. цены).



*Рис. 1.6. Задача регрессии: спрогнозировать значение, имея входной признак (обычно существует множество признаков и временами множество выходных значений)*



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* взаимозаменяя.

<sup>1</sup> 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.

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)<sup>2</sup>.

### Обучение без учителя

При обучении без учителя, как вы могли догадаться, обучающий набор не помечен (рис. 1.7). Система пытается обучаться без учителя.



Рис. 1.7. Непомеченный обучающий набор для обучения без учителя

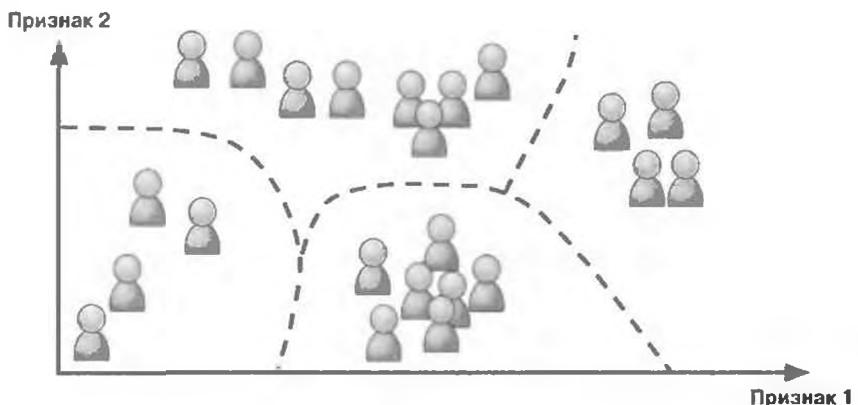
<sup>2</sup> Некоторые архитектуры нейронных сетей могут быть без учителя, такие как *автомоделировщики (autoencoder)* и *ограниченные машины Больцмана (restricted Boltzmann machine)*. Они также могут быть частичными, как в *глубоких сетях доверия (deep belief network)* и предварительном обучении без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

Далее приведен список наиболее важных алгоритмов обучения без учителя (большинство из них раскрывается в главах 8 и 9).

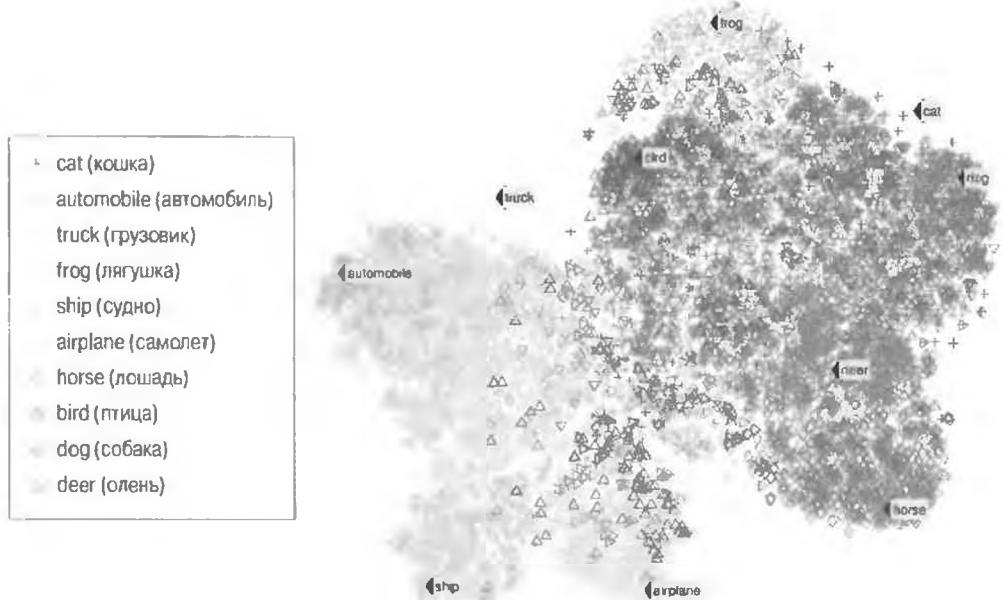
- Кластеризация:
  - K-Means (K-средние)
  - DBSCAN
  - иерархический кластерный анализ (Hierarchical Cluster Analysis — HCA)
- Обнаружение аномалий и обнаружение новизны:
  - одноклассовый SVM
  - изолирующий лес
- Визуализация и понижение размерности:
  - анализ главных компонентов (Principal Component Analysis — PCA)
  - ядерный анализ главных компонентов (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 по вечерам, тогда как 20% — юные любители научной фантастики, посещающие блог в выходные дни. 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.

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).



*Рис. 1.8. Кластеризация*



*Рис. 1.9. Пример визуализации t-SNE, выделяющей семантические кластеры<sup>3</sup>*

<sup>3</sup> Обратите внимание на то, что животные довольно хорошо отделены от транспортных средств, а лошади близки к оленям, но далеки от птиц. Рисунок разрешен для воспроизведения из работы Ричарда Сокера и других *Zero-Shot Learning Through Cross-Modal Transfer* (Обучение без подготовки посредством межмодальных перемещений), материалы 26-й Международной конференции по нейронным системам обработки информации 1 (2013 г.): с. 935–943

Такие алгоритмы стараются сохранить столько структуры, сколько могут (например, пытаются не допустить наложения при визуализации отдельных кластеров во входном пространстве), поэтому вы можете понять, каким образом данные организованы, и возможно идентифицировать непредвиденные паттерны.

Связанной задачей является *понижение размерности*, цель которого — упростить данные, не теряя чересчур много 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)*.



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 выполнять.

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. Во время обучения системе показываются в основном нормальные образцы, поэтому она учится распознавать их; когда затем система видит новый образец, она может сообщить, выглядит образец как нормальный или вероятно представляет собой аномалию (рис. 1.10).

Очень похожая задача — *обнаружение новизны (novelty detection)* — направлена на выявление новых образцов, которые отличаются от всех образцов в обучающем наборе. Она требует наличия очень “чистого” обучающего набора, свободного от любых образцов, которые хотелось бы, чтобы алгоритм распознавал. Например, если есть тысячи изображений собак и на 1% этих изображений представлены собаки породы чихуахуа, тогда алгоритм обнаружения новизны не должен трактовать новые изображения чихуахуа как новизну. С другой стороны, алгоритмы обнаружения аномалий могут счесть собак такой породы настолько редкими и отличающимися от собак

других пород, что вполне вероятно классифицируют их как аномалии (не в обиду чихуахуа).

Наконец, в число распространенных задач обучения без учителя 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 некоторого 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.



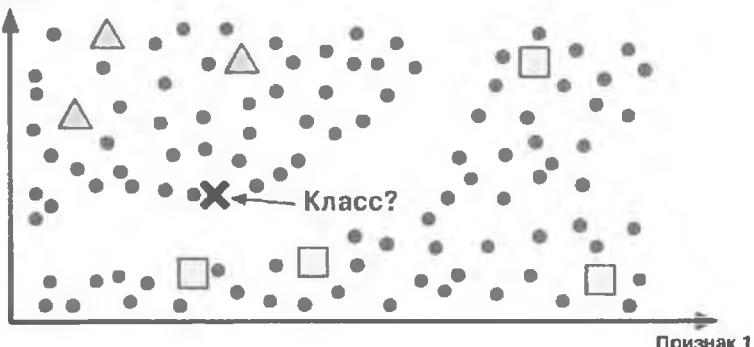
Рис. 1.10. Обнаружение аномалий

### Частичное обучение

Поскольку пометка данных обычно сопряжена с затратами ресурсов и времени, вы часто будете иметь дело с множеством непомеченных образцов и лишь небольшим количеством помеченных. Определенные алгоритмы способны работать с данными, которые помечены частично. Процесс называется *частичным обучением* (рис. 1.11).

Хорошими примерами частичного обучения 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).

Признак 2



Признак 1

**Рис. 1.11.** Частичное обучение с двумя классами (треугольники и квадраты): непомеченные образцы (кружочки) помогают отнести новый образец (крестик) к классу треугольников, а не к классу квадратов, хотя он расположен ближе к помеченным квадратам

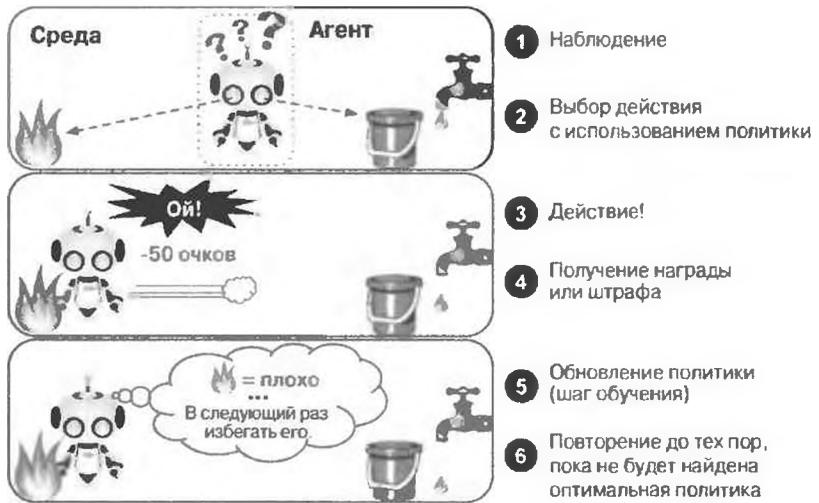
Теперь системе нужно узнать у вас, кто эти люди. Просто добавьте одну метку на особу<sup>4</sup>, и служба сумеет назвать всех на каждой фотографии, что полезно для поиска фотографий.

Большинство алгоритмов частичного обучения являются комбинациями алгоритмов обучения без учителя и с учителем. Например, глубокие сети доверия (*deep belief network — DBN*) основаны на компонентах обучения без учителя, называемых ограниченными машинами Больцмана (*restricted Boltzmann machine — 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 с течением времени получать наибольшую награду. Политика определяет, какое действие агент обязан выбирать, когда он находится в заданной ситуации.

<sup>4</sup> В ситуации, когда система работает безупречно. На практике она часто создает несколько кластеров на особу, а временами смешивает двух людей, выглядящих похожими, поэтому может возникнуть необходимость предоставить несколько меток на особу и вручную почистить некоторые кластеры.



*Рис. 1.12. Обучение с подкреплением*

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.

## 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, после чего помещается в производственную среду и функционирует без дальнейшего обучения; она просто применяет то, что узнала ранее. 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 сумеет 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.

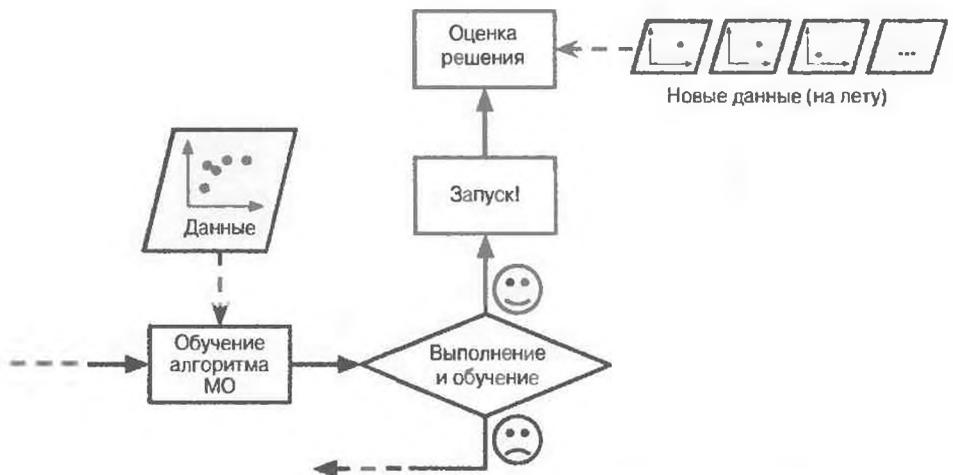
Кроме того, обучение на полном наборе данных требует много вычислительных ресурсов (процессорного времени, памяти, дискового пространства, дискового ввода-вывода, сетевого ввода-вывода и т.д.). При наличии большого объема данных и автоматизации системы для обучения с нуля ежедневно все закончится тем, 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 марсохода), тогда доставка крупных объемов обучающих данных и расходование многочисленных ресурсов для ежедневного обучения в течение нескольких 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).



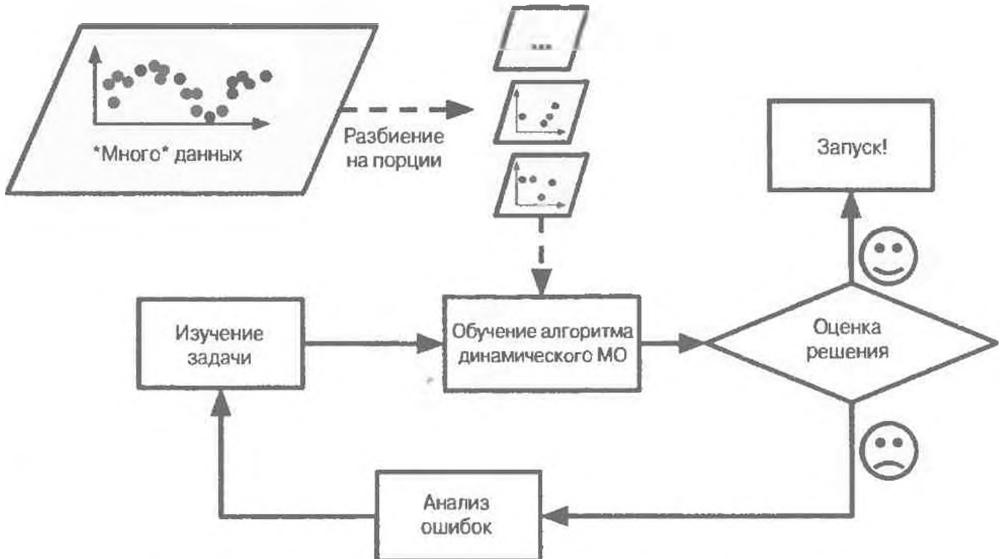
**Рис. 1.13.** При динамическом обучении модель обучается и помещается в производственную среду, после чего она продолжает обучаться по мере поступления новых данных

Динамическое обучение великолепно подходит для систем, которые получают данные в виде непрерывного потока (скажем, курсы акций) и должны адаптироваться к изменениям быстро или самостоятельно. Вдобавок динамическое обучение будет хорошим вариантом, когда вычислительные ресурсы ограничены: узнав о новых образцах данных, система динамического обучения в них больше не нуждается и потому может их отбросить (если только не нужна возможность отката к предыдущему состоянию и “повторное воспроизведение” данных). В итоге можно сберечь громадный объем пространства.

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 по всем данным (рис. 1.14).



Внешнее обучение обычно проводится 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)*.



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

Один 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 инерцией; т.е. она будет обучаться медленнее, но также окажется менее чувствительной к шуму в новых данных или к последовательностям нерепрезентативных точек данных (выбросам).

Крупная проблема с динамическим обучением связана с тем, что в случае передачи в систему неправильных данных ее эффективность будет понемногу снижаться. Если это действующая система, тогда ваши клиенты заметят такое снижение. Например, неправильные данные могли бы поступать из некорректно функционирующего датчика в работе 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 эффективности, сразу же отключать обучение (и возможно возвратиться в предыдущее рабочее состоя-

ние). Может также понадобиться отслеживать входные данные и реагировать на ненормальные 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 при заданном количестве обучающих примеров система должна быть в состоянии вырабатывать полезные прогнозы (обобщать) для примеров, которые она никогда не видела ранее. Наличие подходящего показателя эффективности 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 смог бы маркировать только сообщения, идентичные сообщениям, которые уже были маркированы пользователями — не худшее, но определенно и не лучшее решение.

Вместо того чтобы лишь маркировать 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 за счет измерения сходства, чтобы сравнить новые примеры с уже изученными примерами (или с их подмножеством). Скажем, на рис. 1.15 новый образец был бы классифицирован как треугольник, потому что данному классу принадлежит большинство самых похожих образцов.

Признак 2



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. Процесс называется *обучением на основе моделей* (рис. 1.16).

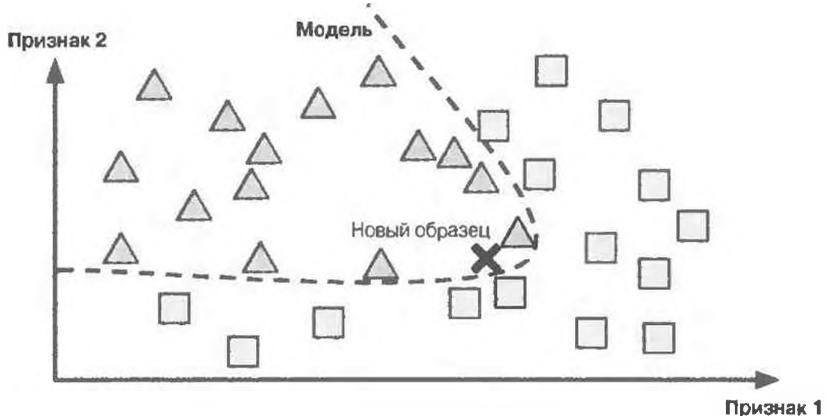


Рис. 1.16. Обучение на основе моделей

Предположим, что вас интересует, делают ли деньги людей счастливыми. Вы загружаете данные “Индекс лучшей жизни” из веб-сайта Организации экономического сотрудничества и развития (ОЭСР), <https://hom1.info/4>, а также статистические данные по валовому внутреннему продукту (ВВП) на душу населения из веб-сайта Международного валютного фонда (МВФ), <https://hom1.info/5>. 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 счастливее?

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 указанных в табл. 1.1 стран (рис. 1.17).

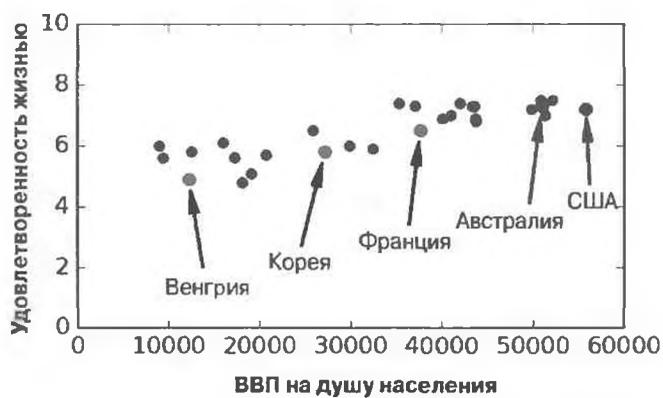


Рис. 1.17. Уловили здесь тенденцию?

Кажется, здесь видна 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. Таким образом, вы решаете моделировать удовлетворенность жизнью как линейную функцию от ВВП на душу населения. Этот шаг называется *выбором модели*: вы выбрали линейную модель удовлетворенности жизнью с только одним атрибутом — ВВП на душу населения (уравнение 1.1).

### Уравнение 1.1. Простая линейная модель

$$\text{удовлетворенность\_жизнью} = \theta_0 + \theta_1 \times \text{ВВП\_на\_душу\_населения}$$

Модель имеет два параметра модели,  $\theta_0$  и  $\theta_1$ .<sup>5</sup> Подстраивая эти параметры, вы можете заставить свою модель представлять любую линейную функцию, как видно на рис. 1.18.

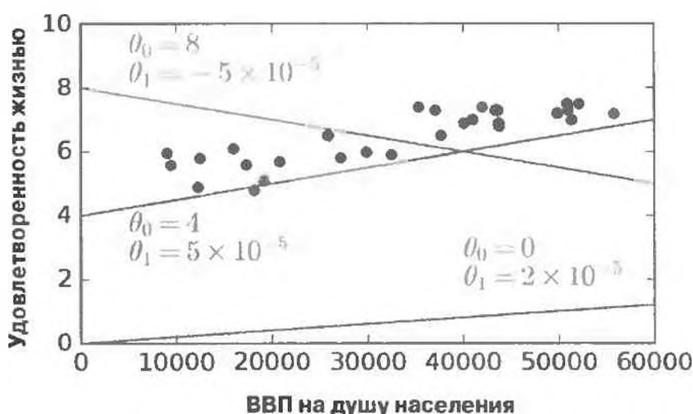


Рис. 1.18. Несколько возможных линейных моделей

Прежде чем модель можно будет использовать, вы должны определить значения параметров  $\theta_0$  и  $\theta_1$ . Как вы узнаете значения, которые обеспечат лучшее выполнение модели? Чтобы ответить на данный вопрос, понадобится указать измерение эффективности. Вы можете определить либо функцию полезности (*utility function*) (или функцию приспособленности (*fitness function*)), которая измеряет, насколько ваша модель хороша, 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 представляете 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  $\theta_1 = 4.91 \times 10^{-5}$ .

<sup>5</sup> По соглашению для представления параметров модели часто применяется греческая буква  $\theta$  (тета).



Сбивает с толку то, что слово “модель” может относиться к *типу модели* (например, линейная регрессия), к *полностью определенной архитектуре модели* (скажем, линейная регрессия с одним входом и одним выходом) или к *финальной обученной модели*, готовой к использованию для выработки прогнозов (например, линейная регрессия с одним входом и одним выходом, применяющая  $\theta_0 = 4.85$  и  $\theta_1 = 4.91 \times 10^{-5}$ ). Подбор модели заключается в выборе типа модели и полном определении ее архитектуры. Обучение модели означает выполнение алгоритма для нахождения параметров модели, которые сделают ее лучше подогнанной к обучающим данным (и надо надеяться вырабатывающей полезные прогнозы на новых данных).

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

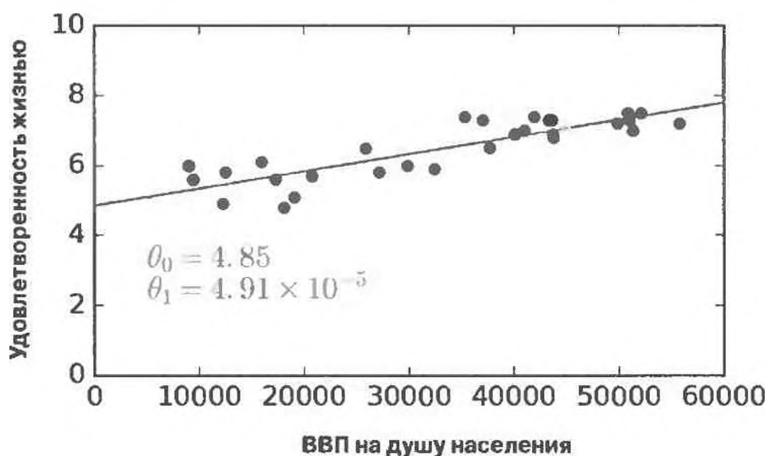


Рис. 1.19. Линейная модель, которая лучше подогнана к обучающим данным

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 их<sup>6</sup>, создает диаграмму рассеяния для визуализации, затем обучает линейную модель и вырабатывает прогноз<sup>7</sup>.

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

```
import          as
import          as
import          as
import

# 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]]
```

<sup>6</sup> Определение функции `prepare_country_stats()` здесь не показано (если вас интересуют детали, тогда просмотрите тетрадь Jupyter для настоящей главы). Она всего лишь содержит скучный код pandas, который соединяет данные ОЭСР об удовлетворенности жизнью с данными МВФ о ВВП на душу населения.

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



Если бы взамен вы применяли 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 рассматриваемом примере  $k = 3$ ).

Замена модели линейной регрессии моделью с регрессией  $k$  ближайших соседей в предыдущем коде сводится к замене приведенных ниже двух строк:

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

следующими строками:

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

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

Подведем итоги:

- вы исследовали данные;
- вы выбрали модель;
- вы обучили модель на обучающих данных (т.е. алгоритм обучения искал для параметров модели значения, которые доводят до минимума функцию издерек);
- 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 многое 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 ребенок узнал, 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. Он гений.

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 (<https://hml.info/6>), опубликованной в 2001 г., исследователи из Microsoft Мишель Банко и Эрик Брилл показали, что очень разные алгоритмы МО, включая довольно простые, выполняются почти одинаково хорошо на сложной задаче устранения неоднозначности в естественном языке<sup>8</sup>, когда им было предоставлено достаточно данных (рис. 1.20).

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

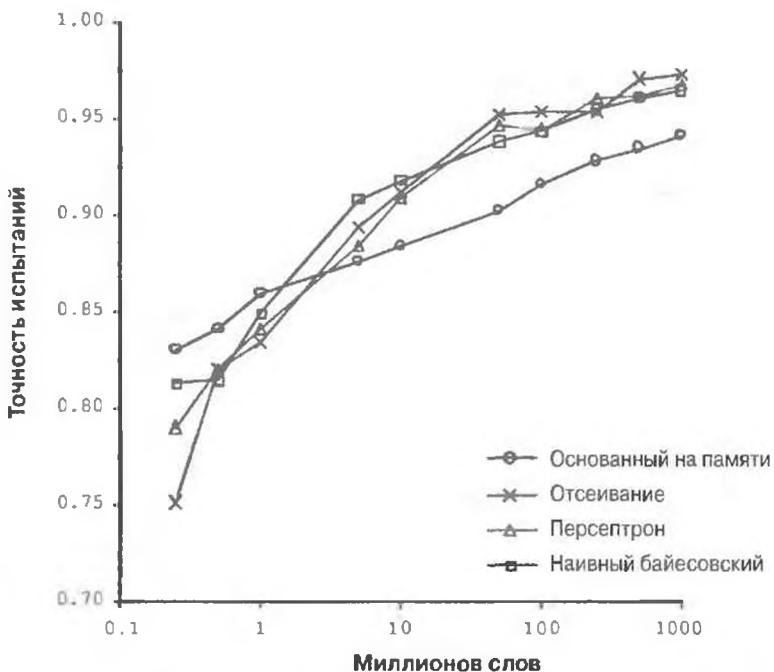


Рис. 1.20. Важность данных в сравнении с алгоритмами<sup>9</sup>

Как отметили авторы, “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 корпуса (текста)”.

Идея о том, что для сложных задач данные более существенны, чем алгоритмы, была в дальнейшем популяризована Питером Норвигом и другими в статье под названием *The Unreasonable Effectiveness of Data* (Необоснованная эффективность данных), <https://homl.info/7>, опубликованной в 2009 году<sup>10</sup>. 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.

<sup>9</sup> Рисунок разрешен для воспроизведения из работы Мишель Банко и Эрика Брилла *Scaling to Very Very Large Corpora for Natural Language Disambiguation* (Масштабирование до очень и очень крупных корпусов для устранения неоднозначности в естественном языке), материалы 39-го ежегодного собрания ассоциации вычислительной лингвистики (2001 г.): с. 26–33.

<sup>10</sup> Питер Норвиг и другие, *The Unreasonable Effectiveness of Data* (Необоснованная эффективность данных), *IEEE Intelligent Systems* 24, номер 2 (2009 г.): с. 8–12.

## 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 на основе образцов и на основе моделей.

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

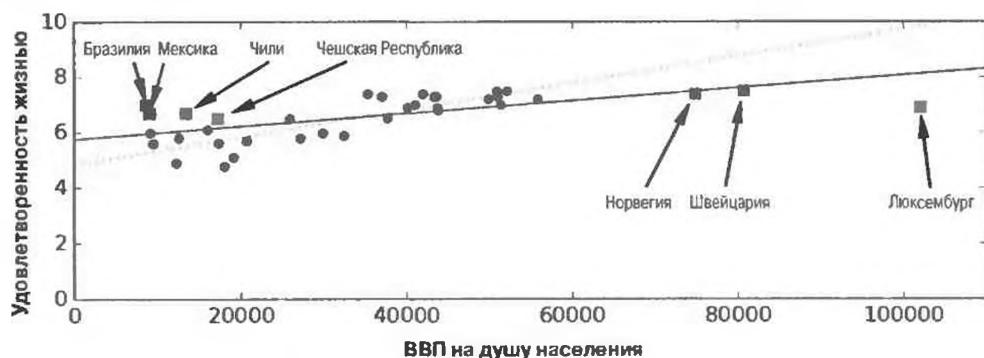


Рис. 1.21. Более репрезентативный обучающий набор

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 простая линейная модель подобного рода 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*).

## Примеры смещения выборки

Пожалуй, самый знаменитый пример смещения выборки случился во время президентских выборов в США в 1936 г., 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, списки 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 потенциального исключения людей, не особо интересующихся политикой, людей, которым не нравится журнал *Literary Digest*, и других ключевых групп. 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 смещеными в пользу популярных исполнителей (а если вы живете в Бразилии, то получите много видеоклипов в стиле фанк-カリока, 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 именно на это уходит значительная часть времени у большинства специалистов по работе с данными. Ниже описана пара ситуаций, когда обучающие данные желательно очистить.

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

## 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 слишком 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 следующие шаги.

- *Выбор признаков (feature selection)* — выбор среди существующих признаков наиболее полезных для обучения.
- *Выделение признаков (feature extraction)* — объединение 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 обучающими данными

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)* и 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, которая чрезмерно переобучается обучающими данными. Хотя она гораздо лучше выполняется на обучающих данных, чем простая линейная модель, действительно ли вы будете доверять ее прогнозам?

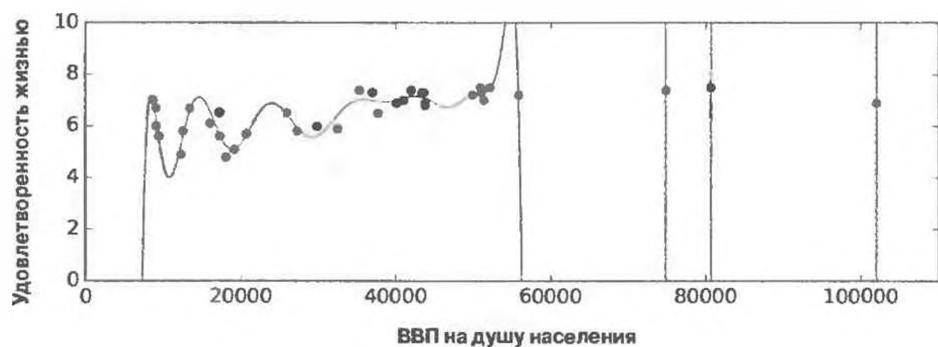


Рис. 1.22. Переобучение обучающих данных

Сложные модели, такие как глубокие нейронные сети, могут обнаруживать едва различимые паттерны в данных, но если обучающий набор зашумлен 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 он просто является результатом шума в данных.



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

- Упрощение модели за счет выбора варианта с меньшим числом параметров (например, линейной модели вместо полиномиальной модели высокого порядка), сокращения количества атрибутов в обучающих данных или ограничения модели.
- Накопление большего объема обучающих данных.
- Понижение шума в обучающих данных (например, исправление ошибок данных и устранение выбросов).

Ограничение модели с целью ее упрощения и снижения риска переобучения называется *регуляризацией* (*regularization*). Например, линейная модель, которую мы определили ранее, имеет два параметра,  $\theta_0$  и  $\theta_1$ . Это дает алгоритму обучения две степени свободы для адаптации модели к обучающим данным: он может подстраивать и высоту линии ( $\theta_0$ ), и ее наклон ( $\theta_1$ ). Если мы принудительно установим  $\theta_1$  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  $\theta_1$ , но вынудим удерживать его небольшим, тогда алгоритм обучения фактически будет находиться где-то между одной и двумя степенями свободы. Он породит модель, которая проще модели с двумя степенями свободы, но сложнее модели с только одной степенью свободы. Вы хотите добиться правильного баланса между идеальной подгонкой к обучающим 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 на странах, обозначенных кружочками (без стран, обозначенных квадратиками), пунктирная линия — вторую модель,

обучающуюся со всеми странами (обозначенными кружочками и квадратиками), а сплошная линия — модель, которая обучалась на тех же данных, что и первая модель, но с ограничением регуляризации. Как видите, регуляризация вынудила модель иметь меньший наклон: эта модель подгоняется к обучающим данным (кружочкам) не настолько хорошо, как первая, но фактически лучше обобщается на новые примеры, которые она не видела во время обучения (квадратики).

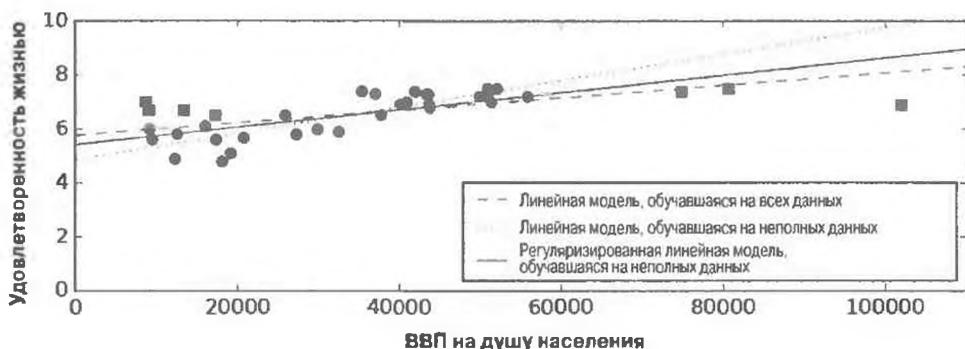


Рис. 1.23. Регуляризация снижает риск переобучения

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

## Недообучение обучающими данными

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 алгоритму обучения лучших признаков (конструирование признаков).
- Уменьшение ограничений на модели (например, снижение величины гиперпараметра регуляризации).

## Шаг назад

К настоящему времени вы много чего знаете о машинном обучении. Тем не менее, мы рассмотрели настолько много концепций, что вы можете испытывать легкую растерянность, поэтому давайте сделаем шаг назад и взглянем 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 множество разных типов систем машинного обучения: с учителем или без, пакетное или динамическое, на основе образцов или на основе моделей.
- В проекте МО вы накапливаете данные в обучающем наборе и представляете его алгоритму обучения. Если алгоритм основан на моделях, тогда он настраивает ряд параметров, чтобы подогнать модель к обучающему набору (т.е., чтобы вырабатывать пригодные прогнозы на самом обучающем наборе), и затем есть надежда, что она будет способна вырабатывать полезные прогнозы также на новых образцах. Если алгоритм основан на образцах, то он просто заучивает образцы на память и обобщается на новые образцы с использованием измерения сходства для их сравнения с изученными образцами.
- 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 это делать.

# Испытание и проверка

Единственный способ узнать, насколько хорошо модель будет обобщаться на новые образцы, предусматривает ее действительное испытание на новых образцах. Один из способов — поместить модель в производственную среду и понаблюдать за ее функционированием. Вполне нормальный прием, но если ваша модель крайне плоха, то пользователи выразят недовольство, что вряд ли можно считать самым лучшим планом.

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. Результатирующее 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 переобучается обучающими данными.



Принято применять 80% данных для обучения и удерживать 20% данных для испытания. Однако все зависит от размера набора данных: если он содержит 10 миллионов образцов, тогда удержание даже 1% означает, что испытательный набор будет включать 100 000 образцов, чего вероятно более чем достаточно для получения хорошей оценки ошибки обобщения.

## Настройка гиперпараметра и подбор модели

Оценивание модели делается достаточно просто: нужно лишь воспользоваться испытательным набором. Но предположим, что вы колебитесь в выборе между двумя типами моделей (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 — например, всего 5%. Затем вы помещаете такую модель в производственную среду, 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 этого конкретного набора. Таким образом, вряд ли модель будет функционировать настолько же хорошо на новых данных.

Общее решение проблемы называется *проверкой с удерживанием* (*holdout validation*): вы просто удерживаете часть обучающего набора для оценки нескольких моделей-кандидатов и выбираете из них лучшую. Новый удерживаемый набор называется *проверочным набором* (*validation set*) либо иногда *развивающим набором* (*development set*). Точнее говоря, вы обучаете множество моделей с различными гиперпараметрами на сокращенном обучающем наборе (т.е. полный обучающий набор минус проверочный набор) и выбираете модель, которая работает лучше всех на проверочном наборе. После такого процесса проверки с удерживанием вы обучаете лучшую модель на полном обучающем наборе (включая проверочный набор), что дает вам финальную модель. В заключение вы оцениваете финальную модель на испытательном наборе, чтобы получить оценку ошибки обобщения.

Описанное решение обычно работает довольно хорошо. Тем не менее, если проверочный набор слишком мал, тогда оценки моделей будут неточными: в результате по ошибке вы можете выбрать квазиоптимальную модель. И наоборот, если проверочный набор чересчур велик, то остающийся обучающий набор окажется намного меньше полного обучающего набора. Почему это плохо? Поскольку финальная модель будет обучаться на полном обучающем наборе, совсем не идеально сравнивать модели-кандидаты, обученные на гораздо меньшем обучающем наборе. Ситуация похожа на привлечение принтера для участия в марафоне. Один из способов решения проблемы предусматривает многократное проведение *перекрестной проверки* (*cross-validation*) с применением множества небольших проверочных наборов. После обучения на остальных данных каждая модель оценивается один раз на каждом проверочном наборе. Усредняя все оценки модели, вы получаете гораздо более точное измерение ее эффективности. Однако есть и отрицательная сторона: время обучения умножается на количество проверочных наборов.

## Несоответствие данных

В ряде случаев легко добыть крупный объем данных для обучения, но эти данные возможно не в полной мере характеризуют данные, которые будут использоваться в производственной среде. Например, предположим, что вы хотите создать мобильное приложение для фотографирования цветов и автоматического определения их видов. Совершенно несложно загрузить миллионы фотографий цветов из веб-сети, но они неидеально представляли бы те фотографии, которые будут получаться приложением на мобильном устройстве. Пусть вы имеете только 10 000 репрезентативных фотографий (т.е. фактически полученных с помощью приложения). В таком случае самое важное правило заключается в том, что проверочный и испытательный наборы обязаны быть насколько возможно репрезентативными в отношении данных, которые вы ожидаете применять в производственной среде. Поэтому наборы должны состоять исключительно из репрезентативных фотографий: вы можете перетасовать их и поместить одну половину в проверочный набор, а другую — в испытательный набор (удостоверившись в отсутствии дубликатов или почти дубликатов в обоих наборах). Но если после обучения модели на фотографиях из веб-сети вы обнаружите, что ее эффективность на проверочном наборе неутешительна, то не будете знать, по какой причине: переобучение модели обучающим набором или просто несоответствие между фотографиями из веб-сети и фотографиями, полученными мобильным приложением. Одно из решений предусматривает удержание части обучающих фотографий (загруженных из веб-сети) в еще одном наборе, который Эндрю Ын называет *обучающе-развивающим набором* (*train-dev set*). После того, как модель обучена (на обучающем, а не на обучающе-развивающем наборе), вы можете оценить ее на обучающе-развивающем наборе. Если она работает хорошо, тогда модель не переобучается обучающим набором. Если она функционирует плохо на проверочном наборе, то проблема должна быть связана с несоответствием данных. Вы можете попытаться решить проблему, предварительно обработав изображения из веб-сети, чтобы сделать их больше похожими на фотографии, которые будут получаться мобильным приложением, и затем заново обучив модель. И наоборот, если модель функционирует плохо на обучающе-развивающем наборе, тогда она обязана переобучаться обучающим набором, поэтому вы должны попробовать упростить или регуляризовать модель, получить больше обучающих данных и очистить обучающие данные.

## Теорема об отсутствии бесплатных завтраков

Модель является упрощенной версией наблюдений. Упрощения означают отбрасывание избыточных деталей, которые вряд ли обобщаются на новые образцы. 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 года (<https://homl.info/8>)<sup>11</sup> Дэвид Вольперт продемонстрировал, что если вы не делаете абсолютно никаких предположений о данных, тогда нет оснований отдавать предпочтение одной модели перед любой другой. Это называется теоремой *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 *a priori* 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.

---

<sup>11</sup> Дэвид Вольперт, *The Lack of A Priori Distinctions Between Learning Algorithms* (Отсутствие априорных различий между алгоритмами обучения), *Neural Computation* 8, номер 7 (1996 г.): с. 1341–1390.

# 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 алгоритма обучения при выработке прогнозов полагается на измерение сходства?
12. В чем разница между параметром модели и гиперпараметром алгоритма обучения?
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 такое обучающе-развивающий набор, когда он может понадобиться и как его использовать?
19. Что может пойти не так при настройке гиперпараметров с использованием испытательного набора?

Решения приведенных упражнений доступны в приложении А.

# Полный проект машинного обучения

В этой главе вы проработаете пример проекта от начала до конца, разыгрывая из себя недавно нанятого специалиста по работе с данными в компании, которая занимается недвижимостью<sup>1</sup>. 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 МО.
5. Выбор модели и ее обучение.
6. Точная настройка модели.
7. Представление своего решения.
8. Запуск, наблюдение и сопровождение системы.

## Работа с реальными данными

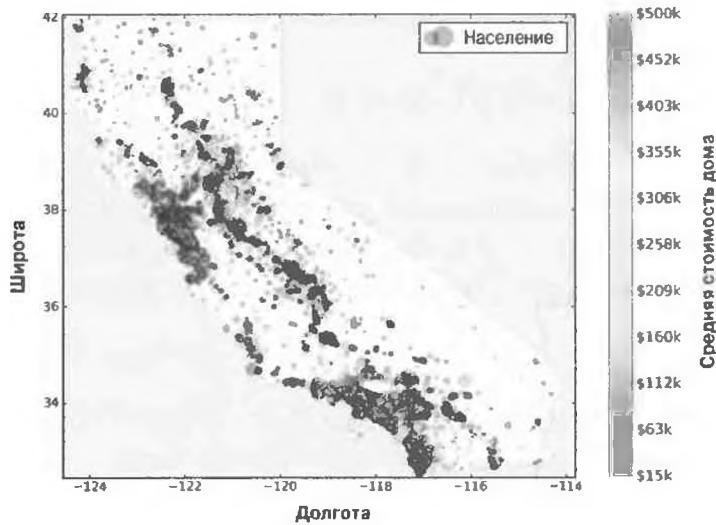
При освоении МО лучше экспериментировать с реально существующими данными, а не с искусственными наборами данных. К счастью, на выбор доступны тысячи открытых баз данных, охватывающих все виды предметных областей. Ниже перечислено несколько мест, куда вы можете заглянуть, чтобы получить данные.

- Популярные открытые хранилища данных:
  - хранилище для машинного обучения Калифорнийского университета в Ирвайне (UC Irvine Machine Learning Repository; <http://archive.ics.uci.edu/ml/>)
  - наборы данных Kaggle (<https://www.kaggle.com/datasets>)
  - наборы данных AWS в Amazon (<https://registry.opendata.aws/>)

<sup>1</sup> Пример проекта является выдумкой; цель в том, чтобы продемонстрировать основные шаги проекта МО, а не узнать что-то о сфере работы с недвижимостью.

- Метапорталы (содержат списки открытых хранилищ данных):
  - Data Portals (<http://dataportals.org/>)
  - OpenDataMonitor (<http://opendatamonitor.eu/>)
  - Quandl (<http://quandl.com/>)
- Другие страницы со списками многих популярных открытых хранилищ данных:
  - список наборов данных для машинного обучения в англоязычной Википедии (<https://homl.info/9>)
  - Quora.com (<https://homl.info/10>)
  - сабреддит Datasets (<https://www.reddit.com/r/datasets>)

В настоящей главе мы будем применять набор данных California Housing Prices (цены на жилье в Калифорнии) из хранилища StatLib<sup>2</sup> (рис. 2.1). Он основан на данных переписи населения Калифорнии 1990 года. Набор не особенно новый (на то время хороший дом в районе залива все еще был доступен по средствам), но обладает многими качествами для обучения, потому мы притворимся, что в нем содержатся последние данные. В целях обучения я добавил категориальный атрибут и удалил несколько признаков.



*Рис. 2.1. Цены на жилье в Калифорнии*

<sup>2</sup> Первоначально этот набор данных появился в статье Р. Келли Пейса и Рональда Барри *Sparse Spatial Autoregressions* (Разреженные пространственные авторегрессии), *Statistics & Probability Letters* 33, номер 3 (1997 г.): с. 291–297.

# Выяснение общей картины

Добро пожаловать в Корпорацию по жилью, вооруженную машинным обучением! Ваша первая задача — воспользоваться данными переписи населения Калифорнии для построения модели цен на жилье в этом штате. Данные переписи включают метрики, такие как население, медианный доход и средняя стоимость дома, 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 “округами”. Ваша модель должна обучаться на таких данных и быть способной прогнозировать среднюю стоимость дома в любом округе, располагая всеми остальными метриками.



Как хорошо организованный специалист по работе с данными первое, что вы должны сделать — достать 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. В текущей главе мы пройдемся по многим пунктам контрольного перечня, но также пропустим несколько из них 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-задание. Возможно, построение модели не является конечной целью. Каким образом компания рассчитывает применять и извлекать пользу от модели? Знание задания важно, поскольку оно 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 еще одну систему МО (рис. 2.2) наряду со многими другими сигналами<sup>3</sup>.

<sup>3</sup> Порцию информации, передаваемой в систему МО, часто называют сигналом, ссылаясь на теорию информации Клода Шеннона, которую он разработал в Bell Labs для улучшения средств телекоммуникации. Согласно его теории вам нужно высокое соотношение сигнал/шум.

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. Затем некоторое время спустя следующий компонент в конвейере захватывает эти данные и выдает собственный вывод.

Каждый компонент довольно самодостаточен: в качестве интерфейса между компонентами выступает всего лишь хранилище данных. В итоге система становится простой для понимания (с помощью графа потока данных), а 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 последние выходные данные из неисправного компонента. Такой подход делает архитектуру вполне надежной.

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

Следующим вы задаете шефу вопрос о том, на что похоже текущее решение (если оно есть). Текущая ситуация часто даст опорный показатель эффективности, а также догадки относительно того, как решать задачу. Ваш шеф отвечает, что в настоящий момент приблизительные цены в округе подсчитываются экспертами вручную: команда собирает новейшую информацию об округе, а когда получить среднюю стоимость дома не удается, тогда она оценивается с применением сложных правил.

Такое решение сопряжено с высокими затратами, отнимает много времени и его оценки далеко не замечательны. В случаях, когда экспертам удается выяснить действительную среднюю стоимость дома, они часто обнаруживают, что их оценки расходятся более чем на 20%. 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.

Собрав всю упомянутую информацию, 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 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: 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). Она также является типовой задачей регрессии, поскольку вам предложено спрогнозировать значение. Точнее говоря, это задача *составной регрессии (multiple regression)*, потому что для выработки прогноза система будет применять множество признаков (население округа, медианный доход и т.д.). Вдобавок она представляет собой задачу *одномерной регрессии (univariate regression)*, т.к. необходимо лишь попробовать спрогнозировать единственное значение для каждого округа. Если бы нужно было прогнозировать несколько значений для каждого округа, то мы имели бы дело с задачей *многомерной регрессии (multivariate regression)*.

Наконец, отсутствует непрерывный поток данных, поступающих в систему, нет потребности в быстром приспособлении к меняющимся данным, а объем самих данных достаточно мал, чтобы они умещались в памяти. Таким образом, должно подойти простое пакетное обучение.



Если бы данные были гигантскими, тогда вы могли бы разнести работу пакетного обучения на множество серверов (с применением технологии MapReduce) либо использовать методику 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). Он 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 2000 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  $-118.29^\circ$  и широтой  $33.91^\circ$ , имеет 1416 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

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

- $X$  — матрица, содержащая все значения признаков (исключая метки) всех образцов в наборе данных. Предусмотрена одна строка на образец, а  $i$ -тая строка эквивалентна транспонированию<sup>4</sup>  $x^{(i)}$ , что обозначается как  $(x^{(i)})^T$ .

Например, если первый округ оказывается таким, как только что описанный, тогда матрица  $X$  выглядит следующим образом:

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

- $h$  — функция прогнозирования системы, также называемая *гипотезой* (*hypothesis*). Когда системе предоставляется вектор признаков образца  $x^{(i)}$ , она выводит для этого образца прогнозируемое значение  $\hat{y}^{(i)} = h(x^{(i)})$ .

Например, если система прогнозирует, что средняя стоимость дома в первом округе равна \$158 400, тогда  $\hat{y}^{(1)} = h(x^{(1)}) = 158\,400$ . Ошибка прогноза для этого округа составляет  $\hat{y}^{(1)} - y^{(1)} = 2\,000$ .

- $RMSE(X, h)$  — функция издержек, измеренная на наборе образцов с использованием гипотезы  $h$ .

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

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

Несмотря на то что 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* (*Euclidean norm*): это понятие расстояния, с которым вы знакомы. Она также называется *нормой*  $\ell_2$  и обозначается как  $\|\cdot\|_2$  (или просто  $\|\cdot\|$ ).
- Вычисление суммы абсолютных величин (MAE) соответствует *норме*  $\ell_1$  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*  $\ell_k$  вектора  $\mathbf{v}$ , содержащего  $n$  элементов, определяется как  $\|\mathbf{v}\|_k = (\lvert v_0 \rvert^k + \lvert v_1 \rvert^k + \dots + \lvert v_n \rvert^k)^{\frac{1}{k}}$ .  $\ell_0$  просто дает количество ненулевых элементов в векторе, а  $\ell_\infty$  — максимальную абсолютную величину в векторе.
- Чем выше индекс нормы, тем больше она концентрируется на крупных значениях и пренебрегает мелкими значениями. Вот почему RMSE чувствительнее к выбросам, чем MAE. Но когда выбросы экспоненциально редкие (как в колоколообразной кривой), то погрешность 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), что может помочь 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 как быть, если нижерасположенная система 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 приступить 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-m12>.

### 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/><sup>5</sup>.

---

<sup>5</sup> Рекомендуется использовать самую последнюю версию Python 3. Возможно, версия Python 2.7+ тоже будет работать, но в настоящее время она устарела, и все основные библиотеки для научных расчетов прекратили ее поддерживать, поэтому вы должны как можно быстрее перейти на Python 3.

Далее понадобится создать каталог рабочей области для кода и наборов данных МО. Откройте окно терминала 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 расчеты 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)<sup>6</sup>. Проверить, установлен ли модуль pip, можно посредством следующей команды:

```
$ python3 -m pip --version  
pip 19.0.2 from [...]/lib/python3.6/site-packages (python 3.6)
```

Вы обязаны удостовериться в том, что установлена самая последняя версия pip. Для обновления модуля pip введите такую команду (точная версия может отличаться)<sup>7</sup>:

```
$ python3 -m pip install --user -U pip  
Collecting pip  
[...]  
Successfully installed pip-19.0.2
```

<sup>6</sup> 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.

<sup>7</sup> Если вы хотите обновить pip для всех пользователей системы, а не только для себя, тогда не указывайте параметр --user и удостоверьтесь в наличии прав администратора (например, добавив sudo перед командой в среде Linux или macOS).

## 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), тогда установите инструмент `virtualenv`<sup>8</sup>, запустив следующую команду `pip` (чтобы установить `virtualenv` для всех пользователей системы, не указывайте `--user` и запускайте команду с правами администратора):

```
$ python3 -m pip install --user -U virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Теперь можно создать изолированную среду Python:

```
$ cd $ML_PATH
$ virtualenv my_env
Using base prefix '[...]'
New python executable in [...]/ml/my_env/bin/python3.6
Also creating executable in [...]/ml/my_env/bin/python
Installing setuptools, pip, wheel...done.
```

Затем каждый раз, когда нужно активировать эту среду, вы должны открыть окно терминала и ввести следующие команды:

```
$ cd $ML_PATH
$ source my_env/bin/activate      # в Linux или macOS
$ .\my_env\Scripts\activate      # в Windows
```

Чтобы деактивировать среду `virtualenv`, введите `deactivate`. 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`.

<sup>8</sup> Альтернативные инструменты включают `venv` (очень похож на `virtualenv` и входит в состав стандартной библиотеки), `virtualenvwrapper` (предоставляет дополнительную функциональность поверх `virtualenv`), `ruenv` (позволяет легко переключаться между версиями Python) и `ripenv` (великолепный инструмент для работы с пакетами от создателя популярной библиотеки `requests`, построенный поверх `pip` и `virtualenv`).

Теперь вы можете установить все обязательные модули и их зависимости с применением следующей простой команды `pip` (если вы не используете `virtualenv`, то должны указать параметр `--user` и иметь права администратора):

```
$ python3 -m pip install  
    -U 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 соблюдались предшествующие инструкции для `virtualenv`).

Создайте новую тетрадь Python, щелкнув на кнопке `New (Новая)` и выбрав подходящую версию Python<sup>9</sup> (рис. 2.3). Такое действие приводит к созданию в рабочей области нового файла тетради по имени `Untitled.ipynb`, запуску ядра Jupyter Python для выполнения этой тетради и открытию тетради в новой вкладке. Вы должны начать с переименования тетради в `Housing` (файл автоматически переименуется в `Housing.ipynb`), щелкнув на `Untitled` и введя новое имя.

<sup>9</sup> Обратите внимание, что Jupyter способен поддерживать несколько версий Python и даже многие другие языки наподобие R или Octave.



Рис. 2.3. Ваша рабочая область в Jupyter

Тетрадь состоит из списка ячеек. Каждая ячейка может содержать 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 оправляется ядру Python этой тетради, которое выполнит ее и возвратит вывод. Результат отображается ниже ячейки, а поскольку достигнут конец тетради, то автоматически создается новая ячейка. Чтобы изучить основы, выберите пункт User Interface Tour (Тур по пользовательскому интерфейсу) в меню Help (Справка) инструмента Jupyter.

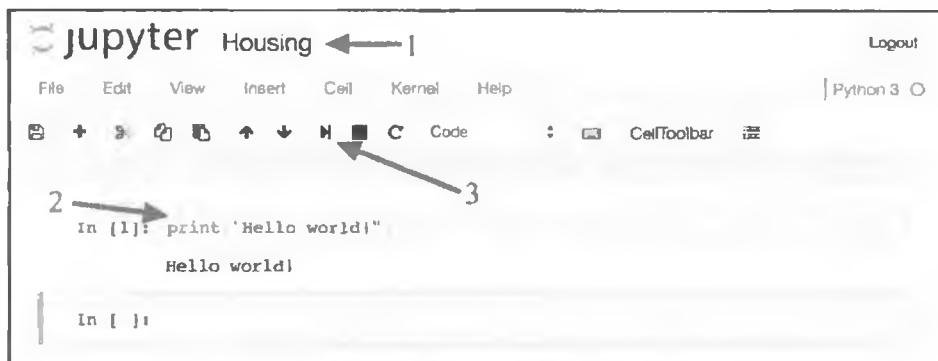


Рис. 2.4. Простейшая тетрадь Python

## Загрузка данных

В типовых средах ваши данные будут находиться в реляционной базе данных (или в каком-то другом общем хранилище данных) и распределяться по множеству таблиц/документов/файлов. Для работы с ними вам сначала понадобится получить учетные данные и авторизацию доступа<sup>10</sup>, 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.

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. Наличие функции, загружающей данные, особенно удобно, если данные периодически изменяются. Вы можете написать небольшой сценарий, который применяет такую функцию для извлечения самых последних данных (или настроить запланированное задание для автоматического запуска на регулярной основе). Автоматизация процесса извлечения данных также удобна, когда набор данных необходимо устанавливать на множество компьютеров. Ниже показана функция для извлечения данных<sup>11</sup>.

```
import
import
import

DOWNLOAD_ROOT =
    "https://raw.githubusercontent.com/ageron/handson-ml2/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):
    os.makedirs(housing_path, exist_ok=True)
    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()
```

<sup>10</sup> Может также возникнуть необходимость проверить ограничения правового характера, такие как персональные поля, которые никогда не должны копироваться в недежные хранилища данных.

<sup>11</sup> В реальном проекте вы сохранили бы этот код в файле Python, но пока его можно просто ввести в тетради Jupyter.

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

Функция `load_housing_data()` возвращает `pandas`-объект `DataFrame`, содержащий все данные.

## 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()

	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, типа каждого атрибута и количества ненулевых значений (рис. 2.6).

В наборе данных есть 20640 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`, так что могло бы 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).

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).

```
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

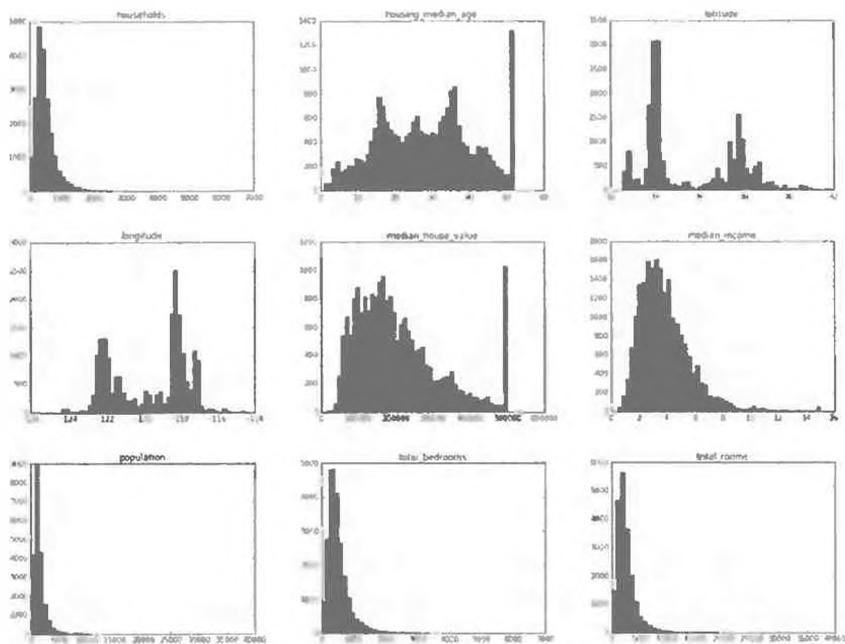
Рис. 2.7. Сводка по числовым атрибутам

В строке `std` показано *стандартное отклонение (standard deviation)*, которое измеряет разброс значений<sup>12</sup>. 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 первым *квартилем (quartile)*), медианой и 75-м процентилем (или третьим *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 вызывать метод `hist()` на целом наборе данных (как демонстрируется в следующем коде), и он вычертит гистограммы для всех числовых атрибутов (рис. 2.8):

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

<sup>12</sup>Стандартное отклонение обычно обозначается как  $\sigma$  (греческая буква “сигма”) и представляет собой квадратный корень из дисперсии (*variance*), которая является усреднением квадратичного отклонения от среднего значения. Когда признак имеет колоконообразное нормальное распределение (также называемое *гауссовым распределением*), что очень распространено, то применяется правило “68–95–99.7”: около 68% значений находятся внутри 1 $\sigma$  среднего, 95% — внутри 2 $\sigma$  и 99.7% — внутри 3 $\sigma$ .



*Рис. 2.8. Гистограммы для числовых атрибутов*



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.

Ниже описано то, что можно отметить в полученных гистограммах.

1. Прежде всего, атрибут медианного дохода (`median_income`) не выглядит выраженным в долларах США. После согласования с командой, собирающей 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. Числа представляют приблизительно десятки тысяч долларов (скажем, 3 в действительности означает около \$30 000). 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 узнатъ, что цены никогда не выходят за упомянутые пределы. Вам понадобится выяснить у клиентской команды (команды, которая будет потреблять вывод вашей системы), будет это проблемой или нет. Если они сообщают, что нуждаются в точных прогнозах даже за рамками \$500 000, тогда у вас есть два варианта.
  - a) 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 в случае прогнозирования цен 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, чем влево от медианы. 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 странице может показаться странной. В конце концов, вы лишь мельком взглянули на данные и непременно должны узнат о них намного больше, прежде чем решить, какие алгоритмы применять, не так ли? Все действительно так, но ваш мозг является удивительной системой обнаружения паттернов, что означает его крайнюю предрасположенность 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 просто: выберите случайным образом ряд образцов, обычно 20% набора данных (или меньше, если набор данных очень крупный), и отложите их:

```
import          as
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]
```

Вот как впоследствии можно применять функцию `split_train_test()`<sup>13</sup>:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Да, все работает, но далеко от совершенства: если вы запустите программу еще раз, то она сгенерирует другой испытательный набор! С течением време-

<sup>13</sup> Когда пример в книге содержит смесь кода и вывода, для улучшения читабельности он представлен сродни тому, как он выглядит в интерпретаторе Python: строки кода снабжены префиксом `>>>` (или `...` для блоков с отступами), а вывод не содержит префиксов.

мени вы (или ваши алгоритмы МО) сумеют увидеть полный набор данных, чего желательно избежать.

Одно из решений предусматривает сохранение испытательного набора после первого запуска и его загрузку при последующих запусках. Другой вариант предполагает установку начального значения генератора случайных чисел (например, `np.random.seed(42)`)<sup>14</sup> 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. Для обеспечения стабильного разделения обучающий/испытательный даже после обновления набора данных обычно при решении, должен ли образец быть помещен в испытательный набор, используют идентификатор образца (при условии, что образцы имеют уникальные и неизменяемые идентификаторы). Например, вы могли бы вычислять хеш-значение идентификатора каждого образца и помещать образец в испытательный набор, если его хеш-значение меньше или равно 20% максимального хеш-значения. 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 включать образцы, которые присутствовали в испытательном наборе ранее.

Вот возможная реализация:

```
from zlib import crc32

def test_set_check(identifier, test_ratio):
    return crc32(np.int64(identifier)) & 0xffffffff < test_ratio * 2**32

def split_train_test_by_id(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio))
    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 в качестве идентификатора индекса строки:

<sup>14</sup> Вы часто будете видеть, что 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](https://ru.wikipedia.org/wiki/Ответ_на_главный_вопрос_жизни,_вселенной_и_всего_такого) — Примеч.пер.).

```
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 идентификатор<sup>15</sup>:

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

Библиотека 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, например, при наличии отдельного объекта `DataFrame` для меток):

```
from               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 привнесения 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

<sup>15</sup> На самом деле информация о местоположении является довольно грубой, в результате чего многие округи будут иметь в точности один и тот же идентификатор, а потому окажутся в том же самом наборе (испытательном или обучающем). Это привносит некоторое неуместное смещение выборки.

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. Если бы люди, проводившие анкетирование, использовали чисто случайную выборку, то вероятность выборки склоненного испытательного набора, содержащего менее 49% или более 54% женщин, составляла бы приблизительно 12%. В обеих ситуациях результаты исследования оказались бы значительно смещеными.

Предположим, вы побеседовали с экспертами, которые сказали, 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 1.5–6 (т.е. \$15 000–\$60 000), но некоторые медианные доходы выходят далеко за пределы 6. 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 применяет функцию `pd.cut()` для создания атрибута категории дохода (`income_cat`) с пятью категориями (помеченными от 1 до 5): категория 1 простирается от 0 до 1.5 (т.е. менее \$15 000), категория 2 — от 1.5 до 3 и т.д.:

```
housing["income_cat"] = pd.cut(housing["median_income"],
                                bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                labels=[1, 2, 3, 4, 5])
```

Категории дохода представлены на рис. 2.9:

```
housing["income_cat"].hist()
```

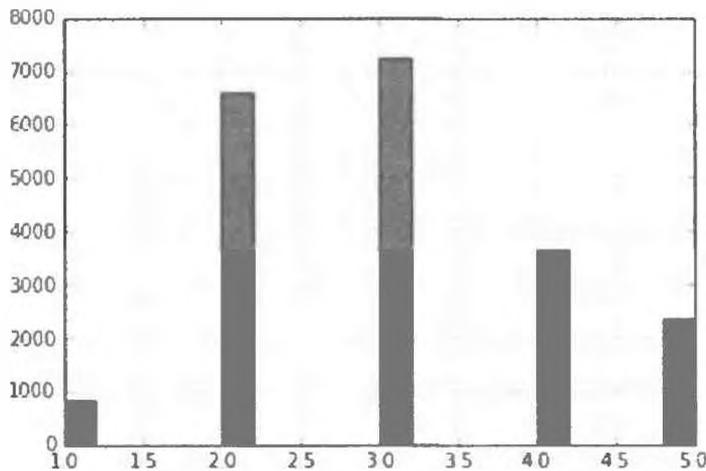


Рис. 2.9. Гистограмма категорий дохода

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

```
from 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 испытательном наборе:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
3    0.350533
2    0.318798
4    0.176357
5    0.114583
1    0.039729
Name: income_cat, dtype: float64
```

С помощью похожего кода вы можете измерить пропорции категорий дохода в полном наборе данных. На рис. 2.10 сравниваются пропорции категорий дохода во всем 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 склоненным.

	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 возвратить данные в первоначальное состояние:

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

Мы потратили довольно много 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 разновидности 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 имеется 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")
```

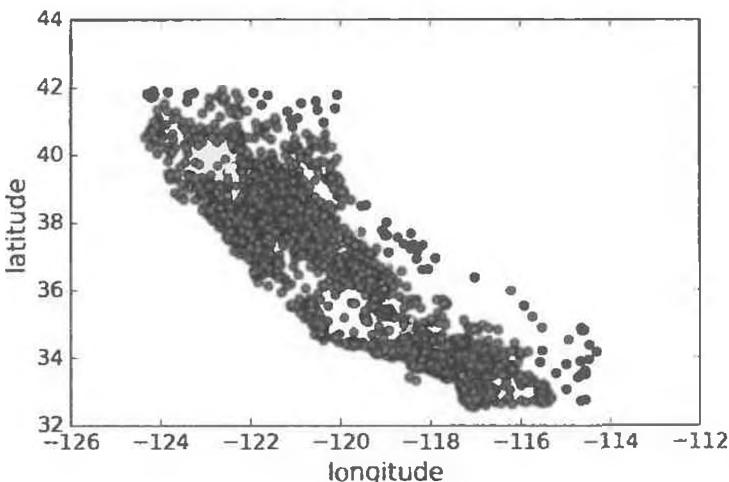
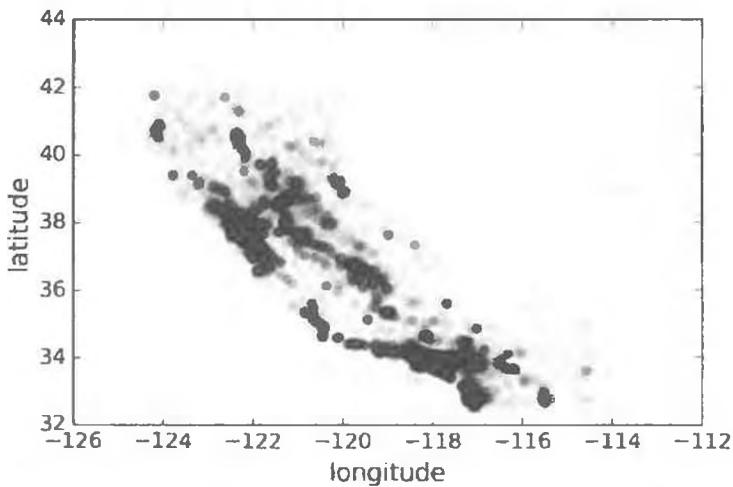


Рис. 2.11. График рассеяния географических данных

Очертания напоминают Калифорнию, но кроме этого 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)
```

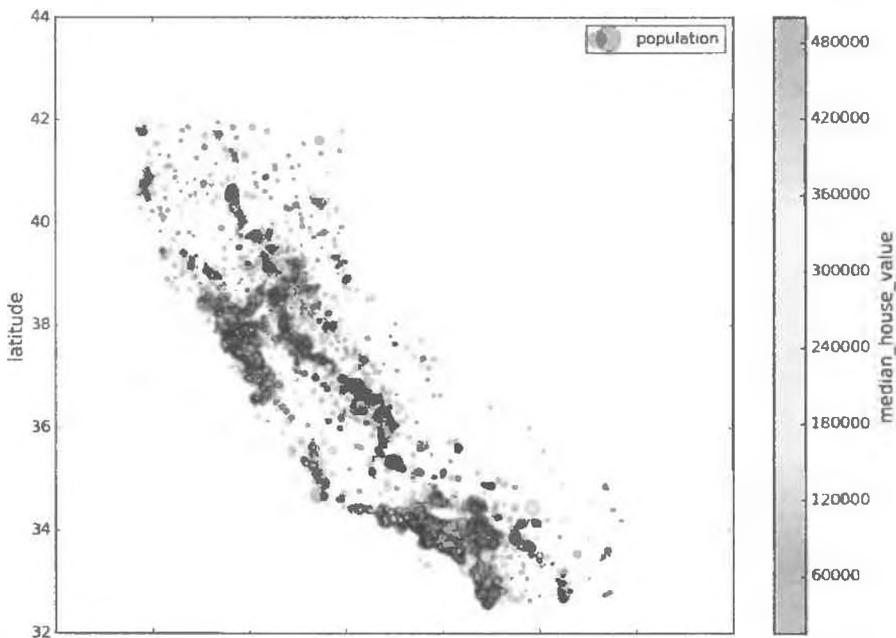
Теперь все стало гораздо лучше: вы можете четко видеть области с высокой плотностью, а именно — область залива Сан-Франциско и поблизости Лос-Анджелеса и Сан-Диего плюс длинная линия довольно высокой плотности в Большой Калифорнийской долине, в частности около Сакраменто и Фресно.



*Рис. 2.12. Улучшенная визуализация выделяет области с высокой плотностью*

Наш мозг очень хорош в выявлении паттернов на рисунках, но может возникнуть необходимость поэкспериментировать с параметрами визуализации, чтобы сделать паттерны более заметными.

Давайте обратимся к ценам на дома (рис. 2.13).



*Рис. 2.13. Цены на дома в Калифорнии: красный — дорогие, синий — дешевые, чем большие круги, тем выше заселенность области*

Радиус каждого круга представляет население округа (параметр `s`), а цвет — цену (параметр `c`). Мы будем применять предварительно определенную карту цветов (параметр `cmap`) по имени `jet`, которая простирается от синего цвета (низкие цены) до красного (высокие цены)<sup>16</sup>:

```
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. Алгоритм кластеризации должен быть полезен для выявления главного кластера и для добавления новых признаков, 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.

## 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
```

<sup>16</sup> 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 и можно было ожидать). Можете также добавить небольшой участок желтого около Сакраменто.

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  $0$ , означают отсутствие линейной корреляции. На рис. 2.14 приведены разнообразные графики наряду с коэффициентом корреляции между их горизонтальными и вертикальными осями.

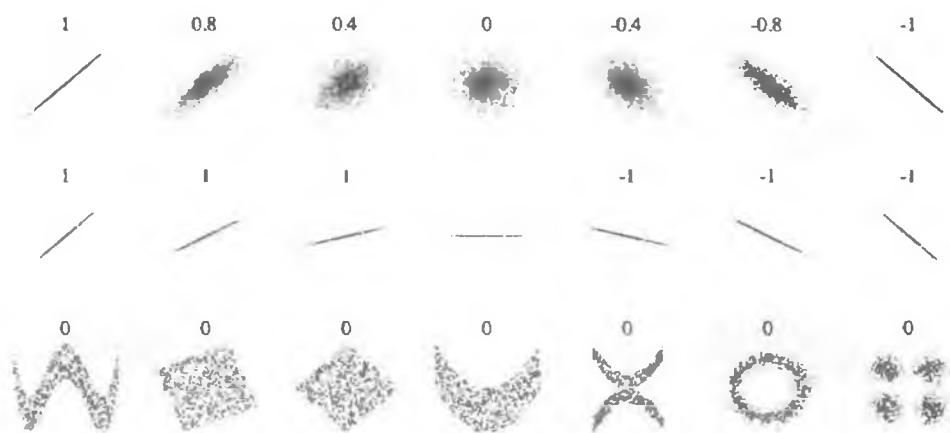


Рис. 2.14. Стандартный коэффициент корреляции различных наборов данных  
(Википедия: источники изображений)



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  $0$ , тогда  $y$  обычно повышается”). Обратите внимание, что все графики в нижнем ряду имеют коэффициент корреляции, равный  $0$ , 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. Например, ваш рост в сантиметрах имеет коэффициент корреляции  $1$  с вашим ростом в метрах или в нанометрах.

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

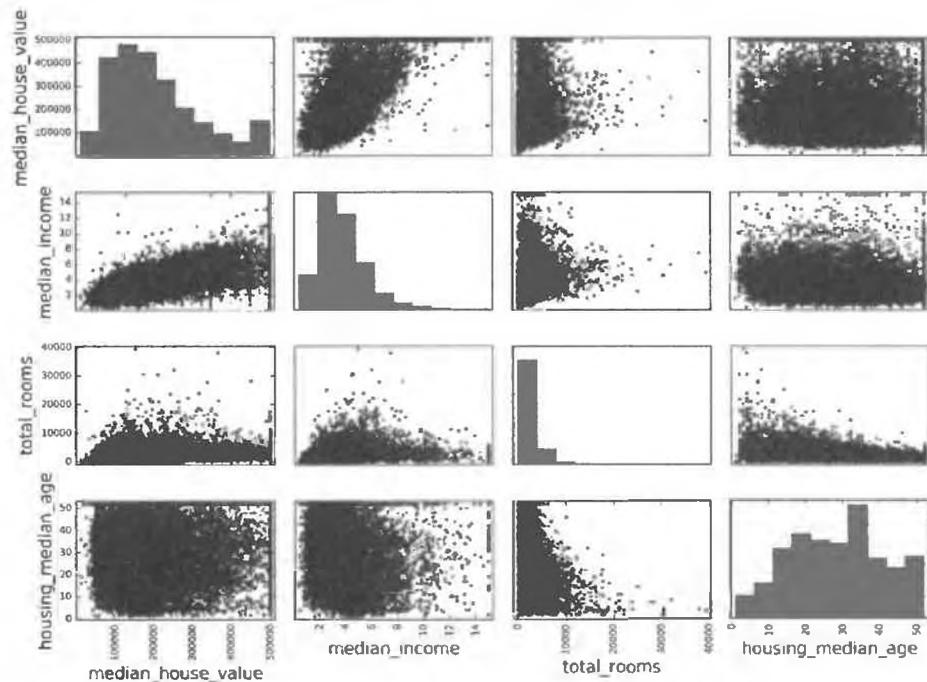


Рис. 2.15. Матрица рассеяния, вычерчивающая каждый числовой атрибут по отношению к каждому другому числовому атрибуту, плюс гистограмма каждого числового атрибута

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 по pandas).

Самым 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)
```

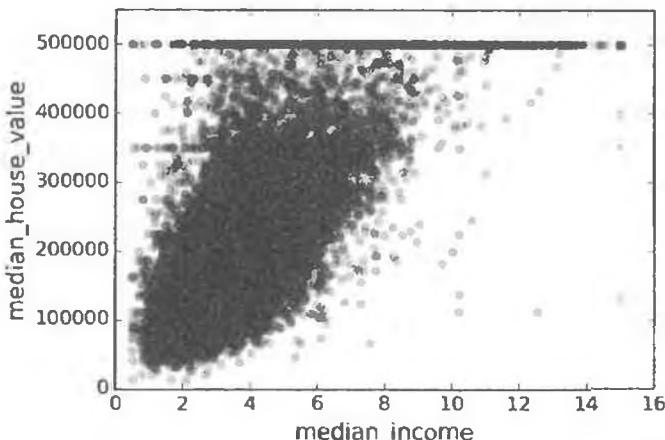


Рис. 2.16. Медианный доход в сравнении со средней стоимостью дома

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 округи, 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 уловить 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 могут потребовать трансформирования (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 подготовкой данных для алгоритмов МО — опробовать разнообразные комбинации атрибутов. Например, суммарное 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 суммарное число комнат в округе — очевидно, чем крупнее дома, тем они дороже.

Описанный раунд исследований вовсе не обязан быть абсолютно исчерпывающим; вопрос в том, чтобы начать все как следует и быстро уловить 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 такие 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 их передачей алгоритмам.
- 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`). Мы также разделим прогнозаторы и метки, поскольку 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:

1. избавиться от соответствующих округов;
2. избавиться от всего атрибута;
3. 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 избрали вариант 3, то должны подсчитать медиану обучающего набора и применять ее для значений, отсутствующих в обучающем наборе. 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 системы в эксплуатацию для замены недостающих значений в новых данных.

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

```
from                 import SimpleImputer
imputer = SimpleImputer(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 обычновенный 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,
                           index=housing_num.index)
```

## Проектное решение Scikit-Learn

API-интерфейс Scikit-Learn необыкновенно хорошо спроектирован. Ниже перечислены главные проектные принципы (<https://homl.info/11>)<sup>17</sup>.

- **Согласованность.** Все объекты разделяют согласованный и простой интерфейс.
  - **Оценщики (estimator).** Любой объект, который способен 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).

<sup>17</sup> За дополнительными сведениями по проектным принципам обращайтесь к работе Ларса Битинка *API Design for Machine Learning Software: Experiences from the Scikit-Learn Project* (Проектирование API-интерфейса для ПО машинного обучения: опыт проекта Scikit-Learn) (препринт arXiv:1309.0238 (2013 г.)).

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 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*. 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 методов в случае алгоритмов обучения с учителем)<sup>18</sup>.
- **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_`).

---

<sup>18</sup> Некоторые прогнозаторы также предоставляют методы для измерения степени достоверности своих прогнозов.

- **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 обоснованные стандартные значения для большинства параметров, облегчая быстрое создание базовой рабочей системы.

## Обработка текстовых и категориальных атрибутов

До сих пор мы имели дело только с числовыми атрибутами, но теперь давайте взглянем на текстовые атрибуты. В нашем наборе данных есть лишь один такой: атрибут `ocean_proximity`. Просмотрим его значение для первых 10 образцов:

```
>>> housing_cat = housing[["ocean_proximity"]]
>>> housing_cat.head(10)
   ocean_proximity
0    <1H OCEAN
1    <1H OCEAN
2    NEAR OCEAN
3    INLAND
4    <1H OCEAN
5    INLAND
6    <1H OCEAN
7    INLAND
8    <1H OCEAN
9    <1H OCEAN
```

Это не произвольный текст: существует ограниченное количество возможных значений, каждое из которых представляет категорию. Таким образом, атрибут `ocean_proximity` является категориальным. Большинство алгоритмов МО предпочитают работать с числами, так что мы преобразуем ка-

тегории из текста в числа, для чего применим класс `OrdinalEncoder` из `Scikit-Learn`<sup>19</sup>:

```
>>> from                                     import OrdinalEncoder
>>> ordinal_encoder = OrdinalEncoder()
>>> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
>>> housing_cat_encoded[:10]
array([[0.],
       [0.],
       [4.],
       [1.],
       [0.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.]])
```

Вы можете получить список категорий с использованием переменной экземпляра `categories_`. Она представляет собой список, содержащий одномерный массив категорий для каждого категориального атрибута (в данном случае список с единственным массивом, т.к. есть только один категориальный атрибут):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
# Менее часа до океана, далеко от океана, остров,
# близко к бухте, близко к океану
```

С таким представлением связана одна проблема: алгоритмы МО будут предполагать, что два соседних значения более похожи, чем два отдаленных значения. В некоторых ситуациях подобное допущение может быть приемлемым (например, для упорядоченных категорий, таких как “плохо”, “средне”, “хорошо” и “великолепно”), но очевидно, что для столбца `ocean_proximity` это не так (скажем, категории 0 и 4, безусловно, более похожи, чем категории 0 и 1). 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 со-

<sup>19</sup> Этот класс доступен в `Scikit-Learn 0.20` и последующих версиях. Если вы используете более раннюю версию, тогда позаботьтесь об обновлении или примените метод `Series.factorize()` из `pandas`.

бой “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). Новые атрибуты иногда называются *фиктивными* (*dummy*) атрибутами. Библиотека Scikit-Learn предлагает класс `OneHotEncoder` для преобразования категориальных значений в векторы в унитарном коде<sup>20</sup>:

```
>>> from import OneHotEncoder  
>>> cat_encoder = OneHotEncoder()  
>>> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)  
>>> housing_cat_1hot  
<16512x5 sparse matrix of type '<class 'numpy.float64'>'>  
with 16512 stored elements in Compressed Sparse Row format>
```

Обратите внимание, что выходными данными 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 массиву<sup>21</sup>, но если вы действительно хотите преобразовать разреженную матрицу в (плотный) массив NumPy, тогда вызовите метод `toarray()`:

```
>>> housing_cat_1hot.toarray()  
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.]])
```

И снова получить список категорий можно с использованием переменной экземпляра `categories_`:

<sup>20</sup> До выхода версии Scikit-Learn 0.20 можно было кодировать только числовые категориальные значения, но начиная с версии Scikit-Learn 0.20, поддерживаются других типы входных данных, включая текстовые категориальные значения.

<sup>21</sup> Дополнительные сведения ищите в документации по SciPy.

```
>>> 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 появлению большого количества входных признаков. В результате может замедлиться обучение и ухудшиться эффективность. Когда подобное происходит, вы можете принять решение заменить категориальные входные данные удобными числовыми признаками, связанными с категориями: скажем, вы могли бы заменить признак `ocean_proximity` расстоянием до океана (аналогично код страны можно было бы заменить населением страны и ВВП на душу населения). В качестве альтернативы вы могли бы заменить каждую категорию поддающимся обучению вектором низкой размерности, который называется *вложением* (*embedding*). Представление каждой категории выяснилось бы во время обучения. Это пример *обучения представлению* (*representation learning*); дополнительные сведения ищите в главах 13 и 17.

## Специальные трансформаторы

Хотя библиотека Scikit-Learn предлагает много удобных трансформаторов, вам придется писать собственные трансформаторы для задач вроде 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 бесплатно, просто добавив `TransformerMixin` в качестве базового класса. Если вы добавите `BaseEstimator` как базовый класс (и откажетесь от параметров `*args` и `**kwargs` в своем конструкторе), тогда также 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 import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6
class (BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # нет *args
                                                # или **kargs
        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[:, households_ix]
        population_per_household =
            X[:, population_ix] / X[:, households_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 практических стандартных значений). Этот гиперпараметр позволит легко выяснить, помогает алгоритмам МО добавление данного атрибута или нет. Как правило, 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 придется применять к своим данным, будет *масштабирование признаков* (*feature scaling*). За немногими исключениями алгоритмы МО не особенно хорошо выполняются, когда входные числовые атрибуты имеют сильно различающиеся масштабы.

Это касается данных о домах: общее количество комнат имеет диапазон 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 называют его *нормализацией (normalization)*) выполняется довольно просто: значения смещаются и изменяются так, чтобы в итоге находиться в диапазоне от 0 до 1. Цель достигается вычитанием минимального значения и делением на разность максимального и минимального значений. Для решения такой задачи библиотека Scikit-Learn предлагает трансформатор по имени `MinMaxScaler`. У него есть гиперпараметр `feature_range`, который позволяет изменять диапазон, если по какой-то причине 0–1 не устраивает.

Стандартизация обеспечивается по-другому: сначала вычитается среднее значение (поэтому стандартизованные значения всегда имеют нулевое среднее) и затем производится деление на стандартное отклонение, так что результирующее распределение имеет единичную дисперсию. В отличие от масштабирования по минимаксу стандартизация не привязывает значения к специальному 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 влияние на стандартизацию оказалось бы не особенно большим. Для стандартизации в библиотеке Scikit-Learn предоставляется 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, библиотека Scikit-Learn предлагает класс `Pipeline`, призванный помочь справиться с такими последовательностями трансформаций. Ниже показан небольшой конвейер для числовых атрибутов:

```
from               import Pipeline
from               import StandardScaler

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

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

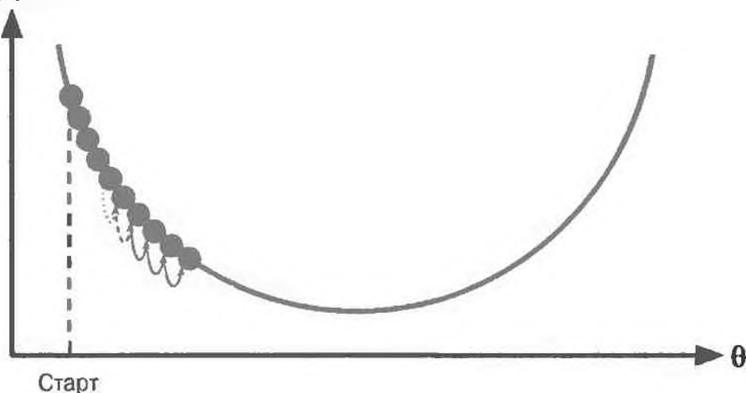
Конструктор `Pipeline` принимает список пар “имя/оценщик”, определяющий последовательность шагов. Все кроме последнего оценщика обязаны быть трансформаторами (т.е. они должны иметь метод `fit_transform()`). Имена могут быть какими угодно (при условии, что они уникальны и не содержат два символа подчеркивания подряд (`__`)); они пригодятся позже при настройке гиперпараметров.

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 (и, конечно же, метод `fit_transform()`, который мы использовали).

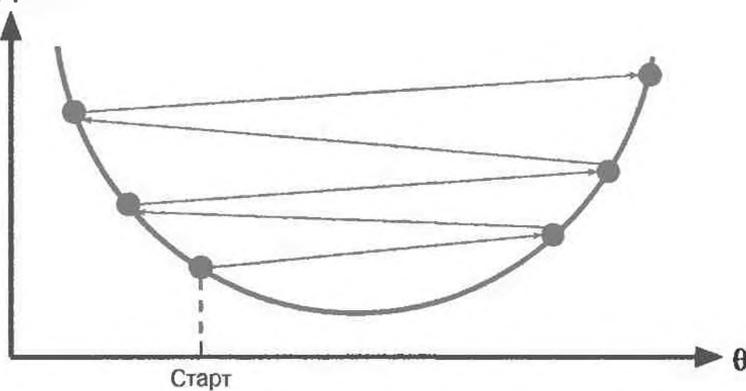
До сих пор мы обрабатывали категориальные и числовые столбцы раздельно. Было бы удобнее иметь единственный трансформатор, способный обрабатывать все столбцы путем применения соответствующих трансформаций к каждому столбцу. Для этой цели в версии Scikit-Learn 0.20 появился класс `ColumnTransformer`, который великолепно работает с pandas-

Издержки

*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).

Издержки

*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 затрудняют схождение в минимуме. 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.

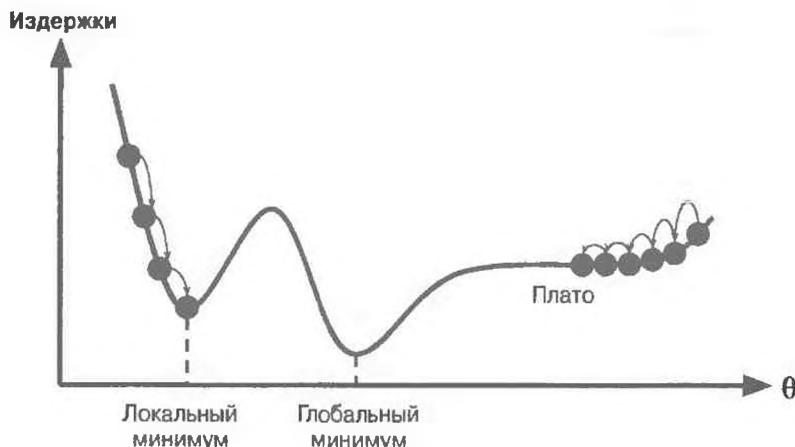


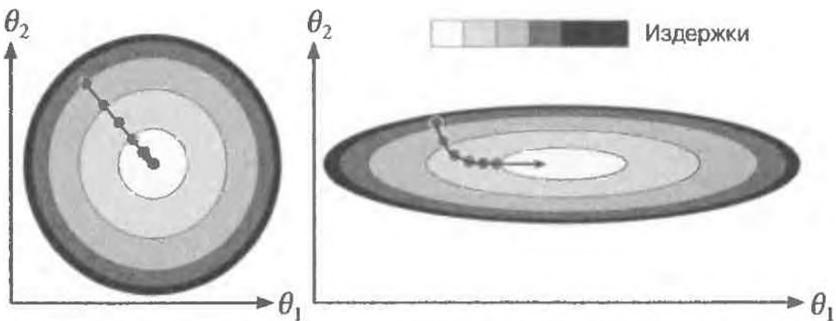
Рис. 4.6. Просчеты градиентного спуска

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 образом<sup>3</sup>. 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 чаши, 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)<sup>4</sup>.

<sup>3</sup> Выражаясь формально, ее производная является липшиц-непрерывной (*Lipschitz continuous*).

<sup>4</sup> Так как признак 1 меньше, для воздействия на функцию издержек требуется большее изменение в  $\theta_1$ , что и объясняет вытянутость чаши вдоль оси  $\theta_1$ .



*Рис. 4.7. Градиентный спуск с масштабированием признаков (слева) и без него (справа)*

Как видите, слева алгоритм градиентного спуска устремляется прямо к минимуму, из-за чего достигает его быстро, а справа он сначала двигается 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 схождение.

Диаграмма на рис. 4.7 также демонстрирует тот факт, что обучение модели означает поиск комбинации параметров модели, которая доводит до минимума 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 иглу 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  $\theta_j$ . Другими словами, вам необходимо подсчитать, насколько сильно функция издержек будет изменяться при небольшом изменении  $\theta_j$ . Это называется *частной производной* (*partial derivative*) и 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, имеющий больше трех измерений). Уравнение 4.5 вычисляет частную производную функции издержек в отношении параметра  $\theta_j$ , известную как  $\frac{\partial}{\partial \theta_j} \text{MSE}(\theta)$ .

#### Уравнение 4.5. Частные производные функции издержек

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

Вместо вычисления таких частных производных по отдельности вы можете воспользоваться уравнением 4.6, чтобы вычислить их все сразу. **Вектор-градиент (gradient vector)**, обозначаемый 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

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



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 (на самом деле его лучше было бы назвать *полным градиентным спуском*). 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 или разложения SVD.

Имея вектор-градиент, который указывает вверх, вы просто двигаетесь в противоположном направлении вниз. Это означает вычитание  $\nabla_{\theta} \text{MSE}(\theta)$  из  $\theta$ . Именно здесь в игру вступает скорость обучения  $\eta$ :<sup>5</sup> умножение вектора-градиента на  $\eta$  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 оказалась 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 спустя долгое время. Посредине скорость обучения выглядит довольно хорошей: алгоритм сходится в решение всего за несколько итераций. Справа скорость обучения чересчур высокая: алгоритм расходится, беспорядочно перескакивая с места на место и с каждым шагом фактически все больше удаляясь от решения.

Для нахождения хорошей скорости обучения вы можете применять решетчатый поиск (см. главу 2).

<sup>5</sup> Эта ( $\eta$ ) — седьмая буква греческого алфавита.

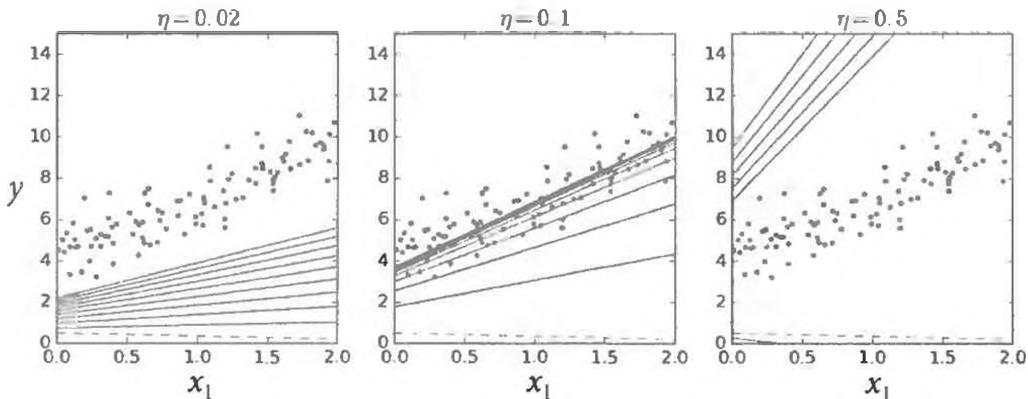


Рис. 4.8. Градиентный спуск с различными скоростями обучения

Тем не менее, количество итераций можно ограничить, если желательно, чтобы решетчатый поиск был в состоянии исключить модели, которые требуют слишком длительного времени на схождение.

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 растратчивать 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  $\epsilon$  (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 в оптимальное решение, но возможно вам придется немного подождать. В зависимости от формы функции издержек он может требовать  $O(1/\epsilon)$  итераций для достижения оптимума внутри диапазона  $\epsilon$ . Если вы разделите допуск на 10, чтобы иметь более точное решение, тогда алгоритм может 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 выбирает из обучающего набора случайный образец и вычисляет градиенты на основе только этого единственного образца. Очевидно, работа только с одним образцом в каждый момент времени делает алгоритм 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 обучения; см. главу 1).

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 затихая (рис. 4.9). Таким образом, после окончания алгоритма финальные значения параметров оказываются хорошими, но не оптимальными.

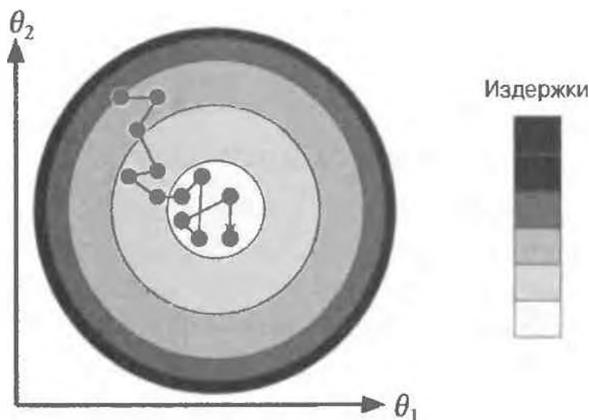


Рис. 4.9. При стохастическом градиентном спуске каждый шаг обучения будет выполняться гораздо быстрее, но также окажется в намного большей степени случайным, чем при пакетном градиентном спуске

Когда функция издержек крайне нерегулярна (как на рис. 4.6), то фактически это 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, чтобы избегать локальных оптимумов, но плоха, т.к. означает, что алгоритм может никогда не осесть в 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 в глобальном минимуме. Такой процесс сродни имитации отжига (*simulated annealing*), а алгоритм навеян процессом закалки в металлургии, где расплавленный металл медленно охлаждается. Функция, которая определяет скорость обучения на каждой итерации, называется *графиком обучения* (*learning module*). 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 квазипримимальное решение, когда обучение прекращается слишком рано.

Приведенный ниже код реализует стохастический градиентный спуск с применением простого графика обучения:

```
n_epochs = 50
t0, t1 = 5, 50    # 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 20 шагов обучения (заметьте, насколько нерегулярны шаги).

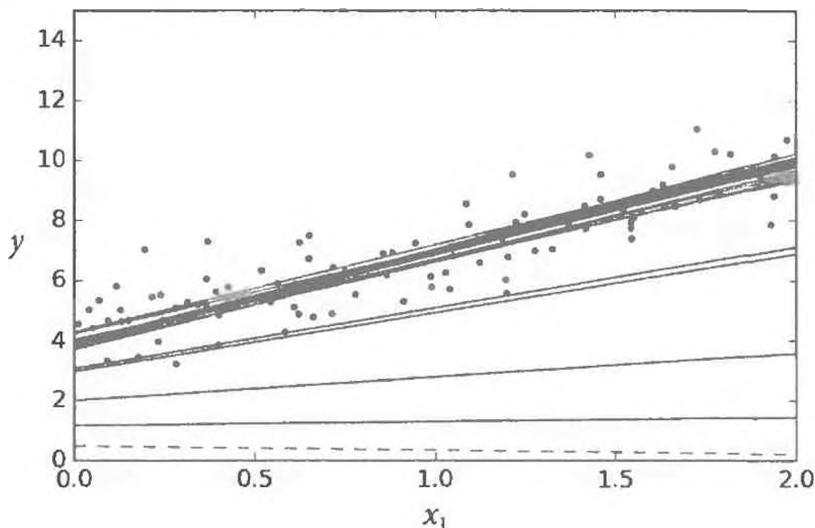


Рис. 4.10. Первые 20 шагов стохастического градиентного спуска

Обратите внимание, что поскольку образцы выбираются случайно, некоторые из них могут быть выбраны несколько раз за эпоху, тогда как другие — вообще не выбираются. Если вы хотите обеспечить, чтобы алгоритм в рамках каждой эпохи проходил по каждому образцу, то другой подход предусматривает тасование обучающего набора (обеспечив одновременное тасование входных признаков и меток), проход по нему образец за образцом, снова тасование и т.д. Однако при таком подходе схождение обычно происходит медленнее.



При использовании стохастического градиентного спуска обучающие образцы должны быть *независимыми и одинаково распределенными* (*independent and identically distributed — IID*) для гарантии того, что параметры доберутся до глобального оптимума в среднем. Простой способ обеспечить это — тасо-

вать образцы во время обучения (скажем, выбирать каждый образец случайным образом или тасовать обучающий набор в начале каждой эпохи). Если не перетасовать образцы (например, если они отсортированы по меткам), тогда алгоритм SGD начнет с оптимизации для одной метки, затем для следующей и т.д., и он не сядет поблизости к глобальному минимуму.

Для выполнения линейной регрессии, применяющей SGD, с помощью Scikit-Learn, вы можете использовать класс `SGDRegressor`, который по умолчанию настроен на оптимизацию функции издержек в виде квадратичной ошибки. Показанный далее код выполняется для максимум 1000 эпох или до тех пор, пока потеря не упадет ниже 0.001 на протяжении одной эпохи (`max_iter=1000, tol=1e-3`). Он начинает со скорости обучения 0.1 (`eta0=0.1`), применяя стандартный график обучения (отличающийся от предыдущего). Наконец, он не использует какую-либо регуляризацию (`penalty=None`; вскоре мы обсудим это более подробно):

```
from import SGDRegressor
sgd_reg = SGDRegressor(max_iter=    , tol= - , penalty=None, eta0=   )
sgd_reg.fit(X, y.ravel())
```

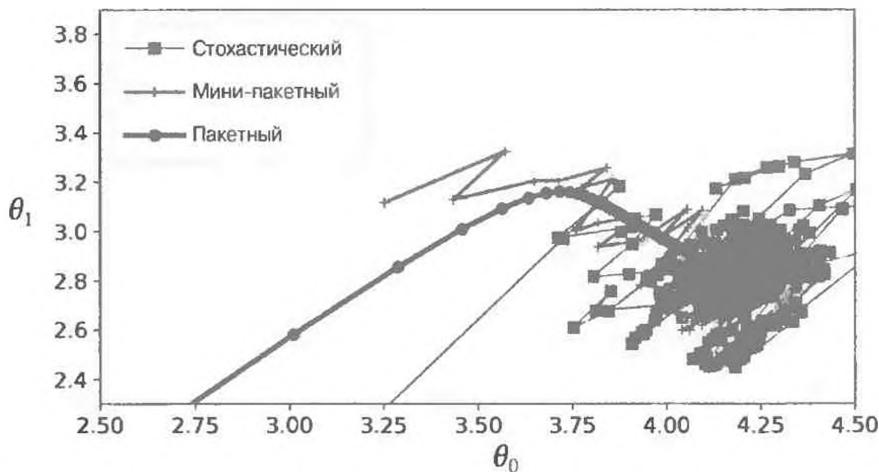
И снова вы находите решение, довольно близкое к тому, что возвращалось нормальным уравнением:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([4.24365286]), array([2.8250878]))
```

## Мини-пакетный градиентный спуск

Последний рассматриваемый алгоритм градиентного спуска называется *мини-пакетным градиентным спуском* (*Mini-batch Gradient Descent*). Понять его 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 менее непредсказуемо, чем у SGD, в особенности при довольно крупных мини-пакетах. В результате мини-пакетный градиентный спуск закончит блуждания чуть ближе к минимуму, чем SGD, но ему может быть труднее уйти от локальных минимумов (в случае задач, которые страдают от локальных минимумов, в отличие от линейной регрессии). 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 мини-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 регрессии<sup>6</sup> (вспомните, что  $m$  — количество обучающих образцов, а  $n$  — количество признаков); взгляните на табл. 4.1.

<sup>6</sup> В то время как нормальное уравнение может выполнять только линейную регрессию, вы увидите, что алгоритмы градиентного спуска могут использоваться для обучения многих других моделей.

**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	-
SVD	Быстрый	Нет	Медленный	0	Нет	LinearRegression
Пакетный градиентный спуск	Медленный	Нет	Быстрый	2	Да	SGDRegressor
Стохастический градиентный спуск	Быстрый	Да	Быстрый	$\geq 2$	Да	SGDRegressor
Мини-пакетный градиентный спуск	Быстрый	Да	Быстрый	$\geq 2$	Да	SGDRegressor



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 сложнее прямой линии? Удивительно, но вы 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*<sup>7</sup> (плюс некоторый шум, как показано на рис. 4.12):

```
m = 100
X = np.random.rand(m, 1) -
y = ... * X**2 + X + ... + np.random.randn(m, 1)
```

Безусловно, прямая линия никогда не будет подогнана под такие данные должным образом. Потому мы воспользуемся классом *Polynomial Features* из Scikit-Learn, чтобы преобразовать наши обучающие данные,

<sup>7</sup> Квадратное уравнение имеет вид  $y = ax^2 + bx + c$ .

добавив в качестве нового признака 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[ ]  
array([-0.75275929])  
>>> X_poly[ ]  
array([-0.75275929,  0.56664654])
```

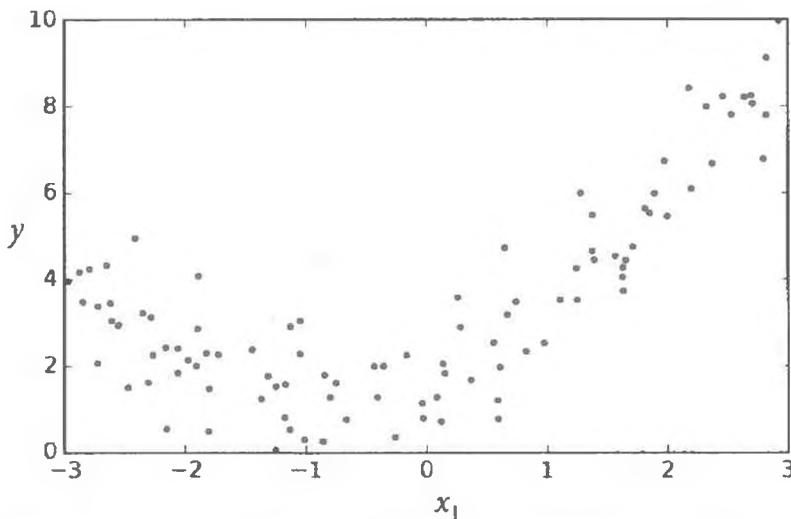


Рис. 4.12. Сгенерированный нелинейный и зашумленный набор данных

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]]))
```

Неплохо: модель оценивает функцию как  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$ , 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 класс `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  $ab^2$ .

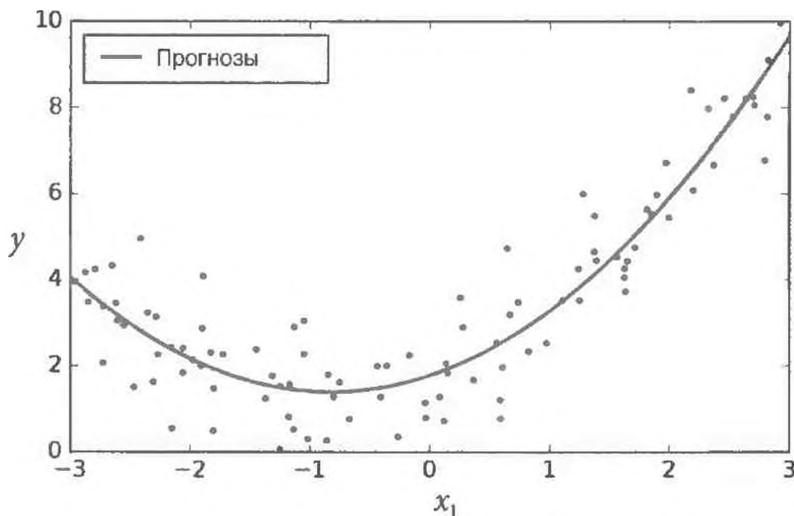


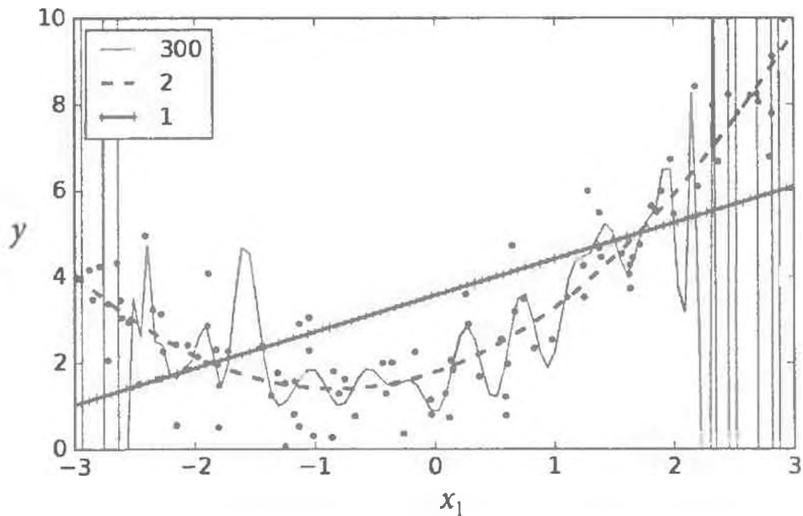
Рис. 4.13. Прогнозы полиномиальной регрессионной модели



Класс `PolynomialFeatures(degree=d)` трансформирует массив, содержащий  $n$  признаков, в массив, содержащий  $\frac{(n+d)!}{d! \cdot n!}$  признаков, где  $n!$  — факториал числа  $n$ , который равен  $1 \times 2 \times 3 \times \dots \times n$ . Остерегайтесь комбинаторного взрыва количества признаков!

## Кривые обучения

Полиномиальная регрессия высокой степени, вероятно, обеспечит гораздо лучшую подгонку к обучающим данным, чем обыкновенная линейная регрессия. 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 второй степени). Обратите внимание на то, как полиномиальная модель 300-й степени колеблется, чтобы как можно больше приблизиться к обучающим образцам.



*Рис. 4.14. Полиномиальная регрессия высокой степени*

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, что имеет смысл, поскольку данные генерировались с использованием квадратного уравнения. Но учитывая, что обычно вы не будете знать функцию, применяемую для генерации данных, то каким образом решать, насколько сложной должна быть модель? Как выяснить, что модель переобучается или недообучается на данных?

В главе 2 с помощью перекрестной проверки оценивалась эффективность обобщения модели. Если согласно метрикам перекрестной проверки модель хорошо работает 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 эффективности модели на обучающем наборе и проверочном наборе как функции от размера обучающего набора (или от 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 import mean_squared_error
from 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=...)
    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[:m],
                                                y_train_predict))
        val_errors.append(mean_squared_error(y_val, y_val_predict))
    plt.plot(np.sqrt(train_errors), "r-", linewidth=2, label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=2, label="val")

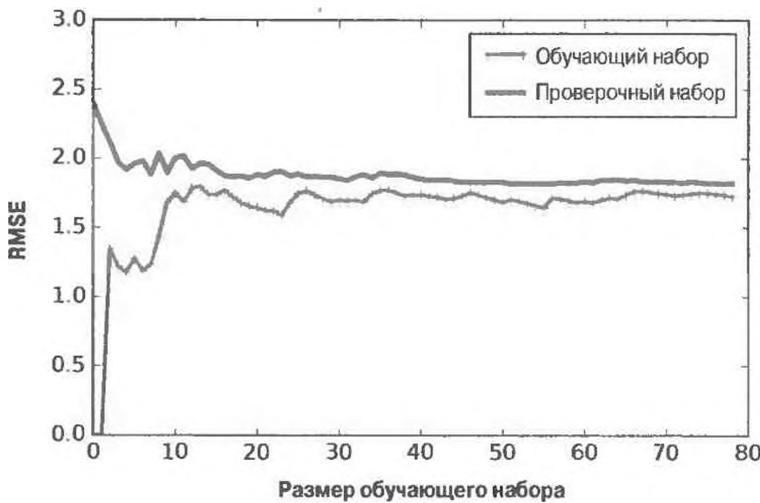
```

Давайте взглянем на кривые обучения обычной 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*

Представленная модель, которая недообучается, заслуживает некоторых пояснений. Первым делом обратите внимание на эффективность 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 двигается вверх, пока не стабилизируется, когда добавление новых образцов в обучающий набор не делает среднюю ошибку намного лучше или хуже. Теперь перейдем к просмотру эффективности 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 вблизи к другой кривой. Такие кривые обучения типичны для модели, которая недообучается. 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 десятой степени на тех же самых данных (рис. 4.16):

```
from               import Pipeline
polynomial_regression = Pipeline([
    ("poly_features", PolynomialFeatures(degree=  , include_bias=False)),
    ("lin_regr", LinearRegression()), ])
plot_learning_curves(polynomial_regression, X, y)
```

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 сближаться.



Один из способов улучшения переобученной модели заключается в предоставлении ей добавочных обучающих данных до тех пор, пока ошибка проверки не достигнет ошибки обучения.

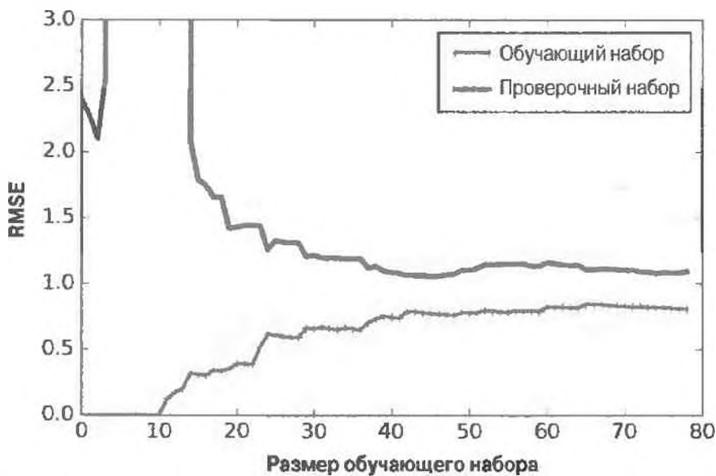


Рис. 4.16. Кривые обучения для полиномиальной модели десятой степени

### Компромисс между смещением и дисперсией

Важным теоретическим результатом статистики и машинного обучения считается 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 являются линейными, когда на самом деле они квадратичные. Модель с высоким смещением почти наверняка недообучится на обучаемых данных<sup>8</sup>.

**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.

**Неустранимая погрешность.** Эта часть появляется вследствие зашумленности самих данных. Единственный способ сократить неустранимую 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.

<sup>8</sup> 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 снизить переобучение заключается в том, чтобы регуляризовать модель (т.е. ограничить ее): чем меньше степеней свободы она имеет, тем труднее ее будет переобучить данными. Простой метод регуляризации полиномиальной модели предполагает сокращение 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 эффективность модели с использованием нерегуляризованной меры эффективности.



Отличие функции издержек, применяемой во время обучения, от меры эффективности, используемой для проверки — довольно распространенное явление. Помимо регуляризации еще одна причина отличия между ними связана с тем, что хорошая функция издержек при обучении должна иметь удобные для оптимизации производные, тогда как мера эффективности, применяемая для проверки, обязана быть как можно ближе к финальной цели. Например, классификаторы часто обучаются с использованием такой функции издержек, как логарифмическая потеря (*log loss*; обсуждается далее в главе), но оценивается с применением точности/полноты.

Гиперпараметр  $\alpha$  управляет тем, насколько необходимо регуляризовать модель. Когда  $\alpha = 0$ , гребневая регрессия оказывается просто линейной регрессией. При очень большом значении  $\alpha$  все веса становятся крайне близкими к нулю, и результатом будет ровная линия, проходящая через середину данных.

В уравнении 4.8 представлена функция издержек гребневой регрессии<sup>9</sup>.

#### 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  $\theta_0$  не регуляризирован (сумма начинается с  $i = 1$ , не 0). Если мы объявим  $w$  как весовой вектор признаков (от  $\theta_1$  до  $\theta_n$ ), тогда член регуляризации просто равен  $\frac{1}{2}(\|w\|_2)^2$ , где  $\|\cdot\|_2$  представляет норму  $\ell_2$  весового вектора<sup>10</sup>. Для градиентного спуска нужно просто добавить  $\alpha w$  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 ряде линейных данных с использованием разных значений  $\alpha$ .

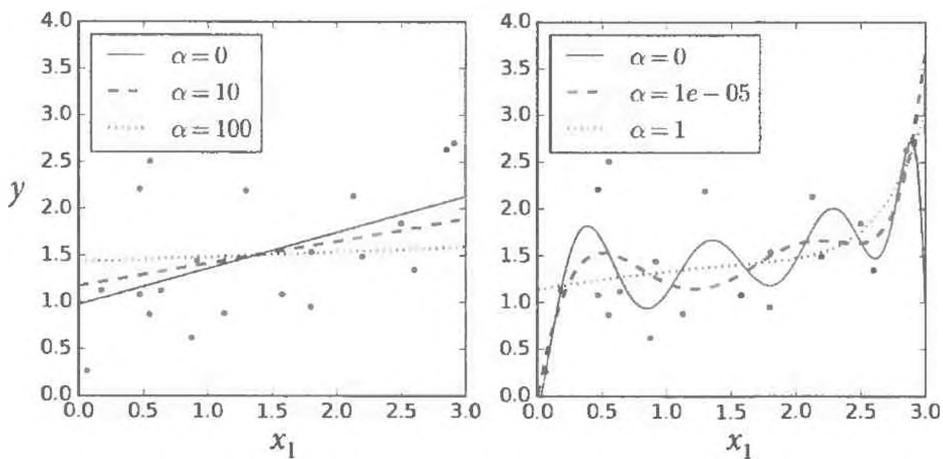


Рис. 4.17. Линейная модель (слева) и полиномиальная модель (справа) с различными уровнями регуляризации гребневой регрессии

<sup>9</sup> 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.

<sup>10</sup> 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  $\alpha$  ведет к получению более ровных (менее предельных, более рациональных) прогнозов, т.е. сокращению дисперсии модели, но увеличению ее смещения.

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 показано решение в аналитическом виде, где  $A$  — это единичная матрица<sup>11</sup> (*identity matrix*) размером  $(n + 1) \times (n + 1)$  за исключением нуля в левой верхней ячейке, соответствующего члену смещения.

#### Уравнение 4.9. Решение в аналитическом виде для гребневой регрессии

$$\hat{\theta} = (X^T X + \alpha A)^{-1} X^T y$$

Вот как выполнять гребневую регрессию с помощью Scikit-Learn, используя решение в аналитическом виде (вариант уравнения 4.9, в котором применяется метод разложения матрицы Андре-Луи Холецкого):

```
>>> from import Ridge  
>>> ridge_reg = Ridge(alpha= , solver="cholesky")  
>>> ridge_reg.fit(X, y)  
>>> ridge_reg.predict([[ ]])  
array([1.55071465])
```

И с применением стохастического градиентного спуска<sup>12</sup>:

<sup>11</sup> Квадратная матрица, полная нулей, за исключением единиц на главной диагонали (от левого верхнего до правого нижнего угла).

<sup>12</sup> 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 (<https://homl.info/12>).

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([1.47012588])
```

Гиперпараметр `penalty` устанавливает используемый тип члена регуляризации. Стока "l2" указывает на то, что алгоритм SGD должен добавлять к функции издержек член регуляризации, равный одной второй квадрата нормы  $\ell_2$  весового вектора: это просто гребневая регрессия.

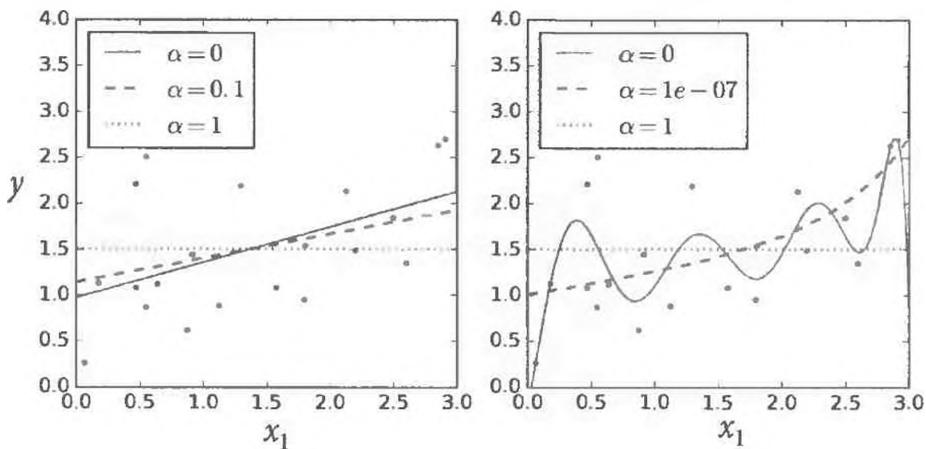
## Лассо-регрессия

*Регрессия методом наименьшего абсолютного сокращения и выбора (Least Absolute Shrinkage and Selection Operator (Lasso) Regression)*, называемая просто **лассо-регрессией**, представляет собой еще одну регуляризированную версию линейной регрессии: в частности как гребневая регрессия она добавляет к функции издержек член регуляризации, но вместо одной второй квадрата нормы  $\ell_2$  весового вектора использует норму  $\ell_1$  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  $\alpha$ .



**Рис. 4.18. Линейная модель (слева) и полиномиальная модель (справа) с различными уровнями регуляризации лассо-регрессии**

Важной характеристикой лассо-регрессии является то, что она стремится полностью исключить веса наименее важных признаков (т.е. устанавливает их в 0). Например, пунктирная линия на графике справа на рис. 4.18 (с  $\alpha = 10^{-7}$ ) выглядит квадратичной и довольно вытянутой в линию: 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).

Понять, почему так происходит, можно с помощью рис. 4.19: оси представляют два параметра модели, а фоновые контуры — различные функции потерь. На левом верхнем графике контуры представляют потерю  $\ell_1 (|\theta_1| + |\theta_2|)$ , которая линейно снижается с приближением к любой оси. Например, если инициализировать параметры модели как  $\theta_1 = 2$  и  $\theta_2 = 0.5$ , то выполнение градиентного спуска одинаково уменьшит оба параметра (как показано с помощью желтой пунктирной линии); следовательно, параметр  $\theta_2$  достигнет 0 первым (т.к. он изначально был ближе к 0). Затем градиентный спуск катится вниз по желобу до тех пор, пока не достигает  $\theta_1 = 0$  (с небольшими колебаниями, поскольку градиенты  $\ell_1$  никогда не приближаются к 0: для каждого параметра они составляют либо -1, либо 1).

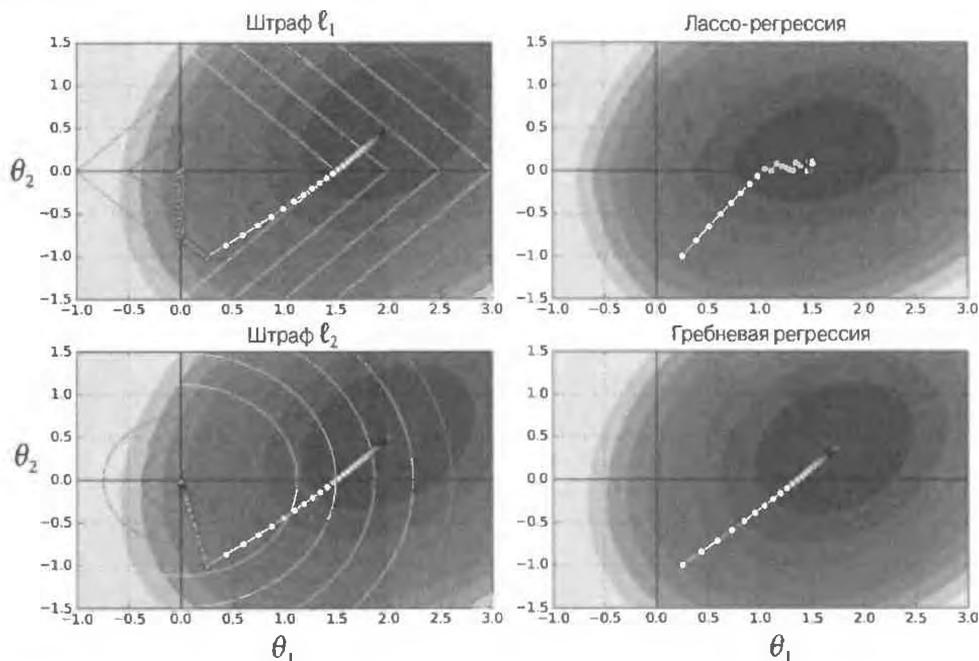


Рис. 4.19. Сравнение регуляризации при лассо-регрессии и гребневой регрессии

На правом верхнем графике контуры представляют функцию издержек для лассо-регрессии (т.е. функцию издержек MSE плюс потерю  $\ell_1$ ). Небольшие белые кружочки показывают путь, который градиентный спуск выбирает для оптимизации ряда параметров модели, инициализированных вблизи  $\theta_1 = 0.25$  и  $\theta_2 = -1$ : снова обратите внимание на то, как путь быстро достигает  $\theta_2 = 0$ , затем катится вниз по желобу и заканчивает колебаниями вокруг глобального оптимума (представленного красным квадратом). Если увеличить  $\alpha$ , тогда глобальный оптимум переместится влево вдоль желтой пунктирной линии, а если уменьшить  $\alpha$ , то глобальный оптимум переместится вправо (в этом примере оптимальные параметры для нерегуляризованной функции издержек MSE выглядят как  $\theta_1 = 2$  и  $\theta_2 = 0.5$ ).

На нижних двух графиках демонстрируется то же самое, но со штрафом  $\ell_2$ . На левом нижнем графике можно видеть, что с ростом расстояния от начала координат потеря  $\ell_2$  уменьшается, поэтому градиентный спуск просто выбирает прямолинейный путь по направлению к данной точке. На правом нижнем графике контуры представляют функцию издержек для гребневой регрессии (т.е. функцию издержек MSE плюс потерю  $\ell_2$ ). У лассо-регрессии есть два главных отличия. Во-первых, по мере приближения к глобальному оптимуму градиенты становятся меньше, так что градиентный спуск естественным образом замедляется, содействуя сходжению (поскольку нет никаких колебаний). Во-вторых, при увеличении  $\alpha$  оптимальные параметры (представленные красным квадратом) становятся все ближе и ближе к началу координат, но никогда не исключаются полностью.



Чтобы устранить колебания градиентного спуска возле оптимума в конце, когда используется лассо-регрессия, необходимо постепенно уменьшать скорость обучения (он по-прежнему будет совершать колебания возле оптимума, но шаги станут все меньше и меньше, так что алгоритм сойдется).

Функция издержек лассо-регрессии не является дифференцируемой при  $\theta_i = 0$  (для  $i = 1, 2, \dots, n$ ), но градиентный спуск все равно хорошо работает, если взамен применять вектор-субградиент (*subgradient vector*)  $\mathbf{g}$ <sup>13</sup>, когда любое  $\theta_i = 0$ . В уравнении 4.11 показано уравнение вектора-субградиента, которое можно использовать для градиентного спуска с функцией издержек лассо-регрессии.

<sup>13</sup> Вы можете представлять вектор-субградиент в недифференцируемой точке как промежуточный вектор между векторами-градиентами вокруг этой точки.

## Уравнение 4.11. Вектор-субградиент лассо-регрессии

$$g(\theta, J) = \nabla_{\theta} \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix}, \quad \text{где } \text{sign}(\theta_i) = \begin{cases} -1, & \text{если } \theta_i < 0 \\ 0, & \text{если } \theta_i = 0 \\ +1, & \text{если } \theta_i > 0 \end{cases}$$

Ниже приведен небольшой пример Scikit-Learn, в котором применяется класс Lasso:

```
>>> from ... import Lasso
>>> lasso reg = Lasso(alpha=...)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[...]])
array([1.53788174])
```

Обратите внимание, что взамен вы могли бы использовать SGDRegressor (penalty="l1").

## Эластичная сеть

Эластичная сеть — это серединная точка между гребневой регрессией и лассо-регрессией. Член регуляризации представляет собой просто смесь членов регуляризации гребневой регрессии и лассо-регрессии, к тому же можно также 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 полезными будут лишь несколько признаков, 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, потому что лассо-регрессия может работать с перебоями, 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 import ElasticNet
>>> elastic_net = ElasticNet(alpha=..., l1_ratio=...)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[...]])
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.

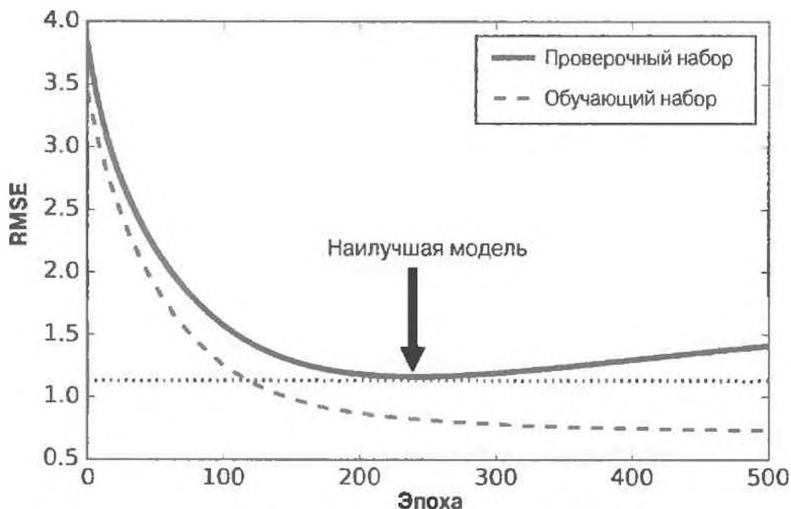


Рис. 4.20. Регуляризация с ранним прекращением

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 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 Джекфри Хинтон назвал его “превосходным бесплатным завтраком”.



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           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(max_iter=1, tol=-np.infty, 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, y_val_predict)
    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 продолжает обучение с места, которое 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 оценка вероятности больше 50%, тогда модель прогнозирует, что образец принадлежит к данному классу (называемому *положительным классом* и помечаемому “1”), а иначе — что не принадлежит (т.е. относится к *отрицательному классу*, помеченному “0”). Это делает ее *двоичным классификатором*.

### Оценивание вероятностей

Так каким же образом работает логистическая регрессия? Подобно линейной регрессионной модели логистическая регрессионная модель подсчитывает взвешенную сумму входных признаков (плюс член смещения), но вместо выдачи результата напрямую, как делает линейная регрессионная модель, она выдает *логистику (logistic)* результата (уравнение 4.13).

**Уравнение 4.13. Логистическая регрессионная модель оценивает вероятность (векторизованная форма)**

$$\hat{p} = h_{\theta}(x) = \sigma(x^T \theta)$$

Логистика, 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.

**Уравнение 4.14. Логистическая функция**

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

После того как логистическая регрессионная модель оценила вероятность  $\hat{p} = h_{\theta}(x)$  принадлежности образца  $x$  положительному классу, она может легко выработать прогноз  $\hat{y}$  (уравнение 4.15).

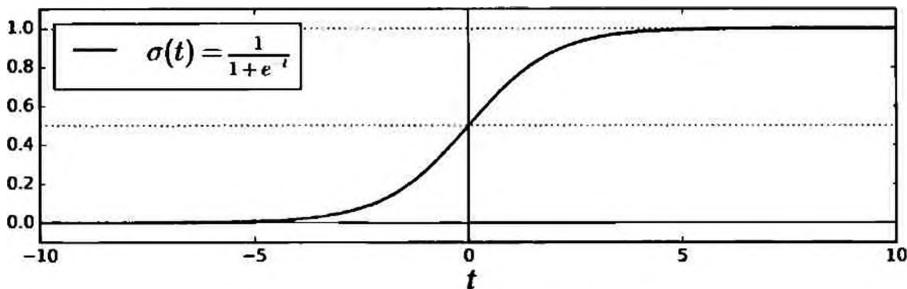


Рис. 4.21. Логистическая функция

**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 значение  $x^T\theta$  положительное, и 0, если оно отрицательное.



Показатель  $t$  часто называют *логитом* (*logit*). Имя происходит из того факта, что логит-функция, определяемая как  $\text{logit}(p) = \log(p / (1 - p))$ , является инверсией логистической функции. На самом деле, если вы вычислите логит оценки вероятности  $p$ , то обнаружите, что результатом будет  $t$ . Логит также называют *логарифмом коэффициента перевеса* (*log-odds*), поскольку он представляет собой логарифм соотношения между оценкой вероятности для положительного класса и оценкой вероятности для отрицательного класса.

## Обучение и функция издержек

Теперь вы знаете, каким образом логистическая регрессионная модель оценивает вероятности и вырабатывает прогнозы. Но как ее обучить? Целью обучения является установка вектора параметров  $\theta$  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 оценка вероятности 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, 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  $\theta$ , которое сводит к минимуму такую функцию издержек (не существует эквивалента нормального уравнения). 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  $\theta_j$  имеют вид, представленный в уравнении 4.18.

#### Уравнение 4.18. Частные производные функции издержек логистической регрессии

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{1}{m} \sum_{i=1}^m (\sigma(\theta^T x^{(i)}) - y^{(i)}) x_j^{(i)}$$

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 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).

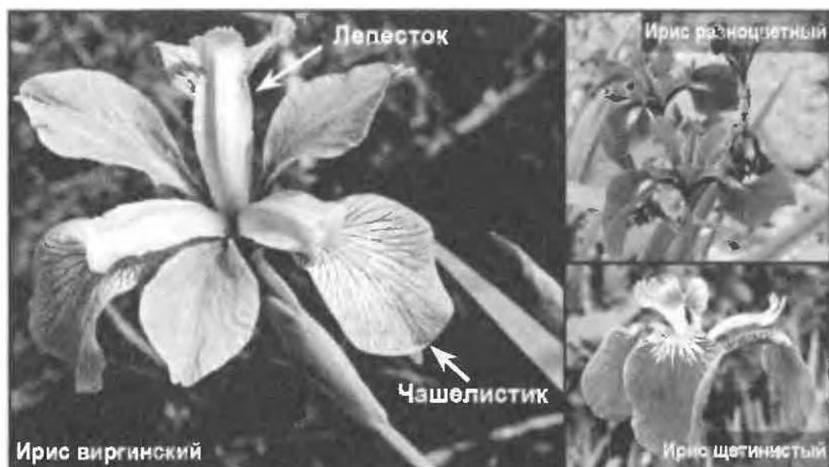


Рис. 4.22. Цветы ирисов трех видов<sup>14</sup>

Попробуем построить классификатор для выявления вида ириса виргинского на основе только признака ширины лепестка (petal width). Сначала загрузим данные:

<sup>14</sup> 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.

```

>>> from      import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target', 'target_names', 'DESCR', 'feature_names', 'filename']
>>> X = iris["data"][:, : ]    # ширина лепестка
>>> y = (iris["target"] == ).astype(np.int) # 1, если ирис
                                         # виргинский, иначе 0

```

Теперь обучим логистическую регрессионную модель:

```

from           import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)

```

Давайте взглянем на оценки вероятности модели для цветов, ширина лепестка которых варьируется от 0 до 3 сантиметров (рис. 4.23)<sup>15</sup>:

```

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. Оценки вероятности и граница решений

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

<sup>15</sup> Функция `reshape()` из NumPy позволяет указывать одно измерение как -1, что означает "неопределенное": его значение выводится из длины массива и остальных изменений.

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 какой угодно 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], [1.9]])
array([1, 0])
```

На рис. 4.24 показан тот же самый набор данных, но теперь 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 линейна<sup>16</sup>. 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.

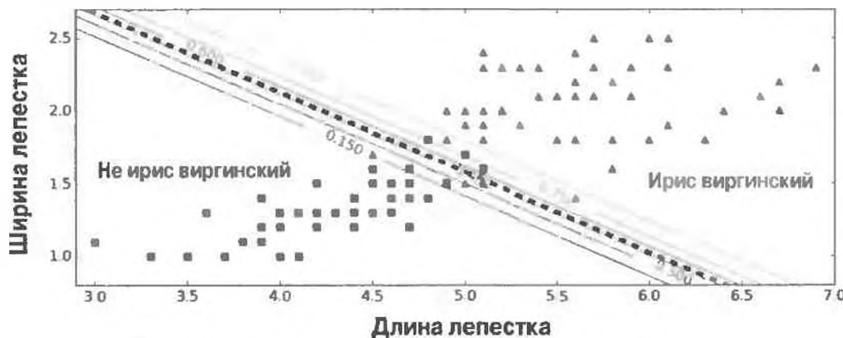


Рис. 4.24. Линейная граница решений

<sup>16</sup> Она представляет собой набор точек  $x$ , так что справедливо уравнение  $\theta_0 + \theta_1x_1 + \theta_2x_2 = 0$ , определяющее прямую линию.

Подобно другим линейным моделям логистические регрессионные модели могут быть регуляризованы с применением штрафов  $\ell_1$  или  $\ell_2$ . В действительности библиотека Scikit-Learn по умолчанию добавляет штраф  $\ell_2$ .



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 регуляризируется.

## Многопараметрическая 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 **многопараметрическая логистическая регрессия (Softmax Regression)**, или **полиномиальная логистическая регрессия (Multinomial Logistic Regression)**.

Идея проста: имея образец  $x$ , многопараметрическая логистическая регрессионная модель сначала вычисляет сумму очков  $s_k(x)$  для каждого класса  $k$  и затем оценивает вероятность каждого класса, применяя к суммам очков **многопараметрическую логистическую функцию (softmax function)**, также называемую **нормализованной экспоненциальной (normalized exponential)** функцией. Уравнение для подсчета  $s_k(x)$  должно выглядеть знакомым, т.к. оно точно такое же, как для прогноза линейной регрессии (уравнение 4.19).

**Уравнение 4.19. Многопараметрическая логистическая сумма очков для класса  $k$**

$$s_k(x) = x^T \theta^{(k)}$$

Обратите внимание, что каждый класс имеет собственный отдельный вектор параметров  $\theta^{(k)}$ . Все эти векторы обычно хранятся как строки в матрице **параметров  $\Theta$** .

После подсчета сумм очков каждого класса для образца  $x$  можно оценить вероятность  $\hat{p}_k$ , что образец принадлежит классу  $k$ , прогнав суммы очков через многопараметрическую логистическую функцию (уравнение 4.20): она вычисляет экспоненту каждой суммы очков и затем нормализует их (путем деления на сумму всех экспонент). Суммы очков в общем случае называются логитами или логарифмами коэффициента перевеса (хотя фактически они являются ненормализованными логарифмами коэффициента перевеса).

## Уравнение 4.20. Многопеременная логистическая функция

$$\hat{p}_k = \sigma(s(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

В приведенном уравнении:

- $K$  — количество классов;
- $s(\mathbf{x})$  — вектор, содержащий суммы очков каждого класса для образца  $\mathbf{x}$ ;
- $\sigma(s(\mathbf{x}))_k$  — оценка вероятности 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 оценкой вероятности (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

$$\hat{y} = \operatorname{argmax}_k \sigma(s(\mathbf{x}))_k = \operatorname{argmax}_k s_k(\mathbf{x}) = \operatorname{argmax}_k \left( (\theta^{(k)})^\top \mathbf{x} \right)$$

Операция *argmax* возвращает значение переменной, которая обращает функцию в максимум. В приведенном уравнении она возвращает значение  $k$ , обращающее в максимум оценку вероятности  $\sigma(s(\mathbf{x}))_k$ .



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. Задача заключается в том, чтобы получить модель, которая дает оценку в виде высокой вероятности для целевого класса (и, следовательно, низкую вероятность для остальных классов). Доведение до минимума функции издержек, которая называется *перекрестной энтропией* (*cross entropy*) и показана в уравнении 4.22, должно решить указанную задачу, поскольку она штрафует модель, когда та

дает оценку в виде низкой вероятности для целевого класса. Перекрестная энтропия часто используется для измерения, насколько хорошо набор оценок вероятности классов соответствует целевым классам.

#### Уравнение 4.22. Функция издержек в форме перекрестной энтропии

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log (\hat{p}_k^{(i)})$$

В данном уравнении:

- $y_k^{(i)}$  — целевая вероятность, что  $i$ -тый образец принадлежит классу  $k$ . В общем случае она равна 1 или 0 в зависимости от того, относится образец к классу или нет.

Обратите внимание, что при наличии только двух классов ( $K = 2$ ) такая 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 три 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). Дополнительные детали я даю в своем видеоролике (<https://homl.info/xentropy>).

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 $k$

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) x^{(i)}$$

Теперь вы можете рассчитать вектор-градиент для каждого класса, после чего применить градиентный спуск (или любой другой алгоритм оптимизации) для нахождения матрицы параметров  $\Theta$ , 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 остальных”, когда он обучается более чем на 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  $\ell_2$ , 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([[6.38014896e-07, 5.74929995e-02, 9.42506362e-01]])
```

На рис. 4.25 показаны результирующие границы решений, представленные с помощью фоновых цветов. Обратите внимание, что границы решений

между любыми двумя классами линейны. Там 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, с которым связана оценка вероятности ниже 50%. Например, в точке, где встречаются все границы решений, все классы имеют одинаковые оценки вероятности 33%.

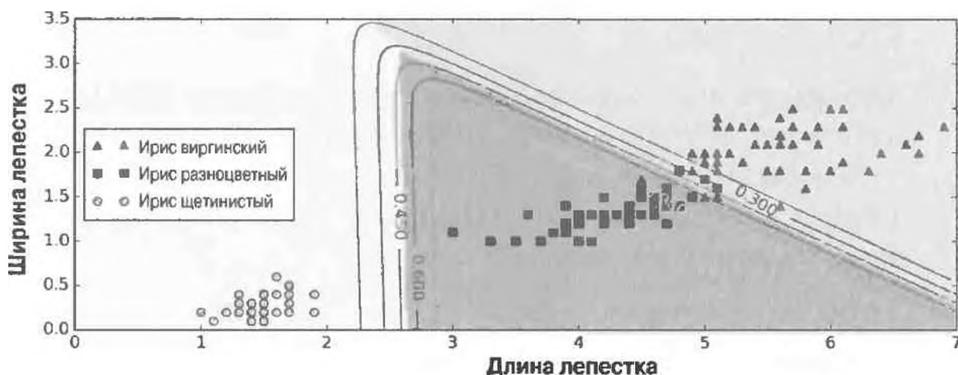


Рис. 4.25. Границы решений многопараметрической логистической регрессии

## Упражнения

1. Какой алгоритм обучения на основе 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 график ошибки проверки на каждой эпохе. Если вы заметите, что ошибка проверки последовательно растет, тогда что вероятнее 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  $\alpha$  или уменьшить его?
10. Почему вы захотели бы применять:
  - а) гребневую регрессию вместо обыкновенной линейной регрессии (т.е. без какой-либо регуляризации)?
  - б) лассо-регрессию вместо гребневой регрессии?
  - в) 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 мощная и универсальная модель МО, 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 наборов данных небольших или средних размеров.

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 раскрывать 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)*.

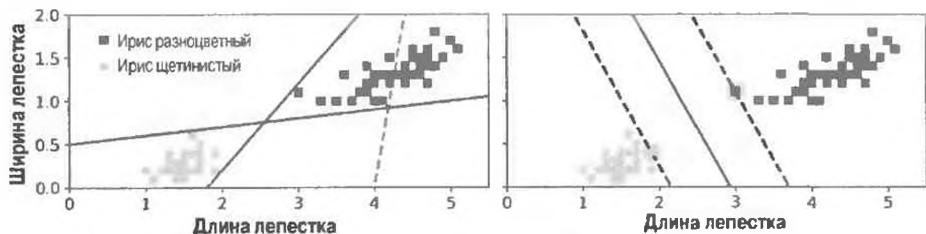


Рис. 5.1. Классификация с широким зазором

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 на графике справа выглядит гораздо лучше.

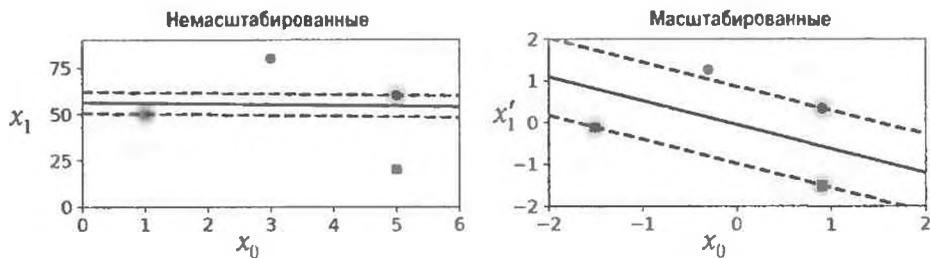


Рис. 5.2. Чувствительность к масштабам признаков

## Классификация с мягким зазором

Если мы строго зафиксируем, что все образцы обязаны находиться 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 обучающим образцом, расположенным вдали от основной группы.

тельным 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.

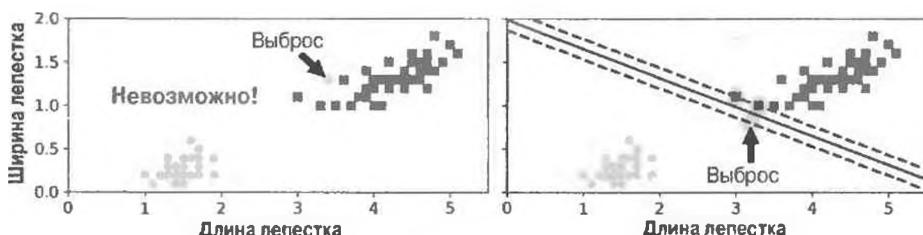


Рис. 5.3. Чувствительность к выбросам жесткого зазора

Чтобы избежать таких проблем, применайте 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)*.

При создании модели SVM с использованием Scikit-Learn мы можем указывать несколько гиперпараметров, одним из которых является  $C$ . Если мы установим его в низкое значение, то в итоге получим модель слева на рис. 5.4. В случае установки в высокое значение мы получим модель справа. Нарушения зазора — это плохо. Обычно лучше, чтобы их было мало. Однако в рассматриваемом случае модель слева имеет много нарушений зазора, но вероятно лучше обобщается.

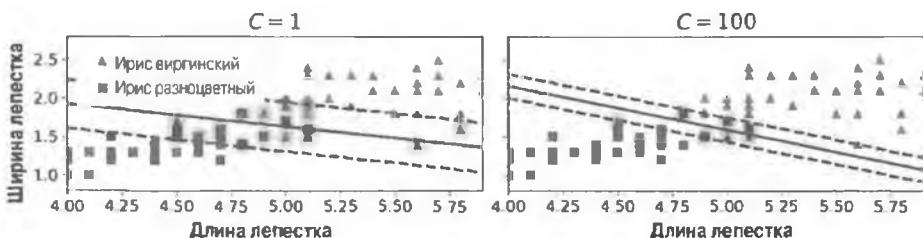


Рис. 5.4. Больший зазор (слева) или меньше нарушений зазора (справа)



Если ваша модель SVM переобучается, тогда можете попробовать ее регуляризировать 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:

```
import          as
from           import datasets
from           import Pipeline
from           import StandardScaler
from           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)
```

Результирующая модель представлена слева на рис. 5.4.

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.

Вместо класса `LinearSVC` мы могли бы применить класс `SVC` с линейным ядром. При создании модели `SVC` (*Support Vector Clustering — кластеризация методом опорных векторов*) мы записали бы `SVC(kernel="linear", C=1)`. Или же мы могли бы использовать класс `SGDClassifier` в виде `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 решения задач динамической классификации или для обработки гигантских наборов данных, которые не умещаются в памяти (внешнее обучение).



Класс `LinearSVC` регуляризирует 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 эффективности 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 во многих случаях работают на удивление хорошо, многочисленные 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 единственным 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.

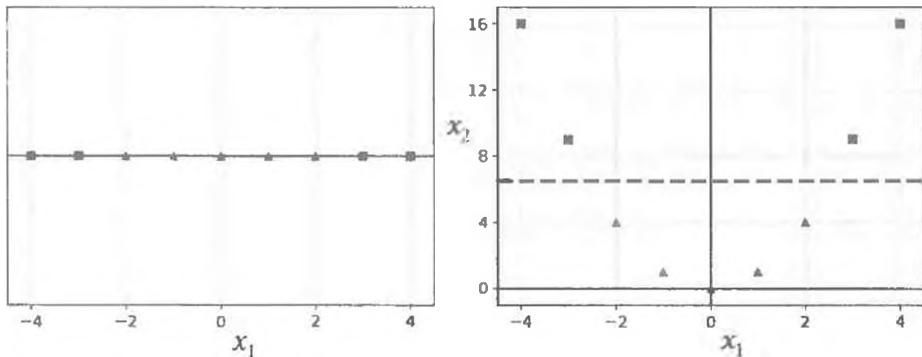


Рис. 5.5. Добавление признаков для превращения набора данных в линейно сепарабельный

Чтобы визуализировать 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 прием на moons — маленьком наборе данных для двоичной классификации, где точки данных имеют форму двух перемежающихся полукругов (рис. 5.6). Сгенерировать набор данных moons можно посредством функции make\_moons():

```
from import make_moons
from import Pipeline
from import PolynomialFeatures

X, y = make_moons(n_samples=100, noise=0.15)
polynomial_svm_clf = Pipeline([
    ("poly_features", PolynomialFeatures(degree=3)),
    ("scaler", StandardScaler()),
    ("svm_clf", LinearSVC(C=10, loss="hinge"))
])
polynomial_svm_clf.fit(X, y)
```

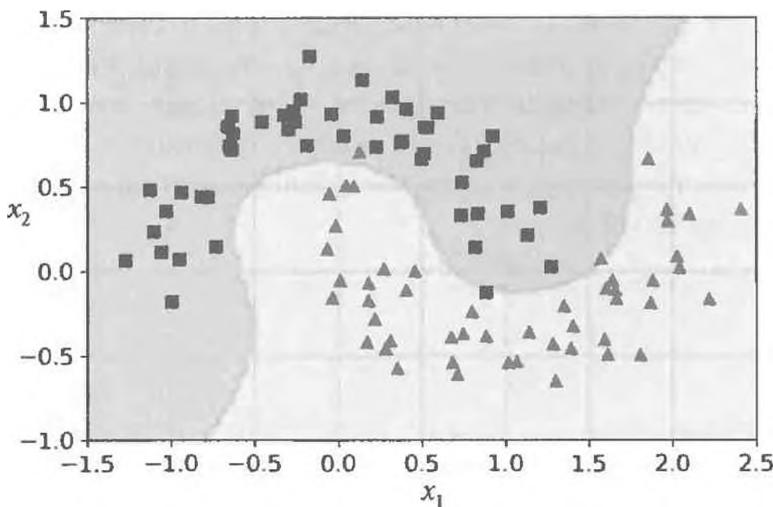


Рис. 5.6. Линейный классификатор SVM, использующий полиномиальные признаки

## 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 такой метод не способен

справляться с очень сложными наборами данных, а при высокой полиномиальной степени он 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. Ядерный трюк 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, комбинаторный взрыв количества признаков отсутствует, поскольку в действительности никакие признаки не добавляются. Ядерный трюк выполняется классом SVC. Давайте проверим его на наборе данных moons:

```
from               import SVC
poly_kernel_svm_clf = Pipeline([
    ("scaler", StandardScaler()),
    ("svm_clf", SVC(kernel="poly", degree= , coef0= , C= ))
])
poly_kernel_svm_clf.fit(X, y)
```

Приведенный выше код обучает классификатор SVM, использующий полиномиальное ядро третьей степени. Он представлен слева на рис. 5.7. Справа показан еще один классификатор SVM, который применяет полиномиальное ядро десятой 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.

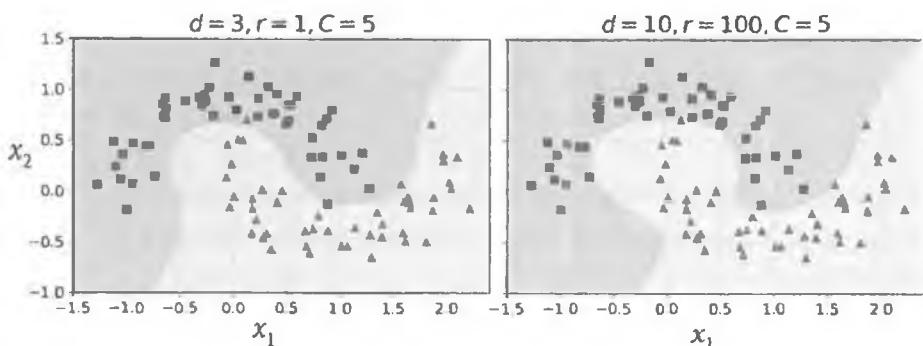


Рис. 5.7. Классификаторы SVM с полиномиальным ядром



Распространенный подход к поиску правильных значений гиперпараметров заключается в использовании решетчатого поиска (см. главу 2). Часто быстрее сначала сделать очень грубый решетчатый поиск и 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.

## Признаки 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  $y = 0.3$  (уравнение 5.1).

### Уравнение 5.1. Гауссова функция RBF

$$\phi_y(\mathbf{x}, \mathbf{t}) = \exp(-y\|\mathbf{x} - \mathbf{t}\|^2)$$

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, теперь он 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.

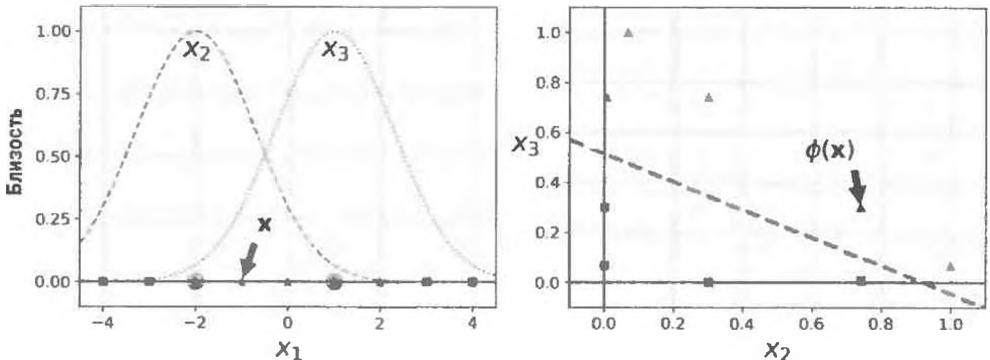


Рис. 5.8. Признаки близости, использующие гауссову функцию RBF

## 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, позволяя получить похожий результат, как если бы добавлялись многочисленные признаки близости. Давайте испытаем класс SVC с гауссовым ядром RBF (*Gaussian RBF kernel*):

```
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  $\gamma$  и  $C$ . Увеличение  $\gamma$  приводит к сужению колоколообразной кривой (см. рис. 5.8), в результате чего сфера влияния каждого образца уменьшается: граница решений становится более неравномерной, шевелящейся 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,  $\gamma$  действует аналогично гиперпараметру регуляризации: если ваша модель переобучается, тогда вы должны уменьшить значение  $\gamma$ , а если недообучается, то увеличить его (подобно гиперпараметру  $C$ ).

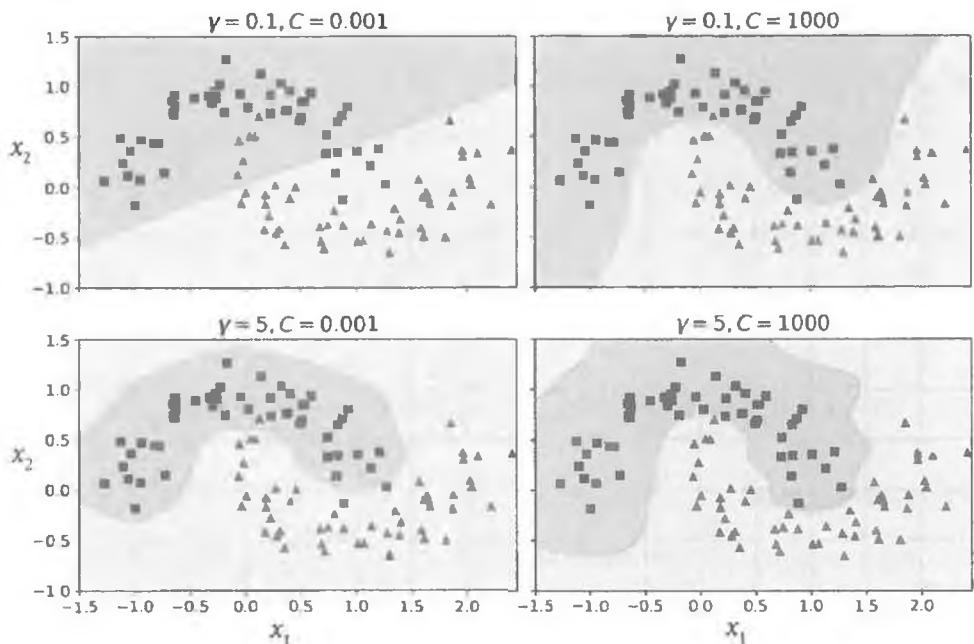


Рис. 5.9. Классификаторы SVM, использующие ядро RBF

Существуют и другие ядра, но они применяются гораздо реже. 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? Вот эмпирическое правило: 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 адаптированы к структуре данных вашего обучающего набора.

## Вычислительная сложность

Класс LinearSVC основан на библиотеке liblinear, которая реализует оптимизированный алгоритм (<https://homl.info/13>) для линейных методов SVM<sup>1</sup>. 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, что управляет гиперпараметром допуска  $\epsilon$  (называется `tol` в Scikit-Learn). Большинству задач классификации подходит стандартный допуск.

Класс SVC основан на библиотеке libsvm, которая реализует алгоритм (<https://homl.info/14>), поддерживающий ядерный трюк<sup>2</sup>. 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 обучающих наборов небольших или средних размеров. 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 образец. 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
LinearSVC	$O(m \times n)$	Нет	Да	Нет
SGDClassifier	$O(m \times n)$	Да	Да	Нет
SVC	от $O(m^2 \times n)$ до $O(m^3 \times n)$	Нет	Да	Да

<sup>1</sup> Чин-Йен Лин и др., *A Dual Coordinate Descent Method for Large-Scale Linear SVM* (Метод двойного покоординатного спуска для крупномасштабного линейного метода опорных векторов), *Proceedings of the 25th International Conference on Machine Learning* (работы 25-й международной конференции по машинному обучению) (2008 г.): с. 408–415.

<sup>2</sup> Джон Платт, *Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines* (Последовательная минимальная оптимизация: быстрый алгоритм для обучения моделей на основе методов опорных векторов), *Microsoft Research technical report*, 21 апреля 1998 г. (<https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-98-14.pdf>)

## Регрессия SVM

Как упоминалось ранее, алгоритм SVM довольно универсален: он поддерживает не только линейную и нелинейную классификацию, но также линейную и нелинейную регрессию. Чтобы использовать методы SVM для регрессии, а не классификации, потребуется обратить цель. Взамен 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 вместе с ограничением нарушений зазора (т.е. образцов *вне* полосы). Ширина полосы управляется гиперпараметром  $\epsilon$ . 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 ( $\epsilon = 0.5$ ).

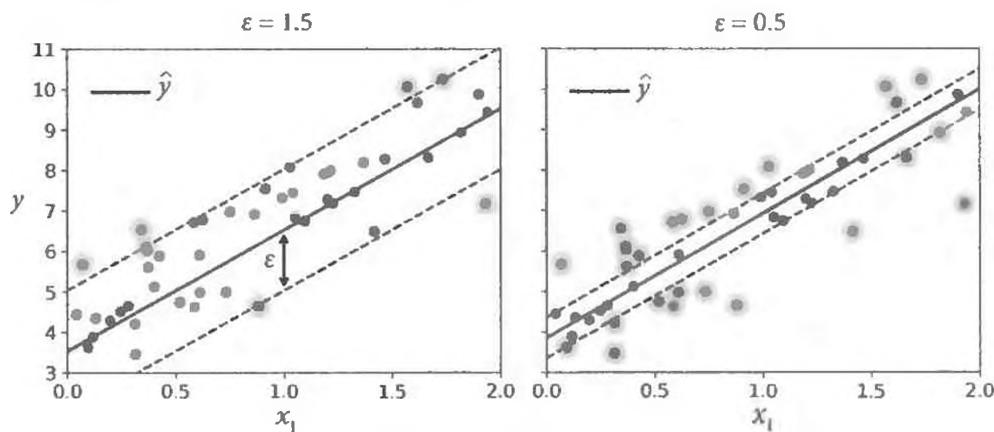


Рис. 5.10. Регрессия SVM

Добавление дополнительных обучающих образцов внутри зазора не влияет на прогнозы модели; соответственно говорят, что модель *нечувствительна* к  $\epsilon$ .

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           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 переводят как “кернелизация” — Примеч. пер.). Например, на рис. 5.11 демонстрируется регрессия SVM на случайному квадратичном обучающем наборе, использующая полиномиальное ядро второго порядка. На графике слева производилось немного регуляризации (т.е. крупное значение C), а на графике справа — гораздо больше регуляризации (т.е. небольшое значение C).

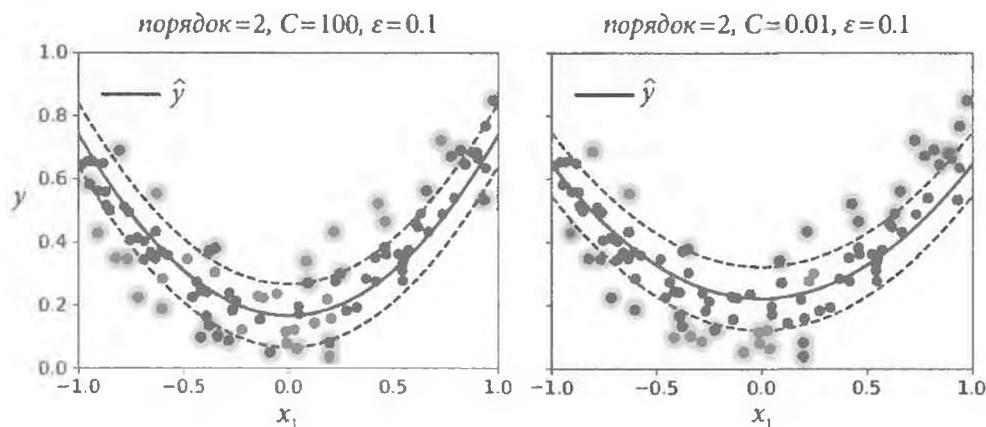


Рис. 5.11. Регрессия SVM, применяющая полиномиальное ядро второго порядка

Показанный ниже код использует класс SVR (поддерживающий ядерный трюк) из Scikit-Learn для порождения модели, которая представлена слева на рис. 5.11:

```
from           import SVR
svr_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svr_poly_reg.fit(X, y)
```

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).



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  $\theta$ , включающий член смещения  $\theta_0$  и входные веса признаков от  $\theta_1$  до  $\theta_n$ , а затем добавление ко всем образцам входного смещения  $x_0 = 1$ . В этой главе мы будем применять 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  $w^T x + b = w_1 x_1 + \dots + w_n x_n + b$ . 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

$$\hat{y} = \begin{cases} 0, & \text{если } w^T x + b < 0, \\ 1, & \text{если } w^T x + b \geq 0. \end{cases}$$

На рис. 5.12 показана функция решения, которая соответствует модели слева на рис. 5.4: это двумерная плоскость, т.к. набор данных имеет два признака (ширина лепестка и длина лепестка). Граница решений представляет собой множество точек, где функция решения равна 0: это пересечение двух плоскостей, которое является прямой (изображена как толстая сплошная линия)<sup>3</sup>.

<sup>3</sup> В более общих чертах, когда имеется  $n$  признаков, функция решения представляет собой  $n$ -мерную гиперплоскость, а граница решений —  $(n - 1)$ -мерную гиперплоскость.

—  $h = 0$   
 - - -  $h = \pm 1$

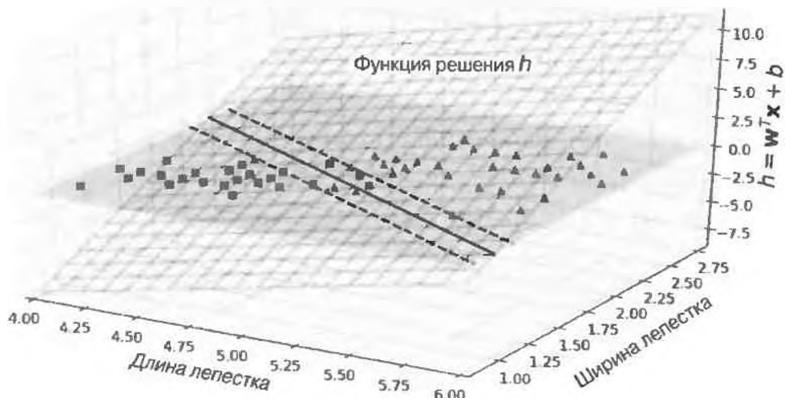


Рис. 5.12. Функция решения для набора данных об ирисах

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  $w$  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,  $\|w\|$ . Если мы разделим наклон на 2, тогда точки, в которых функция решения равна  $\pm 1$ , 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  $w$ , тем шире зазор.

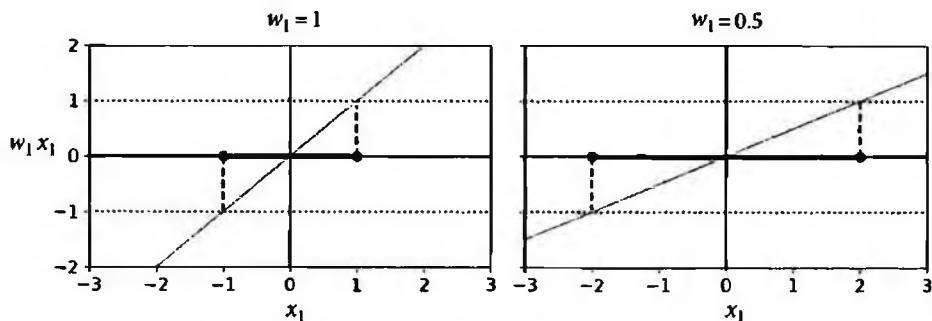


Рис. 5.13. Меньший вектор весов приводит к более широкому зазору

Итак, мы хотим довести до максимума  $\|w\|$ , чтобы получить широкий зазор. 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  $t^{(i)}(w^T x^{(i)} + b) \geq 1$  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

$$\begin{aligned} & \text{минимизировать}_{w,b} \frac{1}{2} w^T w \\ & \text{при условии } t^{(i)}(w^T x^{(i)} + b) \geq 1 \text{ для } i = 1, 2, \dots, m \end{aligned}$$



Мы минимизируем  $\frac{1}{2} w^T w$ , что равносильно  $\frac{1}{2} \|w\|^2$ , вместо минимизации  $\|w\|$ . На самом деле  $\frac{1}{2} \|w\|^2$  имеет подходящую и простую производную (как раз  $w$ ), в то время как  $\|w\|$  не дифференцируется в точке  $w = 0$ . 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 образца<sup>4</sup>:  $\zeta^{(i)}$  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  $\frac{1}{2} w^T w$  как можно меньшим, чтобы расширить зазор. Именно здесь в игру вступает гиперпараметр  $C$ : он позволяет определить компромисс между указанными двумя целями. Мы получаем задачу условной оптимизации, представленную в уравнении 5.4.

<sup>4</sup> Дзета ( $\zeta$ ) — шестая буква греческого алфавита.

#### Уравнение 5.4. Цель линейного классификатора SVM с мягким зазором

$$\text{минимизировать}_{\mathbf{w}, b, \zeta} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \zeta^{(i)}$$

$$\text{при условии } t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \text{ и } \zeta^{(i)} \geq 0 \text{ для } i = 1, 2, \dots, m$$

#### 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 книги<sup>5</sup>. В уравнении 5.5 дана общая постановка задачи.

#### Уравнение 5.5. Задача квадратичного программирования

$$\text{минимизировать}_{\mathbf{p}} \frac{1}{2} \mathbf{p}^T \mathbf{H} \mathbf{p} + \mathbf{f}^T \mathbf{p}$$

$$\text{при условии } \mathbf{A} \mathbf{p} \leq \mathbf{b}$$

где  $\left| \begin{array}{l} \mathbf{p} \text{ — } n_p\text{-размерный вектор (}n_p\text{ = количество параметров),} \\ \mathbf{H} \text{ — матрица } n_p \times n_p, \\ \mathbf{f} \text{ — } n_p\text{-размерный вектор,} \\ \mathbf{A} \text{ — матрица } n_c \times n_p \text{ (}n_c\text{ = количество ограничений),} \\ \mathbf{b} \text{ — } n_c\text{-размерный вектор.} \end{array} \right.$

Обратите внимание, что выражение  $\mathbf{A} \mathbf{p} \leq \mathbf{b}$  определяет  $n_c$  ограничений:  $\mathbf{p}^T \mathbf{a}^{(i)} \leq b^{(i)}$  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:

<sup>5</sup> Чтобы узнать больше о квадратичном программировании, можете начать с чтения книги Стивена Бойда и Ливена Ванденберга *Convex Optimization* (Выпуклая оптимизация), Cambridge University Press (2004 год; <https://home1.info/15>) или просмотреть курс видеолекций (на английском языке) от Ричарда Брауна (<https://home1.info/16>).

- $n_p = n + 1$ , 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 0;
- $\mathbf{b} = -1$ ,  $n_c$ -размерный вектор, заполненный -1;
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{x}}^{(i)}$ , где  $\dot{\mathbf{x}}^{(i)}$  тождественно  $\mathbf{x}^{(i)}$  с добавочным признаком смещения  $\dot{\mathbf{x}}_0$ .

Один из способов обучения линейного классификатора SVM с жестким зазором предусматривает 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).

Чтобы применить ядерный трюк, мы собираемся рассмотреть другую задачу условной оптимизации.

## Двойственная задача

При наличии задачи условной оптимизации, известной как *прямая задача* (*primal problem*), имеется возможность 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 иметь то же самое решение, что и прямая задача. К счастью, задача SVM удовлетворяет этим условиям<sup>6</sup>, 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).

---

<sup>6</sup> Целевая функция является выпуклой, а ограничения неравенства представляют собой непрерывно дифференцируемые и выпуклые функции.

## Уравнение 5.6. Двойственная форма цели линейного классификатора SVM

$$\text{минимизировать}_{\alpha} \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)\top} \mathbf{x}^{(j)} - \sum_{i=1}^m \alpha^{(i)}$$

при условии  $\alpha^{(i)} \geq 0$  для  $i = 1, 2, \dots, m$

После нахождения вектора  $\hat{\mathbf{a}}$ , сводящего к минимуму уравнение 5.6 (используя решатель QP), 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

$$\hat{\mathbf{w}} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\hat{b} = \frac{1}{n_s} \sum_{i=1}^m \left( t^{(i)} - \hat{\mathbf{w}}^\top \mathbf{x}^{(i)} \right)$$
$$\hat{\alpha}^{(i)} > 0$$

Двойственная задача решается быстрее прямой, когда количество обучающих образцов меньше количества признаков. Что более важно, двойственная задача делает возможным ядерный трюк, в то время как прямая задача — нет. 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 `poops`) и затем обучать линейный классификатор SVM на трансформированном обучающем наборе. В уравнении 5.8 показана полиномиальная отображающая функция (*mapping function*) второй степени  $\phi$ , которую желательно применять.

## Уравнение 5.8. Полиномиальное отображение второй степени

$$\phi(\mathbf{x}) = \phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

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<sup>7</sup> трансформированных векторов (уравнение 5.9).

#### Уравнение 5.9. Ядерный трюк для полиномиального отображения второй степени

$$\begin{aligned}\phi(\mathbf{a})^T \phi(\mathbf{b}) &= \begin{pmatrix} a_1^2 \\ \sqrt{2}a_1a_2 \\ a_2^2 \end{pmatrix}^T \begin{pmatrix} b_1^2 \\ \sqrt{2}b_1b_2 \\ b_2^2 \end{pmatrix} = a_1^2b_1^2 + 2a_1b_1a_2b_2 + a_2^2b_2^2 = \\ &= (a_1b_1 + a_2b_2)^2 = \left( \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}^T \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \right)^2 = (\mathbf{a}^T \mathbf{b})^2\end{aligned}$$

Что скажете? Скалярное произведение трансформированных векторов равно квадрату скалярного произведения исходных векторов:  $\phi(\mathbf{a})^T \phi(\mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$ .

Вот основная суть: если вы примените трансформацию  $\phi$  ко всем обучающим образцам, тогда двойственная задача (см. уравнение 5.6) будет содержать скалярное произведение  $\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(j)})$ . Но если  $\phi$  — полиномиальная трансформация второй степени, определенная в уравнении 5.8, то вы можете заменить это скалярное произведение трансформированных векторов просто на  $(\mathbf{x}^{(i)}^T \mathbf{x}^{(j)})^2$ . Таким образом, вы вообще не нуждаетесь в трансформации обучающих образцов: необходимо всего лишь заменить скалярное произведение в уравнении 5.6 его квадратом. Результат будет абсолютно тем же самым, как если бы вы не поленились трансформировать обучающий набор и затем подогнали к нему какой-то линейный алгоритм SVM, но такой трюк делает весь процесс гораздо эффективнее с вычислительной точки зрения.

Функция  $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a}^T \mathbf{b})^2$  называется *полиномиальным ядром* второй степени. В машинном обучении *ядро (kernel)* — это функция, которая способна

<sup>7</sup> Как объяснялось в главе 4, скалярное произведение двух векторов  $\mathbf{a}$  и  $\mathbf{b}$  обычно обозначается с помощью  $\mathbf{a} \cdot \mathbf{b}$ . Однако в МО векторы часто представлены как векторы-столбцы (т.е. матрицы с единственным столбцом), поэтому скалярное произведение получается вычислением  $\mathbf{a}^T \mathbf{b}$ . Ради согласованности с остальными материалами книги мы будем здесь использовать именно такое обозначение, игнорируя тот факт, что формально его результатом будет одноэлементная матрица, а не скалярное значение.

вычислять скалярное произведение  $\phi(\mathbf{a})^T \phi(\mathbf{b})$ , базируясь только на исходных векторах  $\mathbf{a}$  и  $\mathbf{b}$ , без потребности в вычислении трансформации  $\phi$  (или даже знании о ней). В уравнении 5.10 перечислены самые распространенные ядра.

#### Уравнение 5.10. Распространенные ядра

$$\text{Линейное ядро: } K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \mathbf{b}$$

$$\text{Полиномиальное ядро: } K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \mathbf{b} + r)^d$$

$$\text{Гауссово ядро RBF: } K(\mathbf{a}, \mathbf{b}) = \exp(-\gamma \|\mathbf{a} - \mathbf{b}\|^2)$$

$$\text{Сигмоидальное ядро: } K(\mathbf{a}, \mathbf{b}) = \tanh(\gamma \mathbf{a}^T \mathbf{b} + r)$$

### Теорема Мерсера

Согласно теореме Мерсера (*Mercer's theorem*), если функция  $K(\mathbf{a}, \mathbf{b})$  соблюдает несколько математических условий, называемых условиями Мерсера (например,  $K$  должна быть непрерывной и симметричной в своих аргументах, так что  $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$ , и т.д.), тогда существует функция  $\phi$ , которая отображает  $\mathbf{a}$  и  $\mathbf{b}$  на другое пространство (возможно, с гораздо большей размерностью), такое что  $K(\mathbf{a}, \mathbf{b}) = \phi(\mathbf{a})^T \phi(\mathbf{b})$ . Вы можете использовать  $K$  в качестве ядра, поскольку знаете, что  $\phi$  существует, даже если вам неизвестно, что собой представляет  $\phi$ . В случае гауссова ядра RBF можно показать, что  $\phi$  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 в итоге оказываетесь с уравнениями, которые включают  $\phi(\mathbf{x}^{(i)})$ . На самом деле  $\widehat{\mathbf{w}}$  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  $\widehat{\mathbf{w}}$ ?

Хорошая новость в том, что вы можете включить формулу для  $\widehat{\mathbf{w}}$  из уравнения 5.7 в функцию решения для нового образца  $\mathbf{x}^{(n)}$  и получить уравнение с только скалярными произведениями между входными векторами. Это делает возможным использование ядерного трюка (уравнение 5.11).

### Уравнение 5.11. Выработка прогнозов с помощью параметрически редуцированного метода SVM

$$\begin{aligned} h_{\widehat{\mathbf{W}}, \widehat{b}}(\phi(\mathbf{x}^{(n)})) &= \widehat{\mathbf{W}}^T \phi(\mathbf{x}^{(n)}) + \widehat{b} = \left( \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} \phi(\mathbf{x}^{(i)}) \right)^T \phi(\mathbf{x}^{(n)}) + \widehat{b} = \\ &= \sum_{i=1}^m \widehat{\alpha}^{(i)} t^{(i)} (\phi(\mathbf{x}^{(i)})^T \phi(\mathbf{x}^{(n)})) + \widehat{b} = \\ &= \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \widehat{\alpha}^{(i)} t^{(i)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \widehat{b} \end{aligned}$$

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 применить тот же трюк, чтобы получить член смещения  $\widehat{b}$  (уравнение 5.12).

### Уравнение 5.12. Использование ядерного трюка для вычисления члена смещения

$$\begin{aligned} \widehat{b} &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \widehat{\mathbf{w}}^T \phi(\mathbf{x}^{(i)}) \right) = \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \left( \sum_{j=1}^m \widehat{\alpha}^{(j)} t^{(j)} \phi(\mathbf{x}^{(j)}) \right)^T \phi(\mathbf{x}^{(i)}) \right) = \\ &= \frac{1}{n_s} \sum_{\substack{i=1 \\ \widehat{\alpha}^{(i)} > 0}}^m \left( 1 - t^{(i)} \sum_{\substack{j=1 \\ \widehat{\alpha}^{(j)} > 0}}^m \widehat{\alpha}^{(j)} t^{(j)} K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) \right) \end{aligned}$$

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 реализации динамического классификатора SVM 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, градиентный спуск сходится намного медленнее, чем методы, основанные на QP.

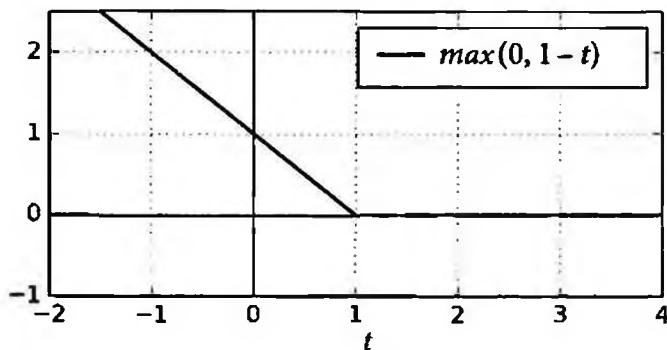
### Уравнение 5.13. Функция издержек линейного классификатора SVM

$$J(\mathbf{w}, b) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^m \max(0, 1 - t^{(i)} (\mathbf{w}^T \mathbf{x}^{(i)} + b))$$

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 или в противном случае 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, как описано в работах *Incremental and Decremental Support Vector Machine Learning* (<https://homl.info/17>)<sup>8</sup> и *Fast Kernel Classifiers with Online and Active Learning* (<https://homl.info/18>)<sup>9</sup>. Такие параметрически редуцированные методы SVM 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 выдавать 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, содержащем миллионы образцов и сотни признаков?
6. Пусть вы обучаете классификатор SVM с ядром RBF, но похоже, что он недообучается на обучающем наборе. Должны вы увеличить или же уменьшить  $\gamma$  (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?

<sup>8</sup> Герт Коффенбергс и Томасо Поджо, *Incremental and Decremental Support Vector Machine Learning* (Инкрементное и декрементное обучение методами опорных векторов), *Proceedings of the 13th International Conference on Neural Information Processing Systems* (2000 г.): с. 388–394.

<sup>9</sup> Антуан Борд и др., *Fast Kernel Classifiers with Online and Active Learning* (Быстрые параметрически редуцированные классификаторы с динамическим и активным обучением), *Journal of Machine Learning Research* 6 (2005 г.): с. 1579–1619.

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 остальных”. 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 A.



# Деревья принятия решений

Подобно методам опорных векторов деревья принятия решений (*Decision Tree*) являются универсальными алгоритмами МО, которые могут заниматься задачами классификации и регрессии, включая даже многовходовые задачи. Они представляют собой 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, каким образом регуляризовать 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 построим одно такое дерево и посмотрим, как оно вырабатывает прогнозы. Следующий код обучает классификатор `DecisionTreeClassifier` на наборе данных `iris` (см. главу 4):

```
from           import load_iris
from           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 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 файл `.dot` в разнообразные форматы, такие как PDF или PNG, с применением инструмента командной строки `dot` из пакета Graphviz<sup>1</sup>. 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.

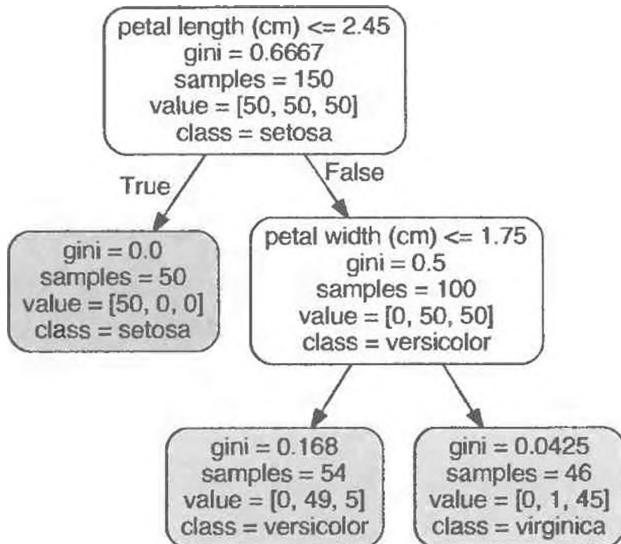


Рис. 6.1. Дерево принятия решений `iris_tree`

<sup>1</sup> 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, вверху): 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 данном 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, длина лепестка которого 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 настолько просто.



Одно из многих качеств деревьев принятия решений заключается в том, что они требуют совсем небольшой подготовки данных. Фактически для них вообще не нужно масштабировать или центрировать признаки.

Атрибут `samples` узла подсчитывает, к скольким обучающим образцам он применяется. Например, 100 обучающих образцов имеют длину лепестка больше 2.45 см (глубина 1, справа), и 54 образца из 100 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. Узел на глубине 2 слева имеет показатель Джини, равный  $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$ .

#### 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 одну границу решений (представленную точечной линией).

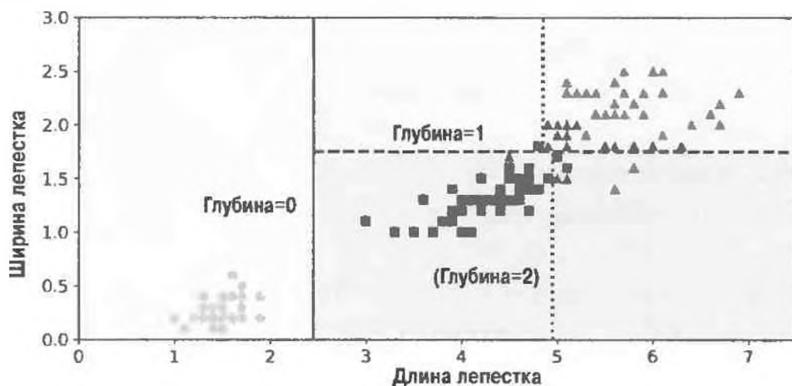


Рис. 6.2. Границы решений в дереве принятия решений

## Интерпретация модели: белый ящик или черный ящик

Деревья принятия решений просты для понимания, а их решения легко интерпретировать. Такие модели часто называют *моделями белого ящика*. Как вы увидите, по контрасту с ними 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, объяснить простыми терминами, почему были выработаны именно такие прогнозы, обычно нелегко. 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. В качестве 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 если вы предложите спрогнозировать класс, то 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 “расщущими” деревьями) библиотека Scikit-Learn использует алгоритм *дерева классификации и регрессии* (*Classification And Regression Tree — CART*). Алгоритм CART сначала расщепляет обучающий набор на два поднабора с применением единственного признака  $k$  и порога  $t_k$  (скажем, “длина лепестка  $\leq 2.45$  см”). Как он выбирает  $k$  и  $t_k$ ? Алгоритм ищет пару  $(k, t_k)$ , которая производит самые чистые поднаборы (взвешенные по их размеру). В уравнении 6.2 представлена функция издержек, которую он пытается минимизировать.

### 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 CART успешно расщепил обучающий набор на два поднабора, он расщепляет полученные поднаборы, используя ту же самую логику, затем расщепляет подподнаборы и продолжает это делать рекурсивно. Алгоритм останавливает рекурсию, когда достигает максимальной глубины (определенной гиперпараметром `max_depth`) или когда не может найти расщепление, которое сократило бы загрязненность. Дополнительные условия остановки управляются рядом других гиперпараметров, 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 последующем 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)* задача<sup>2</sup>: она требует  $O(\exp(m))$  времени, делая задачу трудной для решения даже в случае небольших 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  $O(\log_2(m))$  узлов<sup>3</sup>. 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.

Алгоритм обучения сравнивает все признаки (или меньшее их количество, если установлен параметр `max_features`) на всех образцах в каждом узле. Сравнение всех признаков на всех образцах в каждом узле дает в результате сложность обучения  $O(n \times m \log_2(m))$ . Для небольших обучающих наборов (менее нескольких тысяч образцов) библиотека Scikit-Learn

<sup>2</sup>  $\text{P}$  — набор задач, которые могут быть решены за полиномиальное время.  $\text{NP}$  — набор задач, решения которых могут быть проверены за полиномиальное время.  $\text{NP-трудная (NP-Hard)}$  задача — это такая задача, к которой может быть сокращена любая  $\text{NP}$ -задача за полиномиальное время.  $\text{NP-полнная}$  задача — это и  $\text{NP-задача}$ , и  $\text{NP-трудная}$  задача. Крупным открытым математическим вопросом является выяснение  $\text{P} = \text{NP}$  или нет. Если  $\text{P} \neq \text{NP}$  (что кажется вероятным), тогда для любой  $\text{NP-полней}$  задачи никогда не будет найдено ни одного полиномиального алгоритма (разве что на квантовом компьютере).

<sup>3</sup>  $\log_2$  — это двоичный логарифм. Он равносителен  $\log_2(m) = \log(m) / \log(2)$ .

способна ускорить обучение за счет предварительной сортировки данных (`presort=True`), но это значительно замедлит обучение при более крупных обучающих наборах.

## 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 сообщения<sup>4</sup>: 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  $-\frac{49}{54} \log\left(\frac{49}{54}\right) - \frac{5}{54} \log\left(\frac{5}{54}\right) \approx 0.31$ .

### Уравнение 6.3. Энтропия

$$H_i = - \sum_{k=1}^n p_{i,k} \log(p_{i,k})$$
$$p_{i,k} \neq 0$$

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 деревья<sup>5</sup>.

<sup>4</sup> Сокращение энтропии часто называют *приростом информации* (*information gain*).

<sup>5</sup> Для получения дополнительных сведений ознакомьтесь с интересным анализом Себастьяна Рашки по адресу <https://homl.info/19>.

## Гиперпараметры регуляризации

Деревья принятия решений выдвигают очень мало предположений об обучающих данных (в противоположность, например, линейным моделям, которые очевидным образом предполагают, что данные линейны). Если не связывать древовидную структуру ограничениями, тогда она будет адаптировать себя к обучающим данным, очень близко подгоняясь к ним — на самом деле вполне вероятно допускать 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 ей *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 обязан 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 проверки, такие как критерий  $\chi^2$  (критерий хи-квадрат), 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 значениями гиперпараметров (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.

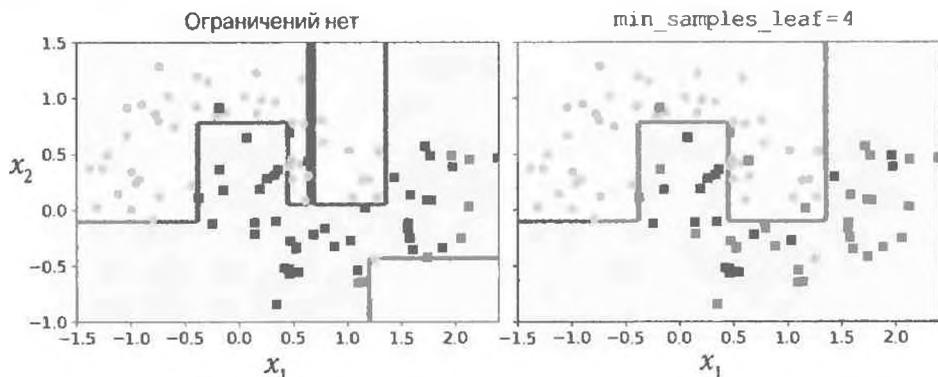


Рис. 6.3. Регуляризация с использованием `min_samples_leaf`

## 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 import DecisionTreeRegressor  
tree_reg = DecisionTreeRegressor(max_depth= )  
tree_reg.fit(X, y)
```

Результирующее дерево изображено на рис. 6.4.

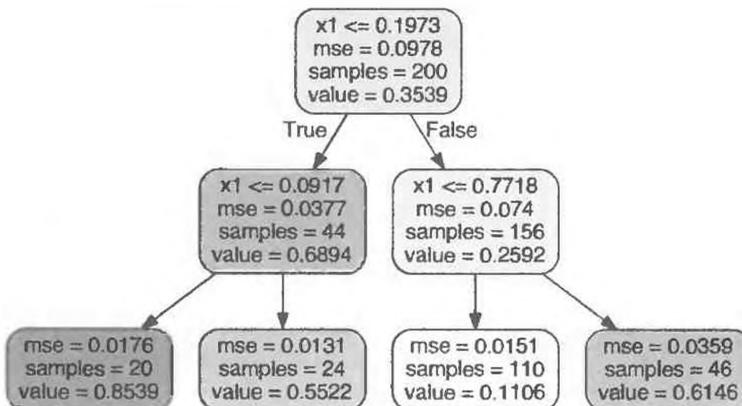


Рис. 6.4. Дерево принятия решений для регрессии

Дерево на рис. 6.4 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 добираетесь до листового узла, который прогнозирует  $\text{value}=0.111$ . Прогноз является просто средним целевым значением 110 обучающих образцов, ассоциированных с этим листовым узлом, и приводит к тому, что среднеквадратическая ошибка (MSE) на таких 110 образцах составляет 0.015.

Прогнозы рассматриваемой модели показаны слева на рис. 6.5. Если вы установите  $\text{max\_depth}=3$ , 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 методом, 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.

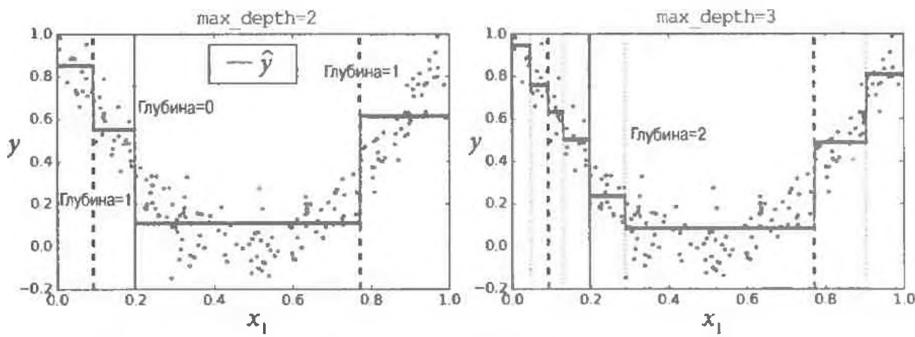


Рис. 6.5. Прогнозы двух регрессионных моделей на основе деревьев принятия решений

#### Уравнение 6.4. Функция издержек алгоритма CART для регрессии

$$J(k, t_k) = \frac{m_{\text{левый}}}{m} \text{MSE}_{\text{левый}} + \frac{m_{\text{правый}}}{m} \text{MSE}_{\text{правый}},$$

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.

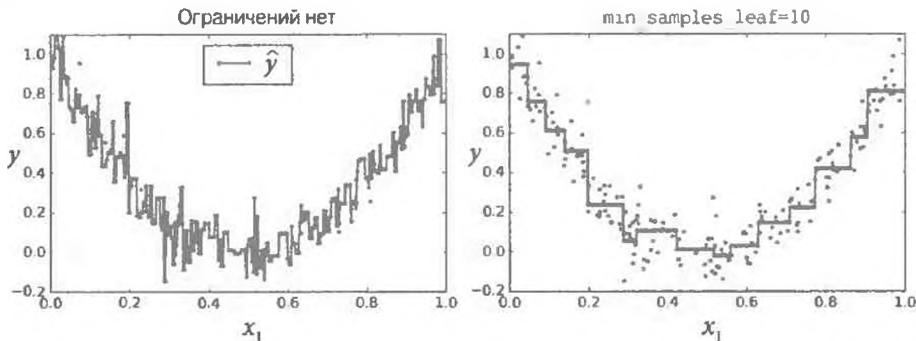


Рис. 6.6. Регуляризация регрессора в виде дерева принятия решений

# Неустойчивость

Наверняка к настоящему моменту вы уже 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 расцепить, тогда как справа, после поворота набора данных на  $45^\circ$ , 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 компонентов*; см. главу 8), который в результате дает лучшую ориентацию обучающих данных.

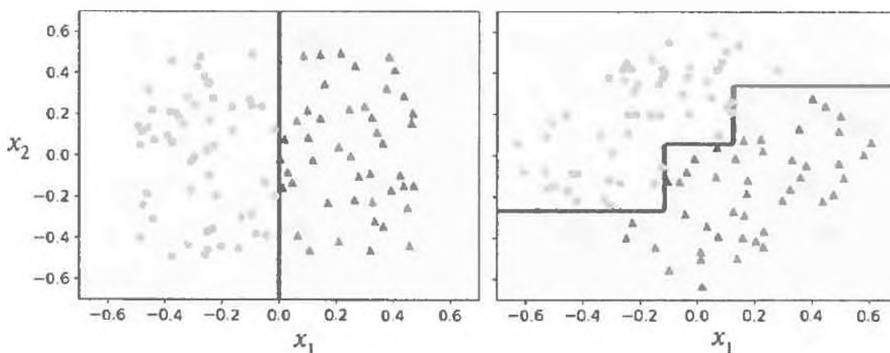


Рис. 6.7. Чувствительность к поворотам обучающего набора

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 стохастическим<sup>6</sup>, вы можете получать очень разные модели даже на тех же самых обучающих данных (если только не установите гиперпараметр `random_state`).

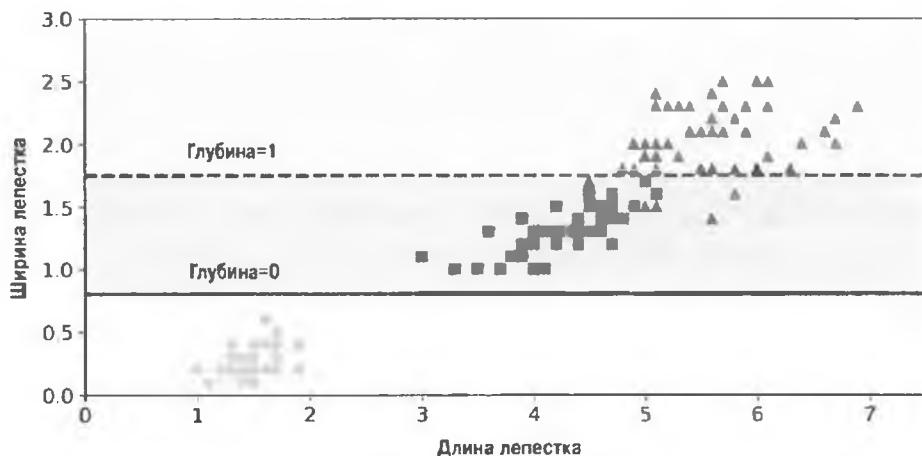


Рис. 6.8. Чувствительность к деталям обучающего набора

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 одним миллионом образцов?
2. Загрязненность Джини 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?

<sup>6</sup> Он случайно выбирает набор признаков для оценки в каждом узле.

5. Если обучение дерева принятия решений на обучающем наборе, содержащем один миллион образцов, занимает один час, тогда сколько примерно времени потребуется для обучения другого дерева принятия решений на обучающем наборе, содержащем десять 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`, выполнив следующие шаги:
  - а) используйте `make_moons(n_samples=10000, noise=0.4)` для генерации набора данных `moons`;
  - б) примените `train_test_split()` для расщепления набора данных `moons` на обучающий набор и испытательный набор;
  - 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 модель на полном обучающем наборе с применением найденных значений гиперпараметров и измерьте ее эффективность на испытательном наборе; вы должны получить правильность примерно от 85% до 87%.
8. Создайте лес в соответствии с перечисленными ниже шагами.
  - а) В продолжение предыдущего упражнения 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 наилучших значений гиперпараметров, найденных в предыдущем упражнении. 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) Наступило время магии. Для каждого образца в испытательном наборе сгенерируйте прогнозы посредством 1000 деревьев принятия решений и сохраните только наиболее частый прогноз (для этого можете воспользоваться функцией `mode()` из SciPy). Такой подход 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 примерно на 0.5–1.5%). Примите поздравления, ведь вы обучили классификатор на основе случного 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 *ансамблевое обучение (Ensemble Learning)*, а алгоритм ансамблевого обучения именуется *ансамблевым методом (Ensemble method)*.

В качестве примера ансамблевого метода вы можете обучать группу классификаторов на основе деревьев принятия решений, задействовав для каждого отличающийся случайный поднабор из обучающего набора. Для вырабатывания 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.

Как упоминалось в главе 2, вы часто будете использовать ансамблевые методы ближе к концу проекта, когда несколько хороших прогнозаторов уже построены, чтобы объединить 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)* и *стекинг (stacking)*. 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 базе случайного леса, классификатор методом k ближайших соседей и вероятно ряд других (рис. 7.1).

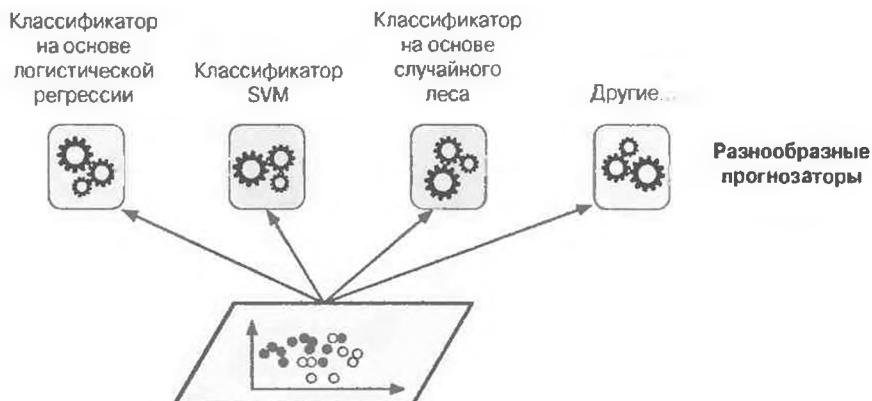


Рис. 7.1. Обучение разнообразных классификаторов

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.

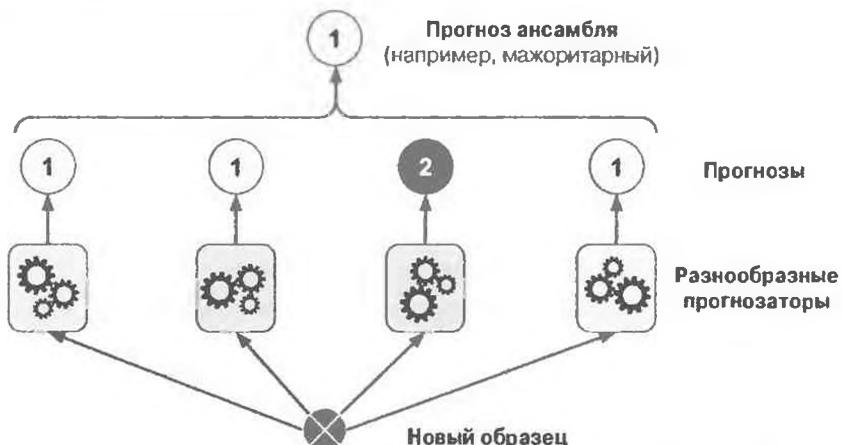


Рис. 7.2. Прогнозы классификатора с жестким голосованием

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 достаточно количество слабых учеников и они достаточно разнообразны.

Как подобное возможно? Пролить свет на эту тайну поможет следующая аналогия. 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 1000 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 1000 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%.



Рис. 7.3. Закон больших чисел

Аналогичным 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.



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 увеличит 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 import RandomForestClassifier
from import VotingClassifier
from import LogisticRegression
from 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, какова правильность каждого классификатора на испытательном наборе:

```
>>> from 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 0.896  
SVC 0.888  
VotingClassifier 0.904
```

Вот так! Классификатор с голосованием слегка превосходит все индивидуальные классификаторы.

Если все классификаторы в состоянии оценивать вероятности классов (т.е. все они 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 добавит метод `predict_proba()`). Если вы модифицируете предыдущий код для использования мягкого голосования, то обнаружите, что классификатор с голосованием добивается правильности свыше 91.2%!

## 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*) (<https://homl.info/20>)<sup>1</sup> — сокращение от *bootstrap aggregating*<sup>2</sup> (бутстрэн-агрегирование)).

<sup>1</sup> Лео Брейман, *Bagging Predictors* (Прогнозаторы с бэггингом), *Machine Learning* 24, выпуск 2 (1996 г.): с. 123–140.

<sup>2</sup> В статистике повторная выборка с заменой (возвращением) называется бутстрэн-пингом (*bootstrapping*).

Когда выборка выполняется без замены, такой метод называется *вставкой* или *вклеиванием* (<https://homl.info/21>)<sup>3</sup>.

Другими словами, бэггинг и вставка позволяют производить выборку обучающих образцов по нескольку раз множеством прогнозаторов, но только бэггинг разрешает осуществлять выборку обучающих образцов по нескольку раз одним и тем же прогнозатором. Процесс выборки и обучения представлен на рис. 7.4.

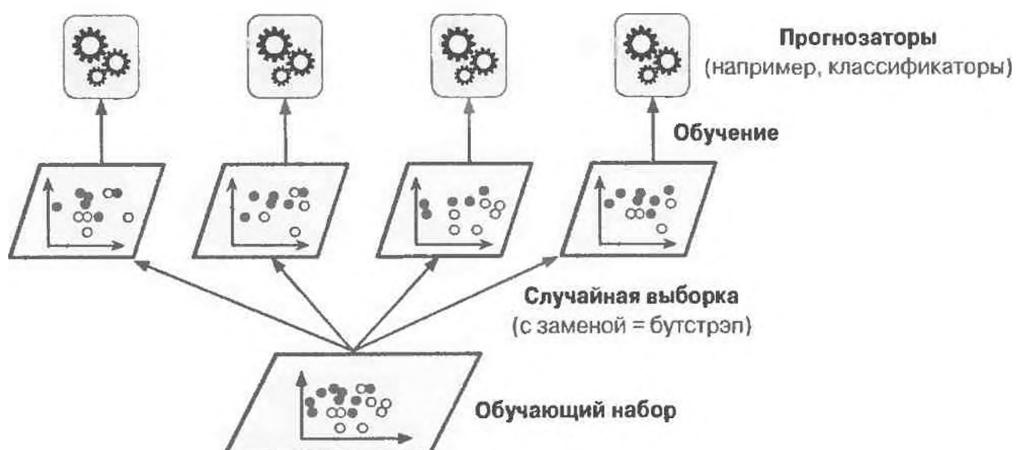


Рис. 7.4. Бэггинг и вставка предусматривают обучение нескольких прогнозаторов на различных случайных образцах из обучающего набора

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 дисперсию<sup>4</sup>. Обычно совокупный результат состоит в том, что ансамбль имеет похожее смещение, но меньшую дисперсию, чем одиночный прогнозатор, обученный на исходном обучающем наборе.

<sup>3</sup> Лео Брейман, *Pasting Small Votes for Classification in Large Databases and On-Line* (Вставка небольших голосов для классификации в крупных базах данных и динамический режим), *Machine Learning* 36, выпуск 1–2 (1999 г.): с. 85–103.

<sup>4</sup> Смещение и дисперсия были введены в главе 4.

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 решений<sup>5</sup>, каждый из которых обучается на 100 обучающих образцах, случайно выбранных из обучающего набора с заменой (пример бэггинга, но если вы хотите 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.

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

<sup>5</sup> 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`.

допускает приблизительно то же количество ошибок на обучающем наборе, но граница решений характеризуется меньшей нерегулярностью).

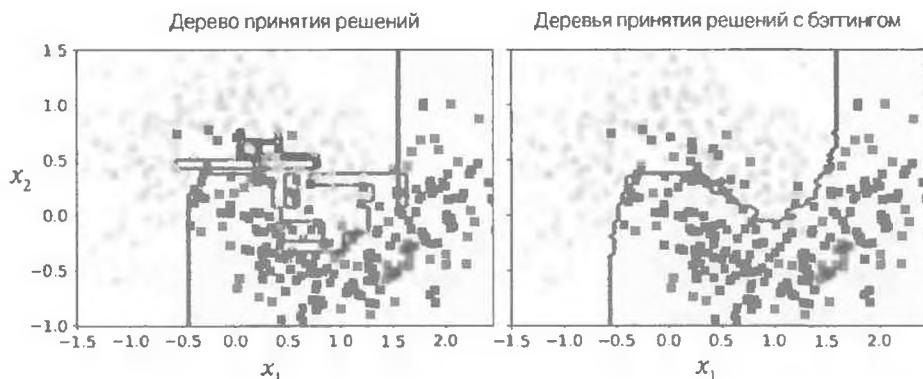


Рис. 7.5. Сравнение одиночного дерева принятия решений (слева) и ансамбля с бэггингом из 500 деревьев (справа)

Бутстрэппинг привносит чуть больше несходства в поднаборы, на которых обучается каждый прогнозатор, и потому бэггинг в итоге дает слегка более высокое смещение, чем вставка, но дополнительное несходство 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 выбираться в среднем только около 63% обучающих образцов<sup>6</sup>. 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.

<sup>6</sup> С ростом  $m$  эта доля приближается к  $1 - \exp(-1) \approx 63.212\%$ .

Поскольку прогнозатор никогда не видит образцы oob во время обучения, его можно оценивать на образцах oob без необходимости в наличии отдельного проверочного набора. 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:

```
>>> from sklearn.metrics import accuracy_score  
>>> y_pred = bag_clf.predict(X_test)  
>>> accuracy_score(y_test, y_pred)  
0.9120000000000003
```

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):

```
>>> 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. Выборка 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.

Методика особенно полезна, 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* (<https://homl.info/22>)<sup>7</sup>). Сбережение всех обучающих образцов (за счет установки `bootstrap=False` и `max_samples=1.0`), но проведение выборки признаков (путем установки `bootstrap_features` в `True` и/или `max_features` в значение меньше `1.0`) называется методом *случайных подпространств* (*Random Subspaces method* (<https://homl.info/23>)<sup>8</sup>).

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* (<https://homl.info/24>)<sup>9</sup>) — 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. Вместо построения экземпляра `BaggingClassifier` и его передачи экземпляру `DecisionTreeClassifier` вы можете применить класс

<sup>7</sup> Жиль Люпп и Пьер Гер, *Ensembles on Random Patches* (Ансамбли на случайных участках), *Lecture Notes in Computer Science* 7523 (2012 г.): с. 346–361.

<sup>8</sup> Тин Кам Хо, *The Random Subspace Method for Constructing Decision Forests* (Метод случайных подпространств для построения лесов принятия решений), *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, выпуск 8 (1998 г.): с. 832–844.

<sup>9</sup> Тин Кам Хо, *Random Decision Forests* (Случайные леса принятия решений), *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995 г.): с. 278.

`RandomForestClassifier`, который является более удобным и оптимизированным для деревьев принятия решений<sup>10</sup> (аналогичным образом имеется класс `RandomForestRegressor` для задач регрессии). Показанный ниже код использует все доступные процессорные ядра для обучения классификатора на основе случайного леса с 500 деревьями (каждое ограничено максимум 16 узлами):

```
from import RandomForestClassifier  
rnd_clf = RandomForestClassifier(n_estimators= ,  
                                  max_leaf_nodes= , n_jobs=- )  
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 ансамблем<sup>11</sup>.

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= ),  
    n_estimators= , max_samples= , bootstrap=True, n_jobs=- )
```

---

<sup>10</sup> Класс `BaggingClassifier` остается полезным, если вам нужен пакет чего-то, отличающегося от деревьев принятия решений.

<sup>11</sup> 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 приятия решений). Лес с такими чрезвычайно случайными деревьями называется ансамблем *особо случайных деревьев* (*Extremely Randomized Trees ensemble* (<https://homl.info/25>)<sup>12</sup> или *Extra-Trees* для краткости). И снова в такой методике 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` выполнять 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).

<sup>12</sup> Пьер Гер и др., *Extremely Randomized Trees* (Особо случайные деревья), *Machine Learning* 63, выпуск 1 (2006 г.): с. 3–42.

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 import load_iris  
>>> iris = load_iris()  
>>> rnd_clf = RandomForestClassifier(n_estimators=100, 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.

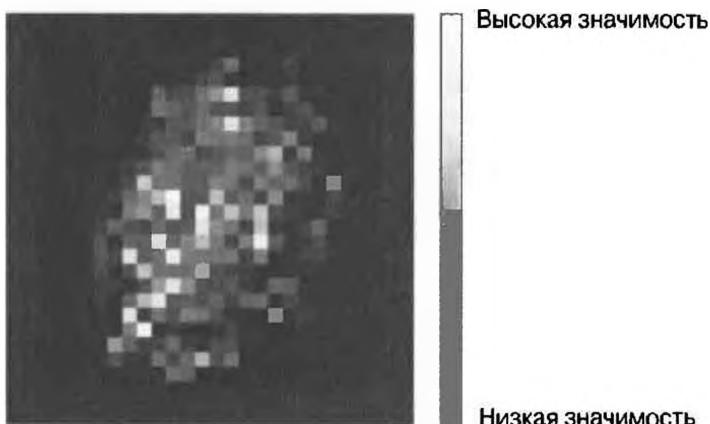


Рис. 7.6. Значимость пикселей в наборе данных MNIST (согласно классификатору на основе случайного леса)

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. Доступно много методов бустинга, но безоговорочно самыми популярными являются *AdaBoost* (<https://homl.info/26>)<sup>13</sup> — 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 обучении классификатора AdaBoost алгоритм сначала обучает базовый классификатор (такой как дерево принятия решений) и использует его для выработки прогнозов на обучающем наборе. Затем алгоритм увеличивает относительный вес некорректно классифицированных обучающих образцов. Далее он обучает второй классификатор с применением обновленных весов, снова вырабатывает прогнозы на обучающем наборе, обновляет веса образцов и т.д. (рис. 7.7).

На рис. 7.8 показаны границы решений пяти последовательных прогнозаторов на наборе данных moons (в рассматриваемом примере каждый прогнозатор является сильно регуляризируемым классификатором SVM с ядром RBF<sup>14</sup>).

<sup>13</sup> Йоав Фройнд и Роберт Шапайр, *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), *Journal of Computer and System Sciences* 55, выпуск 1 (1997 г.): с. 119–139.

<sup>14</sup> 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.

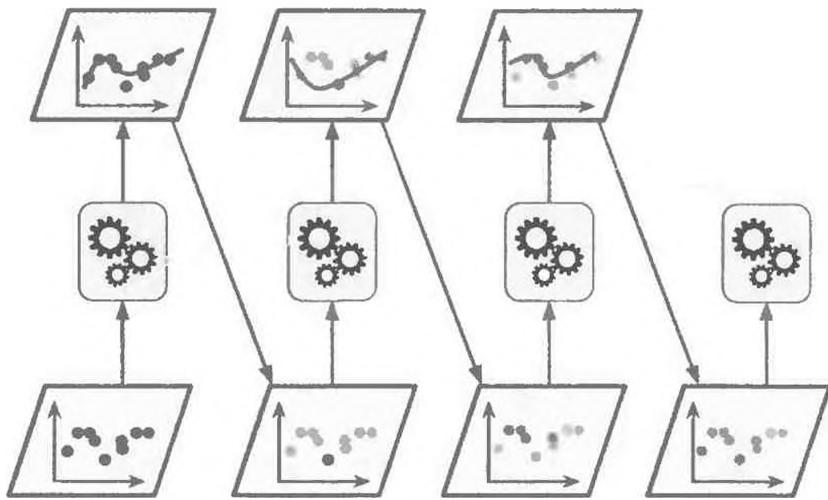


Рис. 7.7. Последовательное обучение классификатора AdaBoost с обновлением весов образцов

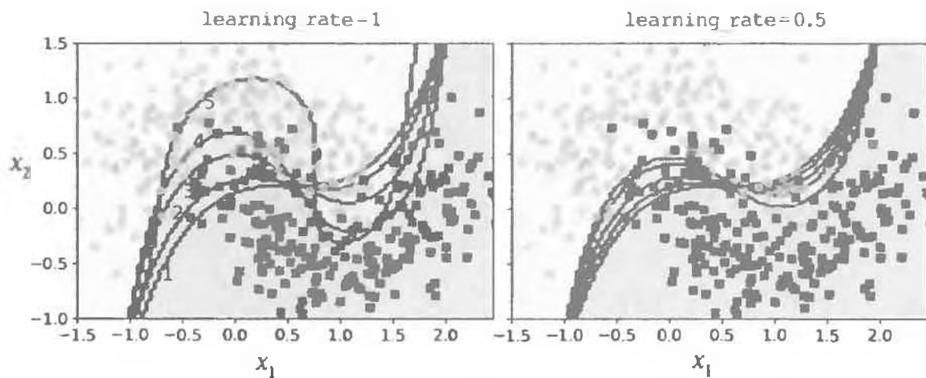


Рис. 7.8. Границы решений следующих друг за другом прогнозаторов

Первый классификатор воспринимает многие образцы неправильно, так что 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.



У методики последовательного обучения есть один важный недостаток: она не допускает распараллеливания (или только частично), поскольку каждый прогнозатор можно обучать лишь после того, как был обучен и оценен предыдущий прогнозатор. В результате такая методика 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

$$r_j = \frac{\sum_{i=1}^m w^{(i)}_{j \neq y^{(i)}}}{\sum_{i=1}^m w^{(i)}},$$

где  $\hat{y}_j^{(i)}$  — прогноз  $j$ -того прогнозатора для  $i$ -того образца.

Затем вычисляется вес  $\alpha_j$  прогнозатора с применением уравнения 7.2, где  $\eta$  — гиперпараметр скорости обучения (со стандартным значением 1)<sup>15</sup>. Чем более правильным является прогнозатор, тем выше будет его вес. Если прогнозатор угадывает всего лишь случайно, тогда его вес будет близок к нулю. Однако если он почти всегда ошибается (т.е. правильность ниже случайного угадывания), то его вес будет отрицательным.

#### Уравнение 7.2. Вес прогнозатора

$$\alpha_j = \eta \log \frac{1 - r_j}{r_j}$$

Далее алгоритм AdaBoost обновляет веса образцов с применением уравнения 7.3, которое поднимает веса некорректно классифицированных образцов.

<sup>15</sup> В исходном алгоритме AdaBoost гиперпараметр скорости обучения не используется.

### 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  $\sum_{i=1}^m w^{(i)}$ ).

В заключение новый прогнозатор обучается с использованием обновленных весов, и 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  $\alpha_j$ . Спрогнозированным классом будет тот, который получает большинство взвешенных голосов (уравнение 7.4).

### Уравнение 7.4. Прогнозы AdaBoost

$$\hat{y}(\mathbf{x}) = \operatorname{argmax}_k \sum_{j=1}^N \alpha_j, \\ \hat{y}_j(\mathbf{x}) = k$$

где  $N$  — количество прогнозаторов.

Библиотека Scikit-Learn использует многоклассовую версию алгоритма AdaBoost, называемую *SAMME* (<https://homl.info/27>)<sup>16</sup>, 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. 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.

<sup>16</sup> Дополнительные сведения ищите в статье Джи Жу и др. “Multi-Class AdaBoost” (“Многоклассовый алгоритм AdaBoost”), *Statistics and Its Interface* 2, выпуск 3 (2009 г.): с. 349–360.

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). Пеньк 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.", learning_rate=1)  
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* (Gradient Boosting; <https://homl.info/28>)<sup>17</sup>. 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 регрессии с градиентным бустингом* (*Gradient Boosted Regression Tree* — *GBRT*). Первым делом подгоним регрессор DecisionTreeRegressor к обучающему набору (например, зашумленному квадратичному обучающему набору):

<sup>17</sup> Впервые был представлен в работе Лео Бреймана *Arcing the Edge* (Образование дуги на краю), опубликованной в 1997 г. (<https://homl.info/arcing>), и далее усовершенствован Джеромом Фридманом в работе *Greedy Function Approximation: A Gradient Boosting Machine* (Жадная аппроксимация функций: механизм градиентного бустинга), вышедшей в 1999 г. (<https://homl.info/gradboost>).

```
from import DecisionTreeRegressor  
tree_reg1 = DecisionTreeRegressor(max_depth= )  
tree_reg1.fit(X, y)
```

Далее 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)
```

Теперь мы располагаем ансамблем, содержащим 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))
```

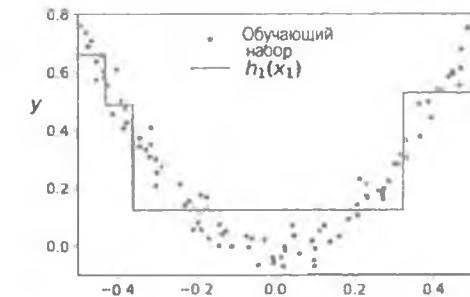
На рис. 7.9 в левой колонке представлены прогнозы этих трех деревьев, а в правой — прогнозы ансамбля. В первой строке ансамбль имел только одно дерево, потому его прогнозы в точности такие же, как у первого дерева. Во второй строке новое дерево обучено на остаточных ошибках первого дерева. Справа видно, 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), а также гиперпараметры для управления обучением ансамбля вроде количества деревьев (n\_estimators).

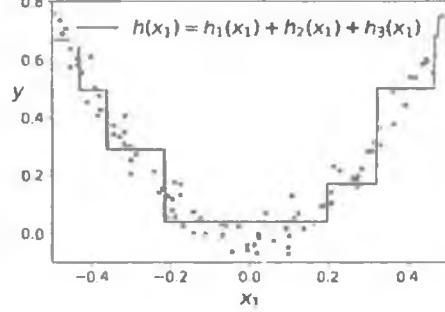
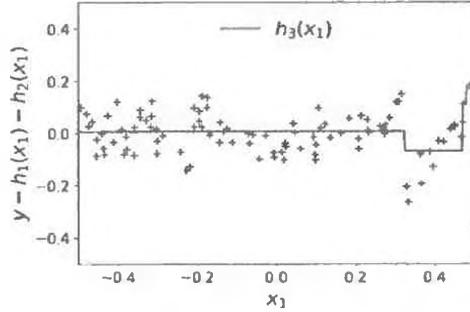
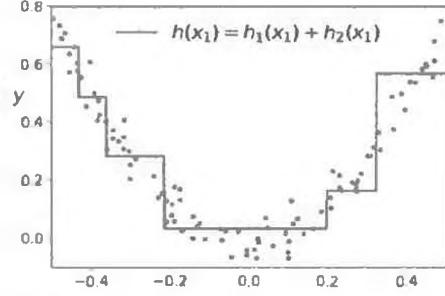
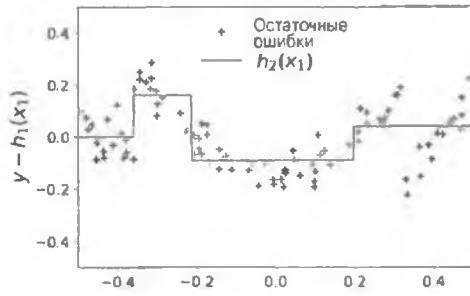
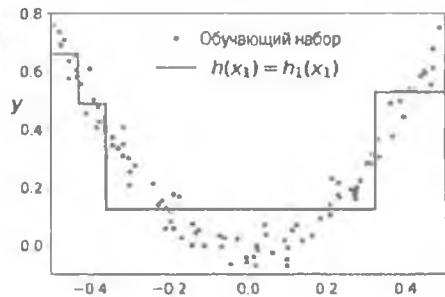
Следующий код создает тот же самый ансамбль, что и предыдущий код:

```
from import GradientBoostingRegressor  
gbdt = GradientBoostingRegressor(max_depth= , n_estimators= ,  
                                  learning_rate= )  
gbdt.fit(X, y)
```

### Остаточные ошибки и прогнозы деревьев

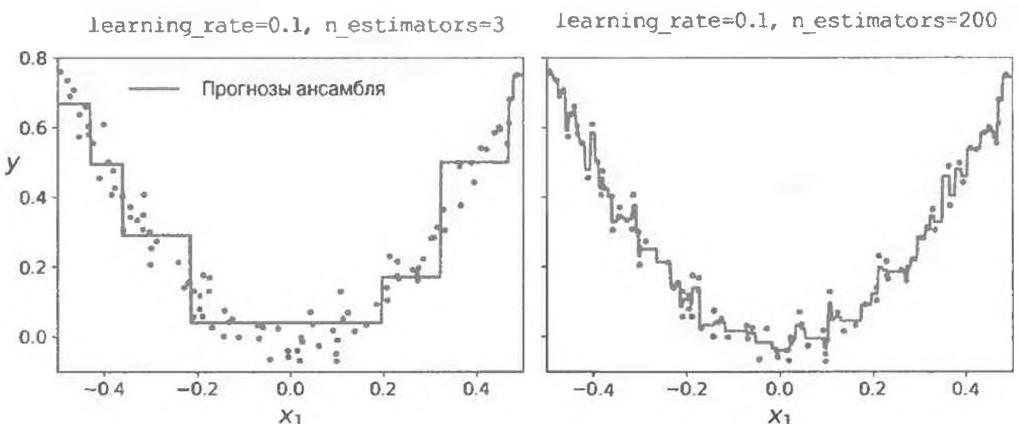


### Прогнозы ансамбля



**Рис. 7.9.** В приведенной картине градиентного бустинга первый прогнозатор (слева вверху) обучался обычным образом, затем каждый последующий прогнозатор (слева посередине и слева внизу) обучался на остаточных ошибках предшествующего прогнозатора; в колонке справа показаны прогнозы результирующего ансамбля

Гиперпараметр скорости обучения (`learning_rate`) задает 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 регуляризации, называемый *сжатием* (*shrinkage*). На рис. 7.10 показаны два ансамбля GBRT, обученные с низкой скоростью обучения: ансамбль слева не имеет достаточного числа деревьев, чтобы подгоняться к обучающему набору, в то время как ансамбль справа имеет чересчур 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           as
from           import train_test_split
from           import mean_squared_error

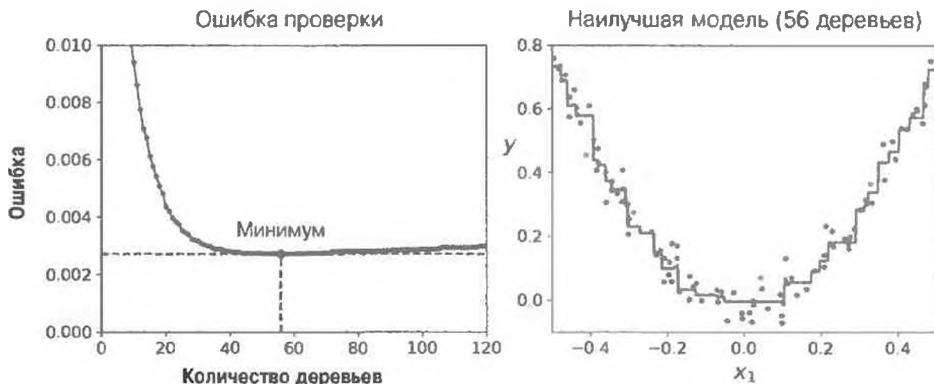
X_train, X_val, y_train, y_val = train_test_split(X, y)

gbrt = GradientBoostingRegressor(max_depth= , n_estimators= )
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) + 1

gbrt_best = GradientBoostingRegressor(max_depth= ,
                                       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` в `True`, что заставит 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, подлежащих 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, такая мето-

дика обменивает более высокое смещение на низкую дисперсию. В добавок обучение значительно ускоряется. Прием называется *стохастическим градиентным бустингом* (*Stochastic Gradient Boosting*).



Градиентный бустинг можно применять с другими функциями издержек, что управляет гиперпараметром *loss* (за дополнительными сведениями обращайтесь в документацию Scikit-Learn).

Нелишне упомянуть об оптимизированной реализации градиентного бустинга, доступной в популярной библиотеке Python под названием XGBoost (<https://github.com/dmlc/xgboost>), которое означает Extreme Gradient Boosting (экстремальный градиентный бустинг). Изначально пакет был разработан Тяньцзи Ченом в сообществе распределенного (глубокого) машинного обучения (*Distributed (Deep) Machine Learning Community — DMIC*) и задуман стать исключительно быстрым, масштабируемым и переносимым. Фактически XGBoost часто является важным компонентом победивших решений в состязаниях по МО. В XGBoost предлагается API-интерфейс, очень похожий на таковой в Scikit-Learn:

```
import xgboost  
  
xgb_reg = xgboost.XGBRegressor()  
xgb_reg.fit(X_train, y_train)  
y_pred = xgb_reg.predict(X_val)
```

В добавок пакет XGBoost обладает рядом замечательных характеристик вроде автоматической поддержки раннего прекращения:

```
xgb_reg.fit(X_train, y_train,  
            eval_set=[(X_val, y_val)], early_stopping_rounds= )  
y_pred = xgb_reg.predict(X_val)
```

Вы обязательно должны его опробовать!

## Стекинг

Последний ансамблевый метод, который мы обсудим в главе, называется *стекингом* (“stacking” — сокращение для “stacked generalization” (<https://homl.info/29>)<sup>18</sup> (стековое обобщение)). Он основан на простой идее: вместо

<sup>18</sup> Дэвид Уолперт, *Stacked Generalization* (Стековое обобщение), *Neural Networks 5*, выпуск 2 (1992 г.): с. 241–259.

того, чтобы использовать тривиальные функции (такие как с жестким голосованием) для агрегирования прогнозов всех прогнозаторов в ансамбле, почему бы нам ни научить какую-то модель делать это агрегирование? На рис. 7.12 демонстрируется ансамбль подобного рода, выполняющий задачу регрессии на новом образце. 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).

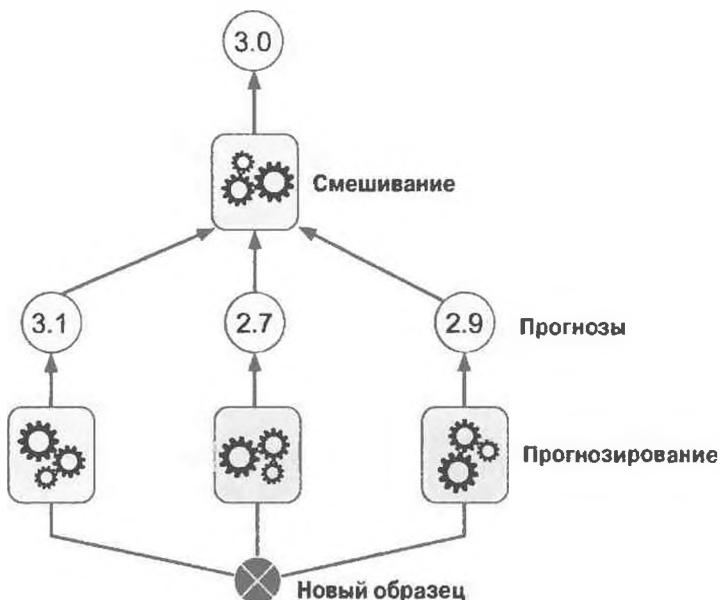


Рис. 7.12. Агрегирование прогнозов с применением смешивающего прогнозатора

Распространенный подход к обучению смесителя предполагает использование удержаняемого (*hold-out*) набора<sup>19</sup>. Давайте посмотрим, как это работает. Для начала обучающий набор расщепляется на два поднабора. Первый поднабор применяется для обучения прогнозаторов на первом уровне (рис. 7.13).

Затем прогнозаторы первого уровня используются для выработывания 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.

<sup>19</sup> В качестве альтернативы можно использовать внеблочные (out-of-fold) прогнозы. В некоторых контекстах это называется стекингом, тогда как применение удержаняемого набора — смешиванием. Однако для многих людей упомянутые термины синонимичны.

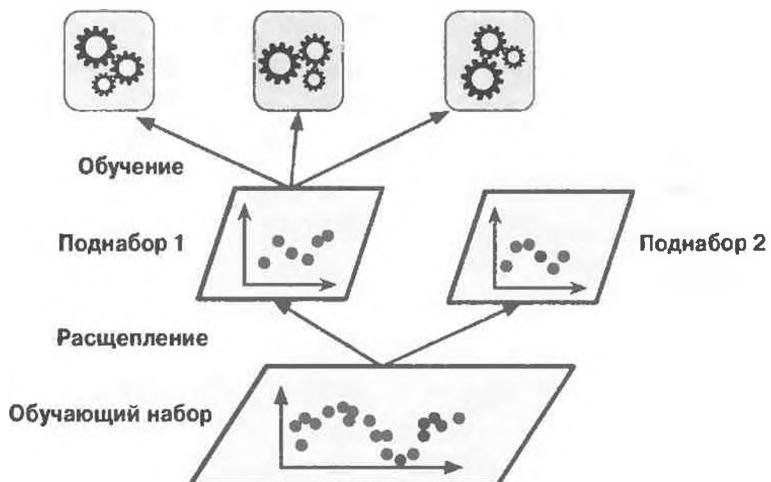


Рис. 7.13. Обучение первого уровня

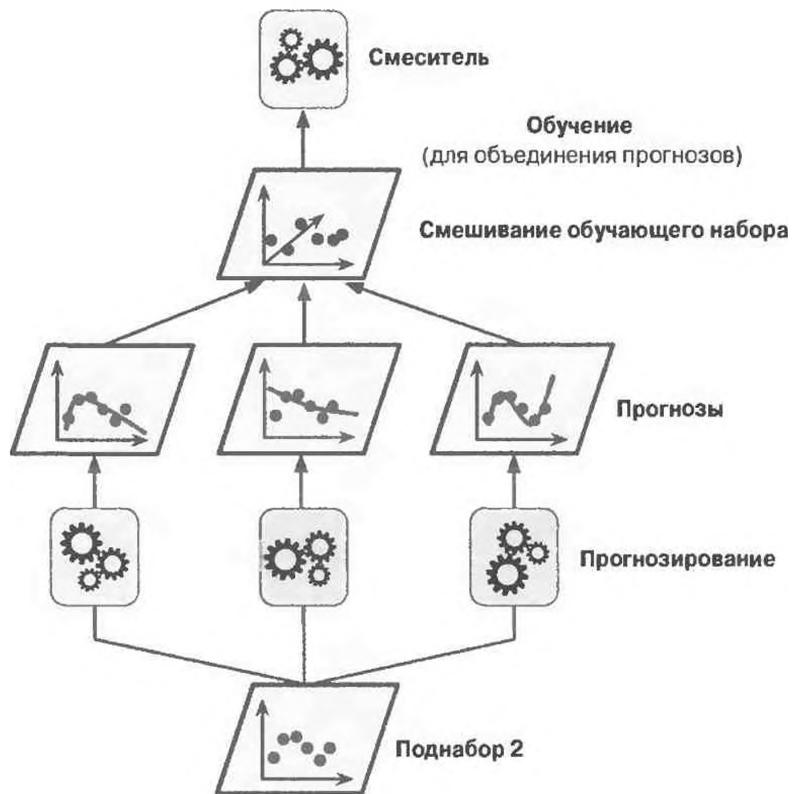


Рис. 7.14. Обучение смесителя

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.

На самом деле подобным образом можно обучить несколько разных смесителей (например, смеситель, использующий линейную регрессию, и еще один смеситель, применяющий регрессию на основе случайного леса), чтобы получить 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).

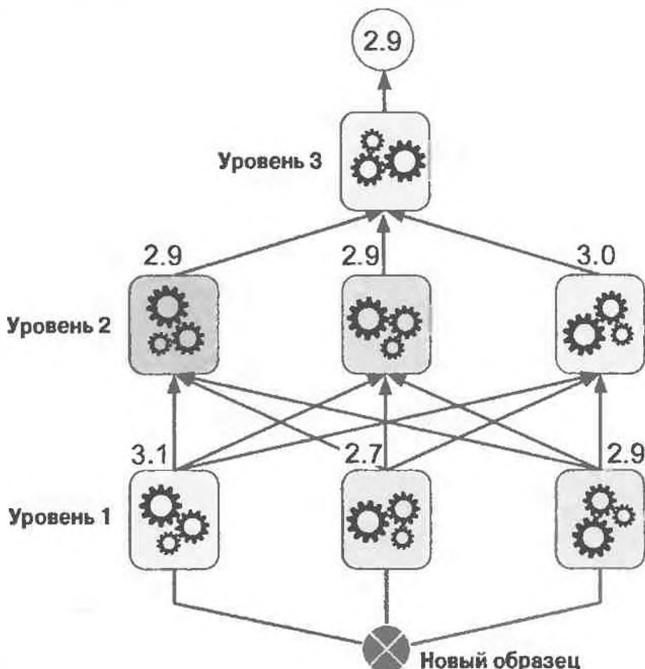


Рис. 7.15. Прогнозы в многоуровневом ансамбле со стекингом

К сожалению, Scikit-Learn не поддерживает стекинг напрямую, но развернуть собственную реализацию не слишком сложно (см. упражнения ниже). В качестве альтернативы вы можете использовать реализацию с открытым кодом, такую как brew (<https://github.com/viisar/brew>).

## 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 со 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? Каким образом может помочь такая добавочная случайность? Особо случайные деревья медленнее или быстрее обычных случайных лесов?
6. Если ваш ансамбль AdaBoost недообучается на обучающих данных, то какие гиперпараметры вы должны подстраивать и как?
7. Если ваш ансамбль с градиентным бустингом переобучается обучающим набором, то вам потребуется увеличить или же уменьшить скорость обучения?
8. Загрузите данные MNIST (представленные в главе 3) и расщепите их на обучающий набор, проверочный набор и испытательный набор (например, применив 50 000 образцов для обучения, 10 000 — для проверки и 10 000 — для испытания). Затем обучите различные классификаторы, например, классификатор на основе случайноголеса, классификатор на базе особо случайных деревьев и классификатор SVM. Далее попытайтесь объединить их в ансамбль, который превосходит каждый индивидуальный классификатор на проверочном наборе, используя клас-

сификатор 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 результатирующими прогнозами: каждый обучающий образец представляет собой вектор, содержащий набор прогнозов от всех классификаторов для изображения, и целью является класс изображения. Обучите классификатор на этом новом обучающем наборе. 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, данное обстоятельство 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, превратив трудноразрешимую задачу в разрешимую. Например, рассмотрим изображения MNIST (представленные в главе 3): пиксели на кромках 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. Вдобавок два смежных пикселя часто сильно связаны: если вы сольете их в один пиксель (скажем, взяв среднее интенсивности 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 функционировать чуть хуже. Оно также немножко усложнит конвейеры, сделав их труднее в сопровождении. Следовательно, если обучение оказывается слишком медленным, то сначала вы должны попробовать обучить свою систему с применением исходных данных, прежде чем обдумывать использование понижения размерности. Однако в ряде случаев понижение размерности обучающих данных может отфильтровать некоторый шум и ненужные детали, обеспечив более высокую эффективность, но в целом так не происходит; оно просто ускоряет обучение.

В добавок к ускорению обучения понижение размерности также чрезвычайно полезно при *визуализации данных* (*data visualization — DataViz*). Уменьшение количества измерений до двух (или трех) делает возможным вычерчивание сжатого представления обучающего набора с высоким числом измерений на графике и часто приводит к важным догадкам благодаря зритальному обнаружению закономерностей, таких как скопления (кластеры). Кроме того, визуализация данных DataViz крайне важна при сообщении ваших заключений людям, не являющимся специалистами в науке о данных — в частности руководителям, принимающим решения, которые будут пользоваться вашими результатами.

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 методикам понижения размерности: PCA, Kernel PCA и LLE.

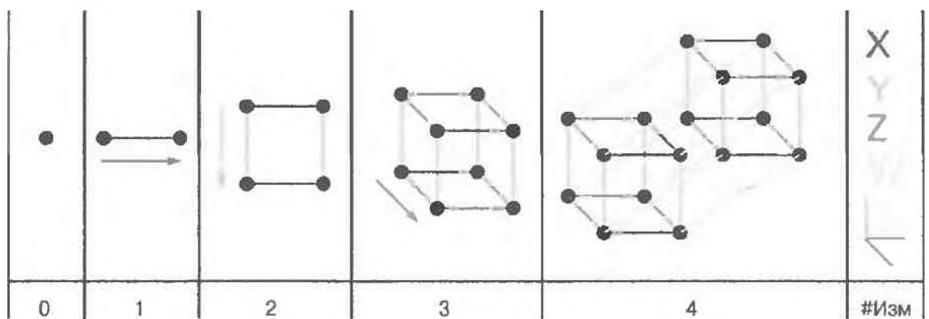
## “Проклятие размерности”

Мы настолько привыкли жить в трех измерениях<sup>1</sup>, что наша интуиция попросту 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.

Оказывается, что в многомерном пространстве многие вещи ведут себя совсем иначе. Например, если вы выбираете произвольную точку внутри единичного квадрата (квадрата  $1 \times 1$ ), то она имеет только приблизительно 0.4%-ный шанс находиться ближе, чем 0.001 от края (другими словами, исключительно 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 краю<sup>2</sup>.

<sup>1</sup> Хорошо, в четырех измерениях, если учесть время, и в еще большем количестве, если вы занимаетесь теорией струн.

<sup>2</sup> 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.



**Рис. 8.1. Точка, отрезок, квадрат, куб и тессеракт  
(от нулевой размерности до четырехмерного гиперкуба)<sup>3</sup>**

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? Хорошо, просто при высоком количестве измерений пространства вдоволь. В результате многомерные наборы данных подвержены риску оказаться чрезвычайно разреженными: большинство обучающих образцов по всей вероятности будут находиться далеко друг от друга. Разумеется, это также означает, что новый образец, видимо, 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

<sup>3</sup> Понаблюдайте за вращением тессеракта, спроектированного на трехмерное пространство: <https://homl.info/30>. 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>.

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.

## 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 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.

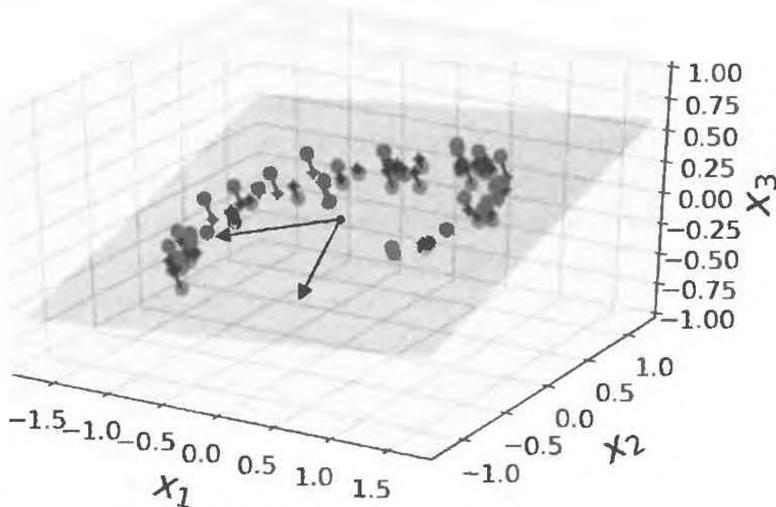


Рис. 8.2. Трехмерный набор данных, лежащий близко к двумерному подпространству

Обратите внимание, что все обучающие образцы расположены близко к некоторой плоскости: это подпространство с меньшим количеством измерений (двумерное) в рамках многомерного (трехмерного) пространства. Если мы спроектируем каждый обучающий образец перпендикулярно на такое подпространство (представлено короткими линиями, соединяющими образцы с плоскостью), то получим новый двумерный набор данных, приведенный на рис. 8.3. Та-дам! Мы только что понизили размерность набора данных с 3-х до 2-х. Имейте в виду, что оси соответствуют новым признакам  $z_1$  и  $z_2$  (координаты проекций на плоскости).

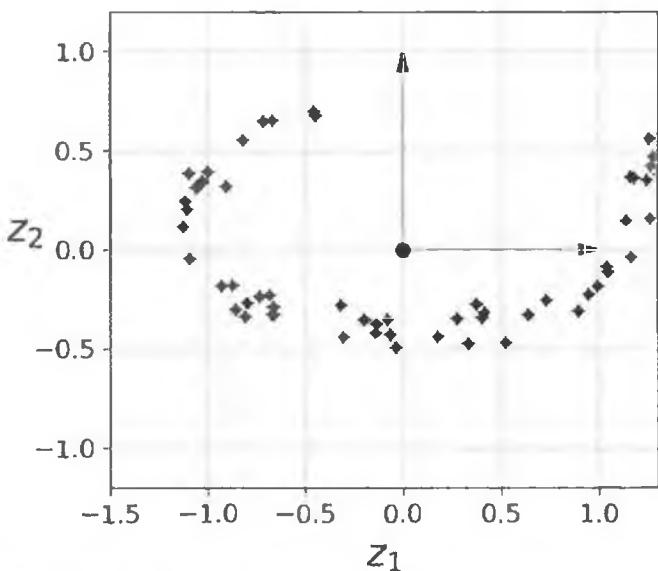


Рис. 8.3. Новый двумерный набор данных после проекции

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.

Простое проецирование 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.

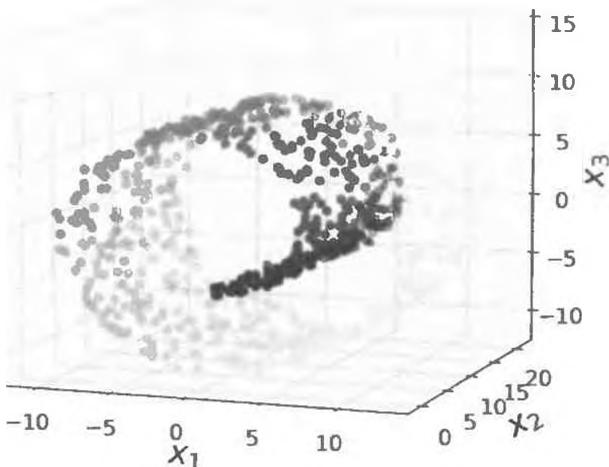


Рис. 8.4. Набор данных Swiss roll

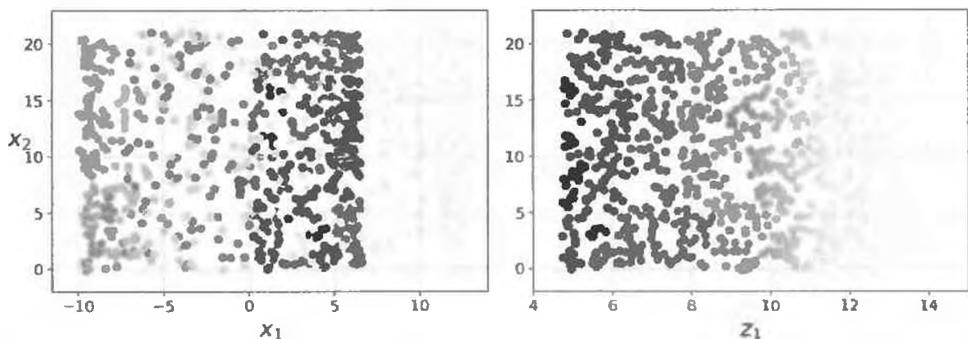


Рис. 8.5. Сплющивание швейцарского рулета за счет проецирования на плоскость (слева) в противоположность его развертыванию (справа)

## 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 на-

зывается 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.

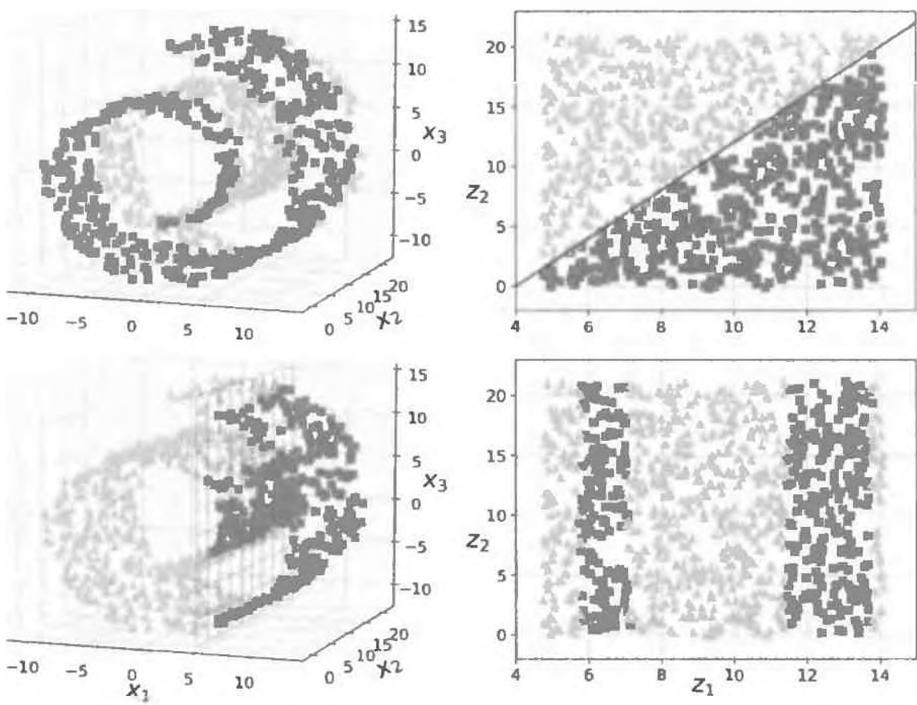
Давайте снова обратимся к набору данных MNIST: все изображения рукописных цифр имеют некоторые сходные черты. Они образованы из соединительных линий, кромки являются белыми и они более или менее центрированы. Если вы генерировали изображения случайным образом, то лишь смехотворно малая их часть будет 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 скать набор данных в многообразие 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 прямую линию.

Тем не менее, такое неявное предположение не всегда сохраняется. 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, понижение размерности обучающего набора перед обучением модели обычно 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.



*Рис. 8.6. Граница решений не всегда может становиться проще при меньшем количестве измерений*

## PCA

Анализ главных компонентов (*Principal Component Analysis — PCA*) является безоговорочно наиболее популярным алгоритмом понижения размерности. Сначала он идентифицирует гиперплоскость, которая находится ближе всего к данным, и затем проецирует на нее данные, в точности как на рис. 8.2.

### Предохранение дисперсии

Прежде чем обучающий набор можно будет спроектировать 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.

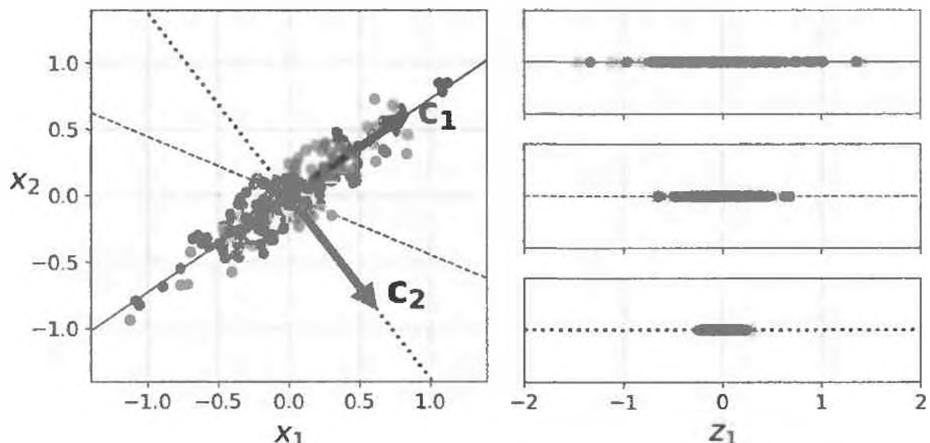


Рис. 8.7. Выбор подпространства для проецирования на него

Кажется разумным выбрать ось, которая предохраняет максимальную величину дисперсии, поскольку она вероятнее всего будет терять меньше информации, чем другие проекции. 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 PCA (<https://homl.info/pca>)<sup>4</sup>.

## 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.

Здесь  $i$ -ая ось называется  $i$ -тым главным компонентом (*principal component* — PC) данных. На рис. 8.7 первым компонентом PC является ось, где находится вектор  $c_1$ , а вторым компонентом PC — ось, на которой расположен вектор  $c_2$ .

<sup>4</sup> Карл Пирсон, *On Lines and Planes of Closest Fit to Systems of Points in Space* (О линиях и плоскостях наиболее точного соответствия системам точек в пространстве), *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, выпуск 11 (1901 г.): с. 559-572.

На рис. 8.2 первые два компонента РС представляют собой ортогональные оси, где находятся две стрелки на плоскости, а третий компонент РС — ось, перпендикулярную к этой плоскости.



Для каждого главного компонента алгоритм РСА находит центрированный в нуле единичный вектор, указывающий в направлении компонента РС. Поскольку два противоположных единичных вектора расположены на той же самой оси, направление единичных векторов, возвращаемых компонентом РСА, не является стабильным: если вы внесете в обучающий набор незначительное возмущение и запустите алгоритм РСА снова, то единичные векторы могут указывать в направлении, противоположном направлению исходных векторов. Однако обычно они по-прежнему будут находиться на тех же самых осях. В ряде случаев пара единичных векторов может даже повернуться или поменяться местами (если величины дисперсии вдоль их двух осей близки), 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 перемножение трех матриц  $U \Sigma V^T$ , где V содержит единичные векторы, определяющие все главные компоненты, которые мы ищем (уравнение 8.1).

### Уравнение 8.1. Матрица главных компонентов

$$V = \begin{pmatrix} | & | & | \\ c_1 & c_2 & \dots & c_n \\ | & | & | \end{pmatrix}$$

Следующий код Python применяет функцию `svd()` из NumPy для получения всех главных компонентов обучающего набора и затем извлекает два единичных вектора, которые определяют первые два компонента РС:

```
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. Как вы увидите, 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.

## Проектирование 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 спроектировать обучающий набор на гиперплоскость и получить сокращенный набор данных  $X_{d\text{-проекция}}$  с размерностью  $d$ , необходимо перемножить матрицу  $X$  обучающего набора и матрицу  $W_d$ , которая определена как матрица, содержащая первые  $d$  столбцов матрицы  $V$ , как показано в уравнении 8.2.

### Уравнение 8.2. Проектирование обучающего набора вниз до $d$ измерений

$$X_{d\text{-проекция}} = X W_d$$

Представленный ниже 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 величину дисперсии.

## Использование Scikit-Learn

Для реализации алгоритм PCA класс PCA из Scikit-Learn применяет разложение SVD, как мы делали ранее в главе. 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 его атрибут `components_` содержит в себе транспонированную матрицу  $W_d$  (например, единичный вектор, который определяет первый главный компонент, эквивалентен `pca.components_.T[:, 0]`).

## Коэффициент объясненной дисперсии

Еще одной 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 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])
```

Вывод говорит о том, что 84.2% дисперсии набора данных лежит вдоль первого компонента РС, а 14.6% — вдоль второго компонента РС. Третьему компоненту РС остается менее 1.2%, потому разумно предположить, что третий компонент РС несет в себе мало информации.

## Выбор правильного количества измерений

Вместо произвольного выбора числа измерений для понижения размерности проще выбирать такое количество измерений, которое дает в сумме достаточно большую долю дисперсии (скажем, 95%). Конечно, рекомендация не относится к понижению размерности с целью визуализации данных — в этом случае вы захотите понизить размерность до 2 или 3.

Следующий код реализует алгоритм РСА 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
```

Затем вы могли бы установить `n_components=d` и запустить алгоритм РСА снова. Однако есть намного лучший вариант: вместо указания количества 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. В нашем случае видно, что понижение размерности до примерно 100 измерений не приведет к слишком большой потере объясненной дисперсии.

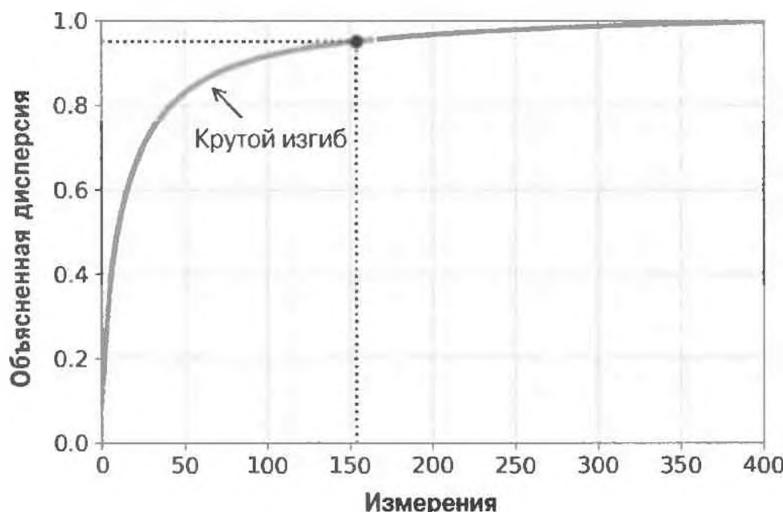


Рис. 8.8. Объясненная дисперсия как функция количества измерений

## Алгоритм РСА для сжатия

После понижения размерности обучающий набор требует гораздо меньше пространства. В качестве примера попробуйте применить алгоритм РСА к набору данных MNIST, предохраняя 95% его дисперсии. Вы должны обнаружить, что каждый образец будет иметь лишь чуть более 150 признаков вместо исходных 784 признаков. Итак, наряду с тем, что наивысшая доля дисперсии предохранена, набор данных теперь меньше, чем 20% его первоначального размера! Это приемлемая степень сжатия, и вы заметите, что такое сокращение размера способно значительно ускорить алгоритм классификации (вроде SVM).

Сокращенный набор данных также можно возвратить к 784 измерениям, применив обратную трансформацию проекции РСА. Первоначальные данные не будут восстановлены, т.к. проекция привела к утрате некоторой информации (в рамках дисперсии 5%, которая была отброшена), но результат, вероятно, окажется близким к исходным данным. Средний квадрат расстояния между первоначальными и восстановленными данными (сжатыми и затем распакованными) называется ошибкой восстановления (*reconstruction error*).

Показанный далее код сжимает набор данных MNIST до 154 измерений и с помощью метода `inverse_transform()` распаковывает его обратно в 784 измерения:

```
pca = PCA(n_components = 154)
X_reduced = pca.fit_transform(X_train)
X_recovered = pca.inverse_transform(X_reduced)
```

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.

В уравнении 8.3 демонстрируется обратная трансформация.

**Уравнение 8.3. Обратная трансформация РСА, восстанавливающая исходное количество измерений**

$$X_{\text{восстановленная}} = X_{d\text{-проекция}} W_d^T$$



Рис. 8.9. Сжатие набора данных MNIST с предохраниением 95% дисперсии

## Рандомизированный анализ главных компонентов

В случае установки гиперпараметра `svd_solver` в "randomized" библиотека Scikit-Learn использует стохастический алгоритм, называемый *рандомизированным анализом главных компонентов (Randomized PCA)*, который быстро находит аппроксимацию первых  $d$  главных компонентов. Его вычислительная сложность составляет  $O(m \times d^2) + O(d^3)$  вместо  $O(m \times n^2) + O(n^3)$  при подходе с полным SVD, так что когда  $d$  намного меньше  $n$ , он значительно быстрее полного SVD:

```
rnd_pca = PCA(n_components=154, svd_solver="randomized")
X_reduced = rnd_pca.fit_transform(X_train)
```

По умолчанию гиперпараметр `svd_solver` на самом деле установлен в "auto": библиотека Scikit-Learn автоматически применяет рандомизированный алгоритм PCA, если  $m$  или  $n$  больше 500 и  $d$  меньше, чем 80% значения  $m$  или  $n$ , а иначе использует подход с полным SVD. Если вы хотите вынудить Scikit-Learn применять полный SVD, тогда можете установить гиперпараметр `svd_solver` в "full".

## Инкрементный анализ главных компонентов

Проблема с предшествующими реализациями алгоритма PCA в том, что для функционирования алгоритма они требуют помещения в память целого обучающего набора. К счастью, были разработаны алгоритмы *инкрементного анализа главных компонентов (Incremental PCA — IPCA)*. Такие алгоритмы

позволяют расщепить обучающий набор на мини-пакеты и передавать алгоритму IPCA по одному мини-пакету за раз. Прием удобен для крупных обучающих наборов и 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` из Scikit-Learn (<https://homl.info/32>)<sup>5</sup>, 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 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 `memmap` 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.memmap(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)
```

## 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

<sup>5</sup> Библиотека Scikit-Learn использует алгоритм, описанный в статье Дэвида Росса и др. *Incremental Learning for Robust Visual Tracking* (Постепенное обучение для надежного визуального слежения), *International Journal of Computer Vision* 77, выпуск 1–3 (2008 г.); с. 125–141.

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*; <https://hml.info/33>)<sup>6</sup>. Часто полезно предохранять кластеры образцов после проекции или временами даже развертывать наборы данных, которые лежат близко к скрученному многообразию.

В следующем коде используется 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 import KernelPCA
rbf_pca = KernelPCA(n_components= , kernel="rbf", gamma= )
X_reduced = rbf_pca.fit_transform(X)
```

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).

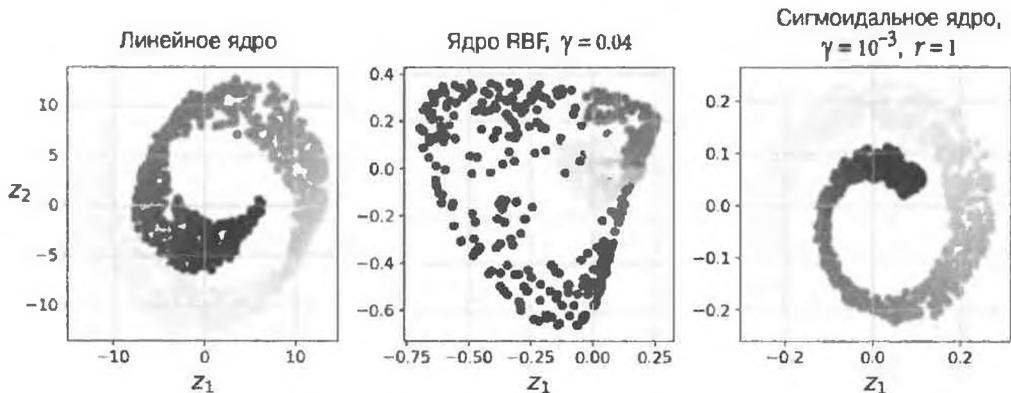


Рис. 8.10. Набор данных *Swiss roll*, пониженный до двух измерений за счет применения kPCA с разнообразными ядрами

<sup>6</sup> Бернхард Шолькопф и др., *Kernel Principal Component Analysis* (Ядерный анализ главных компонентов), *Lecture Notes in Computer Science 1327* (Берлин: Springer, 1997 г.): с. 583–588.

## Выбор ядра и подстройка гиперпараметров

Так как kPCA является алгоритмом обучения без учителя, нет никаких очевидных показателей эффективности, 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 эффективности на такой задаче. Показанный ниже код создает двухшаговый конвейер, сначала поникающий 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 гамма для kPCA с целью достижения наибольшей правильности классификации в конце конвейера:

```
from import GridSearchCV
from import LogisticRegression
from import Pipeline

clf = Pipeline([
    ("k pca", KernelPCA(n_components=2)),
    ("log_reg", LogisticRegression())
])

param_grid = [
    "k pca_gamma": np.linspace(0.03, 0.05, 10),
    "k pca_kernel": ["rbf", "sigmoid"]
]

grid_search = GridSearchCV(clf, param_grid, cv=3)
grid_search.fit(X, y)
```

Наилучшее ядро и гиперпараметры доступны через переменную best\_params\_:

```
>>> print(grid_search.best_params_)
{'k pca_gamma': 0.04333333333333335, 'k pca_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 вверху) и результирующий двумерный набор данных после применения kPCA с ядром RBF (справа вверху).

ва вверху). Благодаря ядерному трюку такая трансформация математически эквивалентна использованию карты признаков (*feature map*)  $\phi$  для отображения обучающего набора на бесконечномерное пространство признаков (справа внизу) и последующему проецированию трансформированного обучающего набора в двумерный набор с применением линейного PCA.

Обратите внимание, что если бы мы могли инвертировать шаг линейного алгоритма 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 в состоянии 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 гипериараметры, которые сводят к минимуму ошибку прообраза восстановления.

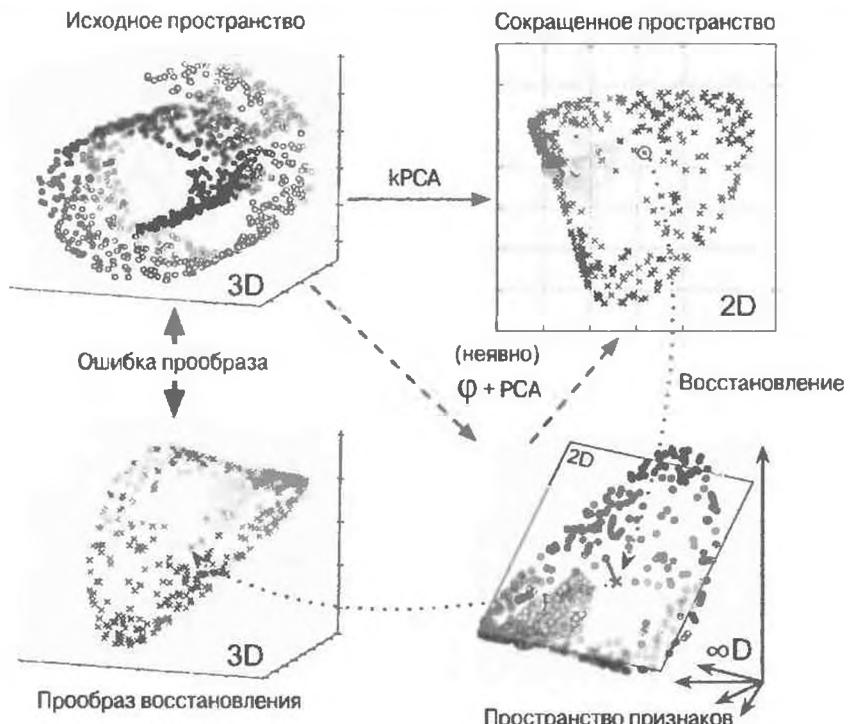


Рис. 8.11. Ядерный PCA и ошибка прообраза восстановления

Вас может интересовать, как выполнить такое восстановление. Одно из решений предусматривает обучение регрессионной модели с учителем, при котором спроектированные 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` в `True`, как демонстрируется в следующем коде<sup>7</sup>:

```
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`.

Затем можно подсчитать ошибку прообраза восстановления:

```
>>> from                  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 эту ошибку.

## LLE

Локальное линейное вложение (*Locally Linear Embedding — LLE*; <https://homl.info/lle>)<sup>8</sup> является еще одним мощным приемом нелинейного понижения размерности (*nonlinear dimensionality reduction — NDLR*). Это методика обучения на основе многообразий, которая не полагается на

<sup>7</sup> Если вы установите `fit_inverse_transform=True`, то библиотека Scikit-Learn будет использовать алгоритм (основанный на ядерной гребневой регрессии (Kernel Ridge Regression)), который описан в работе Гохана Бакира и др. *Learning to Find Pre-Images* (Обучение для нахождения прообразов) (<https://homl.info/34>), *Proceedings of the 16th International Conference on Neural Information Processing Systems* (2004 г.): с. 449–456.

<sup>8</sup> Сэм Ровайс и Лоуренс Саул, *Nonlinear Dimensionality Reduction by Locally Linear Embedding* (Нелинейное понижение размерности путем локального линейного вложения), *Science* 290, выпуск 5500 (2000 г.): с. 2323–2326.

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 подход LLE оказывается очень эффективным при развертывании скрученных многообразий, особенно когда шум не слишком большой.

В приведенном ниже коде с применением класса LocallyLinearEmbedding из Scikit-Learn развертывается набор данных Swiss roll:

```
from sklearn import LocallyLinearEmbedding
lle = LocallyLinearEmbedding(n_components= , n_neighbors= )
X_reduced = lle.fit_transform(X)
```

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 растянута, в то время как правая часть сдавлена. Несмотря на это, прием LLE выполнил неплохую работу по моделированию многообразия.

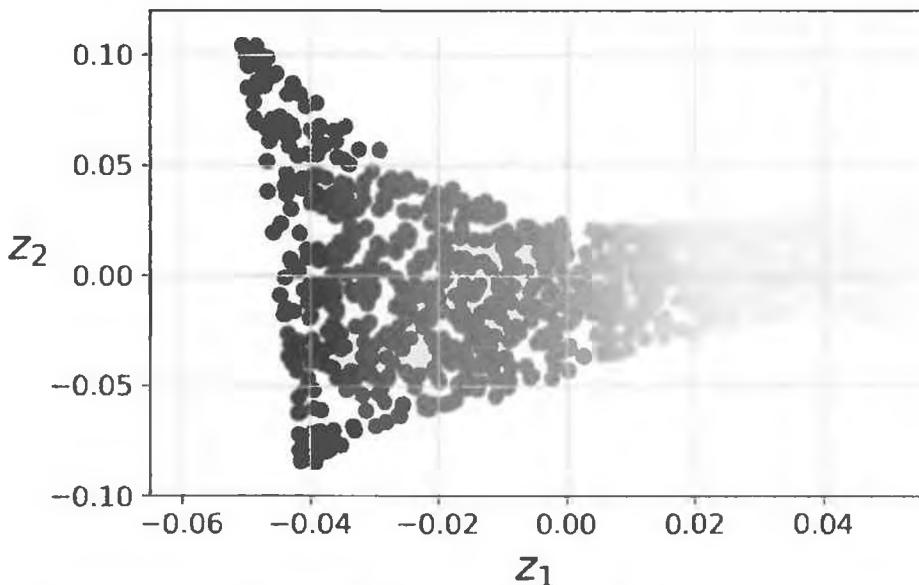


Рис. 8.12. Неразвернутый набор данных Swiss roll, использующий LLE

Вот как действует LLE: 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 веса  $w_{i,j}$ , такие что квадрат расстояния между  $\mathbf{x}^{(i)}$  и  $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$  был бы 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

$$\widehat{\mathbf{W}} = \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2$$

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 первого шага матрица весов  $\widehat{\mathbf{W}}$  (содержащая веса  $\widehat{w}_{i,j}$ ) представляет локальные линейные связи между обучающими образцами. Второй шаг заключается в отображении обучающих образцов на  $d$ -мерное пространство (где  $d < n$ ) с одновременным предохранением как можно большего числа имеющихся 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  $\sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)}$  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

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{z}^{(i)} - \sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Реализация алгоритма LLE в Scikit-Learn имеет следующую вычислительную сложность:  $O(m \log(m) n \log(k))$  для нахождения  $k$  ближайших соседей,  $O(mnk^3)$

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 методики понижения размерности

Существует много других методик понижения размерности, часть которых реализована в библиотеке Scikit-Learn; наиболее популярные из них описаны ниже.

### Случайное проецирование (*Random Projections*)

Как следует из названия, проецирует данные на пространство с меньшим количеством измерений, используя случайную линейную проекцию. Хотя это может показаться ненормальным, но оказывается, что такая случайная проекция на самом деле очень хорошо предохраняет расстояния, как было продемонстрировано Уильямом Джонсоном и Йорамом Линденштраусом в знаменитой лемме о малом искажении. Качество понижения размерности зависит от количества образцов и целевой размерности, но на удивление не от исходной размерности. Дополнительные детали доступны в документации по пакету `sklearn.random_projection`.

### Многомерное шкалирование (*Multidimensional Scaling — MDS*)

Понижает размерность, одновременно пытаясь предохранить расстояния между образцами.

### Изометрическое отображение (*Isomap*)

Создает граф, соединяя каждый образец с его ближайшими соседями, и затем понижает размерность, одновременно пытаясь предохранить геодезические расстояния (*geodesic distance*)<sup>9</sup> между образцами.

### Стochasticное вложение соседей с $t$ -распределением

#### (*t-distributed Stochastic Neighbor Embedding — t-SNE*)

Понижает размерность, одновременно пытаясь сохранять похожие образцы proximity и непохожие образцы на отдалении. Применяется главным образом для визуализации, в частности для визуального представления кластеров образцов в многомерном пространстве (например, при двумерной визуализации изображений MNIST).

<sup>9</sup> Геодезическое расстояние между двумя узлами в графе представляет собой количество узлов на кратчайшем пути между этими узлами.

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 такого подхода заключается 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, подобного SVM.

На рис. 8.13 показаны результаты применения нескольких перечисленных выше приемов.

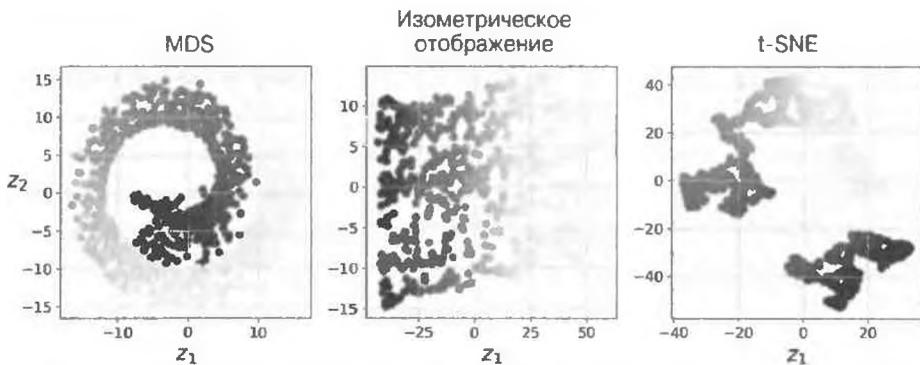


Рис. 8.13. Использование различных методик для понижения размерности набора данных *Swiss roll* до двух измерений

## 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 эффективность 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 (взьмите 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 заняло обучение, после чего оцените результатирующую 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 можете заменить каждую точку в графике рассеяния классом соответствующего образца (цифрой от 0 до 9) или даже рисовать версии с уменьшенным размером самих изображений цифр. (В случае вычерчивания всех цифр визуализация будет слишком перегруженной, поэтому имеет смысл 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 результатирующие визуализации.

Решения приведенных упражнений доступны в приложении A.



# Методики обучения без учителя

Хотя большинство приложений МО в наши дни основано на обучении с учителем (и как следствие именно сюда направлен основной объем инвестиций), подавляющая масса доступных данных являются непомеченными: мы имеем входные признаки  $X$ , но не располагаем метками  $y$ . Ученый в области компьютерных наук Ян Лекун превосходно отметил, что “если бы интеллект был тортом, то обучение без учителя было бы основой торта, обучение с учителем — сахарной глазурью на нем, а обучение с подкреплением — вишней на торте”. Другими словами, обучение без учителя обладает огромным потенциалом, куда мы только начали вонзать свои зубы.

Допустим, вы хотите создать систему, которая будет делать несколько фотографий каждого изделия на производственной линии и определять, какие из них дефектные. Вы можете довольно легко создать систему, которая будет автоматически делать снимки и ежедневно выдавать вам тысячи фотографий. Всего за несколько недель вы сумеете построить достаточно крупный набор данных. Но подождите, меток-то нет! Если вы пожелаете обучить обычный двоичный классификатор, который будет прогнозировать, является изделие дефектным или нет, то вам потребуется пометить все до единого фотографии изделий как “дефектное” или “нормальное”. Как правило, это будет требовать привлечения людей-экспертов, которым придется вручную перебрать все фотографии. Задача долгая, затратная и утомительная, так что она обычно будет выполняться для небольшого подмножества доступных фотографий. В результате помеченный набор данных будет довольно малым, а эффективность классификатора окажется неутешительной. Кроме того, каждый раз, когда компания вносит какие-то изменения в выпускаемые изделия, весь процесс необходимо начинать с нуля. Разве не было бы замечательно, если бы алгоритм сумел задействовать непомеченные данные, не заставляя людей помечать каждую фотографию? Вот тут на помощь и приходит обучение без учителя.

В главе 8 мы рассматривали самую распространенную задачу обучения без учителя: понижение размерности. В настоящей главе мы ознакомимся с дополнительными задачами и алгоритмами обучения без учителя.

## Кластеризация

Цель заключается в группировании похожих образцов в *кластеры*. Кластеризация представляет собой великолепный инструмент для анализа данных, сегментации заказчиков, систем выдачи рекомендаций, поисковых механизмов, сегментирования изображений, частичного обучения, понижения размерности и многого другого.

## Обнаружение аномалий

Целью является выяснение того, каким образом выглядят “нормальные” данные, с последующим применением этих знаний для обнаружения ненормальных образцов, таких как дефектные изделия на производственной линии или новая тенденция во временном ряде.

## Оценка плотности

Представляет собой задачу оценки функции плотности вероятности (*probability density function — PDF*) случайного процесса, который генерирует набор данных. Оценка плотности часто используется для обнаружения аномалий: образцы, расположенные в областях с очень низкой плотностью, вероятнее всего будут аномалиями. Она также полезна для анализа данных и визуализации.

Готовы к продолжению? Мы начнем с кластеризации посредством алгоритмов *K-Means* (*K-средние*) и *DBSCAN* (*Density-based Spatial Clustering of Applications with Noise* — плоскостная пространственная кластеризация приложений с шумом), после чего обсудим модели со смесями гауссовых распределений (*Gaussian mixture model*), а также посмотрим, каким образом их можно применять для оценки плотности, кластеризации и обнаружения аномалий.

## Кластеризация

Наслаждаясь прогулкой в горах, вы наталкиваетесь на растение, которого никогда не видели раньше. Вы оглядываетесь вокруг и замечаете еще несколько. Растения не идентичны, но достаточно похожи, чтобы вы знали, что

почти наверняка они относятся к одному и тому же виду (или, по крайней мере, к тому же самому роду). Вам может понадобиться ботаник, который сообщит точный вид, но вам определенно не нужен эксперт для идентификации групп сходно выглядящих объектов. Прием называется *кластеризацией* (*clustering*): это задача опознавания похожих образцов и назначения их *кластерам*, или группам похожих образцов.

Как и в классификации, каждый образец назначается группе. Однако в отличие от классификации кластеризация является задачей обучения без учителя. Взгляните на рис. 9.1: слева показан набор данных *iris* (введенный в главе 4), где вид каждого образца (т.е. его класс) представлен с помощью отличающегося маркера. Это помеченный набор данных, для которого хорошо подходят такие алгоритмы классификации, как логистическая регрессия, методы опорных векторов или случайный лес. Справа изображен тот же набор данных, но без меток, а потому использовать какой-то алгоритм классификации больше нельзя. Именно теперь в игру вступают алгоритмы кластеризации: многие из них способны легко обнаружить левый нижний кластер. Не настолько очевидно, но также довольно легко увидеть нашими собственными глазами, что правый верхний кластер состоит из двух отдельных подкластеров. Тем не менее, набор данных имеет два дополнительных признака (длину и ширину чашелистика), которые здесь не представлены, а алгоритмы кластеризации могут эффективно задействовать все признаки, поэтому они хорошо идентифицируют три кластера (например, в случае применения модели со смесью гауссовых распределений только 5 образцов из 150 назначаются неправильному кластеру).

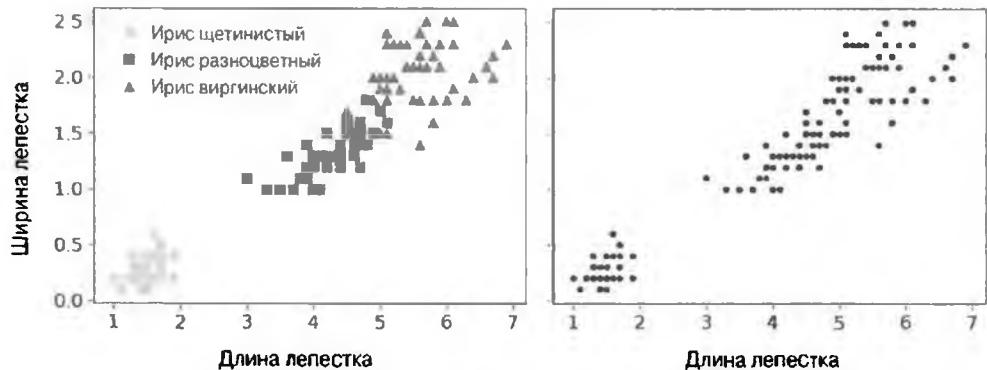


Рис. 9.1. Сравнение классификации (слева) и кластеризации (справа)

Кластеризация используется в разнообразных приложениях, включая перечисленные ниже.

### Для сегментации заказчиков

Вы можете кластеризовать своих заказчиков на основе их покупок и активности на вашем веб-сайте. Это полезно для понимания, кто ваши заказчики и в чем они нуждаются, так что вы сумеете адаптировать свои товары и маркетинговые кампании к каждому сегменту. Скажем, сегментация заказчиков может быть полезной в системах выдачи рекомендаций для предложения содержимого, которое понравилось другим пользователям в том же самом кластере.

### Для анализа данных

Когда вы анализируете новый набор данных, может быть полезно запустить алгоритм кластеризации и затем проанализировать каждый кластер по отдельности.

### В качестве методики понижения размерности

После кластеризации набора данных обычно появляется возможность измерить *похожесть (affinity)* каждого образца с каждым кластером (похожесть — это любая мера того, насколько хорошо образец подгоняется к кластеру). Затем вектор признаков  $\mathbf{x}$  каждого экземпляра может быть заменен вектором его похожестей с кластерами. Если есть  $k$  кластеров, тогда этот вектор будет  $k$ -мерным. Обычно он будет иметь гораздо меньшую размерность, чем исходный вектор признаков, но способен предохранять достаточно информации для дальнейшей обработки.

### Для обнаружения аномалий (также называемого обнаружением выбросов)

Любой образец, который имеет низкую похожесть со всеми кластерами, может быть аномалией. Например, если вы кластеризуете пользователей своего веб-сайта на основе их поведения, то сможете выявлять пользователей с необычным поведением, таким как нестандартное количество запросов в секунду. Обнаружение аномалий особенно полезно при выявлении дефектов в производстве или при обнаружении мошенничества.

### Для частичного обучения

Если вы располагаете незначительным числом меток, то могли бы выполнить кластеризацию и распространить метки на все образцы в од-

ном и том же кластере. Такая методика способна значительно увеличить количество меток, доступных для последующего алгоритма обучения с учителем, и тем самым увеличить его эффективность.

### Для поисковых механизмов

Определенные поисковые механизмы позволяют искать изображения, которые похожи на опорное изображение. Для построения такой системы вы сначала примените алгоритм кластеризации ко всем изображениям в вашей базе данных; похожие изображения окажутся в одном кластере. Затем, когда пользователь предоставит опорное изображение, вам придется лишь воспользоваться обученной моделью кластеризации для нахождения кластера этого изображения и просто возвратить все изображения из найденного кластера.

### Для сегментирования изображений

За счет кластеризации пикселей в соответствии с их цветом и последующей замены цвета каждого пикселя усредненным цветом его кластера становится возможным значительное сокращение количества разных цветов в изображении. Сегментирование изображений применяется во многих системах обнаружения и отслеживания объектов, т.к. оно облегчает распознавание контуров каждого объекта.

Не существует универсального определения того, что такое кластер: это действительно зависит от контекста, и разные алгоритмы будут захватывать отличающиеся виды кластеров. Одни алгоритмы ищут образцы, центрированные возле конкретной точки, называемой центроидом. Другие занимаются поиском непрерывных областей плотно упакованных образцов: такие кластеры могут принимать любую форму. Некоторые алгоритмы являются иерархическими и ищут кластеры кластеров. Список можно продолжить.

В текущем разделе мы исследуем два популярных алгоритма кластеризации, K-Means и DBSCAN, а также ознакомимся с их приложениями вроде нелинейного понижения размерности, частичного обучения и обнаружения аномалий.

## K-Means

Рассмотрим непомеченный набор данных, представленный на рис. 9.2: вы можете четко заметить пять пятен образцов. Алгоритм K-Means — это простой алгоритм, который способен выполнить кластеризацию набора данных

такого рода очень быстро и эффективно, часто всего лишь за несколько итераций. Он был предложен Стюартом Ллойдом из Bell Labs в 1957 году как методика импульсно-кодовой модуляции, но опубликован за пределами компании только в 1982 году (<https://homl.info/36>)<sup>1</sup>. В 1965 году Эдвард Форги опубликовал фактически тот же самый алгоритм и потому K-Means иногда называют алгоритмом Ллойда-Форги.

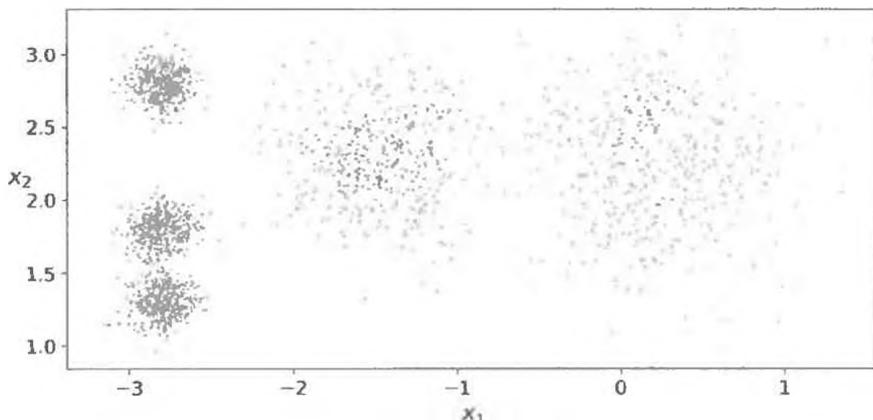


Рис. 9.2. Непомеченный набор данных, состоящий из пяти пятен образцов

Давайте обучим кластеризатор K-Means на таком наборе данных. Он будет пытаться отыскать центр каждого пятна и назначить каждый образец ближайшему пятну:

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

Обратите внимание, что вы должны указать количество кластеров  $k$ , которые алгоритм обязан найти. Взглянув на данные в текущем примере, вполне очевидно, что  $k$  необходимо установить в 5, но в целом все не так легко. Вскоре мы обсудим суть более подробно.

Каждый образец был назначен одному из пяти кластеров. В контексте кластеризации метка образца является индексом кластера, который алгоритм назначил образцу: ее не следует путать с метками классов в классифи-

<sup>1</sup> Стюарт Ллойд, *Least Squares Quantization in PCM* (Квантование методом наименьших квадратов в импульсно-кодовой модуляции), *IEEE Transactions on Information Theory* 28, выпуск 2 (1982 г.): с. 129–137.

кации (вспомните, что кластеризация представляет собой задачу обучения без учителя). Экземпляр KMeans сохраняет копию меток образцов, на которых он обучался, доступную через переменную экземпляра `labels_`:

```
>>> y_pred  
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)  
>>> y_pred is kmeans.labels_  
True
```

Вы также можете посмотреть на пять центроидов, которые обнаружил алгоритм:

```
>>> kmeans.cluster_centers_  
array([[-2.80389616,  1.80117999],  
       [ 0.20876306,  2.25551336],  
       [-2.79290307,  2.79641063],  
       [-1.46679593,  2.28585348],  
       [-2.80037642,  1.30082566]])
```

Вы легко можете назначить новые образцы кластеру, чей центройд находится ближе всего:

```
>>> X_new = np.array([[ ,  ], [ ,  ], [- ,  ], [- ,  ]])  
>>> kmeans.predict(X_new)  
array([1, 1, 2, 2], dtype=int32)
```

Если вы вычертите границы решений кластеров, то получите диаграмму (или замощение) Вороного (Voronoi tessellation), показанную на рис. 9.3, где каждый центройд представлен посредством X.

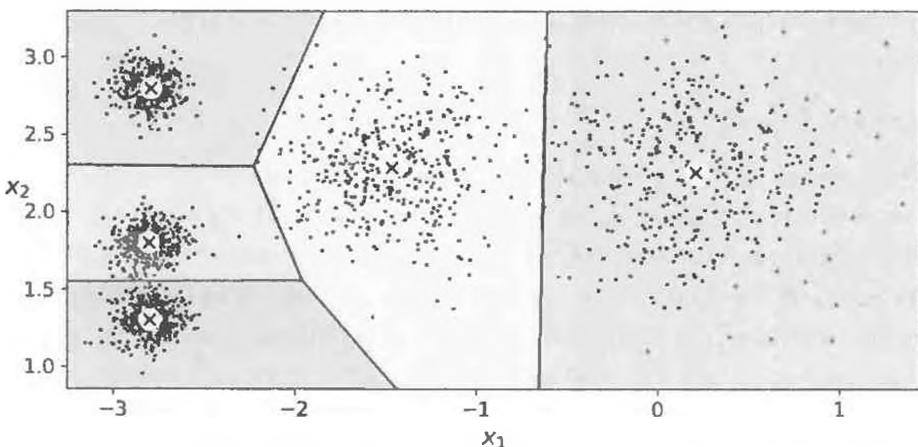


Рис. 9.3. Границы решений алгоритма K-Means (диаграмма Вороного)

Подавляющее большинство образцов были четко назначены надлежащим кластерам, но несколько образцов оказались, возможно, неправильно помеченными (особенно вблизи границы между левым верхним и центральным кластерами). В действительности алгоритм K-Means не очень хорошо работает, когда пятна имеют сильно отличающиеся диаметры, поскольку при назначении образца кластеру его заботит только расстояние до центроида.

Вместо назначения каждого образца одиночному кластеру, что называется *жесткой кластеризацией* (*hard clustering*), временами полезно выдавать каждому образцу оценку для каждого кластера, что называется *мягкой кластеризацией* (*soft clustering*). Оценкой может быть расстояние между образцом и центроидом; и наоборот, это может быть оценка близости (или похожести), такая как *гауссова радиальная базисная функция* (*Gaussian RBF*), представленная в главе 5. Метод `transform()` класса `KMeans` измеряет расстояние от каждого образца до всех центроидов:

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

В рассмотренном примере первый образец в `X_new` расположен на расстоянии 2.81 от первого центроида, 0.33 от второго центроида, 2.90 от третьего центроида, 1.49 от четвертого центроида и 2.89 от пятого центроида. Если вы трансформируете подобным образом имеющийся многомерный набор данных, то в итоге получите  $k$ -мерный набор данных: такая трансформация может быть очень эффективной методикой нелинейного понижения размерности.

## Алгоритм K-Means

Итак, каким же образом работает алгоритм K-Means? Предположим, что вы получили центроиды. Вы могли бы легко снабдить метками все образцы в наборе данных, назначая каждый из них кластеру, чей центроид находится ближе всего. И наоборот, если бы вам предоставили все метки образцов, тогда вы легко смогли бы найти все центроиды, вычислив среднее по образцам для каждого кластера. Но вам не дают ни меток, ни центроидов — как вы сможете продолжить? Хорошо, просто начните со случайного размещения центроидов (скажем, выберите  $k$  образцов произвольным образом и исполь-

зуйте их местоположения в качестве центроидов). Затем снабдите метками образцы, обновите центроиды, снабдите метками образцы, обновите центроиды и продолжайте делать это до тех пор, пока центроиды прекратят перемещаться. Алгоритм гарантированно сходится за конечное число шагов (обычно довольно малое); он не будет колебаться вечно<sup>2</sup>.

На рис. 9.4 алгоритм демонстрируется в действии: центроиды инициализируются случайным образом (слева вверху), затем помечаются образцы (справа вверху), далее центроиды обновляются (слева по центру), образцы помечаются повторно (справа по центру) и т.д. Как видите, всего лишь за три итерации алгоритм добился кластеризации, которая выглядит близкой к оптимальной.

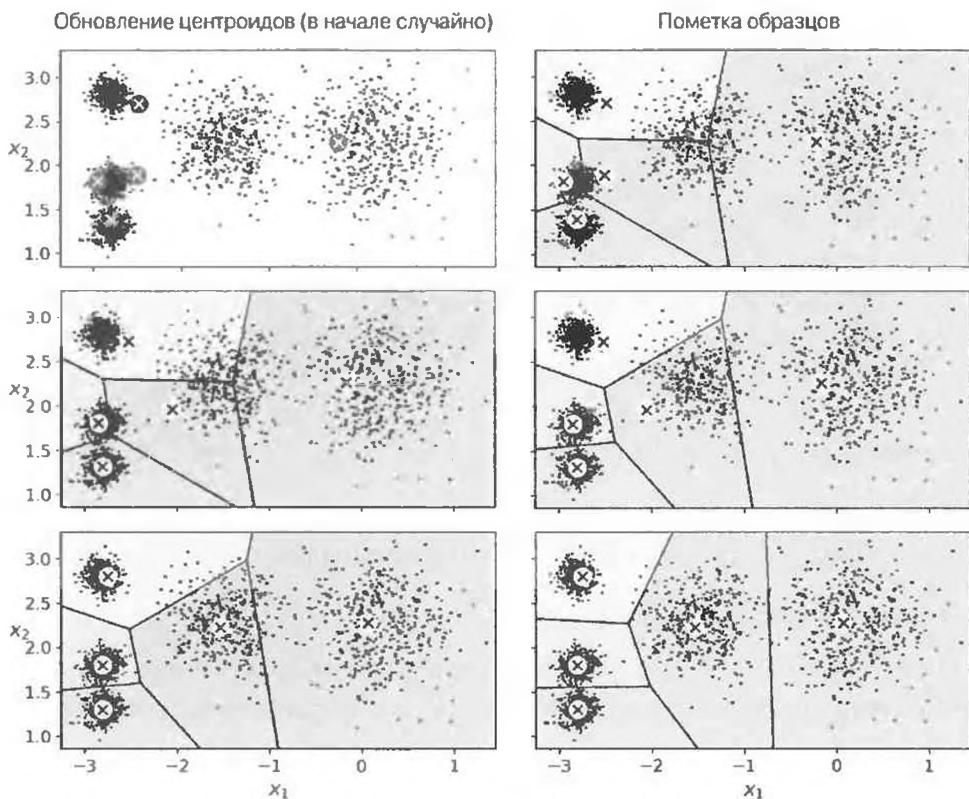


Рис. 9.4. Алгоритм K-Means в действии

<sup>2</sup> Причина в том, что среднеквадратическое расстояние между образцами и их ближайшими центроидами на каждом шаге может только уменьшаться.



Вычислительная сложность алгоритма обычно линейна в отношении количества образцов  $m$ , количества кластеров  $k$  и количества измерений  $n$ . Однако утверждение справедливо, только когда данные обладают кластерной структурой. Если кластерной структуры нет, то в худшем случае сложность может расти экспоненциально с увеличением количества образцов. На практике подобное происходит редко и, как правило, K-Means является одним из самых быстрых алгоритмов кластеризации.

Хотя алгоритм гарантированно сходится, он может не сойтись в правильное решение (т.е. сойтись в локальный оптимум): все зависит от инициализации центроидов. На рис. 9.5 показаны два субоптимальных решения, в которые может сходиться алгоритм, если шаг случайной инициализации окажется неудачным.

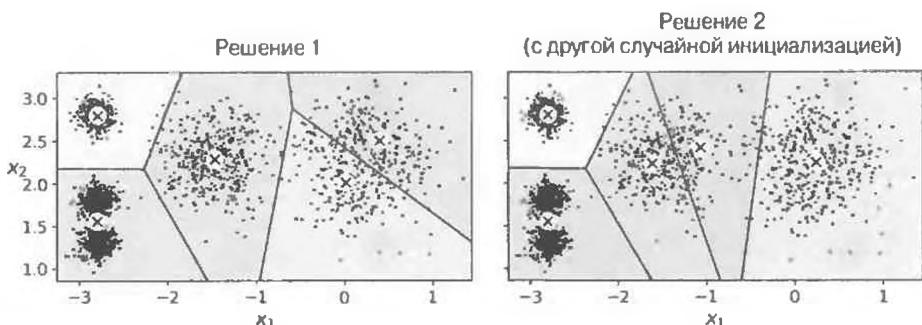


Рис. 9.5. Субоптимальные решения из-за неудачной инициализации центроидов

Давайте рассмотрим несколько способов уменьшения риска попадания в такую ситуацию путем совершенствования инициализации центроидов.

### Методы инициализации центроидов

Если так случилось, что вы знаете, где приблизительно должны располагаться центроиды (например, по причине того, что ранее запускали еще один алгоритм кластеризации), тогда можете установить гиперпараметр `init` в массив NumPy, содержащий список центроидов, и установить `n_init` в 1:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Другое решение предусматривает многократный запуск алгоритма с различными случайными инициализациями и сохранение наилучшего решения. Количество

случайных инициализаций управляет гиперпараметром `n_init`: по умолчанию он равен 10, т.е. при вызове `fit()` описанный ранее полный алгоритм выполняется 10 раз и библиотека Scikit-Learn сохраняет наилучшее решение. Но как в точности она узнает, какое решение является наилучшим? Библиотека использует показатель эффективности! Такой показатель называется *инерцией* модели, которая представляет собой среднеквадратическое расстояние между каждым образцом и его ближайшим центроидом. Оно составляет приблизительно 223.3 для модели слева на рис. 9.5, 237.5 для модели справа на рис. 9.5 и 211.6 для модели на рис. 9.3. Класс `KMeans` запускает алгоритм `n_init` раз и сохраняет модель с самой низкой инерцией. В данном примере будет выбрана модель на рис. 9.3 (если только крайне не повезет с `n_init` последовательными случайными инициализациями). Если вам интересно, то инерция модели доступна через переменную экземпляра `inertia_`:

```
>>> kmeans.inertia_
211.59853725816856
```

Метод `score()` возвращает отрицательную инерцию. Почему отрицательную? Потому что метод `score()` прогнозатора обязан всегда соблюдать правило “больше значит лучше” библиотеки Scikit-Learn: если один прогнозатор лучше другого, то его метод `score()` должен возвращать более высокую оценку.

```
>>> kmeans.score(X)
-211.59853725816856
```

Дэвид Артур и Сергей Васильевский в своей статье 2006 года (<https://hml.info/37>)<sup>3</sup> предложили важное усовершенствование алгоритма K-Means под названием *K-Means++*. Они ввели более интеллектуальный шаг инициализации, имеющий тенденцию выбирать центроиды, которые находятся далеко друг от друга, и такое усовершенствование значительно снижает вероятность сходжения алгоритма K-Means в субоптимальное решение. Авторы статьи показали, что дополнительные вычисления, требуемые на более интеллектуальном шаге инициализации, вполне того стоят, поскольку они позволяют радикально сократить количество запусков алгоритма для нахождения оптимального решения.

<sup>3</sup> Дэвид Артур и Сергей Васильевский, *K-Means++: The Advantages of Careful Seeding (K-Means++: преимущества осмотрительной инициализации)*, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007 г.): с. 1027–1035.

Ниже приведен алгоритм инициализации K-Means++.

1. Взять один центроид  $c^{(1)}$ , выбирая равномерно случайно из набора данных.
2. Взять новый центроид  $c^{(i)}$ , выбирая образец  $x^{(i)}$  с вероятностью  $D(x^{(i)})^2 / \sum_{j=1}^m D(x^{(j)})^2$ , где  $D(x^{(i)})$  — расстояние между образцом  $x^{(i)}$  и ближайшим центроидом, который уже был выбран. Такое распределение вероятностей гарантирует, что в качестве центроидов с гораздо большей вероятностью будут выбираться образцы, расположенные дальше от уже выбранных центроидов.
3. Повторять предыдущий шаг до тех пор, пока не будут выбраны все  $k$  центроидов.

Класс KMeans применяет описанный метод инициализации по умолчанию. Если вы хотите, чтобы он применял исходный метод (т.е. случайный выбор  $k$  образцов для определения начальных центроидов), тогда можете установить гиперпараметр `init` в "random". Поступать так придется редко.

### Ускоренный K-Means и мини-пакетный K-Means

Чарльз Элкан в своей статье 2003 года (<https://homl.info/38>)<sup>4</sup> предложил еще одно важное усовершенствование алгоритма K-Means. Оно значительно ускоряет алгоритм, избегая многих ненужных расчетов расстояний. Элкан достиг такого результата за счет использования неравенства треугольника (т.е. что кратчайшим расстоянием между двумя точками всегда является прямая линия<sup>5</sup>), а также отслеживания нижней и верхней границ для расстояний между образцами и центроидами. Именно данный алгоритм класс KMeans применяет по умолчанию (вы можете принудительно использовать исходный алгоритм, установив гиперпараметр `algorithm` в "full", но вряд ли это когда-либо понадобится).

Дэвид Скалли в своей статье 2010 года (<https://homl.info/39>)<sup>6</sup> предложил очередной важный вариант алгоритма K-Means. Вместо применения

<sup>4</sup> Чарльз Элкан, *Using the Triangle Inequality to Accelerate K-Means* (Использование неравенства треугольника для ускорения алгоритма K-Means), *Proceedings of the 20th International Conference on Machine Learning* (2003 г.): с. 147–153.

<sup>5</sup> Неравенство треугольника выглядит как  $AC \leq AB + BC$ , где А, В и С — три точки, а  $AB$ ,  $AC$  и  $BC$  — расстояния между этими точками.

<sup>6</sup> Дэвид Скалли, *Web-Scale K-Means Clustering* (Масштабирование кластеризации K-Means для веб-приложений), *Proceedings of the 19th International Conference on World Wide Web* (2010 г.): с. 1177–1178.

на каждой итерации полного набора данных алгоритм способен использовать мини-пакеты, лишь слегка перемещая центроиды на каждой итерации. Прием ускоряет алгоритм обычно в три или четыре раза и позволяет кластеризовать гигантские наборы данных, которые не умещаются в память. В библиотеке Scikit-Learn этот алгоритм реализован в классе MiniBatchKMeans. Его можно применять аналогично классу KMeans:

```
from sklearn.cluster import MiniBatchKMeans  
minibatch_kmeans = MiniBatchKMeans(n_clusters= )  
minibatch_kmeans.fit(X)
```

Если набор данных не умещается в памяти, то простейшим вариантом будет использование класса `memmap`, как делалось для инкрементного анализа главных компонентов в главе 8. В качестве альтернативы можно передавать методу `partial_fit()` по одному мини-пакету за раз, но такой подход требует намного больше работы, потому что придется самостоятельно выполнять множество инициализаций и выбирать наилучшую из них (ищите пример в разделе “Mini-Batch K-Means” (Мини-пакетный K-Means) тетради Jupyter для настоящей главы).

Несмотря на то что мини-пакетный алгоритм K-Means гораздо быстрее обычного алгоритма K-Means, его инерция чуть хуже, особенно при увеличении количества кластеров. Вы можете видеть это на рис. 9.6: на графике слева сравнивается инерция моделей мини-пакетного K-Means и обычного K-Means, обученных на предыдущем наборе данных с применением разного количества кластеров  $k$ .

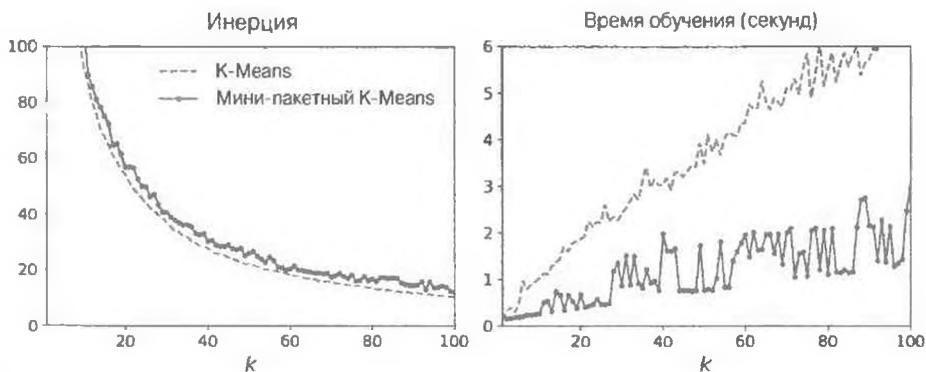


Рис. 9.6. Мини-пакетный алгоритм K-Means имеет более высокую инерцию, чем обычный K-Means (график слева), но гораздо быстрее (график справа), особенно при увеличении  $k$

Разность между двумя кривыми остается почти постоянной, но с ростом  $k$  становится все более и более значительной, т.к. инерция все меньше и меньше. На графике справа можно заметить, что мини-пакетный K-Means намного быстрее, чем обычный K-Means, и разница растет с увеличением  $k$ .

### Нахождение оптимального количества кластеров

До сих пор мы устанавливали количество кластеров  $k$  в 5, поскольку по данным было видно, что это корректное значение. Но в общем случае будет нелегко узнать, каким образом устанавливать  $k$ , а результат может оказаться довольно плохим, если установить  $k$  в неверное значение. Как показано на рис. 9.7, установка  $k$  в 3 или 8 дает явно неудачные модели.

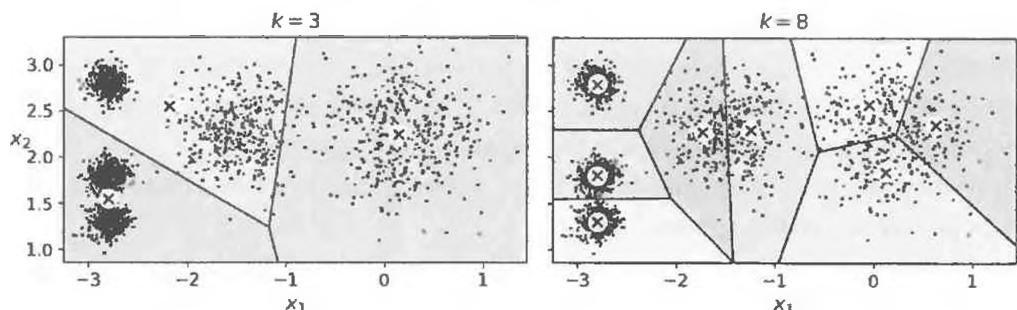


Рис. 9.7. Неудачный выбор количества кластеров: когда значение  $k$  слишком малое, обособленные кластеры сливаются (слева), а когда значение  $k$  чрезмерно большое, некоторые кластеры рассекаются на множество фрагментов (справа)

Вы могли подумать, что достаточно было бы выбрать модель с самой низкой инерцией, верно? К сожалению, все не так просто. Инерция для  $k = 3$  составляет 653.2, что намного выше, чем инерция для  $k = 5$  (которая была 211.6). Но в случае  $k = 8$  инерция равна 119.1. При попытке выбора  $k$  инерция не является подходящим показателем эффективности, потому что с ростом  $k$  она продолжает уменьшаться. На самом деле, чем больше кластеров, тем ближе будет каждый образец к своему ближайшему центроиду и, следовательно, тем ниже будет инерция. Давайте вычертим график инерции как функции количества кластеров  $k$  (рис. 9.8).

Как видите, инерция очень быстро падает при увеличении  $k$  вплоть до 4, но затем при дальнейшем росте  $k$  уменьшается гораздо медленнее. По форме кривая напоминает руку, имеющую "локоть" в точке  $k = 4$ .

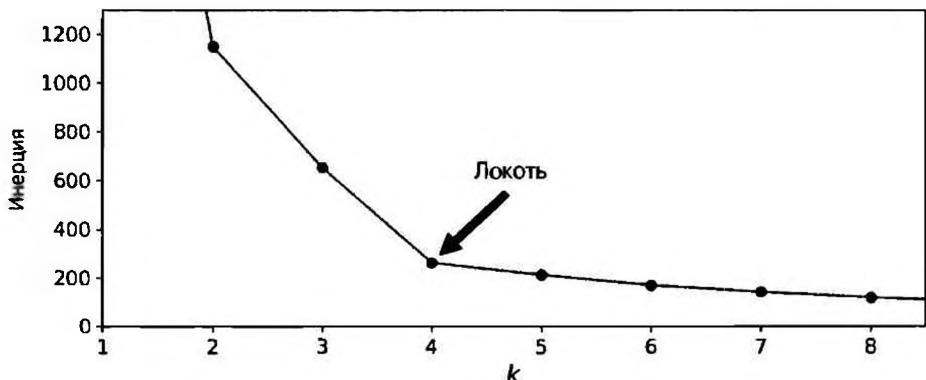


Рис. 9.8. Когда вычерчивается график инерции как функции количества кластеров  $k$ , кривая часто содержит точку перегиба, называемую “локтем”

Итак, если бы мы не знали лучший вариант, то 4 было бы хорошим выбором: любое более низкое значение окажется неподходящим, в то время как любое более высокое значение не особо поможет — мы бы просто разделяли вполне нормальные кластеры пополам без веской причины.

Такая методика выбора наилучшего значения для количества кластеров является довольно грубой. Более точный подход (но более затратный с точки зрения вычислений) предусматривает использование оценки силуэта (*silhouette score*), которая представляет собой средний коэффициент силуэта (*silhouette coefficient*) по всем образцам. Коэффициент силуэта образца равен  $(b - a) / \max(a, b)$ , где  $a$  — среднее расстояние до других образцов в том же самом кластере (т.е. среднее внутрикластерное расстояние) и  $b$  — среднее расстояние до ближайшего кластера (т.е. среднее расстояние до образцов следующего ближе всех расположенного кластера, определяемого как тот, который минимизирует  $b$ , исключая собственный кластер образца). Коэффициент силуэта может варьироваться между  $-1$  и  $+1$ . Коэффициент, близкий к  $+1$ , означает, что образец находится внутри своего кластера и далеко от других кластеров; коэффициент, близкий к  $0$ , говорит о том, что образец расположен близко к границе кластера; коэффициент, близкий к  $-1$ , указывает на то, что образец мог быть назначен ошибочному кластеру.

Для вычисления оценки силуэта можно применять функцию `silhouette_score()` из Scikit-Learn, предоставив ей все образцы в наборе данных и метки, которые были назначены:

```
>>> from import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

Давайте сравним оценки силуэтов для разного количества кластеров (рис. 9.9).

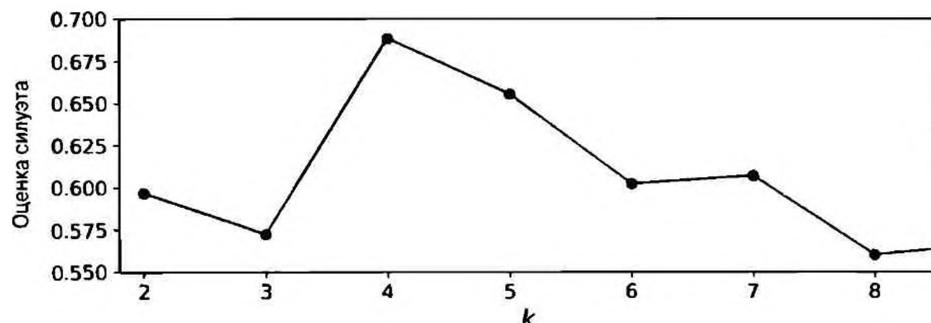
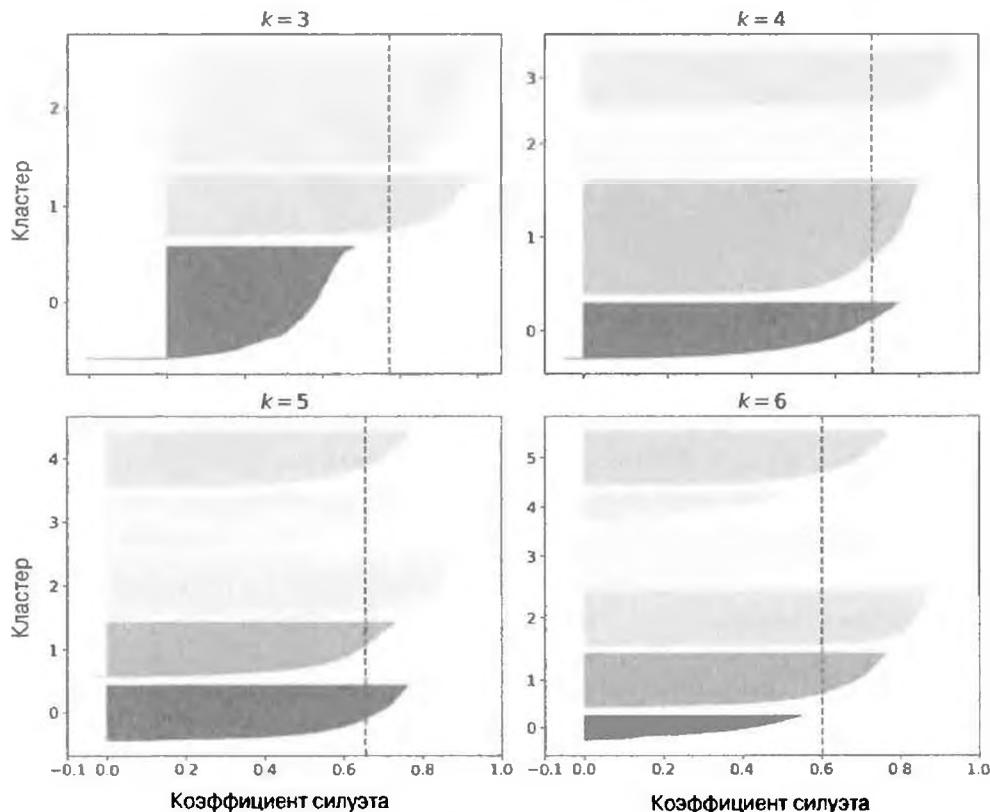


Рис. 9.9. Выбор количества кластеров  $k$  с использованием оценки силуэта

Легко заметить, что такая визуализация гораздо информативнее предыдущей: хотя она подтверждает, что  $k = 4$  является хорошим выбором, она также подчеркивает тот факт, что  $k = 5$  — тоже неплохой вариант, который намного лучше, чем  $k = 6$  или  $k = 7$ . При сравнении инерции этого не было видно.

Даже еще более информативная визуализация получится, когда вычертить график коэффициентов силуэтов всех образцов, отсортированные по кластерам, которым они назначены, и по значениям коэффициента. Такой график называется *диаграммой силуэтов (silhouette diagram)* и приведен на рис. 9.10. Каждая диаграмма содержит по одной форме, напоминающей лезвие ножа, на кластер. Высота формы показывает количество образцов, находящихся в кластере, а ее ширина представляет отсортированные коэффициенты силуэтов образцов в кластере (чем шире, тем лучше). Пунктирной линией обозначается средний коэффициент силуэта.

Вертикальные пунктирные линии представляют оценку силуэта для заданного количества кластеров. Когда большинство образцов в кластере имеют коэффициент меньше данной оценки (т.е. если многие образцы останавливаются практически на пунктирной линии, заканчиваясь слева от нее), то кластер довольно плох, поскольку это означает, что его образцы находятся слишком близко к другим кластерам. Мы видим, что при  $k = 3$  и  $k = 6$  получаются плохие кластеры. Но при  $k = 4$  или  $k = 5$  кластеры выглядят вполне хорошо: большинство образцов простираются за пунктирную линию вправо и ближе к 1.0. Когда  $k = 4$ , кластер с индексом 1 (третий сверху) великоват. Когда  $k = 5$  все кластеры имеют похожие размеры.

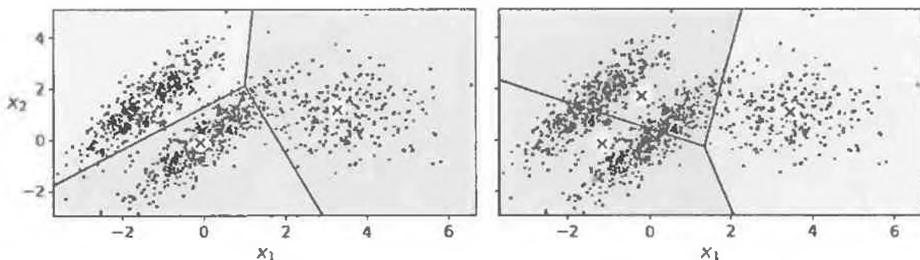


*Рис. 9.10. Анализ диаграммы силуэттом для различных значений  $k$*

Таким образом, хотя общая оценка силуэта при  $k=4$  чуть больше, чем при  $k=5$ , представляется неплохой идеей выбрать  $k=5$  для получения кластеров похожих размеров.

## Ограничения K-Means

Несмотря на многие достоинства, особенно высокую скорость и масштабируемость, алгоритм K-Means отнюдь не совершенен. Мы видели, что алгоритм необходимо запускать несколько раз, чтобы избежать субоптимальных решений, а также указывать количество кластеров, что может оказаться затруднительным. Кроме того, алгоритм K-Means ведет себя не очень хорошо, когда кластеры имеют варьирующиеся размеры, отличающиеся плотности или несферические формы. Например, на рис. 9.11 показано, как K-Means кластеризирует набор данных, содержащий три эллиптических пятна с разным размером, плотностью и ориентацией.



**Рис. 9.11.** Алгоритм K-Means терпит неудачу с надлежащей кластеризацией таких эллиптических пятен

Как видите, ни одно из решений не годится. Решение слева лучше, но оно по-прежнему отсекает 25% образцов среднего кластера и назначает их кластеру справа. Решение справа просто ужасно, хоть его инерция ниже. Таким образом, в зависимости от данных разные алгоритмы кластеризации могут работать лучше. С эллиптическими пятнами подобного рода великолепноправляются модели со смесями гауссовых распределений.



Перед запуском алгоритма K-Means важно масштабировать входные признаки, иначе кластеры могут оказаться очень растянутыми, и K-Means будет выполняться плохо. Масштабирование признаков вовсе не гарантирует, что все кластеры станут элегантными и сферическими, но в целом улучшает ситуацию.

Теперь давайте взглянем на несколько способов получения преимущества от кластеризации. Мы будем применять алгоритм K-Means, но вы вольны экспериментировать и с другими алгоритмами кластеризации.

## Использование кластеризации для сегментирования изображений

Сегментирование изображений представляет собой задачу разделения изображения на множество сегментов. При *семантическом сегментировании* все пиксели, являющиеся частью одного и того же типа объекта, назначаются тому же самому сегменту. Скажем, в системе машинного зрения беспилотного автомобиля все пиксели, которые являются частью изображения пешехода, могут быть назначены сегменту “пешеход” (будет один сегмент, содержащий всех пешеходов). При *сегментировании образцов* все пиксели, являющиеся частью того же индивидуального объекта, назначаются тому же самому сегменту. В данном случае для каждого пешехода будет отдельный сегмент.

Современный технический уровень семантического сегментирования и сегментирования образцов достигается с применением сложных архитектур на основе сверточных нейронных сетей (см. главу 14). Здесь мы собираемся поступить намного проще: делать *сегментирование цветов*. Мы будем назначать пиксели тому же самому сегменту, если они имеют похожий цвет. В ряде приложений этого может быть достаточно. Например, если вы хотите анализировать спутниковые снимки с целью измерения общей площади лесов в какой-то области, тогда вполне подойдет сегментация цветов.

Для начала с использованием функции `imread()` из библиотеки `Matplotlib` загрузим изображение (показано слева вверху на рис. 9.12):

```
>>> from           import imread # или from imageio import imread
>>> image = imread(os.path.join("images", "unsupervised_learning",
                                 "adybg.png"))
>>> image.shape
(533, 800, 3)
```

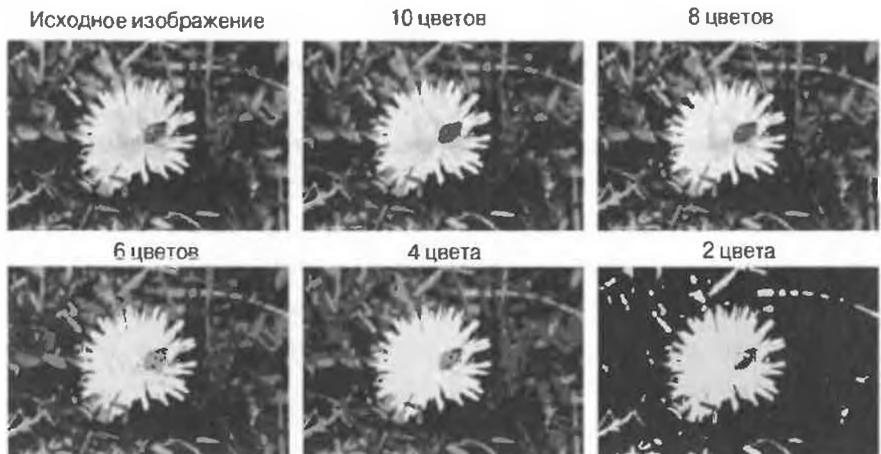


Рис. 9.12. Сегментация изображения посредством алгоритма *K-Means* с различным количеством цветовых кластеров

Изображение представлено в виде трехмерного массива. Размером первого измерения является высота, второго — ширина, а третьего — количество цветовых каналов, в данном случае красного, зеленого и синего (red, green, blue — RGB). Другими словами, для каждого пикселя предусмотрен трехмерный вектор, содержащий интенсивности красного, зеленого и синего, каждая в пределах от 0.0 до 1.0 (или от 0 до 255, если применяется `imageio.imread()`). Некоторые изображения могут иметь меньше каналов, например, полуточечные (один канал). Другие изображения могут располагать большим чис-

лом каналов, такие как изображения с дополнительным альфа-каналом для прозрачности или спутниковые снимки, которые часто содержат каналы для многих световых частот (скажем, инфракрасный). Следующий код изменяет форму массива с целью получения длинного списка цветов RGB и затем кластеризирует цвета, используя K-Means:

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

Например, код может идентифицировать цветовой кластер для всех оттенков зеленого. Затем для каждого цвета (пусть темно-зеленого) он ищет средний цвет цветового кластера пикселя. Скажем, все оттенки зеленого могут быть заменены одинаковым светло-зеленым цветом (предполагается, что светло-зеленый — средний цвет зеленого кластера). В заключение код изменяет форму длинного списка цветов, чтобы получить такую же форму, как у исходного изображения. Готово!

Выходное изображение представлено справа вверху на рис. 9.12, где также иллюстрируются результаты экспериментов с разным количеством кластеров. Обратите внимание, что в случае применения менее восьми кластеров бросающимся в глаза красному цвету божьей коровки не удается получить собственный кластер: он сливаются с цветами из окружения. Причина в том, что алгоритм K-Means отдает предпочтение кластерам похожих размеров. Божья коровка маленькая — намного меньше остальной части изображения — и хотя она имеет бросающийся в глаза цвет, K-Means отказывается выделить кластер для него.

Это было не слишком сложно, не так ли? А теперь давайте посмотрим на еще одно приложение кластеризации: предварительную обработку.

## Использование кластеризации для предварительной обработки

Кластеризация может служить эффективным подходом понижения размерности, в особенности как шаг предварительной обработки перед выполнением какого-то алгоритма обучения с учителем. В качестве примера применения кластеризации для понижения размерности мы займемся набором данных `digits` — простым MNIST-подобным набором данных, который содержит 1797 полутоновых изображений  $8 \times 8$ , представляющих цифры от 0 до 9. Первым делом загрузим набор данных:

```
from import load_digits  
X_digits, y_digits = load_digits(return_X_y=True)
```

Теперь расщепим его на обучающий набор и испытательный набор:

```
from import train_test_split  
X_train, X_test, y_train, y_test = train_test_split(X_digits,  
y_digits)
```

Далее произведем подгонку логистической регрессионной модели:

```
from import LogisticRegression  
log_reg = LogisticRegression()  
log_reg.fit(X_train, y_train)
```

Оценим ее правильность на испытательном наборе:

```
>>> log_reg.score(X_test, y_test)  
0.9688888888888889
```

Итак, мы имеем базовый уровень: правильность 96.9%. Давайте посмотрим, можем ли мы добиться улучшения, используя K-Means как шаг предварительной обработки. Мы создадим конвейер, который сначала кластеризует обучающий набор в 50 кластеров и заменит изображения их расстояниями до полученных 50 кластеров, а затем применит логистическую регрессионную модель:

```
from import Pipeline  
pipeline = Pipeline([  
    ("!", KMeans(n_clusters=50)),  
    ("!", log_reg),  
)  
pipeline.fit(X_train, y_train)
```



Поскольку имеется 10 разных цифр, заманчиво установить количество кластеров в 10. Тем не менее, каждая цифра может быть записана несколькими отличающимися способами, поэтому предпочтительнее использовать большее количество кластеров, такое как 50.

Оценим созданный конвейер классификации:

```
>>> pipeline.score(X_test, y_test)  
0.9777777777777777
```

Что скажете? Мы сократили частоту ошибок почти на 30% (от приблизительно 3.1% до примерно 2.2%)!

Но мы выбрали количество кластеров  $k$  произвольным образом; безусловно, ситуацию можно еще улучшить. Из-за того, что алгоритм K-Means представляет собой лишь шаг предварительной обработки в конвейере классификации, найти подходящее значение для  $k$  гораздо проще, чем было ранее. Нет никакой необходимости проводить анализ силузтов или минимизировать инерцию; наилучшим значением  $k$  будет то, которое в итоге обеспечивает наилучшую эффективность классификации во время перекрестной проверки. Мы можем применить GridSearchCV для нахождения оптимального количества кластеров:

```
from sklearn.cluster import KMeans
param_grid = dict(kmeans_n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=5, verbose=1)
grid_clf.fit(X_train, y_train)
```

Давайте взглянем на наилучшее значение для  $k$  и эффективность результирующего конвейера:

```
>>> grid_clf.best_params_
{'kmeans_n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

При  $k=99$  кластеров мы получаем значительное повышение правильности, достигающее 98.22% на испытательном наборе. Отлично! Возможно, вы пожелаете продолжить исследование на более высоких значениях для  $k$ , т.к. 99 было наибольшим значением в заданном нами диапазоне.

## Использование кластеризации для частичного обучения

Еще один сценарий использования кластеризации касается частичного обучения, когда мы имеем обилие непомеченных и совсем немного помеченных образцов. Обучим логистическую регрессионную модель на выборке в 50 помеченных образцов из набора данных digits:

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

Какова эффективность этой модели на испытательном наборе?

```
>>> log_reg.score(X_test, y_test)
0.8333333333333334
```

Правильность составляет только 83.3%. Не должен вызывать удивления тот факт, что она намного ниже, чем ранее, когда мы обучали модель на полном обучающем наборе. Давайте посмотрим, как мы можем улучшить положение дел. Для начала кластеризируем обучающий набор в 50 кластеров. Затем для каждого кластера отыщем изображение, ближайшее к центроиду. Мы будем называть такие изображения *репрезентативными*:

```
k = 50  
kmeans = KMeans(n_clusters=k)  
X_digits_dist = kmeans.fit_transform(X_train)  
representative_digit_idx = np.argmin(X_digits_dist, axis=0)  
X_representative_digits = X_train[representative_digit_idx]
```

На рис. 9.13 показаны найденные 50 репрезентативных изображений.

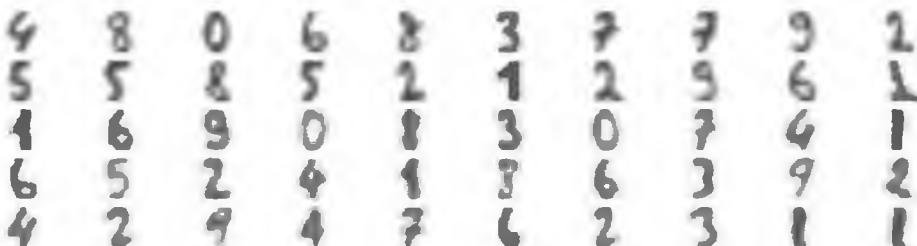


Рис. 9.13. Пятьдесят репрезентативных изображений цифр (по одному на кластер)

Посмотрим каждое изображение и вручную снабдим его меткой:

```
y_representative_digits =  
np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Теперь мы имеем набор данных со всего лишь 50 помеченных образцов, но вместо того, чтобы быть случайными образцами, каждое из них является репрезентативным изображением своего кластера. Посмотрим, улучшилась ли хоть как-то эффективность:

```
>>> log_reg = LogisticRegression()  
>>> log_reg.fit(X_representative_digits, y_representative_digits)  
>>> log_reg.score(X_test, y_test)  
0.9222222222222223
```

Вот здорово! Мы перескочили от правильности 83.3% до 92.2%, несмотря на то, что обучаем модель по-прежнему на 50 образцах. Поскольку часто снабжение образцов метками оказывается затратным и мучительным, осо-

бенно когда оно должно выполняться вручную экспертами, рекомендуется помечать не просто случайные, а репрезентативные образцы.

Но возможно мы сумеем продвинуть на шаг дальше: что если распространить метки на все остальные образцы в том же самом кластере? Прием называется *распространением меток* (*label propagation*):

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] = y_representative_digits[i]
```

Давайте обучим модель еще раз и выясним ее эффективность:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.9333333333333333
```

Мы получили приемлемое повышение правильности, но здесь нет абсолютно ничего удивительного. Проблема в том, что мы распространяли метку каждого репрезентативного образца на все образцы в том же кластере, включая те, которые расположены близко к границам кластера и весьма вероятно помечены неправильно. Посмотрим, что произойдет, если мы распространим метки на 20% образцов, являющихся ближайшими к центроидам:

```
percentile_closest = 20
X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1
partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Снова обучим модель на наборе данных с частичным распространением меток:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train_partially_propagated,
                 y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.94
```

В точку! С помощью всего лишь 50 помеченных образцов (в среднем только по 5 образцов на класс!) мы добились правильности 94.0%, которая довольно близка к эффективности логистической регрессионной модели, обученной на полностью помеченном наборе данных digits (она составляла 96.9%). Настолько хорошая эффективность обусловлена тем фактом, что распространенные метки в действительности довольно хороши — их правильность очень близка к 99%, как показывает следующий код:

```
>>> np.mean(y_train_partially_propagated ==  
y_train[partially_propagated])  
0.9896907216494846
```

## Активное обучение

В продолжение совершенствования модели и обучающего набора следующим шагом могло бы быть проведение нескольких раундов *активного обучения* (*active learning*), при котором человек-эксперт взаимодействует с алгоритмом обучения, предоставляя метки для индивидуальных образцов, когда алгоритм их запрашивает. Стратегий активного обучения существует много, но одна из наиболее распространенных называется *выборкой по наименьшей уверенности* (*uncertainty sampling*). Вот как она работает.

1. Модель обучается на помеченных образцах, собранных до сих пор, и эта модель применяется для вырабатывания прогнозов на всех непомеченных образцах.
2. Образцы, для которых модель оказывается наименее уверенной (т.е. когда оценка вероятности самая низкая), передаются эксперту для снабжения метками.
3. Процесс повторяется до тех пор, пока улучшение эффективности не перестанет стоить усилий по снабжению метками.

Другие стратегии включают пометку образцов, которые могли бы привести к наибольшему изменению модели или самому крупному падению ошибки проверки модели, либо образцов, которые не согласуются с разными моделями (например, на основе SVM или случайного леса).

Прежде чем переходить к исследованию моделей со смесями гауссовых распределений, давайте взглянем на DBSCAN — еще один популярный алгоритм кластеризации, который иллюстрирует совершенно другой подход, основанный на оценке локальной плотности. Такой подход позволяет алгоритму идентифицировать кластеры произвольных форм.

## DBSCAN

Алгоритм DBSCAN определяет кластеры как непрерывные области высокой плотности. Ниже описана его работа.

- Для каждого образца алгоритм подсчитывает количество образцов, расположенных в рамках небольшого расстояния  $\epsilon$  (эпсилон) от него. Такая область называется  $\epsilon$ -соседством экземпляра.
- Если образец имеет, по крайней мере, `min_samples` образцов в своем  $\epsilon$ -соседстве (включая себя), тогда он считается *центральным образцом* (*core instance*). Другими словами, центральные образцы — это такие образцы, которые расположены в плотных областях.
- Все образцы в соседстве центрального образца принадлежат тому же самому кластеру. Это соседство может включать другие центральные образцы; таким образом, длинная последовательность соседствующих центральных образцов формирует одиночный кластер.
- Любой образец, не являющийся центральным и не имеющий такового в своем соседстве, считается аномалией.

Алгоритм DBSCAN хорошо работает, если все кластеры достаточно плотные и хорошо разделены областями низкой плотности. Как и можно было ожидать, класс DBSCAN из Scikit-Learn использовать очень просто. Давайте протестируем его на наборе данных `moons`, представленном в главе 5:

```
from           import DBSCAN
from           import make_moons
X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

Теперь метки всех образцов доступны в переменной экземпляра `labels_`:

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])
```

Обратите внимание, что некоторые образцы имеют индекс кластера, равный -1; это означает, что алгоритм посчитал их аномалиями. Индексы центральных образцов доступны в переменной экземпляра `core_sample_indices`, а сами центральные образцы — в переменной экземпляра `components`:

```
>>> len(dbSCAN.core_sample_indices_)
808
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbSCAN.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```

Полученная кластеризация представлена слева на рис. 9.14. Как видите, она идентифицировала много аномалий и семь разных кластеров. Какое разочарование! К счастью, если мы расширим соседство, увеличив `eps` до 0.2, то получим кластеризацию, которая выглядит безупречно и показана справа на рис. 9.14. Давайте продолжим работу с данной моделью.

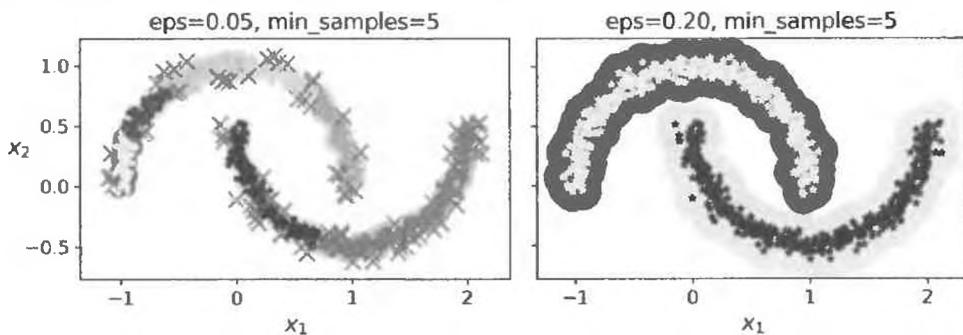


Рис. 9.14. Кластеризация DBSCAN, использующая два разных радиуса соседства

Немного удивляет то, что в классе DBSCAN отсутствует метод `predict()`, хотя имеется метод `fit_predict()`. Другими словами, он не способен прогнозировать, к какому кластеру принадлежит новый образец. Так поступили из-за того, что для решения разных задач могут быть более подходящими разные алгоритмы классификации, и пользователю предоставляется возможность выбора. Более того, реализовать это нетрудно. Например, давайте обучим экземпляр `KNeighborsClassifier`:

```
from import KNeighborsClassifier  
knn = KNeighborsClassifier(n_neighbors= )  
knn.fit(dbSCAN.components_,  
       dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

Теперь при наличии нескольких новых образцов мы можем вырабатывать прогнозы относительно кластеров, к которым они, по всей видимости, будут принадлежать, и даже оценивать вероятность для каждого кластера:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])  
>>> knn.predict(X_new)  
array([1, 0, 1, 0])  
>>> knn.predict_proba(X_new)  
array([[0.18, 0.82],  
      [1., 0.],  
      [0.12, 0.88],  
      [1., 0.]])
```

Обратите внимание, что мы обучали классификатор только на центральных образцах, но могли бы обучать его на всех образцах или на всех кроме аномалий: выбор зависит от финальной задачи.

Граница решений представлена на рис. 9.15 (крестиками обозначены четыре образца в  $X_{\text{new}}$ ).

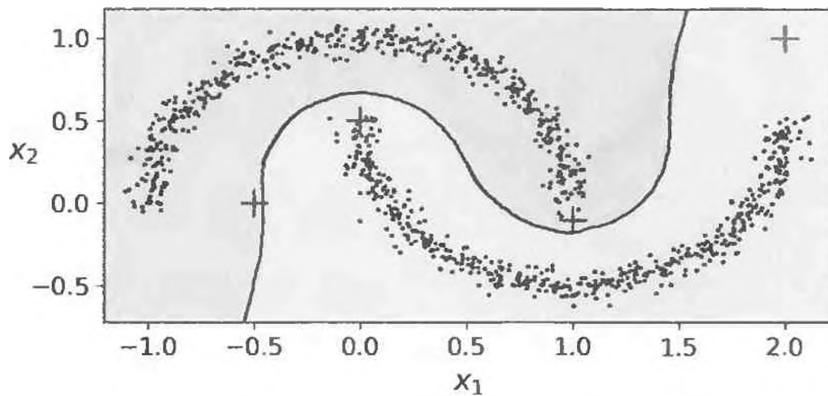


Рис. 9.15. Граница решений между двумя кластерами

Важно отметить, что поскольку в обучающем наборе отсутствуют аномалии, классификатор всегда выбирает какой-то кластер, даже когда данный кластер расположен далеко. Довольно просто ввести максимальное расстояние, и тогда два образца, которые находятся дальше него от обоих

кластеров, классифицируются как аномалии. Для этого применяется метод `kneighbors()` класса `KNeighborsClassifier`. Получив набор образцов, он возвращает расстояния и индексы  $k$  ближайших соседей в обучающем наборе (две матрицы, содержащие  $k$  столбцов):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

Говоря кратко, DBSCAN — очень простой, но мощный алгоритм, способный идентифицировать любое количество кластеров любой формы. Он устойчив к выбросам и имеет только два гиперпараметра (`eps` и `min_samples`). Однако если плотность значительно варьируется между кластерами, то надлежащий захват всех кластеров для него может оказаться невозможным. Вычислительная сложность DBSCAN составляет приблизительно  $O(m \log m)$ , что близко к линейной зависимости от количества образцов, но его реализация в Scikit-Learn может требовать вплоть до  $O(m^2)$  памяти при большом значении `eps`.



У вас может возникнуть желание опробовать также *иерархический алгоритм DBSCAN* (*Hierarchical DBSCAN* — *HDBSCAN*), который реализован в проекте `scikit-learn-contrib` (<https://github.com/scikit-learn-contrib/hdbscan/>).

## Другие алгоритмы кластеризации

В библиотеке Scikit-Learn реализовано несколько дополнительных алгоритмов кластеризации, с которыми вы должны ознакомиться. Мы не можем раскрыть их здесь во всех подробностях, но ниже приведен краткий обзор.

### Агломеративная кластеризация

Иерархия кластеров строится снизу вверх. Подумайте о множестве крошечных пузырьков, всплывающих на водную поверхность, которые постепенно склеиваются друг с другом до тех пор, пока не образуется одна большая группа пузырьков. Аналогичным образом *агломеративная кластеризация* (*agglomerative clustering*) на каждой итерации соединяет ближайшую пару кластеров (начиная с индиви-

дуальных образцов). Если нарисовать дерево с ветвями для каждой пары кластеров, которые объединяются, то получится двоичное дерево кластеров, где листья будут индивидуальными образцами. Такой подход очень хорошо масштабируется на большое число образцов или кластеров. Он способен захватывать кластеры различных размеров, выдавать гибкое и информативное дерево кластеров вместо того, чтобы заставлять вас выбирать конкретное соотношение между кластерами, и применяться с любым расстоянием для пар. Агломеративная кластеризация может хорошо масштабироваться на большое число образцов, если вы предоставите матрицу смежности, представляющую собой разреженную матрицу  $m \times m$ , которая указывает, какие пары образцов являются соседями (например, возвращаемую `sklearn.neighbors.kneighbors_graph()`). Без матрицы смежности алгоритм не будет хорошо масштабироваться на крупные наборы данных.

## BIRCH

Алгоритм *BIRCH* (*Balanced Iterative Reducing and Clustering using Hierarchies* — сбалансированное итеративное сокращение и кластеризация с использованием иерархий) был разработан специально для очень крупных наборов данных и может быть быстрее, чем пакетный K-Means, давая похожие результаты при условии, что количество признаков не слишком велико (<20). Во время обучения он строит древовидную структуру, содержащую только информацию, которой достаточно для быстрого назначения каждого нового образца кластеру, без необходимости в хранении всех образцов в дереве: такой подход позволяет алгоритму использовать ограниченную память наряду с обработкой гигантских наборов данных.

## Mean-Shift

Алгоритм *Mean-Shift* (*перемещение по среднему*) начинает с того, что размещает круги с центрами в каждом образце, после чего для каждого круга вычисляет среднее по всем образцам, находящимся внутри него, и смещает круг так, чтобы он был центрирован на этом среднем. Далее он повторяет такой шаг перемещения по среднему до тех пор, пока все круги не перестанут двигаться (т.е. пока каждый из них не окажется центрированным на среднем для образцов, которые он со-

держит). Алгоритм Mean-Shift перемещает круги в направлении более высокой плотности до тех пор, пока каждый из них не найдет локальный максимум плотности. В заключение все образцы, круги которых обосновались в одном и том же месте (или достаточно близко к нему), назначаются тому же самому кластеру. Алгоритм Mean-Shift обладает рядом таких же возможностей, как DBSCAN, вроде того, что он способен отыскать любое количество кластеров любой формы, имеет совсем немного гиперпараметров (всего лишь один — радиус кругов, называемый *шириной полосы*) и полагается на оценку локальной плотности. Но в отличие от DBSCAN алгоритм Mean-Shift склонен разбивать кластеры на части, когда в них наблюдаются внутренние вариации плотности. К сожалению, его вычислительная сложность составляет  $O(m^2)$  и потому он не подходит для крупных наборов данных.

### *Распространение похожести*

Этот алгоритм применяет систему голосования, где образцы голосуют за похожие образцы, чтобы стать для них репрезентативными, и как только алгоритм сходится, каждый репрезентативный образец и проголосовавшие за него образуют кластер. Алгоритм распространения похожести способен обнаруживать любое количество кластеров разных размеров. К сожалению, его вычислительная сложность составляет  $O(m^2)$  и потому он тоже не подходит для крупных наборов данных.

### *Спектральная кластеризация*

Алгоритм *спектральной кластеризации* (*spectral clustering*) берет матрицу подобия между образцами и создает из нее маломерное вложение (т.е. понижает ее размерность), после чего применяет еще один алгоритм кластеризации в полученном маломерном пространстве (реализация в Scikit-Learn использует K-Means). Спектральная кластеризация способна захватывать сложные кластерные структуры и также может применяться для сокращения графов (например, с целью идентификации друзей в какой-то социальной сети). Она не очень хорошо масштабируется на большие количества образцов и не особенно хорошо себя ведет, когда кластеры имеют крайне отличающиеся размеры.

А теперь давайте займемся моделями со смесями гауссовых распределений, которые можно использовать для оценки плотности, кластеризации и обнаружения аномалий.

# Смеси гауссовых распределений

Модель со смесью гауссовых распределений (*Gaussian mixture model* — GMM) — это вероятностная модель, которая предполагает, что образцы были созданы из смеси нескольких гауссовых распределений с неизвестными параметрами. Все образцы, сгенерированные из гауссова распределения, формируют кластер, который выглядит как эллипсоид. Каждый кластер может иметь другую эллиптическую форму, размер, плотность и ориентацию, как было показано на рис. 9.11. Заметив образец, вы знаете, что он был создан с помощью одного из гауссовых распределений, но неизвестно какого и с какими параметрами.

Существует несколько вариантов моделей GMM. В простейшем варианте, реализованном в классе GaussianMixture, вы должны знать заранее количество  $k$  гауссовых распределений. Предполагается, что набор данных  $X$  был сгенерирован посредством следующего вероятностного процесса.

- Для каждого образца кластер выбирается случайным образом среди  $k$  кластеров. Вероятность выбора  $j$ -того кластера определяется весом кластера  $\phi^{(j)}$ <sup>7</sup>. Индекс кластера, выбранного для  $i$ -того образца, обозначается как  $z^{(i)}$ .
- Если  $z^{(i)} = j$ , т.е.  $i$ -тый образец был назначен  $j$ -тому кластеру, то местоположение  $x^{(i)}$  данного образца выбирается случайным образом из гауссова распределения со средним  $\mu^{(j)}$  и ковариационной матрицей  $\Sigma^{(j)}$ . Это обозначается как  $x^{(i)} \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$ .

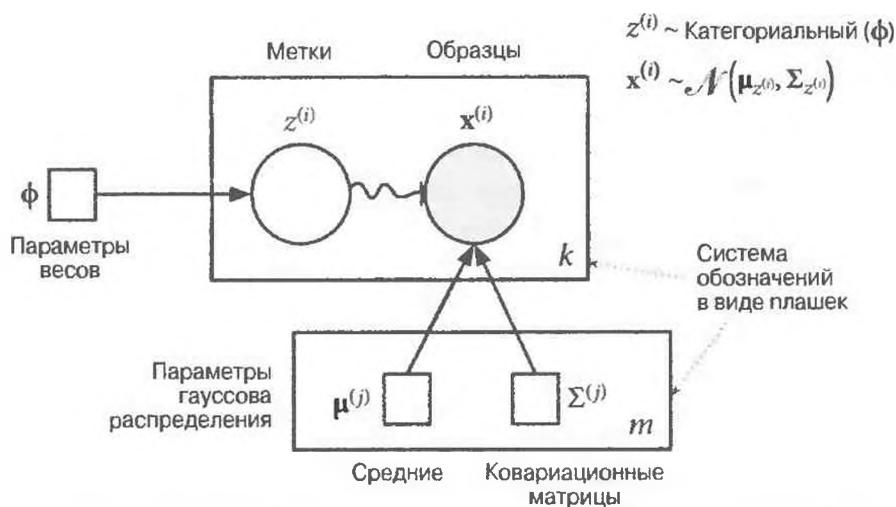
Такой процесс генерирования может быть представлен в виде графической модели. На рис. 9.16 показана структура условных зависимостей между случайными переменными.

Ниже приведена интерпретация рис. 9.16<sup>8</sup>.

- Круги представляют случайные переменные.
- Квадраты представляют фиксированные значения (т.е. параметры модели).

<sup>7</sup> Фи ( $\phi$  или  $\varphi$ ) — 21-я буква греческого алфавита.

<sup>8</sup> Большинство обозначений являются стандартными, но несколько дополнительных взяты из статьи в англоязычной Википедии по системе обозначений в виде плашек ([https://en.wikipedia.org/wiki/Plate\\_notation](https://en.wikipedia.org/wiki/Plate_notation)).



**Рис. 9.16. Графическое представление модели со смесью гауссовых распределений, включающее ее параметры (квадраты), случайные переменные (круги) и условные зависимости (сплошные линии со стрелками)**

- Большие прямоугольники называются *плашками* (*plate*). Они указывают, что их содержимое повторяется много раз.
- Число в нижнем правом углу каждой плашки указывает на то, сколько раз ее содержимое повторяется. Таким образом, есть  $m$  случайных переменных  $z^{(i)}$  (от  $z^{(1)}$  до  $z^{(m)}$ ) и  $m$  случайных переменных  $x^{(i)}$ . Имеется также  $k$  средних  $\mu^{(j)}$  и  $k$  ковариационных матриц  $\Sigma^{(j)}$ . Наконец, есть лишь один весовой вектор  $\phi$  (содержащий все веса от  $\phi^{(1)}$  до  $\phi^{(k)}$ ).
- Каждая переменная  $z^{(i)}$  берется из *категориального распределения* (*categorical distribution*) с весами  $\phi$ . Каждая переменная  $x^{(i)}$  берется из нормального распределения со средним и ковариационной матрицей, определенными ее кластером  $z^{(i)}$ .
- Сплошными линиями со стрелками обозначены условные зависимости. Например, распределение вероятностей для каждой случайной переменной  $z^{(i)}$  зависит от весового вектора  $\phi$ . Обратите внимание, что когда линия со стрелкой пересекает границу плашки, это значит, что она применяется ко всем повторениям данной плашки. Скажем, весовой вектор  $\phi$  обуславливает распределения вероятностей всех случайных переменных от  $x^{(1)}$  до  $x^{(m)}$ .

- Волнистая линия от  $z^{(i)}$  до  $x^{(i)}$  представляет переключатель: в зависимости от значения  $z^{(i)}$  образец  $x^{(i)}$  будет выбираться из различного гауссова распределения. Например, если  $z^{(i)} = j$ , тогда  $x^{(i)} \sim \mathcal{N}(\mu^{(j)}, \Sigma^{(j)})$ .
- Затушеванные узлы указывают на то, что значение известно. Таким образом, в данном случае известные значения имеет только случайная переменная  $x^{(i)}$ : они называются наблюдаемыми переменными (*observed variable*). Неизвестные переменные  $z^{(i)}$  называются латентными переменными (*latent variable*).

Так что же вы можете делать с такой моделью? Располагая набором данных  $X$ , вы обычно пожелаете начать с оценки весов  $\phi$  и всех параметров распределения от  $\mu^{(1)}$  до  $\mu^{(k)}$  и от  $\Sigma^{(1)}$  до  $\Sigma^{(k)}$ . Класс GaussianMixture из Scikit-Learn делает оценку чрезвычайно простой:

```
from import GaussianMixture
gm = GaussianMixture(n_components= , n_init= )
gm.fit(X)
```

Давайте взглянем на параметры, которые оценил алгоритм:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
       [-0.03270354,  0.95496237]],
       [[ 0.63478101,  0.72969804],
       [ 0.72969804,  1.1609872 ]],
       [[ 0.68809572,  0.79608475],
       [ 0.79608475,  1.21234145]]])
```

Великолепно, все работает хорошо! В действительности для генерирования данных использовались веса 0.2, 0.4 и 0.4; подобным же образом средние и ковариационные матрицы были очень близки к тем, что нашел алгоритм. Но как? Класс GaussianMixture опирается на алгоритм максимизации ожидания (*Expectation-Maximization (EM) algorithm*), который имеет много сходных черт с алгоритмом K-Means: он тоже инициализирует параметры кластеров случайнным образом и далее повторяет два шага, пока не сойдется,

сначала назначая образцы кластерам (*шаг ожидания*) и затем обновляя кластеры (*шаг максимизации*). Выглядит знакомым, не так ли? В контексте кластеризации вы можете думать об алгоритме EM как об обобщении K-Means, которое находит не только центры кластеров ( $\mu^{(1)} - \mu^{(k)}$ ), но также их размер, форму и ориентацию ( $\Sigma^{(1)} - \Sigma^{(k)}$ ) плюс их относительные веса ( $\phi^{(1)} - \phi^{(k)}$ ). Тем не менее, в отличие от K-Means алгоритм EM применяет *мягкую* кластеризацию, а не *жесткую*. Для каждого образца на шаге ожидания алгоритм оценивает вероятность того, что он принадлежит каждому кластеру (на основе текущих параметров кластеров). Затем на шаге максимизации каждый кластер обновляется с использованием *всех* образцов, причем каждый образец взвешивается по оценке вероятности того, что он принадлежит этому кластеру. Такие вероятности называются *ответственостями* (*responsibility*) кластеров за образцы.

Во время шага максимизации на обновление каждого кластера больше всего влияют образцы, за которые он несет наибольшую ответственность.



К сожалению, почти как K-Means, алгоритм EM может сходить в плохие решения, так что его приходится прогонять несколько раз, сохраняя только наилучшее решение. Именно потому мы установили `n_init` в 10. Будьте осторожны: по умолчанию `n_init` равно 1.

Вы можете проверить, сошелся ли алгоритм и сколько итераций ему потребовалось:

```
>>> gm.converged_
True
>>> gm.n_iter_
3
```

Теперь при наличии оценки местоположения, размера, ориентации и относительного веса каждого кластера модель способна легко назначить каждый образец наиболее вероятному кластеру (жесткая кластеризация) или оценить вероятность того, что он принадлежит конкретному кластеру (мягкая кластеризация). Просто применяйте метод `predict()` для жесткой кластеризации либо метод `predict_proba()` для мягкой кластеризации:

```
>>> gm.predict(X)
array([2, 2, 1, ..., 0, 0, 0])
>>> gm.predict_proba(X)
```

```
array([[2.32389467e-02, 6.77397850e-07, 9.76760376e-01],
       [1.64685609e-02, 6.75361303e-04, 9.82856078e-01],
       [2.01535333e-06, 9.99923053e-01, 7.49319577e-05],
       ...,
       [9.99999571e-01, 2.13946075e-26, 4.28788333e-07],
       [1.00000000e+00, 1.46454409e-41, 5.12459171e-16],
       [1.00000000e+00, 8.02006365e-41, 2.27626238e-15]])
```

Модель со смесью гауссовых распределений является *порождающей моделью (generative model)*, т.е. из нее можно выбирать новые образцы ( обратите внимание, что они отсортированы по индексу кластера):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([[ 2.95400315,  2.63680992],
       [-1.16654575,  1.62792705],
       [-1.39477712, -1.48511338],
       [ 0.27221525,  0.690366 ],
       [ 0.54095936,  0.48591934],
       [ 0.38064009, -0.56240465]])
```

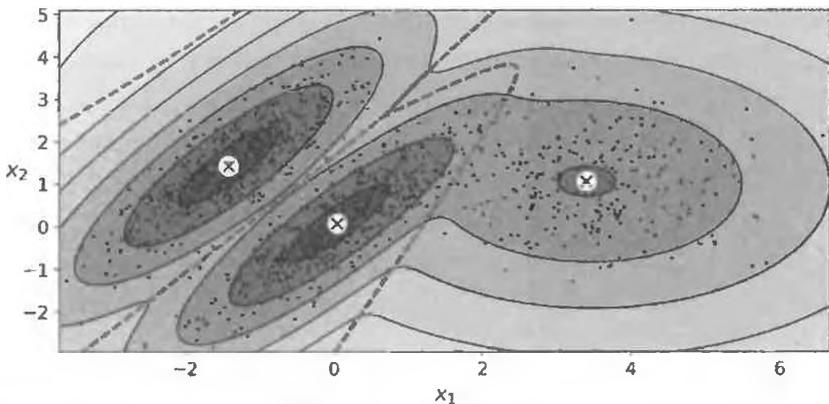
```
>>> y_new
array([0, 1, 2, 2, 2, 2])
```

Также есть возможность оценить плотность модели в любом заданном местоположении. Задача решается с использованием метода `score_samples()`: для каждого передаваемого образца метод оценивает логарифм функции плотности вероятности (*probability density function — PDF*) в данном местоположении. Чем больше оценка, тем выше плотность:

```
>>> gm.score_samples(X)
array([-2.60782346, -3.57106041, -3.33003479, ..., -3.51352783,
       -4.39802535, -3.80743859])
```

Вычислив экспоненту таких оценок, вы получите значение функции PDF в местоположении заданных образцов. Значения являются не вероятностями, но плотностями вероятности: они могут быть любыми положительными значениями, а не только значениями между 0 и 1. Чтобы оценить вероятность того, что образец попадет внутрь конкретной области, понадобилось бы интегрировать функцию PDF по этой области (если вы поступите так по всему пространству местоположений образцов, то результатом будет 1).

На рис. 9.17 показаны средние кластеров, границы решений (пунктирные линии) и контуры плотностей данной модели.



**Рис. 9.17.** Средние кластеров, границы решений и контуры плотностей обученной модели со смесью гауссовых распределений

Отлично! Алгоритм явно нашел превосходное решение. Конечно, мы облегчили задачу за счет генерирования данных с применением множества двумерных гауссовых распределений (к сожалению, реальные данные не всегда оказываются настолько гауссовыми и двумерными). Мы также предоставили алгоритму корректное количество кластеров. Когда есть много измерений, много кластеров или мало образцов, алгоритм EM может стараться изо всех сил сойтись в оптимальном решении. У вас может возникнуть необходимость понизить сложность задачи, ограничивая количество параметров, которые алгоритм должен выяснить. Один из способов сделать это предусматривает ограничение диапазона форм и ориентаций, которые кластера могут иметь. Цель достигается накладыванием ограничений на ковариационные матрицы, для чего гиперпараметр `covariance_type` устанавливается в одно из следующих значений.

#### **"spherical"**

Все кластеры обязаны быть сферическими, но они могут иметь разные диаметры (т.е. разные величины дисперсии).

#### **"diag"**

Кластеры могут принимать любую эллиптическую форму любого размера, но оси эллипсоида должны быть параллельны координатным осям (т.е. ковариационные матрицы обязаны быть диагональными).

#### **"tied"**

Все кластеры должны иметь ту же самую эллиптическую форму, размер и ориентацию (т.е. все кластеры разделяют одну и ту же ковариационную матрицу).

По умолчанию `covariance_type` равен "full", а это значит, что каждый кластер может принимать любую форму, размер и ориентацию (он имеет собственную не связанные ограничениями ковариационную матрицу). На рис. 9.18 изображены решения, найденные алгоритмом EM, когда `covariance_type` устанавливается в "tied" или "spherical".

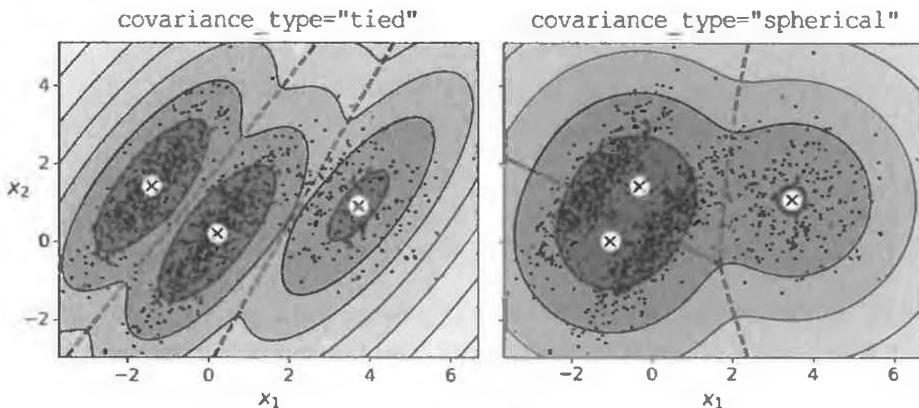


Рис. 9.18. Смеси гауссовых распределений для связанных (слева) и сферических (справа) кластеров



Вычислительная сложность обучения модели GaussianMixture зависит от количества образцов  $m$ , числа измерений  $n$ , количества кластеров  $k$  и ограничений на ковариационных матрицах. Если гиперпараметр `covariance_type` установлен в "spherical" или "diag", то сложность составляет  $O(kmn)$  в предположении, что данные имеют кластерную структуру. Если `covariance_type` установлен в "tied" или "full", то сложность выглядит как  $O(kmn^2 + kn^3)$ , и алгоритм не будет масштабироваться на крупные количества признаков.

Модели со смесями гауссовых распределений могут также использоваться для обнаружения аномалий. Давайте посмотрим, каким образом.

## Обнаружение аномалий с использованием смесей гауссовых распределений

Обнаружение аномалий (также называемое обнаружением выбросов) представляет собой задачу выявления образцов, которые сильно отклоняются от нормы. Такие образцы называются *аномалиями* или *выбросами*, тогда как нормальные образцы — *не-выбросами (inlier)*. Обнаружение аномалий по-

лезно в широком спектре приложений, таких как выявление мошенничества, обнаружение дефектных изделий в производстве или удаление выбросов из набора данных перед обучением другой модели (что может значительно повысить эффективность результирующей модели).

Применять модель со смесью гауссовых распределений для обнаружения аномалий довольно просто: любой образец, расположенный в области низкой плотности, может считаться аномалией. Вы должны определить желаемый порог плотности. Например, в производственной компании, где пытаются выявлять дефектные изделия, пропорция дефектных изделий обычно хорошо известна. Скажем, она составляет 4%. Затем вы устанавливаете порог плотности в значение, которое в результате обеспечит наличие 4% изделий в областях ниже заданного порога плотности. Если вы заметили, что получаете чересчур много ложноположительных классификаций (т.е. вполне качественных изделий, которые помечены как дефектные), тогда можете уменьшить порог. И наоборот, если появляется слишком много ложноотрицательных классификаций (т.е. дефектных изделий, которые система не пометила как дефектные), то можете увеличить порог. Это обычное соотношение точность/полнота (см. главу 3). Вот как вы могли бы идентифицировать выбросы, используя в качестве порога четвертый процентиль наименьшей плотности (т.е. приблизительно 4% образцов будут помечаться как аномалии):

```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 4)
anomalies = X[densities < density_threshold]
```

На рис. 9.19 аномалии представлены в виде звездочек.

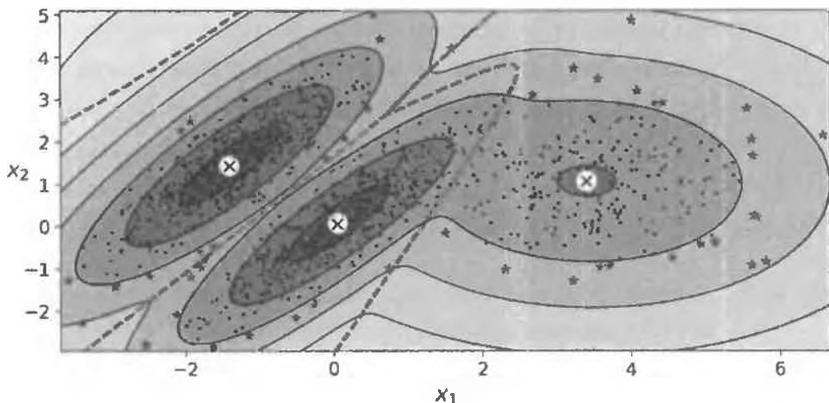


Рис. 9.19. Обнаружение аномалий с применением модели со смесью гауссовых распределений

Тесно связанной задачей является *обнаружение новизны* (*novelty detection*); оно отличается от обнаружения аномалий тем, что алгоритм предположительно обучается на “чистом” наборе данных, незагрязненным выбросами, тогда как при обнаружении аномалий такое предположение не выдвигается. На самом деле обнаружение выбросов часто используется для очистки набора данных.



Модели со смесями гауссовых распределений пытаются выполнить подгонку ко всем данным, включая выбросы, поэтому если их чрезмерно много, то представление “нормальности” модели будет смещено, и некоторые выбросы могут неправильно трактоваться как нормальные данные. Если подобное происходит, тогда можете попробовать подогнать модель один раз, применить ее для обнаружения и удаления наиболее экстремальных выбросов, а затем снова подогнать модель к очищенному набору данных. Другой подход предусматривает использование устойчивых методов ковариационной оценки (см. класс `EllipticEnvelope`).

Как и K-Means, алгоритм GaussianMixture требует указания количества кластеров. А как его можно найти?

## Выбор количества кластеров

В алгоритме K-Means для выбора подходящего количества кластеров вы могли бы применять инерцию или оценку силузтов. Но в случае смесей гауссовых распределений использовать указанные показатели невозможно, потому что они не являются надежными, когда кластеры не сферические либо имеют разные размеры. Взамен вы можете попробовать найти модель, которая минимизирует *теоретический информационный критерий* (*theoretical information criterion*), такой как *байесовский информационный критерий* (*Bayesian information criterion — BIC*) или *информационный критерий Акаике* (*Akaike information criterion — AIC*), определенный в уравнении 9.1.

### Уравнение 9.1. Байесовский информационный критерий (BIC) и информационный критерий Акаике (AIC)

$$BIC = \log(m)p - 2 \log(\hat{L})$$

$$AIC = 2p - 2 \log(\hat{L})$$

В приведенных уравнениях:

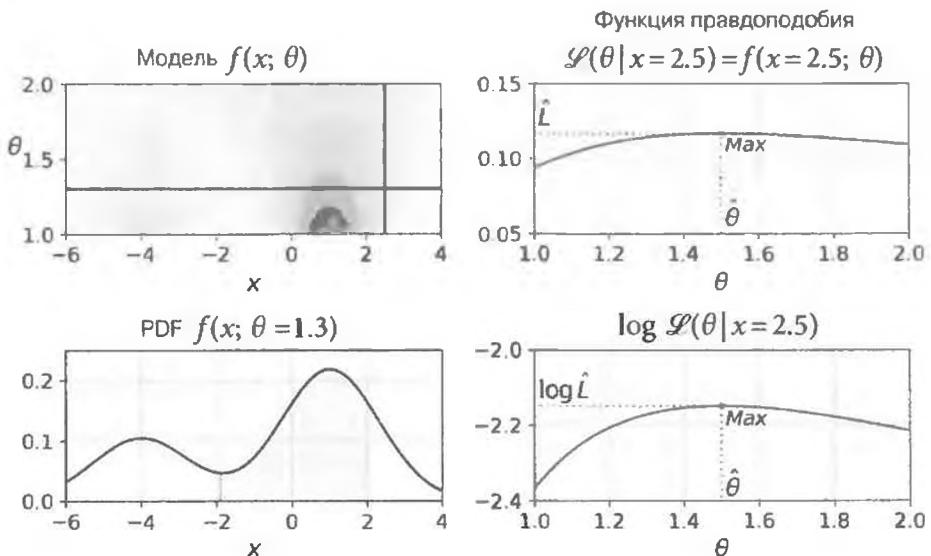
- $m$  — как всегда, количество образцов;
- $p$  — количество параметров, выясненных моделью;
- $\hat{L}$  — максимизированное значение функция правдоподобия (*likelihood function*) модели.

Критерии BIC и AIC штрафуют модели, которые имеют больше параметров, подлежащих выяснению (например, больше кластеров), и награждают модели, хорошо подгоняемые к данным. Они часто в итоге приводят к выбору той же самой модели. Когда модели отличаются, выбранная критерием BIC модель имеет тенденцию быть проще (меньше параметров), чем модель, выбранная критерием AIC, но она обычно не очень хорошо подгоняется к данным (особенно в случае более крупных наборов данных).

### Функция правдоподобия

Термины “вероятность” и “правдоподобие” в естественном языке часто применяются взаимозаменяющими, но в статистике они имеют очень разный смысл. Для заданной статистической модели с рядом параметров  $\theta$  слово “вероятность” используется для описания, насколько внушающим доверие является исход  $x$  (зная значения параметров  $\theta$ ), в то время как слово “правдоподобие” применяется для описания, насколько внушающим доверие был отдельный набор значений параметров  $\theta$ , после того, как известен исход  $x$ .

Рассмотрим одномерную модель со смесью двух гауссовых распределений, центрированных в  $-4$  и  $+1$ . Ради простоты эта игрушечная модель имеет единственный параметр  $\theta$ , который управляет стандартным отклонением обоих распределений. Контурный график слева вверху на рис. 9.20 изображает полную модель  $f(x; \theta)$  как функцию от  $x$  и  $\theta$ . Чтобы оценить распределение вероятностей будущего исхода  $x$ , понадобится установить параметр модели  $\theta$ . Например, если вы установите  $\theta$  в  $1.3$  (горизонтальная линия), то получите функцию плотности вероятности  $f(x; \theta = 1.3)$ , показанную на графике ниже слева. Допустим, вы хотите оценить вероятность того, что  $x$  окажется между  $-2$  и  $+2$ . Вы должны рассчитать интеграл функции PDF на указанном диапазоне (т.е. поверхность затушеванной области). Но что, если вы не знаете  $\theta$ , а замен наблюдаете одиночный образец  $x = 2.5$  (вертикальная линия на графике слева вверху)? В таком случае вы получаете функцию правдоподобия  $\mathcal{L}(\theta | x = 2.5) = f(x = 2.5; \theta)$ , представленную на графике справа вверху.



**Рис. 9.20.** Параметрическая функция модели (слева вверху) и несколько производных функций: PDF (слева внизу), функция правдоподобия (справа вверху) и функция логарифмического правдоподобия (справа внизу)

Выражаясь кратко, PDF — это функция  $x$  (при фиксированном значении  $\theta$ ), тогда как функция правдоподобия — функция  $\theta$  (при фиксированном значении  $x$ ). Важно понимать, что функция правдоподобия не является распределением вероятностей: если вы интегрируете распределение вероятностей по всем возможным значениям  $x$ , то всегда получите 1; но интегрирование функции правдоподобия по всем возможным значениям  $\theta$  может дать в результате любое положительное значение.

Распространенная задача для имеющегося набора данных  $X$  заключается в попытке оценить наиболее вероятные значения параметров модели. Вы должны отыскать значения, которые доводят до максимума функцию правдоподобия для заданного  $X$ . В рассматриваемом примере, если вы наблюдаете одиничный образец  $x = 2.5$ , тогда оценкой максимального правдоподобия (maximum likelihood estimate — MLE) для  $\theta$  будет  $\hat{\theta} = 1.5$ . Если существует предварительное распределение вероятностей  $g$  по  $\theta$ , то его можно учесть, доводя до максимума  $\mathcal{L}(\theta|x)g(\theta)$ , а не просто  $\mathcal{L}(\theta|x)$ . Это называется оценкой апостериорного максимума (maximum a posteriori — MAP). Поскольку оценка MAP ограничивает значения параметров, вы можете считать ее регуляризированной версией оценки MLE.

Обратите внимание, что доведение до максимума функции правдоподобия эквивалентно максимизации ее логарифма (см. график справа внизу на рис. 9.20). На самом деле логарифм является строго возрастающей функцией, так что если  $\theta$  доводит до максимума логарифмическое правдоподобие, то максимизирует и само правдоподобие. Оказывается, что обычно легче доводить до максимума логарифмическое правдоподобие. Например, если вы наблюдаете несколько независимых образцов от  $x^{(1)}$  до  $x^{(m)}$ , то вам потребуется найти такое значение  $\theta$ , которое доводит до максимума произведение индивидуальных функций правдоподобия. Но эквивалентным действием, к тому же гораздо более простым, будет доведение до максимума суммы (не произведения) функций логарифмического правдоподобия благодаря магии логарифма, который преобразует произведения в суммы:  $\log(ab) = \log(a) + \log(b)$ .

После того, как вы получили оценку  $\hat{\theta}$ , т.е. значение  $\theta$ , доводящее до максимума функцию правдоподобия, вы готовы вычислить  $\hat{L} = \mathcal{L}(\hat{\theta}, X)$ , которое представляет собой значение, используемое для расчета критериев AIC и BIC; можете считать это мерой того, насколько хорошо модель подгоняется к данным.

Для вычисления BIC и AIC вызовите методы `bic()` и `aic()`:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

На рис. 9.21 показаны критерии BIC и AIC для разного количества кластеров  $k$ . Как видите, оба критерия оказываются самыми низкими при  $k=3$ , а потому вполне вероятно такое количество кластеров будет наилучшим выбором. Обратите внимание, что мы также могли бы поискать лучшее значение для гиперпараметра `covariance_type`. Скажем, если он установлен в "spherical" вместо "full", тогда модель имеет гораздо меньше параметров, подлежащих выяснению, но не настолько хорошо подгоняется к данным.

## Байесовские модели со смесями гауссовых распределений

Вместо того чтобы вручную искать оптимальное количество кластеров, вы можете применять класс `BayesianGaussianMixture`, который способен назначать веса, равные (или близкие к) нулю, излишним кластерам.

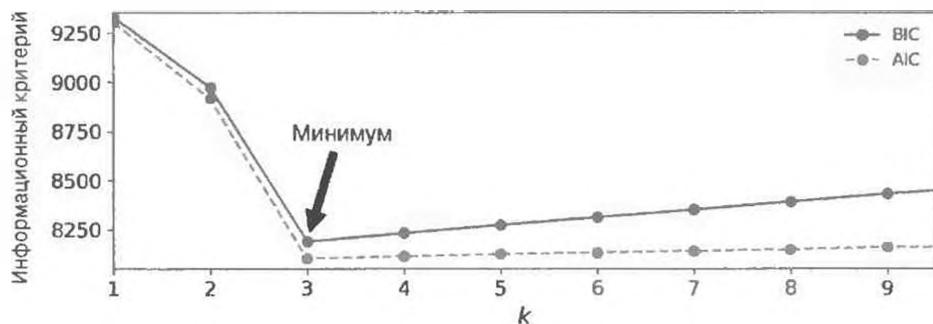


Рис. 9.21. Критерии AIC и BIC для разного количества кластеров  $k$

Установите количество кластеров `n_components` равным значению, в отношении которого у вас есть веские основания полагать, что оно превышает оптимальное число кластеров (это допускает минимальные знания о решаемой задаче), и алгоритм автоматически удалит ненужные кластеры. Например, давайте установим количество кластеров в 10 и посмотрим, что произойдет:

```
>>> from import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 1)
array([0.4, 0.21, 0.4, 0., 0., 0., 0., 0., 0., 0.])
```

Великолепно: алгоритм автоматически выявил, что необходимо лишь три кластера, и результирующие кластеры почти идентичны кластерам, приведенным на рис. 9.17.

В этой модели параметры кластеров (включая веса, средние и ковариационные матрицы) больше не трактуются как фиксированные параметры модели, а рассматриваются как латентные случайные переменные, подобные назначениям кластеров (рис. 9.22). Таким образом, `z` теперь содержит в себе и параметры кластеров, и назначения кластеров.

*Бета-распределение (Beta distribution)* обычно используется для моделирования случайных переменных, чьи значения находятся внутри фиксированного диапазона. В нашем случае диапазон — от 0 до 1. *Процесс ломки палки (Stick-Breaking Process — SBP)* лучше объяснять посредством примера: допустим, что  $\Phi = [0.3, 0.6, 0.5, \dots]$ , тогда 30% образцов будут назначены кластеру 0, после чего 60% оставшихся образцов будут назначены кластеру 1, затем 50% оставшихся образцов будут назначены кластеру 2 и т.д.

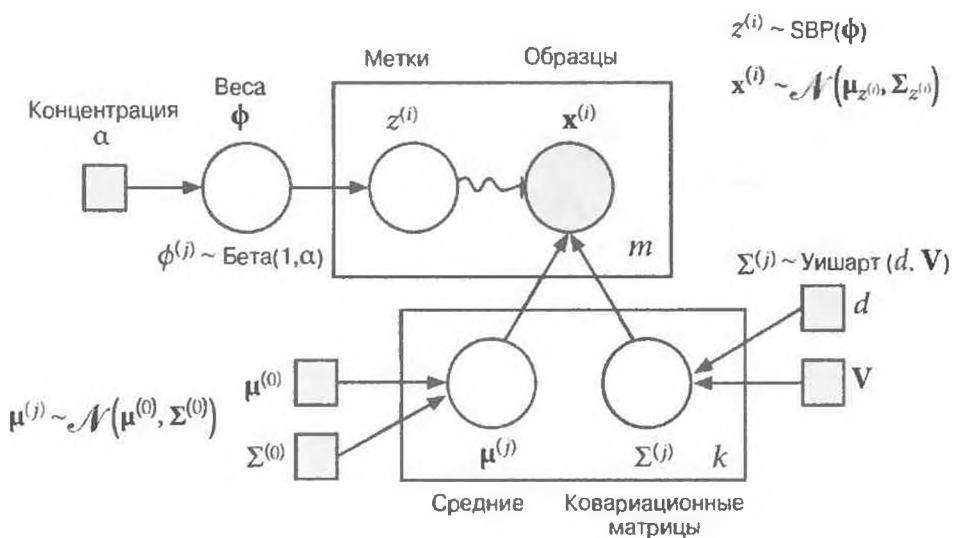
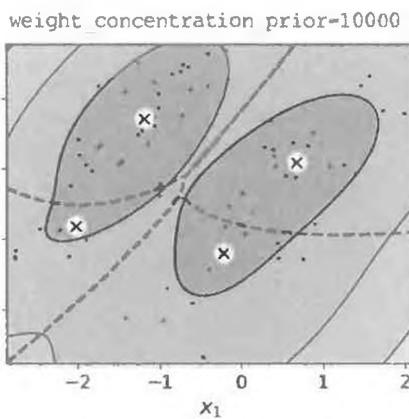
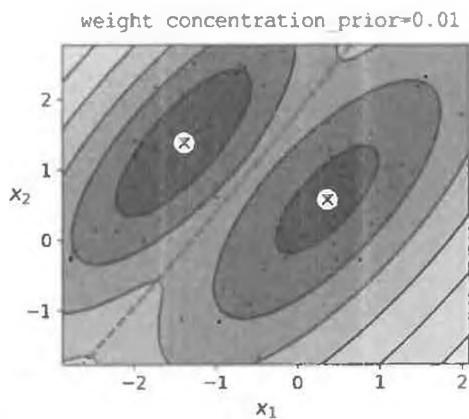


Рис. 9.22. Байесовская модель со смесью гауссовых распределений

Такой процесс является хорошей моделью для наборов данных, где новые образцы с более высокой вероятностью присоединяются к крупным кластерам, а не к мелким (например, люди с наибольшей вероятностью переезжают в более крупные города). Если концентрация  $\alpha$  высока, тогда значения  $\Phi$ , вероятно, будут близкими к 0, и процесс SBP генерирует много кластеров. Наоборот, при низкой концентрации значения  $\Phi$ , вероятно, будут близкими к 1, и кластеров окажется мало. Наконец, распределение Уишарта применяется для выборки из ковариационных матриц: параметры  $d$  и  $V$  управляют распределением форм кластеров.

Предварительное знание о латентных переменных  $z$  может быть закодировано в распределении вероятностей  $p(z)$ , которое называется *априорным (prior)*. Скажем, мы можем иметь априорное убеждение в том, что кластеров, вероятно, будет мало (низкая концентрация) или, наоборот, в изобилии (высокая концентрация). Такое априорное убеждение относительно количества кластеров может регулироваться с использованием гиперпараметра `weight_concentration_prior`. Его установка в 0.01 или 10 000 дает очень разную кластеризацию (рис. 9.23). Но чем больше данных у нас есть, тем меньшую роль играют априорные убеждения. На самом деле для вычерчивания диаграмм с такими большими различиями вы должны применять очень сильные априорные убеждения и немного данных.



**Рис. 9.23.** Использование разных предпосылок относительно концентрации на тех же самых данных в результате дает разное количество кластеров

Теорема Байеса (уравнение 9.2) сообщает нам о том, как обновить распределение вероятностей по латентным переменным после наблюдения некоторых данных  $\mathbf{X}$ . Она рассчитывает апостериорное распределение  $p(\mathbf{z}|\mathbf{X})$ , представляющее собой условную вероятность  $\mathbf{z}$  при условии  $\mathbf{X}$ .

### Уравнение 9.2. Теорема Байеса

$$p(\mathbf{z}|\mathbf{X}) = \frac{\text{апостериорное}}{\text{распределение}} = \frac{\text{правдоподобие} \times \text{априорное убеждение}}{\text{доказательство}} = \frac{p(\mathbf{X}|\mathbf{z}) p(\mathbf{z})}{p(\mathbf{X})}$$

К сожалению, в модели со смесью гауссовых распределений (и многих других задачах) знаменатель  $p(\mathbf{X})$  является трудным для решения, т.к. требует интегрирования по всем возможным значениям  $\mathbf{z}$  (уравнение 9.3), что нуждается в учете всех возможных комбинаций параметров кластеров и назначений кластеров.

### Уравнение 9.3. Доказательство $p(\mathbf{X})$ часто оказывается трудным для решения

$$p(\mathbf{X}) = \int p(\mathbf{X}|\mathbf{z}) p(\mathbf{z}) d\mathbf{z}$$

Трудность для решения — одна из центральных проблем в байесовской статистике, и существует несколько подходов к ее решению. Среди них *вариационный вывод* (*variational inference*), который выбирает семейство распределений  $q(\mathbf{z}; \lambda)$  с собственными вариационными параметрами  $\lambda$  (лямбда) и затем оптимизирует эти параметры, чтобы сделать  $q(\mathbf{z})$  хорошим приближением  $p(\mathbf{z}|\mathbf{X})$ . Цель достигается нахождением значения  $\lambda$ , которое минимизирует *расстояние* (расхождение) Кульбака–Лейблера (Kullback–

*Leibler (KL) divergence*) от  $q(\mathbf{z})$  до  $p(\mathbf{z} | \mathbf{X})$ , обозначаемое  $D_{KL}(q || p)$ . Расстояние Кульбака–Лейблера вычисляется согласно уравнению 9.4, которое можно переписать как логарифм доказательства ( $\log p(\mathbf{X})$ ) минус *нижняя граница доказательства* (*evidence lower bound — ELBO*). Поскольку логарифм доказательства не зависит от  $q$ , он является постоянным членом, так что минимизация расстояния Кульбака–Лейблера требует лишь максимизации ELBO.

#### Уравнение 9.4. Расстояние Кульбака–Лейблера от $q(\mathbf{z})$ до $p(\mathbf{z} | \mathbf{X})$

$$\begin{aligned} D_{KL}(q || p) &= \mathbb{E}_q \left[ \log \frac{q(\mathbf{z})}{p(\mathbf{z} | \mathbf{X})} \right] = \\ &= \mathbb{E}_q [ \log q(\mathbf{z}) - \log p(\mathbf{z} | \mathbf{X}) ] = \\ &= \mathbb{E}_q \left[ \log q(\mathbf{z}) - \log \frac{p(\mathbf{z}, \mathbf{X})}{p(\mathbf{X})} \right] = \\ &= \mathbb{E}_q [ \log q(\mathbf{z}) - \log p(\mathbf{z}, \mathbf{X}) + \log p(\mathbf{X}) ] = \\ &= \mathbb{E}_q [ \log q(\mathbf{z}) ] - \mathbb{E}_q [ \log p(\mathbf{z}, \mathbf{X}) ] + \mathbb{E}_q [ \log p(\mathbf{X}) ] = \\ &= \mathbb{E}_q [ \log p(\mathbf{X}) ] - (\mathbb{E}_q [ \log p(\mathbf{z}, \mathbf{X}) ] - \mathbb{E}_q [ \log q(\mathbf{z}) ]) = \\ &= \log p(\mathbf{X}) - \text{ELBO}, \end{aligned}$$

$$\text{где } \text{ELBO} = \mathbb{E}_q [ \log p(\mathbf{z}, \mathbf{X}) ] - \mathbb{E}_q [ \log q(\mathbf{z}) ]$$

На практике доступны различные методики для минимизации ELBO. При *вариационном выводе среднего поля* (*mean field variational inference*) семейство распределений  $q(\mathbf{z}; \boldsymbol{\lambda})$  и априорное убеждение  $p(\mathbf{z})$  необходимо выбирать очень тщательно, чтобы гарантировать упрощение уравнения для ELBO в форму, которую можно вычислить. К сожалению, не существует общего способа сделать это. Выбор правильного семейства распределений и правильного априорного убеждения зависит от задачи и требует наличия математических навыков. Скажем, распределения и уравнения нижней границы, применяемые в классе `BayesianGaussianMixture` из `Scikit-Learn`, приведены в документации (<https://homl.info/40>). Из этих уравнений можно вывести уравнения обновления для параметров кластеров и переменных назначения: затем они используются почти так, как в алгоритме максимизации ожидания. Фактически вычислительная сложность класса `BayesianGaussianMixture` подобна вычислительной сложности класса `GaussianMixture` (но обычно значительно ниже). Более простой подход максимизации ELBO называется *стохастическим вариационным выводом по*

принципу “черного ящика” (*black box stochastic variational inference — BBSVI*): на каждой итерации из  $q$  берется несколько образцов и они применяются для оценки градиентов ELBO относительно вариационных параметров  $\lambda$ , которые затем используются на шаге градиентного подъема. Такой подход делает возможным применение байесовского вывода (*Bayesian inference*) с моделью любого вида (при условии ее дифференцируемости), даже глубоких нейронных сетей; использование байесовского вывода с глубокими нейронными системами называется байесовским глубоким обучением (*Bayesian Deep Learning*).



Если вы желаете глубже изучить байесовскую статистику, тогда обратитесь к книге *Bayesian Data Analysis* (Байесовский анализ данных) (<https://home.info/bda>), написанной Эндрю Гелманом и др. (Chapman & Hall).

Модели со смесями гауссовых распределений прекрасно работают на кластерах с эллиптическими формами, но если вы попытаетесь выполнить подгонку к набору данных с другими формами, то можете столкнуться с не приятными сюрпризами. Например, давайте посмотрим, что произойдет в случае применения модели со смесью гауссовых распределений для кластеризации набора данных *moons* (рис. 9.24).

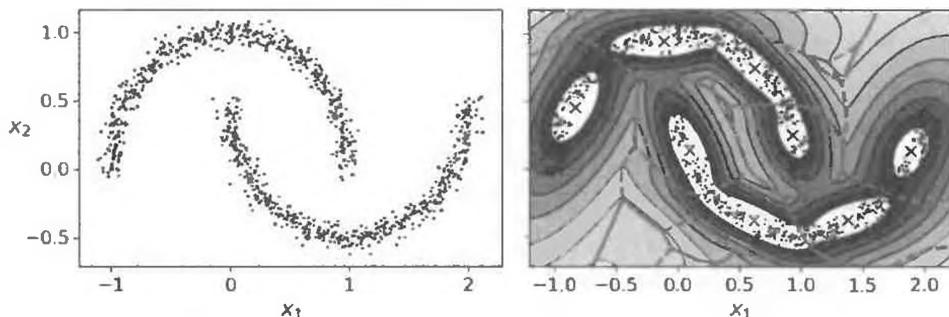


Рис. 9.24. Подгонка модели со смесью гауссовых распределений к кластерам неэллиптической формы

Ох! Алгоритм отчаянно искал эллипсоиды, поэтому вместо двух он нашел восемь разных кластеров. Оценка плотности не слишком плоха, так что модель, вероятно, могла бы использоваться для обнаружения аномалий, но она не сумела идентифицировать два кластера. Давайте теперь взглянем на несколько алгоритмов кластеризации, способных иметь дело с кластерами произвольной формы.

## Другие алгоритмы для обнаружения аномалий и новизны

В библиотеке Scikit-Learn реализованы другие алгоритмы, предназначенные для обнаружения аномалий и новизны.

### *PCA (и другие методики понижения размерности через метод `inverse_transform()`)*

Если сравнить ошибку восстановления нормального образца и ошибку восстановления аномалии, то последняя, как правило, оказывается намного больше. Это простой и часто довольно эффективный подход обнаружения аномалий (его применение отражено в упражнениях в конце главы).

### *Fast-MCD (minimizes covariance determinant — минимальный определитель ковариационных матриц)*

Реализованный классом EllipticEnvelope алгоритм Fast-MCD пригоден для обнаружения выбросов, в частности для очистки набора данных. Он предполагает, что нормальные образцы (не-выбросы) генерируются из единственного гауссова распределения (не смеси). Алгоритм также допускает, что набор данных загрязнен выбросами, которые не были сгенерированы из этого гауссова распределения. Когда алгоритм оценивает параметры гауссова распределения (т.е. форму эллиптической огибающей вокруг не-выбросов), он старательно игнорирует образцы, которые наиболее вероятно являются выбросами. Такая методика обеспечивает наилучшую оценку эллиптической огибающей и тем самым делает алгоритм более эффективным в плане обнаружения выбросов.

### *Isolation Forest (изолирующий лес)*

Эффективный алгоритм для обнаружения выбросов, особенно в многомерных наборах данных. Алгоритм строит случайный лес, в котором каждое дерево принятия решений выращено случайным образом: в каждом узле он наугад выбирает признак и затем случайное пороговое значение (между минимальным и максимальным значениями), чтобы расщепить набор данных на два. Набор данных разделяется на части подобным образом до тех пор, пока все образцы не окажутся изолированными друг от друга. Аномалии обычно находятся далеко от остальных образцов, поэтому в среднем (по всем деревьям принятия решений) они, как правило, изолируются за меньшее число шагов по сравнению с нормальными образцами.

## **LOF (Local Outlier Factor — локальный уровень выброса)**

Алгоритм LOF также хорош для обнаружения выбросов. Он сравнивает плотность образцов вокруг заданного образца с плотностью вокруг его соседей. Аномалия часто оказывается более изолированной, чем его  $k$  ближайших соседей.

## **One-class SVM (одноклассовый SVM)**

Данный алгоритм лучше подходит для обнаружения новизны. Вспомните, что классификатор на основе ядерного метода опорных векторов (*kernelized SVM*) разделяет два класса, сначала (неявно) отображая все образцы на многомерное пространство и затем отделяя классы с использованием классификатора на основе линейного SVM внутри этого многомерного пространства (см. главу 5). Поскольку у нас есть лишь один класс образцов, алгоритм на базе одноклассового SVM пытается разделить образцы в многомерном пространстве из исходного. В исходном пространстве это будет соответствовать нахождению небольшой области, которая охватывает все образцы. Если новый образец не попадает внутрь такой области, тогда он является аномалией. Существует несколько гиперпараметров, подлежащих настройке: обычно гиперпараметры для ядерного SVM плюс гиперпараметр зазора, который соответствует вероятности, что новый образец ошибочно будет расценен как новизна, в то время как он фактически нормален. Алгоритм работает прекрасно, особенно с многомерными наборами данных, но подобно всем методам SVM он не масштабируется на крупные наборы данных.

## **Упражнения**

1. Как вы определили кластеризацию? Можете ли вы назвать несколько алгоритмов кластеризации?
2. Назовите ряд главных приложений алгоритмов кластеризации.
3. Опишите две методики выбора правильного количества кластеров при использовании K-Means.
4. Что такое распространение меток? Зачем вам его реализовывать и как?

5. Можете ли вы назвать два алгоритма кластеризации, которые способны масштабироваться на крупные наборы данных? А еще два, которые ищут области высокой плотности?
6. Можете ли вы вспомнить сценарий использования, когда активное обучение было бы полезным? Как бы вы его реализовали?
7. В чем отличие между обнаружением аномалий и обнаружением новизны?
8. Что такое смесь гауссовых распределений? Для каких задач вы можете ее применять?
9. Можете ли вы назвать две методики для нахождения правильного количества кластеров при использовании модели со смесью гауссовых распределений?
10. Классический набор данных Olivetti faces (база изображений лиц, подготовленная в научно-исследовательской лаборатории компании Olivetti) содержит 400 полутоновых изображений лиц размером  $64 \times 64$  пикселя. Каждое изображение разглажено в одномерный вектор размером 4 096. Было сфотографировано 40 человек (каждый по 10 раз), и обычная задача заключается в обучении модели, которая может вырабатывать прогнозы о том, какой человек представлен на каждой фотографии. Загрузите набор данных с применением функции `sklearn.datasets.fetch_olivetti_faces()`, после чего расщепите его на обучающий набор, проверочный набор и испытательный набор (имейте в виду, что набор данных уже отмасштабирован между 0 и 1). Поскольку набор данных довольно мал, возможно, вы решите использовать стратифицированную выборку, чтобы гарантировать наличие в каждом наборе того же самого количества изображений для каждого человека. Затем кластеризуйте изображения с применением K-Means и удостоверьтесь, что имеете подходящее количество кластеров (используя одну из методик, которые обсуждались в этой главе). Визуализируйте кластеры: видите ли вы похожие лица в каждом кластере?
11. Продолжите работу с набором данных Olivetti faces. Обучите классификатор для выработки прогнозов относительно того, какой человек представлен на каждой фотографии, и оцените ее на проверочном наборе. Далее примените алгоритм K-Means в качестве инструмента по-

- нижения размерности и обучите классификатор на сокращенном наборе. Отыщите количество кластеров, которое позволяет классификатору добиться наилучшей эффективности: какой эффективности вы можете достичь? Что, если вы добавите признаки из сокращенного набора к исходным признакам (снова отыскав наилучшее количество кластеров)?
12. Обучите модель со смесью гауссовых распределений на наборе данных Olivetti faces. Чтобы ускорить выполнение алгоритма, вероятно, вы должны понизить размерность набора данных (например, примените PCA с предохраниением 99% величины дисперсии). Используйте модель для генерирования нескольких новых лиц (с помощью метода `sample()`) и визуализируйте их (если вы применяли PCA, то вам придется использовать его метод `inverse_transform()`). Попробуйте модифицировать ряд изображений (скажем, поверните их, зеркально отобразите, сделайте темнее) и посмотрите, может ли модель обнаруживать аномалии (т.е. сравните вывод метода `score_samples()` для нормальных изображений и для аномалий).
13. Некоторые методики понижения размерности могут применяться также для обнаружения аномалий. Например, возьмите набор данных Olivetti faces и сократите его с помощью PCA, предохраняя 99% величины дисперсии. Затем подсчитайте ошибку восстановления для каждого изображения. Далее возьмите некоторые модифицированные изображения, созданные в предыдущем упражнении, и взгляните на их ошибку восстановления: обратите внимание, насколько больше ошибки восстановления. Вычертив восстановленное изображение, вы увидите причину: алгоритм пытается восстановить нормальное лицо.

Решения приведенных упражнений доступны в приложении А.

# Нейронные сети и глубокое обучение



# Введение в искусственные нейронные сети с использованием Keras

Птицы вдохновили нас летать, репейники подтолкнули к созданию застежки типа “липучка” и сама природа способствовала бесчисленным другим изобретениям. В таком случае при творческих исканиях способов построения интеллектуальной машины вполне логично взглянуть на архитектуру мозга. Это и есть ключевая идея,ложенная в основу *искусственных нейронных сетей* (*artificial neural network* — ANN): сеть ANN — это модель МО, создатели которой были вдохновлены сетями биологических нейронов, находящимися в человеческих мозгах. Однако хотя появление самолетов было инспирировано птицами, самолеты 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 системами<sup>1</sup>.

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 Siri), рекомендация лучших видеороликов для просмотра сотнями миллионов людей ежедневно (например, YouTube) или обучение с целью победы чемпиона мира в игре *го* (DeepMind AlphaGo).

<sup>1</sup> Вы можете взять 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.

В первой части главы мы представим искусственные нейронные сети, начав с краткого турне в самые ранние архитектуры ANN и добравшись до многослойных персептронов (*Multi-Layer Perceptron — MLP*), которые интенсивно применяются в наши дни (другие архитектуры будут исследоваться в последующих главах). Во второй части главы мы посмотрим, как реализовывать нейронные сети, используя популярный API-интерфейс Keras. Он представляет собой превосходно спроектированный и простой высокоуровневый API-интерфейс для построения, обучения, оценки и запуска нейронных сетей. Но не обманывайтесь его простотой: он вполне выразительный и гибкий, чтобы позволить вам строить широкий выбор архитектур нейронных сетей. На самом деле API-интерфейса Keras, вероятно, будет достаточно для большинства возникающих у вас сценариев применения. И если вам когда-нибудь понадобится дополнительная гибкость, то вы всегда сможете создать специальные компоненты Keras, используя его низкоуровневый API-интерфейс, как будет показано в главе 12.

Но сначала давайте перенесемся назад во времени, чтобы посмотреть, как появились нейронные сети!

## 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 статье (<https://holm.info/43>)<sup>2</sup> 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 привели к широко распространенной уверенности в том, что вскоре мы будем общаться с по-настоящему интеллектуальными машинами. Когда в 1960-х годах стало ясно, что такое обещание неосуществимо (по крайней мере, какое-то время), финансирование улетучилось, и для сетей ANN началась долгая зима. В начале 1980-х годов были изобретены новые архитектуры сетей и разработаны лучшие методики обу-

<sup>2</sup> Уоррен Мак-Каллок и Уолтер Питтс, *A Logical Calculus of Ideas Immanent in Nervous Activity* (Логическое исчисление идей, присущих нервной деятельности), *The Bulletin of Mathematical Biology* 5, выпуск 4 (1943 г.): с. 115–113.

чения, что возобновило интерес к коннекционизму (исследованию нейронных сетей). Но прогресс был медленным, и в 1990-х годах появились другие мощные методики МО, такие как методы опорных векторов (см. главу 5). Казалось, что они обеспечивали лучшие результаты и обладали более строгими теоретическими основами, чем сети ANN, поэтому исследование нейронных сетей в очередной раз было приостановлено.

Сейчас мы наблюдаем еще одну волну интереса к сетям ANN. Ослабнет ли новая волна подобно предшествующим? Есть несколько веских причин полагать, что на этот раз ситуация иная и возобновленный интерес к сетям ANN 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-ми годами делает возможным обучение больших нейронных сетей за приемлемое время. Отчасти ситуация объясняется законом Мура (в течение последних 50 лет количество компонентов в интегральных микросхемах удваивалось примерно каждые два года), но также связана с индустрией компьютерных игр, которая стимулировала производство мощных графических процессоров миллионами единиц. Кроме того, облачные платформы сделали эту мощь доступной любому.
- 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 мягкими. 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 (и 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). Это необычно выглядящая клетка встречается главным образом в мозге животных. Она состоит из *тела клетки* (*cell body*), содержащего ядро и большинство сложных компонентов клетки, множества ветвящихся удлинений, называемых *дendritами* (*dendrite*), плюс одного очень крупного удлинения, называемого *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 телами клеток других нейронов<sup>3</sup>.

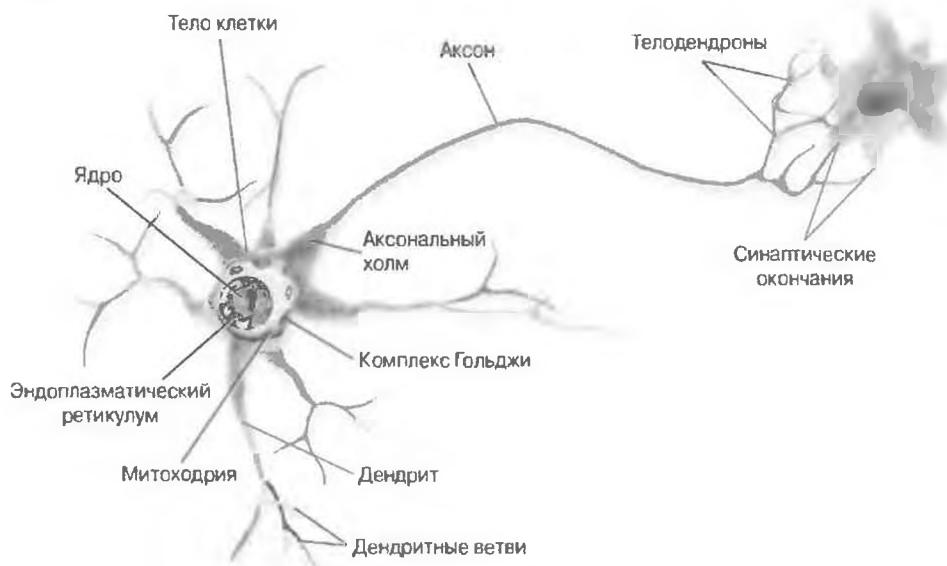


Рис. 10.1. Биологический нейрон<sup>4</sup>

<sup>3</sup> В действительности они не привязаны, а просто настолько близки, что способны очень быстро обмениваться химическими сигналами.

<sup>4</sup> Рисунок Брюса Блауса (Creative Commons 3.0 (<https://creativecommons.org/licenses/by/3.0/>)). Воспроизведен из <https://en.wikipedia.org/wiki/Neuron>.

Биологические нейроны производят короткие электрические импульсы, называемые биоэлектрическими потенциалами (*action potential* — AP), или просто сигналами (*signal*), которые перемещаются вдоль аксонов и заставляют синапсы высвобождать химические сигналы, называемые нейротрансмиттерами (*neurotransmitter*). Когда нейрон получает достаточное количество нейротрансмиттеров в пределах нескольких миллисекунд, он возбуждает собственные электрические импульсы (фактически это зависит от нейротрансмиттеров, т.к. некоторые из них препятствуют возбуждению нейрона).

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 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)<sup>5</sup> все еще является объектом активных исследований, но некоторые части мозга были отражены, и кажется, что нейроны часто организованы в последовательные слои, особенно в коре головного мозга (т.е. внешнем слое мозга), как показано на рис. 10.2.

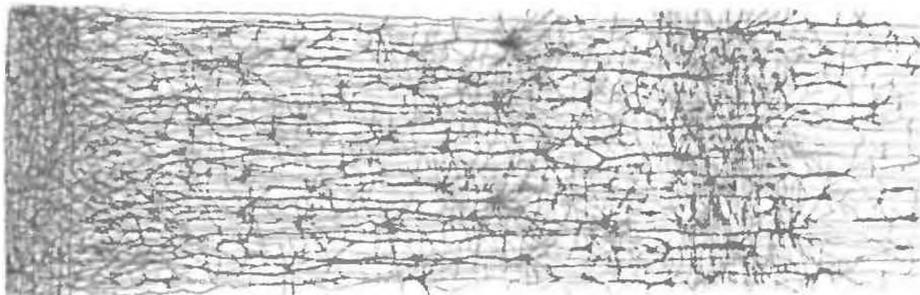


Рис. 10.2. Множество слоев в биологической нейронной сети  
(кора головного мозга человека)<sup>6</sup>

<sup>5</sup> В контексте машинного обучения выражение “нейронные сети” в большинстве случаев относится к сетям ANN, а не BNN.

<sup>6</sup> Изображение расслоения коры головного мозга, автором которого является Сантьяго Рамон-и-Кахаль (публичная собственность). Воспроизведено из [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex).

## Логические вычисления с помощью нейронов

Мак-Каллок и Питтс предложили очень простую модель биологического нейрона, которая позже стала известной как *искусственный нейрон (artificial neuron)*: он имеет один или большее количество двоичных входов (включен/выключен) и один двоичный выход. Искусственный нейрон 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. Чтобы взглянуть, каким образом работает такая сеть, давайте построим несколько сетей ANN, которые выполняют разнообразные логические вычисления (рис. 10.3), исходя из предположения, что нейрон активируется, когда активны хотя бы два из его входов.

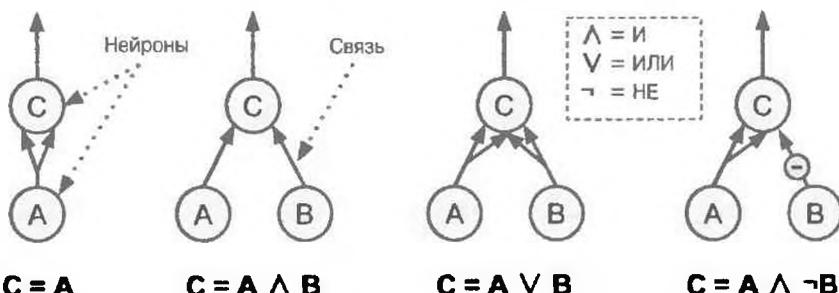


Рис. 10.3. Сети ANN, выполняющие простые логические вычисления

Посмотрим, что делают сети на рис. 10.3.

- Первая сеть слева — это функция тождественного отображения: если нейрон A активирован, тогда нейрон C тоже активируется (т.к. он получает два входных сигнала от нейрона A), но если нейрон A отключен, то нейрон C также отключен.
- Вторая сеть выполняет операцию логического “И”: нейрон C активируется, только когда активированы и нейрон A, и нейрон B (для активации нейрона C одиночного входного сигнала недостаточно).
- Третья сеть выполняет операцию логического “ИЛИ”: нейрон C активируется, если активирован либо нейрон A, либо нейрон B (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 активирован, а нейрон В отключен. Если нейрон А активирован все время, тогда получается операция логического “НЕ”: нейрон С активирован, когда нейрон В отключен, и наоборот.

Несложно представить, каким образом такие сети можно комбинировать для вычисления сложных логических выражений (за примером обращайтесь к упражнениям в конце главы).

## Персепtron

Персепtron (*Perceptron*) — одна из простейших архитектур ANN, придуманная Фрэнком Розенблаттом в 1957 году. Она основана на слегка отличающемся искусственном нейроне (рис. 10.4), который называется пороговым логическим элементом (*threshold logic unit* — TLU) либо иногда линейным пороговым элементом (*linear threshold unit* — L TU). Входы и выход являются числами (а не двоичными значениями “включено/выключено”) и с каждой входной связью ассоциирован вес. Элемент TLU вычисляет взвешенную сумму своих входов ( $z = w_1x_1 + w_2x_2 + \dots + w_nx_n = \mathbf{x}^T\mathbf{w}$ ), затем применяет к полученной сумме ступенчатую функцию (*step function*) и выдает результат:  $h_w(\mathbf{x}) = \text{step}(z)$ , где  $z = \mathbf{x}^T\mathbf{w}$ .

Самой распространенной ступенчатой функцией, используемой в персептранах, является ступенчатая функция Хевисайда (*Heaviside step function*), приведенная в уравнении 10.1. Иногда вместо нее применяется функция знака (или сигнум-функция; *sign function*).

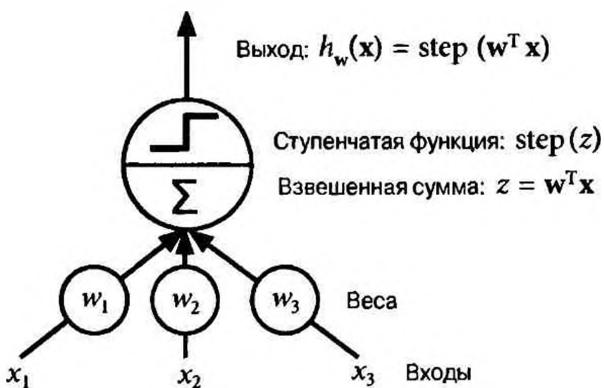


Рис. 10.4. Пороговый логический элемент: искусственный нейрон, который вычисляет взвешенную сумму своих входов и применяет к ней ступенчатую функцию

## Уравнение 10.1. Распространенная ступенчатая функция, применяемая в персептранах (предполагается, что порог равен 0)

$$\text{heaviside}(z) = \begin{cases} 0, & \text{если } z < 0 \\ 1, & \text{если } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1, & \text{если } z < 0 \\ 0, & \text{если } z = 0 \\ +1, & \text{если } z > 0 \end{cases}$$

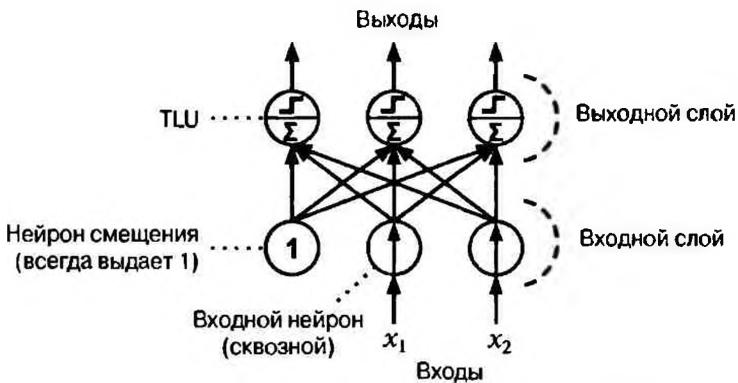
Одиночный элемент TLU может использоваться для простой линейной двоичной классификации. Он рассчитывает линейную комбинацию входов и если результат превышает некоторый порог, то выдает положительный класс, а иначе — отрицательный (подобно классификатору на основе логистической регрессии или линейного метода опорных векторов). Например, вы могли бы применять одиночный элемент TLU 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 TLU в таком случае означает нахождение правильных значений для  $w_0$ ,  $w_1$  и  $w_2$  (алгоритм обучения мы вскоре обсудим).

Персептрон просто состоит из единственного слоя элементов TLU<sup>7</sup>, где каждый элемент TLU соединен со всеми входами. Когда все нейроны внутри слоя соединены с каждым нейроном из предыдущего слоя (т.е. с его входными нейронами), такой слой называется *полносвязанным* (*fully connected*), или *плотным* (*dense*).

Входы персептрана подаются специальным сквозным нейронам, называемым *входными нейронами* (*input neuron*): они передают на выход все, что было подано на входе. Все входные нейроны формируют *входной слой*. Кроме того, как правило, добавляется дополнительный признак смещения ( $x_0 = 1$ ): он обычно представляется с применением нейрона специального типа, называемого *нейроном смещения* (*bias neuron*), который все время выдает 1. На рис. 10.5 показан персептрон с двумя входами и тремя выходами. Такой персептрон способен классифицировать образцы одновременно в три разных двоичных класса, что делает его многовыходовым классификатором.

Благодаря магии линейной алгебры уравнение 10.2 делает возможным эффективный расчет выходов слоя искусственных нейронов для нескольких образцов за один раз.

<sup>7</sup> Название “персептрон” иногда используется для обозначения крошечной сети с единственным элементом TLU.



**Рис. 10.5.** Архитектура персептрана с двумя входными нейронами, одним нейроном смещения и тремя выходными нейронами

### Уравнение 10.3. Расчет выходов полносвязанного слоя

$$h_{W,b}(X) = \phi(XW + b)$$

Вот что находится в этом уравнении.

- Как всегда,  $X$  представляет матрицу входных признаков. В ней предусмотрено по одной строке на образец и по одному столбцу на признак.
- Матрица весов  $W$  содержит все веса связей за исключением весов связей из нейрона смещения. В ней предусмотрено по одной строке на входной нейрон и по одному столбцу на искусственный нейрон в слое.
- Вектор смещения  $b$  содержит все веса связей между нейроном смещения и искусственными нейронами. В нем предусмотрено по одному члену смещения на искусственный нейрон.
- Функция  $\phi$  называется *функцией активации (activation function)*: когда искусственными нейронами являются элементы TLU, это ступенчатая функция (но вскоре мы обсудим другие функции активации).

Итак, каким же образом обучается персептрон? Алгоритм обучения персептрана, предложенный 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 Зигрид Левель подытожил идею Хебба в виде легко запоминающейся фразы: “Клетки, которые срабатывают вместе, связаны вместе” (“Cells that

fire together, wire together”); т.е. вес связи между двумя нейронами имеет тенденцию к увеличению, когда они срабатывают одновременно. Впоследствии правило стало известным под названием правило Хебба (или обучение по Хеббу (*Hebbian learning*)). Персептроны обучаются с применением варианта данного правила, в котором во внимание принимается ошибка, допущенная сетью, когда она вырабатывает прогноз; правило обучения персептрана укрепляет связи, помогающие сократить ошибку. Говоря точнее, персептруну передается один обучающий образец за раз, и для каждого образца он вырабатывает свои прогнозы. Для каждого выходного нейрона, который выдает неправильный прогноз, персептран усиливает веса связей из входов, которые способствовали корректному прогнозу. Правило описано в уравнении 10.3.

### Уравнение 10.3. Правило обучения персептрана (обновление весов)

$$w_{i,j} \text{ (следующий шаг)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

В этом уравнении:

- $w_{i,j}$  — 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;
- $\eta$  — скорость обучения.

Граница решений каждого выходного нейрона линейна, так что персептраны неспособны к обучению на сложных паттернах (подобно классификаторам на основе логистической регрессии). Тем не менее, если обучающие образцы являются линейно сепарабельными, то Розенблatt демонстрирует, что алгоритм сойдется в решение<sup>8</sup>. Это называется *теоремой о сходимости персептрана (Perceptron convergence theorem)*.

Библиотека Scikit-Learn предоставляет класс `Perceptron`, который реализует сеть с одним элементом TLU. Он может использоваться практически так, как вы могли ожидать — скажем, с набором данных `iris` (введенным в главе 4):

<sup>8</sup> Обратите внимание, что решение не уникально: когда данные линейно сепарабельны, существует бесконечное число гиперплоскостей, которые могут их разделять.

```

import          as
from           import load_iris
from           import Perceptron

iris = load_iris()
X = iris.data[:, ( , )]    # длина лепестка, ширина лепестка
y = (iris.target == ).astype(np.int)    # ирис щетинистый?

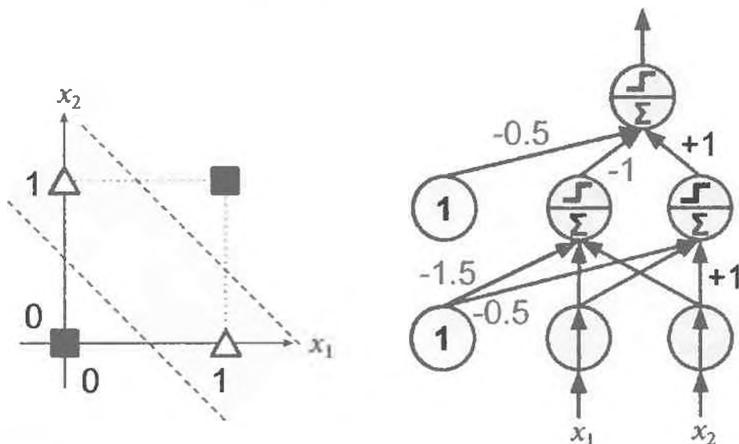
per_clf = Perceptron()
per_clf.fit(X, y)

y_pred = per_clf.predict([[ , ]])

```

Вы можете обнаружить, что алгоритм обучения персептрана 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). Обратите внимание, что в противоположность классификаторам на основе логистической регрессии персептраны не выдают вероятность класса; взамен они вырабатывают прогнозы, базируясь на жестком пороге. Это одна из веских причин отдавать предпочтение логистической регрессии перед персептранами.

В своей монографии *Perceptrons* (Персептраны), вышедшей в 1969 году, Марвин Мински и Сеймур Пейперт выделили несколько серьезных недостатков персептранов, 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).



*Рис. 10.6. Задача классификации XOR и решающий ее многослойный персептран*

Сказанное справедливо в отношении любой другой модели линейной классификации (вроде классификаторов на основе логистической регрессии), но исследователи ожидали от персепtronов намного большего, и некоторые были настолько разочарованы, что вообще перестали заниматься нейронными сетями, переключившись на высокоуровневые задачи, такие как логика, решение и поиск.

Однако выяснилось, что определенные ограничения персепtronов можно устраниТЬ за счет укладывания друг на друга множества персепtronов. Результирующая сеть ANN называется *многослойным персепtronом* (*Multi-Layer Perceptron — MLP*). Так, MLP способен решать задачу XOR, в чем вы можете удостовериться путем вычисления выхода MLP, представленного справа на рис. 10.6: для входов (0, 0) или (1, 1) сеть выдает 0, а для входов (0, 1) или (1, 0) — 1. Все связи имеют вес, равный 1, кроме четырех связей, для которых веса показаны явно. Попытайтесь удостовериться в том, что данная сеть действительно решает задачу XOR!

## Многослойный персепtron и обратное распространение

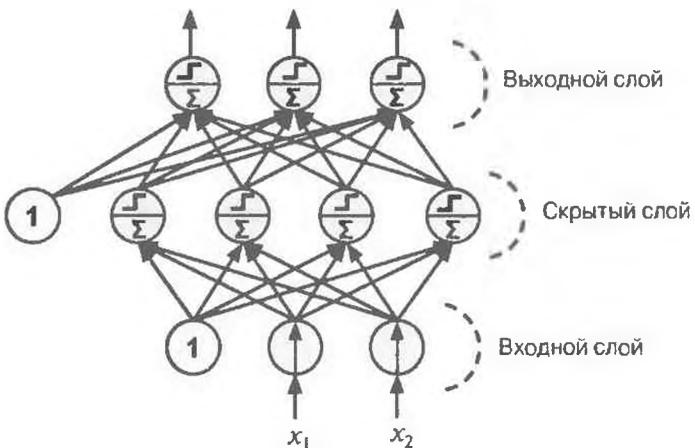
Многослойный персепtron состоит из одного (сквозного) входного слоя, одного или большего числа слоев элементов TLU, называемых *скрытыми слоями* (*hidden layer*), и еще одного слоя элементов TLU, который называется *выходным слоем* (*output layer*), как показано на рис. 10.7. Слои, расположенные близко к входному слою, обычно называют *низшими слоями*, а слои, находящиеся близко к выходам — *высшими слоями*. Каждый слой кроме выходного включает нейрон смещения и полностью связан со следующим слоем.



Сигнал протекает только в одном направлении (от входов к выходам), поэтому такая архитектура является примером *нейронной сети прямого распространения* (*feedforward neural network — FNN*).

Когда сеть ANN содержит глубокую стопку скрытых слоев<sup>9</sup>, она называется *глубокой нейронной сетью* (*deep neural network — DNN*). Область глубокого обучения исследует сети DNN и в более общем случае модели, содержащие глубокие стопки вычислений. Тем не менее, многие люди говорят о глубоком обучении всякий раз, когда задействованы нейронные сети (даже мелкие).

<sup>9</sup> В 1990-х годах сеть ANN с более чем двумя скрытыми слоями считалась глубокой. В наши дни часто встречаются сети ANN с десятками и даже сотнями слоев, поэтому определение “глубокая” стало довольно расплывчатым.



**Рис. 10.7.** Архитектура многослойного персептрона с двумя входами, одним скрытым слоем из четырех нейронов и тремя выходными нейронами (здесь показаны нейроны смещения, но обычно они являются неявными)

В течение многих лет исследователи изо всех сил старались отыскать способ обучения многослойных персептронов, но без какого-либо успеха. Наконец, в 1986 году, Джекфри Хинтон и Рональд Уильямс опубликовали революционную статью (<https://homl.info/44>)<sup>10</sup>, представляющую алгоритм обучения с обратным распространением (*backpropagation training algorithm*; *backpropagation* — обратное распространение ошибки обучения), который по-прежнему используется в наши дни. Выражаясь кратко, он представляет собой градиентный спуск (введенный в главе 4), применяющий эффективную методику для автоматического вычисления градиентов<sup>11</sup>: всего за два прохода через сеть (один вперед, один назад), алгоритм с обратным распространением способен вычислить градиент ошибки сети относительно каждого отдельного параметра модели. Другими словами, он может выяснить, каким образом каждый вес связи и каждый член смещения должны быть подстроены для снижения ошибки. Получив эти градиенты, алгоритм с обратным распространением просто выполняет шаг обычного градиентного спуска, затем полный процесс повторяется до тех пор, пока сеть не сойдется в решение.

<sup>10</sup> Дэвид Румельхарт и др., *Learning Internal Representations by Error Propagation* (Изучение внутренних представлений путем распространения ошибки), технический отчет Центра защиты технической информации (сентябрь 1985 года).

<sup>11</sup> В действительности эта методика была независимо несколько раз придумана разными исследователями в различных областях, начиная с Пола Уэрбоса в 1974 году.



Автоматическое вычисление градиентов называется *автоматическим дифференцированием* (*automatic differentiation* — *autodiff*). Существуют различные методики автоматического дифференцирования, обладающие своими достоинствами и недостатками. При обратном распространении используется так называемое *автоматическое дифференцирование в обратном режиме* (*reverse-mode autodiff*). Оно отличается большой скоростью, высокой точностью и хорошо подходит, когда дифференцируемая функция имеет много переменных (например, весов связей) и мало выходов (скажем, одну потерю). Дополнительные сведения об автоматическом дифференцировании приведены в приложении Г.

Давайте рассмотрим алгоритм более подробно.

- Он обрабатывает по одному мини-пакету за раз (каждый из них содержит, например, 32 образца) и многократно проходит по полному обучающему набору. Каждый проход называется *эпохой* (*epoch*).
- Каждый мини-пакет передается входному слою сети, который отправляет его первому скрытому слою. Затем алгоритм вычисляет выход всех нейронов в этом слое (для каждого образца в мини-пакете). Результат передается следующему слою, его выход вычисляется и передается следующему слою — процесс продолжается до тех пор, пока не будет получен выход последнего слоя, выходного. Это *прямой проход*: он похож на выработку прогнозов за исключением того, что все промежуточные результаты сохраняются, поскольку они нужны для обратного прохода.
- Далее алгоритм измеряет выходную ошибку сети (т.е. использует функцию потерь, которая сравнивает желаемый выход сети с действительным и возвращает количественный показатель ошибки).
- Следующим он вычисляет размер вклада в ошибку каждой выходной связи. Это делается аналитически за счет применения *цепного правила* (*chain rule*), являющегося вероятно наиболее фундаментальным правилом в исчислении, которое делает данный шаг быстрым и точным.
- Затем алгоритм измеряет, сколько таких вкладов в ошибку поступает от каждой связи в слое ниже, снова используя цепное правило, и работает в обратном направлении до тех пор, пока не достигнет входного слоя.

Как объяснялось ранее, такой обратный проход эффективно измеряет градиент ошибок по всем весам связей в сети за счет обратного распространения градиента ошибок через сеть (отсюда и название алгоритма).

- Наконец, алгоритм выполняет шаг градиентного спуска для подстройки всех весов связей в сети с применением только что вычисленных градиентов ошибок.

Алгоритм с обратным распространением настолько важен, что заслуживает короткого резюме: сначала он вырабатывает прогноз для каждого обучающего образца (прямой проход) и измеряет ошибку, затем проходит через каждый слой в обратном направлении, измеряя вклад в ошибку каждой связи (обратный проход), и в заключение подстраивает веса связей, чтобы уменьшить ошибку (шаг градиентного спуска).



Важно инициализировать веса связей всех скрытых слоев случайным образом, иначе обучение потерпит неудачу. Например, если вы инициализируете все веса и смещения нулями, тогда все нейроны в заданном слое будут полностью идентичными, и потому обратное распространение повлияет на них совершенно одинаково, так что они станут теми же самыми. Другими словами, невзирая на наличие сотен нейронов на слой, ваша модель будет действовать так, как если бы был только один нейрон на слой: она не окажется слишком интеллектуальной. Если взамен вы инициализируете веса случайным образом, то нарушите симметрию и позволите алгоритму с обратным распространением обучать несходную группу нейронов.

Для надлежащей работы алгоритма с обратным распространением его авторы внесли ключевое изменение в архитектуру MLP: они заменили ступенчатую функцию логистической (сигмоидальной) 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. На самом деле алгоритм с обратным распространением хорошо работает со многими другими функциями активации (*activation function*), а не только с логистической функцией. Ниже описаны еще два популярных варианта.

## Функция гиперболического тангенса: $\tanh(z) = 2\sigma(2z) - 1$

Подобно логистической функции эта функция активации 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). Такой диапазон имеет тенденцию делать выход каждого слоя более или менее центрированным вокруг  $0$  в начале обучения, что часто помогает ускорить сходимость.

## Функция выпрямленного линейного элемента (rectified linear unit — ReLU): $\text{ReLU}(z) = \max(0, z)$

Функция ReLU непрерывная, но, к сожалению, не дифференцируемая в точке  $z = 0$  (наклон резко меняется, что может привести к колебаниям градиентного спуска вокруг), и ее производная равна  $0$  для  $z < 0$ . Тем не менее, на практике эта функция работает очень хорошо и обладает преимуществом более быстрого вычисления, так что она стала стандартным вариантом<sup>12</sup>. Кроме того, тот факт, что она не имеет максимального выходного значения, 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. Но подождите! Для чего вообще нужны функции активации? Соединив в цепочку несколько линейных трансформаций, все, что вы в итоге получите — это линейную трансформацию. Например, если  $f(x) = 2x + 3$  и  $g(x) = 5x - 1$ , тогда соединение в цепочку указанных двух функций дает еще одну линейную функцию:  $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ . Таким образом, если какая-то нелинейность между слоями отсутствует, то даже глубокая стопка слоев эквивалентна одиночному слою, а потому с ее помощью вы не сможете решать очень сложные задачи. И наоборот, достаточно крупная сеть DNN с нелинейными активациями теоретически способна аппроксимировать любую непрерывную функцию.

Итак, вы знаете, откуда произошли нейронные сети, их архитектуру и способ вычисления выходов. Вы также ознакомились с алгоритмом обратного распространения. Но что со всем этим делать?

<sup>12</sup> Биологические нейроны, по-видимому, реализуют примерно сигмоидальную (S-образную) функцию активации и потому исследователи очень долго придерживались сигмоидальных функций. Но оказалось, что в сетях ANN функция ReLU обычно работает лучше. Это один из случаев, когда биологическая аналогия вводила в заблуждение.

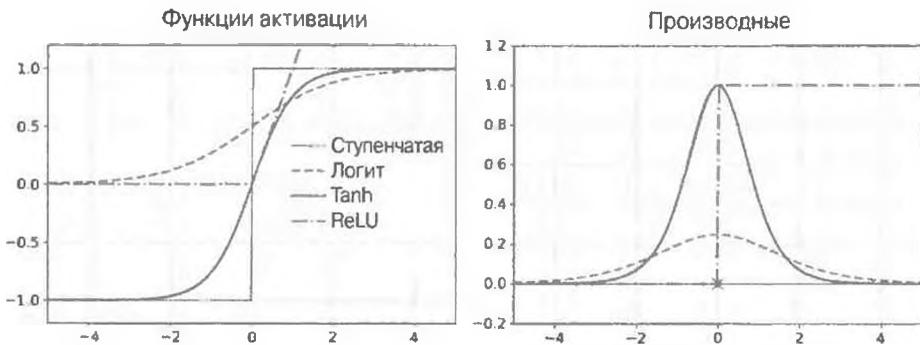


Рис. 10.8. Функции активации и их производные

## Многослойные персептроны для регрессии

Прежде всего, многослойные персептроны можно использовать для задач регрессии. Если вы хотите прогнозировать одиночное значение (скажем, стоимость дома, располагая множеством его признаков), тогда вам понадобится единственный выходной нейрон: его выходом будет спрогнозированное значение. Для многомерной регрессии (т.е. для прогнозирования множества значений за раз) вам необходим один выходной нейрон на измерение выхода. Например, чтобы найти центр объекта в изображении, нужно спрогнозировать двумерные координаты, а потому потребуется два выходных нейрона. Если вы также хотите поместить ограничивающий прямоугольник вокруг объекта, то вам понадобятся еще два числа: ширина и высота объекта. В итоге получается четыре выходных нейрона.

В общем случае при построении многослойного персептрана для регрессии вы не захотите применять какую-либо функцию активации для выходных нейронов, поэтому они вольны выдавать любой диапазон значений. Если вы желаете гарантировать, что выход всегда будет положительным, тогда можете использовать в выходном слое функцию активации ReLU. В качестве альтернативы вы можете применять функцию активации *softplus*, которая является гладким вариантом ReLU:  $\text{softplus}(z) = \log(1 + \exp(z))$ . Она близка к 0 при отрицательном  $z$  и близка к  $z$  при положительном  $z$ . Наконец, если вы хотите гарантировать, что прогнозы будут входить в заданный диапазон значений, тогда можете использовать логистическую функцию или гиперболический тангенс и затем масштабировать метки к надлежащему диапазону: от 0 до 1 для логистической функции и от -1 до 1 для гиперболического тангенса.

Функцией потерь, применяемой во время обучения, обычно будет среднеквадратическая ошибка, но если в обучающем наборе имеется много выбросов, то вы можете отдать предпочтение средней абсолютной ошибке. Альтернативно вы можете применять *потерю Хьюбера* (*Huber loss*), которая представляет собой комбинацию обеих ошибок.



Потеря Хьюбера является квадратичной, когда ошибка меньше порога  $\delta$  (обычно равного 1), но линейной, когда ошибка больше  $\delta$ . Линейная часть делает ее менее чувствительной к выбросам, чем среднеквадратическая ошибка, а квадратичная часть позволяет быстрее сходиться и быть более точной, чем средняя абсолютная ошибка.

В табл. 10.1 приведена сводка по типовой архитектуре многослойного персептрона для регрессии.

**Таблица 10.1. Типовая архитектура многослойного персептрона для регрессии**

Гиперпараметр	Типичное значение
Количество входных нейронов	1 на входной признак (например, $28 \times 28 = 784$ для MNIST)
Количество скрытых слоев	Зависит от задачи, но обычно от 1 до 5
Количество нейронов на скрытый слой	Зависит от задачи, но обычно от 10 до 100
Количество выходных нейронов	1 на измерение прогноза
Активация скрытых слоев	ReLU (или SELU (scaled exponential linear unit — масштабированный экспоненциальный линейный элемент), см. главу 11)
Активация выходного слоя	Отсутствует либо ReLU/softplus (для положительных выходов) или логистическая/tanh (для ограниченных выходов)
Функция потерь	MSE или MAE/Хьюбера (при наличии выбросов)

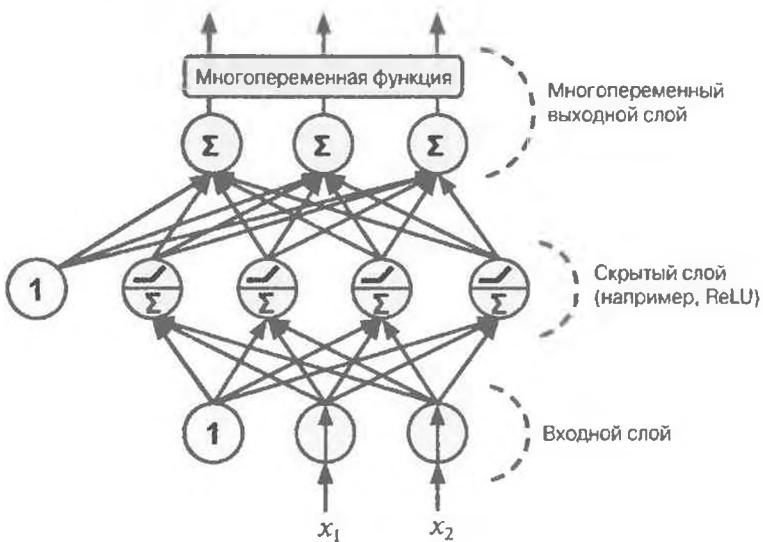
## Многослойные персептроны для классификации

Многослойные персептроны также могут применяться при решении задач классификации. Для задачи двоичной классификации вам необходим лишь один выходной нейрон, использующий логистическую функцию активации: выходом будет число между 0 и 1, которое вы можете интерпретировать как оценку вероятности положительного класса. Оценка вероятности отрицательного класса равна единице минус это число.

Многослойные персептроны также способны легко решать задачи многозначной двоичной классификации (см. главу 3). Например, вы могли бы

иметь систему классификации почтовых сообщений, которая прогнозирует, является ли входящее сообщение спамом или нет, и одновременно выдает прогноз, срочное оно или несрочное. В такой ситуации вам потребовалось бы два выходных нейрона, в которых применяется логистическая функция активации: первый выдавал бы вероятность того, что сообщение представляет собой спам, а второй — вероятность того, что оно срочное. В более общем случае вы выделили бы по одному выходному нейрону на каждый положительный класс. Обратите внимание, что выходные вероятности не обязательно дают в сумме 1. Это позволяет модели выдавать любые комбинации меток: вы можете иметь несрочный не спам, несрочный спам и вероятно даже срочный спам (хотя, наверное, такой вариант оказался бы ошибкой).

Если каждый образец может принадлежать только одному из трех и более возможных классов (скажем, от 0 до 9 при классификации изображений цифр), тогда вам нужно предусмотреть по одному выходному нейрону на класс и использовать *многопеременную (softmax)* функцию активации для всего выходного слоя (рис. 10.9). Многопеременная функция активации (введенная в главе 4) обеспечит нахождение всех оценок вероятности между 0 и 1, а также то, что в сумме они дадут 1 (это обязательно, если классы взаимоисключающие). Задача называется *многоклассовой классификацией*.



*Рис. 10.9. Современный многослойный персепtron (включающий ReLU и многопеременную функцию), предназначенный для классификации*

Что касается функции потерь, то поскольку мы прогнозируем распределения вероятностей, потеря в виде *перекрестной энтропии* (*cross-entropy loss*), также называемая логарифмической потерей (см. главу 4) в целом будет хорошим выбором.

В табл. 10.2 приведена сводка по типовой архитектуре многослойного персептрона для классификации.

**Таблица 10.2. Типовая архитектура многослойного персептрона для классификации**

Гиперпараметр	Двоичная классификация	Многозначная двоичная классификация	Мноклассовая классификация
Входной слой и скрытые слои	Такие же, как при регрессии	Такие же, как при регрессии	Такие же, как при регрессии
Количество выходных нейронов	1	1 на метку	1 на класс
Активация выходного слоя	Логистическая	Логистическая	Многопеременная
Функция потерь	Перекрестная энтропия	Перекрестная энтропия	Перекрестная энтропия



Прежде чем двигаться дальше, я рекомендую проработать упражнение 1 в конце главы. Вы попрактикуетесь с разнообразными архитектурами нейронных сетей и визуализируете их вывод с применением *TensorFlow Playground*. Это будет крайне полезно для лучшего понимания многослойных персептронов, включая влияние всех гиперпараметров (количества слоев и нейронов, функций активации и т.д.).

Теперь вам известны все концепции, необходимые для реализации многослойных персептронов с помощью Keras!

## Реализация многослойных персептронов с помощью Keras

Библиотека Keras представляет собой API-интерфейс для глубокого обучения, который позволяет легко строить, обучать, оценивать и запускать все виды нейронных сетей. Её документация (или спецификация) доступна по адресу <https://keras.io/>. Эталонная реализация (<https://github.com/keras-team/keras>), также называемая Keras, была разработана Франсуа

Шолле как часть исследовательского проекта<sup>13</sup> и выпущена в виде проекта с открытым кодом в марте 2015 года. Она быстро обрела популярность благодаря простоте использования, гибкости и великолепному проектному решению. При выполнении массивных вычислений, требуемых нейронными сетями, эталонная реализация полагается на вычислительный сервер. В настоящее время вы можете выбирать из трех популярных библиотек с открытым кодом для глубокого обучения: TensorFlow, Microsoft Cognitive Toolkit (CNTK) и Theano. По указанной причине во избежание любой путаницы мы будем ссылаться на эту эталонную реализацию как на *многосерверную библиотеку Keras*.

Начиная с конца 2016 года, были выпущены другие реализации. Теперь Keras можно запускать на платформах Apache MXNet, Apple Core ML, JavaScript или TypeScript (чтобы выполнять код Keras в веб-браузере) и PlaidML (может запускаться на всех видах графических процессоров, а не только Nvidia). Кроме того, сама библиотека TensorFlow теперь поставляется с собственной реализацией Keras, `tf.keras`. Она поддерживает в качестве сервера только TensorFlow, но обладает тем преимуществом, что предлагает ряд очень полезных дополнительных средств (рис. 10.10): скажем, поддерживает API-интерфейс TensorFlow для работы с данными (Data API), который обеспечивает простую и эффективную загрузку и предварительную обработку данных. По этой причине в книге мы будем пользоваться `tf.keras`. Однако в настоящей главе мы не станем применять какие-то специфичные для TensorFlow средства, так что код должен нормально выполняться и под управлением других реализаций Keras (по крайней мере, в Python) с минимальными модификациями, такими как изменение операторов импортирования.

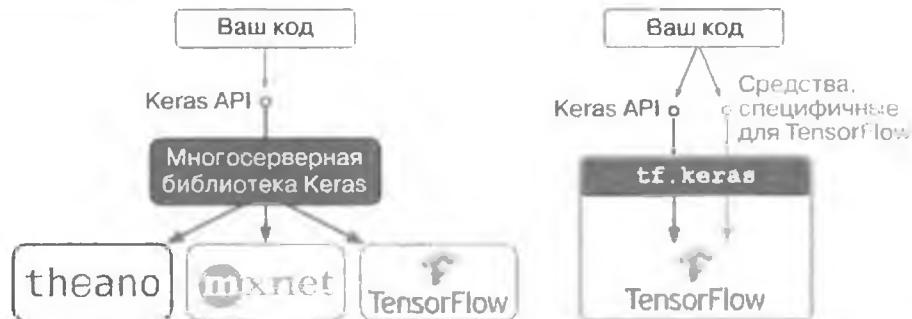


Рис. 10.10. Две реализации API-интерфейса Keras: многосерверная библиотека Keras (слева) и `tf.keras` (справа)

<sup>13</sup> Проект ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System — расширяемая операционная система для нейро-электронных интеллектуальных роботов).

Самой популярной библиотекой для глубокого обучения после Keras и TensorFlow является PyTorch (<https://pytorch.org/>) от Facebook. Хорошая новость в том, что API-интерфейс PyTorch похож на API-интерфейс Keras (отчасти потому, что оба API-интерфейса оказались под влиянием Scikit-Learn и Chainer (<https://chainer.org/>)), поэтому если вы знаете Keras, то при желании вам несложно будет переключиться на PyTorch. В 2018 году популярность PyTorch росла в геометрической прогрессии, главным образом благодаря своей простоте и великолепной документации, что не входило в список основных достоинств TensorFlow 1.x. Тем не менее, версия TensorFlow 2 вероятно столь же проста, как PyTorch, поскольку она приняла Keras в виде официального высокогоуровневого API-интерфейса, а ее разработчики значительно упростили и очистили остальной API-интерфейс. Документация также была полностью реорганизована, и теперь в ней гораздо легче найти интересующую информацию. Аналогичным образом главные недостатки PyTorch (например, ограниченная переносимость и отсутствие анализа вычислительных графов) в версии PyTorch 1.0 были в основном устранены. Здоровая конкуренция выгодна всем.

Хорошо, время приступить к написанию кода! Так как библиотека `tf.keras` упакована с TensorFlow, давайте начнем с установки TensorFlow.

## Установка TensorFlow 2

Предполагая, что вы установили Jupyter и Scikit-Learn в соответствии с инструкциями из главы 2, воспользуйтесь `pip` для установки TensorFlow. Если вы создали изолированную среду с применением `virtualenv`, тогда активизируйте ее:

```
$ cd $ML_PATH          # ваш рабочий каталог (например, $HOME/ml)
$ source my_env/bin/activate  # в среде Linux или macOS
$ .\my_env\Scripts\activate  # в среде Windows
```

Далее установите TensorFlow 2 (если вы не используете `virtualenv`, тогда понадобятся права администратора или добавление параметра `--user`):

```
$ python3 -m pip install --upgrade tensorflow
```



Для поддержки графических процессоров на момент написания книги вместо `tensorflow` необходимо устанавливать `tensorflow-gpu`, но команда разработчиков TensorFlow тру-

дится над тем, чтобы иметь единственную библиотеку, которая будет поддерживать системы, как оснащенные, так и не оснащенные графическими процессорами. Вам по-прежнему придется устанавливать добавочные библиотеки для поддержки графических процессоров (дополнительные сведения имеются по ссылке <https://tensorflow.org/install>). Мы рассмотрим графические процессоры более подробно в главе 19.

Чтобы протестировать установку, откройте окно командной оболочки Python или тетрадь Jupyter, импортируйте TensorFlow и `tf.keras` и выведите их версии:

```
>>> import          as
>>> from           import keras
>>> tf.__version__
'2.0.0'
>>> keras.__version__
'2.2.4-tf'
```

Второй отображается версия API-интерфейса Keras, реализованного в `tf.keras`. Обратите внимание, что она заканчивается на `-tf`, подчеркивая тот факт, что `tf.keras` реализует API-интерфейс Keras плюс ряд дополнительных средств, специфичных для TensorFlow.

А теперь зайдемся `tf.keras`! Мы начнем с построения простого классификатора изображений.

## Построение классификатора изображений с использованием API-интерфейса Sequential

Первым делом нам необходимо загрузить набор данных. В этой главе мы будем работать с Fashion MNIST, который является неожиданной заменой набора данных MNIST (введенного в главе 3). Он имеет точно такой же формат, как и MNIST (70000 полутоновых изображений  $28 \times 28$  пикселей с 10 классами), но изображения представляют предметы моды, а не рукописные цифры, так что каждый класс более разнообразен, и задача оказывается значительно сложнее, чем в случае MNIST. Скажем, простая линейная модель достигает примерно 92%-ной правильности на MNIST, но лишь около 83% на Fashion MNIST.

## Использование Keras для загрузки набора данных

Библиотека Keras предлагает несколько служебных функций для извлечения и загрузки распространенных наборов данных, включая MNIST, Fashion MNIST и набор данных с ценами на жилье в Калифорнии, который мы применяли в главе 2. Давайте загрузим Fashion MNIST:

```
fashion_mnist = keras.datasets.fashion_mnist  
(X_train_full, y_train_full), (X_test, y_test) =  
    fashion_mnist.load_data()
```

Когда набор данных MNIST или Fashion MNIST загружается с использованием Keras, а не Scikit-Learn, существует одно важное отличие: изображение представляется в виде массива  $28 \times 28$  вместо одномерного массива с размером 784. Вдобавок интенсивности пикселей представлены как целые числа (от 0 до 255), а не числа с плавающей точкой (от 0.0 до 255.0). Взглянем на форму и тип данных обучающего набора:

```
>>> X_train_full.shape  
(60000, 28, 28)  
>>> X_train_full.dtype  
dtype('uint8')
```

Обратите внимание, что набор данных уже расщеплен на обучающий набор и испытательный набор, но нет проверочного набора, поэтому мы его создадим. Кроме того, поскольку мы собираемся обучать нейронную сеть с применением градиентного спуска, то должны масштабировать входные признаки. Ради простоты мы будем масштабировать интенсивности пикселей вниз к диапазону 0–1, разделив их на 255.0 (что также преобразует их в числа с плавающей точкой):

```
X_valid, X_train =  
    X_train_full[:5000] / 255.0, X_train_full[5000:] / 255.0  
y_valid, y_train = y_train_full[:5000], y_train_full[5000:]
```

В наборе данных MNIST метка, равная 5, означала, что изображение представляет рукописную цифру 5. Было легко. Однако в наборе данных Fashion MNIST нам нужен список названий классов, чтобы знать, с чем мы имеем дело:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",  
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]  
# ["Футболка/топик", "Брюки", "Свитер", "Платье", "Пальто",  
#  "Сандалии", "Рубашка", "Кроссовки", "Сумка", "Башмаки"]
```

Например, первое изображение в обучающем наборе представляет пальто (Coat):

```
>>> class_names[y_train[0]]  
'Coat'
```

На рис. 10.11 показано несколько образцов из набора данных Fashion MNIST.



Рис. 10.11. Образцы из Fashion MNIST

### Создание модели с использованием API-интерфейса Sequential

А теперь давайте построим нейронную сеть! Вот многослойный персепtron для классификации, содержащий два скрытых слоя:

```
model = keras.models.Sequential()  
model.add(keras.layers.Flatten(input_shape=[28, 28]))  
model.add(keras.layers.Dense(300, activation="relu"))  
model.add(keras.layers.Dense(100, activation="relu"))  
model.add(keras.layers.Dense(10, activation="softmax"))
```

Разберем приведенный код строка за строкой.

- Первая строка создает модель Sequential — простейший вид модели Keras для нейронных сетей, которая состоит из единственной стопки последовательно соединенных слоев. Это называется API-интерфейсом Sequential.
- Затем мы строим первый слой и добавляем его к модели. Он представляет собой слой Flatten, чья роль заключается в преобразовании каждого входного изображения в одномерный массив: получив входные данные X, слой вычисляет X.reshape(-1, 1). Этот слой не имеет параметров; он предназначен лишь для выполнения простой предва-

рительной обработки. Поскольку он является первым слоем в модели, потребуется указать `input_shape`, что не включает размер пакета, а только форму образцов. Альтернативно в качестве первого слоя можно было бы добавить `keras.layers.InputLayer`, установив `input_shape=[28, 28]`.

- Далее мы добавляем скрытый слой `Dense` с 300 нейронами. Он будет применять функцию активации `ReLU`. Каждый слой `Dense` управляет собственной матрицей весов, содержащей все веса связей между нейронами и их входами. Он также управляет вектором членов смещения (один на нейрон). Получив входные данные, слой вычисляет уравнение 10.2.
- Затем мы добавляем второй скрытый слой `Dense`, который состоит из 100 нейронов и также использует функцию активации `ReLU`.
- Наконец, мы добавляем выходной слой `Dense` с 10 нейронами (по одному на класс), применяющий многопараметрическую функцию активации (т.к. классы взаимоисключающие).



Указание `activation="relu"` эквивалентно указанию `activation=keras.activations.relu`. В пакете `keras.activations` доступны другие функции активации, многие из которых мы будем использовать позже в книге. За полным списком обращайтесь по адресу <https://keras.io/activations/>.

Вместо добавления слоев по одному, как только что было сделано, вы можете передать список слоев при создании модели `Sequential`:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(300, activation="relu"),
    keras.layers.Dense(100, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Метод `summary()` модели отображает все слои модели<sup>14</sup>, в том числе имя слоя (которое генерируется автоматически, если оно не было установлено при создании слоя), форму его выхода (`None` означает, что размер пакета может быть каким угодно) и количество его параметров.

<sup>14</sup> Для генерации изображения вашей модели можете применять метод `keras.utils.plot_model()`.

## Использование примеров кода из keras.io

Примеры кода, документированные на веб-сайте keras.io, будут нормально работать с tf.keras, но понадобится изменить операторы импортирования. Скажем, возьмем следующий код из keras.io:

```
from           import Dense
output_layer = Dense( )
```

Вы должны изменить операторы импортирования, как показано ниже:

```
from           import Dense
output_layer = Dense( )
```

Или просто применять полные пути, если вам так больше нравится:

```
from           keras
output_layer = keras.layers.Dense( )
```

Такой подход многословнее, но я использую его в книге, чтобы вы легко видели, какие пакеты применять, и чтобы избежать путаницы между стандартными и специальными классами. В производственном коде я отдаю предпочтение предыдущему подходу. Многие люди также используют оператор from tensorflow.keras import layers, за которым следует layers.Dense(10).

Сводка заканчивается общим количеством параметров, включающим обучаемые и необучаемые параметры. Здесь мы имеем только обучаемые параметры (примеры необучаемых параметров мы увидим в главе 11):

```
>>> model.summary()
Model: "sequential"
```

Layer (type)	Output Shape	Param #
flatten (Flatten)	(None, 784)	0
dense (Dense)	(None, 300)	235500
dense_1 (Dense)	(None, 100)	30100
dense_2 (Dense)	(None, 10)	1010

Total params: 266,610

Trainable params: 266,610

Non-trainable params: 0

Обратите внимание, что слои Dense часто имеют много параметров. Скажем, первый скрытый слой поддерживает  $784 \times 300$  весов связей плюс 300 членов смещения, что в сумме дает 235 500 параметров! Это обеспечивает высокую гибкость модели в плане подгонки к обучающим данным, но также означает наличие риска ее переобучения, особенно когда вы не располагаете большим объемом обучающих данных. Позже мы еще вернемся к такому вопросу.

Вы можете легко получить список слоев модели, а также извлекать слой по индексу или по его имени:

```
>>> model.layers
[<tensorflow.python.keras.layers.core.Flatten at 0x132414e48>,
 <tensorflow.python.keras.layers.core.Dense at 0x1324149b0>,
 <tensorflow.python.keras.layers.core.Dense at 0x1356ba8d0>,
 <tensorflow.python.keras.layers.core.Dense at 0x13240d240>]
>>> hidden1 = model.layers[1]
>>> hidden1.name
'dense'
>>> model.get_layer('dense') is hidden1
True
```

Доступ ко всем параметрам слоя возможен с использованием его методов `get_weights()` и `set_weights()`. Для слоя Dense они включают веса связей и члены смещения:

```
>>> weights, biases = hidden1.get_weights()
>>> weights
array([[ 0.02448617, -0.00877795, -0.02189048, ..., -0.02766046,
         0.03859074, -0.06889391],
       ...,
       [-0.06022581,  0.01577859, -0.02585464, ..., -0.00527829,
        0.00272203, -0.06793761]], dtype=float32)
>>> weights.shape
(784, 300)
>>> biases
array([0., 0., 0., 0., 0., 0., 0., ..., 0., 0., 0.], dtype=float32)
>>> biases.shape
(300,)
```

Обратите внимание, что слой Dense инициализирует веса слоев случайным образом (как обсуждалось ранее, это необходимо для нарушения симметрии), а смещения были инициализированы нулями, т.е. все нормально. Если вы хотите применять другой метод инициализации, тогда при создании слоя можете установить `kernel_initializer` (*ядро (kernel)* — еще одно

называние матрицы весов связей) или `bias_initializer`. Мы продолжим обсуждать инициализаторы в главе 11, но если вас интересует полный список, то он доступен по адресу <https://keras.io/initializers/>.



Форма матрицы весов зависит от количества входов. Именно поэтому рекомендуется указывать `input_shape`, когда создается первый слой в модели `Sequential`. Тем не менее, даже если вы не укажете форму входа, то и тогда все будет в порядке: прежде чем действительно строить модель, Keras просто будет ждать до тех пор, пока форма входа не станет известной. Это произойдет, когда вы либо подадите фактические данные (например, во время обучения), либо вызовите метод `build()`. До тех пор, пока модель в действительности не построена, слои не будут иметь никаких весов, а вы не сможете делать определенные вещи (вроде вывода сводки по модели или сохранения модели). Таким образом, если при создании модели вам известна форма входа, то лучше указать ее.

## Компиляция модели

После создания модели вы должны вызвать ее метод `compile()`, чтобы указать используемую функцию потерь и оптимизатор. Дополнительно вы можете указать список добавочных показателей для вычисления во время обучения и оценки:

```
model.compile(loss="sparse_categorical_crossentropy",
               optimizer="sgd",
               metrics=["accuracy"])
```



Применение `loss="sparse_categorical_crossentropy"` эквивалентно использованию `loss=keras.losses.sparse_categorical_crossentropy`.



Аналогично указание `optimizer="sgd"` эквивалентно указанию `optimizer=keras.optimizers.SGD()`, а `metrics=["accuracy"]` — указанию `metrics=[keras.metrics.sparse_categorical_accuracy]` (когда применяется такая потеря). В книге мы будем использовать многие другие потери, оптимизаторы и показатели; полные их списки доступны по адресам <https://keras.io/losses>, <https://keras.io/optimizers> и <https://keras.io/metrics>.

Приведенный выше код требует пояснения. Прежде всего, мы применяем потерю "sparse\_categorical\_crossentropy", потому что располагаем разреженными метками (т.е. для каждого образца имеется лишь индекс целевого класса, от 0 до 9 в данном случае), а классы взаимоисключающие. Если бы взамен у нас была одна целевая вероятность на класс для каждого образца (такая как векторы в унитарном коде, например, [0., 0., 0., 1., 0., 0., 0., 0., 0.] для представления класса 3), тогда нам пришлось бы использовать потерю "categorical\_crossentropy". Если бы мы выполняли двоичную классификацию (с одной и более двоичными метками), то применяли бы в выходном слое функцию активации "sigmoid" (т.е. логистическую), а не "softmax", и использовали бы потерю "binary\_crossentropy".



Если вы хотите преобразовать разреженные метки (т.е. индексы классов) в метки, представляемые векторами в унитарном коде, тогда воспользуйтесь функцией `keras.utils.to_categorical()`. Чтобы сделать наоборот, примените функцию `np.argmax() axis=1`.

В отношении оптимизатора "sgd" означает, что мы будем обучать модель с использованием простого стохастического градиентного спуска. Другими словами, Keras будет выполнять описанный ранее алгоритм с обратным распространением (т.е. автоматическое дифференцирование в обратном режиме плюс градиентный спуск). В главе 11 мы обсудим более эффективные оптимизаторы (они улучшают часть градиентного спуска, но не автоматическое дифференцирование).



Когда применяется оптимизатор SGD, важно отрегулировать скорость обучения. Таким образом, обычно вы пожелаете использовать `optimizer=keras.optimizers.SGD(lr=???)`, чтобы устанавливать скорость обучения, а не `optimizer="sgd"`, где по умолчанию принимается `lr=0.01`.

Наконец, поскольку это классификатор, полезно измерить его показатель "accuracy" во время обучения и оценки.

## Обучение и оценка модели

Теперь модель готова к обучению, для чего просто нужно вызвать ее метод `fit()`:

```
>>> history = model.fit(X_train, y_train, epochs=30,
...                      validation_data=(X_valid, y_valid))
...
Train on 55000 samples, validate on 5000 samples
Epoch 1/30
55000/55000 [=====] - 3s 49us/sample - loss: 0.7218 - accuracy: 0.7660
- val_loss: 0.4973 - val_accuracy: 0.8366
Epoch 2/30
55000/55000 [=====] - 2s 45us/sample - loss: 0.4840 - accuracy: 0.8327
- val_loss: 0.4456 - val_accuracy: 0.8480
[...]
Epoch 30/30
55000/55000 [=====] - 3s 53us/sample - loss: 0.2252 - accuracy: 0.9192
- val_loss: 0.2999 - val_accuracy: 0.8926
```

Мы передаем методу `fit()` входные признаки (`X_train`) и целевые классы (`y_train`), а также количество эпох при обучении (иначе по умолчанию принимается только 1 эпоха, чего определенно не будет достаточно для схождения в хорошее решение). Мы передаем и проверочный набор (что необязательно). В конце каждой эпохи Keras будет измерять потерю и дополнительные показатели на этом наборе, которые очень полезны, позволяя видеть, как на самом деле работает модель. Если эффективность на обучающем наборе гораздо лучше, чем на проверочном наборе, тогда вероятно модель переобучается обучающим набором (или есть дефект вроде несоответствия данных между обучающим и проверочным наборами).

Вот и все! Нейронная сеть обучается<sup>15</sup>. На каждой эпохе во время обучения Keras отображает количество обработанных до сих пор образцов (наряду с индикатором продвижения), среднее время обучения на образец, а также потерю и правильность (или любые другие добавочные показатели, которые вы запросили) на обучающем и проверочном наборах. Вы можете заметить, что потеря при обучении снижается, что является хорошим сигналом, и правильность при проверке достигает 89.26% после 30 эпох.

<sup>15</sup> Если ваши обучающие или проверочные данные не соответствуют ожидаемой форме, тогда вы получите исключение. Пожалуй, это самая распространенная ошибка, а потому вы должны ознакомиться с сообщением об ошибке. На самом деле сообщение довольно ясное: например, если вы попытаетесь обучить имеющуюся модель на массиве, содержащем сглаженные изображения (`X_train.reshape(-1, 784)`), то получите следующее исключение: `ValueError: Error when checking input: expected flatten_input to have 3 dimensions, but got array with shape (60000, 784)` (Ошибка значения: ошибка при проверке входа: ожидалось `flatten_input` с 3 измерениями, но получен массив с формой (60000, 784)).

Она не слишком далека от правильности при обучении, поэтому не похоже, что происходило сильное переобучение.



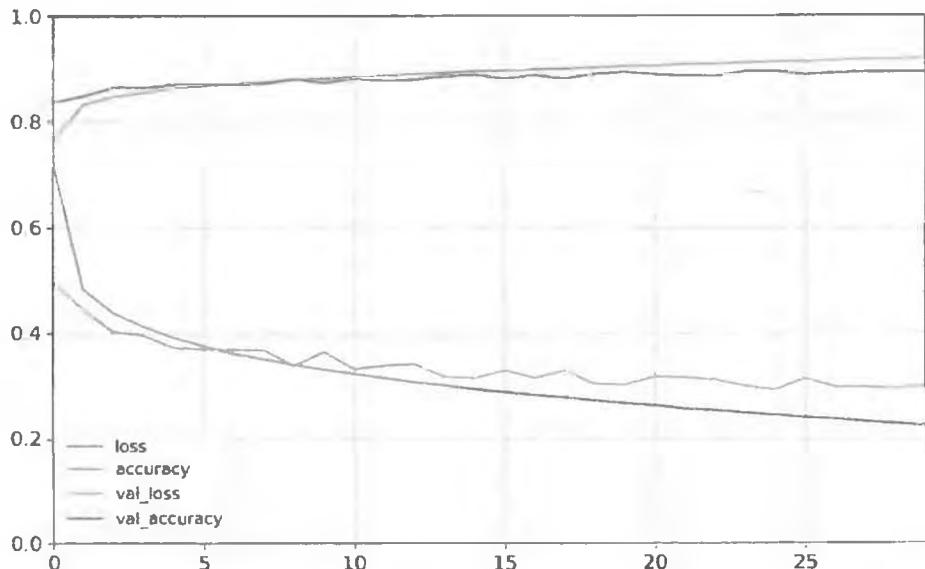
Вместо передачи проверочного набора с применением аргумента `validation_data` вы могли бы указать в `validation_split` относительную часть образцов из обучающего набора, которые библиотека Keras должна использовать для проверки. Скажем, `validation_split=0.1` сообщает Keras о необходимости применения для проверки последних 10% данных (перед тасованием).

Если обучающий набор был очень скошенным из-за того, что одних классов в нем чрезмерно, а других не хватает, то при вызове метода `fit()` было бы полезно установить аргумент `class_weight`, который придал бы больший вес недостаточно представленным классам и меньший вес классам, представленным чрезмерно. Эти веса использовались бы Keras при вычислении потери. Если необходимо иметь вес для каждого образца, тогда следует установить аргумент `sample_weight` (он заменяет `class_weight`). Веса для каждого образца оказываются полезными, когда некоторые образцы были помечены экспертами, в то время как остальные помечались с применением краудсорсинговой платформы: у вас может возникнуть желание назначить первым больший вес. Вы также можете предоставить веса образцов (но не веса классов) для проверочного набора, добавив их как третий элемент в кортеж `validation_data`.

Метод `fit()` возвращает объект `History`, содержащий параметры обучения (`history.params`), список прошедших эпох (`history.epoch`) и самое важное — словарь (`history.history`), в котором находятся потеря и добавочные показатели, измеренные в конце каждой эпохи на обучающем наборе и проверочном наборе (когда он есть). Если вы используете этот словарь для создания pandas-объекта `DataFrame` и вызовите его метод `plot()`, то получите кривые обучения, показанные на рис. 10.12:

```
import          as
import          as

pd.DataFrame(history.history).plot(figsize=( , ))
plt.grid(True)
plt.gca().set_ylim( , ) #устанавливает диапазон по вертикали в [0-1]
plt.show()
```



*Рис. 10.12. Кривые обучения: средняя потеря и правильность при обучении, измеренные в течение каждой эпохи, а также средняя потеря и правильность при проверке, измеренные в конце каждой эпохи*

Несложно заметить, что правильность при обучении и правильность при проверке монотонно возрастают во время обучения, тогда как потеря при обучении и потеря при проверке уменьшаются. Хорошо! Кроме того, кривые проверки близки к кривым обучения, т.е. модель не слишком сильно переобучается. В этом конкретном случае модель выглядит лучше работающей на проверочном наборе, чем на обучающем наборе в начале процесса обучения. Но в действительности все не так: на самом деле ошибка при проверке вычисляется в конце каждой эпохи, в то время как ошибка при обучении подсчитывается с применением скользящего среднего в течение каждой эпохи. Таким образом, кривая обучения должна быть сдвинута на половину эпохи влево. Если вы поступите так, то увидите, что кривые обучения и проверки в начале обучения практически полностью перекрываются.



При вычерчивании кривая обучения должна быть сдвинута на половину эпохи влево.

Эффективность при обучении в итоге превосходит эффективность при проверке, как обычно бывает, когда обучение выполняется достаточно долго. Вы можете сказать, что модель еще не полностью сошлась, т.к. потеря

при проверке по-прежнему понижается, а потому обучение вероятно должно продолжаться. Для этого нужно лишь снова вызвать метод `fit()`, потому что Keras просто продолжит обучение с того места, где оно было остановлено (вы должны быть в состоянии приблизиться к 89%-ной правильности при проверке).

Если вас не устраивает эффективность имеющейся модели, тогда вам придется возвратиться и отрегулировать гиперпараметры. Первым делом необходимо проверить скорость обучения. Если это не помогло, то нужно попробовать использовать другой оптимизатор (после изменения любого гиперпараметра всегда заново регулируйте скорость обучения). Если эффективность все еще не идеальна, тогда попробуйте подрегулировать такие гиперпараметры, как число слоев, количество нейронов на слой и типы функций активации, применяемые для каждого скрытого слоя. Вы также можете попробовать настроить другие гиперпараметры наподобие размера пакета (он устанавливается при вызове метода `fit()` с использованием аргумента `batch_size`, который по умолчанию равен 32). В конце главы мы возобновим обсуждение регулирования гиперпараметров. После того, как вы удовлетворены правильностью модели при проверке, потребуется оценить ее на испытательном наборе, чтобы получить оценку ошибки обобщения, прежде чем помещать модель в производственную среду. Достичь цели легко с применением метода `evaluate()`, который поддерживает также несколько других аргументов вроде `batch_size` и `sample_weight` (дополнительные сведения ищите в документации):

```
>>> model.evaluate(X_test, y_test)
10000/10000 [=====] - 0s 29us/sample - loss: 0.3340
                                         - accuracy: 0.8851
[0.3339798209667206, 0.8851]
```

Как выяснилось в главе 2, на испытательном наборе обычно получается чуть меньшая эффективность, чем на проверочном наборе, поскольку гиперпараметры подстраиваются на проверочном наборе, а не на испытательном (однако в рассмотренном примере мы не занимались регулированием гиперпараметров, так что более низкая правильность — просто невезение). Помните о необходимости устоять перед искушением отрегулировать гиперпараметры на испытательном наборе, иначе ваша оценка ошибки обобщения окажется слишком оптимистичной.

## Использование модели для выработывания прогнозов

Далее мы можем применять метод `predict()` модели для выработывания прогнозов на новых образцах. Так как фактически новые образцы отсутствуют, мы будем использовать первые три образца из испытательного набора:

```
>>> X_new = X_test[:3]
>>> y_proba = model.predict(X_new)
>>> y_proba.round(2)
array([[0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.01, 0. , 0.96],
       [0. , 0. , 0.98, 0. , 0.02, 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ]],
      dtype=float32)
```

Здесь видно, что для каждого образца модель оценивает по одной вероятности на класс, от класса 0 до класса 9. Например, для первого изображения она оценивает, что вероятность класса 9 (башмаки) составляет 96%, вероятность класса 5 (сандалии) — 3%, вероятность класса 7 (кроссовки) — 1%, а вероятности остальных классов пренебрежимо малы. Другими словами, модель “верит” в то, что первое изображение представляет обувь, вероятнее всего башмаки, но возможно сандалии или кроссовки. Если вас заботит только класс с наивысшей оценкой вероятности (даже когда она довольно невелика), тогда взамен можете применять метод `predict_classes()`:

```
>>> y_pred = model.predict_classes(X_new)
>>> y_pred
array([9, 2, 1])
>>> np.array(class_names)[y_pred]
array(['Ankle boot', 'Pullover', 'Trouser'], dtype='<U11')
# башмаки, свитер, брюки
```

В действительности классификатор корректно классифицировал все три изображения (рис. 10.13):

```
>>> y_new = y_test[:3]
>>> y_new
array([9, 2, 1])
```

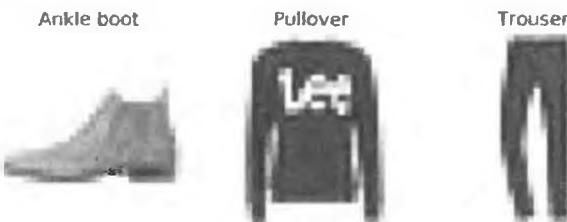


Рис. 10.13. Корректно классифицированные изображения из набора данных Fashion MNIST

Теперь вы знаете, как использовать API-интерфейс Sequential для построения, обучения, оценки и применения многослойного персептрона для классификации. Но как насчет регрессии?

## Построение многослойного персептрона для регрессии с использованием API-интерфейса Sequential

Давайте переключимся на задачу с жильем в Калифорнии и решим ее с применением нейронной сети для регрессии. Ради простоты для загрузки данных мы будем использовать функцию `fetch_california_housing()` из Scikit-Learn. Этот набор данных проще того, который применялся в главе 2, т.к. он содержит только числовые признаки (признак `ocean_proximity` отсутствует) и нет пропущенных значений. После загрузки данных мы расщепляем их на обучающий набор, проверочный набор и испытательный набор, а также масштабируем все признаки:

```
from import fetch_california_housing
from import train_test_split
from import StandardScaler

housing = fetch_california_housing()

X_train_full, X_test, y_train_full, y_test = train_test_split(
    housing.data, housing.target)
X_train, X_valid, y_train, y_valid = train_test_split(
    X_train_full, y_train_full)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_valid = scaler.transform(X_valid)
X_test = scaler.transform(X_test)
```

Использование API-интерфейса Sequential для построения, обучения, оценки и применения многослойного персептрона для регрессии с целью вырабатывания прогнозов довольно похоже на то, что мы делали для классификации. Основные отличия обусловлены тем фактом, что выходной слой имеет единственный нейрон (т.к. мы хотим прогнозировать одиночное значение) и не использует функцию активации, а функцией потерь является среднеквадратическая ошибка. Поскольку набор данных зашумлен, мы просто применяем единственный скрытый слой с меньшим количеством нейронов, чем было ранее, чтобы избежать переобучения:

```

model = keras.models.Sequential([
    keras.layers.Dense(30, activation="relu",
                       input_shape=X_train.shape[1:]),
    keras.layers.Dense(1)
])
model.compile(loss="mean_squared_error", optimizer="sgd")
history = model.fit(X_train, y_train, epochs=20,
                      validation_data=(X_valid, y_valid))
mse_test = model.evaluate(X_test, y_test)
X_new = X_test[:3]      # притворимся, что это новые образцы
y_pred = model.predict(X_new)

```

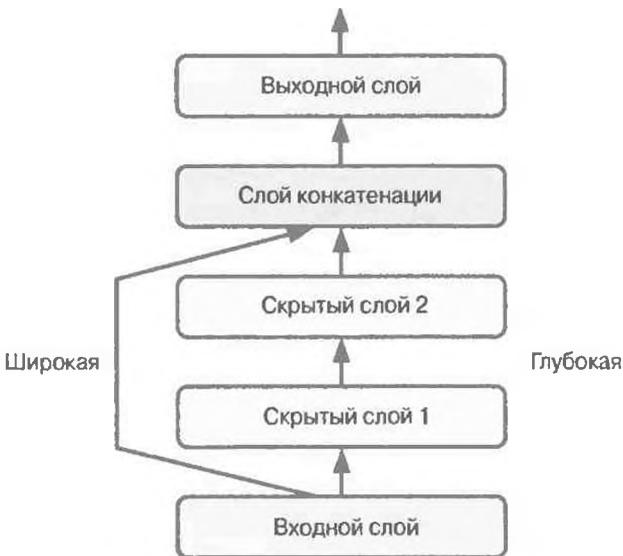
Как видите, использовать API-интерфейс Sequential достаточно легко. Тем не менее, хотя модели Sequential чрезвычайно распространены, иногда полезно строить нейронные сети с более сложными топологиями либо с множеством входов или выходов. Для такой цели библиотека Keras предлагает API-интерфейс Functional.

## Построение сложных моделей с использованием API-интерфейса Functional

Одним из примеров непоследовательных нейронных сетей может служить нейронная сеть *Wide & Deep* (*широкая и глубокая*). Ее архитектура была представлена в статье 2016 года (<https://homl.info/widedeep>)<sup>16</sup>, написанной Хэн-Цзей Чэном и др. Она связывает все или часть входов напрямую с выходным слоем, как показано на рис. 10.14. Такая архитектура позволяет нейронной сети узнавать и глубокие паттерны (применяя глубокий путь), и простые правила (через короткий путь)<sup>17</sup>. По контрасту с этим обычный многослойный персептрон вынуждает все данные протекать через полную стопку слоев; таким образом, простые паттерны в данных могут в итоге быть искажены подобной последовательностью трансформаций.

<sup>16</sup> Хэн-Цзей Чэн и др., *Wide & Deep Learning for Recommender Systems* (Широкое и глубокое обучение для систем выдачи рекомендаций), *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016 г.): с. 7–10.

<sup>17</sup> Короткий путь может также использоваться для снабжения нейронной сети вручную сконструированными признаками.



*Рис. 10.14. Нейронная сеть Wide & Deep*

Давайте построим такую нейронную сеть, чтобы решить задачу с жильем в Калифорнии:

```

input_ = keras.layers.Input(shape=X_train.shape[1:])
hidden1 = keras.layers.Dense(30, activation="relu")(input_)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_, hidden2])
output = keras.layers.Dense(1)(concat)
model = keras.Model(inputs=[input_], outputs=[output])

```

Разберем приведенный код строка за строкой.

- Первым делом нам необходимо создать объект `Input`<sup>18</sup>. Это спецификация вида входа, который будет получать модель, включая его форму (`shape`) и тип данных (`dtype`). Как мы вскоре увидим, в действительности модель может иметь множество входов.
- Далее мы создаем слой `Dense` с 30 нейронами, использующий функцию активации ReLU. При его создании обратите внимание, что мы вызываем его подобно функции, передавая вход. Именно потому API-интерфейс называется `Functional` (функциональный). Следует отметить, что мы лишь сообщаем Keras о том, как слои должны связываться вместе; никакие фактические данные пока не обрабатываются.

<sup>18</sup> Имя `input_` применяется для того, чтобы избежать путаницы со встроенной функцией `input()` языка Python.

- Затем мы создаем второй скрытый слой и снова применяем его как функцию. Обратите внимание, что мы передаем выход первого скрытого слоя.
- Далее мы создаем слой `Concatenate` и в очередной раз непосредственно используем его как функцию для конкатенации входа и выхода второго скрытого слоя. Вы можете отдать предпочтение функции `keras.layers.concatenate()`, которая создает слой `Concatenate` и немедленно вызывает его с заданными входами.
- Затем мы создаем выходной слой с единственным нейроном и без функции активации, а также вызываем его подобно функции, передавая результат конкатенации.
- В заключение мы создаем объект `Model` из Keras, указывая применяемые входы и выходы.

После построения модели Keras все остальное делается в точности, как было ранее, поэтому нет нужды здесь повторяться: вы должны скомпилировать модель, обучить ее, оценить и использовать для вырабатывания прогнозов.

Но что, если вы хотите передавать подмножество признаков через широкий путь и другое подмножество (возможно перекрывающееся) через глубокий путь (рис. 10.15)?

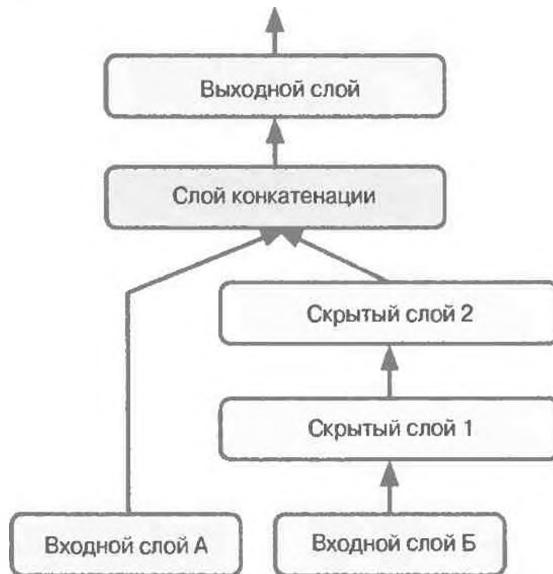


Рис. 10.15. Обработка множества входов

В таком случае одно из решений предусматривает применение множества входов. Например, пусть мы желаем передавать пять признаков через широкий путь (признаки от 0 до 4) и шесть признаков через глубокий путь (признаки от 2 до 7):

```
input_A = keras.layers.Input(shape=[5], name="wide_input")
input_B = keras.layers.Input(shape=[6], name="deep_input")
hidden1 = keras.layers.Dense(30, activation="relu")(input_B)
hidden2 = keras.layers.Dense(30, activation="relu")(hidden1)
concat = keras.layers.concatenate([input_A, hidden2])
output = keras.layers.Dense(1, name="output")(concat)
model = keras.Model(inputs=[input_A, input_B], outputs=[output])
```

Код не требует пояснений. Вы должны именовать, по крайней мере, самые важные слои, особенно когда модель становится чуть сложнее подобно текущей. Обратите внимание, что при создании модели мы указываем inputs=[input\_A, input\_B]. Теперь мы можем компилировать модель обычным образом, но при вызове метода fit() вместо передачи единственной входной матрицы X\_train мы обязаны передавать пару матриц (X\_train\_A, X\_train\_B): по одной на вход<sup>19</sup>. То же самое справедливо для X\_valid, а также для X\_test и X\_new, когда производится вызов evaluate() или predict():

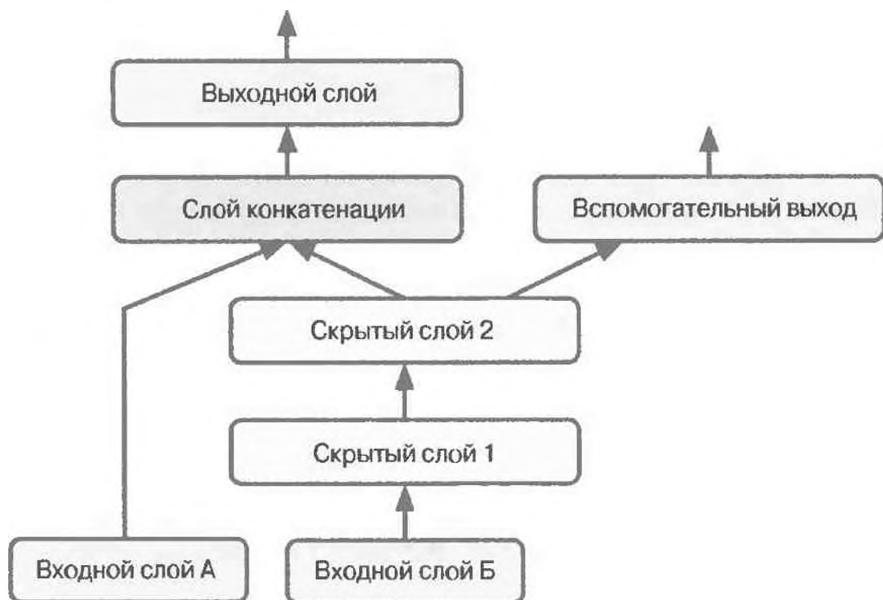
```
model.compile(loss="mse", optimizer=keras.optimizers.SGD(lr=1e-3))
X_train_A, X_train_B = X_train[:, :5], X_train[:, 2:]
X_valid_A, X_valid_B = X_valid[:, :5], X_valid[:, 2:]
X_test_A, X_test_B = X_test[:, :5], X_test[:, 2:]
X_new_A, X_new_B = X_test_A[:3], X_test_B[:3]
history = model.fit((X_train_A, X_train_B), y_train, epochs=20,
                     validation_data=((X_valid_A, X_valid_B), y_valid))
mse_test = model.evaluate((X_test_A, X_test_B), y_test)
y_pred = model.predict((X_new_A, X_new_B))
```

Существует много сценариев использования, в которых может возникнуть желание иметь множество выходов.

- Этого может требовать задача. Например, пусть необходимо определить местоположение и классифицировать главный объект на фотографии. Мы имеем как задачу регрессии (нахождение координат центра объекта, а также его ширины и высоты), так и задачу классификации.

<sup>19</sup> В качестве альтернативы вы можете передавать словарь, отображающий имена входов на значения входов, вроде {"wide\_input": X\_train\_A, "deep\_input": X\_train\_B}. Поступать так особенно удобно, когда имеется много входов, чтобы избежать нарушения порядка.

- Аналогично у вас может быть множество независимых задач, основанных на тех же самых данных. Конечно, вы могли бы обучить по одной нейронной сети на задачу, но во многих случаях результаты для всех задач будут лучше, если обучить единственную нейронную сеть с одним выходом на задачу. Причина в том, что нейронная сеть может узнавать признаки в данных, которые полезны между задачами. Скажем, вы могли бы выполнять *многозадачную классификацию* (*multitask classification*) на фотографиях лиц с применением одного выхода для классификации выражения лица человека (улыбается, удивлен и т.д.), а другого выхода для идентификации, носит ли он очки.
- Еще один сценарий использования — методика регуляризации (т.е. ограничение обучения, целью которого является сокращение переобучения и тем самым улучшение возможности модели обобщаться). Например, у вас может возникнуть желание добавить ряд вспомогательных выходов в архитектуру нейронной сети (рис. 10.16) для гарантирования того, что лежащая в основе часть сети узнает что-то полезное самостоятельно, не полагаясь на остаток сети.



*Рис. 10.16. Поддержка множества выходов; в этом примере добавляется вспомогательный выход для регуляризации*

Добавлять дополнительные выходы очень легко: просто свяжите их с соответствующими слоями и добавьте в список выходов модели. Скажем, приведенный ниже код строит сеть, представленную на рис. 10.16:

```
[...] # то же, что и ранее, вплоть до главного выходного слоя
output = keras.layers.Dense(1, name="main_output") (concat)
aux_output = keras.layers.Dense(1, name="aux_output") (hidden2)
model = keras.Model(inputs=[input_A, input_B],
                     outputs=[output, aux_output])
```

Каждому выходу понадобится собственная функция потерь. Следовательно, когда мы компилируем модель, то обязаны передать список потерь<sup>20</sup> (в случае передачи единственной потери Keras предполагает, что для всех выходов должна применяться та же самая потеря). По умолчанию Keras будет вычислять все указанные потери и просто складывать их, чтобы получить финальную потерю, используемую для обучения. Нас намного больше заботит главный выход, нежели вспомогательный (т.к. он применяется только для регуляризации), поэтому мы хотим назначить потере главного выхода гораздо больший вес.

К счастью, при компиляции модели имеется возможность установить все веса потерь:

```
model.compile(loss=["mse", "mse"],
               loss_weights=[0.9, 0.1], optimizer="sgd")
```

Теперь, когда мы обучаем модель, нам необходимо предоставить метки для каждого выхода. В рассматриваемом примере главный и вспомогательный выходы должны пытаться прогнозировать те же самые вещи, так что они обязаны применять те же самые метки. Таким образом, вместо передачи `y_train` нам необходимо передать (`y_train, y_train`) (и то же самое касается `y_valid` и `y_test`):

```
history = model.fit(
    [X_train_A, X_train_B], [y_train, y_train], epochs=20,
    validation_data=([X_valid_A, X_valid_B], [y_valid, y_valid]))
```

При оценке модели Keras будет возвращать суммарную потерю, а также все индивидуальные потери:

---

<sup>20</sup> В качестве альтернативы вы можете передавать словарь, который отображает каждое имя вывода на соответствующую потерю. В частности как для входов это удобно при наличии множества выходов, чтобы избежать нарушения порядка. Веса и показатели потерь (обсуждаются вскоре) также могут устанавливаться с использованием словарей.

```
total_loss, main_loss, aux_loss = model.evaluate(  
    [X_test_A, X_test_B], [y_test, y_test])
```

Подобным образом метод `predict()` будет возвращать прогнозы для каждого выхода:

```
y_pred_main, y_pred_aux = model.predict([X_new_A, X_new_B])
```

Как видите, с помощью API-интерфейса `Functional` вы довольно легко можете построить архитектуру любого желаемого вида. Давайте взглянем на последний способ построения моделей Keras.

## Использование API-интерфейса Subclassing для построения динамических моделей

Оба API-интерфейса, `Sequential` и `Functional`, являются декларативными: вы начинаете с объявления того, какие слои хотите применять, и как они должны быть связаны, и только затем можете заняться подачей в модель данных для обучения или вывода. Такой подход обладает многими преимуществами: модель легко сохранять, клонировать и совместно использовать; ее структуру можно отображать и анализировать; фреймворк способен выводить формы и проверять типы, поэтому ошибки могут выявляться на ранней стадии (т.е. еще до прохождения любых данных через модель). Кроме того, облегчается отладка, т.к. целая модель представляет собой статический граф слоев. Но есть и обратная сторона: модель статична. Некоторые модели задействуют циклы, варьирующиеся формы, условное ветвление и другие линии динамического поведения. Для таких случаев, или если вы просто предпочитаете более императивный стиль программирования, предусмотрен API-интерфейс `Subclassing` (создание подклассов).

Просто определите подкласс класса `Model`, создайте необходимые слои в конструкторе и применяйте их для желаемых вычислений в методе `call()`. Например, создание экземпляра приведенного ниже класса `WideAndDeepModel` дает модель, эквивалентную той, которая была построена с помощью API-интерфейса `Functional`. Затем вы можете ее скомпилировать, оценить и использовать для выработывания прогнозов, как делалось ранее:

```
class WideAndDeepModel(keras.Model):  
    def __init__(self, units=1, activation="relu", **kwargs):  
        super().__init__(**kwargs) # обрабатывает стандартные аргументы  
        # (например, name)
```

```

self.hidden1 = keras.layers.Dense(units, activation=activation)
self.hidden2 = keras.layers.Dense(units, activation=activation)
self.main_output = keras.layers.Dense(1)
self.aux_output = keras.layers.Dense(1)

def call(self, inputs):
    input_A, input_B = inputs
    hidden1 = self.hidden1(input_B)
    hidden2 = self.hidden2(hidden1)
    concat = keras.layers.concatenate([input_A, hidden2])
    main_output = self.main_output(concat)
    aux_output = self.aux_output(hidden2)
    return main_output, aux_output

model = WideAndDeepModel()

```

Пример выглядит очень похожим на реализацию с применением API-интерфейса Functional за исключением того, что нам не нужно создавать входы; мы просто используем аргумент `input` метода `call()` и отделяем создание слоев<sup>21</sup> в конструкторе от их применения в методе `call()`. Крупное отличие состоит в том, что в методе `call()` можно выполнять практически все, что угодно: циклы `for`, операторы `if`, низкоуровневые операции TensorFlow — ограничением может быть разве что ваше воображение (см. главу 12)! Это делает его великолепным API-интерфейсом для исследователей, экспериментирующих с новыми идеями.

За такую дополнительную гибкость приходится платить: архитектура вашей модели скрыта внутри метода `call()`, поэтому библиотеке Keras не легко ее инспектировать, модель невозможно сохранить или клонировать, а вызов метода `summary()` дает лишь список слоев без какой-либо информации о том, как они связаны друг с другом. Более того, Keras не в состоянии заблаговременно проверять типы и формы, поэтому легче допускать ошибки. Таким образом, если только вам действительно не нужна подобная добавочная гибкость, то вероятно имеет смысл придерживаться API-интерфейса `Sequential` или `Functional`.



Модели Keras могут использоваться в точности как обычные слои, так что вы можете легко их комбинировать для построения сложных архитектур.

---

<sup>21</sup> Модели Keras имеют атрибут `output`, так что мы не можем использовать это имя для главного выходного слоя и потому назвали его `main_output`.

Теперь, когда вам уже известно, как строить и обучать нейронные сети с применением Keras, вы наверняка захотите их сохранить!

## Сохранение и восстановление модели

Когда используется API-интерфейс Sequential или Functional, сохранять обученную модель Keras столь же просто, как и ее получать:

```
model = keras.layers.Sequential([...])      # или keras.Model([...])
model.compile([...])
model.fit([...])
model.save("my_keras_model.h5")
```

Библиотека Keras будет применять формат HDF5 для сохранения архитектуры модели (включая гиперпараметры каждого слоя) и значений всех параметров модели для каждого слоя (например, веса и смещения связей). Keras также сохраняет оптимизатор (в том числе его гиперпараметры и любое состояние, которое он может иметь).

Обычно вы будете иметь сценарий, который обучает и сохраняет модель, а также один или большее число сценариев (либо веб-служб), загружающих модель и использующих ее для вырабатывания прогнозов. Загружать модель тоже легко:

```
model = keras.models.load_model("my_keras_model.h5")
```



Прием будет работать в случае применения API-интерфейса Sequential или Functional, но, к сожалению, не в ситуации, когда используется создание подклассов моделей. Вы можете применять методы `save_weights()` и `load_weights()` для сохранения и восстановления хотя бы параметров моделей, но все остальное вам придется сохранять и восстанавливать самостоятельно.

Но что, если обучение длится несколько часов? Так происходит довольно часто, особенно при обучении на крупных наборах данных. В этом случае вы должны сохранять не только модель в конце обучения, но также контрольные точки через регулярные интервалы во время процесса обучения, чтобы избежать утраты чего-либо, если возникнет аварийный отказ компьютера. А каким образом можно сообщить методу `fit()` о необходимости сохранения контрольных точек? Используйте обратные вызовы (*callback*).

## Использование обратных вызовов

Метод `fit()` принимает аргумент `callbacks`, позволяющий указывать список объектов, к которым Keras будет обращаться в начале и конце обучения, в начале и конце каждой эпохи и даже до и после обработки каждого пакета. Например, обратный вызов `ModelCheckpoint` сохраняет контрольные точки вашей модели через регулярные интервалы в течение процесса обучения — по умолчанию в конце каждой эпохи:

```
[...] # построить и скомпилировать модель  
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5")  
history = model.fit(X_train, y_train, epochs=10,  
 callbacks=[checkpoint_cb])
```

Кроме того, если во время обучения вы применяете проверочный набор, то при создании `ModelCheckpoint` можете установить `save_best_only=True`. В таком случае модель будет сохраняться, только когда ее эффективность на проверочном наборе оказывается наилучшей до сих пор. Таким образом, вам не придется беспокоиться о том, что обучение выполняется слишком долго и происходит переобучение обучающим набором: просто восстановите последнюю модель, сохраненную после обучения, и она будет наилучшей моделью на проверочном наборе. В следующем коде демонстрируется простой способ реализации раннего прекращения (представленного в главе 4):

```
checkpoint_cb = keras.callbacks.ModelCheckpoint("my_keras_model.h5",  
                                                 save_best_only=True)  
history = model.fit(X_train, y_train, epochs=10,  
                      validation_data=(X_valid, y_valid),  
                      callbacks=[checkpoint_cb])  
model = keras.models.load_model("my_keras_model.h5")  
# откат к наилучшей модели
```

Еще один способ реализации раннего прекращения предусматривает использование обратного вызова `EarlyStopping`. Он прервет обучение, когда измерения покажут отсутствие прогресса на проверочном наборе для заданного количества эпох (определенного аргументом `patience`), и неизбежно произведет откат к наилучшей модели. Вы можете скомбинировать оба обратных вызова, чтобы сохранять контрольные точки своей модели (на случай аварийного отказа компьютера) и заблаговременно прерывать обучение, когда дополнительный прогресс отсутствует (во избежание излишней траты времени и ресурсов):

```
early_stopping_cb = keras.callbacks.EarlyStopping(patience=10,
                                                 restore_best_weights=True)
history = model.fit(X_train, y_train, epochs=100,
                     validation_data=(X_valid, y_valid),
                     callbacks=[checkpoint_cb, early_stopping_cb])
```

Количество эпох может быть установлено большим, т.к. обучение автоматически остановится при отсутствии дальнейшего прогресса. В таком случае нет необходимости восстанавливать сохраненную наилучшую модель, поскольку обратный вызов EarlyStopping будет отслеживать наилучшие веса и восстанавливать их в конце обучения.



Множество других обратных вызовов доступно в пакете `keras.callbacks` (<https://keras.io/callbacks/>).

Если вам необходим добавочный контроль, тогда вы легко можете написать собственные специальные обратные вызовы. В качестве примера приведенный далее специальный обратный вызов будет во время обучения отображать соотношение между потерей при проверке и потерей при обучении (скажем, для обнаружения переобучения):

```
class MyCallback(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        print("Val / loss: {:.2f} / {:.2f}.".format(logs["val_loss"] / logs["loss"]))
```

Как и следовало ожидать, вы можете реализовать методы `on_train_begin()`, `on_train_end()`, `on_epoch_begin()`, `on_epoch_end()`, `on_batch_begin()` и `on_batch_end()`. Обратные вызовы могут применяться также во время оценки и прогнозирования, если они когда-либо вам потребуются (например, в целях отладки). Для оценки вы должны реализовать методы `on_test_begin()`, `on_test_end()`, `on_test_batch_begin()` или `on_test_batch_end()` (вызываются методом `evaluate()`), а для прогнозирования — методы `on_predict_begin()`, `on_predict_end()`, `on_predict_batch_begin()` или `on_predict_batch_end()` (вызываются методом `predict()`).

Теперь давайте взглянем на еще один инструмент, который вы определенно обязаны иметь в своем комплекте инструментов при использовании `tf.keras: TensorBoard`.

## Использование TensorBoard для визуализации

TensorBoard представляет собой замечательный интерактивный инструмент визуализации, который можно применять для отображения кривых обучения во время процесса обучения, сравнения кривых обучения между множеством запусков, визуализации вычислительного графа, анализа статистических данных по обучению, просмотра изображений, генерируемых моделью, визуализации сложных многомерных данных, спроектированных на три измерения и автоматически кластеризованных, а также многое другое! Инструмент TensorBoard устанавливается автоматически при установке TensorFlow, поэтому он у вас уже есть.

Для его использования вы должны модифицировать свою программу, чтобы она выводила данные, которые желательно визуализировать, в специальные двоичные журнальные файлы, называемые *файлами событий* (*event file*). Каждая двоичная запись данных называется *сводкой* (*summary*). Сервер TensorBoard будет отслеживать журналный каталог и автоматически подхватывать изменения для обновления визуализаций: это позволяет вам визуализировать актуальные данные (с короткой задержкой), такие как кривые обучения во время процесса обучения. Как правило, вы хотите указывать серверу TensorBoard корневой журналный каталог и так конфигурировать программу, чтобы при каждом запуске она записывала в другой подкаталог. Таким образом, тот же самый экземпляр сервера TensorBoard позволит визуализировать и сравнивать данные из множества запусков вашей программы, ничего не путая.

Давайте начнем с определения корневого журнального каталога, предназначенного для записи журналов TensorBoard, а также небольшой функции, которая будет генерировать путь к подкаталогу на основе текущей даты и времени, чтобы он отличался при каждом запуске. Вы можете пожелать включить в имя журнального каталога дополнительную информацию, такую как проверяемые значения гиперпараметров, облегчив понимание того, что просматривается в TensorBoard:

```
import
root_logdir = os.path.join(os.curdir, "my_log")
def get_run_logdir():
    import
    run_id = time.strftime("run_%Y-%m-%d-%H-%M-%S")
    return os.path.join(root_logdir, run_id)
run_logdir = get_run_logdir() # например,
                            # '/my_logs/run_2019_06_07-15_15_22'
```

Хорошая новость в том, что Keras предоставляет подходящий обратный вызов `TensorBoard()`:

```
[...] # построить и скомпилировать модель
tensorboard_cb = keras.callbacks.TensorBoard(run_logdir)
history = model.fit(X_train, y_train, epochs=30,
                     validation_data=(X_valid, y_valid),
                     callbacks=[tensorboard_cb])
```

И это все, что нужно было сделать! Вряд ли что-то будет проще в применении. После запуска приведенного кода обратный вызов `TensorBoard()` позаботится о создании журнального каталога (при необходимости наряду с его родительскими каталогами) и во время процесса обучения он будет создавать файлы событий и записывать в них сводки. Запустив программу во второй раз (возможно с изменением значения какого-то гиперпараметра), вы получите структуру каталогов следующего вида:

```
my_logs/
└── run_2019_06_07-15_15_22
    ├── train
    │   ├── events.out.tfevents.1559891732.mycomputer.local.38511.694049.v2
    │   ├── events.out.tfevents.1559891732.mycomputer.local.profile-empty
    │   ├── plugins/profile/2019-06-07_15-15-32
    │   └── local.trace
    └── validation
        └── events.out.tfevents.1559891733.mycomputer.local.38511.696430.v2
└── run_2019_06_07-15_15_49
[...]
```

Для каждого запуска предусмотрен один каталог, который содержит один подкаталог для журналов обучения и один для журналов проверки. Оба каталога содержат файлы событий, но журналы обучения также включают профилирующие трассировки: это позволяет `TensorBoard` точно показать, сколько времени модель потратила на каждую свою часть на всех устройствах, что отлично подходит для локализации узких мест в плане производительности.

Далее вам понадобится запустить сервер `TensorBoard`. Один из способов предусматривает ввод команды в окне терминала. Если вы установили `TensorFlow` внутри среды `virtualenv`, тогда должны активизировать ее. Затем введите показанную ниже команду в корневом каталоге проекта (или откуда угодно при условии, что указан надлежащий журнальный каталог):

```
$ tensorboard --logdir=./my_logs --port=6006
TensorBoard 2.0.0 at http://mycomputer.local:6006/
(Press CTRL+C to quit)
```

Если ваша командная оболочка не может найти сценарий `tensorboard`, тогда вы должны обновить переменную среды `PATH`, чтобы она содержала каталог, куда был установлен упомянутый сценарий (или же вы можете просто заменить `tensorboard` в командной строке на `python3 -m tensorboard.main`). После запуска сервера можете открыть окно веб-браузера и перейти по ссылке `http://localhost:6006`.

В качестве альтернативы вы можете использовать `TensorBoard` прямо внутри `Jupyter`, выполнив следующие команды. Первая строка загружает расширение `TensorBoard`, а вторая запускает сервер `TensorBoard` на порте 6006 (если он еще не запущен) и подключается к нему:

```
%load_ext tensorboard  
%tensorboard --logdir=./my_logs --port=6006
```

В любом случае вы должны увидеть веб-интерфейс `TensorBoard`. Щелкните на вкладке `SCALARS` (`Скаляры`), чтобы просмотреть кривые обучения (рис. 10.17). Выберите слева внизу журналы, которые хотите визуализировать (например, журналы обучения из первого и второго запуска), и щелкните на скаляре `epoch_loss`. Обратите внимание на то, что потеря при обучении хорошо снижалась во время обоих запусков, но во втором запуске снижение проходило намного быстрее. На самом деле мы применяли скорость обучения 0.05 (`optimizer=keras.optimizers.SGD(lr=0.05)`) вместо 0.001.



Рис. 10.17. Визуализация кривых обучения с помощью `TensorBoard`

В добавок вы можете визуализировать полный граф, выясненные веса (спроектированные в три измерения) или профилирующие трассировки. Обратный вызов `TensorBoard()` также имеет параметры для записи в журнал дополнительных данных, таких как вложения (см. главу 13).

Кроме того, TensorFlow предлагает низкоуровневый API-интерфейс в пакете `tf.summary`. Приведенный далее код создает экземпляр `SummaryWriter` посредством функции `create_file_writer()` и использует его как контекст для записи в журнал скаляров, гистограмм, изображений, аудиоданных и текста, которые затем могут быть визуализированы с применением `TensorBoard` (обязательно опробуйте самостоятельно):

```
test_logdir = get_run_logdir()
writer = tf.summary.create_file_writer(test_logdir)
with writer.as_default():
    for step in range(1, 1000 + 1):
        tf.summary.scalar("my_scalar", np.sin(step / 10), step=step)
        data = (np.random.randn(100) + 2) * step / 100
                    # случайные данные
        tf.summary.histogram("my_hist", data, buckets=50, step=step)
        images = np.random.rand(2, 32, 32, 3)
                    # случайные изображения RGB размером 32×32
        tf.summary.image("my_images", images * step / 1000, step=step)
        texts = ["The step is " + str(step), "Its square is " + str(step**2)]
        tf.summary.text("my_text", texts, step=step)
        sine_wave = tf.math.sin(tf.range(12000) / 48000 * 2 * np.pi * step)
        audio = tf.reshape(tf.cast(sine_wave, tf.float32), [1, -1, 1])
        tf.summary.audio("my_audio", audio, sample_rate=48000, step=step)
```

В действительности это полезный инструмент визуализации даже за рамками библиотеки TensorFlow или глубокого обучения.

Давайте подведем итоги того, что вы узнали до сих пор в главе: откуда произошли нейронные сети, что такое многослойный персепtron и как его использовать для классификации и регрессии, каким образом применять API-интерфейс `Sequential` из `tf.keras` для построения многослойных персептронов и как использовать API-интерфейс `Functional` или `Subclassing` для построения моделей со сложными архитектурами. Вы научились сохранять и восстанавливать модель и применять обратные вызовы для организации контрольных точек, раннего прекращения и т.д. Наконец, вы узнали, как использовать `TensorBoard` для визуализации. Вы уже можете применять нейронные сети для решения многих задач! Однако вас может интересовать, каким образом выбирать количество скрытых слоев, число нейронов в сети и все остальные гиперпараметры. Выясним перечисленные вопросы.

# 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 вообразимую архитектуру, 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?

Один из вариантов предусматривает просто опробование многих комбинаций гиперпараметров и выяснение, какая из них обеспечивает наилучшую работу на проверочном наборе (или применение перекрестной проверки по K блокам). Например, мы можем использовать GridSearchCV или RandomizedSearchCV для исследования пространства гиперпараметров, как поступали в главе 2. Для этого необходимо поместить наши модели Keras внутрь объектов, которые имитируют обычные регрессоры Scikit-Learn.

На первом шаге создается функция, которая будет строить и компилировать модель Keras с заданным набором гиперпараметров:

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=0.03,
                input_shape=[8]):
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu"))
    model.add(keras.layers.Dense(1))
    optimizer = keras.optimizers.SGD(lr=learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

Функция `build_model()` создает простую модель `Sequential` для одномерной регрессии (только один выходной нейрон) с заданной формой входа и количеством скрытых слоев и нейронов, после чего компилирует ее с применением оптимизатора SGD, сконфигурированного с указанной скоростью обучения. Хорошей практикой является предоставление разумных стандартных значений для максимально возможного числа гиперпараметров, как поступает Scikit-Learn.

Далее создадим объект `KerasRegressor` на основе функции `build_model()`:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

Объект `KerasRegressor` — это тонкая оболочка вокруг модели Keras, построенной с использованием `build_model()`. Поскольку при его создании мы не указываем ни одного гиперпараметра, он будет применять стандартные гиперпараметры, которые были определены в `build_model()`. Теперь мы можем использовать объект `KerasRegressor` подобно обычному регрессору Scikit-Learn: обучить его с применением метода `fit()`, оценить с помощью метода `score()` и поручить ему вырабатывание прогнозов посредством метода `predict()`, как показано в следующем коде:

```
keras_reg.fit(X_train, y_train, epochs=100,
               validation_data=(X_valid, y_valid),
               callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

Обратите внимание, что любой дополнительный параметр, передаваемый методу `fit()`, будет передаваться лежащей в основе модели Keras. Также имейте в виду, что оценка будет противоположной MSE, т.к. Scikit-Learn нужны оценки, а не потери (т.е. выше должно быть лучше).

Мы не хотим обучать и оценивать единственную модель вроде этой, а обучить сотни вариантов и посмотреть, какой из них работает лучше на проверочном наборе. Поскольку гиперпараметров много, вместо решетчатого поиска предпочтительнее использовать рандомизированный поиск (как обсуждалось в главе 2). Давайте попробуем выяснить количество скрытых слоев, число нейронов и скорость обучения:

```
from           import reciprocal
from           import RandomizedSearchCV

param_distrib = [
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
]

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distrib,
                                    n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                   validation_data=(X_valid, y_valid),
                   callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

Прием идентичен тому, что мы делали в главе 2, но здесь мы передаем методу `fit()` дополнительные параметры, которые отправляются лежащим в основе моделям Keras. Обратите внимание на то, что объект `RandomizedSearchCV` применяет перекрестную проверку по К блокам, поэтому он не использует `X_valid` и `y_valid`, которые необходимы только для раннего прекращения.

Выяснение может занять много часов в зависимости от оборудования, размеров набора данных, сложности модели и значений `n_iter` и `cv`. По окончании вы можете получить доступ к найденным наилучшим значениям параметров, лучшей оценке и обученной модели Keras:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

Теперь вы можете сохранить модель, оценить ее на испытательном наборе и в случае, если вас устраивает эффективность, развернуть модель в производственной среде. Применять рандомизированный поиск не слишком сложно, и он хорошо работает для многих довольно простых задач. Тем не менее, когда обучение оказывается медленным (скажем, для более сложных задач с более крупными наборами данных), такой подход позволит исследовать лишь крошечную порцию пространства гиперпараметров. Вы можете частично смягчить проблему, вручную помогая процессу поиска: сначала запустите быстрый случайный поиск, используя широкие диапазоны значений гиперпараметров, затем выполните еще один поиск с применением меньших диапазонов значений, центрированных по лучшим значениям, которые были найдены при первом поиске, и т.д. Надо надеяться, что такой подход раскроет хороший набор гиперпараметров. Однако он отнимает слишком много времени, и вероятно не будет стоить потраченных усилий.

К счастью, есть много методик исследования пространства поиска гораздо эффективнее, чем случайным образом. Их основная идея проста: когда область пространства оказывается хорошей, она должна исследоваться дальше. Такие методики позаботятся о процессе “раскрытия” и приведут к намного лучшим решениям за значительно меньшее время. Ниже перечислены библиотеки Python, которые вы можете использовать для оптимизации гиперпараметров.

## *Hyperopt (<https://github.com/hyperopt/hyperopt>)*

Популярная библиотека для оптимизации всех видов сложных пространств поиска (включая вещественные значения, такие как скорость обучения, и дискретные значения, подобные количеству слоев).

## *Hyperas (<https://github.com/maxpumperla/hyperas>),*

*kopt (<https://github.com/Avsecz/kopt>)*

*или Talos (<https://github.com/autonomio/talos>)*

Полезные библиотеки оптимизации гиперпараметров для моделей Keras (первые две основаны на Hyperopt).

## *Keras Tuner (<https://homl.info/kerastuner>)*

Легкая в применении библиотека оптимизации гиперпараметров для моделей Keras производства Google с размещенной службой для визуализации и анализа.

## *Scikit-Optimize (модуль skopt)*

*(<https://scikit-optimize.github.io/>)*

Универсальная библиотека оптимизации. Класс BayesSearchCV выполняет байесовскую оптимизацию с использованием интерфейса, похожего на GridSearchCV.

## *Spearmint (<https://github.com/JasperSnoek/spearmint>)*

Библиотека байесовской оптимизации.

## *Hyperband (<https://github.com/zygmuntz/hyperband>)*

Библиотека быстрой подстройки гиперпараметров, основанная на недавней статье Лиши Ли и др. о Hyperband (<https://homl.info/hyperband>)<sup>22</sup>.

## *Sklearn-Deap (<https://github.com/rsteca/sklearn-deap>)*

Библиотека оптимизации гиперпараметров, основанная на эволюционных алгоритмах, с интерфейсом, подобным GridSearchCV.

Кроме того, многие компании предлагают службы для оптимизации гиперпараметров. В главе 19 мы обсудим службу подстройки гиперпараметров.

<sup>22</sup> Лиша Ли и др., *Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization* (Hyperband: новый основанный на бандите подход к оптимизации гиперпараметров), *Journal of Machine Learning Research* 18 (апрель 2018 г.): с. 1–52.

ров платформы AI Platform инфраструктуры Google Cloud (<https://homl.info/gogletuning>). Другие варианты включают службы от Arimo (<https://arimo.com/>) и SigOpt (<https://sigopt.com/>), а также Oscar от CallDesk (<http://oscar.calldesk.ai/>).

Подстройка гиперпараметров все еще является областью активных исследований, и эволюционные алгоритмы возвращаются. Например, ознакомьтесь с великолепной статьей 2017 года от DeepMind (<https://homl.info/pbt>)<sup>23</sup>, в которой авторы совместно оптимизируют совокупность моделей и их гиперпараметры. Компания Google также применяет эволюционный подход не только при поиске гиперпараметров, но и при нахождении наилучшей архитектуры нейронной сети для решения задачи; комплект AutoML от Google уже доступен в виде облачной службы (<https://cloud.google.com/automl/>). Возможно, дни построения нейронных сетей вручную скоро канут в прошлое? Почитайте запись в блоге Google AI Blog (<https://homl.info/automlpost>), посвященную этой теме. В действительности эволюционные алгоритмы успешно использовались для обучения индивидуальных нейронных сетей, заменяя вездесущий градиентный спуск! Примеры приводятся в статье 2017 года (<https://homl.info/neuroevol>) от Uber, где авторы представляют свою методику глубокой нейроэволюции (*Deep Neuroevolution*).

Несмотря на весь этот захватывающий прогресс и все упомянутые инструменты и службы, наличие представления о том, какие значения будут подходящими для каждого гиперпараметра, по-прежнему помогает, а потому вы можете построить быстрый прототип и ограничить пространство поиска. В последующих разделах предлагаются руководящие указания по выбору количества скрытых слоев и нейронов в многослойном персептроне и по подбору хороших значений для ряда главных гиперпараметров.

## Количество скрытых слоев

При решении многих задач вы можете начать с единственного скрытого слоя и получить приемлемые результаты. Многослойный персептрон с только одним скрытым слоем теоретически способен моделировать даже самые сложные функции при условии, что он имеет достаточное число нейронов. В течение долгого времени такое положение дел убеждало ис-

<sup>23</sup> Макс Джадерберг и др., *Population Based Training of Neural Networks* (Обучение нейронных сетей на основе совокупностей), препринт arXiv:1711.09846 (2017 г.).

следователей, что нет никакой нужды изучать более глубокие нейронные сети. Но для сложных задач глубокие сети обладают гораздо более высокой эффективностью параметров, нежели сети с меньшей глубиной: они могут моделировать сложные функции с применением экспоненциально меньшего количества нейронов, чем неглубокие сети, позволяя им достичь намного большей эффективности при том же объеме обучающих данных.

Чтобы понять причину, представьте, что вам предложено нарисовать лес с применением какого-то программного обеспечения для рисования, но использовать копирование и вставку чего-либо запрещено. Вам придется рисовать каждое дерево по отдельности, ветвь за ветвью, листик за листиком. Если бы взамен вы могли нарисовать один листик, скопировать и вставить его для рисования ветви, затем скопировать и вставить готовую ветвь для создания дерева и в заключение скопировать и вставить полученное дерево для образования леса, то закончили бы работу в кратчайший срок. Реальные данные часто структурированы в соответствии с иерархией подобного рода, и глубокие нейронные сети автоматически извлекают преимущество из этого факта: самые нижние скрытые слои моделируют низкоуровневые структуры (например, линейные сегменты разнообразных форм и ориентаций), промежуточные скрытые слои объединяют такие низкоуровневые структуры для моделирования структур промежуточного уровня (скажем, квадратов и окружностей), а высокоуровневые скрытые слои и выходной слой объединяют промежуточные структуры с целью моделирования высокоуровневых структур (например, лиц).

Такая иерархическая архитектура не только помогает сетям DNN быстрее сходиться в хорошее решение, но также улучшает их способность к обобщению на новые наборы данных. Скажем, если модель уже обучена распознавать лица на фотографиях и теперь необходимо обучить новую нейронную сеть распознаванию причесок, тогда вы можете ускорить обучение, повторно задействовав самые нижние слои первой сети. Вместо инициализации случайнм образом весов и смещений первых нескольких слоев в новой нейронной сети вы можете инициализировать их значениями весов и смещений самых нижних слоев первой сети. В этом случае сети не придется обучаться с нуля всем низкоуровневым структурам, которые встречаются на большинстве фотографий; она должна будет узнать только структуры более высоких уровней (например, прически). Прием называется *обучением передачей знаний* (*Transfer learning*).

Итак, для многих задач вы можете начать с одного или двух скрытых слоев, и нейронная сеть будет нормально работать. Например, на наборе данных MNIST можно 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 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 14), и они нуждаются в обучающих данных огромного объема. Обучать такие сети с нуля придется редко: гораздо чаще будут повторно использоваться части заранее обученной современной сети, которая выполняет похожую задачу. Тогда обучение пройдет намного быстрее и потребует меньшего объема данных (мы обсудим это в главе 11).

## Количество нейронов на скрытый слой

Количество нейронов внутри входного и выходного слоев определяется требуемым для задачи типом входа и выхода. Скажем, задача с набором данных MNIST требует  $28 \times 28 = 784$  входных нейрона и 10 выходных нейронов.

Устоявшаяся практика в отношении скрытых слоев предусматривает установление их размеров для образования пирамиды с все меньшим и меньшим числом нейронов на каждом слое — логическое обоснование того, что многие низкоуровневые признаки могут объединяться в намного меньшее количество высокоуровневых признаков. Например, типовая нейронная сеть для MNIST могла бы иметь 3 скрытых слоя, где первый слой содержит 300, второй — 200 и третий — 100 нейронов. Тем не менее, от такой практики почти полностью отказались, потому что применение одинакового количества нейронов во всех скрытых слоях, кажется, в большинстве случаев работает хорошо или даже лучше; плюс есть только один гиперпараметр, подлежащий настройке, а не по одному на слой. Тем не менее, в зависимости от набора данных иногда может помочь, если первый скрытый слой будет больше остальных.

Как и количество слоев, вы можете попробовать постепенно увеличивать число нейронов до тех пор, пока не начнется переобучение сети. Но на практике часто проще и эффективнее выбирать модель с большим числом слоев и нейронов, чем фактически необходимо, после чего использовать раннее

прекращение и другие методики регуляризации, чтобы предотвратить ее переобучение. Винсент Ванхаук, научный сотрудник из Google, скопировал это с подхода “растягивающихся брюк”: вместо того, чтобы тратить время на поиск брюк, идеально подходящих вам по размеру, просто возьмите растягивающиеся брюки, которые будут ужиматься до правильного размера. Благодаря такому подходу вы избегаете критических слоев, которые могли бы разрушить вашу модель. С другой стороны, если слой имеет слишком мало нейронов, у него не будет достаточной представительской способности для предохранения всей полезной информации из входов (скажем, слой с двумя нейронами может выводить только двумерные данные, так что если он обрабатывает трехмерные данные, то некоторая информация будет утрачена). Независимо от того, насколько крупным и мощным является остаток сети, утраченная информация никогда не восстановится.



В целом вы получите гораздо большую отдачу, увеличивая количество слоев, а не число нейронов на слой.

## Скорость обучения, размер пакета и другие гиперпараметры

Количество скрытых слоев и нейронов — не единственные гиперпараметры, которые можно регулировать в многослойном персептроне. Ниже перечислены некоторые из самых важных гиперпараметров и приведены советы относительно их установки.

### Скорость обучения

Скорость обучения является, пожалуй, наиболее важным гиперпараметром. В общем случае оптимальная скорость обучения составляет около половины максимальной скорости обучения (т.е. скорости обучения, выше которой алгоритм обучения расходится, как было показано в главе 4). Один из способов нахождения надлежащей скорости обучения предусматривает обучение модели в течение нескольких сотен итераций, начиная с очень низкой скорости обучения (например,  $10^{-5}$ ) и постепенно повышая ее до очень большого значения (скажем, 10). Это делается путем умножения скорости обучения на постоянный множитель на каждой итерации (например, на  $\exp(\log(10^6)/500)$  для прохода от  $10^{-5}$  до 10 за 500 итераций). Вычертив график потери как функции от скорости обучения (с применением логарифмической шкалы для

скорости обучения), вы должны увидеть, что сначала она падает. Но через некоторое время скорость обучения станет слишком большой, поэтому потеря будет быстро расти: оптимальная скорость обучения окажется немного ниже, чем точка, где потеря начинает подниматься (обычно примерно в 10 раз ниже экстремума). Затем вы можете заново инициализировать модель и нормально обучить ее, используя найденную хорошую скорость обучения. Мы рассмотрим методики выбора скорости обучения в главе 11.

## Оптимизатор

Выбор оптимизатора, который лучше простого старого мини-пакетного градиентного спуска (и подстройка его гиперпараметров), также довольно важен. Мы обсудим несколько развитых оптимизаторов в главе 11.

## Размер пакета

Размер пакета может оказывать существенное влияние на эффективность и время обучения модели. Главное преимущество применения больших размеров пакета заключается в том, что аппаратные ускорители вроде графических процессоров способны эффективно обрабатывать такие пакеты (см. главу 19), поэтому алгоритм обучения будет видеть больше образцов в секунду. По указанной причине многие исследователи и специалисты-практики рекомендуют использовать наибольший размер пакета, который способен уместиться в ОЗУ графического процессора. Однако здесь есть одна загвоздка: на практике крупные размеры пакета часто приводят к нестабильности обучения, особенно в его начале, и результирующая модель может не обобщаться настолько же хорошо, как модель, обученная с небольшим размером пакета. В апреле 2018 года Ян Лекун даже написал в Твиттере “Friends don’t let friends use mini-batches larger than 32” (Друзья не разрешают друзьям использовать мини-пакеты крупнее 32), ссылаясь на статью 2018 года (<https://homl.info/smallbatch>)<sup>24</sup> Доминика Мастерса и Карло Луски, в которой был сделан вывод, что применение небольших размеров пакета (от 2 до 32) предпочтительнее, поскольку они приводят к

<sup>24</sup> Доминик Мастерс и Карло Луски, *Revisiting Small Batch Training for Deep Neural Networks* (Снова об обучении с небольшими пакетами для глубоких нейронных сетей), препринт arXiv:1804.07612 (2018 г.).

лучшим моделям за меньшее время обучения. Тем не менее, другие статьи указывают в противоположном направлении; в 2017 году в статьях Элада Хоффера и др. (<https://homl.info/largebatch>)<sup>25</sup> и Прайя Гойяла и др. (<https://homl.info/largebatch2>)<sup>26</sup> было показано, что можно выбирать очень большие размеры пакета (вплоть до 8 192), используя разнообразные методики наподобие разгона скорости обучения (т.е. начать с небольшой скорости обучения и затем повышать ее, как мы увидим в главе 11). Это приводит к очень короткому времени обучения без каких-либо пробелов в обобщении. Таким образом, одна стратегия заключается в том, чтобы попробовать применить крупный размер пакета, используя разгон скорости обучения, и если обучение нестабильно или финальная эффективность не устраивает, тогда попробовать взамен применить небольшой размер пакета.

## Функция активации

Мы обсуждали, как выбирать функцию активации, ранее в этой главе: в общем случае хорошим стандартным вариантом для всех скрытых слоев будет функция активации ReLU. Для выходного слоя выбор на самом деле зависит от решаемой задачи.

## Количество итераций

В большинстве случаев количество итераций обучения фактически не нуждается в подстройке: взамен просто используйте раннее прекращение.



Оптимальная скорость обучения зависит от других гиперпараметров (особенно от размера пакета), так что если вы модифицировали любой гиперпараметр, то не забудьте обновить также и скорость обучения.

<sup>25</sup> Элад Хоффер и др., *Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks* (Дольше обучение, лучше обобщение: устранение пробела в обобщении при обучении с крупными пакетами для нейронных сетей), *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017 г.): с. 1729–1739.

<sup>26</sup> Прайя Гойял и др., *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour* (Точный стохастический градиентный спуск с крупными мини-пакетами: обучение на ImageNet за 1 час), препринт arXiv: 1706.02677 (2017 г.).

Дополнительные передовые приемы подстройки гиперпараметров ищите в великолепной статье Лесли Смита (<https://homl.info/1cycle>)<sup>27</sup>, опубликованной в 2018 году.

На этом мы завершаем введение в искусственные нейронные сети и их реализацию с помощью Keras. В следующих нескольких главах мы обсудим методики обучения очень глубоких сетей. Мы также выясним, как настраивать модели с применением низкоуровневого API-интерфейса TensorFlow, и каким образом загружать и предварительно обрабатывать данные, используя API-интерфейс Data. Вдобавок мы углубимся в другие популярные архитектуры нейронных сетей: сверточные нейронные сети для обработки изображений, рекуррентные нейронные сети для последовательных данных, автокодировщики для обучения представлению и порождающие состязательные сети для моделирования и генерации данных<sup>28</sup>.

## Упражнения

1. Инструмент TensorFlow Playground (<https://playground.tensorflow.org/>) является удобным эмулятором нейронных сетей, построенным командой разработчиков TensorFlow. В этом упражнении вы обучите ряд двоичных классификаторов всего за несколько щелчков, а также настроите архитектуру и гиперпараметры модели для получения некоторого представления о том, как работают нейронные сети и что делают их гиперпараметры. Потратьте некоторое время на изучение следующего.

а) Паттерны, которые узнала нейронная сеть. Попробуйте обучить стандартную нейронную сеть, щелкнув на кнопке Run (Запуск) слева вверху. Обратите внимание, насколько быстро находится хорошее решение для задачи классификации. Нейроны в первом скрытом слое узнали простые паттерны, в то время как нейроны во втором скрытом слое научились комбинировать простые паттерны из первого скрытого слоя в более сложные паттерны. В целом, чем больше имеется слоев, тем более сложными могут быть паттерны.

<sup>27</sup> Лесли Смит, *A Disciplined Approach to Neural Network Hyper-Parameters: Part 1 — Learning Rate, Batch Size, Momentum, and Weight Decay*” (Дисциплинированный подход к гиперпараметрам: часть 1 — скорость обучения, размер пакета, инерция и уменьшение весов), препринт arXiv:1803.09820 (2018 г.).

<sup>28</sup> Несколько дополнительных архитектур искусственных нейронных сетей представлены в приложении Д.

- б) Функции активации. Попробуйте заменить функцию активации `tanh` функцией активации `ReLU` и снова обучите сеть. Обратите внимание, что она находит решение даже быстрее, но на этот раз границы линейны. Причина связана с формой функции `ReLU`.
- в) Риск попадания в локальные минимумы. Модифицируйте архитектуру сети, чтобы она имела только один скрытый слой с тремя нейронами. Многократно обучите ее (чтобы сбросить веса сети, щелкните на кнопке `Reset` (Сброс) рядом с кнопкой `Play`). Обратите внимание, что время обучения сильно варьируется, а иногда даже случается попадание в локальный минимум.
- г) Что происходит, когда нейронные сети слишком малы. Удалите один нейрон, оставив только два. Обратите внимание, что нейронная сеть теперь неспособна отыскать хорошее решение, даже если вы попробуете обучить ее несколько раз. Модель имеет слишком мало параметров и систематически недообучается на обучающем наборе.
- д) Что происходит, когда нейронные сети достаточно велики. Доведите количество нейронов до восьми и обучите сеть несколько раз. Обратите внимание, что теперь обучение оказывается согласованно быстрым и никогда не попадает в локальные минимумы. Это подчеркивает важное открытие в теории нейронных сетей: крупные нейронные сети почти никогда не попадают в локальные минимумы, и даже когда подобное происходит, такие локальные оптимумы почти столь же хороши, как глобальный оптимум. Однако они по-прежнему могут на долгое время застревать на длинных плато.
- е) Риск исчезновения градиентов в глубоких сетях. Выберите спиралевидный (`Spiral`) набор данных (правый нижний набор данных в разделе `DATA` (Данные)) и измените архитектуру сети, чтобы она содержала четыре скрытых слоя с восемью нейронами в каждом. Обратите внимание, что обучение продолжается намного дольше и часто на долгие периоды времени задерживается на плато. Кроме того, нейроны на самых высоких слоях (справа) имеют тенденцию развиваться быстрее, чем нейроны на самых низких слоях (слева). Эту проблему, называемую проблемой “исчезновения градиентов”, можно смягчить с помощью лучшей инициализации весов и других методик, лучших оптимизаторов (таких как `AdaGrad` или `Adam`) либо пакетной нормализации (обсуждаемой в главе 11).

- ж) Двигайтесь дальше. Выделите час или около того на эксперименты с другими параметрами и почувствуйте то, что они делают, чтобы сформировать у себя интуитивное представление о нейронных сетях.
2. 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:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
  3. Почему обычно предпочтительнее применять классификатор на основе логистической регрессии, а не классический персептрон (т.е. единственный слой пороговых логических элементов, обученных с использованием алгоритма обучения персепtronов)? Каким образом вы могли бы подстроить персептрон, чтобы сделать его эквивалентом классификатора на основе логистической регрессии?
  4. Почему логистическая функция активации является ключевым ингредиентом при обучении первых многослойных персепtronов?
  5. Назовите три популярных функции активации. Можете ли вы их представить?
  6. 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  $X$ ?
    - Каковы формы вектора весов  $W_h$  скрытого слоя и его вектора смещений  $b_h$ ?
    - Каковы формы вектора весов  $W_o$  выходного слоя и 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  $b_o$ .
  7. 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 вза-

мен вы захотите взяться за набор данных MNIST, то сколько нейронов потребуется иметь в выходном слое и какую функцию активации применять? Как насчет настройки вашей сети на вырабатывание прогнозов цен на дома (см. главу 2)?

8. Что такое обратное распространение и как оно работает? В чем разница между обратным распространением и автоматическим дифференцированием в обратном режиме?
9. Можете ли вы перечислить все гиперпараметры, которые допускают подстройку в базовом многослойном персептроне? Если многослойный персептрон переобучается обучающими данными, тогда каким образом вы могли бы подстроить эти гиперпараметры, чтобы попытаться устранить проблему?
10. Обучите глубокий многослойный персептрон на наборе данных MNIST (вы можете загрузить его с использованием `keras.datasets.mnist.load_data()`). Посмотрите, можете ли вы получить точность свыше 98%. Попробуйте поискать оптимальную скорость обучения, применив описанный в этой главе подход (т.е. экспоненциально увеличивайте скорость обучения, вычертите график ошибки и найдите точку, где ошибка быстро растет). Попробуйте добавить все украшения — сохраните контрольные точки, используйте ранее прекращение и вычертите кривые обучения с применением TensorBoard.

Решения приведенных упражнений доступны в приложении А.



# Обучение глубоких нейронных сетей

В главе 10 мы представили искусственные нейронные сети и обучили наши первые глубокие нейронные сети. Но они были мелкими сетями, имеющими совсем немного скрытых слоев. Что, если вам нужно заняться сложной задачей, такой как обнаружение сотен типов объектов в изображениях с высоким разрешением? Вам может понадобиться обучить гораздо более глубокую сеть DNN, вероятно с 10 или намного большим числом слоев, каждый из которых содержит сотни нейронов, соединенных сотнями тысяч связей. Обучение такой глубокой сети DNN не будет простой прогулкой по парку. Ниже перечислено несколько проблем, которые могут возникнуть.

- Вы можете столкнуться с коварной проблемой исчезновения градиентов (*vanishing gradients*) или связанной с ней проблемой взрывного роста градиентов (*exploding gradients*). Ситуация такова, что градиенты становятся все меньше и меньше или все больше и больше, когда протекают обратно через сеть DNN во время обучения. Обе проблемы делают самые нижние слои весьма трудными в обучении.
- У вас может быть недостаточно обучающих данных для такой крупной сети либо снабжение их метками может оказаться слишком затратным.
- Обучение может быть крайне медленным.
- Модель с миллионами параметров подвержена высокому риску переобучения обучающим набором, особенно если обучающих образцов недостаточно или они крайне зашумлены.

В настоящей главе мы разберем каждую из упомянутых проблем и покажем методики их решения. Мы начнем с исследования проблем исчезновения и взрывного роста градиентов и ряда самых популярных их решений. Далее мы рассмотрим обучение передачей знаний и предварительное обучение без учителя, которые могут помочь вам справиться со сложными задачами, даже когда имеется мало помеченных данных. Затем мы обсудим разнообразные оптимизаторы, которые могут чрезвычайно ускорить обучение крупных мо-

делей. Наконец, мы обратимся к нескольким популярным методикам 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 и обучение никогда не сходится в хорошее решение. Мы называем это проблемой исчезновения градиентов. В ряде случаев может произойти противоположное: градиенты способны расти все больше и больше до тех пор, пока слои не получат безумно высокие обновления весов и алгоритм расходится. Мы имеем проблему взрывного роста градиентов, которая возникает в рекуррентных нейронных сетях (глава 15). В целом глубокие нейронные сети страдают от нестабильных градиентов; разные слои могут обучаться с широко варьирующимиися скоростями.

Такое неподходящее поведение эмпирически наблюдалось давно, и оно было одной из причин, по которым от нейронных сетей обычно отказывались в начале 2000-х годов. Было непонятно, что служило причиной нестабильности градиентов при обучении сети DNN, но кое-что прояснилось в статье 2010 года (<https://homl.info/47>), написанной Ксавье Глоро и Йошуа Бенджи<sup>1</sup>. Авторы обнаружили ряд подозреваемых виновников, включая сочетание популярной логистической сигмоидальной функции активации и методики инициализации весов, которая на то время была самой широко распространенной (т.е. нормального распределения со средним 0 и стандартным отклонением 1). Выражаясь кратко, они показали, что с такой функцией активации и схемой инициализации дисперсия выходов каждого

<sup>1</sup> Ксавье Глоро и Йошуа Бенджи, *Understanding the Difficulty of Training Deep Feedforward Neural Networks* (Осмысление сложности обучения глубоких нейронных сетей прямого распространения), *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010 г.): с. 249–256.

слоя намного больше дисперсии его входов. С продвижением вперед по сети дисперсия продолжает увеличиваться после каждого слоя до тех пор, пока функция активации не насыщается в верхних слоях. Насыщение в действительности 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 логистической функции).

Глядя 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.

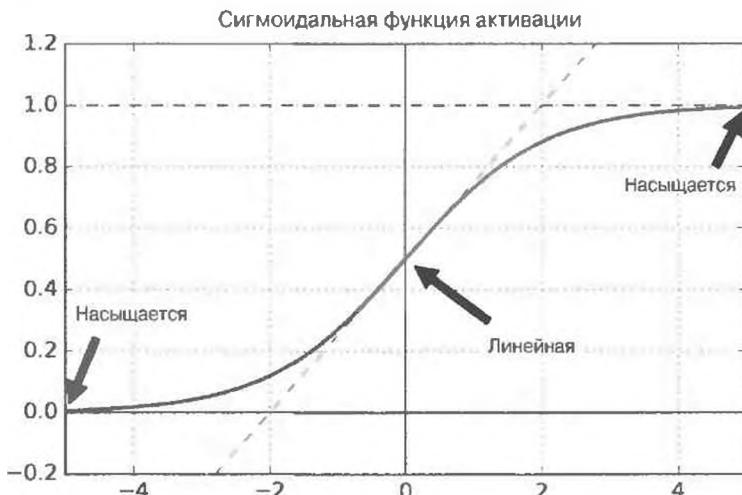


Рис. 11.1. Насыщение логистической функции активации

Таким образом, когда происходит обратное распространение, то у него практически нет градиента для передачи обратно по сети, и существующий небольшой градиент продолжает понижаться с продвижением обратного распространения вниз через верхние слои, поэтому для нижних слоев фактически ничего не остается.

## Инициализация Глоро и Хе

Глоро и Бенджи в своей статье предложили способ значительного смягчения проблемы нестабильных градиентов. Они указали, что нам необходимо, чтобы сигнал должным образом протекал в обоих направлениях: в прямом направлении при вырабатывании прогнозов и в обратном направлении при обратном распространении градиентов. Мы не хотим, чтобы сигнал затухал или взрывообразно возрастал и насыщался. Авторы доказали, что для надле-

жащего протекания сигнала нужно, чтобы дисперсия выходов каждого слоя была равна дисперсии его входов<sup>2</sup>, и необходимо, чтобы градиенты имели равную дисперсию до и после протекания через слой в обратном направлении (если вам интересно, то в статье приведены математические детали). На самом деле гарантировать соблюдение обоих условий невозможно, если только речь не идет о слое с одинаковым количеством входов и нейронов (такие количества называются *коэффициентом разветвления по входу* (*fan-in*) и *коэффициентом разветвления по выходу* (*fan-out*) слоя), но Глоро и Бенджи предложили хороший компромисс, который доказал свою эффективность на практике: веса связей каждого слоя должны быть инициализированы случайным образом, как описано в уравнении 11.1, где  $fan_{\text{среднее}} = (fan_{\text{входа}} + fan_{\text{выхода}})/2$ . Такая стратегия инициализации называется *инициализацией Ксавье либо инициализацией Глоро* в честь первого автора статьи.

### Уравнение 11.1. Инициализация Глоро

(при использовании логистической функции активации)

Нормальное распределение со средним 0 и стандартным отклонением

$$\sigma^2 = \frac{1}{fan_{\text{среднее}}}.$$

Или равномерное распределение между  $-r$  и  $+r$ , где  $r = \sqrt{\frac{3}{fan_{\text{среднее}}}}$ .

Поменяв  $fan_{\text{среднее}}$  на  $fan_{\text{входа}}$  в уравнении 11.1, вы получите стратегию инициализации, которую Ян Лекун предложил в 1990-х годах. Он назвал ее *инициализацией Лекуна*. Женевьеве Опп и Клаус-Роберт Мюллер даже рекомендовали ее в своей книге *Neural Networks: Tricks of the Trade* (Нейронные сети: специфические приемы), вышедшей в 1998 году (Springer). Инициализация Лекуна является эквивалентом инициализации Глоро, когда  $fan_{\text{входа}} = fan_{\text{выхода}}$ . Исследователям понадобилось более десятилетия, чтобы осознать важность этого трюка. Применение инициализации Глоро может существенно ускорить обучение, и она является одним из трюков, которые привели к успеху глубокого обучения.

<sup>2</sup> 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.

В нескольких статьях<sup>3</sup> представлены похожие стратегии для разных функций активации. Они отличаются только шкалой дисперсии и тем, используется  $fan_{\text{среднее}}$  или  $fan_{\text{входа}}$ , как показано в табл. 11.1 (для равномерного распределения просто вычислите  $r = \sqrt{3\sigma^2}$ ). Стратегию инициализации (<https://homl.info/48>) для функции активации ReLU (и ее разновидностей, включая активацию ELU, которая вскоре будет описана) временами называют инициализацией  $Xe$  в честь первого автора статьи. Функция активации SELU объясняется позже в главе. Она должна применяться с инициализацией Лекуна (как мы увидим, предпочтительнее с нормальным распределением).

**Таблица 11.1. Параметры инициализации для функций активации разных типов**

Инициализация	Функция активации	$\sigma^2$ (нормальное распределение)
Глоро	Отсутствует, гиперболический тангенс, логистическая, многопеременная логистическая	$1/fan_{\text{среднее}}$
$Xe$	ReLU и разновидности	$2/fan_{\text{выхода}}$
Лекуна	SELU	$1/fan_{\text{входа}}$

По умолчанию Keras использует инициализацию Глоро с равномерным распределением. При создании слоя вы можете изменить ее на инициализацию  $Xe$ , установив `kernel_initializer="he_uniform"` или `kernel_initializer="he_normal"`:

```
keras.layers.Dense(10, activation="relu",
                   kernel_initializer="he_normal")
```

Если вам нужна инициализация  $Xe$  с равномерным распределением, но на основе  $fan_{\text{среднее}}$ , а не  $fan_{\text{входа}}$ , тогда можете применить инициализатор `VarianceScaling`:

```
he_avg_init = keras.initializers.VarianceScaling(scale=2.,
                                                 mode='fan_avg', distribution='uniform')
keras.layers.Dense(10, activation="sigmoid",
                   kernel_initializer=he_avg_init)
```

<sup>3</sup> Например, Кайминг  $Xe$  и др., *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* (Погружение в выпрямители: превышение человеческой эффективности при классификации ImageNet), *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015 г.): с. 1026–1034.

## Ненасыщаемые функции активации

Одна из идей, выдвинутых Глоро и Бенджи в своей статье 2010 года, заключалась в том, что проблемы с нестабильными градиентами 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, главным образом потому, что она не насыщается для положительных значений (и оттого, что ее быстро вычислять).

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”, т.е. перестают выдавать что-либо, отличающееся от 0. В ряде случаев вы можете обнаружить, что половина нейронов вашей сети погибли, особенно если применяется большая скорость обучения.

Нейрон отмирает, когда его веса модифицируются так, что взвешенная сумма входов нейрона оказывается отрицательной для всех образцов в обучающем наборе. Как только это произошло, он просто продолжает выдавать нули и градиентный спуск больше на него не влияет, поскольку градиент функции ReLU равен 0 при отрицательном входе<sup>4</sup>.

Для решения описанной проблемы 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. Гиперпараметр  $\alpha$  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 длительное коматозное состояние, но у них есть шанс со временем очнуться.

<sup>4</sup> Погибший нейрон иногда способен возвратиться к жизни, если только он не является частью первого скрытого слоя: на самом деле градиентный спуск может подстраивать нейроны в слоях ниже таким образом, что взвешенная сумма входов погибшего нейрона снова становится положительной.

В статье 2015 года (<https://homl.info/49>)<sup>5</sup> 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). В статье также провели оценку *рандомизированного ReLU с утечкой* (*randomized leaky ReLU — RReLU*), где  $\alpha$  выбирается случайно в заданном диапазоне во время обучения и фиксируется на среднем значении во время испытаний. RReLU также выполняется достаточно хорошо и, кажется, действует в качестве регуляризатора (снижая риск переобучения обучающим набором). Наконец, в главе оценивался *параметрический ReLU с утечкой* (*parametric leaky ReLU — PReLU*), где  $\alpha$  разрешено учиться во время обучения (вместо того, чтобы быть гиперпараметром,  $\alpha$  становится параметром, который может быть модифицирован обратным распространением как любой другой параметр). Это говорило о сильном превосходстве ReLU с утечкой над просто ReLU на крупных наборах данных с изображениями, но на меньших наборах данных возникал риск переобучения обучающим набором.

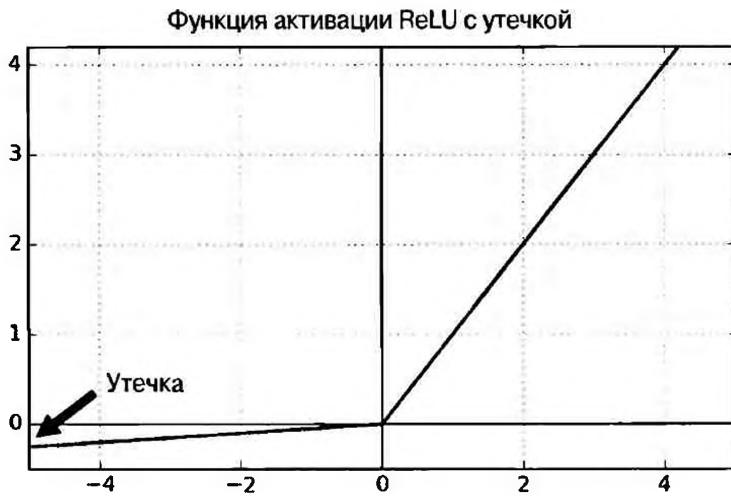


Рис. 11.2. ReLU с утечкой: похоже на ReLU, но с небольшим наклоном для отрицательных значений

<sup>5</sup> Бинг Ксу и др., *Empirical Evaluation of Rectified Activations in Convolution Network* (Эмпирическая оценка выпрямленных активаций в сверточной сети), препринт arXiv:1505.00853 (2015 г.).

Последнее, но не менее важное: в статье, написанной в 2015 году (<https://homl.info/50>) Дьорком-Арне Клевером и др.<sup>6</sup>, 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.

На рис. 11.3 изображен график функции и в уравнении 11.2 приведено ее определение.

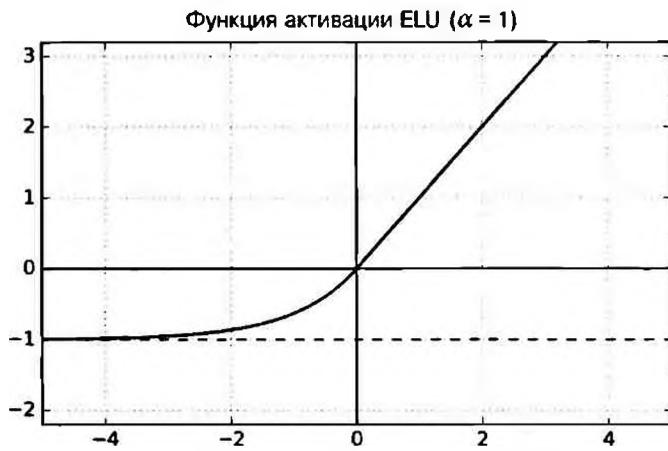


Рис. 11.3. Функция активации ELU

#### 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 некоторыми важными отличиями.

- Она принимает отрицательные значения, когда  $z < 0$ , что позволяет элементу иметь средний выход ближе к 0 и помогает смягчить проблему исчезновения градиентов. Гиперпараметр  $\alpha$  определяет значение, к которому функция ELU приближается, когда  $z$  представляет собой большое отрицательное число. Обычно он устанавливается в 1, но вы можете подстраивать его подобно любому другому гиперпараметру.

<sup>6</sup> Дьорк-Арне Клевер и др., *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)* (Быстрое и точное обучение глубоких сетей с помощью экспоненциальных линейных элементов (ELU)), *Proceedings of the International Conference on Learning Representations* (2016 г.).

- Она имеет ненулевой градиент для  $z < 0$ , что позволяет избежать проблемы погибших нейронов.
- Если гиперпараметр  $\alpha$  равен 1, тогда функция будет гладкой везде, включая точки поблизости  $z = 0$ , что помогает ускорить градиентный спуск, т.к. он не отскакивает настолько сильно влево и вправо от  $z = 0$ .

Главный недостаток функции активации ELU связан с тем, что она вычисляется медленнее, чем функция ReLU и ее разновидности (из-за применения экспоненциальной функции). Ее более высокая скорость схождения компенсирует это медленное вычисление, но все же во время испытаний сеть ELU будет медленнее сети ReLU.

Затем в статье 2017 года (<https://holml.info/selu>)<sup>7</sup> Гюнтер Кламбауэр и др. представили функцию активации SELU (Scaled ELU — масштабированный экспоненциальный линейный элемент): как следует из ее названия, она является масштабированным вариантом функции активации ELU. Авторы показали, что если построить нейронную сеть, состоящую исключительно из стопки плотных слоев, и если все скрытые слои используют функцию активации SELU, тогда сеть станет *самонормализованной*: вывод каждого слоя будет иметь тенденцию сохранять среднее 0 и стандартное отклонение 1 во время обучения, что решает проблему исчезновения/взрывного роста градиентов. В результате функция активации SELU часто значительно превосходит другие функции активации для таких нейронных сетей (в особенности глубоких). Однако есть несколько условий, которые необходимо соблюдать для того, чтобы произошла самонормализация (в статье приведено математическое обоснование).

- Входные признаки должны быть стандартизованы (среднее 0 и стандартное отклонение 1).
- Веса каждого скрытого слоя должны быть инициализированы с помощью нормальной инициализации Лекуна. В Keras это означает установку `kernel_initializer="lecun_normal"`.
- Архитектура сети должна быть последовательной. К сожалению, если вы попытаетесь применить SELU в непоследовательных архитектурах, таких как рекуррентные сети (см. главу 15) или сети с *обходящими связями* (т.е. связями, которые пропускают слои, как в сетях Wide & Deep),

<sup>7</sup> Гюнтер Кламбауэр и др., *Self-Normalizing Neural Networks* (Самонормализация нейронных сетей), *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017 г.): с. 972–981.

то самонормализация не будет гарантироваться, поэтому SELU не обязательно превзойдет другие функции активации.

- В статье самонормализация гарантируется, только если все слои являются плотными, но некоторые исследователи отмечают, что функция активации SELU может увеличить эффективность также сверточных нейронных сетей (см. главу 14).



Итак, какую функцию активации вы должны использовать для скрытых слоев своих глубоких нейронных сетей? Хотя ваши условия будут варьироваться, в целом выбор выглядит так: функция SELU > функция ELU > функция ReLU с утечкой (и ее разновидности) > функция ReLU > функция  $\tanh$  > логистическая функция. Если архитектура сети не допускает самонормализацию, тогда ELU может выполняться лучше, чем SELU (поскольку функция SELU не является гладкой в точке  $z = 0$ ). Если вас сильно заботит задержка во время выполнения, тогда можете отдать предпочтение функции ReLU с утечкой. Если вы не хотите подстраивать очередной гиперпараметр, то можете применять стандартные значения  $\alpha$ , принятые в Keras (например, 0.3 для ReLU с утечкой). При наличии свободного времени и вычислительной мощности вы можете воспользоваться перекрестной проверкой для оценки других функций активации, таких как RReLU, если сеть переобучается, или PReLU, если имеете дело с гигантским обучающим набором. Тем не менее, из-за того, что ReLU представляет собой самую часто применяемую функцию активации (на данный момент), многие библиотеки и аппаратные ускорители обеспечивают специфичную для ReLU оптимизацию; следовательно, если приоритетом считается скорость, тогда ReLU по-прежнему будет наилучшим вариантом.

Для использования функции активации ReLU с утечкой создайте слой LeakyReLU и добавьте его к модели сразу после слоя, к которому хотите ее применить:

```
model = keras.models.Sequential([
    [...]
    keras.layers.Dense(10, kernel_initializer="he_normal"),
    keras.layers.LeakyReLU(alpha=0.2),
    [...]
])
```

Чтобы использовать PReLU, вместо LeakyRelu(alpha=0.2) укажите PReLU(). В текущий момент библиотека Keras не располагает официальной реализацией RReLU, но вы можете довольно легко реализовать ее самостоятельно (чтобы узнать, каким образом, обратитесь к упражнениям в конце главы 12).

Для применения функции активации SELU при создании слоя установите activation="selu" и kernel\_initializer="lecun\_normal":

```
layer = keras.layers.Dense(10, activation="selu",
                           kernel_initializer="lecun_normal")
```

## 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 статье 2015 года (<https://homl.info/51>)<sup>8</sup> Сергей Иоффе и Кристиан Сегеди предложили методику, называемую *пакетной нормализацией* (*Batch Normalization — BN*), которая решает проблемы исчезновения/взрывного роста градиентов. Методика предусматривает добавление в модель операции прямо перед или сразу после функции активации каждого скрытого слоя. Такая операция просто центрирует относительно нуля и нормализует каждый вход, а затем масштабирует и сдвигает результат с применением двух новых векторов параметров на слой: один для масштабирования и еще один для сдвига. Другими словами, операция позволяет модели узнать оптимальный масштаб и среднее каждого входа слоя. Во многих случаях, если вы добавляете слой BN в качестве самого первого слоя нейронной сети, то не нуждаетесь в стандартизации своего обучающего набора (например, используя StandardScaler); слой BN будет делать это автоматически (хорошо, приблизительно, т.к. он просматривает только один пакет за раз, а также может заново масштабировать и сдвигать каждый входной признак).

Чтобы центрировать относительно нуля и нормализовать входы, алгоритму необходимо произвести оценку средней величины и стандартного откло-

<sup>8</sup> Сергей Иоффе и Кристиан Сегеди, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* (Пакетная нормализация: ускорение обучения глубоких сетей путем сокращения внутреннего ковариационного сдвига), *Proceedings of the 32nd International Conference on Machine Learning* (2015 г.): с. 448–456.

нения каждого входа. Он делает это путем вычисления средней величины и стандартного отклонения входа по текущему мини-пакету (отсюда и название “пакетная нормализация”). Полная операция пошагово представлена в уравнении 11.3.

### Уравнение 11.3. Алгоритм пакетной нормализации

$$\begin{aligned}
 1. \quad \mu_B &= \frac{1}{m_B} \sum_{i=1}^{m_B} x^{(i)} \\
 2. \quad \sigma_B^2 &= \frac{1}{m_B} \sum_{i=1}^{m_B} (x^{(i)} - \mu_B)^2 \\
 3. \quad \hat{x}^{(i)} &= \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \\
 4. \quad z^{(i)} &= \gamma \otimes \hat{x}^{(i)} + \beta
 \end{aligned}$$

В данном алгоритме:

- $\mu_B$  — вектор средних величин входов, вычисленный по всему мини-пакету  $B$  (содержит по одному среднему на вход);
- $\sigma_B$  — вектор стандартных отклонений входов, также вычисленный по всему мини-пакету (содержит по одному стандартному отклонению на вход);
- $m_B$  — количество образцов в мини-пакете;
- $\hat{x}^{(i)}$  — вектор центрированных относительно нуля и нормализованных входов для образца  $i$ ;
- $\gamma$  — вектор параметров выходного масштабирования для слоя (содержит по одному параметру масштабирования на вход);
- $\otimes$  — представляет поэлементное умножение (каждый вход умножается на соответствующий параметр выходного масштабирования);
- $\beta$  — вектор параметров выходного сдвига (смещения) для слоя (содержит по одному параметру смещения на вход); каждый вход смещается на соответствующий параметр сдвига;
- $\epsilon$  — крошечное число, позволяющее избежать деления на ноль (обычно  $10^{-5}$ ); называется *сглаживающим членом (smoothing term)*;
- $z^{(i)}$  — выход операции BN: это масштабированная и сдвинутая версия входов.

Таким образом, во время обучения слой BN стандартизирует свои входы, после чего масштабирует и смещает их. Отлично! А как насчет стадии испытаний? Хорошо, все не так-то просто. На самом деле нам может понадобиться вырабатывать прогнозы для индивидуальных образцов, а не пакетов образцов: в таком случае у нас не будет способа подсчета средней величины и стандартного отклонения. Кроме того, даже если мы располагаем пакетом образцов, он может быть слишком малым или образцы могут не быть независимыми и идентично распределенными, поэтому вычисляемые статистические данные по образцам пакета окажутся ненадежными. Одно из решений могло бы предусматривать ожидание до конца обучения, затем прогон полного обучающего набора через нейронную сеть и подсчет средней величины и стандартного отклонения каждого входа слоя BN. При вырабатывании прогнозов такие “финальные” средние величины и стандартные отклонения входов далее могли бы применяться вместо средних величин и стандартных отклонений входов пакета. Однако большинство реализаций пакетной нормализации оценивают финальные статистические данные во время обучения с использованием скользящего среднего для средних величин и стандартных отклонений входов слоя. Именно это Keras делает автоматически, когда применяется слой `BatchNormalization`. Подведем итоги: на каждом слое, подвергнутом пакетной нормализации, узнаются четыре вектора параметров:  $\gamma$  (вектор параметров выходного масштабирования) и  $\beta$  (вектор параметров выходного сдвига) выясняются через обычное обратное распространение, а  $\mu$  (финальный вектор средних величин входов) и  $\sigma$  (финальный вектор стандартных отклонений входов) оцениваются с использованием экспоненциального скользящего среднего. Обратите внимание, что  $\mu$  и  $\sigma$  оцениваются во время обучения, но применяются только после обучения (для замены средних величин и стандартных отклонений входов пакета в уравнении 11.3).

Иоффе и Сегеди продемонстрировали, что пакетная нормализация значительно усовершенствовала все глубокие нейронные сети, с которыми они экспериментировали, приводя к огромному улучшению в решении задачи классификации ImageNet (*ImageNet* — крупная база данных изображений, систематизированных во множество классов, которая широко используется для оценки систем компьютерного зрения). 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 авторов была возможность вы-

бирать намного более высокие скорости обучения, значительно ускоряющие процесс обучения. В частности, вот что 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: достигаем 4.9%-ной ошибки top-5 при проверке (и 4.8%-ной ошибки при испытаниях), что превышает правильность, показываемую оценщиками-людьми.

Наконец, подобно подарку, не перестающему радовать, пакетная нормализация ведет себя как регуляризатор, сокращая потребность в других приемах регуляризации (вроде отключения (dropout), которое рассматривается далее в главе). Тем не менее, пакетная нормализация привносит в модель некоторую сложность (хотя она способна устраниТЬ необходимость в нормализации входных данных, как мы обсуждали ранее). Кроме того, возникает проблема во время выполнения: нейронная сеть медленнее вырабатывает прогнозы из-за добавочных вычислений, требующихся на каждом слое. К счастью, после обучения слой BN часто можно объединить с предыдущим слоем, избегая тем самым проблемы во время выполнения. Это делается путем обновления весов и смещений предыдущего слоя, так что он напрямую производит выходы с подходящим масштабом и сдвигом. Например, если предыдущий слой вычисляет  $XW + b$ , тогда слой будет вычислять  $\gamma \otimes (XW + b - \mu) / \sigma + \beta$  (игнорируя сглаживающий член  $\epsilon$  в знаменателе). Если мы определим  $W' = \gamma \otimes W / \sigma$  и  $b' = \gamma \otimes (b - \mu) / \sigma + \beta$ , то уравнение упростится до  $XW' + b'$ . Таким образом, заменив веса и смещения предыдущего слоя ( $W$  и  $b$ ) обновленными весами и смещениями ( $W'$  и  $b'$ ), мы можем избавиться от слоя BN (оптимизатор TFLite делает это автоматически; см. главу 19).



Вы можете обнаружить, что обучение проходит весьма медленно, т.к. когда применяется пакетная нормализация, каждая эпоха занимает гораздо больше времени. Обычно это уравновешивается тем фактом, что со слоем BN сходимость намного быстрее, поэтому для достижения той же эффективности потребуется меньше эпох. В общем, *фактическое время*, как правило, будет меньше (оно представляет собой время, измеренное по вашим часам).

## Реализация пакетной нормализации с помощью Keras

Как и большинство вещей, связанных с Keras, реализация пакетной нормализации проста и интуитивно понятна. Просто добавьте слой BatchNormalization до или после функции активации каждого скрытого слоя и дополнительно также добавьте к своей модели слой BN в качестве первого слоя. Например, следующая модель применяет BN после каждого скрытого слоя и как первый слой в модели (после выравнивания входных изображений):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(100, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(10, activation="softmax")
])
```

Вот и все! В таком крошечном примере с двумя скрытыми слоями маловероятно, что пакетная нормализация окажет крупное позитивное влияние, но для более глубоких сетей она может привести к разительным переменам.

Давайте отобразим сводку по модели:

```
>>> model.summary()
Model: "sequential_3"
```

Layer (type)	Output Shape	Param #
flatten_3 (Flatten)	(None, 784)	0
batch_normalization_v2 (BatchNormalization)	(None, 784)	3136
dense_50 (Dense)	(None, 300)	235500
batch_normalization_v2_1 (BatchNormalization)	(None, 300)	1200
dense_51 (Dense)	(None, 100)	30100
batch_normalization_v2_2 (BatchNormalization)	(None, 100)	400
dense_52 (Dense)	(None, 10)	1010

Total params: 271,346

Trainable params: 268,978

Non-trainable params: 2,368

Как видите, каждый слой BN добавляет четыре параметра на каждый вход:  $\gamma$ ,  $\beta$ ,  $\mu$  и  $\sigma$  (скажем, первый слой BN добавляет 3 136 параметров, что представляет собой  $4 \times 784$ ). Последние два параметра,  $\mu$  и  $\sigma$ , являются скользящими средними; обратное распространение на них не влияет и потому в Keras они называются “не поддающимися обучению”<sup>9</sup> (если вы подсчитаете суммарное количество параметров BN,  $3\,136 + 1\,200 + 400$ , и поделите его на 2, то получите 2 368, что представляет собой общее число параметров данной модели, не поддающихся обучению).

Давайте взглянем на параметры первого слоя BN. Два из них обучаемые (обратным распространением) и два — нет:

```
>>> [(var.name, var.trainable) for var in model.layers[1].variables]
[('batch_normalization_v2/gamma:0', True),
 ('batch_normalization_v2/beta:0', True),
 ('batch_normalization_v2/moving_mean:0', False),
 ('batch_normalization_v2/moving_variance:0', False)]
```

Теперь, когда создан слой BN в Keras, он также создает две операции, которые будут вызываться Keras на каждой итерации во время обучения. Эти операции будут обновлять скользящие средние. Поскольку мы используем сервер TensorFlow, операции являются операциями TensorFlow (мы обсудим операции TF в главе 12):

```
>>> model.layers[1].updates
[<tf.Operation 'cond_2/Identity' type=Identity>,
 <tf.Operation 'cond_3/Identity' type=Identity>]
```

Авторы статьи о пакетной нормализации высказались в пользу добавления слоев BN перед функциями активации, а не после них (как мы только что сделали). О данном аспекте ведутся споры, т.к. похоже, что предпочтительный вариант зависит от задачи — вы тоже можете поэкспериментировать с ними, чтобы посмотреть, какой способ лучше работает с вашим набором данных. Для добавления слоев BN перед функциями активации вы должны удалить функции активации из скрытых слоев и добавить их как

<sup>9</sup> Тем не менее, они оцениваются во время обучения на основе обучающих данных, так что вероятно они являются обучаемыми. В контексте библиотеки Keras “не поддающиеся обучению” на самом деле означает “не затрагиваемые обратным распространением”.

отдельные слои после слоев BN. А еще из-за того, что слой пакетной нормализации включает один параметр сдвига на вход, вы можете удалить член смещения из предыдущего слоя (просто передайте `use_bias=False` при его создании):

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.BatchNormalization(),
    keras.layers.Dense(300, kernel_initializer="he_normal",
                      use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(100, kernel_initializer="he_normal",
                      use_bias=False),
    keras.layers.BatchNormalization(),
    keras.layers.Activation("elu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Класс `BatchNormalization` поддерживает порядочное число гиперпараметров, которые можно подстраивать. Обычно хорошо подходят стандартные значения, но иногда может требоваться регулировка `momentum` (момент). Указанный гиперпараметр применяется слоем `BatchNormalization`, когда он обновляет экспоненциальные скользящие средние; имея новое значение  $v$  (т.е. новый вектор средних величин или стандартных отклонений входов, которые вычислены по текущему пакету), слой обновляет скользящее среднее  $\hat{v}$ , используя следующее уравнение:

$$\hat{v} \leftarrow \hat{v} \times \text{момент} + v \times (1 - \text{момент})$$

Хорошее значение момента 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).

Еще одним важным гиперпараметром является `axis`: он определяет ось, которая должна быть нормализована. По умолчанию для него принимается `-1`, означая, что будет нормализоваться последняя ось (с применением средних величин и стандартных отклонений, подсчитанных по остальным осям). Когда входной пакет двумерный (т.е. формой пакета является `[размер пакета, признаки]`), это значит, что каждый входной признак будет нормализован на основе средней величины и стандартного отклонения, вычисленного по всем образцам в пакете. Например, первый слой BN в предыдущем примере кода будет независимо нормализовать (а также масштаби-

ровать и сдвигать) каждый из 784 входных признаков. Если мы переместим первый слой BN в позицию перед слоем `Flatten`, тогда входные пакеты будут трехмерными с формой [размер пакета, высота, ширина]; следовательно, слой BN будет подсчитывать 28 средних величин и 28 стандартных отклонений (одно на столбец пикселей, вычисленное по всем образцам в пакете и по всем строкам в столбце) и нормализовать все пиксели в заданном столбце, используя те же самые среднюю величину и стандартное отклонение. Также будут существовать всего 28 параметров масштабирования и 28 параметров сдвига. Если взамен вы хотите трактовать каждый из 784 пикселей независимо, то должны установить `axis=[1, 2]`.

Обратите внимание, что слой BN не выполняет одинаковые вычисления во время обучения и после обучения: во время обучения он применяет статистические данные по пакету, а после обучения — “финальные” статистические данные (т.е. финальные значения скользящих средних). Давайте посмотрим на исходный код класса `BatchNormalization`, чтобы увидеть, как он это делает:

```
class          (keras.layers.Layer):
    [...]
    def __init__(self, inputs, training=None):
        [...]
```

Метод `call()` выполняет вычисления; как будет показано, он имеет дополнительный аргумент `training`, который по умолчанию устанавливается в `None`, но во время обучения метод `fit()` устанавливает его в `1`. Если вам когда-либо придется создавать специальный слой, который должен вести себя по-разному во время обучения и испытаний, тогда добавьте к методу `call()` аргумент `training` и используйте его в методе для принятия решения, что вычислять<sup>10</sup> (мы обсудим специальные слои в главе 12).

Класс `BatchNormalization` стал до такой степени часто применяемым слоем в глубоких нейронных сетях, что он нередко опускается на диаграммах, т.к. предполагается его наличие после каждого слоя. Но недавно опубликованная статья (<https://homl.info/fixup>)<sup>11</sup> Хонг Йи Чжана и др. может

<sup>10</sup> В API-интерфейсе Keras также определена функция `keras.backend.learning_phase()`, которая должна возвращать `1` во время обучения и `0` в противном случае.

<sup>11</sup> Хонг Йи Чжан и др., *Fixup Initialization: Residual Learning Without Normalization* (Инициализация с фиксированным обновлением: остаточное обучение без нормализации), препринт arXiv: 1901.09321 (2019 г.).

изменить указанное предположение: за счет использования новой методики инициализации весов с фиксированным обновлением (*fixed-update — fixup*) авторам удалось обучить очень глубокую нейронную сеть (10 000 слоев!) без BN, достигнув современного уровня эффективности на сложных задачах классификации изображений. Однако поскольку это новейшее исследование, у вас может возникнуть желание подождать дополнительных исследований, подтверждающих такое открытие, прежде чем отказываться от пакетной нормализации.

## Отсечение градиентов

Популярный прием смягчения проблемы, связанной с взрывным ростом градиентов, предусматривает просто урезание градиентов во время обратного распространения, чтобы они никогда не превышали определенный порог. Прием называется *отсечением градиентов* (*Gradient Clipping*; <https://homl.info/52>)<sup>12</sup> и чаще всего применяется в рекуррентных нейронных сетях, т.к. использовать пакетную нормализацию в них сложно, что будет показано в главе 15. Для остальных типов сетей пакетной нормализации обычно достаточно.

В Keras реализация отсечения градиентов сводится просто к установке аргумента `clipvalue` или `clipnorm` при создании оптимизатора, например:

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
model.compile(loss="mse", optimizer=optimizer)
```

Созданный оптимизатор будет урезать каждый компонент вектора-градиента до величины между -1.0 и 1.0, т.е. все частные производные потери (относительно каждого обучаемого параметра) окажутся в пределах от -1.0 до 1.0. Порог является гиперпараметром, который допускает подстройку. Обратите внимание, что ориентация вектора-градиента может измениться. Скажем, если первоначальным вектором-градиентом был [0.9, 100.0], то он указывает главным образом в направлении второй оси; но после его урезания по величине получается вектор [0.9, 1.0], который указывает приблизительно в направлении диагонали между двумя осями. На практике такой подход работает хорошо. Если вы хотите гарантировать, что отсечение градиентов не изменяет направление вектора-градиента, то должны урезать по

<sup>12</sup> Разван Паскану и др., *On the Difficulty of Training Recurrent Neural Networks* (О трудности обучения рекуррентных нейронных сетей), *Proceedings of the 30th International Conference on Machine Learning* (2013 г.): с. 1310–1318.

норме, устанавливая `clipnorm` вместо `clipvalue`. В таком случае будет урезаться целый градиент, когда его норма  $\ell_2$  больше подобранного порога. Например, если вы установите `clipnorm=1.0`, тогда вектор `[0.9, 100.0]` будет урезан до вектора `[0.00899964, 0.9999595]`, который сохраняет ориентацию, но почти исключает первый компонент. Если во время обучения вы заметили взрывной рост градиентов (размер градиентов можно отслеживать с применением TensorBoard), то можете опробовать как отсечение по величине, так и отсечение по норме с разными порогами и посмотреть, какой вариант работает лучше на проверочном наборе.

## Повторное использование заранее обученных слоев

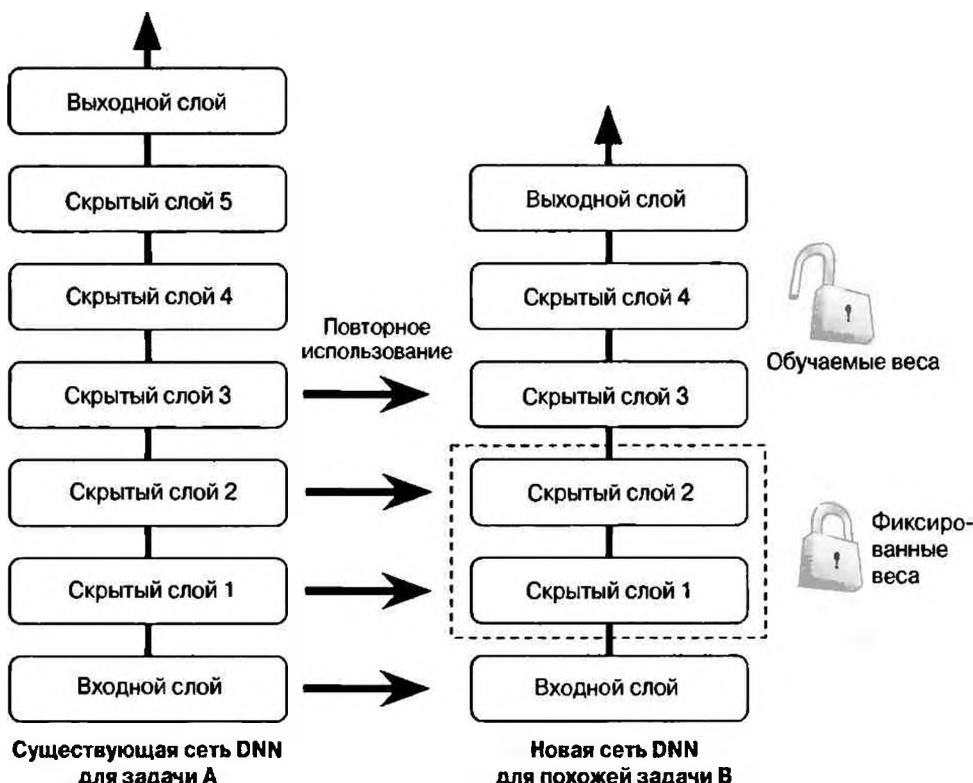
В большинстве случаев обучать очень крупные сети DNN с нуля — не особенно удачная идея: взамен вы должны попытаться отыскать существующую нейронную сеть, которая выполняет задачу, похожую на вашу (мы обсудим, как ее найти, в главе 14), и затем повторно задействовать нижние слои этой сети. Прием называется *обучением передачей знаний* (*transfer learning*). Он не только значительно ускоряет обучение, но также требует гораздо меньше данных.

Пусть у вас есть доступ к сети DNN, обученной классификации фотографий для 100 разных 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, даже частично пересекаются, поэтому вы должны попробовать повторно задействовать части первой сети (рис. 11.4).



Если входные фотографии в новой задаче имеют не такие же размеры, как в первой задаче, тогда вам обычно понадобится добавить шаг предварительной обработки для приведения их размеров к тем, которые ожидает первая модель. Вообще говоря, обучение передачей знаний будет работать лучше всего, когда входы имеют похожие низкоуровневые признаки.

Выходной слой исходной модели всегда должен заменяться, поскольку он почти наверняка окажется совершенно бесполезным для новой задачи и может даже не иметь корректного количества выходов.



*Рис. 11.4. Повторное использование заранее обученных слоев*

Аналогично самые верхние скрытые слои исходной модели с меньшей вероятностью будут столь же полезны, как нижние 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.



Чем больше похожи задачи, тем больше слоев вы пожелаете повторно задействовать (начиная с самых нижних слоев). Для очень похожих задач вы можете попробовать сохранить все скрытые слои и просто заменить выходной слой.

Попробуйте сначала заморозить все повторно используемые слои (т.е. сделайте их веса не поддающимися обучению, чтобы градиентный спуск не модифицировал их), затем обучите модель и посмотрите, как она себя ведет.

Далее разморозьте один или два верхних скрытых слоя, чтобы дать возможность обратному распространению подстроить их, и выясните, улучшилась ли эффективность. Чем больше обучающих данных вы имеете, тем больше слоев вы можете размораживать. При размораживании повторно используемых слоев полезно также снижать скорость обучения, что позволит избежать нарушения их точно настроенных весов.

Если вы по-прежнему не сумели добиться хорошей эффективности и располагаете только 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.

## Обучение передачей знаний с помощью Keras

Давайте рассмотрим пример. Предположим, что набор данных Fashion MNIST содержит только восемь классов — скажем, все кроме сандалий и футболок. Кто-то построил и обучил модель Keras на этом наборе и добился достаточно высокой эффективности (правильность более 90%). Назовем ее моделью A. Вы хотите заняться другой задачей: у вас есть изображения сандалий и футболок и нужно обучить двоичный классификатор (положительный класс = футболка, отрицательный класс = сандалии). Ваш набор данных довольно мал; есть только 200 помеченных изображений. Когда вы обучили новую модель для указанной задачи (назовем ее моделью B) с такой же архитектурой, как у модели A, она работает достаточно хорошо (правильность 97.2%). Но поскольку задача намного проще (существует всего лишь два класса), то вы надеялись на большее. За чашкой утреннего кофе вы осознаете, что ваша задача очень похожа на задачу A, так может быть обучение передачей знаний поможет? Надо выяснить!

Для начала необходимо загрузить модель A и создать новую модель на основе слоев модели A. Мы повторно задействуем все слои кроме выходного:

```
model_A = keras.models.load_model("my_model_A.h5")
model_B_on_A = keras.models.Sequential(model_A.layers[:-1])
model_B_on_A.add(keras.layers.Dense(1, activation="sigmoid"))
```

Обратите внимание, что `model_A` и `model_B_on_A` теперь совместно используют некоторые слои. Когда вы обучаете `model_B_on_A`, это также будет влиять на `model_A`. Если вы хотите избежать подобного, тогда понадобится клонировать модель `model_A` перед тем, как повторно задействовать ее слои. Для этого вы клонируете архитектуру модели A посредством метода `clone_model()` и затем копируете ее веса (т.к. `clone_model()` не клонирует веса):

```
model_A_clone = keras.models.clone_model(model_A)
model_A_clone.set_weights(model_A.get_weights())
```

Теперь вы могли бы обучить `model_B_on_A` для задачи B, но из-за того, что новый выходной слой был инициализирован случайным образом, он будет допускать крупные ошибки (во всяком случае, в течение нескольких начальных эпох), поэтому появятся большие градиенты ошибок, которые могут нарушить повторно используемые веса. Один из подходов решения проблемы предусматривает замораживание повторно задействованных слоев на период прохождения нескольких начальных эпох, давая новому слою некоторое время на узнавание приемлемых весов. Чтобы сделать это, нужно установить атрибут `trainable` каждого слоя в `False` и скомпилировать модель:

```
for layer in model_B_on_A.layers[:-1]:
    layer.trainable = False
model_B_on_A.compile(loss="binary_crossentropy", optimizer="sgd",
                      metrics=["accuracy"])
```



Вы всегда должны компилировать свою модель после замораживания или размораживания слоев.

Теперь вы можете обучить модель на протяжении нескольких эпох, затем разморозить повторно задействованные слои (что снова потребует компиляции модели) и продолжить обучение с целью точной настройки повторно задействованных слоев для задачи B. После размораживания повторно задействованных слоев обычно неплохо снизить скорость обучения снова для того, чтобы не нарушить повторно используемые веса:

```
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=4,
                            validation_data=(X_valid_B, y_valid_B))
```

```
for layer in model_B_on_A.layers[:-1]:  
    layer.trainable = True  
  
optimizer = keras.optimizers.SGD(lr=1e-4)      # по умолчанию lr=1e-2  
model_B_on_A.compile(loss="binary_crossentropy",  
                      optimizer=optimizer,  
                      metrics=["accuracy"])  
history = model_B_on_A.fit(X_train_B, y_train_B, epochs=16,  
                           validation_data=(X_valid_B, y_valid_B))
```

Итак, каков окончательный вердикт? Правильность при проверке модели составляет 99.25%, т.е. обучение передачей знаний сократило частоту ошибок с 2.8% до почти 0.7%! Получается, в четыре раза!

```
>>> model_B_on_A.evaluate(X_test_B, y_test_B)  
[0.06887910133600235, 0.9925]
```

Уверены в этом? Вы не должны быть уверены: я сжульничал! Я перепробовал много конфигураций, пока не нашел одну, которая продемонстрировала серьезное улучшение. Если вы попытаетесь изменить классы или начальное случайное значение, то увидите, что улучшение в целом уменьшится или даже исчезнет либо развернется в противоположную сторону. То, что я сделал, называется “пытать данные пока они не признаются”. Когда статья выглядит чересчур позитивно, это должно вызвать у вас подозрение: возможно сенсационная новая методика в действительности не особенно помогает (фактически она может даже ухудшить эффективность), но авторы перепробовали множество вариантов и отчитались только о наилучших результатах (которые могли быть получены по счастливой случайности), не упоминая о том, сколько неудач они встретили на своем пути. В большинстве случаев в этом нет злого умысла, но является частью причины, по которой так много результатов в науке никогда не могут быть воспроизведены.

Почему я сжульничал? Оказывается, что обучение передачей знаний не очень хорошо работает с небольшими плотными сетями, предположительно из-за того, что небольшие сети узнают мало паттернов и плотные сети выявляют крайне специфические паттерны, которые вряд ли будут полезными в других задачах. Обучение передачей знаний работает лучше с глубокими сверточными нейронными сетями, которые имеют тенденцию узнавать обнаружители признаков, являющиеся более общими (особенно на самых низких слоях). В главе 14 мы еще вернемся к обучению передачей знаний, применяя методики, которые только что обсуждались (и я обещаю, что никакого жульничества не будет!).

## Предварительное обучение без учителя

Допустим, 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 отчайтайтесь! Прежде всего, вы должны попытаться собрать больше помеченных обучающих данных, но если это невозможно, то вы все еще способны провести *предварительное обучение без учителя* (*unsupervised pretraining*), которое проиллюстрировано на рис. 11.5.

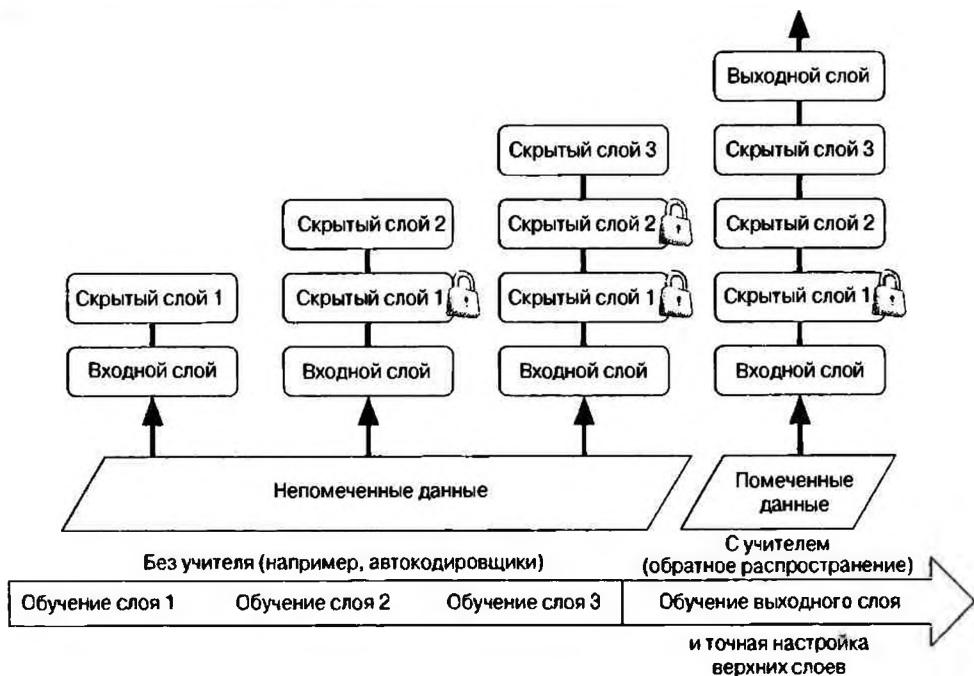


Рис. 11.5. При обучении без учителя модель обучается на непомеченных данных (или на всех данных) с использованием методики обучения без учителя, после чего она точно настраивается для финальной задачи на помеченных данных с применением методики обучения с учителем; часть без учителя может обучать по одному слою за раз, как показано здесь, или обучать полную модель напрямую

На самом деле зачастую легко собрать непомеченные обучающие образцы, но снабжение их метками сопряжено с высокими затратами. Если вы в состоянии собрать множество непомеченных обучающих данных, то можете попробовать задействовать их для обучения модели без учителя, такой как автокодировщик или порождающая состязательная сеть (*generative adversarial network — GAN*), рассматриваемая в главе 17. Затем вы можете

повторно использовать самые нижние слои автокодировщика или дискриминатора GAN, добавить поверх выходной слой для своей задачи и точно настроить финальную сеть с применением обучения с учителем (т.е. с помеченными обучающими образцами).

Именно такую методику Джейфри Хинтон и его команда использовали 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 помощью *ограниченных машин Больцмана* (*restricted Boltzmann machine — RBM*)), и только после смягчения проблемы исчезновения градиентов более распространенным стало обучение сетей DNN исключительно с применением обучения с учителем. Предварительное обучение без учителя (в наши дни обычно с использованием автокодировщиков или сетей GAN, а не машин RBM) по-прежнему является хорошим вариантом в ситуации, когда есть сложная в решении задача, отсутствует похожая модель, которой можно было бы воспользоваться, и доступно мало помеченных, но масса непомеченных обучающих данных.

Обратите внимание, что в первые дни существования глубокого обучения было трудно обучать глубокие модели, поэтому люди применяли методику под названием *жадное послойное предварительное обучение* (*greedy layer-wise pretraining*), изображенное на рис. 11.5. Сначала они обучали модель без учителя с единственным слоем, обычно машиной RBM, затем затем замораживали этот слой и добавляли выше него еще один, далее снова обучали модель (фактически просто обучая новый слой), после чего замораживали новый слой, добавляли поверх него еще один слой, снова обучали модель и т.д. В наши дни все стало значительно проще: люди, как правило, обучаются сразу полную модель без учителя (т.е. на рис. 11.5 старт происходит прямо с третьего шага) и вместо машин RBM используют автокодировщики или сети GAN.

## Предварительное обучение на вспомогательной задаче

Когда помеченных обучающих данных у вас мало, последний вариант заключается в том, чтобы обучить первую нейронную сеть на вспомогательной задаче, для которой легко получить или сгенерировать помеченные обучающие данные, и затем повторно задействовать нижние слои этой сети для решаемой задачи. Нижние слои первой нейронной сети узнают детекторы признаков, которые вероятно будут повторно используемыми второй нейронной сетью.

Например, при построении системы для распознавания лиц у вас может быть лишь несколько фотографий каждого индивидуума — явно недостаточно для обучения хорошего классификатора. Накопление сотен фотографий каждой персоны не будет практическим подходом. Тем не менее, вы могли бы собрать множество фотографий случайных людей в веб-сети и обучить первую нейронную сеть для обнаружения, представляют ли две разных фотографии одну и ту же персону. Такая сеть узнала бы хорошие детекторы признаков для лиц, а потому повторное применение ее нижних слоев позволило бы обучить эффективный классификатор лиц с использованием обучающих данных небольшого объема.

Для приложений обработки естественного языка (*natural language processing* — *NLP*) вы можете загрузить корпус миллионов текстовых документов и автоматически генерировать из него помеченные данные. Скажем, вы могли бы случайным образом замаскировать некоторые слова и обучить модель прогнозированию отсутствующих слов (например, она должна вырабатывать прогноз, что пропущенным словом в англоязычном предложении “*What \_\_\_\_ you saying?*” вероятно является “*are*” или “*were*”). Если вы сумеете обучить модель для достижения хорошей эффективности на такой задаче, тогда она уже будет знать достаточно много о языке, и ее определенно можно будет повторно применить для фактической задачи и точно настраивать на ваших помеченных данных (мы обсудим дополнительные задачи предварительного обучения в главе 15).



*Обучение со своим учителем (self-supervised learning)* происходит, когда вы автоматически генерируете метки из самих данных, после чего обучаете модель на результирующем “помеченном” наборе данных, используя методики обучения с учителем. Поскольку такой подход не требует какой-либо пометки со стороны людей, его лучше всего классифицировать как форму обучения без учителя.

## 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 предварительно обученной сети (возможно, на вспомогательной задаче или с помощью обучения без учителя). Еще один гигантский скачок скорости обеспечивает применение более быстрого оптимизатора, чем обыкновенный оптимизатор на основе градиентного спуска. В настоящем разделе мы представим самые популярные алгоритмы: *моментную оптимизацию* (*momentum optimization*), *ускоренный градиент Нестерова* (*Nesterov Accelerated Gradient*), AdaGrad, RMSProp и в заключение оптимизацию Adam и Nadam.

## Моментная оптимизация

Вообразите большой мяч, катящийся по гладкой поверхности с пологим уклоном: он начинает движение медленно, но быстро накапливает 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 году (<https://hml.info/54>)<sup>13</sup>. Напротив, 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  $\theta$  путем прямого вычитания градиента функции издержек  $J(\theta)$  относительно весов ( $\nabla_{\theta} J(\theta)$ ), умноженного на скорость обучения  $\eta$ . Вот уравнение:  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta)$ . Он не заботится о том, какими были начальные градиенты. Если локальный градиент очень маленький, тогда процесс продвигается крайне медленно.

Моментная оптимизация принимает во внимание предыдущие градиенты: на каждой итерации локальные градиенты вычитываются из *вектора момента* (*momentum vector*)  $m$ , умноженного на скорость обучения  $\eta$ , а веса обновляются добавлением этого вектора момента (уравнение 11.4). Другими словами, градиенты используются для ускорения, а не для скорости. Чтобы эмулировать механизм трения какого-то вида и предотвратить слишком сильный рост момента, в алгоритме вводится новый гиперпараметр  $\beta$ , называемый *моментом* (*momentum*), который должен быть установлен между 0 (высокое трение) и 1 (отсутствие трения). Типичное значение момента составляет 0.9.

<sup>13</sup> Борис Поляк, *Some methods of speeding up the convergence of iteration methods* (Некоторые методы ускорения сходимости методов итерации), *USSR Computational Mathematics and Mathematical Physics* 4, выпуск 5 (1964 г.): с. 1–17.

## 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  $\eta$ , 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, так что моментная оптимизация заканчивается в 10 раз раньше градиентного спуска! Это позволяет моментной оптимизации покидать плато намного быстрее, чем градиентный спуск. 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. Напротив, моментная оптимизация будет скатываться во впадину все быстрее и быстрее, пока не достигнет низа (оптимума). В глубоких нейронных сетях, не применяющих пакетную оптимизацию, нижние слои часто будут иметь входы с очень разными масштабами, поэтому 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 схождение.

Реализовать моментную оптимизацию в Keras очень просто: просто примените оптимизатор SGD и установите его гиперпараметр momentum, после чего наслаждайтесь полученными выгодами!

```
optimizer = keras.optimizers.SGD(lr=0.001, 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 обычный градиентный спуск.

## Ускоренный градиент Нестерова

Небольшая вариация моментной оптимизации, предложенная Юрием Нестеровым в 1983 году (<https://homl.info/55>)<sup>14</sup>, почти всегда быстрее обыкновенной моментной оптимизации. Метод ускоренного градиента Нестерова (*Nesterov Accelerated Gradient* — NAG), называемый также *моментной оптимизацией Нестерова*, измеряет градиент функции издержек не в локальной позиции  $\theta$ , а чуть дальше в направлении момента, в точке  $\theta + \beta m$  (уравнение 11.5).

### Уравнение 11.5. Алгоритм ускоренного градиента Нестерова

1.  $m \leftarrow \beta m - \eta \nabla_{\theta} J(\theta + \beta m)$
2.  $\theta \leftarrow \theta + m$

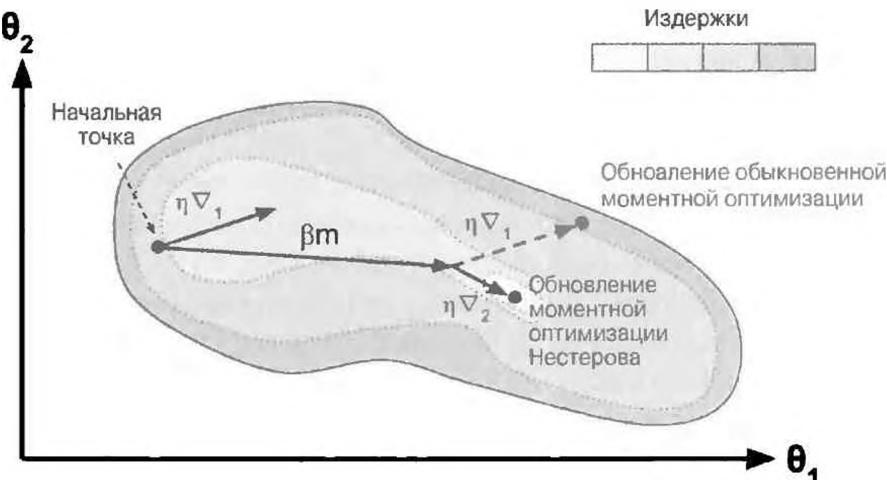
Такая небольшая подстройка работает из-за того, что в общем случае вектор момента будет указывать в правильном направлении (т.е. к оптимуму). Таким образом, использовать градиент, измеренный чуть дальше в данном направлении, будет несколько точнее, чем градиент в исходной позиции, как видно на рис. 11.6 (где  $\nabla_1$  представляет градиент функции издержек, измеренный в начальной точке  $\theta$ , а  $\nabla_2$  — градиент в точке  $\theta + \beta m$ ).

Легко заметить, что обновление моментной оптимизации Нестерова оказывается чуть ближе к оптимуму. Через некоторое время такие мелкие улучшения накапливаются, и алгоритм NAG становится значительно быстрее обыкновенной моментной оптимизации. Кроме того, обратите внимание, что когда момент продвигает веса через впадину,  $\nabla_1$  продолжает двигаться дальше через впадину, тогда как  $\nabla_2$  двигается в направлении нижней точки впадины. Это помогает сократить колебания и в результате сделать схождение алгоритма NAG более быстрым.

Алгоритм NAG, как правило, быстрее обыкновенной моментной оптимизации. Для его использования просто установите `nesterov=True` при создании оптимизатора SGD:

```
optimizer = keras.optimizers.SGD(lr=0.001, momentum=0.9,  
                                  nesterov=True)
```

<sup>14</sup> Юрий Нестеров, *A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence O(1/k<sup>2</sup>)* (Метод для решения задачи неограниченной выпуклой минимизации со скоростью сходимости  $O(1/k^2)$ ), *Doklady AN USSR* 269 (1983 г.): с. 543–547.



**Рис. 11.6.** Сравнение обычной моментной оптимизации и моментной оптимизации Нестерова: первая применяет градиенты, вычисленные перед шагом момента, тогда как вторая применяет градиенты, вычисленные после

## AdaGrad

Снова рассмотрим проблему вытянутой чаши: градиентный спуск начинает с быстрого движения вниз по самому крутому уклону, который не указывает прямо в направлении глобального оптимума, и затем очень медленно перемещается в нижнюю точку впадины. Было бы неплохо, если бы алгоритм мог пораньше скорректировать свое направление к точке, более близкой к глобальному оптимуму. Алгоритм AdaGrad (<https://homl.info/56>)<sup>15</sup> обеспечивает такую коррекцию, пропорционально уменьшая вектор-градиент вдоль самых крутых направлений (уравнение 11.6).

### Уравнение 11.6. Алгоритм AdaGrad

1.  $s \leftarrow s + \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta)$
2.  $\theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon}$

Первый шаг накапливает квадраты градиентов в векторе  $s$  (вспомните, что символ  $\otimes$  представляет поэлементное умножение). Такая векторизованная форма эквивалентна вычислению  $s_i \leftarrow s_i + (\partial J(\theta) / \partial \theta_i)^2$  для каждого элемен-

<sup>15</sup> Джон Дучи и др., *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization* (Адаптивные субградиентные методы для динамического обучения и стохастической оптимизации), *Journal of Machine Learning Research* 12 (2011 г.): с. 2121–2159.

та  $s_i$  вектора  $s$ ; другими словами, каждый  $s_i$  накапливает квадраты частной производной функции издержек относительно параметра  $\theta_i$ . Если функция издержек является крутой вдоль  $i$ -того измерения, тогда  $s_i$  будет становиться все больше и больше на каждой итерации.

Второй шаг почти идентичен градиентному спуску, но с одним большим отличием: вектор-градиент пропорционально уменьшается на коэффициент  $\sqrt{s + \epsilon}$  (символ  $\oslash$  представляет поэлементное деление, а  $\epsilon$  — сглаживающий член, позволяющий избежать деления на ноль, который обычно устанавливается в  $10^{-10}$ ). Такая векторизованная форма эквивалентна одновременному вычислению  $\theta_i \leftarrow \theta_i - \eta \frac{\partial J(\theta)}{\partial \theta_i} / \sqrt{s_i + \epsilon}$  для всех параметров  $\theta_i$ .

Выражаясь кратко, алгоритм ослабляет скорость обучения, но для крутых измерений делает это быстрее, чем для измерений с более пологими уклонаами. Получается то, что называется *адаптивной скоростью обучения* (*adaptive learning rate*). Она помогает направлять результирующие обновления более прямо к глобальному оптимуму (рис. 11.7). Дополнительное преимущество связано с тем, что требуется гораздо меньший объем подстройки гиперпараметра скорости обучения  $\eta$ .

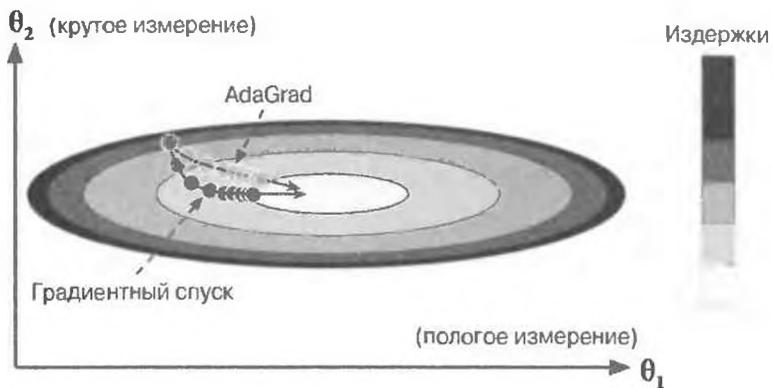


Рис. 11.7. Сравнение алгоритма AdaGrad и градиентного спуска: первый способен раньше корректировать свое направление к точке оптимума

Алгоритм AdaGrad часто работает лучше для простых квадратичных задач, но 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 Keras имеется оптимизатор Adagrad, вы не должны его применять

для обучения глубоких нейронных сетей (однако он может оказаться эффективным для более простых задач вроде линейной регрессии). Тем не менее, понимание алгоритма AdaGrad полезно, т.к. позволяет уловить суть других оптимизаторов с адаптивной скоростью обучения.

## RMSProp

Как вы уже видели, существует риск того, что алгоритм AdaGrad слишком быстро замедляется и никогда не сходится в глобальном оптимуме. В алгоритме RMSProp<sup>16</sup> проблема устраняется за счет накопления только градиентов из самых последних итераций (в противоположность накоплению всех градиентов с начала обучения). Это делается с использованием экспоненциального ослабления на первом шаге (уравнение 11.7).

### Уравнение 11.7. Алгоритм RMSProp

$$\begin{aligned} 1. \quad & s \leftarrow \beta s + (1 - \beta) \nabla_{\theta} J(\theta) \otimes \nabla_{\theta} J(\theta) \\ 2. \quad & \theta \leftarrow \theta - \eta \nabla_{\theta} J(\theta) \oslash \sqrt{s + \epsilon} \end{aligned}$$

Коэффициент ослабления  $\beta$  обычно устанавливается в 0.9. Да, мы снова имеем новый гиперпараметр, но такое стандартное значение часто хорошо работает, так что потребность в его подстройке может вообще не возникать.

Как и можно было ожидать, в Keras имеется оптимизатор RMSprop:

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9)
```

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 и Nadam

Оптимизация Adam (*adaptive moment estimation* — адаптивная оценка момента; <https://hml.info/59>)<sup>17</sup> объединяет идеи моментной опти-

<sup>16</sup> Данный алгоритм был создан Джейфри Хинтоном и Тийменом Тилеманом в 2012 году и презентован Джейфри Хинтоном в его курсе Coursera по нейронным сетям (слайды: <https://hml.info/57>; видеоролик: <https://hml.info/58>). Интересно отметить, что поскольку авторы не написали статью, которая описывала бы его, исследователи в своих работах часто ссылаются на “слайд 29 в лекции 6”.

<sup>17</sup> Дидерик Кингма, Джимми Ба, *Adam: A Method for Stochastic Optimization* (Adam: метод стохастической оптимизации), препринт arXiv: 1412.6980 (2014 г.).

мизации и RMSProp: как и моментная оптимизация, она отслеживает экспоненциально ослабляемое среднее арифметическое прошедших градиентов, и подобно RMSProp она отслеживает экспоненциально ослабляемое среднее арифметическое квадратов прошедших градиентов (уравнение 11.8)<sup>18</sup>.

### 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.  $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4.  $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5.  $\theta \leftarrow \theta + \eta \widehat{\mathbf{m}} \odot \sqrt{\widehat{\mathbf{s}} + \epsilon}$

В уравнении  $t$  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: т.к.  $\mathbf{m}$  и  $\mathbf{s}$  инициализируются значением 0, они будут смешены в сторону 0 в начале обучения, а потому указанные два шага помогут поднять  $\mathbf{m}$  и  $\mathbf{s}$  в начале обучения.

Гиперпараметр ослабления момента  $\beta_1$ , как правило, инициализируется значением 0.9, а гиперпараметр ослабления масштабирования  $\beta_2$  часто инициализируется значением 0.999. Как и ранее, сглаживающий член  $\epsilon$  обычно инициализируется крошечным числом, скажем,  $10^{-7}$ . Таковы стандартные значения для класса Adam (если быть точным, то `epsilon` по умолчанию устанавливается в `None`, что сообщает Keras о необходимости применения `keras.backend.epsilon()` со стандартным значением  $10^{-7}$ ; вы можете изменить его с использованием `keras.backend.set_epsilon()`).

<sup>18</sup> Они являются оценками средней величины и (неотцентрированной) дисперсии градиентов. Среднюю величину часто именуют первым моментом, а дисперсию — вторым моментом, отсюда и название алгоритма.

Вот как создать оптимизатор Adam с помощью Keras:

```
optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2=0.999)
```

Поскольку Adam является алгоритмом с адаптивной скоростью обучения (подобно AdaGrad и RMSProp), он требует меньшей подстройки гиперпараметра скорости обучения  $\eta$ . Часто можно применять стандартное значение  $\eta = 0.001$ , делая использование алгоритма Adam даже более легким, чем градиентного спуска.



Если вы начинаете ощущать себя подавленными всеми численными методиками и интересуетесь, как выбрать подходящую для решаемой задачи, то не стоит переживать: в конце главы дается ряд практических указаний.

В заключение заслуживают упоминания две разновидности Adam.

### AdaMax

Обратите внимание, что на шаге 2 уравнения 11.8 алгоритм Adam накапливает квадраты градиентов в  $s$  (с более высоким весом для самых последних весов). На шаге 5, если мы проигнорируем  $\epsilon$  вместе с шагами 3 и 4 (которые в любом случае являются техническими деталями), то алгоритм Adam уменьшает масштаб обновлений параметров на квадратный корень  $s$ . Выражаясь кратко, Adam уменьшает масштаб обновлений параметров на норму  $\ell_2$  затухающих во времени градиентов (вспомните, что норма  $\ell_2$  представляет собой квадратный корень суммы квадратов). Алгоритм AdaMax, описанный в той же статье, что и Adam, заменяет норму  $\ell_2$  нормой  $\ell_\infty$  (причудливый способ выражения максимума). Более конкретно он меняет шаг 2 в уравнении 11.8 на  $s \leftarrow \beta s + \max(\beta s, \nabla_{\theta} J(\theta))$ , отбрасывает шаг 4, а на шаге 5 уменьшает масштаб обновлений градиентов на коэффициент  $s$ , который является просто максимумом затухающих во времени градиентов. На практике такой прием делает AdaMax более стабильным, чем Adam, но на самом деле он зависит от набора данных и в целом Adam работает лучше. Следовательно, это всего лишь еще один оптимизатор, который вы можете опробовать, если на некоторых задачах возникают проблемы с Adam.

Оптимизация Nadam представляет собой оптимизацию Adam плюс трюк Нестерова, так что она часто будет сходиться быстрее Adam. В своем докладе, представляющем такую методику (<https://hml.info/nadam>)<sup>19</sup>, исследователь Тимоти Дозат сравнивает множество разных оптимизаторов на различных задачах и обнаруживает, что Nadam в целом превосходит Adam, но временами его превосходит RMSProp.



Адаптивные методы оптимизации (включая RMSProp, Adam и Nadam) часто великолепны и быстро сходятся в хорошее решение. Однако в статье 2017 года (<https://hml.info/60>)<sup>20</sup> Аши Вилсон и др. показано, что они могут привести к решениям, которые плохо обобщаются на определенные наборы данных. Таким образом, когда вы разочарованы эффективностью своей модели, попробуйте взамен применить ускоренный градиент Нестерова: ваш набор данных просто может не выносить адаптивные градиенты. Также ознакомьтесь с последними исследованиями, поскольку они быстро развиваются.

Все рассмотренные до сих пор методики оптимизации опирались только на *частные производные первого порядка* (якобианы (*Jacobian*)). В литературе по оптимизации также описаны изумительные алгоритмы, основанные на *частных производных второго порядка* (гессианах (*Hessian*)), которые являются частными производными якобианов). 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 всего лишь  $n$  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.

<sup>19</sup> Тимоти Дозат, *Incorporating Nesterov Momentum into Adam* (Внедрение момента Нестерова в Adam) (2016 г.).

<sup>20</sup> Аша Вилсон и др., *The Marginal Value of Adaptive Gradient Methods in Machine Learning* (Предельное значение адаптивных градиентных методов в машинном обучении), *Advances in Neural Information Processing Systems 30* (2017 г.): с. 4148–4158.

## 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). Обратите внимание, что такой прием обычно не будет приводить к крайне разреженной модели и может вызвать снижение ее эффективности.

Более удачный вариант предполагает применение во время обучения сильной регуляризации  $\ell_1$  (демонстрируется позже в главе), потому что она вынуждает оптимизатор обнулять столько весов, сколько возможно (как обсуждалось в разделе “Лассо-регрессия” главы 4).

Если такие приемы оказываются недостаточными, тогда обратитесь к комплексу инструментов для оптимизации моделей *TensorFlow* (*TensorFlow Model Optimization Toolkit — TF-MOT*; <https://homl.info/tfmot>), который предлагает API-интерфейс отсечения, способный итеративно удалять связи во время обучения на основе их величины.

В табл. 11.2 сравниваются все оптимизаторы, которые мы обсуждали до сих пор (здесь \* означает плохо, \*\* — средне и \*\*\* — хорошо).

Таблица 11.2. Сравнение оптимизаторов

Класс	Скорость сходжения	Качество сходжения
SGD	*	***
SGD(momentum=...)	**	***
SGD(momentum=..., nesterov=True)	**	***
Adagrad	***	* (останавливается слишком рано)
RMSprop	***	** или ***
Adam	***	** или ***
Nadam	***	** или ***
AdaMax	***	** или ***

## Планирование скорости обучения

Очень важно найти хорошую скорость обучения. Если вы установите ее чересчур высокой, тогда обучение может расходиться (как обсуждалось в разделе “Градиентный спуск” главы 4). Если вы установите скорость слишком низкой, то обучение в итоге сойдется в оптимуме, но через очень длительный период времени. Если вы установите ее близкой к чересчур высокой, тогда обучение поначалу будет продвигаться очень быстро, но в конце станет совершать прыжки вокруг оптимума, фактически никогда не попадая в него. 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).



Рис. 11.8. Кривые обучения для различных скоростей обучения  $\eta$

Как было указано в главе 10, найти хорошую скорость обучения можно за счет обучения модели в течение нескольких сотен итераций с экспоненциальным увеличением скорости обучения от очень малой до очень большой величины, последующего просмотра кривых обучения и выбора скорости обучения, чуть меньшей той, при которой потеря начинает подниматься.

Но вы можете поступить лучше, чем использовать постоянную скорость обучения: если начать с высокой скорости обучения и затем понижать ее, как только обучение перестает приводить к скорому прогрессу, то достичь хорошего решения удастся быстрее, нежели с помощью оптимальной постоянной скорости обучения. Существует множество стратегий понижения скорости во время обучения. Также может быть полезно начать с низкой скорости обучения, увеличивать ее и затем снова понижать. Такие стратегии называются *графиками обучения* (мы кратко представляли эту концепцию в главе 4). Ниже перечислены наиболее распространенные из них.

## График мощности (*power scheduling*)

Установите скорость обучения в функцию номера итерации  $t$ :  $\eta(t) = \eta_0(1 + t/s)^c$ . Начальная скорость обучения  $\eta_0$ , степень  $c$  (обычно равная 1) и количество шагов  $s$  являются гиперпараметрами. На каждом шаге скорость обучения понижается. После  $s$  шагов она снижается до  $\eta_0/2$ . После дополнительных  $s$  шагов скорость обучения уменьшается до  $\eta_0/3$ , затем до  $\eta_0/4$ , до  $\eta_0/5$  и т.д. Как вы увидите, такой график сначала понижается быстро, а потом все медленнее и медленнее. Разумеется, график мощности требует подстройки  $\eta_0$  и  $s$  (и возможно  $c$ ).

## Экспоненциальный график (*exponential scheduling*)

Установите скорость обучения в  $\eta(t) = \eta_0 0.1^{ts}$ . Скорость обучения будет постепенно понижаться в 10 раз каждые  $s$  шагов. В то время как график мощности снижает скорость обучения все медленнее и медленнее, экспоненциальный график продолжает сокращать ее на порядок каждые  $s$  шагов.

## Кусочно-линейный постоянный график (*piecewise constant scheduling*)

Применяйте постоянную скорость обучения в течение некоторого количества эпох (скажем,  $\eta_0 = 0.1$  для 5 эпох), затем меньшую скорость обучения для другого количества эпох (например,  $\eta_1 = 0.001$  для 50 эпох) и т.д. Хотя такое решение может работать очень хорошо, оно требует времени на выяснение правильной последовательности скоростей обучения и длительности использования каждой из них.

## График эффективности (*performance scheduling*)

Измеряйте ошибку проверки каждые  $N$  шагов (как при раннем прекращении) и понижайте скорость обучения в  $\lambda$  раз, когда ошибка перестает уменьшаться.

## График одного цикла (*1cycle scheduling*)

В противоположность остальным подходам график одного цикла (представленный в статье 2018 года (<https://hml.info/1cycle>)<sup>21</sup> Лесли Смитом) начинается с увеличения исходной скорости обучения

<sup>21</sup> Лесли Смит, *A Disciplined Approach to Neural Network Hyper-Parameters: Part I — Learning Rate, Batch Size, Momentum, and Weight Decay* (Дисциплинированный подход к гиперпараметрам: часть 1 — скорость обучения, размер пакета, инерция и уменьшение весов), препринт arXiv:1803.09820 (2018 г.).

$\eta_0$  и возрастает линейно до  $\eta_1$  на полпути обучения. Затем во второй половине обучения он линейно понижает скорость обучения снова до  $\eta_0$ , заканчивая последние эпохи понижением скорости на несколько порядков (по-прежнему линейно). Максимальная скорость обучения  $\eta_1$  выбирается с применением того же самого подхода, который мы использовали для нахождения оптимальной скорости обучения, а начальная скорость обучения  $\eta_0$  выбирается приблизительно в 10 раз меньшей. Когда применяется момент, мы начинаем с высокого момента (скажем, 0.95), затем в течение первой половины обучения снижаем его до более низкого момента (например, линейно до 0.85) и далее во время второй половины обучения доводим момент обратно до максимальной величины (0.95), заканчивая последние несколько эпох с этой максимальной величиной. Смит провел много экспериментов, показывая, что данный подход часто был в состоянии значительно ускорять обучение и обеспечивать лучшую эффективность. Скажем, на популярном наборе данных с изображениями CIFAR10 описанный подход достигал 91.9%-ной правильности при проверке всего лишь за 100 эпох в сравнении с 90.3%-ной правильностью за 800 эпох в случае использования стандартного подхода (с той же самой архитектурой нейронной сети).

В статье 2013 года (<https://homl.info/63>)<sup>22</sup> Эндрю Сеньора и др. сравнивается эффективность некоторых наиболее 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 эффективности и экспоненциальный график. Авторы отдали предпочтение экспоненциальному графику, поскольку его легче настраивать, и он чуть быстрее сходится в оптимальное решение (они также упомянули о том, что реализовать экспоненциальный график легче, чем график эффективности, но в Keras просты оба варианта). Тем не менее, график одного цикла похоже работает даже лучше.

Реализовать график мощности в Keras легче всего: при создании оптимизатора просто установите гиперпараметр `decay`:

```
optimizer = keras.optimizers.SGD(lr=0.01, decay=1e-4)
```

<sup>22</sup> Эндрю Сеньор и др., *An Empirical Study of Learning Rates in Deep Neural Networks for Speech Recognition* (Эмпирическое исследование скоростей обучения в глубоких нейронных сетях для распознавания речи), *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2013 г.): с. 6724–6728.

Гиперпараметр `decay` является инверсией  $s$  (количество шагов, необходимых для деления скорости обучения на еще одну единицу измерения) и Keras предполагает, что  $s$  равно 1.

Реализация экспоненциального и кусочно-линейного постоянного графиков тоже довольно проста. Вы сначала определяете функцию, которая принимает текущую эпоху и возвращает скорость обучения. Например, давайте реализуем экспоненциальный график:

```
def exponential_decay_fn(epoch):  
    return 0.01 * 0.1**(epoch / 20)
```

Если вы не хотите жестко кодировать  $\eta_0$  и  $s$ , тогда можете создать функцию, которая возвращает сконфигурированную функцию:

```
def exponential_decay(lr0, s):  
    def exponential_decay_fn(epoch):  
        return lr0 * 0.1**(epoch / s)  
    return exponential_decay_fn  
  
exponential_decay_fn = exponential_decay(lr0=0.01, s=20)
```

Далее вы создаете обратный вызов `LearningRateScheduler`, предоставляя ему функцию графика, и передаете этот обратный вызов методу `fit()`:

```
lr_scheduler =  
    keras.callbacks.LearningRateScheduler(exponential_decay_fn)  
history = model.fit(X_train_scaled, y_train, [...],  
                     callbacks=[lr_scheduler])
```

Обратный вызов `LearningRateScheduler` будет обновлять атрибут `learning_rate` оптимизатора в начале каждой эпохи. Единственного обновления скорости обучения на эпоху обычно достаточно, но если вы хотите обновлять ее чаще, скажем, на каждом шаге, то всегда можете написать собственный обратный вызов (пример ищите в разделе “Exponential Scheduling” (Экспоненциальный график) тетради Jupiter для настоящей главы). Обновление скорости обучения на каждом шаге имеет смысл, когда на эпоху приходится много шагов. В качестве альтернативы вы можете использовать подход с `keras.optimizers.schedules`, который вскоре будет описан.

Функция графика дополнительно может принимать текущую скорость обучения как второй аргумент. Например, следующая функция графика умножает предыдущую скорость обучения на  $0.1^{1/20}$ , давая в итоге то же самое экспоненциальное затухание (за исключением того, что затухание теперь стартует в начале эпохи 0, а не 1):

```
def exponential_decay_fn(epoch, lr):
    return lr * 0.1**(1 / 20)
```

Такая реализация полагается на начальную скорость обучения оптимизатора (в противоположность предыдущей реализации), поэтому обеспечьте ее надлежащую установку.

Когда вы сохраняете модель, вместе с ней сохраняются оптимизатор и его скорость. Это означает, что с помощью новой функции графика вы могли бы всего лишь загрузить обученную модель и без проблем продолжить обучение с места, где оно было оставлено. Однако все не так просто, если ваша функция графика применяет аргумент epoch: эпоха не сохраняется и сбрасывается в 0 при каждом вызове метода `fit()`. Если бы вы продолжили обучение модели с места, где оно было оставлено, то это привело бы к очень высокой скорости обучения, что вполне вероятно повредило бы веса модели. Решение заключается в ручной установке аргумента `initial_epoch` метода `fit()`, так что epoch начнется с правильного значения.

Для кусочно-линейного постоянного графика вы можете использовать функцию графика вроде показанной ниже (как и ранее, при желании можете определить более общую функцию; за примером обращайтесь в раздел “Piecewise Constant Scheduling” (Кусочно-линейный постоянный график) тетради Jupiter для настоящей главы), после чего создайте обратный вызов `LearningRateScheduler` с этой функцией и передайте его методу `fit()`, как поступали в случае экспоненциального графика:

```
def piecewise_constant_fn(epoch):
    if epoch < 5:
        return 0.01
    elif epoch < 15:
        return 0.005
    else:
        return 0.001
```

Для графика эффективности примените обратный вызов `ReduceLROnPlateau`. Например, если вы передадите методу `fit()` приведенный далее обратный вызов, то он будет умножать скорость обучения на 0.5 всякий раз, когда наилучшая потеря при проверке не улучшается в течение пяти следующих друг за другом эпох (доступны другие варианты; дополнительные детали ищите в документации):

```
lr_scheduler = keras.callbacks.ReduceLROnPlateau(factor=0.5, patience=5)
```

Наконец, `tf.keras` предлагает альтернативный способ реализации графика для скорости обучения: определите скорость обучения с использованием одного из графиков, доступных в `keras.optimizers.schedules`, и затем передайте эту скорость обучения любому оптимизатору. Такой подход предусматривает обновление скорости обучения на каждом шаге, а не на каждой эпохе. Скажем, вот как реализовать тот же самый экспоненциальный график, что и определенная ранее функция `exponential_decay_fn()`:

```
s = 20 * len(X_train) // 32      # количество шагов в 20 эпохах
                                    # (размер пакета = 32)
learning_rate =
    keras.optimizers.schedules.ExponentialDecay(0.01, s, 0.1)
optimizer = keras.optimizers.SGD(learning_rate)
```

Результат прост и элегантен, вдобавок при сохранении модели также сохраняются скорость обучения и график ее изменений (включая состояние). Тем не менее, такой подход не является частью API-интерфейса Keras; он специфичен для `tf.keras`.

Что касается графика одного цикла, то реализация не представляет особой сложности: просто создайте специальный обратный вызов, который модифицирует скорость обучения на каждой итерации (вы можете обновлять скорость обучения оптимизатора, изменения `self.model.optimizer.lr`). За примером обращайтесь в раздел “1Cycle scheduling” (График одного цикла) тетради Jupyter для главы.

Подводя итоги, можно сказать, что экспоненциальное затухание, график эффективности и график одного цикла могут значительно ускорить схождение, поэтому обязательно опробуйте их!

## 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 г.), с. 297

С помощью тысяч параметров можно описать целый зоопарк. Глубокие нейронные сети обычно имеют десятки тысяч параметров, а иногда параметров миллионы. Благодаря настолько большому количеству параметров сети обладают невероятной степенью свободы и могут подгоняться к широкому многообразию сложных наборов данных. Но такая огромная гибкость также делает сети предрасположенными к переобучению обучающим набором. Нам нужна регуляризация.

В главе 10 мы уже реализовывали один из лучших приемов регуляризации: раннее прекращение. Кроме того, хотя пакетная нормализация проектировалась для решения проблем нестабильных градиентов, она также действует как довольно хороший регуляризатор. В текущем разделе мы исследуем другие популярные методики регуляризации для нейронных сетей: регуляризацию  $\ell_1$  и  $\ell_2$ , отключение и регуляризацию на основе max-нормы (max-norm regularization).

## Регуляризация $\ell_1$ и $\ell_2$

Подобно тому, как делалось для простых линейных моделей в главе 4, вы можете использовать регуляризацию  $\ell_2$  для ограничения весов связей нейронной сети и/или регуляризацию  $\ell_1$ , если нужна разреженная модель (со многими весами, равными 0). Вот как применить регуляризацию  $\ell_2$  к весам связей слоя Keras, используя коэффициент регуляризации 0.01:

```
layer = keras.layers.Dense(100, activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))
```

Функция `l2()` возвращает регуляризатор, который будет вызываться на каждом шаге во время обучения для подсчета потери регуляризации. Результат затем добавляется к финальной потере. Как и следовало ожидать, если необходима регуляризация  $\ell_1$ , то можете просто применять `keras.regularizers.l1()`; если же вас интересуют обе регуляризации,  $\ell_1$  и  $\ell_2$ , тогда используйте `keras.regularizers.l1_l2()` (указав два коэффициента регуляризации).

Поскольку обычно вы будете применять один и тот же регуляризатор ко всем слоям в сети, а также использовать одинаковую функцию активации и стратегию инициализации во всех скрытых слоях, то можете обнаружить, что повторяете те же самые аргументы. В итоге получается неуклюжий и подверженный ошибкам код. Чтобы избежать этого, вы можете попытать-

ся провести рефакторинг кода с целью применения циклов. Еще один вариант предусматривает использование функции `functools.partial()` из Python, которая позволяет создавать тонкую оболочку для любого вызываемого объекта с рядом стандартных значений аргументов:

```
from           import partial
RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l2(0.01))

model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    RegularizedDense(300),
    RegularizedDense(100),
    RegularizedDense(10, activation="softmax",
                     kernel_initializer="glorot_uniform")
])
```

## Отключение

Отключение (*dropout*) является одним из наиболее популярных методик регуляризации для глубоких нейронных сетей. Оно было предложено Джеком Хинтоном и др. в статье 2012 года (<https://homl.info/64>)<sup>23</sup> и дополнительное детализировано Нитишем Шриваставой и др. в статье 2014 года (<https://homl.info/65>)<sup>24</sup>. Отключение доказало свою высокую успешность: даже современные нейронные сети получают рост правильности в 1–2% просто за счет добавления отключения. Такой рост может не выглядеть большим, но когда модель уже обладает 95%-ной правильностью, 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 (включая входные, но всегда исключая выходные нейроны) имеет вероятность  $p$  быть временно “отключенным”, так что он будет полностью игнорироваться в течение этого шага обучения, но может стать активным во время

<sup>23</sup> Джек Хинтон и др., *Improving neural networks by preventing co-adaptation of feature detectors* (Улучшение нейронных сетей путем предотвращения совместной адаптации детекторов признаков), препринт arXiv:1207.0580 (2012 г.).

<sup>24</sup> Нитиш Шривастава и др., *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* (Отключение: простой способ предотвращения переобучения нейронных сетей), *Journal of Machine Learning Research* 15 (2014 г.): с. 1929–1958.

следующего шага (рис. 11.9). Гиперпараметр  $p$  называется долей отключения (*dropout rate*) и обычно устанавливается между 10% и 50%: ближе к 20–30% в рекуррентных нейронных сетях (см. главу 15) и ближе к 40–50% в сверточных нейронных сетях (см. главу 14). После обучения нейроны больше не отключаются. На этом все (кроме технических деталей, которые мы вскоре обсудим).

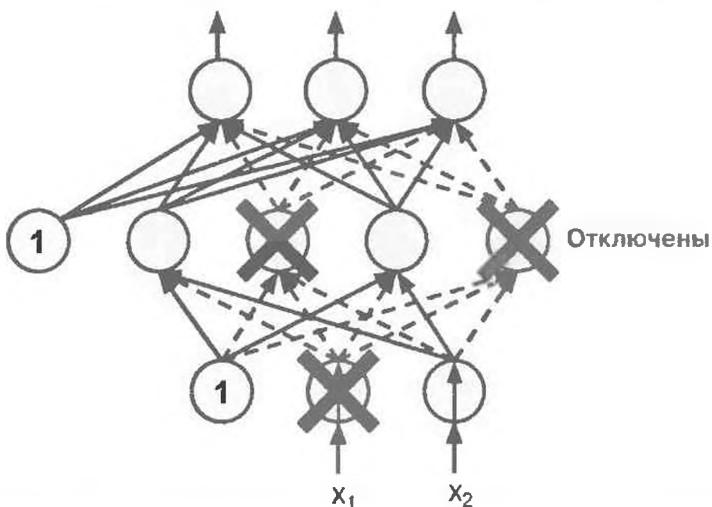


Рис. 11.9. При регуляризации на основе отключения на каждой итерации обучения “отключается” случайное подмножество из всех нейронов в одном или большем числе слоев (кроме выходного); на этой итерации такие нейроны выдают 0 (представленные пунктирными линиями со стрелками)

Поначалу несколько удивляет, что такая весьма деструктивная методика вообще работает. Станет ли компания функционировать 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, обучен-

ные 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 в сумме имеется  $2^N$  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 по существу было обучено 10 000 разных нейронных сетей (каждая со всего лишь одним обучающим образцом). Очевидно, что такие нейронные сети не являются независимыми, т.к. разделяют многие свои веса, но все-таки они все разные. Результатирующую нейронную сеть можно рассматривать как усредненный ансамбль всех этих более мелких нейронных сетей.



На практике отключение обычно можно применять только к нейронам в верхних 1–3 слоях (кроме выходного слоя).

Есть одна небольшая, но важная техническая деталь. Предположим, что  $p = 50\%$ ; в таком случае 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. Как правило, после обучения мы должны умножить вес каждой входной связи на вероятность сохранения ( $1 - p$ ). В качестве альтернативы мы можем поделить выход каждого нейрона на вероятность сохранения во время обучения (такие альтернативы не полностью эквивалентны, но работают одинаково хорошо).

Чтобы реализовать отключение с помощью Keras, вы можете использовать слой `keras.layers.Dropout`. Во время обучения он случайным образом отключает некоторые входы (устанавливая их в 0) и делит оставшиеся входы на вероятность сохранения. После обучения он вообще ничего не делает, а просто передает входы следующему слою. В следующем входе регуляризация

на основе отключения применяется перед каждым слоем Dense, используя долю отключения 0.2:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(300, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(100, activation="elu",
                      kernel_initializer="he_normal"),
    keras.layers.Dropout(rate=0.2),
    keras.layers.Dense(10, activation="softmax")
])
```



Поскольку отключение активно только во время обучения, сравнение потери при обучении и потери при проверке может вводить в заблуждение. В частности, модель может переобучаться обучающим набором, но иметь подобные потери при обучении и при проверке. Таким образом, обязательно оцените потерю при обучении без отключения (скажем, после обучения).

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. Кроме того, во многих современных архитектурах отключение применяется только после последнего скрытого слоя, так что вы можете попробовать поступить аналогично, если полное отключение является слишком сильным.

Отключение имеет тенденцию значительно замедлять схождение, но при надлежащей подстройке оно обычно приводит к гораздо лучшей модели. Таким образом, в большинстве ситуаций отключение стоит потраченного времени и усилий.



Если вы хотите регуляризовать самонормализующуюся нейронную сеть, основанную на функции активации SELU (как обсуждалось ранее), тогда должны использовать *альфа-отключение* (*alpha dropout*): разновидность отключения, которая предохраняет среднюю величину и стандартное отклонение своих входов (оно было представлено в той же статье, что и SELU, т.к. обычное отключение нарушило бы самонормализацию).

## Отключение Монте-Карло

Ярин Галь и Зубин Гахрамани в своей статье 2016 года (<https://homl.info/mcdropout>)<sup>25</sup> добавили еще несколько веских причин для применения отключения.

- Во-первых, в статье установлена глубокая связь между сетями с отключением (т.е. нейронными сетями, содержащими слой Dropout перед каждым слоем весов) и приближенным байесовским выводом (*Bayesian inference*)<sup>26</sup>, что дает отключению прочное математическое обоснование.
- Во-вторых, авторы представили мощный прием, называемый *отключением Монте-Карло (Monte Carlo (MC) Dropout)*, или *отключением MC Dropout*, который способен повысить эффективность любой обученной модели с отключением без необходимости в ее повторном обучении или даже изменении, обеспечивает гораздо лучшую меру неопределенности модели и также является удивительно простым в реализации.

Если все это выглядит как реклама “одного странного трюка”, тогда взгляните на следующий код. В нем приведена полная реализация отключения MC Dropout, повышающая эффективность обученной ранее модели с отключением без ее повторного обучения:

```
y_probas = np.stack([model(X_test_scaled, training=True)
                      for sample in range(100)])
y_proba = y_probas.mean(axis=0)
```

Мы просто вырабатываем 100 прогнозов на испытательном наборе, устанавливая `training=True` для гарантии того, что слой Dropout активен, и укладываем прогнозы в стопку. Так как отключение активно, все прогнозы будут разными. Вспомните, что `predict()` возвращает матрицу с одной строкой на образец и одним столбцом на класс. Поскольку в испытательном наборе есть 10 000 образцов и 10 классов, матрица имеет форму [10000, 10]. Мы укладываем в стопку 100 таких матриц, а потому `y_probas` является

<sup>25</sup> Ярин Галь и Зубин Гахрамани, *Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning* (Отключение как байесовская аппроксимация: представление неопределенности модели в глубоком обучении), *Proceedings of the 33rd International Conference on Machine Learning* (2016 г.): с. 1050–1059.

<sup>26</sup> В частности, они показали, что обучение сети с отключением математически эквивалентно приближенному байесовскому выводу в специфическом типе вероятностной модели, который называется глубоким гауссовым процессом (*Deep Gaussian Process*).

массивом с формой [100, 10000, 10]. В результате усреднения по первому измерению (axis=0) мы получаем y\_proba, массив с формой [10000, 10], как получили бы с помощью единственного прогноза. Вот и все! Усреднение по множеству прогнозов с активным отключением дает нам оценку Монте-Карло, которая в целом более надежна, чем результат единственного прогноза с неактивным отключением. Например, давайте посмотрим на прогноз модели для первого образца в испытательном наборе при неактивном отключении:

```
>>> np.round(model.predict(X_test_scaled[:1]), 2)
array([0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0.99],
      dtype=float32)
```

Кажется, модель почти уверена в том, что изображение относится к классу 9 (башмаки). Стоит ли доверять ей? Неужели так мало места для сомнений? Сравним это с прогнозами, которые вырабатываются при активном отключении:

```
>>> np.round(y_probas[:, :1], 2)
array([[0. , 0. , 0. , 0. , 0. , 0.14, 0. , 0.17, 0. , 0.68],
       [0. , 0. , 0. , 0. , 0. , 0.16, 0. , 0.2 , 0. , 0.64],
       [0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0.01, 0. , 0.97]],
      [...])
```

Результат сообщает нам совсем другую историю: по всей видимости, когда мы активизируем отключение, модель больше не будет уверенной. Похоже, что она по-прежнему отдает предпочтение классу 9, но иногда колеблется с классами 5 (сандалии) и 7 (кроссовки), что имеет смысл, т.к. все они представляют обувь. Усреднив по первому измерению, мы получаем следующие прогнозы с отключением MC Dropout:

```
>>> np.round(y_proba[:1], 2)
array([0. , 0. , 0. , 0. , 0. , 0.22, 0. , 0.16, 0. , 0.62]),
      dtype=float32)
```

Модель все еще полагает, что изображение относится к классу 9, но только с 62%-ным доверием, что выглядит намного более рациональным, чем 99%. Вдобавок полезно знать, какие другие классы модель считает вероятными. Можно также взглянуть на стандартное отклонение оценок вероятности (<https://xkcd.com/2110>):

```
>>> y_std = y_probas.std(axis=0)
>>> np.round(y_std[:1], 2)
```

```
array([0. , 0. , 0. , 0. , 0. , 0.28, 0. , 0.21, 0.02, 0.32]),  
      dtype=float32)
```

В оценках вероятности очевидно наличие довольно высокой дисперсии: если строится чувствительная к рискам система (скажем, медицинская или финансовая), то к такому неуверенному прогнозу возможно пришлось бы относиться с крайней осторожностью. Его определенно нельзя трактовать как прогноз с 99%-ной уверенностью. Кроме того, правильность модели немного повысилась с 86.8 до 86.9:

```
>>> accuracy = np.sum(y_pred == y_test) / len(y_test)  
>>> accuracy  
0.8694
```



Используемое количество образцов Монте-Карло (100 в рассмотренном примере) является гиперпараметром, который можно подстраивать. Чем он выше, тем более точными будут прогнозы и их оценки неопределенности. Однако если вы его удвоите, то время вывода тоже удвоится. Более того, выше определенного количества образцов вы заметите лишь незначительное улучшение. Таким образом, ваша работа состоит в том, чтобы отыскать подходящий для приложения компромисс между задержкой и правильностью.

Если ваша модель содержит другие слои, которые ведут себя особым образом во время обучения (такие как BatchNormalization), тогда вы не должны принудительно применять режим обучения, как только что делалось. Взамен вам потребуется заменить слои Dropout следующим классом MCDropout<sup>27</sup>:

```
class (keras.layers.Dropout):  
    def call(self, inputs):  
        return super().call(inputs, training=True)
```

Здесь мы просто создаем подкласс класса Dropout и переопределяем метод call(), чтобы принудительно установить его аргумент training в True (см. главу 12). Подобным образом вы могли бы определить класс MCAlphaDropout, создав подкласс класса AlphaDropout. Если вы созда-

<sup>27</sup> Этот класс MCDropout будет работать со всеми API-интерфейсами Keras API, в том числе с API-интерфейсом Sequential. Если вас не интересует API-интерфейс Functional или Subclassing, то вы не обязаны создавать класс MCDropout; можете создать обычный слой Dropout и обращаться к нему с training=True.

те модель с нуля, то все сводится только к использованию MCDropout, а не Dropout. Но если у вас есть модель, уже обученная с применением Dropout, тогда придется создать новую модель, которая идентична существующей модели помимо того, что она заменяет слои Dropout классом MCDropout, и затем скопировать веса существующей модели в новую модель.

Короче говоря, отключение MC Dropout — фантастическая методика, которая увеличивает правильность моделей с отключением и обеспечивает лучшие оценки неопределенности. И, конечно же, поскольку оно представляет собой обычное отключение во время обучения, MC Dropout также действует в качестве регуляризатора.

## Регуляризация на основе max-нормы

Еще один прием регуляризации, популярный в нейронных сетях, называется *регуляризацией на основе max-нормы*: для каждого нейрона она ограничивает веса  $w$  входящих связей, так что  $\|w\|_2 \leq r$ , где  $r$  — гиперпараметр max-нормы, а  $\|\cdot\|_2$  — норма  $\ell_2$ .

Регуляризация на основе max-нормы не добавляет член потери регуляризации к общей функции потерь. Взамен он обычно реализуется путем вычисления  $\|w\|_2$  после каждого шага обучения и при необходимости масштабирования  $w$  ( $w \leftarrow w \frac{r}{\|w\|_2}$ ).

Сокращение  $r$  увеличивает величину регуляризации и помогает ослабить переобучение. Регуляризация на основе max-нормы также может содействовать смягчению проблем нестабильных градиентов (если вы не используете пакетную нормализацию).

Чтобы реализовать регуляризацию на основе max-нормы посредством Keras, установите аргумент `kernel_constraint` каждого скрытого слоя в ограничение `max_norm()` с соответствующим максимальным значением:

```
keras.layers.Dense(100, activation="elu",
                   kernel_initializer="he_normal",
                   kernel_constraint=keras.constraints.max_norm(1.))
```

После каждой итерации обучения метод `fit()` модели будет обращаться к объекту, возвращенному `max_norm()`, передавая ему веса слоя и получая обратно масштабированные веса, которыми затем заменяются веса слоя. Как вы увидите в главе 12, при необходимости можно определять собственную функцию ограничения и применять ее в качестве `kernel_constraint`. Вы также можете ограничивать члены смещения, устанавливая аргумент `bias_constraint`.

Функция `max_norm()` принимает аргумент `axis`, по умолчанию устанавливаемый в 0. Слой `Dense` обычно имеет веса с формой [количество входов, количество нейронов], поэтому использование `axis=0` означает, что ограничение на основе max-нормы будет применяться независимо к весовому вектору каждого нейрона. Если вы хотите использовать max-норму со сверточными слоями (см. главу), тогда позаботьтесь о надлежащей установке аргумента `axis` ограничения `max_norm()` (обычно `axis=[0, 1, 2]`).

## Резюме и практические рекомендации

В настоящей главе мы раскрыли широкий диапазон методик, и вас может интересовать, какие из них должны использоваться. Все зависит от задачи, и пока четкого единодушия в данном вопросе нет, но я выяснил, что конфигурация из табл. 11.3 работает в большинстве случаев, не требуя большого объема подстройки гиперпараметров. Тем не менее, не считайте указанные стандартные установки жесткими правилами!

**Таблица 11.3. Стандартная конфигурация сетей DNN**

Гиперпараметр	Стандартное значение
Инициализация ядра	Инициализация Хе
Функция активации	ELU
Нормализация	Отсутствует, если неглубокая; пакетная нормализация, если глубокая
Регуляризация	Раннее прекращение (плюс при необходимости регуляризация $\ell_2$ )
Оптимизатор	Моментная оптимизация (либо RMSProp или Nadam)
График для скорости обучения	Одного цикла

Если сеть представляет собой простую стопку плотных слоев, то она может самонормализоваться, и придется применять конфигурацию из табл. 11.4.

**Таблица 11.4. Конфигурация для самонормализуемой сети**

Гиперпараметр	Стандартное значение
Инициализация ядра	Инициализация Лекуна
Функция активации	SELU
Нормализация	Отсутствует (самонормализация)
Регуляризация	Альфа-отключение, если необходимо
Оптимизатор	Моментная оптимизация (либо RMSProp или Nadam)
График для скорости обучения	Одного цикла

Не забудьте нормализовать входные признаки! Вы также должны пытаться повторно задействовать части заранее обученной нейронной сети, если удалось отыскать сеть, которая решает похожую задачу, или использовать предварительное обучение без учителя, если есть много непомеченных данных, либо применить предварительное обучение на вспомогательной задаче при наличии множества помеченных данных для похожей задачи.

Хотя предшествующие рекомендации должны охватывать большинство случаев, существует несколько исключений.

- Если вам нужна разреженная модель, тогда можете использовать регуляризацию  $\ell_1$  (и дополнительно обнулить крошечные веса после обучения). Когда необходима даже более разреженная модель, то можете применять комплект инструментов для оптимизации моделей (TF-MOT). Он нарушит самонормализацию, поэтому в данном случае вам придется использовать стандартную конфигурацию.
- Если вас интересует модель с низкой задержкой (вырабатывающая молниеносные прогнозы), тогда может понадобиться применить меньше слоев, поместить слои пакетной нормализации внутрь предыдущих слоев и возможно использовать более быструю функцию активации, такую как ReLU с утечкой или просто ReLU. Также поможет наличие разреженной модели. Наконец, вы можете уменьшить точность чисел с плавающей точкой с 32 бит до 16 или даже 8 бит (см. раздел “Развертывание модели на мобильном или встроенном устройстве” главы 19). Опять-таки обратитесь к TF-MOT.
- Если вы строите чувствительную к рискам систему или задержка вывода в приложении не особо важна, тогда можете применять отключение MC Dropout, чтобы повысить эффективность и получать более надежные оценки вероятности вместе с оценками неопределенности.

Располагая такими рекомендациями, вы готовы обучать очень глубокие сети! Я надеюсь, что теперь вы уверены в том, что сумеете пройти довольно длинный путь, используя только Keras. Однако может наступить время, когда вам понадобится иметь даже больший контроль; например, чтобы написать специальную функцию потерь или подстроить алгоритм обучения. В таких случаях необходимо применять низкоуровневый API-интерфейс TensorFlow, как будет показано в следующей главе.

# Упражнения

1. Можно ли инициализировать все веса одним и тем же значением при условии, что оно выбрано случайно с использованием инициализации Хе?
2. Можно ли инициализировать члены смещения значением 0?
3. Назовите три преимущества функции активации SELU по сравнению с ReLU.
4. В каких случаях вы бы использовали каждую из следующих функций активации: SELU, ReLU с утечкой (и разновидности), ReLU,  $\tanh$ , логистическую и многопеременную?
5. Что может произойти, если во время применения оптимизатора SGD вы установите гиперпараметр `momentum` в значение, слишком близкое к 1 (скажем, 0.99999)?
6. Назовите три способа получения разреженной модели.
7. Замедляет ли отключение процесс обучения? Замедляет ли оно вывод (т.е. выработку прогнозов на новых образцах)? Что насчет отключения MC Dropout?
8. Попрактикуйтесь в обучении глубокой нейронной сети на наборе данных с изображениями CIFAR10.
  - а) Постройте сеть DNN с 20 скрытыми слоями по 100 нейронов в каждом (это чересчур много, но такова цель упражнения). Выберите инициализацию Хе и функцию активации ELU.
  - б) Используя оптимизацию Nadam и раннее прекращение, обучите сеть на наборе данных CIFAR10, который можете загрузить с помощью `keras.datasets.cifar10.load_data()`. Набор данных CIFAR10 состоит из 60 000 цветных изображений размером  $32 \times 32$  пикселя (50 000 для обучения, 10 000 для испытаний) с 10 классами, поэтому вам будет нужен многопеременный выходной слой с 10 нейронами. Не забывайте о нахождении правильной скорости обучения всякий раз, когда изменяете архитектуру модели или гиперпараметры.
  - в) Теперь попробуйте добавить пакетную нормализацию и сравните кривые обучения: происходит ли схождение быстрее, чем раньше? Дает ли это лучшую модель? Как это влияет на скорость обучения?

- г) Попробуйте поменять пакетную нормализацию на SELU и внесите необходимые корректировки, чтобы обеспечить самонормализацию сети (т.е. стандартизируйте входные признаки, используйте инициализацию Лекуна с нормальным распределением, удостоверьтесь в том, что сеть DNN содержит только последовательность плотных слоев и т.д.).
- д) Попробуйте регуляризовать модель с помощью альфа-отключения. Затем, не проводя повторное обучение модели, посмотрите, сумете ли вы достигнуть лучшей правильности с применением отключения MC Dropout.
- е) Заново обучите модель, используя график одного цикла, и посмотрите, улучшились ли скорость обучения и правильность модели.

Решения приведенных упражнений доступны в приложении А.

# Специальные модели и обучение с помощью TensorFlow

До сих пор мы использовали только высокоуровневый API-интерфейс TensorFlow, `tf.keras`, но он уже дал нам довольно многое: мы строили нейронные сети с разнообразными архитектурами, в том числе сети для регрессии и классификации, сети Wide & Deep и самонормализуемые сети с применением методик всех видов, таких как пакетная нормализация, отключение и графики для скорости обучения. На самом деле 95% сценариев использования, с которыми вы столкнетесь, не будут требовать ничего кроме `tf.keras` (и `tf.data`; см. главу 13). Но настало время погрузиться глубже в библиотеку TensorFlow и взглянуть на ее низкоуровневый API-интерфейс для языка Python (<https://homl.info/tf2api>). Это будет полезно, когда вам необходим дополнительный контроль для написания специальных функций потерь, специальных метрик, слоев, моделей, инициализаторов, регуляризаторов, ограничений весов и т.д. Возможно, вам даже понадобится контролировать сам цикл обучения, например, для применения особых трансформаций или ограничений к градиентам (помимо простого их отсечения) или для использования множества оптимизаторов в разных частях сети. В главе мы раскроем все случаи подобного рода и также посмотрим, каким образом можно повысить эффективность ваших специальных моделей и алгоритмов обучения с применением средства автоматической генерации графов TensorFlow. Но сначала давайте проведем краткий тур по TensorFlow.



Версия TensorFlow 2.0 (бета) была выпущена в июне 2019 года, гораздо упростив использование TensorFlow. В первом издании книги применялась библиотека TF 1, тогда как в настоящем изда-  
нии используется TF 2.

# Краткий тур по TensorFlow

Как вам известно, TensorFlow является мощной библиотекой для численных расчетов, которая особенно хорошо подходит и точно подогнана под область крупномасштабного машинного обучения (хотя ее можно применять и для чего-то другого, где требуются интенсивные вычисления). Она была разработана командой специалистов Google Brain и приводит в действие многие крупномасштабные службы Google, такие как Google Cloud Speech (распознавание речи с помощью облака Google), Google Photos (Google Фото) и Google Search (Google Поиск). В ноябре 2015 года ее исходный код стал открытым, и теперь она считается самой популярной библиотекой для глубокого обучения (в плане упоминания в статьях, внедрения в компаниях, звезд на GitHub и т.д.). Библиотека TensorFlow используется в бесчисленных проектах для всех видов задач МО, включая классификацию изображений, обработку естественного языка, выдачу рекомендаций и прогнозирование временных рядов.

Так что же предлагает TensorFlow? Ниже приведена сводка.

- Ее ядро очень похоже на NumPy, но имеет поддержку графических процессоров.
- Она поддерживает распределенные вычисления (среди множества устройств и серверов).
- Она включает своего рода *JIT-компилятор* (*just-in-time — оперативный*), который позволяет оптимизировать вычисления в отношении скорости и расхода памяти. Для этого из функции Python извлекается вычислительный граф, затем он оптимизируется (скажем, за счет отсечения неиспользуемых узлов) и в заключение эффективным образом выполняется (например, путем автоматического выполнения независимых операций в параллельном режиме).
- Вычислительные графы могут экспортироваться в переносимый формат, так что модель TensorFlow можно обучать в одной среде (скажем, применяя Python в Linux) и запускать в другой (например, используя Java на устройстве Android).
- Она реализует автоматическое дифференцирование (см. главу 10 и приложение Г) и предоставляет ряд превосходных оптимизаторов, таких как RMSProp и Nadam (см. главу 11), поэтому можно легко доводить до минимума все виды функций потерь.

Библиотека TensorFlow предлагает намного больше возможностей, построенных поверх основных средств: самым важным, конечно же, является `tf.keras`<sup>1</sup>, но в ней также есть операции загрузки и предварительной обработки данных (`tf.data`, `tf.io` и т.д.), операции обработки изображений (`tf.image`), операции обработки сигналов (`tf.signal`) и многие другие (общее представление об API-интерфейсе для Python библиотеки TensorFlow можно получить, взглянув на рис. 12.1).



Мы рассмотрим многие пакеты и функции API-интерфейса TensorFlow, но раскрыть их здесь все невозможно, так что вам действительно нужно потратить некоторое время на ознакомление с полным API-интерфейсом; вы обнаружите, что он очень развит и хорошо документирован.



Рис. 12.1. Общий вид API-интерфейса для Python библиотеки TensorFlow

<sup>1</sup> TensorFlow включает еще один API-интерфейс для глубокого обучения под названием *Estimators API*, но команда создателей TensorFlow рекомендует использовать вместо него `tf.keras`.

На самом низком уровне каждая операция TensorFlow реализована с применением крайне эффективного кода на C++<sup>2</sup>. Многие операции имеют многочисленные реализации, называемые *ядрами (kernel)*: каждое ядро предназначено для специфического типа устройства, такого как центральный процессор, графический процессор или даже *тензорный процессор (tensor processing unit — TPU)*. Как вам может быть известно, графические процессоры способны значительно ускорить вычисления, расщепляя их на множество мелких порций и выполняя параллельно во множестве потоков графического процессора. Тензорные процессоры еще быстрее: они представляют собой прикладные (ASIC) микросхемы, построенные специально для операций глубокого обучения<sup>3</sup> (мы обсудим использование TensorFlow с графическими или тензорными процессорами в главе 19).

На рис. 12.2 показана архитектура TensorFlow. Большую часть времени ваш код будет применять высоконивневые API-интерфейсы (особенно `tf.keras` и `tf.data`); но когда вам необходима более высокая гибкость, то вы можете использовать низкоуровневый API-интерфейс для Python, обрабатывая тензоры напрямую. Обратите внимание, что доступны также API-интерфейсы для других языков. В любом случае исполняющий механизм TensorFlow позаботится об эффективном выполнении операций даже среди множества устройств и машин, если вы сообщите ему.

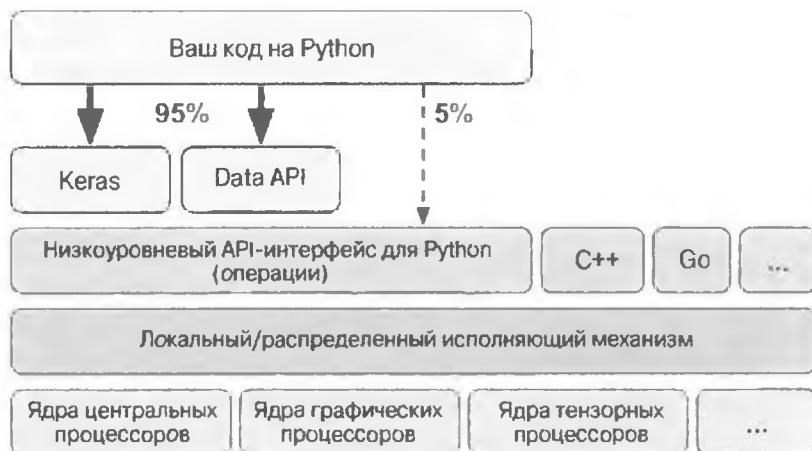


Рис. 12.2. Архитектура TensorFlow

<sup>2</sup> Если вам когда-либо понадобится (что вряд ли), то вы можете написать собственные операции, используя API-интерфейс для C++.

<sup>3</sup> Дополнительные сведения о тензорных процессорах и о том, как они работают, доступны по ссылке <https://homl.info/tpus>.

TensorFlow запускается не только в средах Windows, Linux и macOS, но также на мобильных устройствах (с применением *TensorFlow Lite*), включая iOS и Android (см. главу 19). Если вы не хотите использовать API-интерфейс для Python, то имеются API-интерфейсы для C++, Java, Go и Swift. Существует даже реализация для JavaScript под называнием *TensorFlow.js*, которая делает возможным запуск ваших моделей прямо в браузере.

Следует отметить, что TensorFlow является чем-то большим, нежели просто библиотекой. TensorFlow находится в центре обширной экосистемы библиотек. Прежде всего, есть TensorBoard для визуализации (см. главу 10). Затем имеется *TensorFlow Extended* (TFX) (<https://tensorflow.org/tfx>) — набор библиотек, построенных Google в целях запуска в производство проектов TensorFlow: он включает инструменты для проверки достоверности данных, предварительной обработки, анализа моделей и обслуживания (посредством TF Serving; см. главу 19). Платформа *TensorFlow Hub* от Google предоставляет легкий способ загрузки и многократного применения заранее обученных нейронных сетей. Вы также можете получить много архитектур нейронных сетей, часть которых предварительно обучена, в саду моделей TensorFlow (<https://github.com/tensorflow/models/>). Проверяйте ресурсы TensorFlow (<https://www.tensorflow.org/resources>) и <https://github.com/jtoy/awesome-tensorflow> на предмет дополнительных проектов, основанных на TensorFlow. Вы обнаружите сотни проектов TensorFlow на GitHub, поэтому часто проще найти существующий код для всего, что вы пытаетесь сделать.



Все больше и больше статей по МО выпускаются вместе с их реализациями, а временами даже с заранее обученными моделями. Их можно легко найти на веб-сайте <https://paperswithcode.com/>.

Последнее, но не менее важное: библиотека TensorFlow имеет преданную команду увлеченных и предупредительных разработчиков, а также крупное сообщество, содействующее ее улучшению. Задавать вопросы технического характера необходимо на веб-сайте <http://stackoverflow.com/>, снабжая их метками *tensorflow* и *python*. Регистрировать ошибки и запросы средств можно через GitHub (<https://github.com/tensorflow/tensorflow>). Для участия в общих обсуждениях присоединяйтесь к группе Google (<https://goolm.info/41>).

А теперь приступим к написанию кода!

# Использование TensorFlow подобно NumPy

API-интерфейс TensorFlow вращается вокруг *тензоров* (*tensor*), которые *перетекают* (*flow*) из операции в операцию — отсюда и название *TensorFlow*. Тензор обычно представляет собой многомерный массив (в частности похожий на `ndarray` из `NumPy`), но он также может хранить скаляр (простое значение вроде 42). Тензоры будут важны, когда мы создаем специальные функции издержек, специальные метрики, специальные слои и многое другое, так что давайте выясним, как их создавать и манипулировать ими.

## Тензоры и операции

Создать тензор можно с помощью `tf.constant()`. Например, вот тензор, представляющий матрицу с двумя строками и тремя столбцами чисел с плавающей точкой:

```
>>> tf.constant([[1., 2., 3.], [4., 5., 6.]])    # матрица
<tf.Tensor: id=0, shape=(2, 3), dtype=float32, numpy=
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)>
>>> tf.constant(42)    # скаляр
<tf.Tensor: id=1, shape=(), dtype=int32, numpy=42>
```

Подобно `ndarray` объект `tf.Tensor` имеет форму и тип данных (`dtype`):

```
>>> t = tf.constant([[1., 2., 3.], [4., 5., 6.]])
>>> t.shape
TensorShape([2, 3])
>>> t.dtype
tf.float32
```

Индексирование работает во многом похоже на то, как оно выполняется в `NumPy`:

```
>>> t[:, 1:]
<tf.Tensor: id=5, shape=(2, 2), dtype=float32, numpy=
array([[2., 3.],
       [5., 6.]], dtype=float32)>
>>> t[..., 1, tf.newaxis]
<tf.Tensor: id=15, shape=(2, 1), dtype=float32, numpy=
array([[2.],
       [5.]], dtype=float32)>
```

Что еще более важно, доступны все виды операций с тензорами:

```
>>> t + 10
<tf.Tensor: id=18, shape=(2, 3), dtype=float32, numpy=
array([[11., 12., 13.],
       [14., 15., 16.]], dtype=float32)>
>>> tf.square(t)
<tf.Tensor: id=20, shape=(2, 3), dtype=float32, numpy=
array([[ 1.,   4.,   9.],
       [16., 25., 36.]], dtype=float32)>
>>> t @ tf.transpose(t)
<tf.Tensor: id=24, shape=(2, 2), dtype=float32, numpy=
array([[14., 32.],
       [32., 77.]], dtype=float32)>
```

Обратите внимание, что выражение `t + 10` эквивалентно вызову `tf.add(t, 10)` (на самом деле Python вызывает магический метод `t.__add__(10)`, который просто вызывает `tf.add(t, 10)`). Также поддерживаются другие операции наподобие `-` и `*`. Операция `@` появилась в Python 3.5 и предназначена для перемножения матриц: она эквивалентна вызову функции `tf.matmul()`.

Вы обнаружите все необходимые базовые математические операции (`tf.add()`, `tf.multiply()`, `tf.square()`, `tf.exp()`, `tf.sqrt()` и т.д.) и большинство операций, которые можно найти в NumPy (например, `tf.reshape()`, `tf.squeeze()`, `tf.tile()`). Некоторые функции имеют не такое имя, как в NumPy; скажем, `tf.reduce_mean()`, `tf.reduce_sum()`, `tf.reduce_max()` и `tf.math.log()` являются эквивалентами `np.mean()`, `np.sum()`, `np.max()` и `np.log()`. Отличия в именах часто обусловлены вескими причинами. Например, в TensorFlow вы обязаны записывать `tf.transpose(t)`; нельзя просто написать `t.T` как в NumPy. Причина в том, что функция `tf.transpose()` делает не в точности то же, что и атрибут `T` в NumPy: в TensorFlow создается новый тензор с собственной копией транспонированных данных, тогда как в NumPy атрибут `t.T` является транспонированным представлением тех же самых данных. Аналогично операция `tf.reduce_sum()` названа так оттого, что ее ядро графического процессора (т.е. реализация для графического процессора) применяет алгоритм сокращения (`reduce`), который вовсе не гарантирует определенный порядок добавления элементов: поскольку 32-битные числа с плавающей точкой обладают ограниченной точностью, результат может незначительно изменяться при каждом вызове операции. То же самое справедливо для `tf.reduce_mean()` (но, разумеется, операция `tf.reduce_max()` является детерминированной).



Многие функции и классы имеют псевдонимы. Например, `tf.add()` и `tf.math.add()` — одна и та же функция. Это позволяет TensorFlow располагать лаконичными именами для большинства распространенных операций<sup>4</sup>, одновременно сохраняя хорошо организованные пакеты.

## Низкоуровневый API-интерфейс Keras

Библиотека Keras имеет собственный низкоуровневый API-интерфейс, находящийся в `keras.backend`. Он включает такие функции, как `square()`, `exp()` и `sqrt()`. В `tf.keras` эти функции обычно просто вызывают соответствующие операции TensorFlow. Если вы хотите, чтобы код был переносимым в другие реализации Keras, то должны использовать упомянутые функции Keras. Однако они покрывают лишь подмножество всех функций, доступных в TensorFlow, а потому в книге мы будем применять операции TensorFlow напрямую. Ниже приведен простой пример использования API-интерфейса `keras.backend`, которому обычно для краткости назначается имя K:

```
>>> from           import keras
>>> K = keras.backend
>>> K.square(K.transpose(t)) + 10
<tf.Tensor: id=39, shape=(3, 2), dtype=float32, numpy=
array([[11., 26.],
       [14., 35.],
       [19., 46.]], dtype=float32)>
```

## Тензоры и NumPy

Тензоры хорошо работают с NumPy: вы можете создавать тензор из массива NumPy и наоборот. Вы можете даже применять операции TensorFlow к массивам NumPy и операции NumPy к тензорам:

```
>>> a = np.array([2., 4., 5.])
>>> tf.constant(a)
<tf.Tensor: id=111, shape=(3,), dtype=float64, numpy=array([2., 4., 5.])>
```

<sup>4</sup> Заметным исключением является функция `tf.math.log()`, которая часто применяется, но не имеет псевдонима вроде `tf.log()` (т.к. возникла бы путаница с функцией регистрации в журнале).

```
>>> t.numpy()    # или np.array(t)
array([[1., 2., 3.],
       [4., 5., 6.]], dtype=float32)
>>> tf.square(a)
<tf.Tensor: id=116, shape=(3,), dtype=float64, numpy=
array([4., 16., 25.])>
>>> np.square(t)
array([[ 1.,  4.,  9.],
       [16., 25., 36.]], dtype=float32)
```



Обратите внимание, что NumPy по умолчанию использует 64-битную точность, в то время как TensorFlow — 32-битную. Причина в том, что 32-битной точности для нейронных сетей, как правило, более чем достаточно, вдобавок данные занимают меньше места в памяти и быстрее обрабатываются. Таким образом, при создании тензора из массива NumPy обязательно установите `dtype=tf.float32`.

## Преобразования типов

Преобразования типов способны значительно снизить производительность и могут легко остаться незамеченными, когда делаются автоматически. Чтобы избежать этого, библиотека TensorFlow не выполняет никаких преобразований типов автоматически: при попытке выполнить операцию на тензорах несочетимых типов она просто генерирует исключение. Скажем, вы не можете суммировать тензор с плавающей точкой и целочисленный тензор и даже не можете складывать тензоры с плавающей точкой длиной 32 и 64 бита:

```
>>> tf.constant(2.) + tf.constant(40)
Traceback[...] InvalidArgumentError[...] expected to be a float[...]
Трассировка[...] Недопустимый аргумент[...] ожидалось число
                                с плавающей точкой[...]

>>> tf.constant(2.) + tf.constant(40., dtype=tf.float64)
Traceback[...] InvalidArgumentError[...] expected to be a double[...]
Трассировка[...] Недопустимый аргумент[...] ожидалось число
                                с плавающей точкой двойной точности[...]
```

Поначалу исключение может слегка раздражать, но помните, что это во благо! И, конечно же, вы можете применять `tf.cast()`, когда действительно нуждаетесь в преобразовании типов:

```
>>> t2 = tf.constant(40., dtype=tf.float64)
>>> tf.constant(2.0) + tf.cast(t2, tf.float32)
<tf.Tensor: id=136, shape=(), dtype=float32, numpy=42.0>
```

## Переменные

Значения `tf.Tensor`, которые мы видели до сих пор, являются неизменяемыми: модифицировать их нельзя. Это значит, что мы не можем использовать обычные тензоры для реализации весов в нейронной сети, т.к. веса нуждаются в подстройке обратным распространением. Плюс остальные параметры тоже с течением времени могут требовать изменения (например, моментный оптимизатор отслеживает прошедшие градиенты). Нам необходима переменная `tf.Variable`:

```
>>> v = tf.Variable([[1., 2., 3.], [4., 5., 6.]])  
>>> v  
<tf.Variable 'Variable:0' shape=(2, 3) dtype=float32, numpy=  
array([[1., 2., 3.],  
       [4., 5., 6.]], dtype=float32)>
```

Объект `tf.Variable` действует почти как `tf.Tensor`: вы можете выполнять с ним те же самые операции, он хорошо работает с NumPy и в той же мере разборчив в отношении типов. Но его также можно модифицировать на месте с применением метода `assign()` (либо методов `assign_add()` и `assign_sub()`, которые увеличивают и уменьшают переменную на указанную величину). Кроме того, вы можете модифицировать индивидуальные элементы или срезы, используя метод `assign()` элемента или среза (прямое присваивание элементу работать не будет) либо методы `scatter_update()` или `scatter_nd_update()`:

```
v.assign(2 * v)           # => [[2., 4., 6.], [8., 10., 12.]]  
v[0, 1].assign(42)        # => [[2., 42., 6.], [8., 10., 12.]]  
v[:, 2].assign([0., 1.])   # => [[2., 42., 0.], [8., 10., 1.]]  
v.scatter_nd_update(indices=[[0, 0], [1, 2]], updates=[100., 200.])  
                           # => [[100., 42., 0.], [8., 10., 200.]]
```



На практике вам редко придется создавать переменные вручную, поскольку Keras предоставляет метод `add_weight()`, который позаботится об этом. Вдобавок параметры модели обычно будут обновляться напрямую оптимизаторами, так что обновлять переменные понадобится нечасто.

## Другие структуры данных

Библиотека TensorFlow поддерживает ряд других структур данных, включая описанные ниже (за дополнительными сведениями обращайтесь в тетрадь “Tensors and Operations” (Тензоры и операции) для настоящей главы и в приложение Е).

### *Разреженные тензоры (`tf.SparseTensor`)*

Эффективно представляют тензоры, содержащие главным образом нули.

Пакет `tf.sparse` содержит операции для разреженных тензоров.

### *Массивы тензоров (`tf.TensorArray`)*

Представляют собой списки тензоров. По умолчанию они имеют фиксированный размер, но могут дополнительно делаться динамическими. Все содержащиеся внутри тензоры обязаны иметь одинаковую форму и тот же самый тип данных.

### *Зубчатые тензоры (`tf.RaggedTensor`)*

Представляют статические списки из списков тензоров, где каждый тензор имеет одну и ту же форму и тип данных. Пакет `tf.ragged` содержит операции для зубчатых тензоров.

### *Строковые тензоры*

Это обычные тензоры типа `tf.string`. Они представляют байтовые строки, не строки Unicode, поэтому если вы создаете строковый тензор с применением строки Unicode (скажем, обычной строки наподобие "caf " из Python 3), то она автоматически будет закодирована в UTF-8 (например, `b"caf\xc3\xa9"`). В качестве альтернативы вы можете представлять строки Unicode, используя тензоры типа `tf.int32`, где каждый элемент представляет кодовую точку Unicode (скажем, `[99, 97, 102, 233]`). Пакет `tf.strings` (с `s` в конце) содержит операции для байтовых строк и строк Unicode (и также для преобразования одних в другие). Важно отметить, что тип `tf.string` является атомарным, т.е. его длина в форме тензора не присутствует. После его преобразования в тензор Unicode (тензор типа `tf.int32`, хранящий кодовые точки Unicode) длина появляется в форме.

## Множества

Представляются как обычные тензоры (или разреженные тензоры). Например, `tf.constant([[1, 2], [3, 4]])` представляет два множества `[1, 2]` и `[3, 4]`. В более общем случае каждое множество представлено вектором на последней оси тензора. Манипулировать множествами можно с применением операций из пакета `tf.sets`.

## Очереди

Сохраняют тензоры между множеством шагов. Библиотека TensorFlow предлагает разнообразные виды очередей: *простые очереди FIFO* (*First In, First Out* — “первым пришел — первым обслужен”; `FIFOQueue`), очереди, способные назначать приоритеты некоторым элементам (`PriorityQueue`), тасовать свои элементы (`RandomShuffleQueue`) и группировать элементы разных форм путем дополнения (`PaddingFIFOQueue`). Все упомянутые классы находятся в пакете `tf.queue`.

Имея в своем распоряжении тензоры, операции, переменные и различные структуры данных, теперь вы готовы настраивать свои модели и алгоритмы обучения!

## Настройка моделей и алгоритмов обучения

Давайте начнем с создания специальной функции потерь, что является простым и распространенным сценарием использования.

### Специальные функции потерь

Предположим, что вам нужна регрессионная модель, но ваш обучающий набор слегка зашумлен. Разумеется, вы начинаете с попытки очистить набор данных, удаляя или исправляя выбросы, но оказывается, что этого недостаточно; набор данных по-прежнему зашумлен. Какую функцию потерь вы должны применить? Среднеквадратическая ошибка могла бы слишком сильно штрафовать большие ошибки и привести к неточности модели. Средняя абсолютная ошибка штрафовала бы выбросы не настолько сильно, но обучение могло занять долгое время, чтобы сойтись, а обученная модель оказалась бы не особенно точной. Вероятно, это подходящий случай для использования потери Хьюбера (введенной в главе 10) вместо старой доброй ошибки MSE. В текущий момент потеря Хьюбера не является частью официального API-интерфейса Keras, но она доступна в `tf.keras` (просто создайте экзем-

пляя класса `keras.losses.Huber`). Но давайте притворимся, что потери Хьюбера нет: реализовать ее проще простого! Достаточно создать функцию, которая принимает в качестве аргументов метки и прогнозы и применяет операции TensorFlow для расчета потери каждого образца:

```
def huber_fn(y_true, y_pred):
    error = y_true - y_pred
    is_small_error = tf.abs(error) < 1
    squared_loss = tf.square(error) / 2
    linear_loss = tf.abs(error) - 0.5
    return tf.where(is_small_error, squared_loss, linear_loss)
```



Для большей эффективности вы должны использовать векторизованную реализацию, как в приведенном примере. Кроме того, если вы хотите извлечь преимущество из средств графов TensorFlow, то обязаны применять только операции TensorFlow.

Также предпочтительно возвращать тензор, содержащий одну потерю на образец, а не среднюю потерю. Таким образом, Keras сможет применять веса классов или веса образцов по запросу (см. главу 10).

Теперь созданную функцию потерь можно использовать при компиляции модели Keras и затем ее обучении:

```
model.compile(loss=huber_fn, optimizer="nadam")
model.fit(X_train, y_train, [...])
```

Вот и все! Во время обучения для каждого пакета библиотека Keras будет вызывать функцию `huber_fn()`, чтобы подсчитать потерю и задействовать ее при выполнении шага градиентного спуска. Вдобавок она будет отслеживать общую потерю с начала эпохи и отображать среднюю потерю. Но что происходит с этой специальной потерей, когда вы сохраняете модель?

## Сохранение и загрузка моделей, которые содержат специальные компоненты

Сохранение модели, содержащей специальную функцию потерь, нормально работает, т.к. Keras сохраняет имя функции. Всякий раз, когда вы загружаете модель, вам необходимо предоставить словарь, который сопоставляет имя функции с фактической функцией. В более общем случае при загрузке модели, содержащей специальные объекты, понадобится отобразить имена на объекты:

```
model = keras.models.load_model("my_model_with_a_custom_loss.h5",
                                custom_objects={"huber_fn": huber_fn})
```

В текущей реализации любая ошибка между  $-1$  и  $1$  считается “небольшой”. Но что, если вас интересует другой порог? Одно из решений предусматривает определение функции, которая создает сконфигурированную функцию потерь:

```
def create_huber(threshold=1.0):
    def huber_fn(y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < threshold
        squared_loss = tf.square(error) / 2
        linear_loss = threshold * tf.abs(error) - threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    return huber_fn

model.compile(loss=create_huber(2.0), optimizer="nadam")
```

К сожалению, когда вы сохраняете модель, значение `threshold` не будет сохранено. Таким образом, при загрузке модели вам придется указывать значение `threshold` (обратите внимание, что применяемым именем является `"huber_fn"`, которое представляет собой имя функции, переданной Keras, но не имя функции, созданной вами):

```
model = keras.models.load_model(
    "my_model_with_a_custom_loss_threshold_2.h5",
    custom_objects={"huber_fn": create_huber(2.0)})
```

Решить проблему можно за счет создания подкласса класса `keras.losses.Loss` и реализации его метода `get_config()`:

```
class MyLoss(keras.losses.Loss):
    def __init__(self, threshold=1.0, **kwargs):
        self.threshold = threshold
        super().__init__(**kwargs)
    def call(self, y_true, y_pred):
        error = y_true - y_pred
        is_small_error = tf.abs(error) < self.threshold
        squared_loss = tf.square(error) / 2
        linear_loss =
            self.threshold * tf.abs(error) - self.threshold**2 / 2
        return tf.where(is_small_error, squared_loss, linear_loss)
    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "threshold": self.threshold}
```



В текущий момент API-интерфейс Keras устанавливает, как создавать подклассы для определения только слоев, моделей, обратных вызовов и регуляризаторов. Если вы строите другие компоненты (такие как функции потерь, метрики, инициализаторы или ограничения), используя создание подклассов, то они могут не быть переносимыми в другие реализации Keras. Скорее всего, API-интерфейс Keras будет обновлен, чтобы разрешить создание подклассов для всех компонентов такого рода.

Давайте разберем код.

- Конструктор принимает `**kwargs` и передает аргументы конструктору родительского класса, который поддерживает стандартные гиперпараметры: имя функции потерь и алгоритм сокращения, применяемый для объединения потерь индивидуальных образцов. По умолчанию им является `"sum_over_batch_size"`, который означает, что потеря будет суммой потерь образцов, снабженных весами образцов, если они есть, и деленная на размер пакета (не на сумму весов, а потому получается *не взвешенное среднее*)<sup>5</sup>. Другими возможными значениями являются `"sum"` и `None`.
- Метод `call()` принимает метки и прогнозы, вычисляет все потери образцов и возвращает их.
- Метод `get_config()` возвращает словарь, сопоставляющий имя каждого гиперпараметра с его значением. Он сначала вызывает метод `get_config()` родительского класса, после чего добавляет новые гиперпараметры в этот словарь (обратите внимание на удобный синтаксис `{**x}`, появившийся в Python 3.5).

Затем вы можете использовать любой экземпляр класса `HuberLoss` при компиляции модели:

```
model.compile(loss=HuberLoss(2.), optimizer="nadam")
```

Когда вы сохраняете модель, вместе с ней будет сохраняться порог, и при загрузке модели понадобится лишь отобразить имя класса на сам класс:

```
model = keras.models.load_model(  
    "my_model_with_a_custom_loss_class.h5",  
    custom_objects={ "HuberLoss": HuberLoss })
```

<sup>5</sup> Использование взвешенного среднего было бы не особо хорошей идеей: в таком случае два образца с одинаковым весом, находящиеся в разных пакетах, оказывали бы разное воздействие на обучение, зависящее от общего веса каждого пакета.

Во время сохранения модели библиотека Keras вызывает метод `get_config()` экземпляра класса потери и сохраняет конфигурацию как данные JSON в файле HDF5. Когда модель загружается, Keras вызывает метод `from_config()` класса `HuberLoss`: этот метод реализован в базовом классе (`Loss`) и создает экземпляр класса, передавая `**config` его конструктору.

Вот и все, что касается потерь! Дело было не слишком сложным, не так ли? Столь же простыми являются специальные функции активации, инициализаторы, регуляризаторы и ограничения. Давайте взглянем на них.

## Специальные функции активации, инициализаторы, регуляризаторы и ограничения

Большинство функциональных средств Keras, таких как функции потерь, регуляризаторы, ограничения, инициализаторы, метрики, функции активации, слои и даже полные модели, могут настраиваться почти тем же самым способом. Большую часть времени вам будет нужно лишь написать простую функцию с подходящими входами и выходами. Ниже приведен пример специальной функции активации (эквивалента `keras.activations.softplus()` или `tf.nn.softplus()`), специального инициализатора Глоро (эквивалента `keras.initializers.glorot_normal()`), специального регуляризатора  $\ell_1$  (эквивалента `keras.regularizers.l1(0.01)`) и специального ограничения, которое гарантирует, что все веса положительны (эквивалента `keras.constraints.nonneg()` или `tf.nn.relu()`):

```
def my_softplus(z): # возвращаемое значение - просто tf.nn.softplus(z)
    return tf.math.log(tf.exp(z) + 1.0)
def my_glorot_initializer(shape, dtype=tf.float32):
    stddev = tf.sqrt(2. / (shape[0] + shape[1]))
    return tf.random.normal(shape, stddev=stddev, dtype=dtype)
def my_l1_regularizer(weights):
    return tf.reduce_sum(tf.abs(0.01 * weights))
def my_positive_weights(weights):      # возвращаемое значение -
                                         # просто tf.nn.relu(weights)
    return tf.where(weights < 0., tf.zeros_like(weights), weights)
```

Как видите, аргументы зависят от типа специальной функции. Затем специальные функции можно использовать обычным образом, например:

```
layer = keras.layers.Dense(30, activation=my_softplus,
                           kernel_initializer=my_glorot_initializer,
                           kernel_regularizer=my_l1_regularizer,
                           kernel_constraint=my_positive_weights)
```

Функция активации будет применяться к выходу этого слоя Dense, а ее результат передается следующему слою. Веса слоя будут инициализироваться с использованием значения, возвращенного инициализатором. На каждом шаге обучения веса будут передаваться функции регуляризации для подсчета потери регуляризации, которая добавляется к главной потере, чтобы получить финальную потерю, применяемую при обучении. Наконец, на каждом шаге проверки будет вызываться функция ограничения и веса слоя заменяются ограниченными весами.

Если функция имеет гиперпараметры, которые необходимо сохранять с моделью, тогда понадобится создать подкласс соответствующего класса, такого как keras.regularizers.Regularizer, keras.constraints.Constraint, keras.initializers.Initializer или keras.layers.Layer (для любого слоя, включая функции активации). Во многом подобно тому, как мы поступали со специальной функцией потерь, далее показан простой класс для регуляризации  $\ell_1$ , который сохраняет ее гиперпараметр factor (на этот раз вызывать родительский конструктор или метод get\_config() не нужно, потому что они не определены в родительском классе):

```
class FactorRegularizer(keras.regularizers.Regularizer):
    def __init__(self, factor):
        self.factor = factor
    def __call__(self, weights):
        return tf.reduce_sum(tf.abs(self.factor * weights))
    def get_config(self):
        return {"factor": self.factor}
```

Обратите внимание, что вы обязаны реализовать метод call() для потерь, слоев (включая функции активации) и моделей или метод \_\_call\_\_() для регуляризаторов, инициализаторов и ограничений. Для метрик ситуация несколько иная, как мы увидим в следующем разделе.

## Специальные метрики

Потери и метрики концептуально не являются одним и тем же: потери (скажем, перекрестная энтропия) используются градиентным спуском для обучения модели, так что они должны быть дифференцируемыми (по крайней мере, в местах, где они оцениваются), а их градиенты должны быть везде ненулевыми. Плюс вполне нормально то, что людям их нелегко интерпретировать. Напротив, метрики (например, правильность) применяются для

оценки модели: они обязаны легче поддаваться интерпретации, и могут не быть дифференцируемыми или повсюду иметь нулевые градиенты.

Тем не менее, в большинстве случаев определение специальной функции метрики оказывается точно таким же, как определение специальной функции потерь. Фактически в качестве метрики мы могли бы даже использовать функцию потерь Хьюбера, которую создали ранее<sup>6</sup>; она работала бы вполне хорошо (и постоянство тоже работало бы тем же образом, в данном случае сохранив только имя функции, "huber\_fn"):

```
model.compile(loss="mse", optimizer="nadam",
               metrics=[create_huber(2.0)])
```

Для каждого пакета во время обучения Keras будет вычислять данную метрику и отслеживать ее среднюю величину с начала эпохи. Большую часть времени это в точности то, что мы хотим. Но не всегда! Для примера возьмем точность двоичного классификатора. Как было показано в главе 3, точность представляет собой количество истинно положительных прогнозов, деленное на количество положительных прогнозов (включающих истинно положительные и ложноположительные прогнозы). Пусть модель выработала пять положительных прогнозов в первом пакете, из которых четыре были корректными: имеем точность 80%. Затем предположим, что модель выработала три положительных прогноза во втором пакете, но все они были некорректными: имеем точность 0% для второго пакета. Если вы просто вычислите среднее из указанных двух точностей, то получите 40%. Но секундочку — это не точность модели на двух пакетах! На самом деле из восьми положительных прогнозов (5 + 3) было всего четыре истинно положительных (4 + 0), так что общая точность составляет 50%, не 40%. Нам необходим объект, который способен отслеживать количество истинно положительных и количество ложноположительных прогнозов, а также по запросу рассчитывать их отношение. Именно этим занимается класс keras.metrics.Precision:

```
>>> precision = keras.metrics.Precision()
>>> precision([0, 1, 1, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 1, 0, 1])
<tf.Tensor: id=581729, shape=(), dtype=float32, numpy=0.8>
>>> precision([0, 1, 0, 0, 1, 0, 1, 1], [1, 0, 1, 1, 0, 0, 0, 0])
<tf.Tensor: id=581780, shape=(), dtype=float32, numpy=0.5>
```

---

<sup>6</sup> Однако потеря Хьюбера редко применяется в качестве метрики (MAE или MSE предпочтительнее).

В приведенном примере мы создаем объект `Precision`, затем применяем его подобно функции, передавая ему метки и прогнозы для первого пакета и далее для второго пакета ( обратите внимание, что можно было бы передавать и веса образцов). Мы используем такое же количество истинно положительных и ложноположительных прогнозов, как в только что обсужденном примере. После первого пакета объект `Precision` возвращает точность 80% и затем после второго — точность 50% (которая является общей точностью, а не точностью второго пакета). Это называется *потоковой (streaming) метрикой* (или *метрикой с запоминанием состояния (stateful)*), т.к. постепенно обновляется, пакет за пакетом.

В данной точке мы можем вызвать метод `result()`, чтобы получить текущее значение метрики. Мы также можем просматривать ее переменные (отслеживаемое количество истинно положительных и ложноположительных прогнозов) с применением атрибута `variables` и сбрасывать их с помощью метода `reset_states()`:

```
>>> p.result()
<tf.Tensor: id=581794, shape=(), dtype=float32, numpy=0.5>
>>> p.variables
[<tf.Variable 'true_positives:0' [...] numpy=array([4.],
dtype=float32)>,
 <tf.Variable 'false_positives:0' [...] numpy=array([4.],
dtype=float32)>]
>>> p.reset_states()    # обе переменные сбрасываются в 0.0
```

Если вы хотите создать такую потоковую метрику, тогда определите подкласс класса `keras.metrics.Metric`. Ниже представлен пример, в котором отслеживается общая потеря Хьюбера и количество образцов, встреченных до сих пор. При запросе результата возвращается отношение, которое является просто средней потерей Хьюбера:

```
class (keras.metrics.Metric):
    def __init__(self, threshold=1.0, **kwargs):
        super().__init__(**kwargs)    # поддерживает базовые
                                      # аргументы (например, dtype)
        self.threshold = threshold
        self.huber_fn = create_huber(threshold)
        self.total = self.add_weight("total", initializer="zeros")
        self.count = self.add_weight("count", initializer="zeros")
    def update_state(self, y_true, y_pred, sample_weight=None):
        metric = self.huber_fn(y_true, y_pred)
        self.total.assign_add(tf.reduce_sum(metric))
        self.count.assign_add(tf.cast(tf.size(y_true), tf.float32))
```

```
def result(self):
    return self.total / self.count
def get_config(self):
    base_config = super().get_config()
    return {**base_config, "threshold": self.threshold}
```

Давайте пройдемся по коду<sup>7</sup>.

- Конструктор использует метод `add_weight()`, чтобы создать переменные, необходимые для отслеживания состояния метрики по множеству пакетов — в данном случае суммы всех потерь Хьюбера (`total`) и количества просмотренных до сих пор образцов (`count`). При желании вы могли бы создать переменные вручную. Библиотека Keras отслуживает любой объект `tf.Variable`, который установлен в качестве атрибута (и в более общем случае любой “отслеживаемый” объект, такой как слои или модели).
- Метод `update_state()` вызывается, когда вы применяете экземпляр класса `HuberMetric` как функцию (что делалось с объектом `Precision`). Он обновляет все переменные при заданных метках и прогнозах для одного пакета (и весов образцов, но в рассматриваемом случае мы их игнорируем).
- Метод `result()` вычисляет и возвращает финальный результат, в этом случае среднюю потерю Хьюбера по всем образцам. Когда метрика используется как функция, первым вызывается метод `update_state()`, затем метод `result()` и его вывод возвращается.
- Мы также реализуем метод `get_config()` для обеспечения того, что `threshold` сохраняется вместе с моделью.
- Стандартная реализация метода `reset_states()` сбрасывает все переменные в 0.0 (но при необходимости его можно переопределить).



Библиотека Keras самостоятельно позаботится о постоянстве переменных; никакие действия не требуются.

<sup>7</sup> Класс `HuberMetric` предназначен только для демонстрационных целей. Более простая и лучшая реализация предусматривала бы создание подкласса класса `keras.metrics.Mean`; пример ищите в разделе “Streaming metrics” (Потоковые метрики) тетради Jupiter для настоящей главы.

Когда метрика определяется с применением простой функции, Keras автоматически вызывает ее для каждого пакета и отслеживает среднее на протяжении каждой эпохи, как мы делаем вручную. Следовательно, единственное преимущество нашего класса `HuberMetric` заключается в том, что `threshold` будет сохраняться. Но надо сказать, что некоторые метрики, подобные точности, не могут просто усредняться по пакетам: в таких случаях не остается ничего другого кроме реализации потоковой метрики.

После того, как мы создали потоковую метрику, построение специального слоя покажется не сложнее прогулки в парке!

## Специальные слои

Иногда может требоваться архитектура, содержащая экзотический слой, для которого TensorFlow не предоставляет стандартной реализации. В таком случае понадобится создать специальный слой. Или же может возникнуть необходимость в построении крайне повторяющейся архитектуры, которая содержит идентичные блоки слоев, встречающиеся много раз, и было бы удобно трактовать каждый блок слоев как один слой. Например, если модель является последовательностью слоев A, B, C, A, B, C, A, B, C, тогда можно определить специальный слой D, содержащий слои A, B, C, в результате чего модель будет выглядеть просто как D, D, D. Давайте посмотрим, как строить специальные слои.

Некоторые слои не имеют весов, скажем, `keras.layers.Flatten` или `keras.layers.ReLU`. Если нужно создать специальный слой без каких-либо весов, то простейший вариант предусматривает написание функции и помещение ее внутрь слоя `keras.layers.Lambda`. Например, следующий слой будет применять экспоненциальную функцию к своим входам:

```
exponential_layer = keras.layers.Lambda(lambda x: tf.exp(x))
```

Затем такой специальный слой можно использовать подобно любому другому слою посредством API-интерфейсов `Sequential`, `Functional` или `Subclassing`. Его также можно применять как функцию активации (либо использовать `activation=tf.exp`, `activation=keras.activations.exponential` или просто `activation="exponential"`). Экспоненциальный слой иногда применяется в выходном слое регрессионной модели, когда прогнозируемые значения имеют очень разные масштабы (скажем, 0.001, 10., 1000.).

Как вы вероятно уже догадались, чтобы построить специальный слой с запоминанием состояния (т.е. слой с весами), необходимо создать подкласс класса keras.layers.Layer. Например, приведенный ниже класс реализует упрощенную версию слоя Dense:

```
class (keras.layers.Layer):
    def __init__(self, units, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.units = units
        self.activation = keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.kernel = self.add_weight(
            name="kernel", shape=[batch_input_shape[-1], self.units],
            initializer="glorot_normal")
        self.bias = self.add_weight(
            name="bias", shape=[self.units], initializer="zeros")
    super().build(batch_input_shape)      # этот вызов должен
                                         # быть в конце

    def call(self, X):
        return self.activation(X @ self.kernel + self.bias)

    def compute_output_shape(self, batch_input_shape):
        return tf.TensorShape(batch_input_shape.as_list()[:-1]
                             + [self.units])

    def get_config(self):
        base_config = super().get_config()
        return {**base_config, "units": self.units,
                "activation": keras.activations.serialize(self.activation)}
```

Разберем код.

- Конструктор принимает все гиперпараметры в качестве аргументов (в данном примере `units` и `activation`) и что важнее — также и аргумент `**kwargs`. Он вызывает родительский конструктор, передавая ему `kwargs`: это обеспечит обработку стандартных аргументов, таких как `input_shape`, `trainable` и `name`. Затем конструктор сохраняет гиперпараметры как атрибуты, преобразуя аргумент `activation` в подходящую функцию активации с использованием функции `keras.activations.get()` (она принимает функции, стандартные строки вроде `"relu"` или `"selu"` либо просто `None`)<sup>8</sup>.

<sup>8</sup> Эта функция специфична для `tf.keras`. Взамен можно было бы применять `keras.activations.Activation`.

- Роль метода `build()` заключается в создании переменных слоя путем вызова метода `add_weight()` для каждого веса. Метод `build()` вызывается при первом применении слоя. В данной точке библиотеке Keras будет известна форма входов слоя и она передаст ее методу `build()`<sup>9</sup>, что часто необходимо для создания ряда весов. Скажем, чтобы создать матрицу весов связей (т.е. "kernel"), нам нужно знать количество нейронов в предыдущем слое: оно соответствует размеру последнего измерения входов. В конце метода `build()` (и только в конце), должен вызываться метод `build()` родительского класса: такой вызов сообщает Keras о том, что слой построен (он всего лишь устанавливает `self.built=True`).
- Метод `call()` выполняет желаемые операции. В этом случае мы вычисляем результат матричного перемножения входов  $X$  и ядра слоя, добавляем вектор смещений и применяем к результату функцию активации, что дает нам выход слоя.
- Метод `compute_output_shape()` просто возвращает форму выходов слоя. В данном случае она будет такой же, как у входов, кроме того, что последнее измерение заменяется количеством нейронов в слое. Обратите внимание, что в `tf.keras` формы являются экземплярами класса `tf.TensorShape`, которые можно преобразовать в списки Python, используя `as_list()`.
- Метод `get_config()` такой же, как в предшествующих специальных классах. Мы сохраняем полную конфигурацию функции активации, вызывая `keras.activations.serialize()`.

Теперь вы можете задействовать слой `MyDense` подобно любому другому слою!



Вызов метода `compute_output_shape()` обычно можно опускать, т.к. `tf.keras` автоматически выводит форму выходов за исключением случая, когда слой является динамическим (что мы вскоре увидим). В других реализациях Keras указанный метод либо обязательен, либо его стандартная реализация предполагает, что форма выходов совпадает с формой входов.

<sup>9</sup> В API-интерфейсе Keras этот аргумент называется `input_shape`, но поскольку он включает также размерность пакета, я предлагаю называть его `batch_input_shape`. То же касается и `compute_output_shape()`.

Чтобы создать слой с множеством входов (например, `Concatenate`), аргументом метода `call()` должен быть кортеж, содержащий все входы, и подобным же образом аргументом метода `compute_output_shape()` должен быть кортеж, который содержит формы пакетов каждого входа. Для создания слоя с множеством выходов метод `call()` обязан возвращать список выходов, а метод `compute_output_shape()` — список форм пакетов выходов (по одной на выход). Скажем, следующий игрушечный слой принимает два входа и возвращает три выхода:

```
class (keras.layers.Layer):
    def call(self, X):
        X1, X2 = X
        return [X1 + X2, X1 * X2, X1 / X2]

    def compute_output_shape(self, batch_input_shape):
        b1, b2 = batch_input_shape
        return [b1, b1, b1]      # вероятно должны поддерживаться
                               # правила ретрансляции
```

Этот слой можно применять подобно любому другому слою, но лишь через API-интерфейсы `Functional` и `Subclassing`, но не API-интерфейс `Sequential` (который принимает только слои с одним входом и одним выходом).

Если нужно, чтобы ваш слой обладал отличающимся поведением во время обучения и проверки (скажем, когда он использует слои `Dropout` или `BatchNormalization`), тогда вы должны добавить к методу `call()` аргумент `training` и применять его для решения, что делать. Например, давайте создадим слой, который добавляет гауссов шум (для регуляризации) во время обучения, но ничего не делает во время проверки (в Keras имеется слой `keras.layers.GaussianNoise`, обеспечивающий такое же поведение):

```
class (keras.layers.Layer):
    def __init__(self, stddev, **kwargs):
        super().__init__(**kwargs)
        self.stddev = stddev

    def call(self, X, training=None):
        if training:
            noise = tf.random.normal(tf.shape(X), stddev=self.stddev)
            return X + noise
        else:
            return X

    def compute_output_shape(self, batch_input_shape):
        return batch_input_shape
```

Итак, вы способны строить любой специальный слой, который вам необходим! Теперь займемся созданием специальных моделей.

## Специальные модели

Мы уже сталкивались со специальными классами моделей в главе 10, когда обсуждали API-интерфейс Subclassing API<sup>10</sup>. Все довольно прямолинейно: создайте подкласс класса `keras.Model`, создайте в конструкторе слои и переменные, затем реализуйте метод `call()`, чтобы он делал то, что должна делать модель. Допустим, требуется построить модель, изображенную на рис. 12.3.

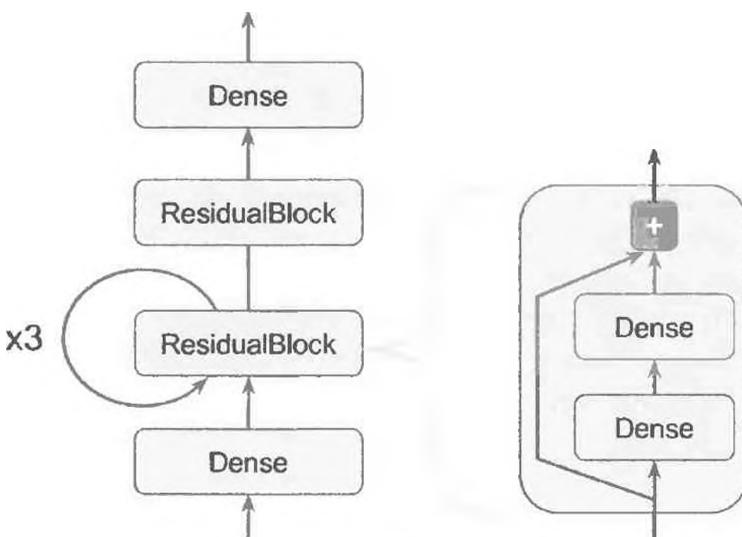


Рис. 12.3. Пример специальной модели: произвольная модель со специальным слоем `ResidualBlock`, который содержит обходящую связь

Входы проходят через первый плотный слой, затем через остаточный блок (*residual block*), образованный из двух плотных слоев и операции сложения (как мы увидим в главе 14, остаточный блок складывает свои входы со своими выходами), далее через тот же самый остаточный блок еще три раза, затем через второй остаточный блок и финальный результат проходит через плотный выходной слой. Обратите внимание, что приведенная модель не имеет особого смысла; она всего лишь является примером, иллюстрирующим

<sup>10</sup> Понятие “API-интерфейс Subclassing” обычно относится только к построению специальных моделей путем создания подклассов, хотя в главе было показано, что с помощью создания подклассов можно делать многие другие вещи.

ющим тот факт, что вы можете легко построить модель любого желаемого вида, даже такую, которая содержит циклы и обходящие связи. Для реализации этой модели лучше сначала создать слой ResidualBlock, поскольку мы собираемся создавать пару идентичных блоков (и можем захотеть повторно задействовать его в другой модели):

```
class (keras.layers.Layer):
    def __init__(self, n_layers, n_neurons, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(n_neurons, activation="elu",
                                         kernel_initializer="he_normal")
                      for _ in range(n_layers)]
    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        return inputs + Z
```

Получившийся слой немного особенный, т.к. он содержит другие слои. Библиотека Keras обрабатывает это прозрачным образом: она автоматически обнаруживает, что атрибут hidden содержит отслеживаемые объекты (здесь слои), так что их переменные автоматически добавляются к списку переменных данного слоя. Остаток класса пояснений не требует. Далее воспользуемся API-интерфейсом Subclassing для определения самой модели:

```
class (keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = keras.layers.Dense(30, activation="elu",
                                         kernel_initializer="he_normal")
        self.block1 = ResidualBlock(2, 30)
        self.block2 = ResidualBlock(2, 30)
        self.out = keras.layers.Dense(output_dim)

    def call(self, inputs):
        Z = self.hidden1(inputs)
        for _ in range(1 + 3):
            Z = self.block1(Z)
        Z = self.block2(Z)
        return self.out(Z)
```

Мы создали слои в конструкторе и задействовали их в методе call(). Далее модель можно применять аналогично любой другой модели (компилировать, подгонять и использовать для выработывания прогнозов). Если

вы также хотите иметь возможность сохранять модель посредством метода `save()` и загружать ее с помощью функции `keras.models.load_model()`, тогда должны реализовать метод `get_config()` (как делалось ранее) в классах `ResidualBlock` и `ResidualRegressor`. В качестве альтернативы можете сохранять и загружать веса с применением методов `save_weights()` и `load_weights()`.

Класс `Model` является подклассом класса `Layer`, поэтому модели можно определять и использовать в точности как слои. Но модель обладает дополнительной функциональностью, включающей методы `compile()`, `fit()`, `evaluate()` и `predict()` (и несколько разновидностей) плюс метод `get_layers()` (который может возвращать любые слои модели по имени или по индексу) и метод `save()` (а также поддержку для `keras.models.load_model()` и `keras.models.clone_model()`).



Если модели предлагают больше функциональности, чем слои, то почему бы не определять каждый слой как модель? Формально можно было бы поступить так, но обычно лучше проводить различие между внутренними компонентами модели (т.е. слоями или многократно применяемыми блоками слоев) и самой моделью (т.е. объектом, который вы будете обучать). В первом случае должны создаваться подклассы класса `Layer`, а во втором — подклассы класса `Model`.

Теперь вы свободно и выразительно можете построить почти любую модель, которую найдете на бумаге, используя API-интерфейсы `Sequential`, `Functional`, `Subclassing` или даже их смесь. “Почти” любую модель? Да, все еще остается несколько вещей, на которые мы должны взглянуть: во-первых, как определять потери или метрики, основанные на внутренностях модели, и, во-вторых, каким образом строить специальный цикл обучения.

## Потери и метрики, основанные на внутренностях модели

Все специальные потери и метрики, которые мы определяли ранее, были основаны на метках и прогнозах (и дополнительно на весах образцов). Есть ситуации, когда необходимо определять потери на основе других частей модели, таких как веса или активации скрытых слоев. Они могут быть полезными для целей регуляризации или для отслеживания какого-то внутреннего аспекта модели.

Чтобы определить специальную потерю, основанную на внутренностях модели, подсчитайте ее на базе любой желаемой части модели и затем передайте результат методу `add_loss()`. Например, давайте построим специальную регрессионную модель на основе многослойного персептрона, состоящую из стопки с пятью слоями и выходного слоя. Эта специальная модель также будет иметь вспомогательный выход на верхушке самого верхнего скрытого слоя. Потеря, ассоциированная со вспомогательным выходом, будет называться *потерей из-за реконструкции* (см. главу 17): она представляет собой средний квадрат разности между реконструкцией и входами. Прибавляя такую потерю из-за реконструкции к главной потере, мы будем подталкивать модель к предохранению как можно большего объема информации, проходящей через скрытые слои — даже информации, которая не будет полезной непосредственно для самой задачи регрессии. На практике эта потеря временами улучшает обобщение (она является потерей регуляризации). Ниже представлен код для такой специальной модели со специальной потерей из-за реконструкции:

```
class MyModel(keras.Model):
    def __init__(self, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.hidden = [keras.layers.Dense(30, activation="selu",
                                         kernel_initializer="lecun_normal")
                      for _ in range(5)]
        self.out = keras.layers.Dense(output_dim)

    def build(self, batch_input_shape):
        n_inputs = batch_input_shape[-1]
        self.reconstruct = keras.layers.Dense(n_inputs)
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = inputs
        for layer in self.hidden:
            Z = layer(Z)
        reconstruction = self.reconstruct(Z)
        recon_loss = tf.reduce_mean(tf.square(reconstruction - inputs))
        self.add_loss(0.05 * recon_loss)
        return self.out(Z)
```

Пройдемся по коду.

- Конструктор создает сеть DNN с пятью плотными скрытыми слоями и одним плотным выходным слоем.

- Метод `build()` создает добавочный плотный слой, который будет применяться для реконструкции входов модели. Он должен быть создан здесь по той причине, что количество элементов обязано быть равным количеству входов, а до вызова метода `build()` это количество неизвестно.
- Метод `call()` прогоняет входы через все пять скрытых слоев, после чего пропускает результат через слой реконструкции, который выпускает реконструкцию.
- Затем метод `call()` подсчитывает потерю из-за реконструкции (средний квадрат разности между реконструкцией и входами) и добавляет ее к списку потерь модели, используя метод `add_loss()`<sup>11</sup>. Обратите внимание, что мы уменьшаем масштаб потери из-за реконструкции, умножая ее на 0.05 (это гиперпараметр, который можно подстраивать). Тем самым гарантируется, что потеря из-за реконструкции не доминирует над главной потерей.
- Наконец, метод `call()` передает выход скрытых слоев выходному слою и возвращает его выход.

Аналогично вы можете добавить специальную метрику, основанную на внутренностях модели, путем ее расчета любым желаемым способом, при условии, что результатом будет выход объекта метрики. Например, можете создать в конструкторе объект `keras.metrics.Mean`, вызвать его в методе `call()`, передав ему `recon_loss`, и в заключение добавить его к модели, вызвав метод `add_metric()` модели. Таким образом, когда вы обучаете модель, Keras будет отображать среднюю потерю по каждой эпохе (эта потеря представляет собой сумму главной потери и умноженной на 0.05 потери из-за реконструкции) и среднюю ошибку восстановления по каждой эпохе. Во время обучения обе будут снижаться:

```
Epoch 1/5
11610/11610 [=====] [...] loss: 4.3092 - reconstruction_
error: 1.7360
Epoch 2/5
11610/11610 [=====] [...] loss: 1.1232 - reconstruction_
error: 0.8964
[...]
```

<sup>11</sup> Вы также можете вызывать `add_loss()` в любом слое внутри модели, т.к. модель рекурсивно собирает потери из всех своих слоев.

Более чем в 99% случаев всего того, что мы обсудили до сих пор, будет достаточно для реализации любой модели, которую нужно построить, даже со сложной архитектурой, потерями и метриками. Однако в редких ситуациях может возникать необходимость в настройке самого цикла обучения. Прежде чем мы до этого доберемся, понадобится выяснить, как градиенты автоматически вычисляются в TensorFlow.

## Вычисление градиентов с использованием автоматического дифференцирования

Чтобы понять, как применяется автоматическое дифференцирование (см. главу 10 и приложение Г) для автоматического вычисления градиентов, давайте возьмем простую игрушечную функцию:

```
def f(w1, w2):  
    return 3 * w1 ** 2 + 2 * w1 * w2
```

Если вы владеете исчислением, то можете аналитически найти, что частная производная этой функции относительно  $w1$  выглядит как  $6 * w1 + 2 * w2$ , а ее частной производной относительно  $w2$  будет  $2 * w1$ . Например, в точке  $(w1, w2) = (5, 3)$  эти частные производные равны соответственно 36 и 10, так что вектором-градиентом в указанной точке окажется  $(36, 10)$ . Но в случае нейронной сети функция была бы гораздо более сложной, обычно с десятками тысяч параметров, и нахождение частных производных аналитически вручную стало бы практически невыполнимой задачей. Одно из решений могло бы предусматривать вычисление приближения каждой частной производной путем измерения того, насколько изменяется выход функции при подстройке соответствующего параметра:

```
>>> w1, w2 = 5, 3  
>>> eps = 1e-6  
>>> (f(w1 + eps, w2) - f(w1, w2)) / eps  
36.000003007075065  
>>> (f(w1, w2 + eps) - f(w1, w2)) / eps  
10.000000003174137
```

На вид правильно! Прием работает довольно хорошо и легко реализуется, но он дает лишь приближение, и важно то, что  $f()$  необходимо вызывать, по крайней мере, один раз на параметр (не дважды, поскольку мы смогли бы вычислить  $f(w1, w2)$  только однократно). Потребность в вызове  $f()$ , по

крайней мере, один раз на параметр делает такой подход неподходящим для крупных нейронных сетей. Таким образом, взамен мы должны использовать автоматическое дифференцирование. Библиотека TensorFlow делает решение довольно простым:

```
w1, w2 = tf.Variable(5.), tf.Variable(3.)
with tf.GradientTape() as tape:
    z = f(w1, w2)
gradients = tape.gradient(z, [w1, w2])
```

Сначала мы определяем две переменные, `w1` и `w2`, затем создаем контекст в виде ленты `tf.GradientTape`, который будет автоматически регистрировать каждую операцию, которая задействует переменную, и в заключение запрашиваем у ленты вычисление градиентов результата `z` относительно обеих переменных [`w1`, `w2`]. Давайте взглянем на градиенты, вычисленные TensorFlow:

```
>>> gradients
[<tf.Tensor: id=828234, shape=(), dtype=float32, numpy=36.0>,
 <tf.Tensor: id=828229, shape=(), dtype=float32, numpy=10.0>]
```

Безупречно! Помимо обеспечения точного результата (точность ограничивается лишь ошибками с плавающей точкой) метод `gradient()` прошел по зарегистрированным вычислениям всего один раз (в обратном порядке) независимо от того, насколько много было переменных, поэтому он невероятно эффективен. Он подобен магии!



Чтобы сберечь память, помещайте внутрь блока `tf.GradientTape()` только строгий минимум. В качестве альтернативы приостановите регистрацию, создав блок `with tape.stop_recording()` внутри блока `tf.GradientTape()`.

Лента автоматически стирается немедленно после того, как вызван ее метод `gradient()`, так что при попытке двукратного вызова `gradient()` будет сгенерировано исключение:

```
with tf.GradientTape() as tape:
    z = f(w1, w2)

dz_dw1 = tape.gradient(z, w1)      # => тензор 36.0
dz_dw2 = tape.gradient(z, w2)      # ошибка времени выполнения!
```

Если метод `gradient()` необходимо вызывать более одного раза, тогда вы должны сделать ленту постоянной и удалять ее каждый раз, когда работа с ней окончена, для освобождения ресурсов<sup>12</sup>:

```
with tf.GradientTape(persistent=True) as tape:  
    z = f(w1, w2)  
  
    dz_dw1 = tape.gradient(z, w1)      # => тензор 36.0  
    dz_dw2 = tape.gradient(z, w2)      # => тензор 10.0, теперь  
                                      #   работает нормально!  
  
del tape
```

По умолчанию лента будет отслеживать только операции, которые **задействуют** переменные, так что если вы попытаетесь вычислить градиент `z` относительно чего угодно другого кроме переменной, то результатом будет `None`:

```
c1, c2 = tf.constant(5.), tf.constant(3.)  
with tf.GradientTape() as tape:  
    z = f(c1, c2)  
  
gradients = tape.gradient(z, [c1, c2])    # возвращает [None, None]
```

Тем не менее, вы можете заставить ленту следить за любыми тензорами, чтобы регистрировать каждую операцию, в которую они вовлечены. Затем можете вычислить градиенты относительно таких тензоров, как если бы они были переменными:

```
with tf.GradientTape() as tape:  
    tape.watch(c1)  
    tape.watch(c2)  
    z = f(c1, c2)  
  
gradients = tape.gradient(z, [c1, c2])    # возвращает  
                                         # [тензор 36., тензор 10.]
```

Прием может быть полезен в ряде случаев вроде того, когда вы хотите реализовать потерю регуляризации, штрафующую активации, которые сильно варьируются при небольшом изменении входов: потеря будет основана на градиенте активаций относительно входов. Поскольку входы не являются переменными, вам понадобится сообщить ленте о необходимости следить за ними.

---

<sup>12</sup> Если лента покидает область видимости, скажем, когда происходит возврат из функции, которая ее использует, то сборщик мусора Python удалит ее самостоятельно.

Большую часть времени лента градиентов применяется для вычисления градиентов единственной величины (обычно потери) относительно набора значений (как правило, параметров модели). Именно здесь ярко выделяется автоматическое дифференцирование в обратном режиме, т.к. для получения одновременно всех градиентов ему нужно лишь сделать один прямой проход и один обратный проход. Если вы пытаетесь вычислить градиенты вектора, например, вектора с множеством потерь, тогда TensorFlow рассчитывает градиенты суммы вектора. Таким образом, если когда-нибудь возникнет потребность в индивидуальных градиентах (скажем, градиентах каждой потери относительно параметров модели), то вам придется вызывать метод `jacobian()` ленты: он выполнит автоматическое дифференцирование в обратном режиме один раз для каждой потери в векторе (по умолчанию все параллельно). Можно даже вычислить частные производные второго порядка (гессианы, т.е. частные производные частных производных), но на практике необходимость в них возникает редко (пример ищите в разделе “Computing Gradients with Autodiff” (Расчет градиентов с помощью автоматического дифференцирования) тетради Jupyter для настоящей главы).

В определенных ситуациях желательно остановить обратное распространение градиентов через некоторые части нейронной сети. Для этого должна использоваться функция `tf.stop_gradient()`, которая возвращает свои входы во время прямого прохода (наподобие `tf.identity()`), но не позволяет градиентам продвигаться через обратное распространение (она действует как константа):

```
def f(w1, w2):
    return 3 * w1 ** 2 + tf.stop_gradient(2 * w1 * w2)
with tf.GradientTape() as tape:
    z = f(w1, w2)  # тот же результат как в отсутствие stop_gradient()
gradients = tape.gradient(z, [w1, w2]) #=>возвращает [тензор 30., None]
```

Наконец, при вычислении градиентов иногда могут случаться численные проблемы. Например, если вы рассчитываете градиенты функции `my_softplus()` для крупных входов, то результатом будет NaN:

```
>>> x = tf.Variable([100.])
>>> with tf.GradientTape() as tape:
...     z = my_softplus(x)
>>> tape.gradient(z, [x])
<tf.Tensor: [...] numpy=array([nan], dtype=float32)>
```

Причина в том, что вычисление градиентов этой функции с применением автоматического дифференцирования приводит к ряду численных затруднений: из-за ошибок точности значений с плавающей точкой автоматическое дифференцирование заканчивает делением бесконечности на бесконечность (которое дает NaN). К счастью, мы можем аналитически выяснить, что производной функции softplus является функция  $1 / (1 + 1 / \exp(x))$ , которая численно устойчива. Далее можно указать TensorFlow на необходимость использования при расчете градиентов функции `my_softplus()` такой численно устойчивой функции. Ее потребуется декорировать с помощью `@tf.custom_gradient` и сделать так, чтобы она возвращала нормальный выход и функцию, вычисляющую производные (обратите внимание, что функция будет получать в качестве входа градиенты, которые до сих пор участвовали в обратном распространении до самой функции softplus; и в соответствии с цепным правилом мы обязаны умножить их на градиенты этой функции):

```
@tf.custom_gradient
def my_better_softplus(z):
    exp = tf.exp(z)
    def my_softplus_gradients(grad):
        return grad / (1 + 1 / exp)
    return tf.math.log(exp + 1), my_softplus_gradients
```

Теперь при вычислении градиентов функции `my_better_softplus()` мы получаем надлежащий результат даже для крупных входных значений (однако, главный выход по-прежнему подвергается взрывному росту, т.к. он экспоненциальный; один обходной путь предусматривает применение `tf.where()` для возвращения входов, когда они велики).

Примите поздравления! Теперь вы можете рассчитывать градиенты любой функции (при условии, что она дифференцируема в точке расчета), блокируя обратное распространение, когда необходимо, и реализовывать собственные функции градиентов. Вероятно, вы добились большей гибкости, чем когда-либо понадобится, даже в случае построения специальных циклов обучения, как вскоре будет показано.

## Специальные циклы обучения

В редких случаях метод `fit()` может оказаться недостаточно гибким для тех целей, которые вы преследуете. Скажем, при обсуждаемом в главе 10 обучении Wide & Deep (<https://homl.info/widedeep>) используются

два разных оптимизатора: один для широкого пути, а другой для глубокого пути. Поскольку метод `fit()` применяет только один оптимизатор (указанный при компиляции модели), реализация обучения Wide & Deep требует написания специального цикла.

Вы также можете быть склонны реализовывать специальные циклы обучения просто для того, чтобы иметь большую уверенность в том, что они делают в точности то, для чего предназначены (возможно, вы не уверены в некоторых деталях метода `fit()`). Временами безопаснее выполнять все явно. Тем не менее, не забывайте о том, что написание специального цикла обучения делает код более длинным, подверженным ошибкам и трудным в сопровождении.



Если только вы действительно не нуждаетесь в дополнительной гибкости, то должны отдавать предпочтение методу `fit()`, а не реализации собственного цикла обучения, особенно при работе в составе команды.

Первым делом давайте построим простую модель. Компилировать ее не нужно, т.к. мы будем поддерживать цикл обучения вручную:

```
l2_reg = keras.regularizers.l2(0.05)
model = keras.models.Sequential([
    keras.layers.Dense(30, activation="elu",
                      kernel_initializer="he_normal",
                      kernel_regularizer=l2_reg),
    keras.layers.Dense(1, kernel_regularizer=l2_reg)
])
```

Далее мы создадим крошечную функцию, которая будет случайным образом выбирать пакет образцов из обучающего набора (в главе 13 мы обсудим API-интерфейс `Data`, который предлагает гораздо лучшую альтернативу):

```
def random_batch(X, y, batch_size=32):
    idx = np.random.randint(len(X), size=batch_size)
    return X[idx], y[idx]
```

Затем мы также определим функцию, которая будет отображать состояние обучения, включая номер шага, общее количество шагов, среднюю потерю с начала эпохи (т.е. для ее подсчета мы воспользуемся метрикой Mean) и другие метрики:

```

def print_status_bar(iteration, total, loss, metrics=None):
    metrics = " - ".join(["{}: {:.4f}"].format(m.name, m.result())
                           for m in [loss] + (metrics or [])))
    end = "" if iteration < total else "\n"
    print("\r{} / {} - ".format(iteration, total) + metrics,
          end=end)

```

Код не требует особых пояснений, если вы знакомы с форматированием строк в Python: `{:.4f}` обеспечивает форматирование числа с плавающей точкой с четырьмя цифрами после десятичной точки, а применение `\r` (возврат каретки) вместе с `end=""` гарантирует, что строка состояния всегда выводится в одной строке. В тетради Jupiter, относящейся к главе, функция `print_status_bar()` включает индикатор хода работ, но вы можете замен использовать удобную библиотеку `tqdm`.

Итак, возьмемся за дело! Прежде всего, нам необходимо определить ряд гиперпараметров и выбрать оптимизатор, функцию потерь и метрики (в примере только MAE):

```

n_epochs = 5
batch_size = 32
n_steps = len(X_train) // batch_size
optimizer = keras.optimizers.Nadam(lr=0.01)
loss_fn = keras.losses.mean_squared_error
mean_loss = keras.metrics.Mean()
metrics = [keras.metrics.MeanAbsoluteError()]

```

И теперь все готово для построения специального цикла!

```

for epoch in range(1, n_epochs + 1):
    print("Epoch {} / {}".format(epoch, n_epochs))
    for step in range(1, n_steps + 1):
        X_batch, y_batch = random_batch(X_train_scaled, y_train)
        with tf.GradientTape() as tape:
            y_pred = model(X_batch, training=True)
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))
            loss = tf.add_n([main_loss] + model.losses)
        gradients = tape.gradient(loss, model.trainable_variables)
        optimizer.apply_gradients(zip(gradients,
                                       model.trainable_variables))
        mean_loss(loss)
        for metric in metrics:
            metric(y_batch, y_pred)
    print_status_bar(step * batch_size, len(y_train),
                     mean_loss, metrics)

```

```
print_status_bar(len(y_train), len(y_train), mean_loss, metrics)
for metric in [mean_loss] + metrics:
    metric.reset_states()
```

В коде много чего происходит, поэтому разберем его.

- Мы создаем два вложенных цикла: один для эпох, а другой для пакетов внутри эпохи.
- Затем мы выбираем случайный пакет из обучающего набора.
- Внутри блока `tf.GradientTape()` мы вырабатываем прогноз для одного пакета (применяя модель как функцию) и подсчитываем потерю: она равна главной потере плюс другие потери (в нашей модели имеется одна потеря регуляризации на слой). Так как функция `mean_squared_error()` возвращает одну потерю на образец, мы вычисляем среднее по пакету, используя `tf.reduce_mean()` (если вы хотите применять к каждому образцу отличающиеся веса, то именно здесь это следует делать). Потери регуляризации уже сокращены до одиночного скаляра каждая, а потому нам нужно лишь просуммировать их (используя функцию `tf.add_n()`, которая суммирует множество тензоров одной и той же формы и типа данных).
- Далее мы запрашиваем у ленты вычисление градиента потери относительно каждой обучаемой переменной (*не* всех переменных!) и применяем градиенты к оптимизатору для выполнения шага градиентного спуска.
- Затем мы обновляем среднюю потерю и метрики (по текущей эпохе) и отображаем строку состояния.
- В конце каждой эпохи мы снова отображаем строку состояния, чтобы придать ей завершенный вид<sup>13</sup> и вывести символ перевода строки, и сбрасываем состояние средней потери и метрик.

Если вы установите гиперпараметр `clipnorm` или `clipvalue` оптимизатора, тогда он позаботится об этом самостоятельно. Если вы хотите применить к градиентам любую другую трансформацию, то просто применяйте ее перед вызовом метода `apply_gradients()`.

<sup>13</sup> Правда в том, что мы не обрабатываем все образцы без исключения в обучающем наборе, поскольку выбираем образцы случайным образом: некоторые были обработаны более одного раза, тогда как другие вообще не обрабатывались. Аналогично, если размер обучающего набора не кратен размеру пакета, то несколько образцов будут утрачены. На практике это нормально.

Если вы добавите к модели весовые ограничения (например, установив `kernel_constraint` или `bias_constraint` при создании слоя), тогда должны обновить цикл обучения, чтобы применить такие ограничения сразу после вызова `apply_gradients()`:

```
for variable in model.variables:  
    if variable.constraint is not None:  
        variable.assign(variable.constraint(variable))
```

Важнее всего то, что построенный цикл обучения не поддерживает слои, которые ведут себя по-разному во время обучения и проверки (скажем, `BatchNormalization` или `Dropout`). Для их поддержки необходимо вызывать модель с аргументом `training=True` и обеспечить его распространение во все слои, которые в нем нуждаются.

Как видите, есть довольно много вещей, которые необходимо понимать правильно, и в отношении них легко допустить ошибку. Но с другой стороны вы обретаете полный контроль, так что вам решать.

Теперь, когда вы знаете, каким образом настраивать любую часть своих моделей<sup>14</sup> и алгоритмов обучения, давайте посмотрим, как можно использовать средство автоматической генерации графов TensorFlow: оно способно значительно ускорить выполнение вашего специального кода и также делает его переносимым на любую платформу, поддерживаемую TensorFlow (см. главу 19).

## Функции и графы TensorFlow

В TensorFlow 1 графы были неминуемыми (как и привносимые ими сложности), потому что они являлись центральной частью API-интерфейса TensorFlow. В TensorFlow 2 они по-прежнему присутствуют, но не настолько центральны, и их намного (намного!) проще использовать. Чтобы показать, насколько просто, мы начнем с тривиальной функции, которая вычисляет куб своего входа:

```
def cube(x):  
    return x ** 3
```

---

<sup>14</sup> За исключением оптимизаторов, т.к. мало кто когда-либо настраивает их; пример ищите в разделе “Custom Optimizers” (Специальные оптимизаторы) тетради Jupiter для настоящей главы.

Очевидно, мы можем вызвать эту функцию со значением Python, таким как целое или число с плавающей точкой, или с тензором:

```
>>> cube(2)
8
>>> cube(tf.constant(2.0))
<tf.Tensor: id=18634148, shape=(), dtype=float32, numpy=8.0>
```

А теперь давайте применим `tf.function()` для преобразования такой функции Python в функцию *TensorFlow (TF Function)*:

```
>>> tf_cube = tf.function(cube)
>>> tf_cube
<tensorflow.python.eager.def_function.Function at 0x1546fc080>
```

Затем функцию TF Function можно использовать в точности как исходную функцию Python и она будет возвращать тот же самый результат (но в виде тензора):

```
>>> tf_cube(2)
<tf.Tensor: id=18634201, shape=(), dtype=int32, numpy=8>
>>> tf_cube(tf.constant(2.0))
<tf.Tensor: id=18634211, shape=(), dtype=float32, numpy=8.0>
```

“За кулисами” `tf.function()` анализирует вычисления, выполняемые функцией `cube()`, и генерирует эквивалентный вычислительный граф! Как видите, все было довольно-таки безболезненно (вскоре мы рассмотрим, как это работает). В качестве альтернативы мы могли бы применить `tf.function` как декоратор; на самом деле так поступают чаще:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

В случае необходимости исходная функция Python доступна через атрибут `python_function` функции TF Function:

```
>>> tf_cube.python_function(2)
8
```

TensorFlow оптимизирует вычислительный граф, отсекая неиспользуемые узлы, упрощая выражения (скажем,  $1 + 2$  заменяется  $3$ ) и т.д. После того, как оптимизированный график готов, функция TF Function эффективно выполняет операции в графике в надлежащем порядке (и по возможности параллельно). В результате функция TF Function обычно будет выполняться гораздо

быстрее исходной функции Python, особенно если она производит сложные вычисления<sup>15</sup>. Большую часть времени вам в действительности нужно знать лишь то, что когда вы хотите ускорить выполнение функции Python, просто преобразуйте ее в функцию TF Function. Вот и все!

Кроме того, когда вы пишете специальную функцию потерь, специальную метрику, специальный слой или любую другую специальную функцию и применяете ее в модели Keras (как делалось повсеместно в главе), то Keras автоматически преобразует вашу функцию в функцию TF Function — использовать `tf.function()` не нужно. Таким образом, почти все время такая магия на 100% прозрачна.



При создании специального слоя или специальной модели вы можете сообщить Keras о том, что ваши функции Python *не* должны преобразовываться в функции TF Function, установив `dynamic=True`. Альтернативно вы можете установить `run_eagerly=True`, когда вызываете метод `compile()` модели.

По умолчанию функция TF Function генерирует новый граф для каждого уникального множества входных форм и типов данных, после чего копирует его для последующих вызовов. Например, если вы вызываете `tf_cube(tf.constant(10))`, то график сгенерируется для тензоров `int32` формы `[1]`. Если вы далее вызовите `tf_cube(tf.constant(20))`, тогда тот же самый график будет задействован повторно. Но если вы затем вызовите `tf_cube(tf.constant([10, 20]))`, то сгенерируется новый график для тензоров `int32` формы `[2]`. Именно так функции TF Function поддерживают полиморфизм (т.е. варьирование типов и форм аргументов). Тем не менее, это справедливо только для тензорных аргументов: если вы передадите функции TF Function числовые значения Python, то новый график будет генерироваться для каждого индивидуального значения: скажем, вызовы `tf_cube(10)` и `tf_cube(20)` приведут к генерации двух графов.



В случае многократного вызова функции TF Function с разными числовыми значениями Python будет сгенерировано много графов, которые замедлят работу вашей программы и займут много пространства в оперативной памяти (чтобы освободить

<sup>15</sup> Однако в таком тривиальном примере вычислительный график настолько мал, что в нем нечего оптимизировать, поэтому `tf_cube()` на самом деле выполняется намного медленнее, чем `cube()`.

его, придется удалить функцию TF Function). Значения Python должны резервироваться для аргументов, которые будут иметь немного уникальных значений, таких как гиперпараметры вроде количества нейронов на слой. Это позволит TensorFlow лучше оптимизировать каждую разновидность вашей модели.

## AutoGraph и трассировка

Так каким же образом TensorFlow генерирует графы? Сначала анализируется исходный код функции Python для сбора всех операторов управления потоком выполнения наподобие циклов `for`, циклов `while` и операторов `if`, а также операторов `break`, `continue` и `return`. Описанный первый шаг называется *AutoGraph*. Причина, по которой библиотека TensorFlow должна анализировать исходный код, заключается в том, что язык Python не предоставляет никакого другого способа для сбора операторов управления потоком выполнения: он предлагает магические методы, такие как `__add__()` и `__mul__()` для сбора операций `+` и `*`, но магические методы вроде `__while__()` или `__if__()` отсутствуют. После анализа кода функции средство AutoGraph выдает ее обновленную версию, где все операторы управления потоком выполнения заменены соответствующими операциями TensorFlow, такими как `tf.while_loop()` для циклов и `tf.cond()` для операторов `if`. Например, на рис. 12.4 средство AutoGraph анализирует исходный код функции Python по имени `sum_squares()` и генерирует функцию `tf_sum_squares()`. В этой функции цикл `for` заменен определением функции `loop_body()` (содержащим тело исходного цикла `for`), за которым следует вызов функции `for_stmt()`. Такой вызов будет строить подходящую операцию `tf.while_loop()` в вычислительном графе.

Далее TensorFlow вызывает “модернизированную” функцию, но вместо аргумента передает ей *символьный тензор* — тензор без фактического значения, имеющий только имя, тип данных и форму. Скажем, если вы вызовите `sum_squares(tf.constant(10))`, тогда функция `tf_sum_squares()` будет вызвана с символьным тензором типа `int32` и формы `[ ]`. Функция запускается в режиме графа, т.е. каждая операция TensorFlow добавляется в граф узел для представления самой себя и своего выходного тензора или тензоров (в противоположность обычному режиму, называемому *энергичным (eager) выполнением* или *энергичным режимом*).

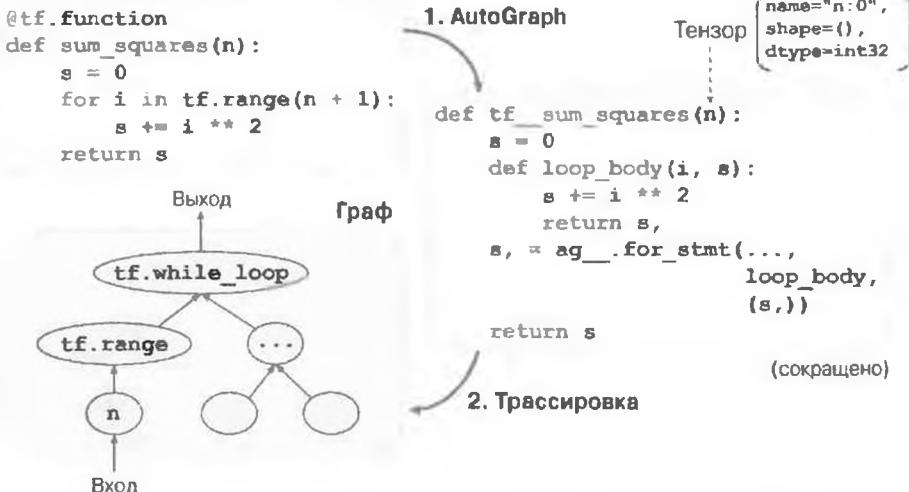


Рис. 12.4. Как TensorFlow генерирует графы, используя AutoGraph и трассировку

В режиме графа операции TensorFlow не выполняют никаких вычислений. Ситуация должна выглядеть знакомой, если вы знаете TensorFlow 1, т.к. там режим графа был стандартным. На рис. 12.4 вы можете видеть функцию `tf_sum_squares()`, вызываемую с символьным тензором в качестве аргумента (в данном случае тензором `int32` формы `[]`) и финальный граф, генерируемый во время трассировки. Узлы представляют операции, а линии со стрелками — тензоры (генерированные функция и график были упрощены).



Чтобы просмотреть исходный код сгенерированной функции, вы можете вызвать `tf.autograph.to_code(sum_squares.python_function)`. Код не обязан выглядеть симпатичным, но временами он может помочь в отладке.

## Правила TF Function

Большую часть времени преобразование функции Python, выполняющей операции TensorFlow, в функцию TF Function оказывается тривиальным: декорируйте ее с помощью `@tf.function` или позвольте Keras позаботится о ней. Однако существует несколько правил, которые необходимо соблюдать.

- Если вы обращаетесь к любой внешней библиотеке, включая NumPy или даже стандартную библиотеку, то такой вызов будет выполняться только во время трассировки; он не будет частью графа. На самом деле график TensorFlow может содержать только конструкции TensorFlow

(тензоры, операции, переменные, набора данных и т.д.). Таким образом, удостоверьтесь в том, что применяете `tf.reduce_sum()` вместо `np.sum()`, `tf.sort()` вместо встроенной функции `sorted()` и т.д. (если только вас действительно не интересует код для запуска только во время трассировки). Отсюда возникает несколько дополнительных последствий.

- Если вы определите функцию TF Function по имени `f(x)`, которая просто возвращает `np.random.rand()`, то случайное число будет генерироваться только при трассировке функции, так что вызовы `f(tf.constant(2.))` и `f(tf.constant(3.))` возвратят то же самое случайное число, но вызов `f(tf.constant([2., 3.]))` возвратит другое случайное число. Если вы замените `np.random.rand()` функцией `tf.random.uniform([])`, тогда при каждом вызове будет генерироваться новое случайное число, поскольку операция стала частью графа.
- Если ваш код, не работающий с TensorFlow, имеет побочные эффекты (наподобие регистрации в журнале чего-либо или обновления счетчика Python), то вам не следует рассчитывать на такие побочные эффекты при каждом вызове функции TF Function, т.к. они произойдут только во время трассировки функции.
- Вы можете поместить внутрь операции `tf.py_function()` произвольный код Python, но это будет снижать производительность, т.к. TensorFlow не имеет возможности проводить оптимизацию графа для подобного кода. Вдобавок снизится переносимость, потому что график будет выполняться только на платформах, где доступен Python (и установлены надлежащие библиотеки).
- Вы можете вызывать другие функции Python или TF Function, но они должны следовать тем же самым правилам, т.к. TensorFlow будет собирать их операции в вычислительный график. Обратите внимание, что вызываемые другие функции не нуждаются в декорировании с помощью `@tf.function`.
- Если функция создает переменную TensorFlow (или любой другой объект TensorFlow, сохраняющий состояние, такой как набор данных или очередь), то она должна делать это при самом первом вызове и только тогда, иначе возникнет исключение. Обычно предпочтительнее со-

здавать переменные вне функции TF Function (например, в методе `build()` специального слоя). Если вы хотите присвоить переменной новое значение, то обязательно вызывайте ее метод `assign()`, а не используйте операцию `=`.

- Исходный код вашей функции Python должен быть доступным для TensorFlow. Если исходный код недоступен (скажем, из-за того, что вы определили функцию в командной оболочке Python, которая не предоставляет доступ к исходному коду, или развернули скомпилированные файлы \*.pyc в производственной среде), тогда процесс генерации графа потерпит неудачу или график будет иметь ограниченную функциональность.
- TensorFlow будет собирать только такие циклы `for`, которые проходят по тензору или набору данных. Следовательно, удостоверьтесь в том, что применяете `for i in tf.range(x)`, а не `for i in range(x)`, в противном случае цикл не будет собран в график. Взамен он выполнится во время трассировки. (Это может оказаться именно тем, что требуется, если цикл `for` предназначен для построения графа, например, с целью создания каждого слоя в нейронной сети.)
- Как всегда, по причинам, связанным с производительностью, по возможности вы должны отдавать предпочтение векторизованной реализации, а не использованию циклов.

Пришло время подвести итоги! В настоящей главе мы начали с краткого обзора библиотеки TensorFlow, а затем ознакомились с низкоуровневым API-интерфейсом TensorFlow, включая тензоры, операции, переменные и специальные структуры данных. Далее мы применяли эти инструменты для настройки почти каждого компонента в `tf.keras`. В заключение мы выяснили, каким образом функции TF Function могут повышать производительность, как генерируются графы с использованием AutoGraph и трассировки и какие правила должны соблюдаться при написании функций TF Function (если вы хотите еще глубже заглянуть в черный ящик, скажем, исследовать сгенерированные графы, то вы найдете технические детали в приложении E).

В следующей главе мы посмотрим, каким образом эффективно загружать и предварительно обрабатывать данные с помощью TensorFlow.

# Упражнения

1. Как бы вы кратко описали библиотеку TensorFlow? Каковы ее главные возможности? Можете ли вы назвать другие популярные библиотеки для глубокого обучения?
2. Является ли TensorFlow готовой заменой NumPy? Каковы основные отличия между ними?
3. Получите ли вы одинаковые результаты с помощью `tf.range(10)` и `tf.constant(np.arange(10))`?
4. Можете ли вы назвать шесть других структур данных, доступных в TensorFlow, помимо обычновенных тензоров?
5. Специальная функция потерь может быть определена путем написания функции или создания подкласса класса `keras.losses.Loss`. Когда бы вы применяли каждый вариант?
6. Аналогично специальная метрика может быть определена в функции и в подклассе класса `keras.metrics.Metric`. Когда бы вы использовали каждый вариант?
7. Когда должен создаваться специальный слой в противоположность специальной модели?
8. Какие сценарии использования требуют написания собственного специального цикла обучения?
9. Могут ли специальные компоненты Keras содержать произвольный код Python или же они должны быть поддающимися преобразованию в функции TF Function?
10. Каковы главные правила, подлежащие соблюдению, если вы хотите, чтобы функция была преобразуемой в функцию TF Function?
11. Когда может возникнуть необходимость в создании динамической модели Keras? Как вы это сделаете? Почему не следует делать все ваши модели динамическими?
12. Реализуйте специальный слой, который выполняет нормализацию по слою (*Layer Normalization*; мы будем применять такой тип слоя в главе 15).
  - а) Метод `build()` должен определять два обучаемых веса  $\alpha$  и  $\beta$ , оба имеющие форму `input_shape[-1:]` и тип данных `tf.float32`. Вес  $\alpha$  должен быть инициализирован единицами, а вес  $\beta$  — нулями.

- 6) Метод `call()` должен вычислять среднее  $\mu$  и стандартное отклонение  $\sigma$  признаков каждого образца. Для этого мы можем использовать функцию `tf.nn.moments(inputs, axes=-1, keepdims=True)`, которая возвращает среднее  $\mu$  и дисперсию  $\sigma^2$  всех образцов (чтобы получить стандартное отклонение, вычислите квадратный корень из дисперсии). Затем функция должна вычислить и возвратить  $\alpha \otimes (X - \mu) / (\sigma + \epsilon) + \beta$ , где  $\otimes$  представляет поэлементное умножение ( $*$ ), а  $\epsilon$  является сглаживающим членом (маленькой константой, позволяющей избежать деления на ноль, скажем, 0.001).
- в) Удостоверьтесь в том, что ваш специальный слой производит тот же самый (или очень близкий) выход, что и слой `keras.layers.LayerNormalization`.
13. Обучите модель с применением специального цикла обучения, чтобы заняться набором данных Fashion MNIST (см. главу 10).
- Отображайте эпоху, итерацию, среднюю потерю при обучении и среднюю правильность по каждой эпохе (обновляемую на каждой итерации), а также потерю при проверке и правильность в конце каждой эпохи.
  - Попробуйте воспользоваться другим оптимизатором с отличающейся скоростью обучения для верхних и нижних слоев.

Решения приведенных упражнений доступны в приложении А.

# Загрузка и предварительная обработка данных с помощью TensorFlow

До сих пор мы использовали только наборы данных, умещающиеся в памяти, но системы глубокого обучения часто обучаются на очень крупных наборах данных, которые не будут умещаться в ОЗУ. Поглощение крупного набора данных и его эффективная предварительная обработка могут оказаться сложными в реализации с помощью других библиотек для глубокого обучения, но TensorFlow облегчает задачу благодаря API-интерфейсу *Data*: вы просто создаете объект набора данных, после чего сообщаете ему, где получить данные и как их трансформировать. Библиотека TensorFlow позаботится обо всех деталях реализации, таких как многопоточная обработка, организация очередей, формирование пакетов и предварительная выборка. Более того, API-интерфейс *Data* гладко работает с `tf.keras`!

Готовый API-интерфейс *Data* способен выполнять чтение из текстовых файлов (наподобие файлов CSV), двоичных файлов с записями фиксированного размера и двоичных файлов в формате TFRecord библиотеки TensorFlow, который поддерживает записи варьирующихся размеров. TFRecord представляет собой гибкий и эффективный двоичный формат, основанный на протокольных буферах (*Protocol Buffer*; двоичный формат с открытым кодом). Кроме того, API-интерфейс *Data* поддерживает чтение из баз данных SQL. Вдобавок доступно множество расширений с открытым кодом для чтения из всех видов источников данных, таких как служба BigQuery от Google.

Эффективное чтение гигантских наборов данных — не единственная трудность: данные также нуждаются в предварительной обработке, обычно нормализации. Более того, они не всегда состоят исключительно из удобных числовых полей: могут встречаться текстовые признаки, категориаль-

ные признаки и т.д. Их необходимо кодировать, например, с применением унитарного кодирования, кодирования суммированием слов в мешок (*bag-of-words encoding*) или вложений (*embedding*); как мы увидим, вложение — это обучаемый плотный вектор, который представляет категорию или маркер. Один вариант поддержки всей обработки такого рода предусматривает написание собственных специальных слоев предварительной обработки. Другой вариант предполагает использование стандартных слоев предварительной обработки, предлагаемых Keras.

В настоящей главе мы раскроем API-интерфейс Data, формат TFRecord, а также способы создания специальных слоев предварительной обработки и применения стандартных таких слоев из Keras. Кроме того, мы бегло взглянем на несколько связанных проектов из экосистемы TensorFlow, которые перечислены ниже.

### *TF Transform (tf.Transform)*

Позволяет написать единственную функцию предварительной обработки, которую можно запускать в пакетном режиме на полном обучавшем наборе перед обучением (для его ускорения), затем экспортить ее в функцию TF Function и встроить в обученную модель, чтобы после развертывания модели в производственной среде она могла позаботиться о предварительной обработке новых образцов на лету.

### *TF Datasets (TFDS; наборы данных TensorFlow)*

Предоставляет удобную функцию для загрузки множества распространенных наборов данных всех видов, включая крупные наборы данных вроде ImageNet, а также удобные объекты наборов данных для манипулирования ими с использованием API-интерфейса Data.

Итак, начнем!

## API-интерфейс Data

Весь API-интерфейс Data вращается вокруг концепции набора данных (*dataset*): как и можно было ожидать, он представляет последовательность элементов данных. В большинстве случаев вы будете применять наборы данных, которые понемногу читают данные из диска, но для простоты давайте создадим набор данных полностью в ОЗУ, используя метод `tf.data.Dataset.from_tensor_slices()`:

```
>>> X = tf.range(10)      # любой тензор данных
>>> dataset = tf.data.Dataset.from_tensor_slices(X)
>>> dataset
TensorSliceDataset shapes: (), types: tf.int32>
```

Функция `from_tensor_slices()` принимает тензор и создает объект `tf.data.Dataset`, все элементы которого являются срезами X (вдоль первого измерения), так что набор данных содержит 10 элементов: тензоры 0, 1, 2, ..., 9. В этом случае мы могли бы получить такой же набор данных с применением `tf.data.Dataset.range(10)`.

Проходить по элементам набора данных очень просто:

```
>>> for item in dataset:
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int32)
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
[...]
tf.Tensor(9, shape=(), dtype=int32)
```

## Формирование цепочки трансформаций

Имея набор данных, к нему можно применять все виды трансформаций за счет вызова его методов трансформаций. Каждый метод возвращает новый набор данных, так что трансформации можно выстраивать в цепочку (которая иллюстрируется на рис. 13.1):

```
>>> dataset = dataset.repeat(3).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 1 2 3 4 5 6], shape=(7,), dtype=int32)
tf.Tensor([7 8 9 0 1 2 3], shape=(7,), dtype=int32)
tf.Tensor([4 5 6 7 8 9 0], shape=(7,), dtype=int32)
tf.Tensor([1 2 3 4 5 6 7], shape=(7,), dtype=int32)
tf.Tensor([8 9], shape=(2,), dtype=int32)
```

В приведенном примере мы сначала вызываем метод `repeat()` на исходном наборе данных и он возвращает новый набор данных, который будет повторять элементы исходного набора данных три раза. Разумеется, при этом все данные не будут три раза копироваться в памяти! (Если вызвать метод `repeat()` без аргументов, то новый набор данных будет повторять

исходный набор данных нескончаемо долго, поэтому в коде, который проходит по набору данных, придется принимать решение, когда остановиться.) Затем мы вызываем метод `batch()` на новом наборе и данных и снова создается новый набор данных, который сгруппирует элементы предыдущего набора данных в пакеты по семь элементов. Наконец, мы проходим по элементам финального набора данных. Как видите, метод `batch()` должен выдать последний пакет размером два, а не семь, но вы можете вызвать его с `drop_remainder=True`, если хотите, чтобы он отбросил этот последний пакет, так что все пакеты будут иметь одинаковые размеры.

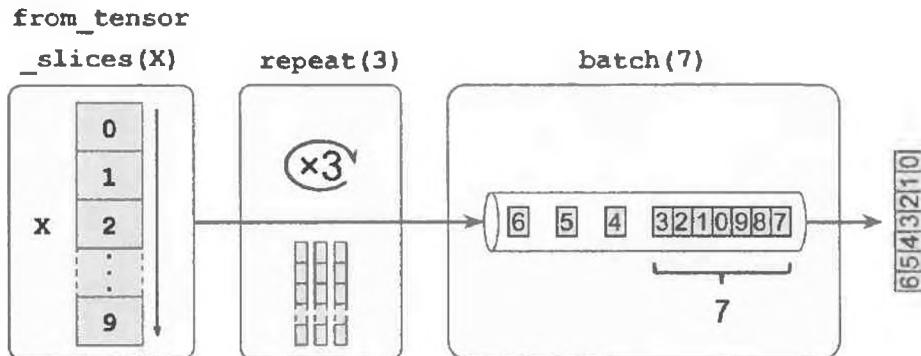


Рис. 13.1. Формирование цепочки трансформаций набора данных



Методы наборов данных *не* модифицируют наборы данных, они создают новые наборы данных, так что обязательно сохраняйте ссылки на них (скажем, посредством `dataset = ...`), иначе ничего не произойдет.

Вы также можете трансформировать элементы, вызывая метод `map()`. Например, следующий вызов создает новый набор данных, в котором все элементы продублированы:

```
>>> dataset = dataset.map(lambda x: x * 2)      # элементы:  
# [0, 2, 4, 6, 8, 10, 12]
```

Метод `map()` предназначен для применения любой необходимой предварительной обработки данных. Иногда это будет включать вычисления, которые могут быть довольно интенсивными, скажем, изменение формы или поворот изображения, так что обычно желательно породить множество потоков, чтобы ускорить выполнение: все сводится просто к установке аргумента `num_parallel_calls`.

мента num\_parallel\_calls. Обратите внимание, что функция, передаваемая методу map(), должна быть преобразуемой в функцию TF Function (см. главу 12).

В то время как метод map() применяет трансформацию к каждому элементу, метод apply() применяет трансформацию к набору данных как к единому целому. Например, показанный ниже код применяет к набору данных функцию unbatch() (в настоящий момент она экспериментальна, но с высокой вероятностью станет частью основного API-интерфейса в будущем выпуске). Каждый элемент в новом наборе данных будет тензором с единственным целым числом, а не пакетом из семи целых чисел:

```
>>> dataset = dataset.apply(tf.data.experimental.unbatch())
      # элементы: 0,2,4,...
```

Можно также просто отфильтровать набор данных с использованием метода filter():

```
>>> dataset = dataset.filter(lambda x: x < 10)    # элементы: 0 2 4 6
      #     8 0 2 4 6...
```

Часто желательно видеть лишь несколько элементов из набора данных. Для этого можете применить метод take():

```
>>> for item in dataset.take(3):
...     print(item)
...
tf.Tensor(0, shape=(), dtype=int64)
tf.Tensor(2, shape=(), dtype=int64)
tf.Tensor(4, shape=(), dtype=int64)
```

## Тасование данных

Как вам известно, градиентный спуск хорошо работает, когда образцы в обучающем наборе являются независимыми и идентично распределенными (см. главу 4). Простой способ обеспечить такие условия предусматривает тасование образцов с использованием метода shuffle(). Он создаст новый набор данных, который начнет с заполнения буфера первыми элементами исходного набора данных. Затем всякий раз, когда у набора данных запрашивается элемент, он будет случайным образом извлекать один элемент из буфера и заменять его свежим элементом из исходного набора данных до тех пор, пока не пройдет через весь исходный набор данных. В этой точке случайное извлечение из буфера продолжится вплоть до его опустошения.

Вы должны задать размер буфера, и важно сделать его достаточно большим, иначе тасование окажется не особенно эффективным<sup>1</sup>. Всего лишь не превышайте имеющийся объем ОЗУ, и даже если его в достатке, нет никакой необходимости выходить за рамки размера набора данных. Если вас интересует один и тот же случайный порядок при каждом запуске программы, тогда можете указать случайное начальное число. Скажем, следующий код создает и отображает набор данных, который содержит целые числа от 0 до 9, повторенные 3 раза, перетасованные с применением буфера размера 5 и случайного начального числа 42, а также сгруппированные в пакеты размером 7:

```
>>> dataset = tf.data.Dataset.range(10).repeat(3) #от 0 до 9, три раза
>>> dataset = dataset.shuffle(buffer_size=5, seed=42).batch(7)
>>> for item in dataset:
...     print(item)
...
tf.Tensor([0 2 3 6 7 9 4], shape=(7,), dtype=int64)
tf.Tensor([5 0 1 1 8 6 5], shape=(7,), dtype=int64)
tf.Tensor([4 8 7 1 2 3 0], shape=(7,), dtype=int64)
tf.Tensor([5 4 2 7 8 9 9], shape=(7,), dtype=int64)
tf.Tensor([3 6], shape=(2,), dtype=int64)
```



В случае вызова метода `repeat()` на перетасованном наборе данных по умолчанию на каждой итерации будет генерироваться новый порядок. Обычно это хорошая идея, но если вы предпочитаете повторно использовать на каждой итерации тот же самый порядок (например, при тестировании или отладке), тогда можете установить `reshuffle_each_iteration=False`.

Для крупного набора данных, который не умещается в памяти, такой простой подход с тасуемым буфером может оказаться неэффективным, поскольку буфер будет мал в сравнении с набором данных. Одним из решений является тасование самих исходных данных (скажем, в среде Linux с приме-

---

<sup>1</sup> Представьте себе отсортированную колоду карт слева от вас: пусть вы берете верхние три карты, тасуете их, затем наугад вытаскиваете одну карту и помещаете ее справа от себя, оставляя другие две в своих руках. Возьмите еще одну карту слева от себя, перетасуйте три карты в своих руках, выберите одну из них наугад и поместите справа от себя. По завершении таких манипуляций со всеми картами справа от вас будет колода карт: как вы думаете, будет ли она идеально перетасована?

нением команды `shuf` можно тасовать текстовые файлы). Прием определенно намного улучшит тасование! Даже когда исходные данные перетасованы, в большинстве случаев вы захотите тасовать их еще, иначе на каждой эпохе будет повторяться тот же самый порядок и модель может в итоге стать смещённой (например, из-за ряда ложных паттернов, случайно образовавшихся в порядке исходных данных). Распространенный подход к дополнительному тасованию образцов предусматривает разделение исходных данных на множество файлов и последующее их чтение в случайному порядке во время обучения. Однако образцы, находящиеся в одном файле, по-прежнему оказываются близкими друг к другу. Чтобы избежать подобной ситуации, вы можете случайным образом выбирать ряд файлов и читать их одновременно, чередуя записи из них. Затем дополнительно вы можете добавить буфер тасования, используя метод `shuffle()`. Если указанные действия кажутся связанными с огромным объемом работы, то не переживайте: API-интерфейс Data делает все это возможным за счет написания лишь нескольких строк кода. Давайте взглянем на реализацию.

### Чередование строк из множества файлов

Первым делом предположим, что мы загрузили набор данных с ценами на жилье в Калифорнии, перетасовали его (если он не был перетасован) и расщепили на обучающий набор, проверочный набор и испытательный набор. Далее мы расщепляем каждый набор на множество файлов CSV, содержимое которых выглядит следующим образом (каждая строка содержит восемь входных признаков плюс целевая медианная стоимость дома):

```
MedInc,HouseAge,AveRooms,AveBedrms,Popul,AveOccup,Lat,Long,  
MedianHouseValue  
3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442  
5.3275,5.0,6.4900,0.9910,3464.0,3.4433,33.69,-117.39,1.687  
3.1,29.0,7.5423,1.5915,1328.0,2.2508,38.44,-122.98,1.621  
[...]
```

Давайте представим, что `train_filepaths` содержит список путей к обучающим файлам (и также есть `valid_filepaths` и `test_filepaths`):

```
>>> train_filepaths  
['datasets/housing/my_train_00.csv',  
'datasets/housing/my_train_01.csv',...]
```

В качестве альтернативы можно было бы применять шаблоны имен файлов, скажем, `train_filepaths = "data_sets/housing/my_train_*.csv"`. Теперь создадим набор данных, содержащий только пути к файлам:

```
filepath_dataset = tf.data.Dataset.list_files(train_filepaths, seed=42)
```

По умолчанию функция `list_files()` возвращает набор данных с перетасованными путями к файлам. В целом это хорошо, но можно установить `shuffle=False`, если по какой-то причине тасование нежелательно.

Затем мы вызываем метод `interleave()` для чтения из пяти файлов одновременно и чередования их строк (с помощью метода `skip()` пропуская первую строку каждого файла, которая является заголовочной):

```
n_readers = 5
dataset = filepath_dataset.interleave(
    lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
    cycle_length=n_readers)
```

Метод `interleave()` создаст набор данных, который будет извлекать из `filepath_dataset` пять путей к файлам и для каждого из них вызывать указанную функцию (в рассматриваемом примере лямбда-функцию) для создания нового набора данных (в примере `TextLineDataset`). Точнее говоря, на этой стадии всего будет семь наборов данных: набор путей к файлам, набор чередования и пять наборов `TextLineDataset`, создаваемых внутренне набором чередования. Когда мы проходим по набору чередования, он циклически проходит по пяти наборам `TextLineDataset`, читая из каждого по одной строке за раз до тех пор, пока не закончатся элементы во всех наборах данных. Далее он извлекает из `filepath_dataset` следующие пять путей к файлам и чередует их таким же способом, продолжая вплоть до исчерпания путей к файлам.



Чтобы чередование работало лучше, предпочтительно иметь файлы одинаковой длины; в противном случае строки в конце самых длинных файлов чередовать не будут.

По умолчанию метод `interleave()` не использует параллелизм; он просто последовательно читает из каждого файла по одной строке за раз. Если вы хотите, чтобы метод `interleave()` читал файлы по-настоящему параллельно, тогда можете указать в аргументе `num_parallel_calls` желаемое количество потоков (обратите внимание, что метод `map()` тоже имеет та-

кой аргумент). Вы даже можете установить его в `tf.data.experimental.AUTOTUNE`, заставив TensorFlow выбирать правильное количество потоков динамически на основе имеющегося в распоряжении центрального процессора (тем не менее, пока что эта возможность является экспериментальной). Давайте посмотрим, что сейчас содержит набор данных:

```
>>> for line in dataset.take(5):
...     print(line.numpy())
...
'b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,-122.2,2.782'
'b'4.1812,52.0,5.7013,0.9965,692.0,2.4027,33.73,-118.31,3.215'
'b'3.6875,44.0,4.5244,0.9930,457.0,3.1958,34.04,-118.15,1.625'
'b'3.3456,37.0,4.5140,0.9084,458.0,3.2253,36.67,-121.7,2.526'
'b'3.5214,15.0,3.0499,1.1065,1447.0,1.6059,37.63,-122.43,1.442'
```

Мы видим первые строки (с игнорированием заголовочной строки) пяти файлов CSV, выбранные случайно. Выглядит неплохо! Но легко заметить, что они представляют собой байтовые строки; нам необходимо провести их разбор и масштабировать данные.

## Предварительная обработка данных

Давайте реализуем небольшую функцию, которая будет выполнять такую предварительную обработку:

```
X_mean, X_std = [...]      # среднее и масштаб каждого признака
                           # в обучающем наборе
n_inputs = 8

def preprocess(line):
    defs = [0.] * n_inputs + [tf.constant([], dtype=tf.float32)]
    fields = tf.io.decode_csv(line, record_defaults=defs)
    x = tf.stack(fields[:-1])
    y = tf.stack(fields[-1:])
    return (x - X_mean) / X_std, y
```

Разберем приведенный код.

- Сначала в коде предполагается, что мы имеем заранее вычисленные среднюю величину и стандартное отклонение каждого признака в обучающем наборе. `X_mean` и `X_std` — это просто одномерные тензоры (или массивы NumPy), содержащие восемь чисел с плавающей точкой, по одной на входной признак.

- Функция `preprocess()` берет одну строку CSV и начинает с ее разбора, применяя функцию `tf.io.decode_csv()`, которая принимает два аргумента: разбираемую строку и массив со стандартными значениями для каждого столбца в файле CSV. Такой массив сообщает TensorFlow не только стандартное значение для каждого столбца, но также количество столбцов и их типы. В рассматриваемом примере мы указываем, что все столбцы признаков являются числами с плавающей точкой, а отсутствующие значения должны по умолчанию устанавливаться в 0, но предоставляем пустой массив типа `tf.float32` в качестве стандартного значения для последнего столбца (цели). Этот массив сообщает TensorFlow о том, что последний столбец содержит числа с плавающей точкой, но стандартных значений нет, так что при отсутствии значения должно генерироваться исключение.
- Функция `decode_csv()` возвращает список скалярных тензоров (по одному на столбец), но нам нужно возвращать массивы одномерных тензоров. Таким образом, мы вызываем функцию `tf.stack()` на всех тензорах кроме последнего (цели): она уложит тензоры стопкой в одномерный массив. Затем мы делаем то же самое для целевого значения (что превращает его в массив одномерных тензоров с единственным значением, а не скалярный тензор).
- В заключение мы масштабируем входные признаки, для чего вычитаем средние величины признаков и делим на стандартные отклонения признаков, и возвращаем кортеж, содержащий масштабированные признаки и цель.

Давайте протестируем функцию предварительной обработки `preprocess()`:

```
>>> preprocess(b'4.2083,44.0,5.3232,0.9171,846.0,2.3370,37.47,  
-122.2,2.782')  
(<tf.Tensor: id=6227, shape=(8,), dtype=float32, numpy=  
array([ 0.16579159,  1.216324 , -0.05204564, -0.39215982, -0.5277444 ,  
       -0.2633488 ,  0.8543046, -1.3072058 ], dtype=float32)>,  
<tf.Tensor: [...], numpy=array([2.782], dtype=float32)>)
```

Смотрится хорошо! Теперь мы можем применить эту функцию к набору данных.

## Собираем все вместе

Чтобы сделать код многократно используемым, давайте соберем все, что мы обсуждали до сих пор, внутри небольшой вспомогательной функции. Она будет создавать и возвращать набор данных, который эффективно загрузит данные с ценами на жилье в Калифорнии из множества файлов CSV, предварительно обработает их, перетасует, дополнительно повторит и сгруппирует в пакеты (рис. 13.2):

```
def csv_reader_dataset(filepaths, repeat=1, n_readers=5,
                      n_read_threads=None, shuffle_buffer_size=10000,
                      n_parse_threads=5, batch_size=32):
    dataset = tf.data.Dataset.list_files(filepaths)
    dataset = dataset.interleave(
        lambda filepath: tf.data.TextLineDataset(filepath).skip(1),
        cycle_length=n_readers, num_parallel_calls=n_read_threads)
    dataset = dataset.map(preprocess, num_parallel_calls=n_parse_threads)
    dataset = dataset.shuffle(shuffle_buffer_size).repeat(repeat)
    return dataset.batch(batch_size).prefetch(1)
```

В коде все должно быть понятным кроме самой последней строки (`prefetch(1)`), которая важна для производительности.

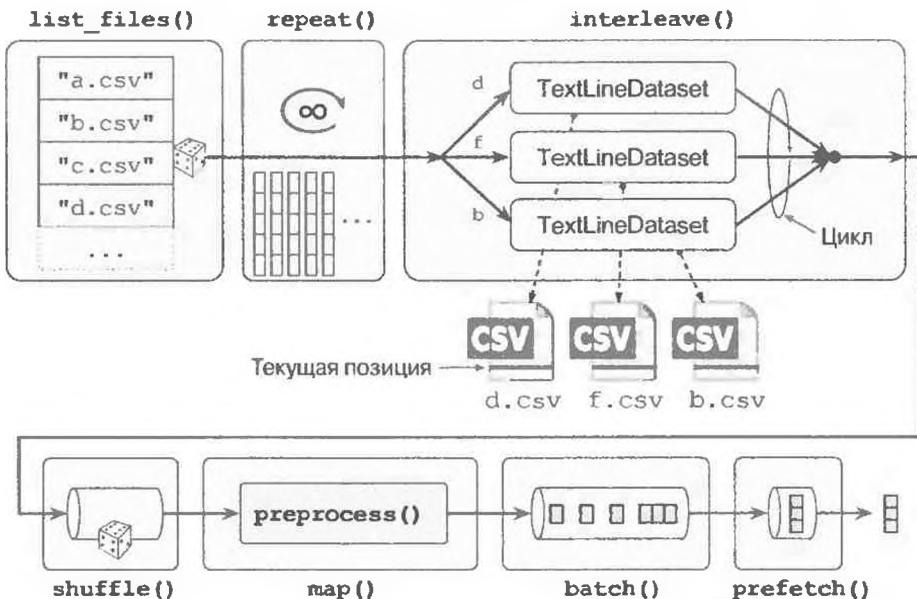
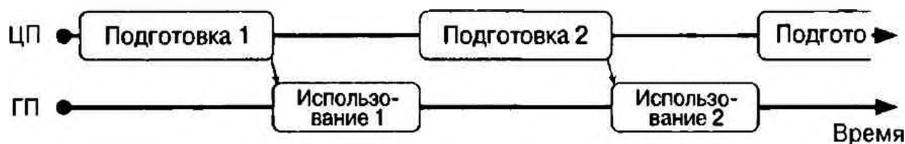


Рис. 13.2. Загрузка и предварительная обработка данных из множества файлов CSV

## Предварительная выборка

Вызывая в конце `prefetch(1)`, мы создаем набор данных, который будет делать все возможное, чтобы всегда находиться на один пакет впереди<sup>2</sup>. Другими словами, пока наш алгоритм обучения работает с одним пакетом, набор данных уже будет параллельно работать над подготовкой следующего пакета (например, читая данные с диска и предварительно обрабатывая их). Такой подход способен значительно улучшить производительность, как демонстрируется на рис. 13.3.

Без предварительной выборки



С предварительной выборкой



С предварительной выборкой, многопоточной загрузкой и предварительной обработкой



Рис. 13.3. При наличии предварительной выборки ЦП и ГП работают параллельно: пока ГП работает с одним пакетом, ЦП работает со следующим

<sup>2</sup> В общем случае предварительной выборки всего одного пакета достаточно, но в некоторых ситуациях может возникать необходимость в предварительной выборке большего их количества. В качестве альтернативы можно позволить TensorFlow принимать решение автоматически, передавая `tf.data.experimental.AUTOTUNE` (пока что это экспериментальная возможность).

Если мы также гарантируем, что загрузка и предварительная обработка будут многопоточными (установив `num_parallel_calls` при вызове `interleave()` и `map()`), то сможем задействовать множество ядер центрального процессора (ЦП) и надо надеяться сделать подготовку одного пакета данных короче выполнения шага обучения в графическом процессоре (ГП): в таком случае ГП будет использоваться почти на 100% (за исключением времени передачи данных из ЦП в ГП<sup>3</sup>), а обучение станет проходить гораздо быстрее.



Если вы планируете приобрести плату ГП, то ее вычислительная мощность и емкость памяти, конечно же, очень важны (в частности, большой объем ОЗУ является решающим для компьютерного зрения). Для обеспечения высокой производительности в той же степени важна *пропускная способность памяти* платы ГП, т.е. количество гигабайтов данных, которые она может получать или передавать в ОЗУ за секунду.

Если набор данных достаточно мал, чтобы уместиться в память, тогда вы можете значительно ускорить обучение за счет применения метода `cache()` набора данных для кэширования его содержимого в ОЗУ. Обычно поступать так следует после загрузки и предварительной обработки данных, но перед тасованием, повторением, группированием в пакеты и предварительной выборкой. В результате каждый образец будет прочитан и предварительно обработан всего лишь один раз (а не раз за эпоху), но данные по-прежнему тасуются по-разному на каждой эпохе и следующий пакет все еще подготавливается заранее.

Итак, вы знаете, как строить эффективные входные конвейеры для загрузки и предварительной обработки данных из множества текстовых файлов. Мы обсудили наиболее распространенные методы наборов данных, но существуют и другие методы, на которые вы можете взглянуть: `concatenate()`, `zip()`, `window()`, `reduce()`, `shard()`, `flat_map()` и `padded_batch()`. Также есть еще пара методов класса: `from_generator()` и `from_tensors()`, которые создают новый набор данных соответственно из генератора Python и из списка тензоров. Дополнительные сведения ищите в документации по

<sup>3</sup> Но обратите внимание на функцию `tf.data.experimental.prefetch_to_device()`, которая способна предварительно выбирать данные с передачей их прямо в ГП.

API-интерфейсу. Кроме того, обратите внимание на экспериментальные возможности, доступные в `tf.data.experimental`, многие из которых с высокой вероятностью станут частью основного API-интерфейса в будущих выпусках (скажем, ознакомьтесь с классом `CsvDataset`, а также с методом `make_csv_dataset()`, который позаботится о выведении типа для каждого столбца).

## Использование набора данных с библиотекой `tf.keras`

Теперь мы можем применить функцию `csv_reader_dataset()`, чтобы создать набор данных для обучающего набора. Обратите внимание, что повторять его не нужно, т.к. об этом позаботится `tf.keras`. Мы также создаем наборы данных для проверочного набора и испытательного набора:

```
train_set = csv_reader_dataset(train_filepaths)
valid_set = csv_reader_dataset(valid_filepaths)
test_set = csv_reader_dataset(test_filepaths)
```

А сейчас мы можем просто построить и обучить модель Keras, используя созданные наборы данных<sup>4</sup>. Нам понадобится лишь передать методу `fit()` обучающий и проверочный наборы данных вместо `X_train`, `y_train`, `X_valid` и `y_valid`<sup>5</sup>:

```
model = keras.models.Sequential([...])
model.compile([...])
model.fit(train_set, epochs=10, validation_data=valid_set)
```

Подобным образом мы можем передавать набор данных методам `evaluate()` и `predict()`:

```
model.evaluate(test_set)
new_set = test_set.take(3).map(lambda X, y: X) # притвориться, что
                                                # у нас есть 3 новых образца
model.predict(new_set) # набор данных, содержащий новые образцы
```

<sup>4</sup> Поддержка наборов данных специфична для `tf.keras`; в других реализациях API-интерфейса Keras она не работает.

<sup>5</sup> Метод `fit()` позаботится о повторении обучающего набора данных. В качестве альтернативы вы могли бы вызвать `repeat()` на обучающем наборе данных, так что он будет повторяться нескончаемо, и указать аргумент `steps_per_epoch` при вызове метода `fit()`. Это может быть удобно в редких случаях, например, если вы хотите использовать буфер тасования, который пересекает эпохи.

В отличие от других наборов `new_set` обычно не будет содержать метки (а если будет, то Keras проигнорирует их). Обратите внимание, что во всех этих случаях при желании вы по-прежнему можете применять вместо наборов данных массивы NumPy (но, разумеется, сначала они должны быть загружены и предварительно обработаны).

Если вы хотите построить собственный специальный цикл обучения (как в главе 12), тогда можете просто вполне естественным образом проходить по обучающему набору:

```
for X_batch, y_batch in train_set:  
    [...] # выполнить один шаг градиентного спуска
```

На самом деле можно даже создать функцию TF Function (см. главу 12), которая выполняет целый цикл обучения:

```
@tf.function  
def train(model, optimizer, loss_fn, n_epochs, [...]):  
    train_set = csv_reader_dataset(train_filepaths, repeat=n_epochs,  
    [...])  
    for X_batch, y_batch in train_set:  
        with tf.GradientTape() as tape:  
            y_pred = model(X_batch)  
            main_loss = tf.reduce_mean(loss_fn(y_batch, y_pred))  
            loss = tf.add_n([main_loss] + model.losses)  
            grads = tape.gradient(loss, model.trainable_variables)  
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Примите поздравления, теперь вы знаете, как строить мощные входные конвейеры с использованием API-интерфейса Data! Однако до сих пор мы применяли файлы CSV, которые распространены, просты и удобны, но в действительности не особо эффективны и не очень хорошо поддерживают крупные или сложные структуры данных (такие как изображения или аудиоклипы). Итак, давайте посмотрим, как вместо них использовать записи TFRecord.



Если вы довольны файлами CSV (или любым другим форматом, который применяете), то *не обязаны* использовать записи TFRecord. Как говорится, не пытайтесь ремонтировать то, что работает! Записи TFRecord удобны, когда узким местом во время обучения оказываются загрузка и разбор данных.

# Формат TFRecord

Формат TFRecord является предпочтительным форматом TensorFlow для хранения крупных объемов данных и их эффективного чтения. Он представляет собой очень простой двоичный формат, который содержит последовательность двоичных записей варьирующихся размеров (каждая запись заключает в себе длину, контрольную сумму CRC для проверки, не повреждена ли длина, фактические данные и контрольную сумму CRC для самих данных). Файл TFRecord легко создать с применением класса `tf.io.TFRecordWriter`:

```
with tf.io.TFRecordWriter("my_data.tfrecord") as f:  
    f.write(b"This is the first record")  
    f.write(b"And this is the second record")
```

Затем можно использовать класс `tf.data.TFRecordDataset` для чтения одного и более файлов TFRecord:

```
filepaths = ["my_data.tfrecord"]  
dataset = tf.data.TFRecordDataset(filepaths)  
for item in dataset:  
    print(item)
```

Вот вывод:

```
tf.Tensor(b'This is the first record', shape=(), dtype=string)  
tf.Tensor(b'And this is the second record', shape=(), dtype=string)
```



По умолчанию объект `TFRecordDataset` будет читать файлы друг за другом, но вы можете заставить его читать множество файлов параллельно и чередовать их записи, установив `num_parallel_reads`. Альтернативно вы могли бы получить тот же самый результат с применением `list_files()` и `interleave()`, как делали ранее при чтении множества файлов CSV.

## Сжатые файлы TFRecord

Временами файлы TFRecord удобно сжимать, особенно если они должны загружаться через сетевое подключение. Создать сжатый файл TFRecord можно путем установки аргумента `options`:

```
options = tf.io.TFRecordOptions(compression_type="GZIP")  
with tf.io.TFRecordWriter("my_compressed.tfrecord", options) as f:  
    [...]
```

При чтении сжатого файла TFRecord необходимо указать тип сжатия:

```
dataset = tf.data.TFRecordDataset(["my_compressed.tfrecord"]),
      compression_type="GZIP")
```

## Краткое введение в протокольные буферы

Хотя каждая запись может использовать любой желаемый формат, файлы TFRecord обычно содержат сериализованные протокольные буферы (также называемые *протобуферами (protobuf)*). Они представляют собой переносимый, расширяемый и эффективный формат, разработанный Google еще в 2001 году, а в 2008 году ставший проектом с открытым кодом; в настоящее время протобуферы широко применяются, в частности в gRPC (<https://grpc.io>) — системе удаленного вызова процедур от компании Google. Протобуферы определяются с использованием простого языка, который выглядит следующим образом:

```
syntax = "proto3";
message Person {
    string name = 1;
    int32 id = 2;
    repeated string email = 3;
}
```

Определение говорит о том, что мы применяем версию 3 формата протобуферов, и указывает, что каждый объект Person<sup>6</sup> может (необязательно) иметь поле name типа string, поле id типа int32 и ноль или большее количество полей email типа string каждое. Числа 1, 2 и 3 являются идентификаторами полей: они будут использоваться в двоичном представлении каждой записи. Поместив определение в файл .proto, его можно скомпилировать. Для этого требуется компилятор протобуферов protoc, генерирующий классы доступа на Python (или на каком-то другом языке). Обратите внимание, что определения протобуферов, которые мы будем применять, уже были скомпилированы, а их классы Python входят в состав TensorFlow, поэтому использовать protoc не придется. Нужно лишь знать, как применять классы доступа протобуферов в Python. Чтобы проиллюстрировать основы, давайте рассмотрим простой пример, в котором используются классы доступа, сгенерированные для протобуфера Person (код объясняется в комментариях):

<sup>6</sup> Из-за того, что объекты протобуферов предназначены для сериализации и передачи, они называются сообщениями (messages).

```

>>> from           import Person      # импортирование сгенери-
                                         #рованного класса доступа
>>> person = Person(name="Al", id=123, email=["a@b.com"])
                                         # создание объекта Person
>>> print(person)      # отображение объекта Person
name: "Al"
id: 123
email: "a@b.com"
>>> person.name       # чтение записи
"Al"
>>> person.name = "Alice"        # модификация записи
>>> person.email[0]            # к повторяющимся записям можно
                                         # обращаться как к массивам
"a@b.com"
>>> person.email.append("c@d.com") # добавление адреса
                                         # электронной почты
>>> s = person.SerializeToString() # сериализация объекта
                                         # в байтовую строку
>>> s
b'\n\x05Alice\x10(\x1a\x07a@b.com\x1a\x07c@d.com'
>>> person2 = Person()    # создание нового объекта Person
>>> person2.ParseFromString(s) # разбор байтовой строки
                                         # (длиной 27 байтов)
27
>>> person == person2     # теперь они эквивалентны
True

```

Выражаясь кратко, мы импортируем класс Person, сгенерированный компилятором protoc, создаем экземпляр Person и проводим с ним различные манипуляции, визуализируя его, а также читая и записывая поля, после чего сериализуем экземпляр Person с применением метода `SerializeToString()`. В итоге получаются двоичные данные, которые готовы к сохранению или передаче по сети. При чтении или получении таких двоичных данных мы можем провести их разбор, используя метод `ParseFromString()`, и получить копию объекта, который ранее был сериализован<sup>7</sup>.

Мы могли бы сохранить сериализованный объект Person в файл TFRecord, затем загрузить его и выполнить разбор: все работало бы нормально. Тем не менее, методы `SerializeToString()` и `ParseFromString()` не являются операциями TensorFlow (равно как и другими операциями в

---

<sup>7</sup> В главе предлагается абсолютный минимум того, что необходимо знать о протобуфах, для использования записей TFRecord. За дополнительными сведениями обращайтесь по ссылке <https://home1.info/protobuf>.

коде), так что они не могут быть включены в функцию TF Function (кроме их помещения внутрь операции `tf.py_function()`, которая сделает код медленным и менее переносимым, как объяснялось в главе 12). К счастью, TensorFlow содержит специальные определения протобуферов, для которых предоставляет операции разбора.

## Протобуферы TensorFlow

Основным протобуфером, обычно применяемым в файле TFRecord, является `Example`, который представляет один образец в наборе данных. Он содержит список именованных признаков, где каждый признак может быть списком байтовых строк, списком чисел с плавающей точкой или списком целых чисел. Вот определение такого протобуфера:

```
syntax = "proto3";
message BytesList { repeated bytes value = 1; }
message FloatList { repeated float value = 1 [packed = true]; }
message Int64List { repeated int64 value = 1 [packed = true]; }
message Feature {
    oneof kind {
        BytesList bytes_list = 1;
        FloatList float_list = 2;
        Int64List int64_list = 3;
    }
};
message Features { map<string, Feature> feature = 1; };
message Example { Features features = 1; };
```

Определения `BytesList`, `FloatList` и `Int64List` довольно прямолинейны. Обратите внимание, что для повторяющихся числовых полей используется `[packed = true]` для более эффективного кодирования. Поле `Feature` содержит `BytesList`, `FloatList` либо `Int64List`. Поле `Features` (с буквой `s`) содержит словарь, который сопоставляет имя признака со значением соответствующего признака. Наконец, поле `Example` содержит только объект `Features`<sup>8</sup>.

---

<sup>8</sup> А зачем вообще определять протобуфер `Example`, если он содержит не более, чем объект `Features`? Дело в том, что разработчики TensorFlow могут однажды принять решение добавить в него дополнительные поля. До тех пор, пока новое определение `Example` по-прежнему содержит поле `features` с тем же самым идентификатором, оно будет обратно совместимым. Такая расширяемость считается одной из замечательных характеристик протобуферов.

Ниже показано, как можно было бы создать объект `tf.train.Example`, представляющий того же человека, что и ранее, и записать его в файл `TFRecord`:

```
from import BytesList, FloatList, Int64List
from import Feature, Features, Example

person_example = Example(
    features=Features(
        feature={
            "name": Feature(bytes_list=BytesList(value=[b"Alice"])),
            "id": Feature(int64_list=Int64List(value=[123])),
            "emails": Feature(bytes_list=BytesList(value=[b"a@b.com",
                b"c@d.com"]))
        }
    )
)
```

Код выглядит слегка многословным и повторяющимся, но он довольно прямолинеен (и его легко можно было бы поместить внутрь небольшой вспомогательной функции). Имея протобуфер `Example`, мы можем сериализовать его, вызвав метод `SerializeToString()`, и затем записать результирующие данные в файл `TFRecord`:

```
with tf.io.TFRecordWriter("my_contacts.tfrecord") as f:
    f.write(person_example.SerializeToString())
```

Обычно вы будете записывать гораздо больше, чем один протобуфер `Example`! Как правило, вы создадите сценарий преобразования, который будет читать данные в текущем формате (скажем, из файлов CSV), создавать протобуфер `Example` для каждого образца, сериализовать его и сохранять в нескольких файлах `TFRecord`, в идеальном случае тасуя их в процессе. Задача требует определенной работы, поэтому еще раз удостоверьтесь в том, что она действительно необходима (возможно, ваш конвейер хорошо работает с файлами CSV).

Теперь при наличии файла `TFRecord`, содержащего сериализованный протобуфер `Example`, давайте попробуем его загрузить.

## Загрузка и разбор протобуферов `Example`

Для загрузки сериализованных протобуферов `Example` мы снова будем применять `tf.data.TFRecordDataset`, а для проведения разбора каждого протобуфера `Example` — использовать `tf.io.parse_single_example()`. Это операция TensorFlow и потому ее можно включить в функцию TF Function.

Она требует, по крайней мере, двух аргументов: строкового скалярного тензора, содержащего сериализированные данные, и описания каждого признака. Описание представляет собой словарь, который сопоставляет имя каждого признака либо с дескриптором `tf.io.FixedLenFeature`, указывающим форму, тип и стандартное значение признака, либо с дескриптором `tf.io.VarLenFeature`, указывающим только тип (если длина списка признака может варьироваться, как в случае признака "emails").

В следующем коде определяется словарь описания, после чего производится проход по набору данных `TFRecordDataset` и разбор сериализованного протобуфера `Example`, содержащегося в этом наборе данных:

```
feature_description = {
    "name": tf.io.FixedLenFeature([], tf.string, default_value=""),
    "id": tf.io.FixedLenFeature([], tf.int64, default_value=0),
    "emails": tf.io.VarLenFeature(tf.string),
}

for serialized_example in tf.data.TFRecordDataset(
    ["my_contacts.tfrecord"]):
    parsed_example = tf.io.parse_single_example(serialized_example,
                                                feature_description)
```

Признаки фиксированной длины разбираются как обычные тензоры, но признаки переменной длины — как разреженные тензоры. Можно преобразовать разреженный тензор в плотный с применением `tf.sparse.to_dense()`, но в данном случае проще всего лишь получить доступ к его значениям:

```
>>> tf.sparse.to_dense(parsed_example["emails"], default_value=b"")
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'],
[...])>
>>> parsed_example["emails"].values
<tf.Tensor: [...] dtype=string, numpy=array([b'a@b.com', b'c@d.com'],
[...])>
```

Поле `BytesList` способно содержать любые двоичные данные, включая любой сериализованный объект. Например, можно использовать `tf.io.encode_jpeg()` для кодирования изображения в формат JPEG и поместить результирующие двоичные данные в `BytesList`. Позже, когда код читает запись `TFRecord`, он начнет с разбора `Example`, а затем ему понадобится вызвать `tf.io.decode_jpeg()`, чтобы выполнить разбор данных и получить первоначальное изображение (или же можно применить функцию

`tf.io.decode_image()`, которая декодирует изображение в форматах BMP, GIF, JPEG и PNG). Кроме того, в `BytesList` можно сохранить любой тензор, сериализируя его с помощью `tf.io.serialize_tensor()` и затем помещая результирующую байтовую строку в поле `BytesList`. При последующем разборе записи `TFRecord` эти данные можно разобрать с использованием функции `tf.io.parse_tensor()`.

Вместо разбора протобуферов `Example` поодиночке с применением `tf.io.parse_single_example()` может быть желательно разбирать их пакетами, используя `tf.io.parse_example()`:

```
dataset = tf.data.TFRecordDataset(["my_contacts.tfrecord"]).batch(10)
for serialized_examples in dataset:
    parsed_examples = tf.io.parse_example(serialized_examples,
                                           feature_description)
```

Как видите, протобуфера `Example` вероятно будет достаточно для большинства сценариев применения. Однако может оказаться, что его несколько обременительно использовать, когда приходится иметь дело списками списков. Например, пусть необходимо классифицировать текстовые документы. Каждый документ может быть представлен в виде списка предложений, где каждое предложение представляется как список слов. Возможно также, что каждый документ имеет список комментариев, где каждый комментарий представлен в виде списка слов. Вдобавок могут существовать контекстные данные, такие как автор, название и дата публикации документа. Для сценариев применения подобного рода в библиотеке TensorFlow предусмотрен протобуфер `SequenceExample`.

## Обработка списка списков с использованием протобуфера `SequenceExample`

Вот определение протобуфера `SequenceExample`:

```
message FeatureList { repeated Feature feature = 1; }
message FeatureLists { map<string, FeatureList> feature_list = 1;
};
message SequenceExample {
    Features context = 1;
    FeatureLists feature_lists = 2;
};
```

Протобуфер `SequenceExample` содержит объект `Features` для контекстных данных и объект `FeatureLists`, который включает один или большее количество именованных объектов `FeatureList` (например, объект `FeatureList` по имени "content" и еще один по имени "comments"). Каждый объект `FeatureList` содержит список объектов `Feature`, каждый из которых может быть списком байтовых строк, списком 64-битных целых чисел либо списком чисел с плавающей точкой (в рассматриваемом примере каждый объект `Feature` будет представлять предложение или комментарий, возможно в форме списка идентификаторов слов). Построение протобуфера `SequenceExample`, его сериализация и разбор аналогичны построению, сериализации и разбору протобуфера `Example`, но для разбора одиночного протобуфера `SequenceExample` придется применять функцию `tf.io.parse_single_sequence_example()`, а для разбора пакета — функцию `tf.io.parse_sequence_example()`. Обе функции возвращают кортеж, содержащий контекстные признаки (в виде словаря) и списки признаков (тоже в виде словаря). Если списки признаков включают последовательности варьирующихся размеров (как в предыдущем примере), тогда вы можете преобразовать их в зубчатые тензоры, используя `tf.RaggedTensor.from_sparse()` (полный код ищите в тетради Jupyter для настоящей главы):

```
parsed_context,  
parsed_feature_lists = tf.io.parse_single_sequence_example(  
    serialized_sequence_example, context_feature_descriptions,  
    sequence_feature_descriptions)  
parsed_content =  
    tf.RaggedTensor.from_sparse(parsed_feature_lists["content"])
```

Теперь, когда вы знаете, как эффективно сохранять, загружать и выполнять разбор данных, следующим шагом является их подготовка, чтобы данные можно было подавать в нейронную сеть.

## Предварительная подготовка входных признаков

Подготовка данных для нейронной сети требует преобразования всех признаков в числовые признаки, как правило, с их нормализацией, и т.д. В частности, если ваши данные содержат категориальные или текстовые признаки, тогда они нуждаются в преобразовании в числа. Это можно делать заблаговременно при подготовке файлов данных с применением любого предпочтаемого инструмента (скажем, NumPy, pandas или Scikit-Learn).

В качестве альтернативы вы можете предварительно обрабатывать свои данные на лету во время их загрузки с помощью API-интерфейса Data (например, используя метод `map()` набора данных, как вы видели ранее) или включить слой предварительной подготовки прямо в модель. Давайте взглянем на последний вариант.

Например, ниже показано, как можно реализовать слой стандартизации с применением слоя Lambda. Для каждого признака он вычитает среднее и делит на его стандартное отклонение (плюс крошечный сглаживающий член во избежание деления на ноль):

```
means = np.mean(X_train, axis=0, keepdims=True)
stds = np.std(X_train, axis=0, keepdims=True)
eps = keras.backend.epsilon()
model = keras.models.Sequential([
    keras.layers.Lambda(lambda inputs: (inputs - means) / (stds + eps)),
    [...]      # остальные слои
])
```

Решение оказалось не слишком сложным! Тем не менее, вы можете отдать предпочтение использованию симпатичного самодостаточного специального слоя (очень похожего на `StandardScaler` из Scikit-Learn), а не наличию встречающихся повсюду глобальных переменных вроде `means` и `stds`:

```
class Standardization(keras.layers.Layer):
    def adapt(self, data_sample):
        self.means_ = np.mean(data_sample, axis=0, keepdims=True)
        self.stds_ = np.std(data_sample, axis=0, keepdims=True)
    def call(self, inputs):
        return (inputs - self.means_) /
               (self.stds_ + keras.backend.epsilon())
```

Прежде чем можно будет задействовать такой слой стандартизации, вам придется адаптировать его к своему набору данных, вызывая метод `adapt()` и передавая ему выборку данных. Это позволит слою применять подходящую среднюю величину и стандартное отклонение для каждого признака:

```
std_layer = Standardization()
std_layer.adapt(data_sample)
```

Выборка должна быть достаточно большой, чтобы представлять ваш набор данных, но она вовсе не обязана быть полным обучающим набором: в общем случае хватит нескольких сотен случайно выбранных образцов (однако, все зависит от задачи). Далее вы можете использовать слой предварительной обработки подобно нормальному слою:

```
model = keras.Sequential()
model.add(std_layer)
[...] # создание остальной части модели
model.compile([...])
model.fit([...])
```

Если вы думаете, что библиотека Keras должна содержать слой стандартизации такого рода, то для вас есть хорошая новость: ко времени чтения этих строк, вероятно, станет доступным слой `keras.layers.Normalization`. Он будет работать очень похоже на наш специальный слой `Standardization`: сначала необходимо создать слой, затем адаптировать его к набору данных, передавая методу `adapt()` выборку данных, и в заключение применять слой нормальным образом.

Давайте взглянем на категориальные признаки. Мы начнем с их кодирования как векторов в унитарном коде.

## Кодирование категориальных признаков с использованием векторов в унитарном коде

Рассмотрим признак `ocean_proximity` в наборе данных, содержащем цены на жилье в Калифорнии, который был исследован в главе 2. Он представляет собой категориальный признак с пятью возможными значениями: "`<1H OCEAN`", "`INLAND`", "`NEAR OCEAN`", "`NEAR BAY`" и "`ISLAND`". Нам необходимо закодировать такой признак до его передачи нейронной сети. Поскольку существует очень мало категорий, мы можем применять унитарное кодирование, для чего должны отобразить каждую категорию на ее индекс (от 0 до 4), что можно сделать с использованием таблицы поиска:

```
vocab = ["<1H OCEAN", "INLAND", "NEAR OCEAN", "NEAR BAY", "ISLAND"]
indices = tf.range(len(vocab), dtype=tf.int64)
table_init = tf.lookup.KeyValueTensorInitializer(vocab, indices)
num_oov_buckets = 2
table = tf.lookup.StaticVocabularyTable(table_init, num_oov_buckets)
```

Давайте пройдемся по коду.

- Сначала мы определяем *словарь*: это список всех возможных категорий.
- Затем мы создаем тензор с соответствующими индексами (от 0 до 4).
- Далее мы создаем инициализатор для таблицы поиска, передавая ему список категорий и соответствующие им индексы. В примере мы уже располагаем такими данными и потому применяем `KeyValueTensorInitializer`;

но если бы категории были перечислены в текстовом файле (по одной категории в строке), то взамен пришлось бы использовать `TextFileInitializer`.

- В последних двух строках мы создаем таблицу поиска, предоставляя ей инициализатор и указывая количество участков *вне словаря* (*out-of-vocabulary — oov*). Если мы ищем категорию, которая в словаре не существует, то таблица поиска вычислит хеш такой категории и применит его для назначения неизвестной категории одному из участков оов. Их индексы начинаются после индексов известных категорий, так что индексами двух участков оов в примере будут 5 и 6.

Зачем использовать участки оов? Дело в том, что если количество категорий велико (скажем, почтовые коды, города, слова, товары или пользователи) и база данных тоже крупная либо продолжает изменяться, тогда получение полного списка категорий может оказаться неудобным. Одно из решений предусматривает определение словаря на основе выборки данных (а не целого обучающего набора) и добавление ряда участков оов для остальных категорий, которые не присутствовали в выборке данных. Чем больше неизвестных категорий вы ожидаете обнаружить во время обучения, тем больше участков оов должны применяться. На самом деле, если участков оов недостаточно, то будут возникать конфликты: разные категории окажутся в том же самом участке, поэтому нейронная сеть не сумеет их различить (во всяком случае, не на основе этого признака).

А теперь воспользуемся таблицей поиска для кодирования небольшого пакета категориальных признаков в векторы в унитарном коде:

```
>>> categories=tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=514, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> cat_one_hot = tf.one_hot(cat_indices, depth=len(vocab)
+ num_oov_buckets)
>>> cat_one_hot
<tf.Tensor: id=524, shape=(4, 7), dtype=float32, numpy=
array([[0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.]], dtype=float32)>
```

Как видите, значение "NEAR BAY" отобразилось на индекс 3, неизвестная категория "DESERT" отобразилась на один из двух участков oov (с индексом 5), а значение "INLAND" отобразилось на индекс 1 дважды. Затем мы применяем функцию `tf.one_hot()` для унитарного кодирования этих индексов. Обратите внимание, что мы должны сообщить указанной функции общее количество индексов, которое равно размеру словаря плюс число участков oov. Итак, вы знаете, как кодировать категориальные признаки в векторы в унитарном коде, используя TensorFlow!

Как и ранее, было бы не особенно сложно связать всю приведенную логику в симпатичном самодостаточном классе. Его метод `adapt()` принимал бы выборку данных и извлекал все содержащиеся в ней отдельные категории. Он создавал бы таблицу поиска для сопоставления каждой категории с ее индексом (включая неизвестные категории с применением участков oov). Затем его метод `call()` использовал бы таблицу поиска для отображения входных категорий на их индексы. Но есть еще одна хорошая новость: со временем чтения вами книги в Keras, видимо, появится слой под названием `keras.layers.TextVectorization`, способный делать именно это — его метод `adapt()` будет извлекать словарь из выборки данных, а метод `call()` преобразовывать каждую категорию в ее индекс внутри словаря. Тогда вы сможете добавить такой слой в начало модели, а за ним слой `Lambda`, который будет применять функцию `tf.one_hot()`, если индексы желательно преобразовывать в векторы в унитарном коде.

Тем не менее, решение может оказаться не самым удачным. Размер каждого вектора в унитарном коде представляет собой длину словаря плюс количество участков oov. Когда существует лишь несколько возможных категорий, то все нормально, но если словарь большой, тогда намного эффективнее кодировать его с использованием вложений.



Запомните одно эмпирическое правило. Если количество категорий меньше 10, тогда обычно подходит унитарное кодирование. Если количество категорий превышает 50 (что часто происходит в случае применения хеш-участков), то вложения, как правило, предпочтительнее. Если количество категорий находится между 10 и 50, тогда имеет смысл позэкспериментировать с обоими вариантами и посмотреть, какой из них лучше работает в конкретном сценарии использования.

## Кодирование категориальных признаков с использованием вложений

Вложение является обучаемым плотным вектором, который представляет категорию. По умолчанию вложения инициализируются случайным образом, поэтому категория "NEAR BAY", например, могла бы первоначально представляться случайным вектором вроде [0.131, 0.890], тогда как категория "NEAR OCEAN" — еще одним случайным вектором, таким как [0.631, 0.791]. В приведенном примере мы применяем двумерные вложения, но количество измерений доступно в виде гиперпараметра, допускающего подстройку. Так как вложения поддаются обучению, они будут постепенно уточняться в процессе обучения; и поскольку они представляют довольно похожие категории, градиентный спуск непременно в итоге сблизит их, в то же время будет стремиться отдалить их от вложения категории "INLAND" (рис. 13.4). В действительности, чем лучше представление, тем легче нейронной сети вырабатывать точные прогнозы, поэтому обучение старается сделать вложения пригодными представлениями категорий. Процесс называется *обучением представлению* (*representation learning*); в главе 17 мы обсудим другие виды обучения представлению.

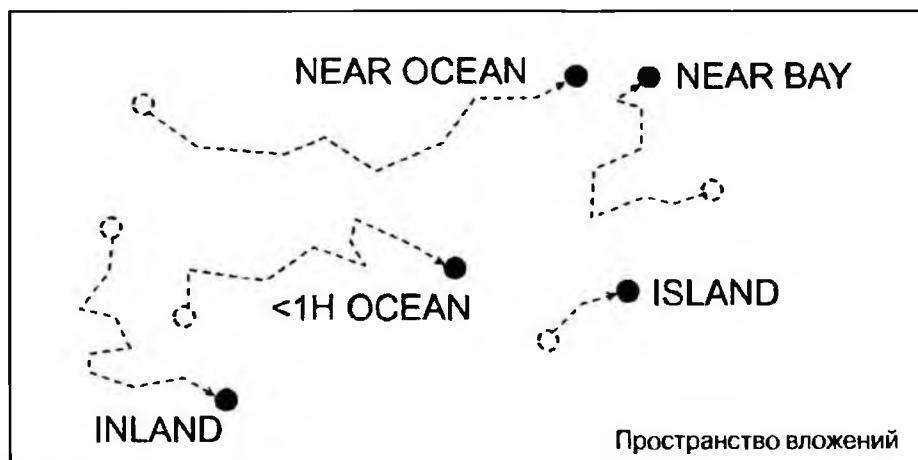


Рис. 13.4. Вложения будут постепенно уточняться в процессе обучения

## Вложения слов

Мало того, что вложения обычно будут удобными представлениями для решаемой задачи, но нередко те же самые вложения могут успешно использоваться для других задач. Самым распространенным примером являются *вложения слов* (*word embeddings*): при работе над задачей обработки естественного языка часто лучше снова применить заранее обученные вложения слов, чем обучать собственные.

Идея использования векторов для представления слов возникла в 1960-х годах и многие сложные устроенные методики использовались для создания успешных векторов, в том числе с применением нейронных сетей. Но дела действительно пошли в гору в 2013 году, когда Томаш Миколов и другие исследователи из Google опубликовали статью (<https://arxiv.org/pdf/1310.4546.pdf>)<sup>9</sup>, описывающую эффективную методику выяснения вложений слов с использованием нейронных сетей, которая значительно превосходила предшествующие попытки. Она позволила узнавать вложения на очень крупном корпусе текстов: они обучали нейронную сеть для прогнозирования слов, близких к любому заданному слову, и получили поразительные вложения слов. Например, синонимы имели очень близкие вложения, а семантически родственные слова, такие как France (Франция), Spain (Испания) и Italy (Италия), в итоге сгруппировались вместе.

Однако речь идет не только о близости: вложения слов также были упорядочены вдоль значимых осей в пространстве вложений. Вот знаменитый пример: если вы вычислите King (король) – Man (мужчина) + Woman (женщина), выполнив сложение и вычитание векторов вложений этих слов, то результат будет очень близким к вложению слова Queen (королева), как видно на рис. 13.5. Получается, что вложения слов кодируют понятие пола! Аналогично вы можете вычислить Madrid (Мадрид) – – Spain (Испания) + France (Франция) и результат будет близким к Paris (Париж), откуда можно сделать вывод, что понятие столицы, похоже, тоже было закодировано во вложениях.

<sup>9</sup> Томаш Миколов и др., *Distributed Representations of Words and Phrases and Their Compositional Properties* (Распределенные представления слов и фраз и их композиционность), *Proceedings of the 26th International Conference on Neural Information Processing Systems 2* (2013 г.): с. 3111–3119.

К сожалению, вложения слов иногда захватывают наши наихудшие предубеждения. Например, хотя вложения слов корректно узнают, что Man (мужчина) относится к King (король), а Woman (женщина) — к Queen (королева), похоже, они также выучат то, что Man (мужчина) относится к Doctor (врач), а Woman (женщина) — к Nurse (медсестра): довольно сексистское предубеждение! Справедливости ради надо сказать, что этот отдельно взятый пример, вероятно, преувеличен, как отмечалось в статье 2019 года (<https://homl.info/fairembeds>)<sup>10</sup> Мальвиной Ниссим и др. Тем не менее, обеспечение равноправия в алгоритмах глубокого обучения является важной и активной темой исследований.

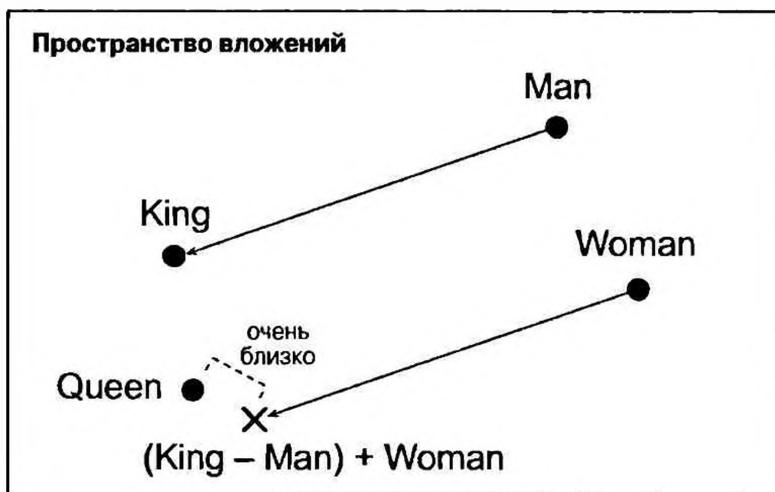


Рис. 13.5. Вложения похожих слов имеют тенденцию сближаться и некоторые оси, похоже, кодируют значимые понятия

Давайте посмотрим, как можно было бы реализовать вложения вручную, чтобы понять, каким образом они работают (далее мы взамен будем применять простой слой Keras). Первым делом нам необходимо создать *матрицу вложений* (*embedding matrix*), которая содержит вложение каждой категории, инициализированное случайным образом; она будет иметь по одной строке на категорию и на участок оов, а также по одному столбцу на измерение вложения:

<sup>10</sup> Мальвина Ниссим и др., *Fair Is Better Than Sensational: Man Is to Doctor as Woman Is to Doctor* (Равноправие лучше, чем чувственность: мужчина относится к врачу в равной степени, как женщина относится к врачу), препринт arXiv:1905.09866 (2019 г.).

```
embedding_dim = 2
embed_init = tf.random.uniform([len(vocab) + num_oov_buckets,
                                embedding_dim])
embedding_matrix = tf.Variable(embed_init)
```

В рассматриваемом примере мы используем двумерные вложения, но согласно эмпирическому правилу вложения обычно имеют от 10 до 300 измерений в зависимости от задачи и размера словаря (этот гиперпараметр придется регулировать).

Матрица вложений представляет собой случайную матрицу  $6 \times 2$ , хранящуюся в переменной (так что ее можно подстраивать с помощью градиентного спуска во время обучения):

```
>>> embedding_matrix
<tf.Variable 'Variable:0' shape=(6, 2) dtype=float32, numpy=
array([[0.6645621 , 0.44100678],
       [0.3528825 , 0.46448255],
       [0.03366041, 0.68467236],
       [0.74011743, 0.8724445 ],
       [0.22632635, 0.22319686],
       [0.3103881 , 0.7223358 ]], dtype=float32)>
```

Теперь закодируем тот же пакет категориальных признаков, что и ранее, но с применением наших вложений:

```
>>> categories = tf.constant(["NEAR BAY", "DESERT", "INLAND", "INLAND"])
>>> cat_indices = table.lookup(categories)
>>> cat_indices
<tf.Tensor: id=741, shape=(4,), dtype=int64, numpy=array([3, 5, 1, 1])>
>>> tf.nn.embedding_lookup(embedding_matrix, cat_indices)
<tf.Tensor: id=864, shape=(4, 2), dtype=float32, numpy=
array([[0.74011743, 0.8724445 ],
       [0.3103881 , 0.7223358 ],
       [0.3528825 , 0.46448255],
       [0.3528825 , 0.46448255]], dtype=float32)>
```

Функция `tf.nn.embedding_lookup()` ищет строки в матрице вложений по заданным индексам — это все, что она делает. Например, таблица поиска сообщает о том, что категория "INLAND" находится по индексу 1, и потому функция `tf.nn.embedding_lookup()` возвращает вложение из строки 1 в матрице вложений (дважды): [0.3528825, 0.46448255].

Библиотека Keras предлагает слой `keras.layers.Embedding`, который обрабатывает матрицу вложений (по умолчанию обучаемый); когда такой

слой создается, он инициализирует матрицу вложений случайным образом и затем при вызове с какими-либо индексами категорий возвращает из матрицы вложений строки по указанным индексам:

```
>>> embedding = keras.layers.Embedding(input_dim=len(vocab)
                                         + num_oov_buckets,
                                         output_dim=embedding_dim)

...
...
>>> embedding(cat_indices)
<tf.Tensor: id=814, shape=(4, 2), dtype=float32, numpy=
array([[ 0.02401174,  0.03724445],
       [-0.01896119,  0.02223358],
       [-0.01471175, -0.00355174],
       [-0.01471175, -0.00355174]], dtype=float32)>
```

Собрав все вместе, мы теперь можем создать модель Keras, которая способна обрабатывать категориальные признаки (наряду с обычными числовыми признаками) и узнавать вложение для каждой категории (а также для каждого участка oov):

```
regular_inputs = keras.layers.Input(shape=[8])
categories = keras.layers.Input(shape=[], dtype=tf.string)
cat_indices = keras.layers.Lambda(lambda cats: table.lookup(cats))
(categories)
cat_embed = keras.layers.Embedding(input_dim=6, output_dim=2)
(cat_indices)
encoded_inputs = keras.layers.concatenate([regular_inputs, cat_embed])
outputs = keras.layers.Dense(1)(encoded_inputs)
model = keras.models.Model(inputs=[regular_inputs, categories],
                           outputs=[outputs])
```

Модель принимает два входа: обычный вход, содержащий восемь числовых признаков на образец, и категориальный вход (содержащий один категориальный признак на образец). Она использует слой Lambda для нахождения индекса каждой категории, после чего ищет вложения для этих индексов. Далее модель объединяет вложения и обычные входы, чтобы получить закодированные входы, которые готовы к подаче в нейронную сеть. Здесь мы могли бы добавить нейронную сеть любого вида, но добавляем лишь плотный выходной слой и создаем модель Keras.

Когда доступен слой keras.layers.TextVectorization, вы можете вызвать его метод adapt(), чтобы заставить слой извлечь словарь из выборки данных (он самостоятельно позаботится о создании таблицы поиска).

Затем можете добавить его в модель, и он будет выполнять поиск индексов (заменяя слой Lambda в предыдущем примере кода).



Унитарное кодирование, за которым следует слой Dense (без функции активации и без смещений), эквивалентно слою Embedding. Однако в слое Embedding применяется гораздо меньше вычислений (с ростом матрицы вложений различия в производительности становятся очевидной). Матрица весов слоя Dense исполняет роль матрицы вложений. Скажем, использование векторов в унитарном коде размером 20 и слоя Dense с 10 элементами эквивалентно применению слоя Embedding с `input_dim=20` и `output_dim=10`. В результате было бы расточительством использовать во вложениях большие измерений, чем количество элементов в слое, который находится после слоя Embedding.

Давайте теперь чуть более пристально взглянем на слои предварительной обработки Keras.

## Слои предварительной обработки Keras

Команда разработчиков TensorFlow трудится над предоставлением набора стандартных слоев предварительной обработки Keras (<https://homl.info/preproc>). Возможно, они станут доступными на момент чтения этих строк; тем не менее, к тому времени API-интерфейс может слегка измениться, так что обращайтесь к тетради Jupiter для настоящей главы, если что-то ведет себя непредсказуемо. Вероятнее всего, новый API-интерфейс заменит существующий API-интерфейс Feature Columns (строки признаков), который сложнее в применении и менее нагляден (если вы все равно хотите узнать больше об API-интерфейсе Feature Columns, тогда обратитесь к тетради Jupiter для этой главы).

Мы уже обсуждали два таких слоя: слой `keras.layers.Normalization`, который будет выполнять стандартизацию признаков (он эквивалентен определенному ранее слою `Standardization`) и слой `TextVectorization`, который будет способен кодировать каждое слово во входах в его индекс внутри словаря. В обоих случаях вы создаете слой, вызываете его метод `adapt()` с выборкой данных и затем нормально используете слой в своей модели. Остальные слои предварительной обработки следуют такому же шаблону.

Новый API-интерфейс также будет включать слой `keras.layers.Discretization`, который разбивает непрерывные данные на разные контейнеры и кодирует каждый контейнер как вектор в унитарном коде. Например, вы могли бы применять его для разделения цен на три категории (низкая, средняя, высокая), которые кодировались бы как `[1, 0, 0]`, `[0, 1, 0]` и `[0, 0, 1]` соответственно. Конечно, при таком подходе утрачивается много информации, но в ряде случаев это помогает модели обнаруживать паттерны, которые при просмотре непрерывных значений не были бы очевидными.



Слой `Discretization` не будет дифференцируемым и должен использоваться только в начале модели. На самом деле слои предварительной обработки модели во время обучения будут замораживаться, так что градиентный спуск не повлияет на их параметры, а потому они не обязаны быть дифференцируемыми. Это также означает, что вы не должны применять слой `Embedding` прямо в специальном слое предварительной обработки, если хотите, чтобы он был обучаемым: взамен его потребуется добавить отдельно к модели, как в предыдущем примере кода.

Также будет возможно связывать в цепочку множество слоев предварительной обработки, используя класс `PreprocessingStage`. Скажем, в приведенном ниже коде создается конвейер предварительной обработки, который сначала нормализует входы и затем дискретизирует их (что может напоминать конвейеры Scikit-Learn). После адаптации такого конвейера к выборке данных вы можете применять его подобно обычному слою в своих моделях (но опять только в начале модели, поскольку он содержит недифференцируемый слой предварительной обработки):

```
normalization = keras.layers.Normalization()
discretization = keras.layers.Discretization([...])
pipeline = keras.layers.PreprocessingStage([normalization,
                                             discretization])
pipeline.adapt(data_sample)
```

Слой `TextVectorization` также будет иметь возможность выводить векторы со счетчиками слов вместо индексов слов. Например, если словарь содержит три слова, `["and", "basketball", "more"]`, тогда текст `"more and more"` отобразится на вектор `[1, 0, 2]`: слово `"and"` встречается один раз, слово `"basketball"` вообще отсутствует, а слово `"more"` появляет-

ся дважды. Текстовое представление подобного рода называется *мешком слов* (*bag of words*), т.к. в нем полностью утрачивается порядок следования слов. Распространенные слова вроде "and" будут иметь крупную величину в большинстве текстов, хотя обычно они наименее интересны (скажем, в тексте "more and more basketball" слово "basketball" определенно самое важное как раз оттого, что не является очень частым словом). Таким образом, счетчики слов должны нормализоваться способом, который снижает важность часто встречающихся слов. Обычный способ предусматривает деление каждого счетчика слова на логарифм общего количества обучающих образцов, в которых слово появлялось. Такая методика называется *TF-IDF* (*Term-Frequency × Inverse-Document-Frequency* — частота терма × обратная частота документа). Например, давайте представим, что слова "and", "basketball" и "more" встречаются соответственно в 200, 10 и 100 текстовых образцах в обучающем наборе: в таком случае финальным вектором будет  $[1/\log(200), 0/\log(10), 2/\log(100)]$ , который приблизенно равен [0.19, 0., 0.43]. Слой *TextVectorization* будет (вероятно) иметь возможность выполнять *TF-IDF*.



Если стандартных слоев предварительной обработки для вашей задачи недостаточно, то у вас по-прежнему есть возможность создать собственный специальный слой предварительной обработки во многом подобно тому, как делалось ранее с классом *Standardization*. Создайте подкласс класса *keras.layers.PreprocessingLayer* с методом *adapt()*, который должен принимать аргумент *data\_sample* и дополнительно аргумент *reset\_state*: если он имеет значение *True*, тогда метод *adapt()* перед расчетом нового состояния обязан сбрасывать любое существующее состояние; если он равен *False*, то метод должен попытаться обновить существующее состояние.

Как видите, указанные слои предварительной обработки Keras намного облегчат предварительную обработку! Независимо от того, решите вы написать собственные слои предварительной обработки либо использовать их реализации из Keras (или даже применять API-интерфейс *Feature Columns*), вся предварительная обработка будет выполняться на лету. Однако во время обучения может быть предпочтительнее проводить предварительную обработку заранее. Давайте посмотрим, почему может возникнуть желание поступать так, и каким образом это делать.

Если предварительная обработка является затратной в вычислительном плане, тогда ее выполнение перед обучением, а не на лету, может обеспечить значительное ускорение: данные будут предварительно обрабатываться только один раз на образец до обучения вместо одного раза на образец и на эпоху во время обучения. Как упоминалось ранее, когда набор данных достаточно мал, чтобы уместиться в ОЗУ, можно использовать его метод `cache()`. Но если набор данных слишком большой, то помогут инструменты, подобные Apache Beam или Spark. Они позволяют запускать эффективные конвейеры предварительной обработки на крупных объемах данных, даже распределяя вычисления по множеству серверов, поэтому вы можете применять их для предварительной обработки всех обучающих данных перед обучением.

Прием прекрасно работает и действительно способен ускорить обучение, но есть одна проблема: предположим, что после обучения модели вы хотите развернуть ее в мобильном приложении. В таком случае вам понадобится написать в приложении код, который позаботится о предварительной обработке данных до их передачи модели. И пусть вы также хотите развернуть модель в `TensorFlow.js`, чтобы она запускалась в веб-браузере. Вам снова необходимо написать какой-то код предварительной обработки. В итоге это может стать кошмаром при сопровождении: всякий раз, когда вы решаете изменить логику предварительной обработки, вам будет требоваться обновлять код Apache Beam, код мобильного приложения и код на JavaScript. Работа подобного рода не только отнимает много времени, она также подвержена ошибкам: вы можете столкнуться с тонкими различиями между операциями предварительной обработки, выполняющимися до обучения, и такими операциями, которые выполняются в вашем приложении или в веб-браузере. Такая асимметрия между обучением и обслуживанием (*training/serving skew*) будет приводить к дефектам или пониженной эффективности.

Одно улучшение могло бы заключаться в том, чтобы взять обученную модель (обученную на данных, которые были предварительно обработаны вашим кодом Apache Beam или Spark) и до ее развертывания в приложении или в веб-браузере добавить дополнительные слои предварительной обработки, чтобы позаботиться о ней на лету. Поступать так определенно лучше, поскольку теперь у вас есть лишь две версии кода предварительной обработки: код Apache Beam или Spark и код слоя предварительной обработки.

Но что, если бы вы могли определить операции предварительной обработки только раз? Именно для этого проектировалась библиотека TF Transform. Она является частью TensorFlow Extended (TFX; <https://tensorflow.org/tfx>) — сквозной платформы для запуска в производственную среду моделей TensorFlow. Для использования компонента TFX, такого как TF Transform, его сначала потребуется установить; он не поставляется в комплекте с TensorFlow. Затем вы определяете функция предварительной обработки только один раз (на Python) с применением функций TF Transform для масштабирования, разбиения на участки и т.д. Вы также можете использовать любые желаемые операции TensorFlow. Вот как выглядит такая функция предварительной обработки при наличии лишь двух признаков:

```
import tensorflow_transform as tft

def preprocess(inputs):          # inputs = пакет входных признаков
    median_age = inputs["housing_median_age"]
    ocean_proximity = inputs["ocean_proximity"]
    standardized_age = tft.scale_to_z_score(median_age)
    ocean_proximity_id =
        tft.compute_and_apply_vocabulary(ocean_proximity)
    return {
        "standardized_median_age": standardized_age,
        "ocean_proximity_id": ocean_proximity_id
    }
```

Далее TF Transform позволяет вам применить функцию `preprocess()` к целому обучающему набору, используя инструмент Apache Beam (он предоставляет класс `AnalyzeAndTransformDataset`, который можно задействовать для этой цели в конвейере Apache Beam). В процессе указанный класс также будет рассчитывать все необходимые статистические данные на целом обучающем наборе: в рассматриваемом примере среднюю величину и стандартное отклонение признака `housing_median_age` и словарь для признака `ocean_proximity`. Компоненты, рассчитывающие такие статистические данные, называются *анализаторами (analyzer)*.

Важно отметить, что TF Transform будет генерировать эквивалентную функцию TensorFlow Function, которую вы можете вставить в развертываемую модель. Эта функция TF Function включает ряд констант, которые соответствуют всем нужным статистическим данным, вычисленным Apache Beam (среднее, стандартное отклонение и словарь).

С помощью API-интерфейса Data, записей TFRecord, слоев предварительной обработки Keras и библиотеки TF Transform вы можете строить высокомасштабируемые входные конвейеры для обучения и извлекать преимущества из быстрой и переносимой предварительной обработки данных в производственной среде.

Но что, если вы просто хотите использовать стандартный набор данных? В таком случае все намного легче: применяйте TFDS!

## Проект TensorFlow Datasets (TFDS)

Проект TensorFlow Datasets (<https://tensorflow.org/datasets>) всерьез облегчает загрузку распространенных наборов данных, начиная с небольших вроде MNIST или Fashion MNIST и заканчивая гигантскими наподобие ImageNet (вам понадобится немало свободного пространства на диске!). В список входят наборы данных с изображениями, с текстом (включая наборы данных для перевода), а также с аудио- и видеоклипами. Пройдите по ссылке <https://homl.info/tfds>, чтобы просмотреть полный список вместе с описаниями каждого набора данных.

TFDS не поставляется в составе TensorFlow, поэтому библиотеку tensorflow-datasets вам придется установить (скажем, используя pip). Затем вызовите функцию `tfds.load()`, которая загрузит желаемые данные (если только они не были загружены ранее) и возвратит их в виде словаря наборов данных (обычно одного для обучения и одного для испытаний, но это зависит от выбранного набора данных). Например, давайте загрузим MNIST:

```
import tensorflow as tfds
dataset = tfds.load(name="mnist")
mnist_train, mnist_test = dataset["train"], dataset["test"]
```

Далее вы можете применить любую трансформацию, какую пожелаете (обычно тасование, группирование в пакеты и предварительную выборку), и все будет готово к обучению модели. Вот простой пример:

```
mnist_train = mnist_train.shuffle(10000).batch(32).prefetch(1)
for item in mnist_train:
    images = item["image"]
    labels = item["label"]
    [...]
```



Функция `load()` тасует каждый фрагмент данных, который она загружает (только для обучающего набора). Этого может оказаться недостаточно, а потому лучше перетасовать обучающие данные дополнительно.

Обратите внимание, что каждый элемент в наборе данных является словарем, содержащим признаки и метки. Но Keras ожидает, что каждый элемент будет кортежем с двумя элементами (опять признаками и метками). Вы могли бы трансформировать набор данных, используя метод `map()`:

```
mnist_train = mnist_train.shuffle(10000).batch(32)
mnist_train = mnist_train.map(lambda items: (items["image"],
                                              items["label"]))
mnist_train = mnist_train.prefetch(1)
```

Но проще поручить выполнение трансформации функции `load()`, устанавливая `as_supervised=True` (очевидно, прием работает только для помеченных наборов данных). При желании можно также указать размер пакета. Затем набор данных можно передать напрямую модели `tf.keras`:

```
dataset = tfds.load(name="mnist", batch_size=32, as_supervised=True)
mnist_train = dataset["train"].prefetch(1)
model = keras.models.Sequential([...])
model.compile(loss="sparse_categorical_crossentropy", optimizer="sgd")
model.fit(mnist_train, epochs=5)
```

В настоящей главе приводились сведения в основном технического характера, и у вас могло возникнуть ощущение, что она несколько отдалась от абстрактной красоты нейронных сетей. Тем не менее, факт заключается в том, что глубокое обучение часто имеет дело с крупными объемами данных, поэтому знание того, как их эффективно загружать, разбирать и предварительно обрабатывать, является критически важным навыком. В следующей главе мы рассмотрим сверточные нейронные сети, которые находятся в числе наиболее успешных архитектур нейронных сетей, предназначенных для обработки изображений и многих других приложений.

# Упражнения

1. По какой причине вы захотели бы применять API-интерфейс Data?
2. Каковы преимущества разделения крупного набора данных на множество файлов?
3. Как во время обучения выяснить, что входной конвейер является узким местом? Что можно предпринять для устранения проблемы?
4. Можно ли сохранять в файле TFRecord любые двоичные данные или же только сериализированные протокольные буферы?
5. Зачем морочиться с преобразованием всех своих данных в формат протобуфера Example? Почему бы не воспользоваться собственным определением протобуфера?
6. Когда вы пожелали бы активизировать сжатие при использовании записей TFRecord? Почему бы не делать это систематически?
7. Данные можно предварительно обрабатывать прямо при записи в файлы данных, в конвейере `tf.data`, в слоях предварительной обработки внутри модели или с применением TF Transform. Можете ли вы перечислить для каждого варианта несколько доводов за и против?
8. Назовите несколько распространенных методик, которые можно использовать для кодирования категориальных признаков. Что насчет текста?
9. Загрузите набор данных Fashion MNIST (представленный в главе 10), расщепите его на обучающий набор, проверочный набор и испытательный набор, перетасуйте обучающий набор и сохраните каждый набор во множество файлов TFRecord. Каждая запись должна быть сериализованным протобуфером Example с двумя признаками: сериализованное изображение (для сериализации каждого изображения применяйте функцию `tf.io.serialize_tensor()`) и метка<sup>11</sup>. Затем воспользуйтесь `tf.data`, чтобы создать эффективный набор данных для каждого набора. Наконец, обучите на этих наборах данных модель Keras, включающую слой предварительной обработки для стандарти-

<sup>11</sup> Для крупных изображений взамен можно было бы использовать функцию `tf.io.encode_jpeg()`. В итоге экономилось бы пространство, но за счет небольшой потери качества изображений.

зации каждого входного признака. Попытайтесь сделать входной конвейер как можно более эффективным, применяя TensorBoard для визуализации данных профилирования.

10. В настоящем упражнении вы загрузите набор данных, расщепите его, создадите объект `tf.data.Dataset` для загрузки и эффективной предварительной обработки, после чего постройте и обучите модель двоичной классификации, содержащую слой `Embedding`.
- Загрузите набор данных Large Movie Review Dataset (<https://homl.info/imdb>), который содержит 50 000 рецензий на фильмы из базы данных фильмов Internet Movie Database (<https://imdb.com/>). Данные организованы в двух каталогах, `train` и `test`, каждый из которых содержит подкаталог `pos` с 12 500 положительных рецензий и подкаталог `neg` с 12 500 отрицательных рецензий. Каждая рецензия хранится в отдельном текстовом файле. Есть также другие файлы и подкаталоги (включая заранее обработанные мешки слов), но в этом упражнении их следует игнорировать.
  - Расщепите испытательный набор на проверочный (15 000) и испытательный (10 000) наборы.
  - Воспользуйтесь `tf.data`, чтобы создать эффективный набор данных для каждого набора.
  - Создайте модель двоичной классификации, применяющую слой `TextVectorization` для предварительной обработки каждой рецензии. Если слой `TextVectorization` пока еще не доступен (или вам нравятся сложные задачи), тогда попытайтесь создать собственный специальный слой предварительной обработки: вы можете использовать функции из пакета `tf.strings`, например, `lower()` для преобразования всех символов в нижний регистр, `regex_replace()` для замены знаков пунктуации пробелами и `split()` для разбиения на слова по пробелам. Вам потребуется применять таблицу поиска для вывода индексов слов, которые должны быть подготовлены в методе `adapt()`.
  - Добавьте слой `Embedding` и рассчитайте среднее вложение для каждой рецензии, умноженное на квадратный корень из количества слов (см. главу 16). Затем это масштабированное среднее вложение можно передать оставшейся части модели.

- е) Обучите модель и посмотрите, какую правильность вы получили. Попробуйте оптимизировать свои конвейеры, чтобы сделать обучение насколько возможно быстрым.
- ж) Воспользуйтесь TFDS для загрузки того же набора данных более легким способом: `tfds.load("imdb_reviews")`.

Решения приведенных упражнений доступны в приложении А.

# Глубокое компьютерное зрение с использованием сверточных нейронных сетей

Хотя суперкомпьютер IBM Deep Blue победил чемпиона мира по шахматам Гарри Каспарова еще в 1996 году, лишь недавно компьютеры стали способными надежно выполнять на вид простые задачи, такие как выявление 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 чувственных модулей в наших мозгах. К тому времени, когда чувственная 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 чувственные 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 эффективности при решении ряда сложных зрительных задач. Они приводят в действие мощные службы поиска изображений, беспилотные автомобили, системы автоматической классификации видеоро-

ликов и т.п. Кроме того, сети CNN не ограничиваются зрительным восприятием: они также успешно решают многие другие задачи, в числе которых распознавание речи и обработка естественного языка. Однако пока что мы сосредоточимся на зрительных приложениях.

В настоящей главе мы выясним, откуда произошли сети CNN, как выглядят их строительные блоки и каким образом реализовать их с использованием TensorFlow и Keras. Затем мы обсудим несколько наилучших архитектур сетей CNN, а также другие зрительные задачи, такие как выявление объектов (классификация множества объектов в изображении и размещение ограничивающих рамок вокруг них) и *семантическая сегментация* (*semantic segmentation*; классификация каждого Пикселя в соответствии с классом объекта, которому он принадлежит).

## Строение зрительной коры головного мозга

Дэвид Х. Хьюбел и Торстен Визель провели серию экспериментов на кошках в 1958 (<https://homl.info/71>)<sup>1</sup> и 1959 (<https://homl.info/72>)<sup>2</sup> годах (а спустя несколько лет также на обезьянах (<https://homl.info/73>)<sup>3</sup>), 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 маленькое локальное рецепторное поле (*local receptive field*), а потому реагируют только на зрительные раздражители, находящиеся в ограниченной области поля зрения (на рис. 14.1 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 —

<sup>1</sup> Дэвид Х. Хьюбел, *Single Unit Activity in Striate Cortex of Unrestrained Cats* (Единичная активность в полосатой коре естественных кошек), *The Journal of Physiology* 147 (1959 г.): с. 226–238.

<sup>2</sup> Дэвид Х. Хьюбел и Торстен Визель, *Receptive Fields of Single Neurons in the Cat's Striate Cortex* (Рецепторные поля одиночных нейронов в полосатой коре кошки), *The Journal of Physiology* 148 (1959 г.): с. 574–591.

<sup>3</sup> Дэвид Х. Хьюбел и Торстен Визель, *Receptive Fields and Functional Architecture of Monkey Striate Cortex* (Рецепторные поля и функциональная архитектура полосатой коры обезьян), *The Journal of Physiology* 195 (1968 г.): с. 215–243.

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. 14.1, 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.



**Рис. 14.1.** Биологические нейроны в зрительной коре реагируют на специфические образы в небольших областях поля зрения, называемых рецепторными полями; по мере прохождения зрительного сигнала через последовательные модули мозга нейроны реагируют на более сложные образы в более крупных рецепторных полях

Проведенные исследования зрительной коры вдохновили на создание в 1980 году неокогнитрона (neocognitron; <https://homl.info/74>)<sup>4</sup>, который постепенно развился в то, что сейчас мы называем *сверточными нейронными сетями*. Важной вехой стала работа 1998 года (<https://homl.info/75>)<sup>5</sup> Яна Лекуна и др., в которой была введена знаменитая архитектура *LeNet-5*, широко применяемая банками для распознавания рукописных чисел на чеках. Эта архитектура содержит ряд строительных блоков, которые вам уже

<sup>4</sup> Кунихико Фукушима, *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), *Biological Cybernetics* 36 (1980 г.): с. 193–202.

<sup>5</sup> Ян Лекун и др., *Gradient-Based Learning Applied to Document Recognition* (Основанное на градиентах обучение, примененное к распознаванию документов), *Proceedings of the IEEE* 86, номер 11 (1998 г.): с. 2278–2324.

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.

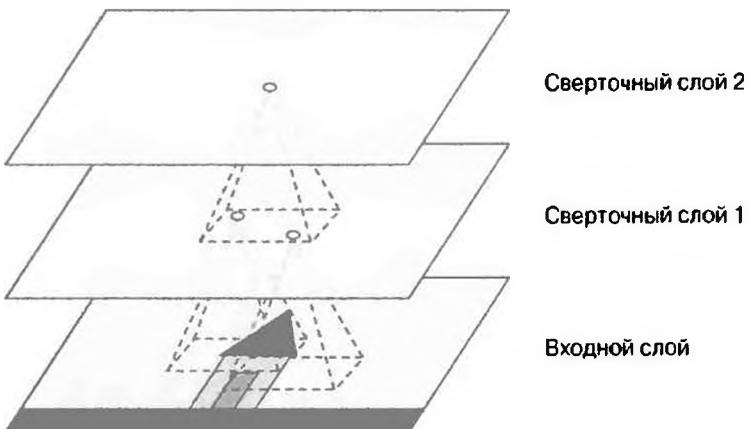


Почему для решения задач распознавания изображений просто не воспользоваться глубокой нейронной сетью с полносвязными слоями? К сожалению, хотя она хорошо работает для небольших изображений (скажем, MNIST), в случае более крупных изображений ее работа нарушается из-за гигантского количества требующихся параметров. Например, изображение  $100 \times 100$  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 связанных слоев и разделения весов.

## Сверточные слои

Самый важный строительный блок сети CNN — это *сверточный слой*<sup>6</sup>: нейроны в первом сверточном слое связаны не с каждым одиночным пикселием во входном изображении (как было в слоях, обсуждавшихся в предыдущих главах), но только с пикселями в собственных рецепторных полях (рис. 14.2). 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 маленьких низкоуровневых признаках в первом скрытом слое, затем скомпоновать их в более крупные высокоуровневые признаки в следующем скрытом слое и т.д. Иерархическая структура подобного рода распространена в реальных изображениях, что и является одной из причин, по которым сети CNN настолько хорошо работают при распознавании изображений.

<sup>6</sup> 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. На самом деле сверточные слои применяют взаимную корреляцию, которая очень похожа на свертку (за дополнительными сведениями обращайтесь по ссылке <https://homl.info/76>).



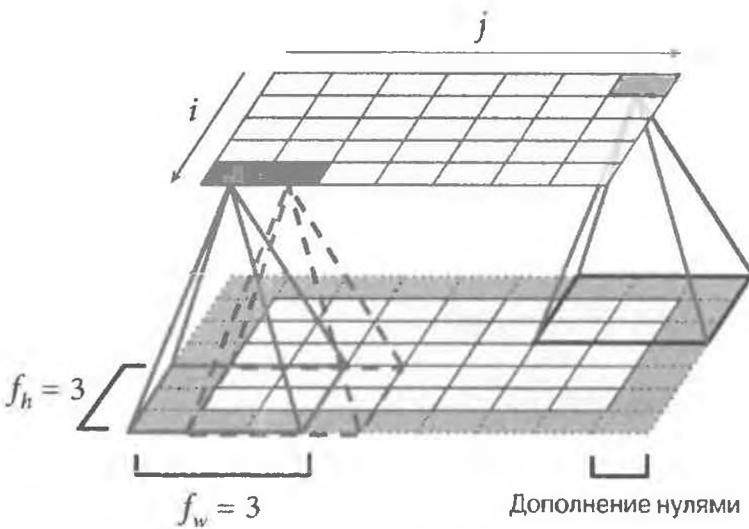
**Рис. 14.2.** Слои сети CNN с прямоугольными локальными рецепторными полями



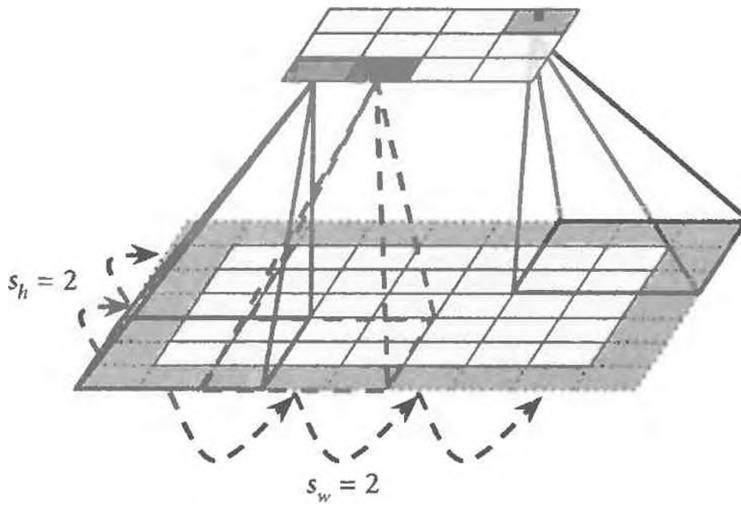
Все рассмотренные до сих пор многослойные нейронные сети имели слои, состоящие из длинной линейки нейронов, и нам приходилось разглаживать входные изображения до одного измерения, прежде чем передавать их нейронной сети. В сети CNN каждый слой представлен в двух измерениях, что облегчает сопоставление 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. 14.3). Для того чтобы слой имел такую же высоту и ширину, как у предыдущего слоя, вокруг входов обычно добавляют нули, что видно на рис. 14.3. Это называется *дополнением нулями (zero padding)*.

Также можно связать крупный входной слой с гораздо меньшим слоем, растягивая рецепторные поля (рис. 14.4). Это значительно сокращает вычислительную сложность модели. Смещение от одного рецепторного поля до следующего называется *страйдом (stride)*, или большим шагом. На рис. 14.4 входной слой  $5 \times 7$  (плюс дополнение нулями) связывается со слоем  $3 \times 4$ , используя рецепторные поля  $3 \times 3$  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. 14.3. Связи между слоями и дополнение нулями*



*Рис. 14.4. Понижение размерности с применением страйда 2*

## Фильтры

Веса нейронов могут быть представлены как небольшие изображения с размером рецепторного поля. Например, на рис. 14.5 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  $7 \times 7$ , 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.

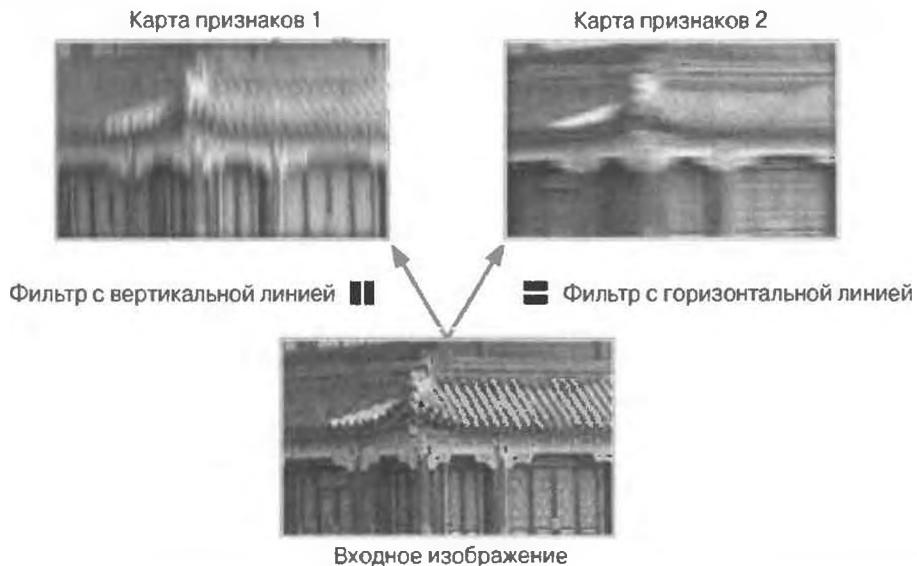


Рис. 14.5. Применение двух разных фильтров для получения карт признаков

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. 14.5 (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 тот же самый 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*), выделяющую области изображения, которые больше всего активируют фильтр. Разумеется, вы не обязаны определять фильтры вручную: во время обучения сверточный слой автоматически выясняет самые пригодные фильтры для своей задачи и слои выше научатся их комбинировать в более сложные образы.

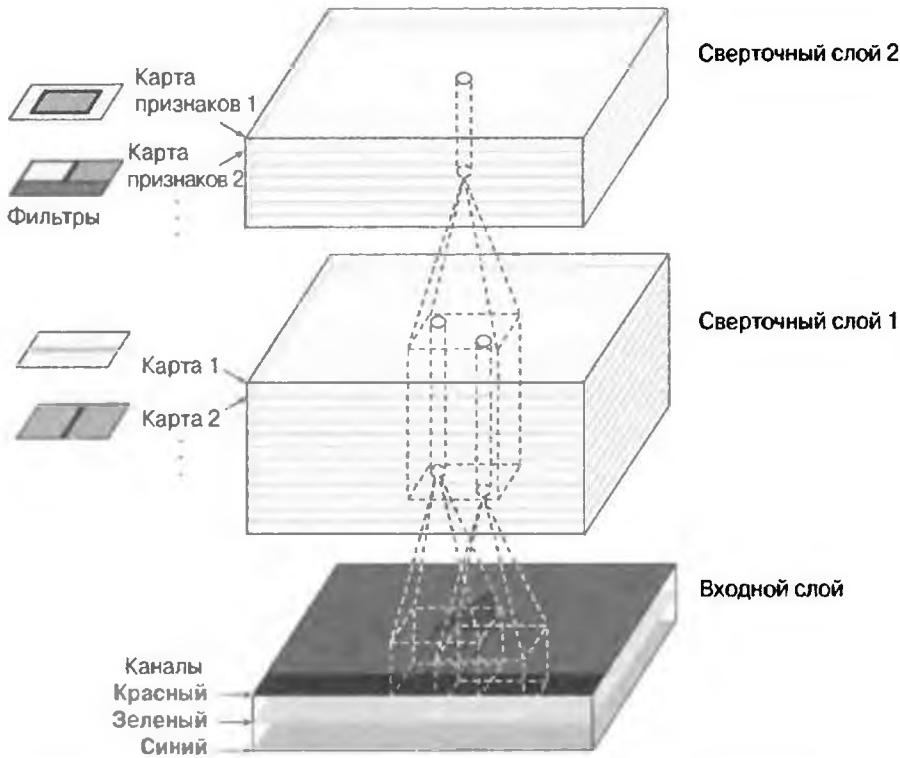
## Наложение множества карт признаков

До сих пор ради простоты выход каждого сверточного слоя представлялся как двумерный слой, но в реальности сверточный слой имеет множество фильтров (вы сами определяете их количество) и выдает по одной карте признаков на фильтр, поэтому более точно представляется в трех измерениях (рис. 14.6). Он имеет один нейрон на пиксель в каждой карте признаков, а все нейроны внутри заданной карты признаков разделяют те же самые параметры (т.е. те же веса и член смещения). Нейроны в разных картах признаков используют 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 обучаемых фильтров к своим входам, становясь способным обнаруживать множество признаков повсюду в своих входах.



Тот факт, что все нейроны в карте признаков разделяют те же самые параметры, значительно сокращает количество параметров в модели. 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.

Входные изображения также состоят из множества подслоев: по одному на цветовой канал. Их обычно три: красный, зеленый и синий (red, green, blue — RGB). Полутоновые изображения имеют только один канал, но некоторые изображения могут иметь гораздо больше каналов — например, изображения со спутников, фиксирующие свет дополнительных частот (такой как инфракрасный).



**Рис. 14.6. Сверточные слои с множеством карт признаков и изображения с тремя цветовыми каналами**

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 14.1 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 14.1. Вычисление выхода нейрона в сверточном слое

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \left| \begin{array}{l} i' = i \times s_h + u \\ j' = j \times s_w + v \end{array} \right.$$

В этом уравнении:

- $z_{i,j,k}$  — выход нейрона, расположенного в строке  $i$  и столбце  $j$  в карте признаков  $k$  сверточного слоя (слой  $l$ );
- как объяснялось ранее,  $s_h$  и  $s_w$  — вертикальный и горизонтальный stride,  $f_h$  и  $f_w$  — высота и ширина рецепторного поля, а  $f_{n'}$  — количество карт признаков в предыдущем слое (слой  $l-1$ );
- $x_{i',j',k'}$  — 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 формы [высота, ширина, каналы]. Мини-пакет представляется как четырехмерный тензор формы [размер мини-пакета, высота, ширина, каналы]. Веса сверточного слоя представляются как четырехмерный тензор формы [ $f_h, f_w, f_{n'}, f_n$ ]. Члены смещения сверточного слоя представляются просто как одномерный тензор формы [ $f_n$ ].

Давайте рассмотрим простой пример. Приведенный ниже код загружает два образца изображений, используя функцию `load_sample_images()` из Scikit-Learn (которая загружает два цветных изображения, одно с китайским храмом и одно с цветком), затем создает два фильтра, применяет их к обоим изображениям и в заключение отображает одну из результирующих карт признаков:

```

from           import load_sample_image
# Загрузить образцы изображений
china = load_sample_image("china.jpg") / 255
flower = load_sample_image("flower.jpg") / 255
images = np.array([china, flower])
batch_size, height, width, channels = images.shape

# Создать два фильтра
filters = np.zeros(shape=(7, 7, channels, 2), dtype=np.float32)
filters[:, 3, :, 0] = 1      # вертикальная линия
filters[3, :, :, 1] = 1      # горизонтальная линия
outputs = tf.nn.conv2d(images, filters, strides=1, padding="same")
plt.imshow(outputs[0, :, :, 1], cmap="gray")
    # вычертить 2-ю карту признаков 1-го изображения
plt.show()

```

Разберем код.

- Интенсивность пикселей для каждого цветового канала представляется в виде байта со значениями 0–255, поэтому мы масштабируем такие признаки, просто разделяя их на 255, чтобы получить числа с плавающей точкой от 0 до 1.
- Затем мы создаем два фильтра  $7 \times 7$  (один с вертикальной белой линией посередине, а другой с горизонтальной белой линией посередине).
- Мы применяем фильтры к обоим изображениям, используя функцию `tf.nn.conv2d()`, которая является частью низкоуровневого API-интерфейса для глубокого обучения в *TensorFlow (Deep Learning API)*. В рассматриваемом примере мы задействовали дополнение нулями (`padding="same"`) и страйд 2.
- Наконец, мы вычерчиваем одну из результирующих карт признаков (похожую на изображение справа вверху на рис. 14.5).

Строка с `tf.nn.conv2d()` заслуживает более подробных пояснений.

- параметр `images` — входной мини-пакет (четырехмерный тензор, как объяснялось ранее).
- параметр `filters` — набор фильтров, подлежащих применению (тоже четырехмерный тензор, как объяснялось ранее).
- параметр `strides` равен 1, но может быть одномерным массивом с четырьмя элементами, 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 начнут использоваться для указания страйда пакета (чтобы пропускать некоторые образцы) и страйда канала (чтобы пропускать некоторые карты признаков или каналы предыдущего слоя).

- параметр `padding` должен быть установлен либо в "same", либо в "valid".
  - Если `padding` установлен в "same", то сверточный слой при необходимости использует дополнение нулями. Выходной размер устанавливается равным округленному в большую сторону результату деления количества входных нейронов на страйд. Скажем, если входной размер составляет 13, а страйд — 5 (рис. 14.7), тогда выходным размером будет 3 (т.е.  $13 / 5 = 2.6$  с округлением в большую сторону до 3). Затем при необходимости вокруг входов как можно более равномерно добавляются нули. Когда `strides=1`, выходы слоя будут иметь те же самые размерности пространства (ширину и высоту), что и его входы, отсюда и имя "same" (одинаковые).
  - Если `padding` установлен в "valid", то сверточный слой не использует дополнение нулями и может игнорировать некоторые строки и столбцы внизу и справа входного изображения в зависимости от страйда, как демонстрируется на рис. 14.7 (для простоты показано только горизонтальное измерение, но вполне естественно, что та же самая логика применима и к вертикальному измерению). Это означает, что рецепторное поле каждого нейрона располагается строго в рамках допустимых позиций внутри входа (оно не выходит за пределы границ), отсюда и имя "valid" (допустимые).

В приведенном примере мы определяем фильтры вручную, но в реальной сети CNN вы обычно будете определять фильтры в виде обучаемых переменных, так что нейронная сеть сможет выяснить, какие фильтры работают лучше, как объяснялось ранее. Вместо ручного создания переменных используйте слой `keras.layers.Conv2D`:

```
conv = keras.layers.Conv2D(filters=32, kernel_size=3, strides=1,
                           padding="same", activation="relu")
```

Код создает слой Conv2D с 32 фильтрами, каждый размером  $3 \times 3$ , со страйдом 1 (по горизонтали и вертикали), дополнением "same" и применением функции активации ReLU к его выходам.

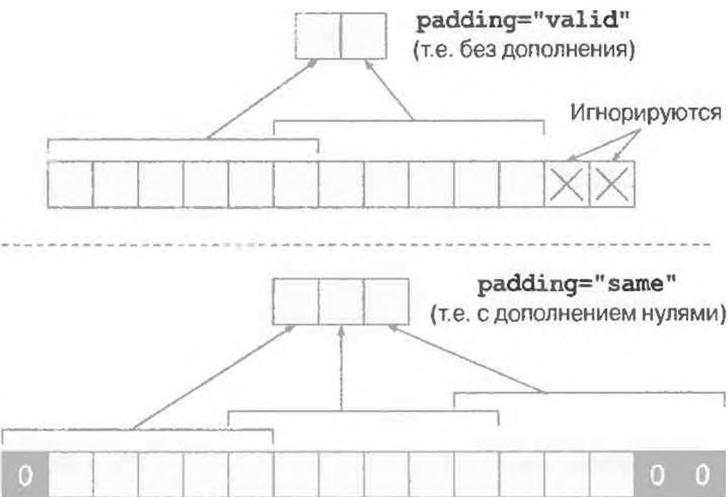


Рис. 14.7. Дополнение "same" или "valid" (с шириной входа 13, шириной фильтра 6, страйдом 5)

Как вы увидите, сверточные слои имеют немало 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 ОЗУ, 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  $5 \times 5$ , выдающий 200 карт признаков размером  $150 \times 100$ , со страйдом 1 и дополнением "same". Если на вход передается RGB-изображение  $150 \times 100$  (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 слоем<sup>7</sup>. Тем не менее, каждая из 200 карт признаков содержит  $150 \times 100$  ней-

<sup>7</sup> Полносвязный слой с  $150 \times 100$  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!

ронов, а каждый нейрон нуждается в вычислении взвешенной суммы своих  $5 \times 5 \times 3 = 75$  входов: всего получается 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-битных значений с плавающей точкой, то выход сверточного слоя будет занимать  $200 \times 150 \times 100 \times 32 = 96$  миллионов битов (12 мегабайтов) оперативной памяти<sup>8</sup>. И это лишь для одного образца! Если обучающий пакет содержит 100 образцов, тогда данный слой будет использовать вплоть до 1.2 гигабайта ОЗУ!

Во время выведения (т.е. при выработке прогноза для нового образца) оперативная память, занятая одним слоем, может быть освобождена, как только вычислен следующий слой, поэтому ОЗУ необходимо иметь столько, сколько требуют два последовательных слоя. 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 требуют все слои.



Если обучение терпит неудачу из-за ошибки, связанной с нехваткой памяти, тогда вы можете попробовать сократить размер мини-пакета. В качестве альтернативы можете попытаться понизить размерность с применением страйда или удалить несколько слоев. Либо вы можете использовать 16-битные значения с плавающей точкой вместо 32-битных. Или же можете распределить сеть CNN между множеством устройств.

А теперь давайте посмотрим на второй общий строительный блок в сетях CNN: *объединяющий слой*.

## Объединяющие слои

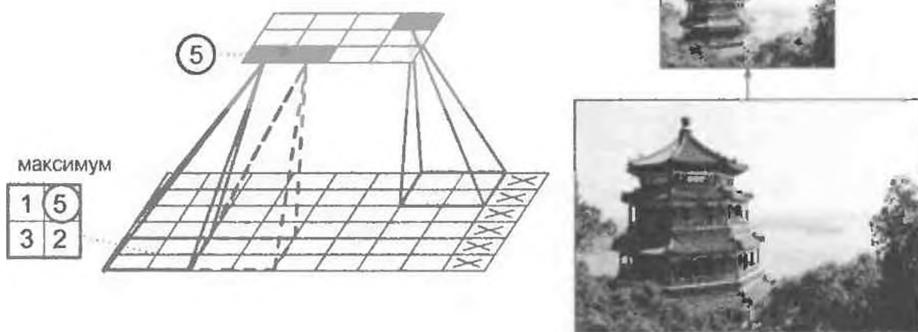
Разобравшись в том, как работают сверточные слои, понять назначение объединяющих слоев (называемых также субдискретизирующими слоями или слоями подвыборки — Примеч. пер.) довольно легко. Их цель заключается в том, чтобы *проредить* (т.е. сжать) входное изображение для сокращения вычислительной нагрузки, расхода памяти и количества параметров (тем самым ограничивая риск переобучения).

<sup>8</sup> В Международной системе единиц (СИ) 1 мегабайт = 1 000 килобайтов =  $1000 \times 1000$  байтов =  $1000 \times 1000 \times 8$  битов.

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, в точности как раньше. Однако объединяющий нейрон не имеет весов; он лишь агрегирует входы с применением функции агрегирования, такой как максимум или среднее. На рис. 14.8 показан слой объединения по максимуму (*max pooling layer*), который является самым распространенным типом объединяющего слоя. В данном примере мы используем объединяющее ядро (*pooling kernel*)<sup>9</sup> размером  $2 \times 2$ , страйд 2, без дополнения. На следующий слой попадает только максимальное входное значение в каждом рецепторном поле, тогда как остальные входы отбрасываются. Например, в левом нижнем рецепторном поле на рис. 14.8 входными значениями являются 1, 5, 3, 2, поэтому следующему слою передается только максимальное значение, 5. Из-за страйда 2 выходное изображение имеет вдвое меньшую высоту и вдвое меньшую ширину, чем входное изображение (с округлением в меньшую сторону, т.к. дополнение не применяется).



Объединяющий слой обычно работает на каждом входном канале независимо, поэтому выходная глубина будет такой же, как входная.



*Рис. 14.8. Слой объединения по максимуму  
(объединяющее ядро  $2 \times 2$ , страйд 2, без дополнения)*

<sup>9</sup> Другие ядра, которые мы обсуждали до сих пор, имеют веса, но объединяющие ядра — нет: они представляют собой просто скользящие окна, лишенные состояния.

Помимо сокращения объема вычислений, расходуемой памяти и количества параметров слой объединения по максимуму также привносит некоторую степень инвариантности (*invariance*) в небольшие трансляции (рис. 14.9). Здесь мы предполагаем, что яркие пиксели имеют значения ниже, чем темные пиксели, и рассматриваем три изображения (A, B, C), проходящие через слой объединения по максимуму с ядром  $2 \times 2$  и страйдом 2. Изображения B и C являются такими же, как изображение A, но сдвинутыми на один и два пикселя вправо. Как видите, выходы слоя объединения по максимуму для изображений A и B идентичны. Вот что означает *трансляционная инвариантность* (*translation invariance*). Для изображения C выход отличается: он сдвинут на один пиксель вправо (но по-прежнему есть 75%-ная инвариантность). Вставляя слой объединения по максимуму через каждые несколько слоев в сети CNN, можно получить определенную степень трансляционной инвариантности в большем масштабе. Кроме того, объединение по максимуму предлагает небольшую величину *ротационной* (*rotational*) инвариантности и легкую *масштабную* (*scale*) инвариантность. Такая инвариантность (даже если она ограничена) может быть полезной в случаях, когда прогноз не должен зависеть от этих деталей, как в задачах классификации.

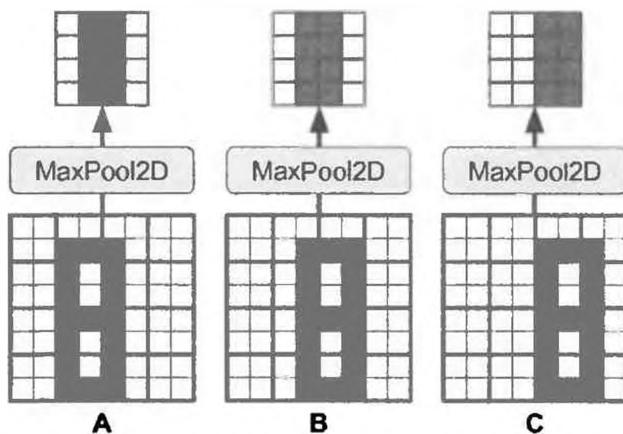


Рис. 14.9. Привнесение инвариантности в небольшие трансляции

Тем не менее, объединение по максимуму обладает и рядом недостатков. Прежде всего, оно очевидно крайне деструктивно: даже с крошечным ядром  $2 \times 2$  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. К тому же в некоторых приложениях инвариантность нежелательна. Возьмем семантическую сегментацию (зада-

чу классификации каждого пикселя внутри изображения в соответствии с классом объекта, которому он принадлежит; мы исследуем ее позже в главе): ясно, что если входное изображение транслируется на один пиксель вправо, то выход тоже окажется транслированным на один пиксель вправо. Целью в этом случае будет **эквивариантность** (*equivariance*), не инвариантность: небольшое изменение во входах должно приводить к соответствующему небольшому изменению в выходе.

## Реализация в TensorFlow

Реализовать слой объединения по максимуму в TensorFlow довольно легко. Следующий код создает слой объединения по максимуму, используя ядро  $2 \times 2$ . Страйды по умолчанию имеют размер ядра, так что слой будет применять страйд 2 (по горизонтали и по вертикали). По умолчанию он использует дополнение "valid" (т.е. вообще без дополнения):

```
max_pool = keras.layers.MaxPool2D(pool_size=2)
```

Чтобы создать слой *объединения по среднему* (*average pooling layer*), просто примените AvgPool2D вместо MaxPool2D. Как и можно было ожидать, он работает подобно слою объединения по максимуму, но рассчитывает среднее, а не максимум. Слои объединения по среднему раньше были очень популярными, но теперь люди в основном используют слои объединения по максимуму, т.к. в целом они работают лучше. Это может показаться удивительным, поскольку при вычислении среднего обычно утрачивается меньше информации, чем при расчете максимума. Но с другой стороны объединение по максимуму предохраняет только самые сильные признаки, избавляясь от всех незначащих признаков, так что последующие слои получают более чистый сигнал для обработки. Вдобавок объединение по максимуму обеспечивает более сильную трансляционную инвариантность, чем объединение по среднему, и требует чуть меньшего объема вычислений.

Обратите внимание, что объединение по максимуму и объединение по среднему могут выполняться по измерению в глубину, а не по пространственным измерениям, хотя и не так часто. Такой прием может позволить сети CNN научиться быть инвариантной к разнообразным признакам. Скажем, она могла бы узнавать множество фильтров, каждый из которых обнаруживает отличающийся поворот того же образа (такого как рукописные цифры (рис. 14.10)), и слой объединения по максимуму в глубину гарантировал бы, что выход будет одинаковым независимо от поворота. Сеть CNN могла

бы аналогичным образом научиться быть инвариантной к чему-нибудь еще: толщине, яркости, скосу, цвету и т.д.

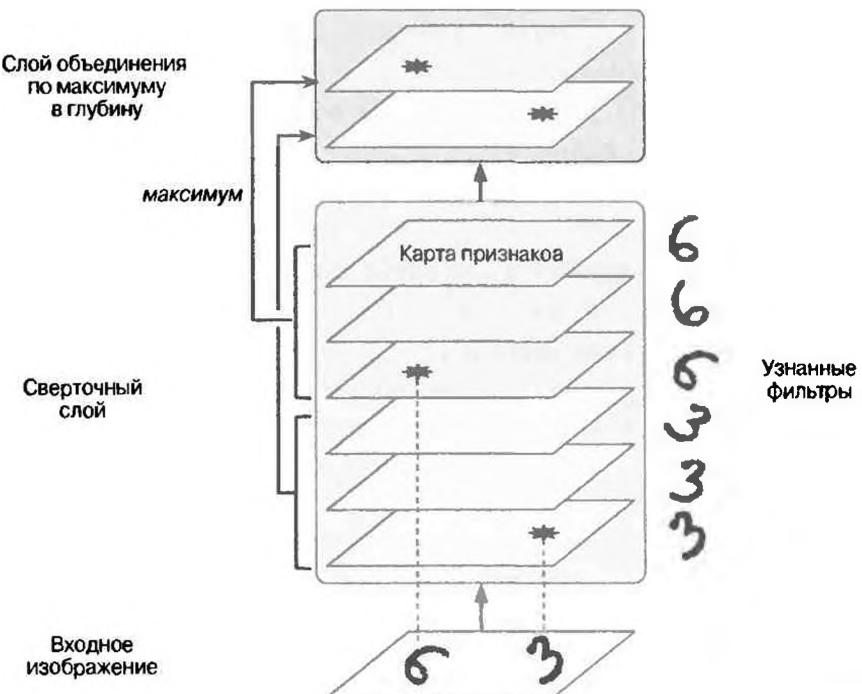


Рис. 14.10. Объединение по максимуму в глубину может помочь сети CNN научиться любой инвариантности

Библиотека Keras не включает слой объединения по максимуму в глубину, но низкоуровневый API-интерфейс для глубокого обучения в TensorFlow его содержит: просто вызовите функцию `tf.nn.max_pool()`, передав ей размер ядра и страйды в виде кортежей размера 4. Первые три значения каждого кортежа должны быть равны 1, указывая на то, что размер ядра и страйды по измерениям пакета, высоты и ширины обязаны быть 1. Последнее значение должно быть любым желаемым размером ядра и страйдом по измерению глубины — например, 3 (обязано быть делителем входной глубины; оно не будет работать, если предыдущий слой выдает 20 карт признаков, т.к. 20 не кратно 3):

```
output = tf.nn.max_pool(images,
                        ksize=(1, 1, 1, 3),
                        strides=(1, 1, 1, 3),
                        padding="valid")
```

Если вы хотите добавить это в качестве слоя в свою модель Keras, тогда поместите его внутрь слоя Lambda (или создайте специальный слой Keras):

```
depth_pool = keras.layers.Lambda(  
    lambda X: tf.nn.max_pool(X, ksize=(1, 1, 1, 3),  
                             strides=(1, 1, 1, 3),  
                             padding="valid"))
```

Последней разновидностью слоя объединения, которую вы часто будете встречать в современных архитектурах, является слой объединения по глобальному среднему (*global average pooling layer*). Он работает совсем по-другому: все, что он делает — рассчитывает среднее каждой полной карты признаков (он похож на слой объединения по среднему, который применяет объединяющее ядро с такими же пространственными измерениями, как у входов). В итоге он выдает единственное число на карту признаков и на образец. Несмотря на то, что такой слой вполне понятно крайне деструктивен (большая часть информации в карте признаков утрачивается), он может оказаться полезным в качестве выходного слоя, как мы увидим позже в главе. Для создания слоя подобного рода используйте класс keras.layers.GlobalAvgPool2D:

```
global_avg_pool = keras.layers.GlobalAvgPool2D()
```

Данный слой эквивалентен простому слою Lambda, который вычисляет среднее по пространственным измерениям (высоты и ширины):

```
global_avg_pool =  
    keras.layers.Lambda(lambda X: tf.reduce_mean(X, axis=[1, 2]))
```

Теперь вы знаете о строительных блоках все, чтобы создавать сети 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 слоев 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. 14.11). На верхушку стопки добавляется обыкновенная нейронная сеть прямого распространения, состоящая из нескольких полно связанных слоев (плюс слоев ReLU), и финальный

слой выпускает прогноз (например, многопеременный слой, который выдает оценки вероятностей классов).

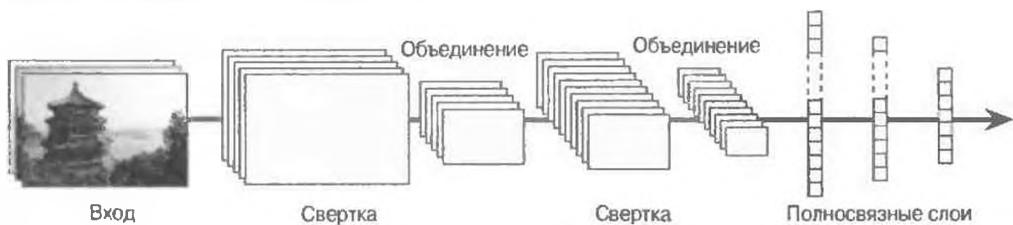


Рис. 14.11. Типовая архитектура сети CNN



Общее заблуждение связано с применением слишком крупных сверточных ядер. Скажем, вместо использования сверточного слоя с ядром  $5 \times 5$  уложите стопкой два слоя с ядрами  $3 \times 3$ : в результате потребуется меньше параметров и меньший объем вычислений, а решение обычно будет работать лучше. Единственное исключение относится к первому сверточному слою: обычно он может иметь крупное ядро (например,  $5 \times 5$ ), как правило, со страйдом 2 или выше: это понизит пространственное измерение изображения без утраты слишком большого объема информации, и поскольку входное изображение имеет в общем только три канала, затраты будут не особо велики.

Ниже показано, как можно реализовать простую сеть CNN, которая занимается набором данных (введенным в главе 10):

```
model = keras.models.Sequential([
    keras.layers.Conv2D(64, 7, activation="relu", padding="same",
                       input_shape=[28, 28, 1]),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(128, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.Conv2D(256, 3, activation="relu", padding="same"),
    keras.layers.MaxPooling2D(2),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation="softmax")])
```

1)

Давайте пройдемся по модели.

- Первый слой применяет 64 довольно крупных фильтра ( $7 \times 7$ ), но без страйда, потому что входные изображения не особенно велики. Он также устанавливает `input_shape=[28, 28, 1]`, т.к. изображения имеют размер  $28 \times 28$  пикселей с единственным цветовым каналом (т.е. полутона).
- Далее следует слой объединения по максимуму, который использует размер объединения 2, так что делит каждое пространственное измерение на коэффициент 2.
- Затем мы повторяем ту же самую структуру дважды: два сверточных слоя, за которыми находится слой объединения по максимуму. Для более крупных изображений мы могли бы повторить такую структуру еще несколько раз (количество повторений является гиперпараметром, допускающим подстройку).
- Обратите внимание, что количество фильтров растет по мере того, как мы поднимаемся вверх по сети CNN в направлении выходного слоя (первоначально оно составляет 64, далее 128, затем 256). Увеличивать его имеет смысл, поскольку количество низкоуровневых признаков часто оказывается довольно низким (скажем, небольшие кружочки, горизонтальные линии), но есть много разных способов их комбинирования в высокоуровневые признаки. Распространенная практика предусматривает удвоение количества фильтров после каждого объединяющего слоя: поскольку объединяющий слой делит каждое пространственное измерение на коэффициент 2, мы можем себе позволить удваивать количество карт признаков в следующем слое, не опасаясь взрывоопасного роста количества параметров, потребляемой памяти или вычислительной нагрузки.
- Далее располагается полносвязная сеть, образованная из двух плотных скрытых слоев и плотного выходного слоя. Имейте в виду, что мы должны расправить ее входы, т.к. плотная сеть ожидает получать одномерный массив признаков для каждого образца. Мы также добавляем два слоя отключения с долей отключения 50% в каждом, чтобы снизить риск переобучения.

Построенная сеть CNN достигает правильности свыше 92% на испытательном наборе. Не самый современный технический уровень, но довольно-таки хорошо и, несомненно, намного лучше правильности, которой мы добивались с помощью плотных сетей в главе 10.

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 Large Scale Visual Recognition Challenge* — кампания по широкомасштабному распознаванию образов в ImageNet; <http://image-net.org/>). В рамках данного состязания частота ошибок топ-5 для классификации изображений упала с более 26% до менее 2.3% 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 1000 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 г.), а затем три победивших решения задачи ILSVRC: AlexNet (2012 г.), GoogLeNet (2014 г.) и ResNet (2015 г.).

## LeNet-5

Архитектура LeNet-5 (<https://homl.info/lenet5>)<sup>10</sup> является, вероятно, 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. 14.1.

Ниже отмечено несколько добавочных деталей.

- Изображения MNIST имеют размер  $28 \times 28$  пикселей, но перед передачей в сеть они дополняются нулями до  $32 \times 32$  пикселей и нормализуются. В оставшейся части сети никакое дополнение не используется, а потому по мере прохождения через сеть размер продолжает уменьшаться.

<sup>10</sup> Ян Лекун и др., *Gradient-Based Learning Applied to Document Recognition* (Основанное на градиентах обучение, примененное к распознаванию документов), *Proceedings of the IEEE* 86, номер 11 (1998 г.): с. 2278–2324.

Таблица 14.1. Архитектура LeNet-5

Слой	Тип	Карты	Размер	Размер ядра	Страйд	Активация
Out (выходной)	Полносвязный	—	10	—	—	RBF
F6	Полносвязный	—	84	—	—	tanh
S5	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	—	—	—

- 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 C3 связываются с нейронами лишь из трех или четырех карт S2 (вместо всех шести карт S2). За деталями обращайтесь к таблице 1 (с. 8) в исходной статье (<https://yann.lecun.com/exdb/lenet/>).
- Выходной слой несколько специфичен: вместо матричного перемножения входов и вектора весов каждый нейрон выдает возвведенное в квадрат евклидово расстояние между своим вектором входов и своим вектором весов. Каждый выход измеряет, в какой степени изображение принадлежит отдельному классу цифры. Теперь отдается предпочтение функции издерек на основе перекрестной энтропии, т.к. она намного больше штрафует неправильные прогнозы, порождая крупные градиенты и в итоге ускоряя схождение.

На веб-сайте Яна Лекуна (<http://yann.lecun.com/exdb/lenet/index.html>) предлагаются замечательные демонстрации классификации цифр LeNet-5.

## AlexNet

В 2012 году архитектура сетей CNN под названием *AlexNet* (<https://hml.info/80>)<sup>11</sup> с большим отрывом победила в решении задачи 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 Джоном Хинтоном. Она похожа на LeNet-5, но гораздо крупнее и глубже, к тому же сверточные слои в ней впервые укладывались непосредственно друг на друга вместо помещения объединяющего слоя поверх каждого сверточного слоя. Архитектура AlexNet представлена в табл. 14.2.

Таблица 14.2. Архитектура AlexNet

Слой	Тип	Карты	Размер	Размер ядра	Страйд	Дополнение	Активация
Out (выходной)	Полносвязный	—	1 000	—	—	—	Многопараметрическая
F9	Полносвязный	—	4 096	—	—	—	ReLU
F8	Полносвязный	—	4 096	—	—	—	ReLU
C7	Сверточный	256	$13 \times 13$	$3 \times 3$	1	same	ReLU
C6	Сверточный	384	$13 \times 13$	$3 \times 3$	1	same	ReLU
C5	Сверточный	384	$13 \times 13$	$3 \times 3$	1	same	ReLU
S4	Объединение по максимуму	256	$13 \times 13$	$3 \times 3$	2	valid	—
C3	Сверточный	256	$27 \times 27$	$5 \times 5$	1	same	ReLU
S2	Объединение по максимуму	96	$27 \times 27$	$3 \times 3$	2	valid	—
C1	Сверточный	96	$55 \times 55$	$11 \times 11$	4	same	ReLU
In (входной)	Входной	3 (RGB)	$227 \times 227$	—	—	—	—

Чтобы сократить риск переобучения, авторы использовали две методики регуляризации. Во-первых, при обучении они применяли к выходам слоев F8 и F9 отключение (представленное в главе 11) с долей отключения 50%. Во-вторых, они реализовали дополнение данных (*data augmentation*), случайным образом сдвигая обучающие изображения на различные смещения, переворачивая их по горизонтали и изменяя условия освещения.

<sup>11</sup> Алекс Крижевский и др., *ImageNet Classification with Deep Convolutional Neural Networks* (Классификация ImageNet с помощью глубоких сверточных нейронных сетей), *Proceedings of the 25th International Conference on Neural Information Processing Systems 1* (2012 г.): с. 1097–1105.

## Дополнение данных

Дополнение данных искусственно увеличивает размер обучающего набора за счет генерации множества реалистичных вариантов для каждого обучающего образца. В итоге сокращается риск переобучения, что делает дополнение данных методикой регуляризации. Генерируемые образцы должны быть как можно более реалистичными: в идеале, взяв изображение из дополненного обучающего набора, человек не должен быть в состоянии сказать, дополненное оно или нет. Простое добавление белого шума не поможет; модификации обязаны быть обучаемыми (белый шум таковым не является).

Например, вы можете немного сдвинуть, повернуть и изменить размеры каждой фотографии из обучающего набора на варьирующиеся величины, после чего добавить результирующие фотографии в обучающий набор (рис. 14.12). Это заставит модель быть толерантной к вариациям в позиции, ориентации и размерах объектов на фотографиях. Для модели, толерантной к разным условиям освещения, вы можете аналогичным образом сгенерировать множество фотографий с разнообразными уровнями контрастности. В общем случае вы также можете поворачивать фотографии горизонтально (исключая текст и другие асимметричные объекты). Комбинируя такие трансформации, вы сумеете значительно увеличить размер своего обучающего набора.

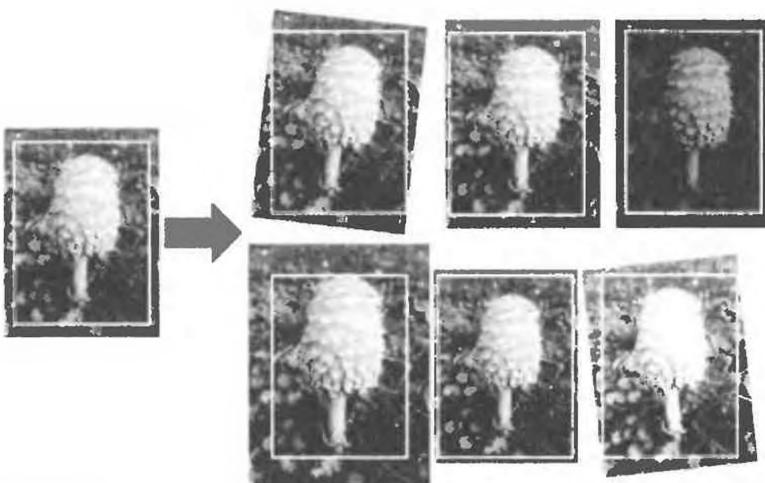


Рис. 14.12. Генерация новых обучающих образцов на основе существующих

Непосредственно после шага ReLU слоев C1 и C3 в AlexNet также используется шаг состязательной нормализации (competitive normalization), называемый локальной нормализацией ответа (*local response normalization — LRN*): наиболее сильно активированные нейроны подавляют нейроны, которые находятся в тех же самых позициях в соседствующих картах признаков (состязательная активация наблюдалась в биологических нейронах). Такой 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 14.2 показано, как применять LRN.

### Уравнение 14.2. Локальная нормализация ответа

$$b_i = a_i \left( k + \alpha \sum_{j=j_{\text{low}}}^{j_{\text{high}}} a_j^2 \right)^{-\beta} \quad \text{с } \begin{cases} j_{\text{high}} = \min \left( i + \frac{r}{2}, f_n - 1 \right) \\ j_{\text{low}} = \max \left( 0, i - \frac{r}{2} \right) \end{cases}$$

Вот что входит в уравнение.

- $b_i$  — нормализованный выход нейрона, расположенного внутри карты признаков  $i$ , в некоторой строке  $i$  и столбце  $v$  (обратите внимание, что в уравнении мы учтем только нейроны, находящиеся в данной строке и столбце, а потому  $i$  и  $v$  не показаны).
- $a_i$  — активация этого нейрона после шага ReLU, но перед нормализацией.
- $k, \alpha, \beta$  и  $r$  представляют собой 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 его собственной карты.

Гиперпараметры в AlexNet устанавливаются следующим образом:  $r = 2$ ,  $\alpha = 0.00002$ ,  $\beta = 0.75$  и  $k = 1$ . Шаг LRN может быть реализован с помощью функции `tf.nn.local_response_normalization()`, которую вы можете поместить внутрь слоя Lambda, если хотите ее использовать в модели Keras.

В 2013 году Мэттью Зайлер и Роб Фергюс разработали вариант AlexNet под названием сеть ZF Net (<https://homl.info/zfnet>)<sup>12</sup>, которая победила в решении задачи ILSVRC 2013. По существу она представляет собой AlexNet с некоторыми подстроенными гиперпараметрами (количество карт признаков, размер ядра, страйд и т.д.).

## GoogLeNet

Архитектура GoogLeNet (<https://homl.info/81>) была разработана Кристианом Сегеди и др. из Google Research<sup>13</sup> и победила в решении задачи ILSVRC 2014, уронив частоту ошибок top-5 ниже 7%. Столь высокая эффективность по большей части проистекала из того факта, что сеть была намного глубже предшествующих сетей CNN (как вы увидите на рис. 14.14 далее в главе). Это стало возможным благодаря подсетям, называемым *модулями начала* (*inception module*)<sup>14</sup>, которые позволяют GoogLeNet использовать параметры гораздо эффективнее, чем предыдущие архитектуры: GoogLeNet фактически имеет в 10 раз меньше параметров, чем AlexNet (приблизительно 6 миллионов вместо 60 миллионов).

На рис. 14.13 показана архитектура модуля начала. Запись “ $3 \times 3 + 2(S)$ ” означает, что слой применяет ядро  $3 \times 3$ , страйд 2 и дополнение “same”. Входной сигнал сначала копируется и подается в четыре разных слоя. Все сверточные слои используют функцию активации ReLU. Обратите внимание, что второй набор сверточных слоев применяет ядра отличающихся размеров ( $1 \times 1$ ,  $3 \times 3$  и  $5 \times 5$ ), 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 concatenation layer*), т.е. укладывание друг на друга карт признаков из всех четырех верхних сверточных слоев.

<sup>12</sup> Мэттью Зайлер и Роб Фергюс, *Visualizing and Understanding Convolutional Networks* (Визуализация и осмысление сверточных сетей), *Proceedings of the European Conference on Computer Vision* (2014 г.): с. 818–833.

<sup>13</sup> Кристиан Сегеди и др., *Going Deeper with Convolutions* (Углубление с помощью сверток), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015 г.): с. 1–9.

<sup>14</sup> В фильме “Inception” (“Начало”) 2010 г. действующие лица погружались все глубже и глубже в многочисленные слои сновидений, отсюда и такое название у модулей.

В TensorFlow такой слой конкатенации может быть реализован посредством операции `tf.concat()` с `axis=3` (ось глубины).

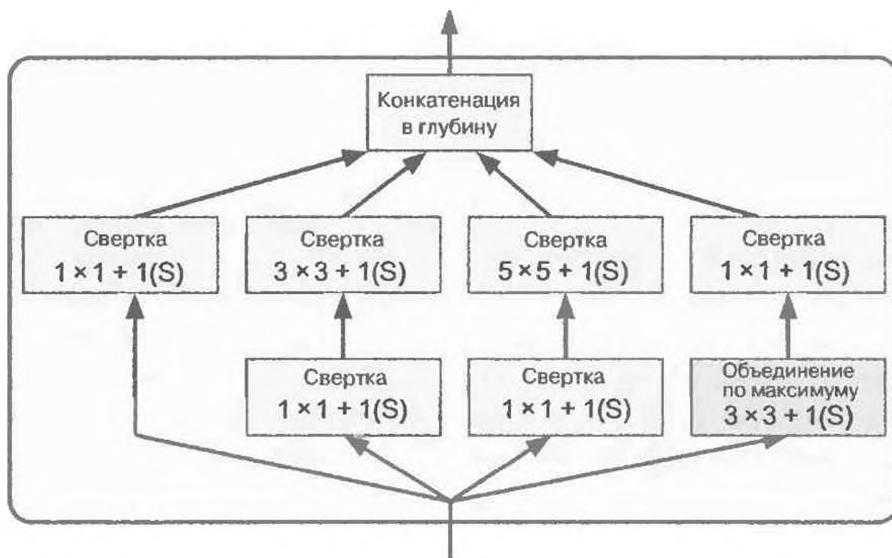


Рис. 14.13. Модуль начала

Вас может интересовать, почему модули начала имеют сверточные слои с ядрами  $1 \times 1$ . Действительно ли эти слои не могут захватывать любые признаки, поскольку видят только один пиксель за раз? По сути, такие слои служат трем целям.

- Хотя они не могут захватывать пространственные образы, они способны захватывать образы по измерению глубины.
- Они сконфигурированы на выдачу меньшего числа карт признаков, чем их входы, так что выступают в качестве *суживающих слоев* (*bottleneck layer*), т.е. понижают размерность. В результате снижаются вычислительные затраты и количество параметров, из-за чего ускоряется обучение и улучшается обобщение.
- Каждая пара сверточных слоев ( $[1 \times 1, 3 \times 3]$  и  $[1 \times 1, 5 \times 5]$ ) действует подобно одиночному и мощному сверточному слою, который может захватывать более сложные образы. На самом деле вместо пропускания 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. 14.14). Количество карт признаков, выдаваемых каждым сверточным слоем и каждым слоем объединения, указывается перед размером ядра. Архитектура настолько глубока, что ее пришлось изображать в виде трех колонок, но на самом деле GoogLeNet выглядит как одна высокая стопка, включающая девять модулей начала (прямоугольники с волчками). Шесть чисел в модулях начала представляют количество карт признаков, выдаваемых каждым сверточным слоем в модуле (в том же порядке, как на рис. 14.13). Обратите внимание, что все сверточные слои применяют функцию активации ReLU.

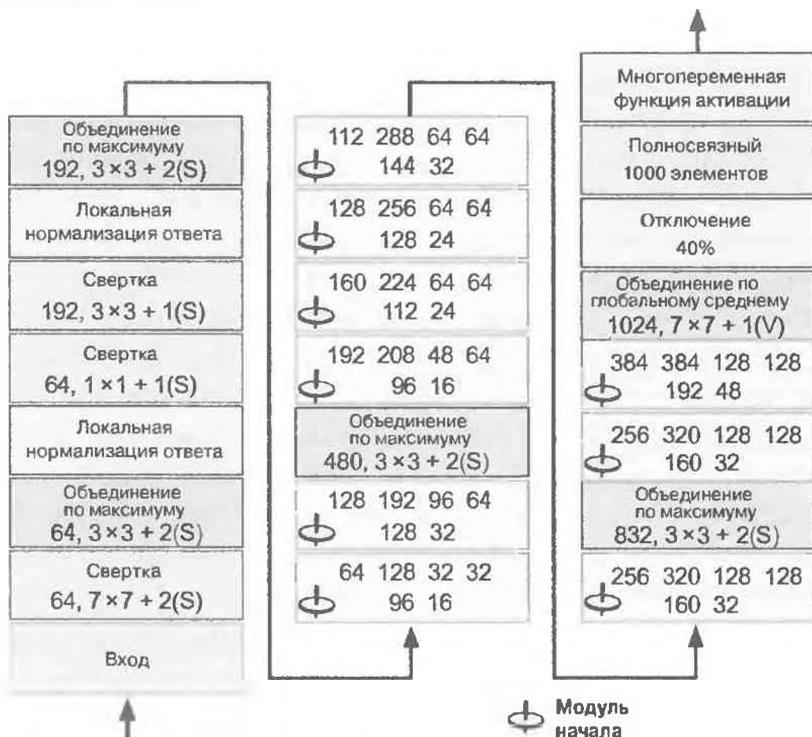


Рис. 14.14. Архитектура GoogLeNet

## Пройдемся по этой сети.

- Первые два слоя делят высоту и ширину изображения на 4 (так что его область делится на 16) для снижения вычислительной нагрузки. Первый слой использует крупный размер ядра, так что большая часть информации предохраняется.
- Затем слой локальной нормализации ответа гарантирует то, 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.
- Очередной слой локальной нормализации ответа гарантирует то, 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 глобальному среднему выдает среднее каждой карты признаков: это отбрасывает любую оставшуюся пространственную информацию, что хорошо, поскольку в данной точке пространственной информации было не так уж много. В действительности обычно ожидается, что входные изображения GoogLeNet будут иметь размер  $224 \times 224$  пикселей, поэтому после 5 слоев объединения по максимуму, каждый из которых делит высоту и ширину на 2, карты признаков доводятся до размера  $7 \times 7$ . Кроме того, мы имеем дело с задачей классификации, а не установления местонахождения, и потому не имеет значения, где находится объект. Благодаря понижению размерности, обеспечиваемому данным слоем, нет необходимости иметь несколько полносвязных слоев в верхней части сети CNN (подобно AlexNet), и это значительно уменьшает количество параметров в сети, а также ограничивает риск переобучения.
- Последние слои самоочевидны: слой отключения для регуляризации и затем полносвязный слой с 1000 элементов (т.е. есть 1000 классов) и многограниченной функцией активации для выдачи оценки 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. Целями была борьба с проблемой исчезновения градиентов и регуляризация сети. Тем не менее, позже они показали, что эффект от них оказался относительно небольшим.

Позже исследователи из Google предложили несколько вариантов архитектуры GoogLeNet, в том числе Inception-v3 и Inception-v4, использующие слегка отличающиеся модули начала и достигающие даже лучшей эффективности.

## VGGNet

Второе место в решении задачи ILSVRC 2014 заняла сеть VGGNet (<https://homl.info/83>)<sup>15</sup>, разработанная Кареном Симоняном и Эндрю Циссерманом из исследовательской лаборатории VGG (*Visual Geometry Group — Группа зрительной геометрии*) в Оксфордском университете. Она имела очень простую и классическую архитектуру с 2 или 3 сверточными слоями и объединяющим слоем, затем снова 2 или 3 сверточными слоями и объединяющим слоем и т.д. (доходя до общего количества в 16 или 19 сверточных слоев в зависимости от варианта VGGNet) плюс финальная плотная сеть с 2 скрытыми слоями и выходным слоем. В ней применялись только фильтры  $3 \times 3$ , но много.

## ResNet

Победителем в решении задачи ILSVRC 2015 стала остаточная сеть (*Residual Network* или *ResNet*), разработанная Каймингом Хе и др. (<https://homl.info/82>)<sup>16</sup>, которая давала поразительную частоту ошибок топ-5 ниже 3.6%. В выигравшем варианте использовалась крайне глубокая сеть CNN, состоящая из 152 слоев (в других вариантах было 34, 50 и 101 слой).

<sup>15</sup> Карен Симонян и Эндрю Циссерман, *Very Deep Convolutional Networks for Large-Scale Image Recognition* (Очень глубокие сверточные сети для крупномасштабного распознавания изображений), препринт arXiv:1409.1556 (2014 г.).

<sup>16</sup> Кайминг Хе и др., *Deep Residual Learning for Image Recognition* (Глубокое остаточное обучение для распознавания изображений), препринт arXiv:1512:03385 (2015 г.).

Она подтвердила общую тенденцию: модели становятся все глубже и глубже, а количество параметров все уменьшается и уменьшается. 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. 14.15.

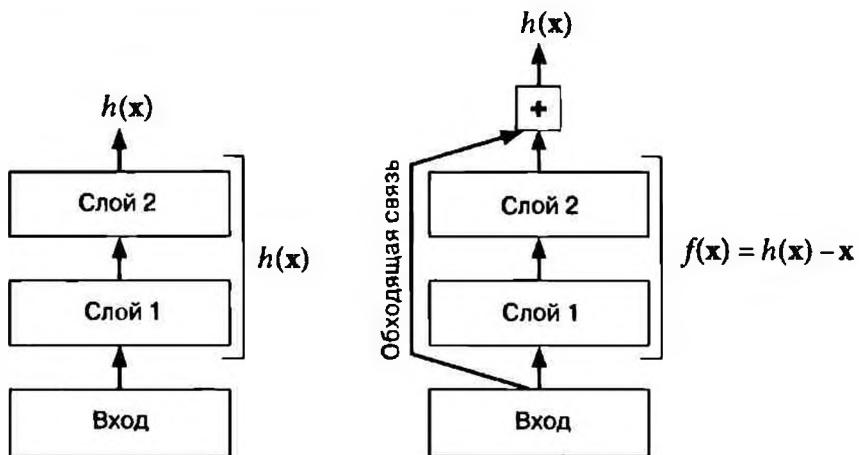


Рис. 14.15. Остаточное обучение

После инициализации обычной 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), обучение значительно ускорится.

Кроме того, если добавить много обходящих связей, то сеть может делать успехи, даже когда некоторые слои еще не начали обучаться (рис. 14.16). Благодаря обходящим связям сигнал может легко отыскать свой путь через всю сеть. Глубокую остаточную сеть можно рассматривать как стопку остаточных элементов (*residual unit* — *RU*), где каждый остаточный элемент представляет собой небольшую нейронную сеть с обходящей связью.

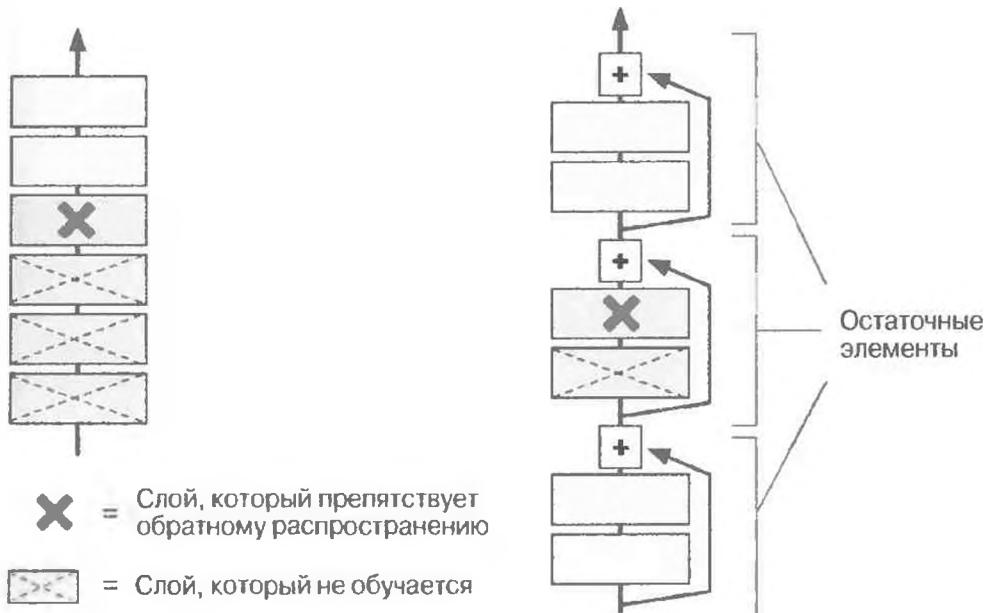


Рис. 14.16. Обыкновенная глубокая нейронная сеть (слева) и глубокая остаточная сеть (справа)

А теперь рассмотрим архитектуру ResNet (рис. 14.17). Она удивительно 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 (без объединяющего слоя!) с пакетной нормализацией (BN) и активацией ReLU, которые применяют ядра  $3 \times 3$  и предохраняют пространственные измерения (страйд 1, дополнение "same").

Обратите внимание, что каждые несколько остаточных элементов удваивают количество карт признаков, одновременно деля пополам их высоту и ширину (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. 14.17 пунктирной линией со стрелкой). Чтобы решить проблему, входы пропускаются через сверточный слой  $1 \times 1$  со страйдом 2 и правильным количеством выходных карт признаков (рис. 14.18).

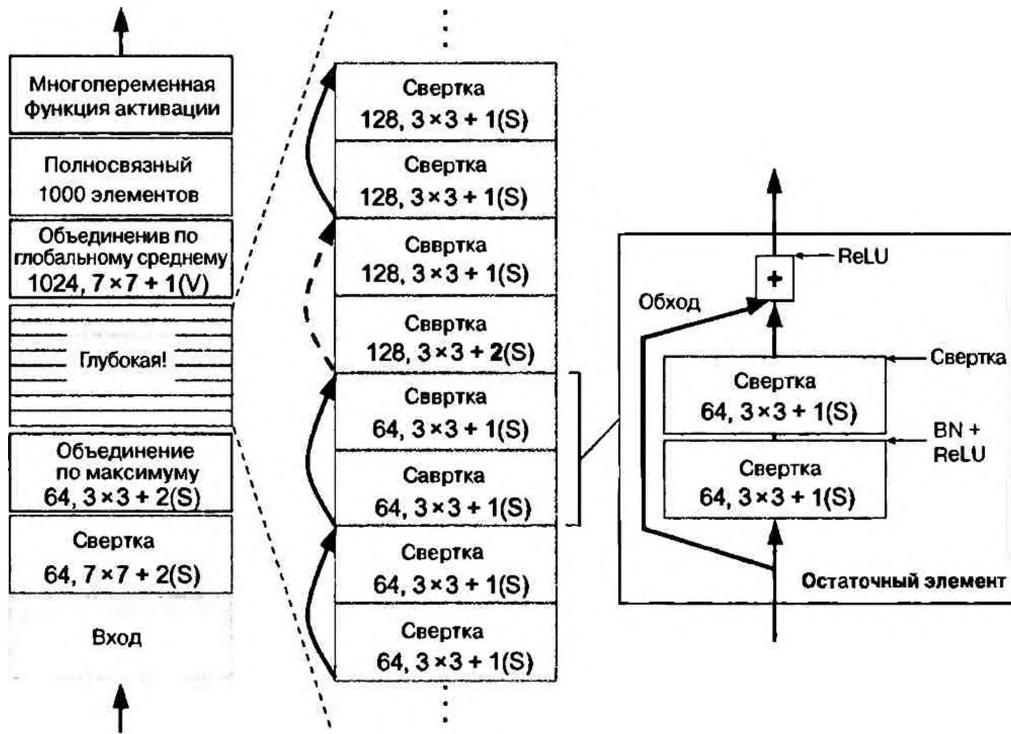


Рис. 14.17. Архитектура ResNet

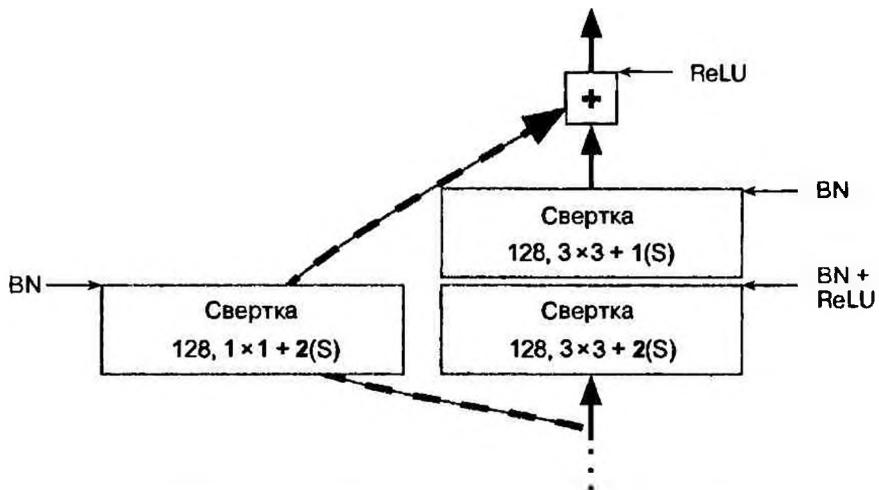


Рис. 14.18. Обходящая связь при изменении размера и глубины карты признаков

ResNet-34 представляет собой сеть ResNet с 34 слоями (подсчитываются только сверточные слои и полносвязный слой)<sup>17</sup>, 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. Позже в главе мы реализуем такую архитектуру.

В более глубокой сети ResNet, такой как ResNet-152, применяются немногого отличающиеся остаточные элементы. Вместо двух сверточных слоев  $3 \times 3$  со, скажем, 256 картами признаков они используют три сверточных слоя: первый сверточный слой  $1 \times 1$  со всего лишь 64 картами признаков (в 4 раза меньше), действующий как суживающий слой (обсуждался ранее), затем сверточный слой  $3 \times 3$  с 64 картами признаков и, наконец, еще один сверточный слой  $1 \times 1$  с 256 картами признаков (4 раза по 64), который восстанавливает первоначальную глубину. Сеть ResNet-152 содержит 3 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 2048 картами.



Архитектура Inception-v4 (<https://homl.info/84>)<sup>18</sup> от Google объединила идеи GoogLeNet и ResNet и при классификации ImageNet достигла частоты ошибок топ-5, близкой к 3%.

## Xception

Заслуживает упоминания еще один вариант архитектуры GoogLeNet: Xception (<https://homl.info/xception>)<sup>19</sup>, означающий *Extreme Inception* (экстремальное начало). Архитектура Xception была предложена в 2016 году Франсуа Шолле (автор Keras) и она значительно превосходит Inception-v3 на огромных задачах зрения (350 миллионов изображения и 17 000 классов). Подобно Inception-v4 она объединяет идеи GoogLeNet и ResNet, но заменяет модули начала слоем специального типа, который называется *сепарабельным слоем свертки по глубине* (*depthwise separable convolution layer*) или для крат-

<sup>17</sup> При описании нейронной сети принято считать только слои с параметрами.

<sup>18</sup> Кристиан Сегеди и др., *Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning* (Inception-v4, Inception-ResNet и влияние остаточных связей на обучение), препринт arXiv:1602.07261 (2016 г.).

<sup>19</sup> Франсуа Шолле, *Xception: Deep Learning with Depthwise Separable Convolutions* (Xception: глубокое обучение с помощью сепарабельных сверток по глубине), препринт arXiv: 1610.02357 (2016 г.).

кости сепарабельным сверточным слоем (*separable convolution layer*)<sup>20</sup>. Такие слои применялись ранее в ряде архитектур CNN, но они не были настолько центральными, как в архитектуре Xception. Наряду с тем, что обыкновенный сверточный слой использует фильтры, которые стараются одновременно захватить пространственные образы (скажем, овал) и межканальные образы (например, рот + нос + глаза = лицо), сепарабельный сверточный слой делает сильное предположение о том, что пространственные и межканальные образы могут моделироваться отдельно (рис. 14.19). Таким образом, он состоит из двух частей: первая часть применяет единственный пространственный фильтр для каждой входной карты признаков, после чего вторая часть ищет исключительно межканальные образы — это просто обыкновенный сверточный слой с фильтрами  $1 \times 1$ .

Поскольку сепарабельные сверточные слои имеют только один пространственный фильтр на входной канал, вы должны избегать использования их после слоев со слишком малым числом каналов, таких как входной слой (да, это то, что изображено на рис. 14.19, но лишь в целях иллюстрации).

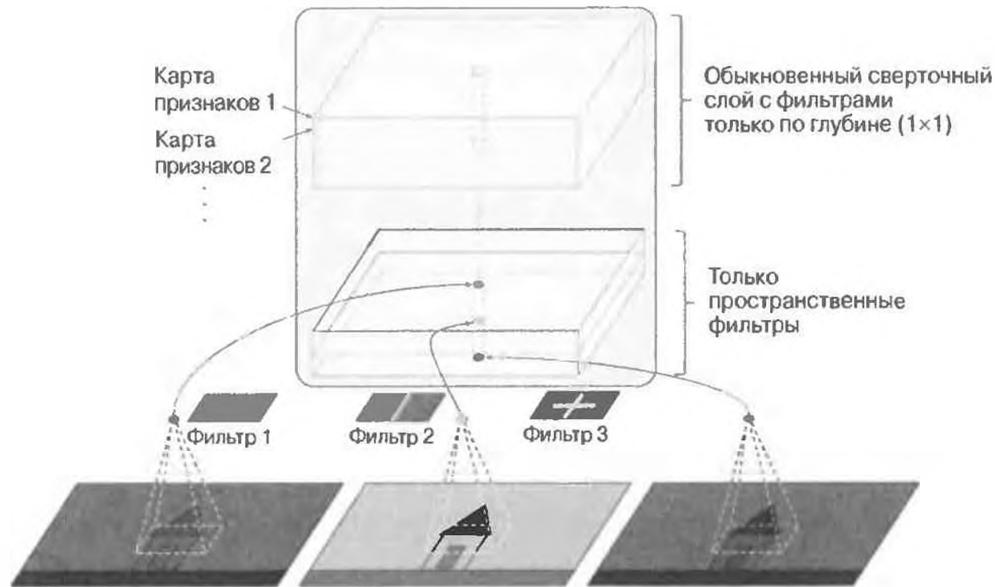


Рис. 14.19. Сепарабельный слой свертки по глубине

<sup>20</sup> Указанное название временами может быть неоднозначным, поскольку *пространственно сепарабельные свертки* (*spatially separable convolution*) нередко называют просто *сепарабельными свертками* (*separable convolution*).

По указанной причине архитектура Xception начинается с 2 обычновенных сверточных слоев, но затем в оставшейся части архитектуры применяются только сепарабельные свертки (всего их 34) плюс несколько слоев объединения по максимуму и обычные финальные слои (слой объединения по глобальному среднему и плотный выходной слой).

Вы можете задаться вопросом, почему архитектура Xception считается вариантом GoogLeNet с учетом того, что она вообще не содержит модулей начала. Как мы обсуждали ранее, модуль начала включает в себя сверточные слои с фильтрами  $1 \times 1$ : они ищут исключительно межканальные образы. Однако идущие следом сверточные слои являются обычновенными сверточными слоями, которые ищут и пространственные, и межканальные образы. Следовательно, вы можете думать о модуле начала как о промежуточном компоненте между обычновенным сверточным слоем (который учитывает пространственные и межканальные образы вместе) и сепарабельным сверточным слоем (который учитывает их по отдельности). На практике сепарабельные сверточные слои, по-видимому, в целом работают лучше.



Сепарабельные сверточные слои используют меньше параметров, меньше памяти и меньше вычислений, чем обычновенные сверточные слои, и в общем случае они работают даже лучше, поэтому вы должны рассматривать возможность их применения по умолчанию (но не после слоев с небольшим количеством каналов).

В решении задачи ILSVRC 2016 победила команда CUImage из Китайского университета Гонконга. Они использовали ансамбль из множества разных методик, включая сложно устроенную систему выявления объектов под названием GBD-Net (<https://homl.info/gbdnet>)<sup>21</sup>, чтобы достичь частоты ошибок топ-5 ниже 3%. Хотя результат, безусловно, впечатляет, сложность решения разительно контрастирует с простотой сетей ResNet. Кроме того, как мы вскоре увидим, год спустя появилось еще одна довольно простая архитектура, которая работала даже лучше.

<sup>21</sup> Хингуй Цзэн и др., *Crafting GBD-Net for Object Detection* (Изготовление GBD-Net для выявления объектов), *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, номер 9 (2018 г.): с. 2109–2123.

## SENet

В решении задачи ILSVRC 2017 победила сеть сжатия и возбуждения (*Squeeze-and-Excitation Network* — SENet; <https://hml.info/senet>)<sup>22</sup>. Ее архитектура расширяет существующие архитектуры, такие как сети с модулями начала и ResNet, и поднимает их эффективность. Это позволило сети SENet победить с поразительной частотой ошибок топ-5, составившей 2.25%! Расширенные версии сетей с модулями начала ResNet называются соответственно *SE-Inception* и *SE-ResNet*. Подъем объясняется тем фактом, что SENet добавляет небольшую нейронную сеть под названием блок SE (*SE block*) к каждому элементу в исходной архитектуре (т.е. к каждому модулю начала или каждому остаточному элементу), как показано на рис. 14.20.

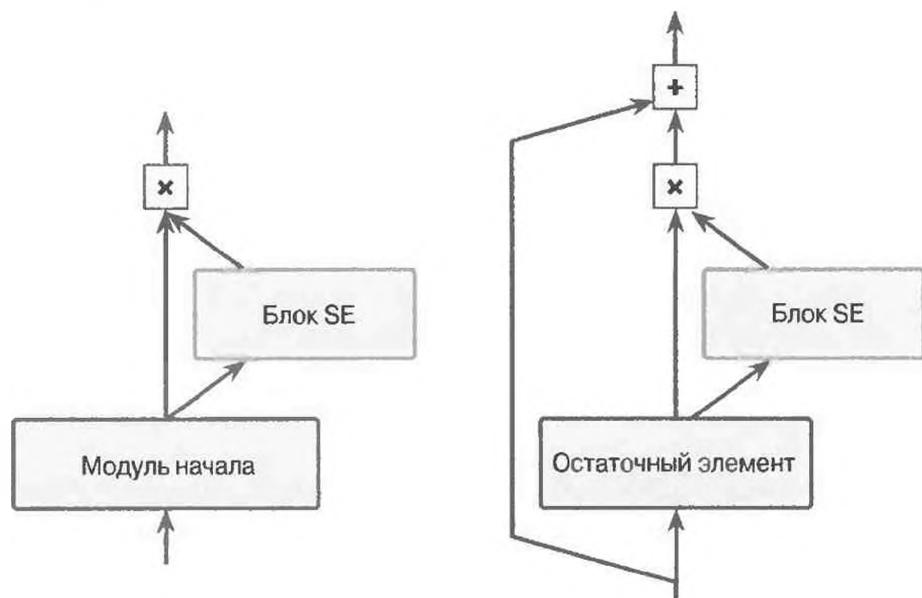
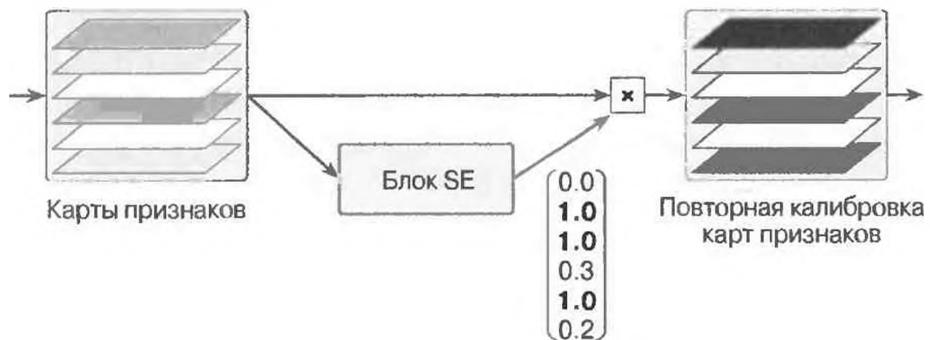


Рис. 14.20. Модуль SE-Inception (слева) и элемент SE-ResNet (справа)

Блок SE анализирует выход элемента, к которому он присоединен, фокусируясь исключительно на измерении глубины (он не ищет пространственные образы), и выясняет, какие признаки обычно наиболее активны вместе. Затем он применяет эту информацию для повторной калибровки карт признаков (рис. 14.21).

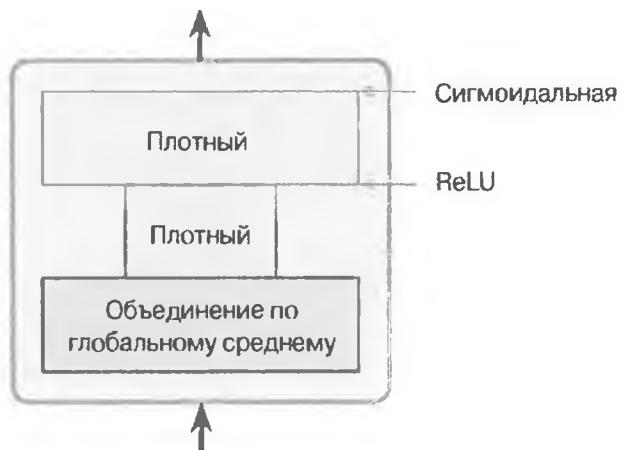
<sup>22</sup> Дж. Ху и др., *Squeeze-and-Excitation Networks* (Сети сжатия и возбуждения), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018 г.): с. 7132–7141.



*Рис. 14.21. Блок SE выполняет повторную калибровку карт признаков*

Скажем, блок SE может узнать, что рты, носы и глаза на фотографиях обычно появляются вместе: если вы видите рот и нос, то должны ожидать увидеть также и глаза. Следовательно, если блок замечает сильную активацию в картах признаков рта и носа, но лишь умеренную в карте признаков глаз, он поднимет карту признаков глаз (точнее говоря, он ослабит не относящиеся к делу карты признаков). Если глаза слегка путаются с чем-то другим, тогда такая повторная калибровка карт признаков поможет устраниć неоднозначность.

Блок SE включает в себя всего лишь три слоя: слой объединения по глобальному среднему, плотный скрытый слой, использующий функцию активации ReLU, и плотный выходной слой (рис. 14.22).



*Рис. 14.22. Архитектура блока SE*

Как и ранее, слой объединения по глобальному среднему вычисляет среднюю активацию для каждой карты признаков: например, если его вход содержит 256 карт признаков, то он выдаст 256 чисел, представляющих общий уровень реакции для каждого фильтра. В следующем слое происходит “сжатие”: данный слой имеет значительно меньше 256 нейронов — обычно в 16 раз меньше количества карт признаков (скажем, 16 нейронов) — так что 256 чисел сжимаются в небольшой вектор (например, с 16 измерениями). Это маломерное векторное представление (т.е. вложение) распределения реакций признаков. Такой шаг сужения заставляет блок SE узнат общее представление комбинаций признаков (мы снова увидим данный принцип в действии, когда будем обсуждать автокодировщики в главе 17). Наконец, выходной слой получает вложение и выдает вектор повторной калибровки, содержащий по одному числу на карту признаков (скажем, 256), каждое между 0 и 1. Затем карты признаков умножаются на этот вектор повторной калибровки, так что признаки, не имеющие отношения к делу (с низким показателем повторной калибровки), сокращаются, тогда как релевантные признаки (с показателем повторной калибровки, близким к 1) остаются.

## Реализация сверточной нейронной сети ResNet-34 с использованием Keras

Большинство описанных до сих пор архитектур CNN довольно прямолинейны в реализации (хотя, как выяснится далее, вы обычно будете взамен загружать заранее обученную сеть). Чтобы проиллюстрировать процесс, давайте реализуем сеть ResNet-34 с нуля, используя Keras. Для начала создадим слой ResidualUnit:

```
class ResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="relu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            keras.layers.Conv2D(filters, 3, strides=strides,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization(),
            self.activation,
            keras.layers.Conv2D(filters, 3, strides=1,
                               padding="same", use_bias=False),
            keras.layers.BatchNormalization()]
        self.skip_layers = []
```

```

if strides > 1:
    self.skip_layers = [
        keras.layers.Conv2D(filters, 1, strides=strides,
                            padding="same", use_bias=False),
        keras.layers.BatchNormalization()]

def call(self, inputs):
    Z = inputs
    for layer in self.main_layers:
        Z = layer(Z)
    skip_Z = inputs
    for layer in self.skip_layers:
        skip_Z = layer(skip_Z)
    return self.activation(Z + skip_Z)

```

Как видите, код довольно близко соответствует рис. 14.18. В конструкторе мы создаем все необходимые слои: главными слоями являются те, что указаны справа на диаграмме, а обходящие слои изображены слева (нужны, только если страйд больше 1). Затем в методе `call()` мы заставляем входы проходить через главные слои и обходящие слои (если они есть), после чего складываем оба выхода и применяем функцию активации.

Далее мы можем построить сеть ResNet-34, используя модель `Sequential`, т.к. она действительно представляет собой длинную последовательность слоев (теперь при наличии класса `ResidualUnit` каждый остаточный элемент можно трактовать как одиночный слой):

```

model = keras.models.Sequential()
model.add(keras.layers.Conv2D(64, 7, strides=2,
                           input_shape=[224, 224, 3],
                           padding="same", use_bias=False))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2,
                               padding="same"))

prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))

```

Единственная немного сложная часть в приведенном коде — цикл, который добавляет к модели слои ResidualUnit: как объяснялось ранее, первые 3 остаточных элемента имеют 64 фильтра, следующие 4 остаточных элемента — 128 фильтров и т.д. Затем мы устанавливаем страйд в 1, если количество фильтров является таким же, как в остаточном элементе, а иначе в 2. Далее мы добавляем ResidualUnit и в заключение обновляем prev\_filters.

Удивительно, что с помощью менее 40 строк кода мы в состоянии построить модель, которая победила в решении задачи ILSVRC 2015! Это демонстрирует элегантность модели ResNet и выразительность API-интерфейса Keras. Реализация остальных архитектур CNN не намного сложнее. Тем не менее, в состав Keras входит несколько встроенных архитектур подобного рода, так почему не воспользоваться ими?

## Использование заранее обученных моделей из Keras

В общем случае вам не придется реализовывать стандартные модели вроде GoogLeNet или ResNet вручную, т.к. в пакете keras.applications предлагаются заранее обученные сети, доступные посредством одной строки кода. Например, вы можете загрузить модель ResNet-50, заранее обученную на ImageNet, с помощью следующей строки кода:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

Вот и все! Приведенная строка кода создаст модель ResNet-50 и загрузит веса, заранее выученные на наборе данных ImageNet. Чтобы задействовать ее, сначала необходимо удостовериться в том, что изображения имеют надлежащие размеры. Модель ResNet-50 ожидает изображения  $224 \times 224$  пикселя (другие модели могут ожидать другие размеры, такие как  $299 \times 299$ ), а потому применим функцию `tf.image.resize()` из TensorFlow для изменения размеров загруженных ранее изображений:

```
images_resized = tf.image.resize(images, [224, 224])
```



Функция `tf.image.resize()` не предохраняет соотношение размеров. Если это проблема, тогда перед изменением размеров попробуйте обрезать изображения до соответствующего соотношения. Обе операции могут быть выполнены за раз с помощью функции `tf.image.crop_and_resize()`.

Заранее обученные модели предполагают, что изображения определенным образом предварительно обработаны. В ряде случаев они могут ожидать масштабирования входов от 0 до 1 или от -1 до 1 и т.д. Каждая модель предлагает функцию `preprocess_input()`, которую вы можете вызывать для предварительной обработки изображений. Указанная функция предполагает, что значения пикселей находятся в диапазоне от 0 до 255, поэтому мы обязаны умножить их на 255 (т.к. ранее они были масштабированы до диапазона 0–1):

```
inputs = keras.applications.resnet50.preprocess_input(  
    images_resized * 255)
```

Теперь заранее обученную модель можно использовать для вырабатывания прогнозов:

```
Y_proba = model.predict(inputs)
```

Как обычно, выход `Y_proba` представляет собой матрицу, содержащую по одной строке на изображение и по одному столбцу на класс (в рассматриваемом случае имеется 1 000 классов). Если вы хотите отобразить верхние  $K$  прогнозов, включая имя класса и оценку вероятности каждого спрогнозированного класса, тогда примените функцию `decode_predictions()`. Для каждого изображения она возвращает массив с верхними  $K$  прогнозами, где каждый прогноз выглядит как массив, содержащий идентификатор класса<sup>23</sup>, его имя и соответствующую меру доверия:

```
top_K=keras.applications.resnet50.decode_predictions(Y_proba, top=3)  
for image_index in range(len(images)):  
    print("Image #{}".format(image_index))  
    for class_id, name, y_proba in top_K[image_index]:  
        print("{}-{:12s} {:.2f}%".format(class_id, name, y_proba * 100))  
    print()
```

Вот на что похож вывод:

```
Image #0  
n03877845 - palace      42.87%  
n02825657 - bell_cote   40.57%  
n03781244 - monastery   14.56%
```

<sup>23</sup> Каждое изображение в наборе данных ImageNet ассоциировано со словом в наборе данных WordNet (<https://wordnet.princeton.edu/>): идентификатор класса — это просто идентификатор в WordNet.

### Image #1

n04522168 - vase	46.83%
n07930864 - cup	7.78%
n11939491 - daisy	4.87%

Корректные классы (*monastery* (монастырь) и *daisy* (маргаритка)) присутствуют в верхних трех результатах для обоих изображений. Неплохо, если учесть, что модель должна была выбирать среди 1000 классов.

Как видите, создать хороший классификатор изображений, используя заранее обученную модель, довольно легко. В пакете `keras.applications` доступны и другие модели зрения, включая варианты ResNet, варианты GoogLeNet наподобие Inception-v3 и Xception, варианты VGGNet, а также MobileNet и MobileNetV2 (легковесные модели для применения в мобильных приложениях).

Но что, если вы пожелаете использовать классификатор изображений для классов изображений, которые не являются частью ImageNet? В таком случае вы все равно можете извлечь выгоду из заранее обученных моделей, организовав обучение передачей знаний.

## Использование заранее обученных моделей для обучения передачей знаний

Если вы хотите построить классификатор изображений, но не располагаете достаточным объемом обучающих данных, то часто имеет смысл повторно применить самые нижние слои заранее обученной модели, как обсуждалось в главе 11. Например, давайте обучим модель для классификации фотографий цветков, повторно применяя заранее обученную модель Xception. Первым делом загрузим набор данных с помощью TensorFlow Datasets (см. главу 13):

```
import tensorflow as tfds
dataset, info = tfds.load("tf_flowers", as_supervised=True,
                           with_info=True)
dataset_size = info.splits["train"].num_examples      # 3670
class_names = info.features["label"].names           # ["dandelion",
                                                       # "daisy", ...]
n_classes = info.features["label"].num_classes       # 5
```

Обратите внимание, что обзавестись информацией о наборе данных можно посредством установки `with_info=True`. Здесь мы получаем размер базы данных и имена классов. К сожалению, имеется только обучающий

набор ("train"), но испытательный или проверочный набор отсутствует, а потому нам придется расщепить обучающий набор. Проект TensorFlow Datasets предлагает для этого API-интерфейс. Например, давайте возьмем первые 10% набора данных для испытаний, следующие 15% для проверки и оставшиеся 75% для обучения:

```
test_split, valid_split, train_split =  
    tfds.Split.TRAIN.subsplit([10, 15, 75])  
  
test_set = tfds.load("tf_flowers", split=test_split,  
                     as_supervised=True)  
valid_set = tfds.load("tf_flowers", split=valid_split,  
                     as_supervised=True)  
train_set = tfds.load("tf_flowers", split=train_split,  
                     as_supervised=True)
```

Далее мы должны предварительно обработать изображения. Сеть CNN ожидает изображения  $224 \times 224$ , так что нам нужно изменить их размеры. Нам также необходимо прогнать изображения через метод `preprocess_input()` модели Xception:

```
def preprocess(image, label):  
    resized_image = tf.image.resize(image, [224, 224])  
    final_image =  
        keras.applications.xception.preprocess_input(resized_image)  
    return final_image, label
```

Применим функцию предварительной обработки `preprocess()` ко всем трем наборам, перетасуем обучающий набор и добавим группирование в пакеты и предварительную выборку ко всем наборам данных:

```
batch_size = 32  
train_set = train_set.shuffle(1000)  
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)  
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)  
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

Если вы хотите провести какое-то дополнение данных, тогда измените функцию предварительной обработки для обучающего набора, добавив ряд случайных трансформаций к обучающим изображениям. Скажем, воспользуйтесь функцией `tf.image.random_crop()`, чтобы случайным образом обрезать изображения, функцию `tf.image.random_flip_left_right()`, чтобы случайным образом перевернуть их по горизонтали, и т.д. (пример ищите в разделе "Pretrained Models for Transfer Learning" ("Использование за-

ранее обученных моделей для обучения передачей знаний") тетради Jupiter для настоящей главы).



Класс `keras.preprocessing.image.ImageDataGenerator` облегчает загрузку изображений из диска и их дополнение различными способами: вы можете сдвигать каждое изображение, поворачивать его, изменять его масштаб, переворачивать по горизонтали или вертикали, растягивать или применять к нему любую желаемую функцию трансформации. Упомянутый класс очень удобен в простых проектах. Однако построение конвейера `tf.data` обеспечивает множество преимуществ. Он способен эффективно (например, параллельно) читать изображения из любого источника, а не только с локального диска. Вы можете каким угодно способом манипулировать объектом `Dataset`. Если вы напишете функцию предварительной обработки, основываясь на операциях `tf.image`, то она может использоваться в конвейере `tf.data` и в модели, которая будет развернута в производственной среде (см. главу 19).

Давайте загрузим модель Xception, заранее обученную на наборе данных ImageNet. Мы исключаем верхушку сети, устанавливая `include_top=False`: это изымет слой объединения по глобальному среднему и плотный выходной слой. Затем мы добавляем собственный слой объединения по глобальному среднему, основанный на выходе базовой модели, за которым следует плотный выходной слой, содержащий по одному элементу на класс, с применением многопеременной функции активации. В заключение мы создаем объект `Model` из Keras:

```
base_model = keras.applications.Xception(weights="imagenet",
                                             include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.Model(inputs=base_model.input, outputs=output)
```

Как объяснялось в главе 11, обычно неплохо замораживать веса заранее обученных слоев, по крайней мере, в начале обучения:

```
for layer in base_model.layers:
    layer.trainable = False
```



Поскольку наша модель напрямую использует слои базовой модели, а не сам объект `base_model`, установка `base_model.trainable=False` не оказывает никакого влияния.

Наконец, мы можем скомпилировать модель и начать обучение:

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy",
               optimizer=optimizer,
               metrics=["accuracy"])
history = model.fit(train_set, epochs=5, validation_data=valid_set)
```



В отсутствие графического процессора обучение будет очень медленным. Если вы не располагаете графическим процессором, тогда должны запускать тетрадь Jupiter для этой главы в службе Google Colaboratory с применением исполняющей среды ГП (она бесплатна!). Инструкции ищите по ссылке <https://github.com/ageron/handson-ml2>.

После обучения модели на протяжении нескольких эпох ее правильность при проверке должна составить около 75–80% и прекратить особо улучшаться. Это означает, что верхние слои довольно хорошо обучены, так что мы готовы разморозить все слои (или можно было бы попробовать разморозить только верхние слои) и продолжить обучение (не забывайте компилировать модель в случае замораживания или размораживания слоев). Во избежание повреждения заранее обученных весов мы используем намного меньшую скорость обучения:

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

Процесс займет некоторое время, но в итоге модель должна достичь 95%-ной правильности на испытательном наборе. С ее помощью можно приступить к обучению изумительных классификаторов изображений! Но компьютерное зрение — нечто большее, нежели просто классификация. Скажем, а что, если вы также хотите знать, где цветок находится на фотографии? Давайте теперь посмотрим на это.

# Классификация и установление местонахождения

Как обсуждалось в главе 10, установление местонахождения объекта на фотографии может быть выражено в виде задачи регрессии: распространенный подход к прогнозированию ограничивающей рамки вокруг объекта заключается в том, чтобы прогнозировать горизонтальные и вертикальные координаты центра объекта, а также его высоту и ширину. Таким образом, мы имеем четыре числа для прогнозирования. Это не требует больших изменений в модели; нам всего лишь нужно добавить второй плотный выходной слой с четырьмя элементами (обычно поверх слоя объединения по глобальному среднему), который можно обучить с применением потери MSE:

```
base_model = keras.applications.Xception(weights="imagenet",
                                            include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output =
    keras.layers.Dense(n_classes, activation="softmax")(avg)
loc_output = keras.layers.Dense(4)(avg)
model = keras.Model(inputs=base_model.input,
                     outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2],      # зависит от того,
                                           # что вас больше заботит
              optimizer=optimizer, metrics=["accuracy"])
```

Но теперь мы столкнулись с проблемой: набор данных с изображениями цветков не имеет ограничивающих рамок вокруг цветков. Следовательно, нам придется добавить их самостоятельно. Получение меток часто оказывается одним из самых трудных и дорогостоящих частей проекта МО. Имеет смысл потратить какое-то время на поиск подходящих инструментов. Для аннотирования изображений ограничивающими рамками вы можете использовать инструмент снабжения изображений метками с открытым кодом вроде VGG Image Annotator, LabelImg, OpenLabeler или ImgLab либо коммерческий инструмент наподобие LabelBox или Supervisely. Кроме того, вы можете обдумать применение краудсорсинговых платформ, таких как Amazon Mechanical Turk, если изображений, подлежащих аннотированию, очень много. Тем не менее, потребуется проделать немалую работу, чтобы настроить краудсорсинговую платформу, подготовить форму для отправки работникам, контролировать их и удостоверяться в том, что качество производимых ими ограничивающих рамок приемлемо, поэтому хорошо подумайте, стоит

ли того дела. Если речь идет о нескольких тысячах изображений, которые нужно снабдить метками, и делать это планируется нечасто, тогда возможно предпочтительнее выполнить работу самостоятельно. Адриана Ковашка и др. написали весьма практическую статью (<https://homl.info/crowd>)<sup>24</sup> о краудсорсинге в компьютерном зрении. Я рекомендую вам ознакомиться с ней, даже если вы не собираетесь прибегать к краудсорсингу.

Давайте предположим, что вы получили ограничивающие рамки для всех изображений цветков в наборе данных (пока мы допускаем, что есть по одной ограничивающей рамке на изображение). Далее понадобится создать набор данных, элементы которого будут пакетами предварительно обработанных изображений наряду с метками классов и их ограничивающими рамками. Каждый элемент должен быть кортежем формы (`images`, `(class_labels, bounding_boxes)`). После этого все готово для обучения модели!



Ограничивающие рамки должны быть нормализованы, чтобы горизонтальные и вертикальные координаты, а также высота и ширина находились в диапазоне от 0 до 1. Кроме того, обычно прогнозируют квадратный корень высоты и ширины, а не высоту и ширину непосредственно: в итоге 10-пиксельная ошибка для крупной ограничивающей рамки не будет штрафоваться так же сильно, как 10-пиксельная ошибка для небольшой ограничивающей рамки.

Потеря MSE часто неплохо работает в качестве функции издержек при обучении модели, но она не является выдающимся показателем для оценки, насколько хорошо модель способна прогнозировать ограничивающие рамки. Самым распространенным показателем для такой цели служит *пересечение по объединению* (*Intersection over Union — IoU*): площадь перекрытия между спрогнозированной ограничивающей рамкой и целевой ограничивающей рамкой, деленная на площадь их объединения (рис. 14.23). В `tf.keras` такой показатель реализуется классом `tf.keras.metrics.MeanIoU`.

Классификация и установление местонахождения одиночного объекта — это замечательно, но что, если изображения содержат много объектов (как часто бывает в наборе данных с изображениями цветков)?

<sup>24</sup> Адриана Ковашка и др., *Crowdsourcing in Computer Vision* (Краудсорсинг в компьютерном зрении), *Foundations and Trends in Computer Graphics and Vision* 10, номер 3 (2014 г.): с. 177–243.

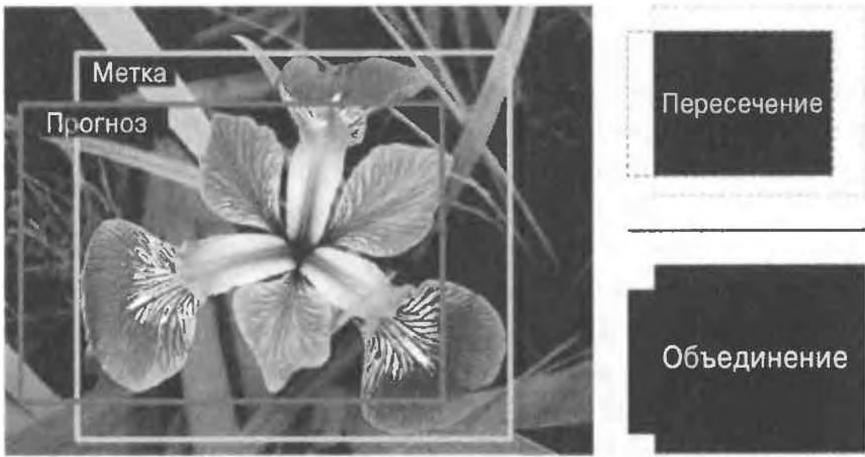


Рис. 14.23. Показатель пересечения по объединению (IoU)  
для ограничивающих рамок

## Выявление объектов

Задача классификации и установления местонахождения множества объектов в изображении называется *выявлением объектов (object detection)*. Еще несколько лет назад обычный подход предусматривал привлечение сети CNN, обученной классификации и установлению местонахождения одиночного объекта, и ее плавное продвижение по изображению, как показано на рис. 14.24.

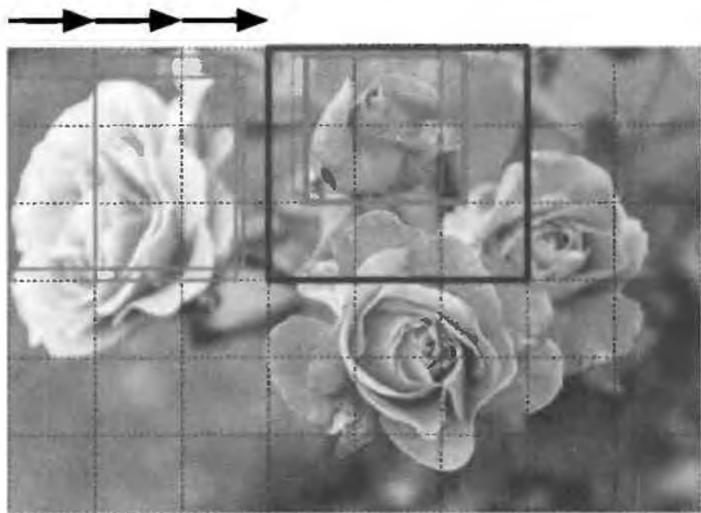


Рис. 14.24. Выявление множества объектов путем плавного  
продвижения сети CNN по изображению

В этом примере изображение было разрезано в виде решетки  $6 \times 8$  и видно, что сеть CNN (прямоугольник, представленный толстой линией черного цвета) проходит по всем областям  $3 \times 3$ . Когда сеть CNN просматривала левый верхний угол изображения, она обнаружила крайнюю слева розу, после чего обнаружила ту же розу снова, как только в первый раз сместилась на один раз вправо. На следующем шаге сеть CNN начала выявлять часть самой верхней розы и затем обнаружила ее еще раз, сдвинувшись на очередной шаг вправо. Далее продвижение сети CNN продолжается по всему изображению с просмотром всех областей  $3 \times 3$ . Более того, поскольку объекты могут иметь варьирующиеся размеры, вы плавно продвигали бы сеть CNN также по областям разных размеров. Скажем, закончив с областями  $3 \times 3$ , вы могли бы двигать сеть CNN и по областям  $4 \times 4$ .

Методика довольно-таки прямолинейна, но как было показано, она будет выявлять тот же самый объект много раз в слегка отличающихся позициях. Затем потребуется заключительная обработка, чтобы избавиться от излишних ограничивающих рамок. Часто используемый подход называется *подавлением немаксимумов* (*non-max suppression*). Вот в чем он заключается.

1. Добавьте к сети CNN дополнительный выход *объектности* (*objectness*) для оценки вероятности того, что цветок действительно присутствует на изображении (в качестве альтернативы вы могли бы добавить класс “цветков нет”, но обычно он работает не настолько хорошо). Он должен применять сигмоидальную функцию активации, а обучать его можно с использованием двоичной функции *потери перекрестной энтропии* (*cross-entropy loss*). Затем избавьтесь от всех ограничивающих рамок, для которых мера объектности ниже определенного порога: это отбросит все ограничивающие рамки, на самом деле не содержащие цветок.
2. Найдите ограничивающую рамку с наивысшей мерой объектности и избавьтесь от всех остальных ограничивающих рамок, которые значительно перекрываются с ней (скажем, с IoU более 60%). Например, на рис. 14.24 ограничивающей рамкой с максимальной мерой объектности является та, что представлена толстой линией и охватывает самую верхнюю розу (мера объектности задается толщиной линии ограничивающих рамок). Другая ограничивающая рамка над той же розой значительно перекрывает с максимальной ограничивающей рамкой, так что от нее следует избавиться.
3. Повторяйте шаг 2 до тех пор, пока больше не останется ограничивающих рамок, от которых нужно избавиться.

Такой простой подход к выявлению объектов работает неплохо, но требует многократного запуска сети CNN, а потому он довольно медленный. К счастью, существует гораздо более быстрый способ плавного продвижения сети CNN по изображению: применение *полностью сверточной сети* (*fully convolutional network* — FCN).

## Полностью сверточные сети

Идея сетей FCN впервые была представлена в статье 2015 года (<https://homl.info/fcn>)<sup>25</sup>, написанной Джонатаном Лонгом и др., для семантической сегментации (задачи классификации каждого пикселя внутри изображения в соответствии с классом объекта, которому он принадлежит). Авторы указали на возможность замены плотных слоев в верхней части сети CNN сверточными слоями. Чтобы уловить суть, давайте рассмотрим пример: допустим, что плотный слой с 200 нейронами располагается поверх сверточного слоя, который выдает 100 карт признаков, каждая размером  $7 \times 7$  (это размер карты признаков, не размер ядра). Каждый нейрон будет вычислять взвешенную сумму всех  $100 \times 7 \times 7$  активаций из сверточного слоя (плюс член смещения). Теперь выясним, что произойдет, если мы заменим плотный слой сверточным слоем, использующим 200 фильтров, каждый размером  $7 \times 7$ , и дополнение "valid". Такой слой будет выдавать 200 карт признаков, каждая размером  $1 \times 1$  (поскольку ядро точно соответствует размеру входных карт признаков и применяется дополнение "valid"). Другими словами, он будет выдавать 200 чисел, как делал плотный слой; и если внимательно взглянуть на вычисления, выполняемые сверточным слоем, то несложно заметить, что числа будут совершенно такими же, как числа, выпускаемые плотным слоем. Единственное отличие в том, что выходом плотного слоя был тензор формы [размер пакета, 200], тогда как сверточный слой будет выдавать тензор формы [размер пакета, 1, 1, 200].



Чтобы преобразовать плотный слой в сверточный, количество фильтров в сверточном слое должно быть равно количеству элементов в плотном слое, размер фильтров обязан быть равным размеру входных карт признаков и должно использоваться дополнение "valid". Как вскоре будет показано, страйд может быть установлен в 1 и более.

<sup>25</sup> Джонатан Лонг и др., *Fully Convolutional Networks for Semantic Segmentation* (Полностью сверточные сети для семантической сегментации), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2015 г.): с. 3431–3440.

Почему это важно? В то время как плотный слой ожидает определенного размера входа (потому что имеет один вес на входной признак), сверточный слой будет охотно обрабатывать изображения любого размера<sup>26</sup> (однако, он все же ожидает, что его входы имеют конкретное количество каналов, т.к. каждое ядро содержит разный набор весов для каждого входного канала). Поскольку сеть FCN содержит только сверточные слои (и объединяющие слои с такими же характеристиками), ее можно обучать и запускать на изображениях любого размера!

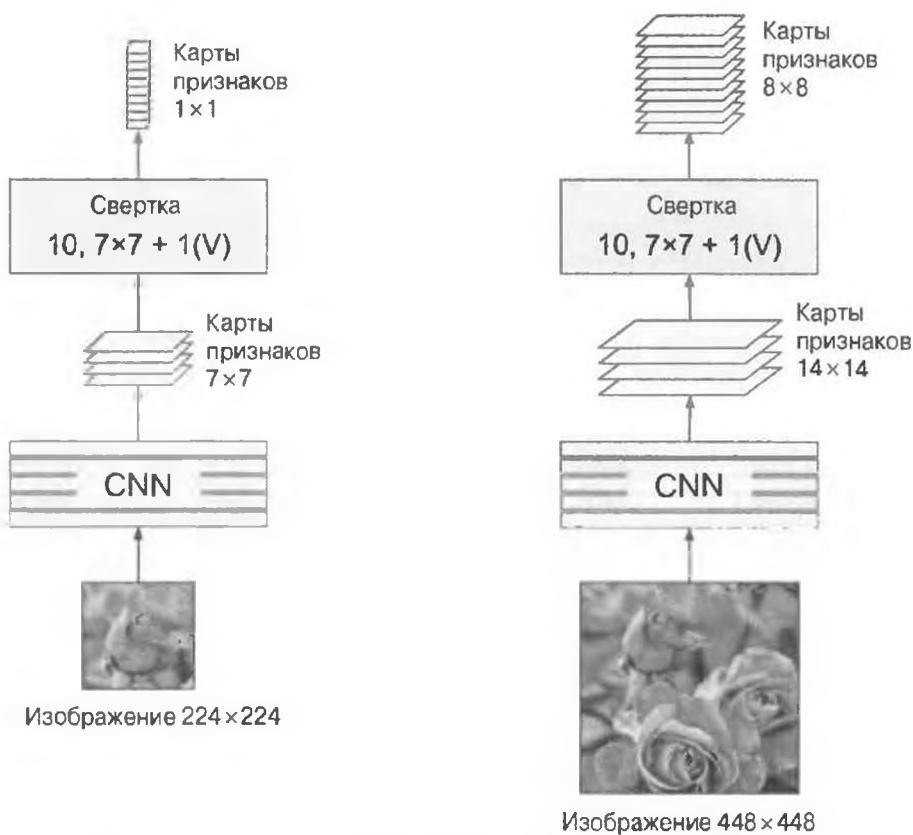
Например, пусть мы уже обучили сеть CNN для классификации и установления местонахождения цветков. Она обучалась на изображениях  $224 \times 224$  и выдавала 10 чисел: выходы 0–4 пропускались через многопараметрическую функцию активации, что давало вероятности классов (по одной на класс); выход 5 пропускался через логистическую функцию активации, давая меру объектности; выходы 6–9 не применяли никакой функции активации и представляли координаты центра ограничивающей рамки, а также ее высоту и ширину. Теперь мы можем преобразовать ее плотные слои в сверточные. На самом деле нам даже не придется обучать их заново; мы можем просто скопировать веса из плотных слоев в сверточные слои! В качестве альтернативы перед обучением мы могли бы преобразовать сеть CNN в сеть FCN.

Далее предположим, что последний сверточный слой перед выходным слоем (также называемый суживающим слоем) выдает карты признаков  $7 \times 7$ , когда сети подается изображение  $224 \times 224$  (как показано в левой части рис. 14.25). Если мы подадим сети FCN изображение  $448 \times 448$  (см. правую часть рис. 14.25), то суживающий слой выдаст карты признаков  $14 \times 14$ .<sup>27</sup> Так как плотный выходной слой был заменен сверточным слоем, использующим 10 фильтров размера  $7 \times 7$  с дополнением "valid" и страйдом 1, выход будет состоять из 10 карт признаков, каждая размером  $8 \times 8$  (потому что  $14 - 7 + 1 = 8$ ). Другими словами, сеть FCN будет обрабатывать все изображение только один раз и выдаст решетку  $8 \times 8$ , где каждая ячейка содержит 10 чисел (5 ве-

<sup>26</sup> Есть одно небольшое исключение: сверточный слой, использующий дополнение "valid", будет выражать недовольство, если размер входа меньше размера ядра.

<sup>27</sup> Здесь предполагается, что мы используем в сети только дополнение "same": на самом деле дополнение "valid" сократило бы размер карт признаков. Кроме того, 448 можно аккуратно поделить на 2 несколько раз, пока не будет достигнуто 7, без каких-либо ошибок округления. Если любой слой применяет страйд, отличающийся от 1 или 2, тогда может возникать ошибка округления, так что карты признаков снова могут иметь меньший размер.

роятностей классов, 1 меру объективности и 4 координаты ограничивающей рамки). Это все равно, что взять исходную сеть CNN и плавно перемещать ее по изображению, применяя 8 шагов на строку и 8 шагов на столбец. Чтобы представить себе такое действие, вообразите рассечение исходного изображения согласно решетке  $14 \times 14$  и затем продвижение окна  $7 \times 7$  по указанной решетке; для окна будет  $8 \times 8 = 64$  возможных местоположений, отсюда  $8 \times 8$  прогнозов. Тем не менее, подход с сетью FCN намного более эффективен, поскольку сеть просматривает изображение только один раз. В действительности YOLO (*You Only Look Once* — вы просматриваете только раз) представляет собой название очень популярной архитектуры выявления объектов, которую мы рассмотрим следующей.



*Рис. 14.25. Та же самая полностью сверточная сеть, обрабатывающая небольшое (слева) и крупное (справа) изображение*

## Вы просматриваете только раз (YOLO)

YOLO — это исключительно быстрая и точная архитектура выявления объектов, предложенная Джозефом Редмоном и др. в статье 2015 года (<https://homl.info/yolo>)<sup>28</sup>, которая впоследствии была усовершенствована в статьях 2016 года (<https://homl.info/yolo2>)<sup>29</sup> (YOLOv2) и 2018 года (<https://homl.info/yolo3>)<sup>30</sup> (YOLOv3). Она настолько быстрая, что может работать в реальном времени с видеороликами, как видно в демонстрации Редмона (<https://homl.info/yolodemo>).

Архитектура YOLOv3 довольно похожа на только что обсужденную архитектуру, но обладает несколькими важными различиями.

- Она выдает для каждой ячейки решетки пять ограничивающих рамок (вместо только одной), и каждая ограничивающая рамка поступает с мерой объектности. Она также выдает для каждой ячейки решетки 20 вероятностей классов, т.к. обучалась на наборе данных PASCAL VOC, который содержит 20 классов. В сумме получается 45 чисел на ячейку решетки: 5 ограничивающих рамок, каждая с 4 координатами, 5 мер объектности и 20 вероятностей классов.
- Вместо прогнозирования абсолютных координат центров ограничивающих рамок архитектура YOLOv3 прогнозирует смещение относительно координат ячейки решетки, где (0, 0) означает левый верхний угол ячейки, а (1, 1) — правый нижний угол ячейки. Для каждой ячейки решетки YOLOv3 обучается прогнозированию только ограничивающих рамок, чьи центры находятся в этой ячейке (но сама ограничивающая рамка в общем случае может простираться за пределы ячейки решетки). YOLOv3 применяет к координатам ограничивающих рамок логистическую функцию активации, гарантируя то, что они остаются в диапазоне 0–1.

<sup>28</sup> Джозеф Редмон и др., *You Only Look Once: Unified, Real-Time Object Detection* (Вы просматриваете только раз: унифицированное выявление объектов в реальном времени), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2016 г.): с. 779–788.

<sup>29</sup> Джозеф Редмон и Али Фархади, *YOLO9000: Better, Faster, Stronger* (YOLO9000: лучше, быстрее, стабильнее), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2017 г.): с. 6517–6525.

<sup>30</sup> Джозеф Редмон и Али Фархади, *YOLOv3: An Incremental Improvement* (YOLOv3: постепенное улучшение), препринт arXiv:1804.02767 (2018 г.).

- Перед обучением нейронной сети YOLOv3 ищет пять репрезентативных измерений ограничивающих рамок, которые называются *опорными рамками* (*anchor box*) или *бывшими ограничивающими рамками* (*bounding box prior*). Это делается путем применения алгоритма K-Means (см. главу 9) к высоте и ширине ограничивающих рамок обучающего набора. Например, если обучающие изображения содержат много пешеходов, тогда одна из опорных рамок, вероятно, будет иметь измерения типичного пешехода. Когда нейронная сеть впоследствии прогнозирует пять ограничивающих рамок на ячейку решетки, фактически она вырабатывает прогноз того, насколько изменять масштаб каждой опорной рамки. Например, пусть одна опорная рамка имеет 100 пикселей в высоту и 50 пикселей в ширину, а сеть прогнозирует, скажем, коэффициент масштабирования по вертикали 1.5 и коэффициент масштабирования по горизонтали 0.9 (для одной из ячеек решетки). Результатом будет спрогнозированная ограничивающая рамка размером  $150 \times 45$  пикселей. Говоря точнее, для каждой ячейки решетки и каждой опорной рамки сеть прогнозирует логарифм коэффициентов масштабирования по вертикали и горизонтали. Наличие бывших ограничивающих рамок увеличивает вероятность того, что сеть спрогнозирует ограничивающие рамки с подходящими измерениями, и также ускорит обучение, поскольку она быстрее выяснит, как выглядят приемлемые ограничивающие рамки.
- Сеть обучается с использованием изображений разных масштабов: для каждого нескольких пакетов во время обучения сеть случайным образом выбирает новое измерение изображения (от  $330 \times 330$  до  $608 \times 608$  пикселей). Это позволяет сети учиться выявлять объекты при разных масштабах. Вдобавок появляется возможность применять YOLOv3 при разных масштабах: наименьший масштаб будет менее точен, но быстрее более крупного масштаба, так что вы можете установить надлежащий компромисс в своем сценарии использования.

Есть еще несколько интересных нововведений, такие как применение обходящих связей с целью восстановления части пространственного разрешения, которая утрачивается в сети CNN (мы обсудим это очень скоро, когда начнем рассматривать семантическую сегментацию). В своей статье 2017 года авторы ввели модель YOLO9000, которая использует иерархическую классификацию: модель прогнозирует вероятность для каждого узла в зрительной иерархии, называемой *WordTree*. В итоге сеть получает возможность с высо-

кой степенью уверенности прогнозировать, что изображение представляет, скажем, собаку, даже если неясно, какой именно породы. Я призываю вас двигаться дальше и ознакомиться со всеми тремя статьями: их довольно приятно читать и они предлагают великолепные примеры того, как можно последовательно совершенствовать системы глубокого обучения.

## Усредненная средняя точность (mAP)

Очень распространенной метрикой, применяемой в задачах выявления объектов, является *усредненная средняя точность* (*mean Average Precision* — *mAP*). “Усредненная средняя” звучит несколько избыточно, не так ли? Для лучшего понимания этой метрики давайте возвратимся к двум метрикам классификации, которые мы обсуждали в главе 3: точность и полнота. Вспомните соотношение: чем выше полнота, тем ниже точность. Его можно визуализировать в виде кривой точность/полнота (см. рис. 3.5). Чтобы свести такую кривую в единственное число, мы могли бы вычислить площадь под кривой (AUC). Но обратите внимание, что кривая точность/полнота может содержать несколько разделов, где точность на самом деле повышается, когда полнота растет, особенно при низких величинах полноты (взгляните на левый верхний угол рис. 3.5). Это одна из движущих сил метрики *mAP*.

Предположим, что классификатор имеет 90%-ную точность при полноте 10%, но 96%-ную точность при полноте 20%. В действительности здесь нет никакого компромисса: просто разумнее использовать классификатор при полноте 20%, нежели 10%, т.к. мы получим и высокую полноту, и высокую точность. Таким образом, вместо того, чтобы смотреть на точность при 10% полноте, на самом деле мы должны смотреть на *максимальную* точность, которую классификатор способен предложить с полнотой, *по крайней мере*, 10%. Она была бы 96%, не 90%. Следовательно, один из способов получить четкое представление об эффективности модели предусматривает расчет максимальной точности, которой можно добиться с полнотой, по крайней мере, 0%, затем 10%, 20% и т.д. вплоть до 100%, после чего вычислить среднюю величину этих максимальных точностей. Она называется метрикой *средней точности* (*Average Precision* — *AP*). Теперь, когда есть более двух классов, мы можем рассчитать *AP* для каждого класса и затем вычислить среднюю величину *AP* (*mAP*). Вот так!

В системе выявления объектов имеется дополнительный уровень сложности: что, если система обнаруживает корректный класс, но в неправильном местоположении (т.е. ограничивающая рамка находится совершенно не там)? Конечно, мы не должны учитывать это как положительный прогноз. Один из подходов заключается в том, чтобы определить порог IoU: например, мы можем полагать, что прогноз корректен, только если больше, скажем, 0,5, и спрогнозированный класс является правильным. Соответствующая метрика mAP обычно обозначается как mAP@0.5 (либо mAP@50% или временами просто AP<sub>50</sub>). В некоторых состязаниях (вроде задачи PASCAL VOC) так и поступили. В других (таких как задача COCO) величина mAP вычисляется для разных порогов IOU (0,50, 0,55, 0,60, ..., 0,95), а финальной метрикой будет среднее из всех рассчитанных величин mAP (обозначается в виде AP@[.50:.95] или AP@[.50:0.05:.95]). Да, это среднее усредненное среднее.

На GitHub доступно несколько реализация YOLO, построенных с применением TensorFlow. В частности, просмотрите реализацию с использованием TensorFlow 2 от Зихао Чжана (<https://homl.info/yolotf2>). В рамках проекта TensorFlow Models предлагаются другие модели выявления объектов, многие с заранее обученными весами; и некоторые даже были перенесены в TF Hub, например, довольно популярные SSD (<https://homl.info/ssd>)<sup>31</sup> и Faster-RCNN (<https://homl.info/fasterrcnn>)<sup>32</sup>. SSD также является моделью “однократного” обнаружения, похожей на YOLO. Модель Faster R-CNN сложнее: изображение сначала проходит через сеть CNN, после чего выход передается в сеть для предложения областей (*Region Proposal Network — RPN*), которая предлагает ограничивающие рамки, наиболее вероятно содержащие объект, и для каждой ограничивающей рамки запускается классификатор на основе обрезанного выхода сети CNN.

<sup>31</sup> Вей Лю и др., *SSD: Single Shot Multibox Detector* (SSD: однократный многорамочный детектор), *Proceedings of the 14th European Conference on Computer Vision 1* (2016 г.): с. 21–37.

<sup>32</sup> Шаоцин Рен и др., *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks* (Faster R-CNN: в направлении выявления объектов в реальном времени с помощью сетей для предложения областей), *Proceedings of the 28th International Conference on Neural Information Processing Systems 1* (2015 г.): с. 91–99.

Выбор системы выявления зависит от многих факторов: скорости, правильности, доступных заранее обученных моделей, времени обучения, сложности и т.д. В статьях приводятся таблицы метрик, но тестовые среды характеризуются большой переменчивостью, а технологии развиваются настолько быстро, что трудно провести справедливое сравнение, которое было бы полезным для большинства людей в течение хотя бы нескольких месяцев.

Итак, мы можем устанавливать местонахождение объектов, вычерчивая вокруг них ограничивающие рамки. Замечательно! Но возможно вы стремитесь к чуть большей точности. Давайте посмотрим, как опуститься ниже до уровня пикселей.

## Семантическая сегментация

При семантической сегментации (*semantic segmentation*) каждый пиксель классифицируется в соответствии с классом объекта, которому он принадлежит (например, дорога, автомобиль, пешеход, здание и т.п.), как показано на рис. 14.26. Обратите внимание, что между разными объектами того же самого класса различие не проводится. Скажем, все велосипеды в правой части сегментированного изображения становятся большой совокупностью пикселей. Главная трудность в этой задаче связана с тем, что когда изображения прогоняются через обычновенную сеть CNN, они постепенно теряют свое пространственное разрешение (из-за слоев со страйдами выше 1). Таким образом, обыкновенная сеть CNN может в итоге узнать, что где-то слева внизу изображения присутствует человек, но не более того.

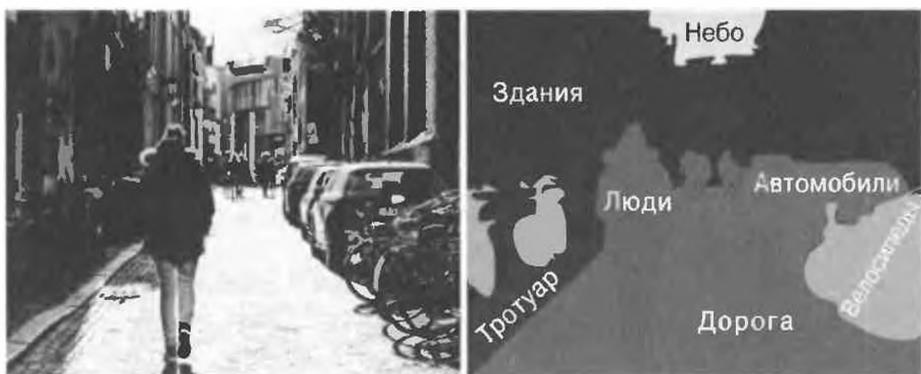


Рис. 14.26. Семантическая сегментация в действии

Как и для выявления объектов, существует много разных подходов к решению такой задачи, среди которых есть довольно сложные. Однако в обсуждаемой ранее статье Джонатана Лонга и др., опубликованной в 2015 году, было предложено относительно простое решение. Авторы начинают с того, что получают заранее обученную сеть CNN и превращают ее в сеть FCN. Сеть CNN применяет к входному изображению суммарный страйд 32 (т.е. если сложить все страйды выше 1) в том смысле, что последний слой выдает карты признаков, которые в 32 раза меньше входного изображения. Это явно слишком грубо, так что они добавляют одиночный слой повышения дискретизации (*upsampling layer*), умножающий разрешение на 32.

Существует несколько решений для повышения дискретизации (увеличения размера изображения), такие как билинейная интерполяция (*bilinear interpolation*), но прием приемлемо работает только до увеличения  $\times 4$  или  $\times 8$ . Взамен авторы использовали транспонированный сверточный слой (*transposed convolutional layer*)<sup>33</sup>: он эквивалентен растягиванию изображения путем вставки пустых строк и столбцов (заполненных нулями) и выполнению обычной свертки (рис. 14.27).

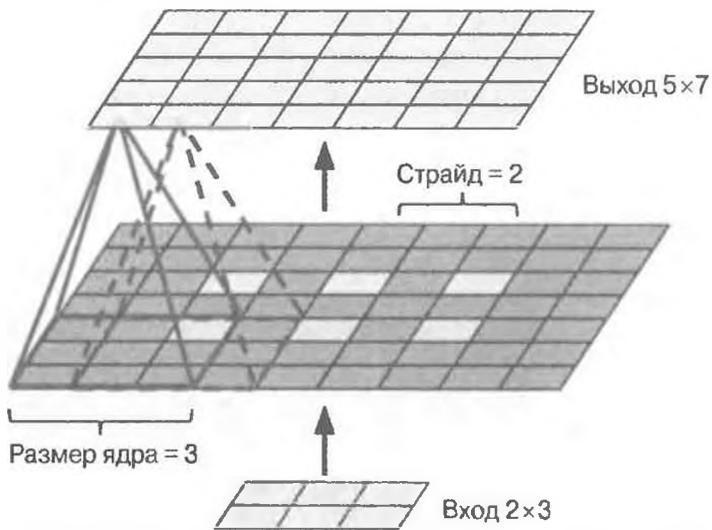


Рис. 14.27. Повышение дискретизации с использованием транспонированного сверточного слоя

<sup>33</sup> На такой тип слоя иногда ссылаются как на слой обращения свертки (*deconvolution layer*), но он как раз не делает то, что математики называют обращением свертки, поэтому упомянутого названия следует избегать.

В качестве альтернативы некоторые предпочитают думать о нем, как об обычном сверточном слое с дробными страйдами (например,  $1/2$  на рис. 14.27). Транспонированный сверточный слой можно инициализировать для выполнения чего-то близкого к линейной интерполяции, но поскольку слой является обучаемым, то во время обучения он научится работать лучше. В `tf.keras` можно применять слой `Conv2DTranspose`.



В транспонированном сверточном слое страйд определяет то, насколько вход будет растягиваться, а не размер шагов фильтра, и потому чем больше страйд, тем крупнее выход (в отличие от сверточных или объединяющих слоев).

## Операции свертки TensorFlow

Библиотека TensorFlow предлагает также и другие виды сверточных слоев. `keras.layers.Conv1D`. Создает сверточный слой для одномерных входов, таких как временной ряд или текст (последовательность букв или слов); мы рассмотрим его в главе 15.

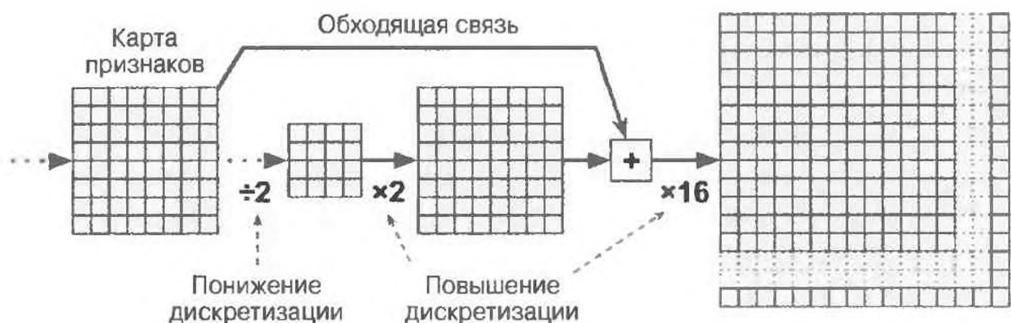
`keras.layers.Conv3D`. Создает сверточный слой для трехмерных входов, таких как трехмерные изображения в позитронно-эмиссионной томографии.

`dilation_rate`. Установка гиперпараметра `dilation_rate` любого сверточного слоя в значение 2 и выше приводит к созданию сверточного слоя “*à trous*” (“à trous” (фр.) — “с отверстиями”). Это эквивалентно применению обычновенного сверточного слоя с фильтром, расширенным за счет вставки строк и столбцов нулей (т.е. отверстий). Например, фильтр  $1 \times 3$ , 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 дополнительных вычислительных затрат и добавочных параметров.

`tf.nn.depthwise_conv2d()`. Может использоваться для создания слоя свертки по глубине (но вам придется создавать переменные самостоятельно). Он независимым образом 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.

Решение правильное, но все еще слишком неточное. Чтобы улучшить его, авторы добавили обходящие связи из нижних слоев: например, они повышают дискретизацию в 2 раза (вместо 32) и добавляют выход нижнего слоя, который имеет такое удвоенное разрешение. Затем авторы повышают дискретизацию результата в 16 раз, что приводит к общему коэффициенту повышения дискретизации 32 (рис. 14.28). Это восстанавливает часть пространственного разрешения, утраченную в более ранних объединяющих слоях. В наилучшей архитектуре они использовали вторую похожую обходящую связь для восстановления даже более мелких деталей из слоев, расположенных еще ниже.

Выражаясь кратко, выход исходной сети CNN пропускается через следующие дополнительные шаги: увеличение масштаба  $\times 2$ , добавление выхода более низкого слоя (подходящего масштаба), увеличение масштаба  $\times 2$ , добавление выхода еще более низкого слоя и заключительное увеличение масштаба  $\times 8$ . Увеличивать масштаб допускается даже выше размера исходного изображения, что позволяет повысить разрешение изображения; такой прием называется *супер-разрешением* (*super-resolution*).



*Рис. 14.28. Пропуск слоев восстанавливает часть пространственного разрешения из нижних слоев*

Многочисленные хранилища GitHub предоставляют реализации семантической сегментации с применением TensorFlow (пока что TensorFlow 1), а в проекте TensorFlow Models можно даже найти заранее обученные модели для сегментации экземпляров (*instance segmentation*). Сегментация экземпляров похожа на семантическую сегментацию, но вместо объединения всех объектов одного класса в крупную совокупность каждый объект отделяется от остальных (например, она распознает каждый отдельный велосипед).

В настоящее время модели для сегментации экземпляров, доступные в проекте TensorFlow Models, основаны на архитектуре *Mask R-CNN* (маскирующая R-CNN), которая была предложена в статье 2017 года (<https://homl.info/maskrcnn>)<sup>34</sup>: она расширяет модель Faster R-CNN за счет дополнительного вырабатывания пиксельной маски для каждой ограничивающей рамки. Таким образом, вы получаете не только ограничивающую рамку вокруг каждого объекта с набором оценок вероятностей классов, но также пиксельную маску, определяющую местоположение пикселей в ограничивающей рамке, которые принадлежат объекту.

Как видите, область глубокого компьютерного зрения громадна и быстро развивается, с ежегодным появлением самых разнообразных архитектур, основанных на сверточных нейронных сетях. Прогресс, достигнутый всего за несколько лет, был поразительным, и теперь исследователи сосредоточены на все более и более трудных задачах, таких как *состязательное обучение* (*adversarial learning*), которое пытается сделать сеть более устойчивым к изображениям, предназначенным для ее обмана, возможность объяснения (понимание, по какой причине сеть выработала конкретную классификацию), реалистичная генерация изображений (мы вернемся к ней в главе 17) и *однократное обучение* (*single-shot learning*), т.е. система, которая способна распознать объект, увидев его только один раз. Некоторые даже исследуют совершенно новые архитектуры вроде *капсулых сетей* (*capsule network*) Джейфри Хинтона (<https://homl.info/capsnet>)<sup>35</sup> (я представил их в паре видеороликов (<https://homl.info/capsnetvideos>) и соответствующего кода в тетради *Jupyter*).

Далее мы переходим к следующей главе, где выясним, как обрабатывать последовательные данные, такие как временные ряды, с использованием рекуррентных и сверточных нейронных сетей.

<sup>34</sup> Кайминг Хе и др., *Mask R-CNN* (Маскирующая R-CNN), препринт arXiv:1703.06870 (2017 г.).

<sup>35</sup> Джейфри Хинтон и др., *Matrix Capsules with EM Routing* (Матричные капсулы с маршрутизацией по алгоритму максимизации ожидания), *Proceedings of the International Conference on Learning Representations* (2018 г.).

# 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  $3 \times 3$ , 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  $200 \times 300$  пикселей.

Сколько всего будет параметров в сети CNN? Если используются 32-битные 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 минипакете 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, ResNet, SENet и Xception?
7. Что такое полностью сверточная сеть? Как можно преобразовать плотный слой в сверточный слой?
8. В чем заключается главная техническая трудность семантической сегментации?
9. Постройте собственную сеть CNN с нуля и попробуйте достичь самой высокой возможной правильности на наборе MNIST.
10. Воспользуйтесь обучением передачей знаний для классификации крупных изображений, выполнив описанные ниже шаги.
  - а) Создайте обучающий набор, содержащий минимум 100 изображений на класс. Например, вы могли бы классифицировать соб-

ственные фотографии на основе места съемки (пляж, горы, город и т.д.) либо применить существующий набор данных (скажем, из TensorFlow Datasets).

- 6) Расщепите его на обучающий, проверочный и испытательный наборы.
  - в) Постройте входной конвейер, который включает надлежащие операции предварительной обработки, и факультативно добавьте дополнение данных.
  - г) Отрегулируйте заранее обученную модель на имеющемся наборе данных.
11. Просмотрите руководство по передаче стилей TensorFlow (<https://homl.info/styletuto>). Это забавный способ генерации графических материалов с использованием глубокого обучения.

Решения приведенных упражнений доступны в приложении А.



# Обработка последовательностей с использованием рекуррентных и сверточных нейронных сетей

Бэттер отбивает мяч. Аутфилдер немедленно начинает бежать, предугадывая траекторию мяча. Он следит за ним, подстраивает свое движение и, наконец, ловит его (под гром аплодисментов). Прогнозирование будущего — вот то, что вы делаете постоянно, заканчивая фразу друга или предвкушая 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 нужно покупать или продавать. В системах автопилотов они способны предугадывать траектории автомобилей и препятствовать авариям. В более общем плане сети RNN могут работать с *последовательностями* (*sequences*) произвольной длины, а не с входными данными фиксированного размера, как все сети, рассмотренные до сих пор. Например, они в состоянии принимать на входе предложения, документы или аудио-образцы, что делает их чрезвычайно полезными для приложений обработки естественного языка (NLP) вроде автоматического перевода или преобразования речи в текст.

В настоящей главе мы сначала взглянем на фундаментальные концепции, лежащие в основе сетей RNN, и посмотрим, как их обучать с применением обратного распространения во времени, после чего будем использовать сети RNN для прогнозирования временного ряда. Затем мы займемся исследованием основных двух трудностей, с которыми сталкиваются сети:

- нестабильные градиенты (обсуждались в главе 11), которых можно избежать за счет применения разнообразных методик, в том числе рекуррентного отключения и нормализации рекуррентных слоев;

- (очень) ограниченная кратковременная память, которую можно расширить с использованием ячеек LSTM и GRU.

Сети RNN — не единственный тип нейронных сетей, способных обрабатывать последовательные данные: с небольшими последовательностями может справиться обыкновенная плотная сеть; в случае очень длинных последовательностей, таких как аудио-образцы или текст, работу неплохо выполняют сверточные нейронные сети. Мы обсудим обе возможности и в заключение главы реализуем *WaveNet* — архитектуру сетей CNN, которая в состоянии обрабатывать последовательности из десятков тысяч временных шагов (*time step*). В главе 16 мы продолжим исследование сетей RNN и посмотрим, как их применять для обработки естественного языка, а также ознакомимся с недавно появившимися архитектурами, основанными на механизмах внимания. Итак, приступим!

## Рекуррентные нейроны и слои

До сих пор мы были сосредоточены на нейронных сетях прямого распространения, где активации протекали только в одном направлении, от входного слоя до выходного слоя (за исключением нескольких сетей, обсуждаемых в приложении D). Рекуррентная нейронная сеть выглядит очень похожей на нейронную сеть прямого распространения, но также имеет связи, указывающие в обратном направлении. Давайте взглянем на простейшую сеть RNN, состоящую из одного нейрона, который получает входы, производит выход и передает этот выход обратно самому себе (рис. 15.1 (слева)). На каждом временном шаге  $t$  (также называется *фреймом (frame)*) такой *рекуррентный нейрон (recurrent neuron)* получает входы  $x_{(t)}$ , а также собственный выход из предыдущего временного шага,  $y_{(t-1)}$ . Поскольку на первом временном шаге предыдущего выхода не существует, в общем случае он устанавливается в 0. Мы можем представить такую крошечную сеть по оси времени, как показано на рис. 15.1 (справа). Процесс называется *развертыванием сети во времени* (это тот же самый рекуррентный нейрон, представленный по одному разу на временной шаг).

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  $\mathbf{y}_{(t-1)}$  (рис. 15.2). Как видите, теперь и входы, и выходы являются векторами (когда существовал только один нейрон, выход был скаляром).

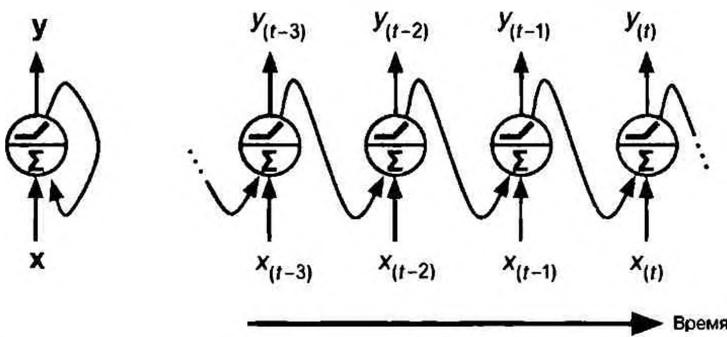


Рис. 15.1. Рекуррентный нейрон (слева), развернутый во времени (справа)

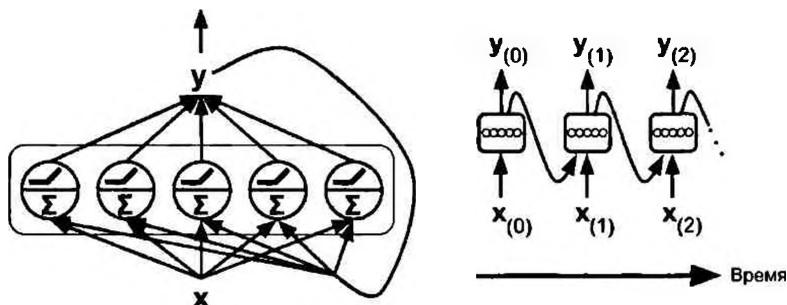


Рис. 15.2. Слой рекуррентных нейронов (слева), развернутый во времени (справа)

Каждый рекуррентный нейрон имеет два набора весов: один для входов  $x_{(t)}$  и один для выходов предыдущего временного шага  $y_{(t-1)}$ . Назовем эти весовые векторы  $w_x$  и  $w_y$ . Если мы примем во внимание целый рекуррентный слой, а не лишь один рекуррентный нейрон, тогда можем поместить все весовые векторы в две весовые матрицы,  $W_x$  и  $W_y$ . Затем выходной вектор целого рекуррентного слоя можно вычислить вполне ожидаемым образом, как показано в уравнении 15.1 (здесь  $b$  — вектор смещений, а  $\phi(\cdot)$  — функция активации, скажем, ReLU<sup>1</sup>).

<sup>1</sup> 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 со статьей Ву Фама и др. *Dropout Improves Recurrent Neural Networks for Handwriting Recognition* (Отключение улучшает рекуррентные нейронные сети для распознавания почерка) (<https://homl.info/91>). Сети RNN на основе ReLU также возможны, как поясняют Куок Вьет Ле и др. в своей статье *A Simple Way to Initialize Recurrent Networks of Rectified Linear Units* (Простой способ инициализации рекуррентных сетей выпрямленных линейных элементов) (<https://homl.info/92>).

## Уравнение 15.1. Выход рекуррентного слоя для одиночного образца

$$y_{(t)} = \phi(W_x^T x_{(t)} + W_y^T y_{(t-1)} + b)$$

Подобно нейронным сетям прямого распространения мы можем вычислять выход рекуррентного слоя сразу для целого мини-пакета, помещая все входы на временном шаге  $t$  во входную матрицу  $X_{(t)}$  (уравнение 15.2).

## Уравнение 15.2. Выходы слоя рекуррентных нейронов для всех образцов в мини-пакете

$$\begin{aligned} Y_{(t)} &= \phi(X_{(t)} W_x + Y_{(t-1)} W_y + b) \\ &= \phi([X_{(t)} \ Y_{(t-1)}] W + b) \quad \text{с } W = \begin{bmatrix} W_x \\ W_y \end{bmatrix} \end{aligned}$$

В этом уравнении:

- $Y_{(t)}$  — 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,  $n_{\text{нейронов}}$  — 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 15.2);
- 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 де-

ляет  $Y_{(t)}$  функцией от всех входов с временного шага  $t = 0$  (т.е.  $X_{(0)}, X_{(1)}, \dots, X_{(t)}$ ). На первом временном шаге,  $t = 0$ , предшествующие выходы отсутствуют, потому 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, способную узнавать только короткие образы (как правило, длиной около 10 шагов, но она может меняться в зависимости от задачи). Позже в главе мы рассмотрим ряд более сложных и мощных типов ячеек, которые умеют узнавать более длинные образы (длиннее примерно в 10 раз, но опять-таки это зависит от задачи).

В общем случае состояние ячейки на временном шаге  $t$ , обозначаемое  $h_{(t)}$  ("h" означает "hidden" — "скрытое"), является функцией от некоторых входов на данном временном шаге и ее состояния на предыдущем временном шаге:  $h_{(t)} = f(h_{(t-1)}, x_{(t)})$ . Выход ячейки на временном шаге  $t$ , обозначаемый  $y_{(t)}$ , 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. 15.3).

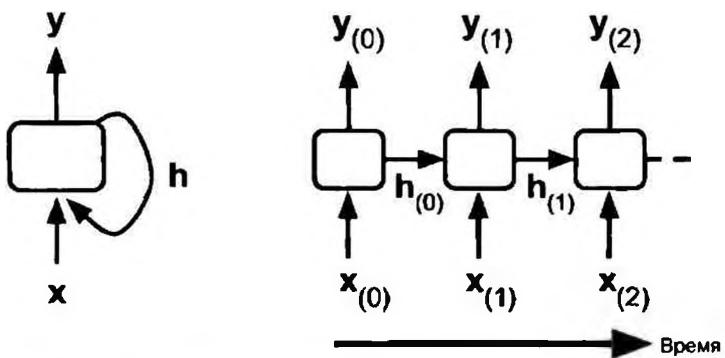


Рис. 15.3. Скрытое состояние ячейки и ее выход могут отличаться

## Входные и выходные последовательности

Сеть RNN может одновременно принимать последовательность входов и порождать последовательность выходов (левая верхняя сеть на рис. 15.4). 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).

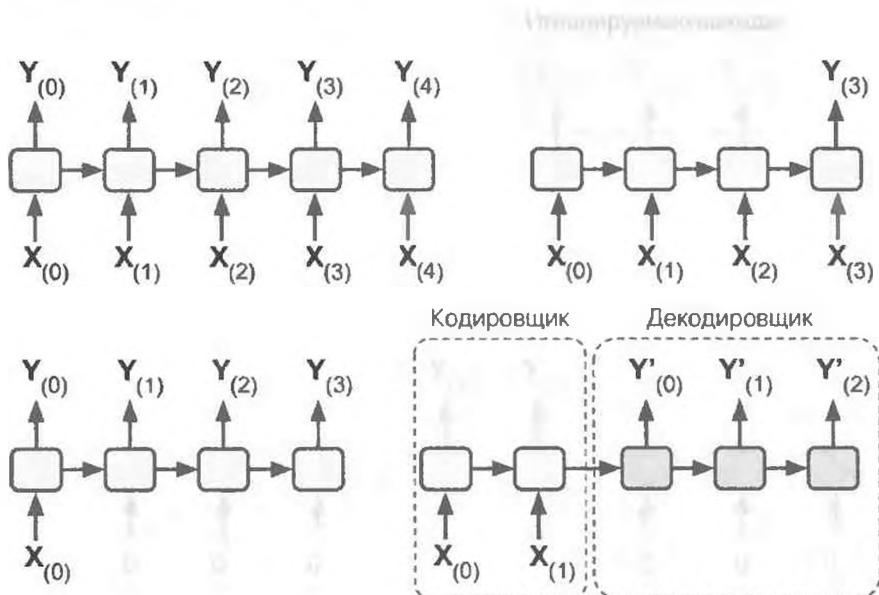


Рис. 15.4. Сети “последовательность в последовательность” (слева вверху), “последовательность в вектор” (справа вверху), “вектор в последовательность” (слева внизу) и “кодировщик-декодировщик” (справа внизу)

В качестве альтернативы вы могли бы передать сети последовательность входов и проигнорировать все выходы кроме последнего (правая верхняя сеть на рис. 15.4). 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 передавать сети один и тот же входной вектор снова и снова на каждом временном шаге и позволить ей выдать последовательность (левая нижняя сеть на рис. 15.4). Это сеть “вектор в последовательность”. Например, входом может быть изображение (или выход сети CNN), а выходом — подпись для 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. 15.4. Такую конфигурацию можно было бы применять, например, 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. 15.4). Дело в том, что последние слова предложения могут повлиять на первые слова перевода (в случае английского языка — Примеч. *пер.*), а потому необходимо подождать завершения предложения, прежде чем переводить его. В главе 16 мы посмотрим, как реализовывать сеть “**кодировщик–декодировщик**” (как выяснится, она сложнее, чем могло показаться, глядя на рис. 15.4).

Звучит многообещающе, но как обучать рекуррентную нейронную сеть?

## Обучение рекуррентных нейронных сетей

Хитрость при обучении сети RNN в том, что ее необходимо развернуть во времени (как только что делалось) и затем просто использовать обыкновенное обратное распространение (рис. 15.5). Такая стратегия называется **обратным распространением во времени** (*backpropagation through time* — *BPTT*).

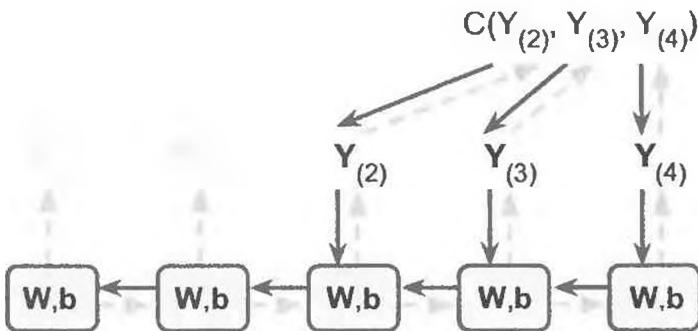


Рис. 15.5. Обратное распространение во времени

Подобно обыкновенному обратному распространению есть первый проход вперед через развернутую сеть (представлен пунктирными линиями со стрелками). Затем выходная последовательность оценивается с использованием функции издержек  $C(Y_{(0)}, Y_{(1)}, \dots Y_{(T)})$ , где  $T$  — максимальный временной шаг. Следует отметить, что указанная функция издержек может игнорировать некоторые выходы, как показано на рис. 15.5 (например, в сети RNN “вектор в последовательность” игнорируются все выходы кроме самого последнего). Далее 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 ВРТГ. Обратите внимание, что градиенты протекают назад через все выходы, использованные функцией издержек, а не только через финальный выход (скажем, на рис. 15.5 функция издержек вычисляется с применением последних трех выходов сети,  $Y_{(2)}$ ,  $Y_{(3)}$  и  $Y_{(4)}$ , поэтому градиенты протекают через упомянутые 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.

К счастью, `tfr.keras` позаботится обо всех сложностях, а потому давайте зайдемся написанием кода!

## Прогнозирование временных рядов

Предположим, что вы изучаете количество активных пользователей в час на своем веб-сайте, ежедневную температуру в своем городе или финансовое состояние своей компании, измеряемое поквартально с применением множества метрик. Во всех указанных случаях данные будут последовательностью из одного и более значений за временной шаг. Она называется *временным рядом*. В первых двух примерах используется одиночное значение на временной шаг, поэтому последовательности будут *одномерными временными рядами*, тогда как в финансовом примере присутствует множество значений на временной шаг (скажем, доход компании, ее долг и т.д.), так что последовательность является *многомерным временным рядом*. Типовая задача заключается в предсказании будущих значений и называется *прогнозированием* (*forecasting*). Еще одна распространенная задача предусматривает заполнение пробелов: чтобы предсказать (или скорее “послесказать”) отсутствующие значения из прошлого. Это называется *вменением* (*imputation*). Например,

на рис. 15.6 показаны три одномерных временных рядов, каждый длиной 50 временных шагов, и цель здесь заключается в том, чтобы спрогнозировать значение на следующем временном шаге (представленным посредством X) для каждого из них.

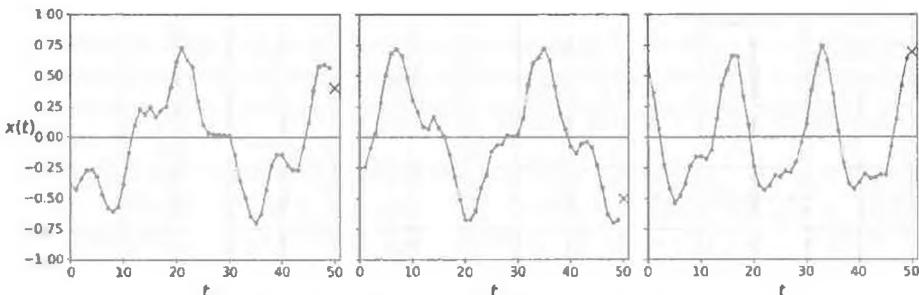


Рис. 15.6. Прогнозирование временных рядов

Для простоты мы применяем временной ряд, сгенерированный приведенной ниже функцией `generate_time_series()`:

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))
        # колебание 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20))
        # + колебание 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5) # + шум
    return series[... , np.newaxis].astype(np.float32)
```

Функция `generate_time_series()` создает столько временных рядов, сколько было запрошено (через аргумент `batch_size`), каждый длиной `n_steps`, и в каждом ряде есть только одно значение на временной шаг (т.е. все ряды одномерные). Функция возвращает массив NumPy формы [размер пакета, количество временных шагов, 1], где каждый ряд представляет собой сумму двух синусоидальных колебаний с фиксированными амплитудами, но случайными частотами и фазами плюс некоторый шум.



При работе с временными рядами (и другими типами последовательностей, такими как предложения) входные признаки обычно представляются в виде трехмерных массивов формы [размер пакета, количество временных шагов, размерность], где размерность равна 1 для одномерных временных рядов и больше 1 для многомерных временных рядов.

Теперь давайте создадим обучающий, проверочный и испытательный наборы, используя эту функцию:

```
n_steps = 50
series = generate_time_series(10000, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

`X_train` содержит 7 000 временных рядов (т.е. имеет форму [7000, 50, 1]), `X_valid` — 2 000 временных рядов (с 7000-ного временного ряда по 8 999-й) и `X_test` — 1 000 временных рядов (с 9 000-ного временного ряда по 9 999-й). Так как мы хотим прогнозировать одиночное значение для каждого ряда, целями будут векторы-столбцы (например, `y_train` имеет форму [7000, 1]).

## Метрики базисного уровня

Перед началом использования сетей RNN часто неплохо иметь несколько метрик базисного уровня, иначе нам может казаться, что модель работает прекрасно, тогда как фактически она хуже базовых моделей. Скажем, самый простой подход предусматривает предсказание последнего значения в каждом ряду. Он называется *наивным прогнозированием* и на удивление временами его трудно превзойти. В этом случае он дает нам среднеквадратическую ошибку (MSE) около 0.020:

```
>>> y_pred = X_valid[:, -1]
>>> np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
0.020211367
```

Еще один простой подход заключается в применении полносвязной сети. Поскольку для каждого входа она ожидает плоский список признаков, нам понадобится добавить слой `Flatten`. Давайте воспользуемся простой линейной регрессионной моделью, так что каждый прогноз будет линейной комбинацией значений во временном ряде:

```
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(1)
])
```

Если скомпилировать такую модель с применением потери MSE и стандартного оптимизатора Adam, затем подогнать ее к обучающему набору в течение 20 эпох и оценить на проверочном наборе, то мы получим MSE около 0.004. Результат намного лучше, чем при наивном подходе!

## Реализация простой рекуррентной нейронной сети

Посмотрим, сумеем ли мы превзойти последний результат с помощью простой сети RNN:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(1, input_shape=[None, 1])
])
```

Мы построили действительно простейшую сеть RNN из всех возможных. Она содержит единственный слой с одним нейроном, как было показано на рис. 15.1. Нам нет необходимости указывать длину входных последовательностей (в отличие от предыдущей модели), потому что рекуррентная нейронная сеть способна обрабатывать любое количество временных шагов (именно потому мы установили первое входное измерение в `None`). По умолчанию слой `SimpleRNN` использует функцию активации в виде гиперболического тангенса. Он работает в точности, как было описано ранее: начальное состояние  $h_{\text{начальное}}$  устанавливается в 0 и передается единственному рекуррентному нейрону вместе со значением первого временного шага  $x_{(0)}$ . Нейрон рассчитывает взвешенную сумму и применяет к результату функцию активации в виде гиперболического тангенса, что даст первый выход  $y_0$ . В простой сети RNN такой выход является и новым состоянием  $h_0$ , которое передается тому же самому рекуррентному нейрону вместе со следующим входным значением  $x_{(1)}$  и процесс повторяется вплоть до последнего временного шага. Затем слой просто выдает последнее значение,  $y_{49}$ . Все это выполняется одновременно для каждого временного ряда.



По умолчанию рекуррентные слои в Keras возвращают только финальный выход. Чтобы заставить их возвращать по одному выходу на временной шаг, потребуется установить `return_sequences=True`, как вскоре будет показано.

После компиляции, подгонки и оценки модели (как и ранее, мы обучаем ее в течение 20 эпох с использованием оптимизатора Adam) обнаружится, что MSE модели достигает лишь 0.014 — результат лучше наивного подхода, но он не превосходит простую линейную модель. Обратите внимание, что для каждого нейрона линейная модель имеет по одному параметру на вход и на временной шаг плюс член смещения (в применяемой нами простой линейной модели в сумме есть 51 параметр). Напротив, для каждого рекуррент-

ного нейрона в простой сети RNN, имеется только по одному параметру на вход и на измерение скрытого состояния (в простой сети RNN это количество рекуррентных нейронов в слое) плюс член смещения. В нашей простой сети RNN всего есть три параметра.

## Тенденция и сезонность

Для прогнозирования временных рядов доступно много других моделей, таких как модели на основе *взвешенного скользящего среднего (weighted moving average)* или *авторегрессионного интегрального скользящего среднего (autoregressive integrated moving average — ARIMA)*. Некоторые из них требуют предварительного устранения тенденции и сезонности. Например, если вы изучаете количество активных пользователей на своем веб-сайте и оно каждый месяц возрастает на 10%, то вам придется удалить такую тенденцию из временного ряда. После того, как модель обучена и начинает вырабатывать прогнозы, вы должны добавить тенденцию обратно, чтобы получать окончательные прогнозы. Аналогично, если вы пытаетесь предсказать количество солнцезащитного лосьона, продаваемое каждый месяц, то вероятно будете наблюдать сильные сезонные колебания: поскольку он хорошо продается каждое лето, похожий шаблон будет повторяться ежегодно. Вам понадобится устраниТЬ такую сезонность из временного ряда, скажем, за счет вычисления разницы между величиной на каждом временном шаге и величиной на год раньше (методика называется *дифференцированием*). Опять-таки после того, как модель обучена и вырабатывает прогнозы, для получения финальных прогнозов вы должны добавить обратно шаблон сезонности.

При использовании сетей RNN обычно нет необходимости выполнять все описанные выше действия, но в ряде случаев они могут повысить эффективность, т.к. модели не придется изучать тенденцию или сезонность.

Очевидно, наша простая сеть RNN была слишком проста, чтобы обеспечить приемлемую эффективность. Давайте попробуем добавить больше рекуррентных слоев!

## Глубокие рекуррентные нейронные сети

Довольно распространенный прием предусматривает укладывание множества слоев ячеек стопкой, как показано на рис. 15.7. В результате мы получаем глубокую рекуррентную нейронную сеть.

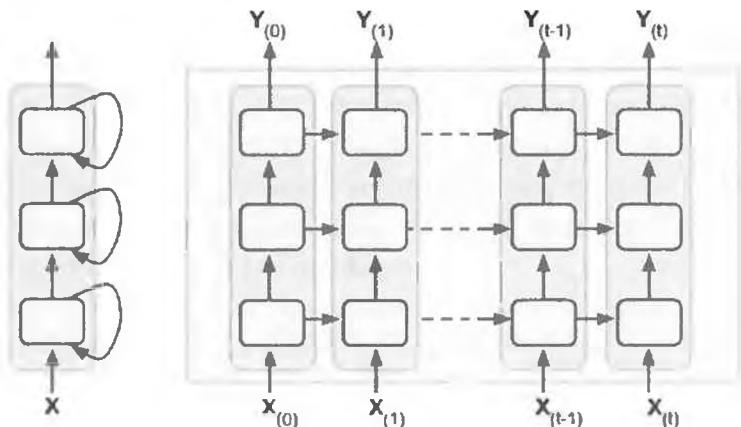


Рис. 15.7. Глубокая рекуррентная нейронная сеть (слева),  
развернутая во времени (справа)

Реализовать глубокую сеть RNN с помощью `tf.keras` достаточно легко: нужно лишь уложить стопкой рекуррентные слои. В приведенном ниже примере мы применяем три слоя SimpleRNN (но можно было бы добавить рекуррентный слой любого другого типа вроде слоя LSTM или GRU, которые мы вскоре обсудим):

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.SimpleRNN(1)
])
```



Обязательно установите `return_sequences=True` для всех рекуррентных слоев (кроме последнего, если вас интересует только последний выход). В противном случае они будут выдавать двумерный массив (содержащий только выход последнего временного шага), а не трехмерный (с выходами всех временных шагов), и следующий рекуррентный слой пожалуется на то, что ему не передаются последовательности в ожидаемом трехмерном формате.

После компиляции, подгонки и оценки модели выяснится, что она достигает величины MSE, равной 0.003. Нам наконец-то удалось обыграть линейную модель!

Обратите внимание, что последний слой не идеален: он обязан иметь единственный элемент, поскольку мы хотим прогнозировать одномерные временные ряды, а значит должны располагать по одному выходному значению на временной шаг. Однако наличие единственного элемента означает, что скрытым состоянием будет лишь одиночное число. В действительности оно дает не так много и не настолько полезно; по-видимому, сеть RNN для переноса всей информации, необходимой от одного временного шага до другого, будет в основном использовать скрытые состояния остальных рекуррентных слоев, и не будет интенсивно задействовать скрытое состояние последнего слоя. Кроме того, поскольку слой SimpleRNN по умолчанию применяет функцию активации `tanh`, прогнозируемые значения обязаны находиться в диапазоне от -1 до 1. Но что, если желательно использовать другую функцию активации? По обеим указанным причинам может быть предпочтительнее заменить выходной слой слоем Dense: он будет быстрее работать, правильность окажется почти такой же и появится возможность выбора любой выходной функции активации. В случае такого изменения также понадобится удалить `return_sequences=True` во втором (теперь последнем) рекуррентном слое:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

Обучив эту модель, вы заметите, что она сходится быстрее и работает в той же степени хорошо. Вдобавок при желании вы можете изменять выходную функцию активации.

## Прогнозирование на несколько временных шагов вперед

До сих пор мы предсказывали значение только на следующем временном шаге, но с той же легкостью можно было бы прогнозировать значение на несколько шагов вперед, надлежащим образом изменения цели (например, для прогнозирования на 10 шагов вперед необходимо лишь изменить цели, что-

бы ими было значение на 10, а не на 1 шаг вперед). Но что, если нужно спрогнозировать 10 последующих значений?

В первом варианте мы применяем уже обученную нами модель, заставляем ее прогнозировать следующее значение, затем добавляем это значение к входам (действуя так, как если бы спрогнозированное значение фактически встретилось), используем модель снова для прогнозирования следующего значения и т.д., как демонстрируется в показанном ниже коде:

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)
Y_pred = X[:, n_steps:]
```

Прогноз для следующего шага вполне ожидаемо будет точнее прогнозов для более поздних временных шагов, т.к. ошибки могут накапливаться (как видно на рис. 15.8).

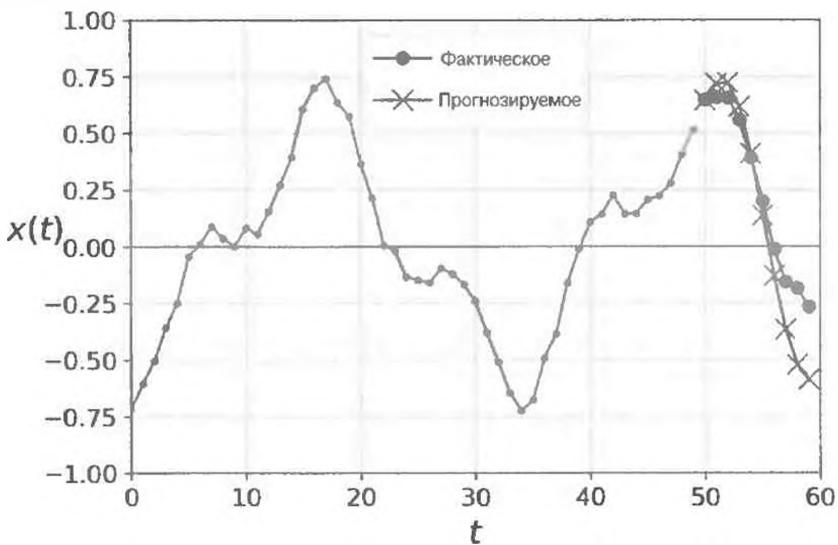


Рис. 15.8. Прогнозирование на 10 шагов вперед по 1 шагу за раз

Оценка этого подхода на проверочном наборе даст величину MSE, равную 0.029, что намного выше, чем в предшествующих моделях, но и сама задача гораздо сложнее, а потому сравнение ничего не значит. Намного важнее

сравнить эффективность данного подхода с эффективностью наивного подхода (прогнозируя, что временной ряд останется постоянным на протяжении 10 временных шагов) или простой линейной модели. Наивный подход ужасен (он дает MSE около 0.223), но линейная модель обеспечивает MSE около 0.0188: намного лучше применения нашей сети RNN для прогнозирования одного шага за раз и также быстрее в плане обучения и работы. Тем не менее, если требуется прогнозирование только на несколько шагов вперед для более сложных задач, тогда описанный подход может работать неплохо.

Второй вариант предусматривает обучение сети RNN для прогнозирования сразу всех 10 последующих значений.

Мы можем использовать модель “последовательность в вектор”, но она будет выдавать 10 значений, а не 1. Однако сначала нам необходимо изменить цели, сделав их векторами с 10 последующими значениями:

```
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps],
                     series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

Теперь нам нужно, чтобы выходной слой имел 10 элементов взамен 1:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

После обучения модели можно очень легко прогнозировать 10 последующих значений:

```
Y_pred = model.predict(X_new)
```

Модель работает хорошо: ошибка MSE для следующих 10 временных шагов составляет около 0.008. Она гораздо лучше линейной модели. Но мы все еще можем добиться большего: вместо обучения модели прогнозированию 10 последующих значений только на самом последнем временном шаге мы можем научить ее прогнозировать 10 последующих значений на каждом временном шаге. Другими словами, мы можем превратить такую сеть RNN типа “последовательность в вектор” в сеть RNN вида “последовательность в последовательность”. Преимущество этой методики в том, что потеря бу-

деть содержать член для выхода сети RNN на каждом временном шаге, а не только на последнем. Тогда через модель будет протекать намного больше градиентов ошибок и им не придется протекать только сквозь время; они также будут вытекать из выхода каждого временного шага. В итоге обучение становится более устойчивым и быстрым.

Чтобы прояснить: на временном шаге 0 модель будет выдавать вектор, содержащий прогнозы для временных шагов 1–10, затем на временном шаге 1 модель спрогнозирует временные шаги 2–11 и т.д. Таким образом, целью должна быть последовательность с той же длиной, что у входной последовательности, содержащей 10-мерный вектор на каждом шаге. Давайте подготовим такие целевые последовательности:

```
Y = np.empty((10000, n_steps, 10))    # каждая цель является
                                         # последовательностью из 10-мерных векторов
for step_ahead in range(1, 10 + 1):
    Y[:, :, step_ahead - 1] =
        series[:, step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```



Может показаться удивительным тот факт, что цели будут содержать значения, которые появляются во входах (между X\_train и Y\_train есть много совпадений). Разве это не обман? К счастью, все нет: на каждом временном шаге модели известно только о прошедших временных шагах, а потому она не может заглядывать вперед. Говорят, что она должна быть причинной (*causal*) моделью.

Для превращения модели в модель типа “последовательность в последовательность” мы должны установить `return_sequences=True` во всех рекуррентных слоях (даже в последнем) и применять выходной слой Dense на каждом временном шаге. Как раз для такой цели в Keras предлагается слой `TimeDistributed`: он оборачивает любой слой (скажем, Dense) и применяет его на каждом временном шаге своей входной последовательности. Он делает это эффективно, изменяя форму входов, так что каждый временной шаг трактуется как отдельный образец (т.е. изменяет форму входов с [размер пакета, количество временных шагов, количество входных измерений] на [размер пакета × количество временных шагов, количество

входных измерений]; в текущем примере количество входных измерений составляет 20, потому что предыдущий слой SimpleRNN имеет 20 элементов). Затем он запускает слой Dense и в заключение изменяет форму выходов обратно на последовательности (т.е. изменяет форму выходов с [размер пакета  $\times$  количество временных шагов, количество выходных измерений] на [размер пакета, количество временных шагов, количество выходных измерений]; в данном примере количество выходных измерений равно 10, т.к. слой Dense имеет 10 элементов)<sup>2</sup>. Вот обновленная модель:

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Слой Dense в действительности поддерживает последовательности в качестве входов (и даже многомерных входов): он обрабатывает их точно как TimeDistributed(Dense(...)), т.е. применяется только к последнему входному измерению (независимо по всем временным шагам). Соответственно, мы могли бы заменить последний слой на Dense(10). Тем не менее, для ясности мы продолжим использовать вариант TimeDistributed(Dense(10)), т.к. он дает понять, что слой Dense применяется независимо на каждом временном шаге, и модель будет выдавать последовательность, а не одиночный вектор.

На стадии обучения необходимы все выходы, но для вырабатывания прогнозов и оценки полезен только вход на последнем временном шаге. Таким образом, хотя при обучении мы будем полагаться на MSE по всем выходам, при оценке мы будем использовать специальную метрику, чтобы вычислять MSE только по выходу на последнем временном шаге:

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1],
                                             Y_pred[:, -1])

optimizer = keras.optimizers.Adam(lr=0.01)
model.compile(loss="mse", optimizer=optimizer,
               metrics=[last_time_step_mse])
```

<sup>2</sup> Обратите внимание, что слой TimeDistributed(Dense(n)) эквивалентен слою Conv1D(n, filter\_size=1).

Ошибка MSE при проверке составляет около 0.006, что на 25% лучше, чем у предыдущей модели. Вы можете скомбинировать этот подход с первым подходом. Просто спрогнозируйте 10 последующих значений с применением данной сети RNN, затем объедините полученные значения и входной временной ряд, после чего используйте модель снова для прогнозирования следующих 10 значений; повторяйте процесс столько раз, сколько необходимо. Благодаря такому подходу вы можете генерировать произвольно длинные последовательности. Он может оказаться не очень точным для долгосрочных прогнозов, но вполне хорош, если ваша цель заключается в генерировании оригинальной музыки или текста, как будет показано в главе 16.



При прогнозировании временных рядов наряду с прогнозами часто удобно иметь какие-то величины ошибок. Эффективной методикой для этого является отключение MC Dropout, представленное в главе 11: добавьте внутрь каждой ячейки памяти слой MC Dropout, отбрасывающий часть входов и скрытых состояний. После того, как модель обучена, для прогнозирования нового временного ряда применяйте ее много раз и рассчитывайте среднюю величину и стандартное отклонение прогнозов на каждом временном шаге.

Простые сети RNN могут быть приемлемыми в случае прогнозирования временных рядов или обработки последовательностей других видов, но на длинных временных рядах или последовательностях они работают не настолько хорошо. Давайте обсудим причины и посмотрим, что можно предпринять.

## Обработка длинных последовательностей

Чтобы обучить сеть RNN на длинных последовательностях, нам придется прогонять ее через множество временных шагов, делая развернутую сеть RNN очень глубокой. Подобно любой глубокой нейронной сети сеть RNN может страдать от проблемы нестабильных градиентов, рассмотренной в главе 11: обучаться бесконечно долго или процесс обучения окажется неустойчивым. Кроме того, когда сеть RNN обрабатывает длинную последовательность, она будет постепенно забывать о первых входах в последовательности. Ниже мы раскроем эти проблемы, начиная с проблемы нестабильных градиентов.

## Борьба с проблемой нестабильных градиентов

Многие трюки, которые мы использовали в глубоких сетях для смягчения проблемы нестабильных градиентов, можно также применять в отношении сетей RNN: хорошая инициализация параметров, более быстрые оптимизаторы, отключение и т.д. Однако *ненасыщаемые функции активации* (скажем, *ReLU*) могут не сильно здесь помочь; на самом деле они способны приводить к тому, что сеть RNN становится еще более нестабильной на стадии обучения. Почему? Допустим, градиентный спуск обновляет веса таким способом, что выходы слегка увеличиваются на первом временном шаге. Поскольку на каждом временном шаге используются одни и те же веса, выходы на втором временном шаге также могут немного увеличиться, то же самое на третьем шаге и т.д., пока не произойдет взрывной рост выходов — ненасыщаемая функция активации не предотвращает это. Вам удастся снизить риск возникновения такой ситуации за счет выбора меньшей скорости обучения, но вы также можете просто применить насыщаемую функцию активации вроде гиперболического тангенса (вот почему она является стандартным вариантом). Во многом аналогичным образом взрывному росту могут быть подвержены и сами градиенты. Если вы замечаете, что процесс обучения нестабилен, тогда имеет смысл отследить размер градиентов (например, используя TensorBoard) и возможно применить отсечение градиентов.

Более того, *пакетная нормализация* (BN) не может использоваться с сетями RNN настолько же эффективно, как с глубокими сетями прямого распространения. Фактически ее нельзя задействовать между временными шагами, а лишь между рекуррентными слоями. Точнее говоря, формально можно добавить слой BN к ячейке памяти (как вскоре будет показано), так что он будет применяться к каждомуциальному временишагу (на входах для текущего временного шага и на скрытом состоянии из предыдущего шага). Тем не менее, на каждом временном шаге будет использоваться тот же самый слой BN с теми же самыми параметрами, не считаясь с действительным масштабом и смещением входов и скрытого состояния. На практике это не приводит к хорошим результатам, как продемонстрировали Сезар Лоран и др. в своей статье 2015 года (<https://homl.info/rnnbn>)<sup>3</sup>: авторы обнаружили, что пакетная нормализация была немного полезной только когда применялась к входам, но не

<sup>3</sup> Сезар Лоран и др., *Batch Normalized Recurrent Neural Networks* (Рекуррентные нейронные сети с пакетной нормализацией), *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2016 г.): с. 2657–2661.

к скрытым состояниям. Другими словами, она была чуть лучше, чем ничего, в случае применения между рекуррентными слоями (т.е. вертикально на рис. 15.7), но не внутри рекуррентных слоев (т.е. горизонтально). В Keras это можно сделать, просто добавив слой `BatchNormalization` перед каждым рекуррентным слоем, но не ожидайте от него слишком много.

С сетями RNN часто лучше работает другая форма нормализации: нормализация по слою (*Layer Normalization*). Ее идея была представлена Джимми Ли Ба и др. в своей статье 2016 года (<https://homl.info/layernorm>)<sup>4</sup>: она очень похожа на пакетную нормализацию, но вместо того, чтобы нормализовать по измерению пакета, она нормализирует по измерению признаков. Одно из преимуществ заключается в том, что нормализация по слою способна рассчитывать требуемые статистические данные на лету, на каждом временном шаге, независимо для каждого образца. Это также означает, что она ведет себя одинаково на стадии обучения и испытаний (в противоположность BN) и не нуждается в использовании экспоненциальных скользящих средних при расчете оценок статистических данных для признаков по всем образцам в обучающем наборе. Подобно BN нормализация по слою узнает масштаб и параметр смещения для каждого входа. В сети RNN она обычно применяется сразу после линейной комбинации входов и скрытых состояний.

Давайте воспользуемся `tf.keras`, чтобы реализовать нормализацию по слою внутри простой ячейки памяти. Для этого нам необходимо определить специальную ячейку памяти. Она похожа на обычный слой за исключением того, что ее метод `call()` принимает два аргумента: входы `inputs` на текущем временном шаге и скрытые состояния `states` из предыдущего временного шага. Обратите внимание, что аргумент `states` является списком, содержащим один или большее количество тензоров. В случае простой ячейки RNN он содержит единственный тензор, равный выходам предыдущего временного шага, но другие ячейки могут иметь множество тензоров состояния (например, ячейка `LSTMCell` обладает долгосрочным и краткосрочным состоянием, как вскоре будет показано). Ячейка также обязана иметь атрибуты `state_size` и `output_size`. В простой сети RNN оба атрибута равны количеству элементов. В следующем коде реализована спе-

<sup>4</sup> Джимми Ли Ба и др., *Layer Normalization* (Нормализация по слою), препринт arXiv:1607.06450 (2016 г.).

циальная ячейка памяти, которая ведет себя подобно SimpleRNNCell, но также применяет нормализацию по слою на каждом временном шаге:

```
class LNSimpleRNNCell(keras.layers.Layer):
    def __init__(self, units, activation="tanh", **kwargs):
        super().__init__(**kwargs)
        self.state_size = units
        self.output_size = units
        self.simple_rnn_cell = keras.layers.SimpleRNNCell(units,
                                                          activation=None)
        self.layer_norm = keras.layers.LayerNormalization()
        self.activation = keras.activations.get(activation)
    def call(self, inputs, states):
        outputs, new_states = self.simple_rnn_cell(inputs, states)
        norm_outputs = self.activation(self.layer_norm(outputs))
        return norm_outputs, [norm_outputs]
```

Код довольно прямолинеен<sup>5</sup>. Как и любой специальный слой, наш класс LNSimpleRNNCell унаследован от класса keras.layers.Layer. Конструктор принимает количество элементов и желаемую функцию активации, устанавливает атрибуты state\_size и output\_size, после чего создает ячейку SimpleRNNCell без какой-либо функции активации (поскольку мы хотим выполнять нормализацию по слою после линейной операции, но перед функцией активации). Затем конструктор создает слой LayerNormalization и в заключение извлекает желаемую функцию активации. Метод call() начинается с применения простой ячейки RNN, которая вычисляет линейную комбинацию текущих входов с предыдущими скрытыми состояниями и возвращает результат дважды (на самом деле выходы в ячейке SimpleRNNCell просто эквивалентны скрытым состояниям: другими словами, new\_states[0] равно outputs, так что мы можем безопасно игнорировать new\_states в оставшейся части метода call()). Далее метод call() применяет нормализацию по слою, за которой следует функция активации. Наконец, он возвращает выходы дважды (один раз как выходы и еще раз как новые скрытые состояния). Чтобы использовать такую специальную ячейку, нам понадобится лишь создать слой keras.layers.RNN, передав ему экземпляр ячейки:

---

<sup>5</sup> Было бы проще взамен наследовать от класса SimpleRNNCell, чтобы затем не приходилось создавать внутреннюю ячейку SimpleRNNCell или обрабатывать атрибуты state\_size и output\_size, но целью здесь была демонстрация создания специальной ячейки с нуля.

```
model = keras.models.Sequential([
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True,
                      input_shape=[None, 1]),
    keras.layers.RNN(LNSimpleRNNCell(20), return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Аналогичным образом вы могли бы создать специальную ячейку для применения отключения между временными шагами. Но существует более простой способ: все рекуррентные слои (кроме `keras.layers.RNN`) и все ячейки, предоставляемые Keras, имеют гиперпараметры `dropout` и `recurrent_dropout`: первый определяет долю отключения для применения к входам (на каждом временном шаге), а второй — долю отключения для применения к скрытым состояниям (тоже на каждом временном шаге). Нет нужды создавать специальную ячейку, чтобы применять отключение на каждом временном шаге в сети RNN.

С помощью описанных методик вы можете смягчить проблему нестабильных градиентов и гораздо эффективнее обучать сеть RNN. Теперь посмотрим, что делать с проблемой краткосрочной памяти.

## Борьба с проблемой краткосрочной памяти

Из-за трансформаций, которым подвергаются данные при проходе через сеть RNN, на каждом временном шаге определенная информация утрачивается. Через некоторое время состояние сети RNN практически не содержит следов первых входов. Может возникнуть накладка. Вообразите себе рыбку Дори<sup>6</sup>, пытающуюся интерпретировать длинное предложение; к тому времени, как Дори закончит его читать, она не будет иметь ни малейшего понятия, с чего предложение начиналось. Чтобы справиться с проблемой, были предложены 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.

---

<sup>6</sup> Персонаж из мультипликационных фильмов “Finding Dory” (“В поисках Дори”) и “Finding Nemo” (“В поисках Немо”), страдающий провалами в памяти.

## Ячейки LSTM

Ячейка долгой краткосрочной памяти (*Long Short-Term Memory — LSTM*) была предложена в 1997 году (<https://homl.info/93>)<sup>7</sup> Сеппом Хохрайтером и Юргеном Шмидхубером, а с годами понемногу совершенствовалась несколькими исследователями, в том числе Алексом Грэйвзом (<https://homl.info/graves>), Хасимом Саком (<https://homl.info/94>)<sup>8</sup> и Войцехом Зарембой (<https://homl.info/95>)<sup>9</sup>. 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 Keras можно просто применить слой LSTM вместо слоя SimpleRNN:

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

В качестве альтернативы можно было бы использовать универсальный слой keras.layers.RNN, передав ему в аргументе ячейку LSTMCell:

```
model = keras.models.Sequential([
    keras.layers.RNN(keras.layers.LSTMCell(20),
                    return_sequences=True,
                    input_shape=[None, 1]),
    keras.layers.RNN(keras.layers.LSTMCell(20),
                    return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

Однако в случае запуска в графическом процессоре (см. главу 19) слой LSTM применяет оптимизированную реализацию, поэтому в целом исполь-

<sup>7</sup> Сепп Хохрайтер и Юрген Шмидхубер, *Long Short-Term Memory* (Долгая краткосрочная память), *Neural Computation* 9, номер 8 (1997 г.): с. 1735–1780.

<sup>8</sup> Хасим Сак и др., *Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition* (Архитектуры рекуррентных нейронных сетей с долгой краткосрочной памятью для распознавания речи с большими словарями), препринт arXiv:1402.1128 (2014 г.).

<sup>9</sup> Войцех Заремба и др., *Recurrent Neural Network Regularization* (Регуляризация рекуррентных нейронных сетей), препринт arXiv:1409.2329 (2014 г.).

зователь его предпочтительнее (слой RNN в основном полезен, когда определяются специальные ячейки, как делалось ранее).

Так каким же образом работает ячейка LSTM? Ее архитектура показана на рис. 15.9.

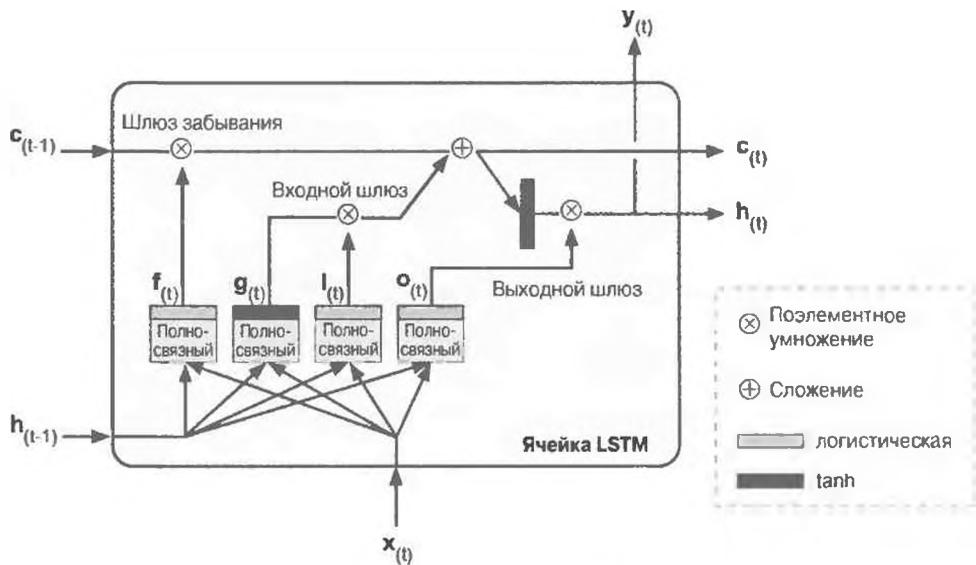


Рис. 15.9. Ячейка LSTM

Если не смотреть на то, что находится внутри ящика, то ячейка LSTM выглядит в точности как обыкновенная ячейка, но с расщеплением своего состояния на два вектора:  $h_{(t)}$  и  $c_{(t)}$  ("с" обозначает "cell" ("ячейка")). Можете считать  $h_{(t)}$  краткосрочным состоянием, а  $c_{(t)}$  — долгосрочным состоянием.

А теперь давайте приоткроем ящик! Основная идея заключается 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  $c_{(t-1)}$  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  $c_{(t)}$  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  $h_{(t)}$  (которое равно выходу ячейки для данного временного шага,  $y_{(t)}$ ). Теперь посмотрим, откуда поступают новые воспоминания, и каким образом работают шлюзы.

Первым делом текущий входной вектор  $x_{(t)}$  и предыдущее краткосрочное состояние  $h_{(t-1)}$  передаются четырем полносвязным слоям. Все они служат разным целям.

- Главный слой выдает  $g_{(t)}$ . Он исполняет обычную роль, анализируя текущие входы  $x_{(t)}$  и предыдущее (краткосрочное) состояние  $h_{(t-1)}$ . 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 взамен его самые важные части сохраняются в долгосрочном состоянии (с отбрасыванием остатка).
- 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;
  - выходной шлюз (контролируемый  $o_{(t)}$ ) управляет тем, какие части долгосрочного состояния должны быть прочитаны и выданы на текущем временном шаге в  $h_{(t)}$  и в  $y_{(t)}$ .

Короче говоря, ячейка LSTM способна научиться распознавать важный вход (роль входного шлюза), записывать его в долгосрочное состояние, хранить 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 многоного другого.

В уравнении 15.3 показано, как вычислять долгосрочное состояние ячейки и ее выход на каждом временном шаге для одиночного образца (уравнение для целого мини-пакета очень похожи).

### Уравнение 15.3. Вычисления, связанные с ячейкой LSTM

$$\begin{aligned} i_{(t)} &= \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t-1)} + b_i) \\ f_{(t)} &= \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t-1)} + b_f) \\ o_{(t)} &= \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t-1)} + b_o) \\ g_{(t)} &= \tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t-1)} + b_g) \\ c_{(t)} &= f_{(t)} \otimes c_{(t-1)} + i_{(t)} \otimes g_{(t)} \\ y_{(t)} &= h_{(t)} = o_{(t)} \otimes \tanh(c_{(t)}) \end{aligned}$$

Разберем уравнение 15.3.

- $W_{xi}$ ,  $W_{xf}$ ,  $W_{xo}$  и  $W_{xg}$  — матрицы весов каждого из четырех слоев для их связи с входным вектором  $x_{(t)}$ .
- $W_{hi}$ ,  $W_{hf}$ ,  $W_{ho}$  и  $W_{hg}$  — матрицы весов каждого из четырех слоев для их связи с предыдущим краткосрочным состоянием  $h_{(t-1)}$ .
- $b_i$ ,  $b_f$ ,  $b_o$  и  $b_g$  — члены смещения для каждого из четырех слоев. Обратите внимание, что TensorFlow инициализирует  $b_f$  вектором, заполненным единицами, а не нулями. Это препятствует забыванию чего-либо в начале обучения.

### Смотровые связи

В обыкновенной ячейке LSTM контроллеры шлюзов способны просматривать только вход  $x_{(t)}$  и предыдущее краткосрочное состояние  $h_{(t-1)}$ . Может оказаться неплохой идеей предоставлять им чуть больше контекста, позволив заглядывать также в долгосрочное состояние. Такая идея была выдвинута Феликсом Герсом и Юргеном Шмидхубером в 2000 году (<https://homl.info/96>)<sup>10</sup>. Они предложили вариант ячейки LSTM с дополнительными связями, называемыми *смотровыми связями* (*recurrent connection*): к контроллерам шлюза забывания и входного шлюза в качестве входа добавляется предыдущее долгосрочное состояние  $c_{(t-1)}$ , а к контроллеру выходного шлюза — текущее долгосрочное состояние  $c_{(t)}$ . Смотровые связи часто повышают эффективность, но не всегда, и нет четкого принципа, для каких

<sup>10</sup> Феликс Герс и Юрген Шмидхубер, *Recurrent Nets That Time and Count* (Рекуррентные сети, которые распределяют время и подсчитывают), *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks* (2000 г.): с. 189–194.

задач их лучше применять, а для каких нет: вам придется опробовать их на своей задаче и посмотреть, помогут ли они.

Слой LSTM в Keras основан на ячейке `keras.layers.LSTMCell`, которая не поддерживает смотровые связи. Тем не менее, экспериментальная ячейка `tf.keras.experimental.PeepholeLSTMCell` их поддерживает, так что вы можете создать слой `keras.layers.RNN`, передав его конструктору ячейку `PeepholeLSTMCell`.

Существует много других вариантов ячейки LSTM. Особенно популярной среди вариантов является ячейка GRU, которую мы сейчас рассмотрим.

### Ячейки GRU

Ячейка управляемого рекуррентного блока (*Gated Recurrent Unit — GRU*), показанная на рис. 15.10, предложена Къенгъеном Чо и др. в статье 2014 года (<https://hml.info/97>)<sup>11</sup>, где также была представлена упомянутая ранее сеть “кодировщик–декодировщик”.

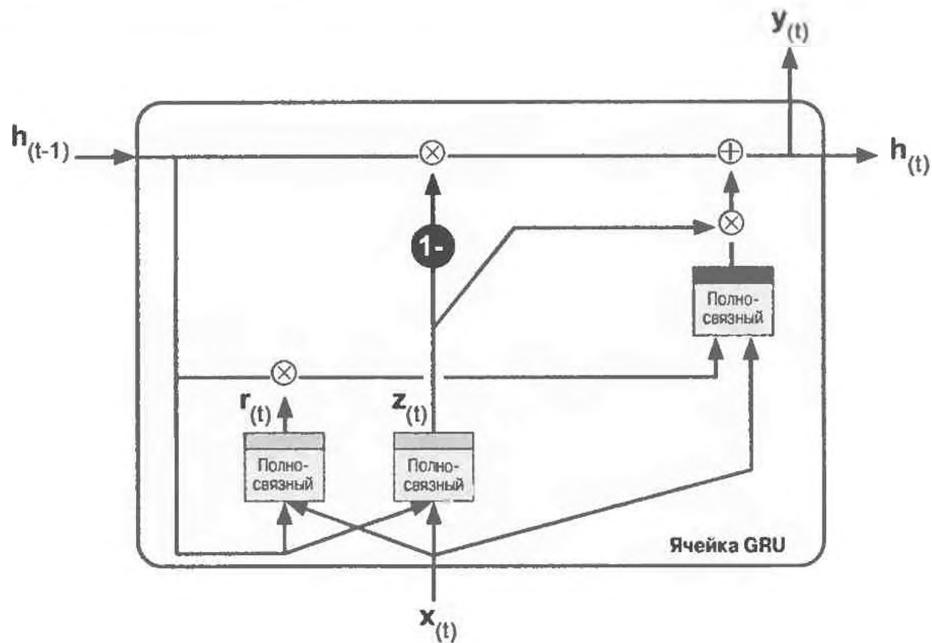


Рис. 15.10. Ячейка GRU

<sup>11</sup> Къенгъен Чо и др., *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), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing* (2014 г.): с. 1724–1734.

Ячейка GRU — это упрощенная версия ячейки LSTM, которая, по-видимому, работает в равной мере хорошо<sup>12</sup> (чем и объясняется ее растущая популярность). Ниже перечислены главные упрощения.

- Оба вектора состояния объединены в единственный вектор  $\mathbf{h}_{(t)}$ .
- Один контроллер шлюза  $\mathbf{z}_{(t)}$  управляет шлюзом забывания и входным шлюзом. Если контроллер шлюза выдает 1, то шлюз забывания открывается ( $= 1$ ), а входной шлюз закрывается ( $1 - 1 = 0$ ). 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 частный вариант и для самой ячейки LSTM.
- Выходной шлюз отсутствует; на каждом временном шаге выдается полный вектор состояния. Тем не менее, имеется новый контроллер шлюза  $\mathbf{r}_{(t)}$ , который управляет тем, какие части предыдущего состояния будут показаны главному слою ( $\mathbf{g}_{(t)}$ ).

В уравнении 15.4 представлен способ вычисления состояния ячейки на каждом временном шаге для одиночного образца.

#### Уравнение 15.4. Вычисления, связанные с ячейкой GRU

$$\begin{aligned}\mathbf{z}_{(t)} &= \sigma\left(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \\ \mathbf{r}_{(t)} &= \sigma\left(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \\ \mathbf{g}_{(t)} &= \tanh\left(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T (\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}) + \mathbf{b}_g\right) \\ \mathbf{h}_{(t)} &= \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + (1 - \mathbf{z}_{(t)}) \otimes \mathbf{g}_{(t)}\end{aligned}$$

В Keras предлагается слой `keras.layers.GRU` (основанный на ячейке памяти `keras.layers.GRUCell`); его применение сводится просто к замене `SimpleRNN` или `LSTM` на `GRU`.

Ячейки LSTM и GRU являются одной из главных причин, лежащих в основе успеха сетей RNN. Несмотря на то что такие ячейки могут справляться с намного более длинными последовательностями, чем простые сети RNN, они по-прежнему имеют довольно ограниченную краткосрочную память и

<sup>12</sup> В статье 2015 г. (<https://homl.info/98>) *LSTM: A Search Space Odyssey* (LSTM: схемы пространства поиска) Клаус Грефф и др. показали, что все варианты LSTM работают примерно одинаково.

им трудно узнавать установившиеся образы в последовательностях из 100 и выше временных шагов, таких как аудио-образцы, длинные временные ряды или длинные предложения. Одно из решений проблемы предусматривает укорачивание входных последовательностей, скажем, с использованием одномерных сверточных слоев.

## Использование одномерных сверточных слоев для обработки последовательностей

В главе 14 было показано, что двумерный сверточный слой работает путем плавного перемещения довольно небольших ядер (или фильтров) по изображению, выдавая множество двумерных карт признаков (по одной на ядро). Аналогично одномерный сверточный слой плавно перемещает несколько ядер через последовательность, производя по одной одномерной карте признаков на ядро. Каждое ядро учится выявлять одиничный очень короткий последовательный образ (не длиннее размера ядра). Если вы применяете 10 ядер, тогда выход слоя будет состоять из 10 одномерных последовательностей (одинаковой длины) или, что эквивалентно, вы можете представить такой вывод в виде одной 10-мерной последовательности. Это значит, что вы можете построить нейронную сеть, образованную из смеси рекуррентных слоев и одномерных сверточных слоев (или даже одномерных объединяющих слоев). В случае использования одномерного сверточного слоя со страйдом 1 и дополнением "same" выходная последовательность будет иметь ту же самую длину, что и входная последовательность. Но если применяется дополнение "valid" или страйд больше 1, тогда выходная последовательность окажется короче входной, а потому удостоверьтесь в том, что соответствующим образом скорректировали цели. Например, приведенная ниже модель такая же, как ранее, за исключением того, что начинается с одномерного сверточного слоя, который понижает дискретизацию входной последовательности в 2 раза, используя страйд 2. Размер ядра больше страйда, поэтому для расчета выхода слоя будут применяться все входы и, следовательно, модель способна научиться предохранять полезную информацию, отбрасывая только неважные детали. За счет укорачивания последовательностей сверточный слой может помогать слоям GRU с выявлением более длинных образов. Обратите внимание, что мы обязаны также подрезать первые три временных шага в целях (поскольку размер ядра составляет 4, первый выход сверточного слоя будет основан на входных временных шагах с 0 по 3) и понизить дискретизацию целей в 2 раза:

```

model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4,
                        strides=2, padding="valid",
                        input_shape=[None, 1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
model.compile(loss="mse", optimizer="adam",
               metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train[:, 3::2], epochs=20,
                      validation_data=(X_valid, Y_valid[:, 3::2]))

```

Выполнив обучение и оценку этой модели, вы обнаружите, что она является наилучшей из всех рассмотренных до сих пор. Сверточный слой действительно помогает. Фактически можно использовать только одномерные сверточные слои и полностью отказаться от рекуррентных слоев!

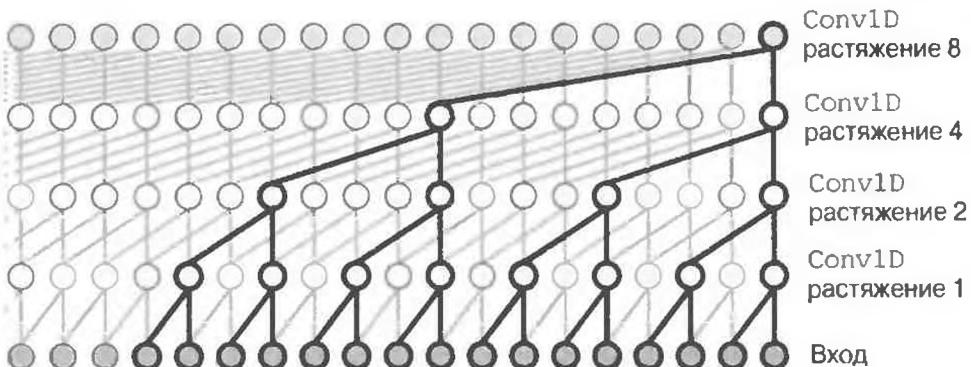
## Архитектура WaveNet

В своей статье 2016 года (<https://homl.info/wavenet>)<sup>13</sup> Аарон ван ден Орд и другие исследователи из DeepMind представили архитектуру под названием *WaveNet*. Они уложили одномерные сверточные слои стопкой, на каждом слое дублируя степень растяжения (насколько далеко разбросаны входы каждого нейрона): первый сверточный слой видит только два временных шага за раз, второй — четыре (его рецепторное поле имеет длину в четыре временных шага), следующий слой — восемь временных шагов и т.д. (рис. 15.11). Таким образом, более низкие слои узнают краткосрочные образы, тогда как более высокие слои — долгосрочные образы. Благодаря дублированию степени растяжения сеть способна очень эффективно обрабатывать исключительно крупные последовательности.

В своей статье по WaveNet авторы на самом деле уложили стопкой 10 сверточных слоев со степенями растяжения 1, 2, 4, 8, ..., 256, 512, затем уложили стопкой еще одну группу из 10 идентичных слоев (тоже со степенями растяжения 1, 2, 4, 8, ..., 256, 512) и далее снова такую же группу из 10 слоев.

---

<sup>13</sup> Аарон ван ден Орд и др., *WaveNet: A Generative Model for Raw Audio* (WaveNet: порождающая модель для необработанных аудиоданных), препринт arXiv:1609.03499 (2016 г.).



*Рис. 15.11. Архитектура WaveNet*

Авторы обосновали эту архитектуру, указав на то, что одиночная стопка из 10 сверточных слоев с такими степенями растяжения будет действовать подобно сверхэффективному сверточному слою с ядром размера 1024 (помимо того, что он быстрее, мощнее и применяет меньше параметров), из-за чего они уложили стопкой 3 таких блока. Авторы также дополняли входные последовательности слева таким количеством нулей, которое равно степени растяжения перед каждым слоем, чтобы предохранить ту же самую длину последовательности на всем протяжении сети. Вот как реализовать упрощенную сеть WaveNet для обработки ранее используемых последовательностей<sup>14</sup>:

```
model = keras.models.Sequential()
model.add(keras.layers.InputLayer(input_shape=[None, 1]))
for rate in (1, 2, 4, 8) * 2:
    model.add(keras.layers.Conv1D(filters=20,
                                 kernel_size=2, padding="causal",
                                 activation="relu", dilation_rate=rate))
model.add(keras.layers.Conv1D(filters=10, kernel_size=1))
model.compile(loss="mse", optimizer="adam",
              metrics=[last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                     validation_data=(X_valid, Y_valid))
```

<sup>14</sup> В полном варианте WaveNet используется еще несколько трюков вроде обходящих связей, как в ResNet, и управляемых элементов активации (*Gated Activation Unit*), аналогичных тем, которые находятся в ячейке GRU. Дополнительные детали ищите в тетради Jupiter для настоящей главы.

Построенная модель `Sequential` начинается с явного входного слоя (это проще, чем пытаться установить `input_shape` только на первом слое) и продолжается одномерным сверточным слоем, применяющим дополнение "causal": прием гарантирует, что сверточный слой не заглядывает в будущее при вырабатывании прогнозов (результат эквивалентен дополнению входов корректным количеством нулей слева и использованию дополнения "valid"). Затем мы добавляем похожие пары слоев с растущими степенями растяжения: 1, 2, 4, 8 и снова 1, 2, 4, 8. В заключение мы добавляем выходной слой: сверточный слой с 10 фильтрами размера 1 и без какой-либо функции активации. Благодаря дополняющим слоям каждый сверточный слой выдает последовательность с такой же длиной, как у входных последовательностей, поэтому цели, применяемые во время обучения, могут быть полными последовательностями: нет необходимости обрезать их или понижать дискретизацию.

Последние две модели обеспечили наилучшие результаты в прогнозировании временных рядов! В статье, посвященной WaveNet, авторы добились современного уровня эффективности на различных аудио-задачах (отсюда и название архитектуры), включая преобразование текста в речь с созданием невероятно реалистичных голосов на нескольких языках. Они также использовали модель для генерации музыки по одному аудио-образцу за раз. Этот ловкий трюк тем более впечатляет, когда вы осознаете, что одна секунда аудио-образца может содержать десятки тысяч временных шагов — даже ячейки LSTM и GRU не способны обрабатывать настолько длинные последовательности.

В главе 16 мы продолжим исследование сетей RNN и посмотрим, как они могут справляться с разнообразными задачами NLP.

## Упражнения

1. Можете ли вы придумать несколько приложений для сети RNN типа "последовательность в последовательность"? Как насчет сети RNN типа "последовательность в вектор" и сети RNN типа "вектор в последовательность"?
2. Сколько измерений обязаны иметь входы слоя RNN? Что представляет каждое измерение? Как насчет его выходов?
3. Если вы хотите построить глубокую сеть RNN типа "последовательность в последовательность", то какие слои RNN должны иметь

`return_sequences=True?` Что насчет сети RNN типа “последовательность в вектор”?

4. Предположим, что у вас есть ежедневный одномерный временной ряд, и вы хотите вырабатывать прогнозы для нескольких последующих дней. Какая архитектура RNN должна использоваться?
5. В чем заключаются основные затруднения при обучении сетей RNN? Как с ними можно справиться?
6. Можете ли вы сделать набросок архитектуры ячейки LSTM?
7. По какой причине вы захотели бы применять одномерные сверточные слои в сети RNN?
8. Какую архитектуру нейронной сети вы могли бы использовать для классификации видеороликов?
9. Обучите модель классификации для набора данных SketchRNN, доступного в рамках проекта TensorFlow Datasets.
10. Загрузите набор данных с хоралами Баха (<https://archive.ics.uci.edu/ml/datasets/Bach+Chorales>) и распакуйте его. Он состоит из 382 хоралов, сочиненных Иоганном Себастьяном Бахом. Каждый хорал имеет длину от 100 до 640 временных шагов, а каждый временной шаг содержит 4 целых числа, где каждое целое число соответствует индексу ноты на фортепьяно (кроме значения 0, которое означает, что никакая нота не воспроизводится). Обучите модель (рекуррентную, сверточную или обе), которая будет способна прогнозировать следующий временной шаг (четыре ноты), располагая последовательностью временных шагов из хорала. Затем используйте обученную модель для генерации музыки в стиле Баха, по одной ноте за раз. Для этого вы можете предоставить модели начало хорала и предложить ей спрогнозировать следующий временной шаг, затем добавить такой временной шаг во входную последовательность и запросить у модели следующую ноту и т.д. Также обязательно ознакомьтесь с моделью Coconet от Google (<https://homl.info/coconet>), которая применялась при создании дудла Google, посвященного дню Иоганна Себастьяна Баха (<https://www.google.com/doodles/celebrating-johann-sebastian-bach>).

Решения приведенных упражнений доступны в приложении А.

# Обработка естественного языка с помощью рекуррентных нейронных сетей и внимания

Когда Аллан Тьюринг придумал свой знаменитый тест Тьюринга (<https://homl.info/turingtest>)<sup>1</sup> в 1950 году, его целью была оценка способности машины соответствовать человеческому интеллекту. Он мог бы проверить многие вещи, такие как возможность узнавать кошек на фотографиях, играть в шахматы, сочинять музыку или выбраться из лабиринта, но, что интересно, все-таки выбрал лингвистическую задачу. В частности, Тьюринг изобрел чатбот, который способен обмануть собеседника, заставив его думать о том, что он человек<sup>2</sup>. У этого теста есть слабые стороны: набор жестко задированных правил может обмануть ничего не подозревающих или наивных людей (например, машина могла бы предоставлять заготовленные размытые ответы, встречая определенные ключевые слова; она могла бы притвориться, что шутит или пьяна, чтобы ее странные ответы прошли; или она могла бы избегать ответа на сложные вопросы, взамен задавая встречные вопросы), а многие аспекты человеческого интеллекта полностью игнорируются (скажем, возможность интерпретировать невербальное общение вроде мимики или выучить ручную задачу). Но тест Тьюринга подчеркивает тот факт, что овладение языком является, вероятно, самой замечательной познавательной

<sup>1</sup> Аллан Тьюринг, *Computing Machinery and Intelligence* (Вычислительные машины и разум), *Mind* 49 (1950 г.): с. 433–460.

<sup>2</sup> Конечно, слово “чатбот” появилось намного позже. Тьюринг назвал свой тест имитационной игрой: машина А и человек Б общаются с допрашивающим человеком В через текстовые сообщения; допрашивающий задает вопросы, чтобы выяснить, кто из них машина (А или Б). Машина проходит тест, если ей удаётся обмануть допрашивающего, в то время как человек Б должен пытаться помочь допрашивающему.

способностью человека разумного. Можем ли мы построить машину, которая сумеет читать и записывать на естественном языке?

Распространенный подход к решению задач с естественным языком предусматривает использование рекуррентных нейронных сетей. По этой причине мы продолжим исследование сетей RNN (введенных в главе 15) и начнем с *символьной сети RNN* (*character RNN*), обученной прогнозированию следующего символа в предложении. Она позволит нам генерировать оригинальный текст, и в процессе мы увидим, как построить объект `Dataset` из TensorFlow на очень длинной последовательности. Сначала мы будем применять *сеть RNN без запоминания состояния* (которая на каждой итерации обучается на случайных порциях текста без какой-либо информации об остальной части текста). Затем мы построим *сеть RNN с запоминанием состояния* (которая предохраняет скрытое состояние между итерациями обучения и продолжает чтение там, где она его оставила, позволяя узнавать более длинные шаблоны). Далее мы создадим сеть RNN для проведения смыслового анализа (например, читая рецензии на фильмы и извлекая мнение о фильме того, кто поставил оценку), теперь трактуя предложения как последовательности слов, а не символов. После этого мы покажем, как использовать сети RNN при построении архитектуры “*кодировщик–декодировщик*”, способной выполнять *нейронный машинный перевод* (*neural machine translation* — NMT), для чего будем применять API-интерфейс `seq2seq`, предлагаемый проектом TensorFlow Addons.

Во второй части главы мы взглянем на механизмы внимания (*attention mechanism*). Как следует из их названия, они являются компонентами нейронной сети, которые учатся выбирать ту часть входов, на которой должна фокусироваться остальная часть модели на каждом временном шаге. Первым делом мы посмотрим, как с использованием внимания поднять эффективность архитектуры “*кодировщик–декодировщик*”, основанной на сетях RNN, а затем полностью отбросим сети RNN и взглянем на очень успешную архитектуру, основанную только на внимании, которая называется “*Преобразователь*” (*Transformer*). Наконец, мы ознакомимся с рядом наиболее важных достижений в области обработки естественного языка, датируемых 2018 и 2019 годами, включая невероятно мощные языковые модели, такие как GPT-2 и BERT, которые обе основаны на преобразователях.

Давайте начнем с простой и забавной модели, которая может писать как Шекспир (хорошо, отчасти).

# Генерация шекспировского текста с использованием символьной сети RNN

В известной записи в блоге 2015 года (<https://homl.info/charrnn>), озаглавленной “The Unreasonable Effectiveness of Recurrent Neural Networks” (Чрезмерная эффективность рекуррентных нейронных сетей), Андрей Карпатый показал, как обучить сеть RNN прогнозированию следующего символа в последовательности. Такая символьная сеть RNN (*Char-RNN*) затем может применяться для генерации нового текста по одному символу за раз. Ниже приведена небольшая выборка из текста, сгенерированного моделью Char-RNN после ее обучения на всей работе Шекспира:

**PANDARUS:**

Alas, I think he shall be come approached and the day  
When little strain would be attain'd into being never fed,  
And who is but a chain and subjects of his death,  
I should not sleep.

**ПАНДАР:**

Увы, я думаю, что он придет, и наступит день,  
Когда пищи станет так мало, чтобы никогда не питаться,  
И те, кто всего лишь цепь и подданные его смерти,  
Я не должен спать.

Нельзя назвать шедевром, но все-таки впечатляет, что модель сумела выучить слова, грамматику, надлежащую пунктуацию и многое другое, просто научившись прогнозировать следующее слово в предложении. Давайте посмотрим, как строить сеть Char-RNN, шаг за шагом, начиная с создания набора данных.

## Создание обучающего набора данных

Прежде всего, загрузим всю работу Шекспира с использованием удобной функции `get_file()` библиотеки Keras и загружая данные из проекта Char-RNN Андрея Карпатого (<https://github.com/karpathy/char-rnn>):

```
shakespeare_url = "https://homl.info/shakespeare" # сокращенный URL
filepath = keras.utils.get_file("shakespeare.txt", shakespeare_url)
with open(filepath) as f:
    shakespeare_text = f.read()
```

Далее мы должны закодировать каждый символ в виде целого числа. Один из вариантов предусматривает создание специального слоя предварительной обработки, как мы поступали в главе 13. Но в данном случае проще применить класс `Tokenizer` библиотеки Keras. Первым делом понадобится выполнить подгонку лексического анализатора к тексту; он отыщет все символы, используемые в тексте, и сопоставит их с различными идентификаторами символов, принимающими значения от 1 до количества несовпадающих символов (поскольку значения начинаются не с 0, мы можем применять 0 для маскирования, как будет показано позже в главе):

```
tokenizer = keras.preprocessing.text.Tokenizer(char_level=True)
tokenizer.fit_on_texts([shakespeare_text])
```

Мы устанавливаем `char_level=True`, чтобы получить кодировку на уровне символов, а не стандартную на уровне слов. Обратите внимание, что по умолчанию лексический анализатор преобразует текст в нижний регистр (но вы можете установить `lower=False`, если преобразование нежелательно). Теперь лексический анализатор способен кодировать предложение (или список предложений) в список идентификаторов символов и обратно, к тому же он сообщает количество несовпадающих символов и общее число символов в тексте:

```
>>> tokenizer.texts_to_sequences(["First"])
[[20, 6, 9, 8, 3]]
>>> tokenizer.sequences_to_texts([[20, 6, 9, 8, 3]])
['f i r s t']
>>> max_id = len(tokenizer.word_index)      # количество несовпадающих
                                            # символов
>>> dataset_size = tokenizer.document_count  # общее число символов
```

Давайте закодируем полный текст, чтобы каждый символ был представлен своим идентификатором (мы вычитаем 1 для получения идентификаторов от 0 до 38, а не от 1 до 39):

```
[encoded] =
    np.array(tokenizer.texts_to_sequences([shakespeare_text])) - 1
```

Прежде чем продолжить, нам необходимо расщепить набор данных на обучающий, проверочный и испытательный наборы. Так как мы не можем просто перетасовать все символы в тексте, возникает вопрос: каким образом расщеплять последовательный набор данных?

## Расщепление последовательного набора данных

Очень важно избегать любого перекрытия между обучающим, проверочным и испытательным наборами. Например, мы можем взять первые 90% текста для обучающего набора, следующие 5% для проверочного набора и последние 5% для испытательного набора. Также неплохо оставить зазоры между этими наборами во избежание риска частичного перекрытия абзаца между двумя наборами.

При работе с временными рядами вы обычно расщепляете во времени: скажем, вы можете выбрать годы 2000–2012 для обучающего набора, годы 2013–2015 для проверочного набора и годы 2016–2018 для испытательного набора. Тем не менее, в некоторых случаях у вас будет возможность расщеплять по другим измерениям, что обеспечит более длительный период времени для обучения. Например, при наличии данных о финансовом состоянии 10 000 компаний за 2000–2018 годы вы сможете расщепить их по разным компаниям. Однако вполне вероятно, что многие из них сильно связаны (скажем, целые секторы экономики могут расти или снижаться совместно). Если у вас есть компании, связанные через обучающий набор и испытательный набор, тогда испытательный набор окажется не настолько полезным, т.к. его оценка ошибки обобщения будет оптимистически смещённой.

Таким образом, часто безопаснее расщеплять во времени, но при этом неявно допускается, что шаблоны, которые сеть RNN могла узнать в прошлом (в обучающем наборе) будут по-прежнему существовать в будущем. Другими словами, мы предполагаем, что временной ряд является *стационарным* (по крайней мере, в широком смысле)<sup>3</sup>. Для многих временных рядов такое допущение будет разумным (например, оно подойдет для химических реакций, поскольку законы химии не меняются ежедневно), но для многих других — нет (скажем, общеизвестно, что финансовые рынки не стационарны, т.к. шаблоны исчезают, как только биржевые маклеры замечают их и начинают эксплуатировать). Для выяснения, действительно ли временной ряд в достаточной мере стационарен, вы можете вычертить график ошибок на проверочном наборе с течением времени. Если модель работает намного лучше в

<sup>3</sup> По определению средняя величина, дисперсия и автокорреляции (т.е. корреляции между значениями во временном ряде, отделенными заданным интервалом) стационарного временного ряда не изменяются с течением времени. Определение довольно ограничивающее; например, оно исключает временные ряды с тенденциями или циклическими шаблонами. Сети RNN в большей степени толерантны в том, что могут узнавать тенденции и циклические шаблоны.

первой части проверочного набора, чем в последней части, тогда временной ряд может оказаться недостаточно стационарным, и модель лучше обучать на более коротком промежутке времени.

Короче говоря, расщепление временного ряда на обучающий, проверочный и испытательный наборы — работа непростая, и способ ее выполнения будет сильно зависеть от имеющейся задачи.

А теперь вернемся к Шекспиру! Давайте возьмем первые 90% текста для обучающего набора (сохранив остаток для проверочного и испытательного наборов) и создадим объект `tf.data.Dataset`, который будет возвращать из этого набора все символы друг за другом:

```
train_size = dataset_size * 90 // 100
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
```

## Разрезание последовательного набора данных на множество окон

В настоящий момент обучающий набор содержит единственную последовательность, включающую более миллиона символов, поэтому мы не можем обучать нейронную сеть непосредственно на нем: сеть RNN была бы эквивалентом глубокой сети, имеющей свыше миллиона слоев, и мы располагали бы одним (очень длинным) образцом для ее обучения. Взамен мы будем использовать метод `window()` набора данных для преобразования такой длинной последовательности символов во множество окон текста меньших размеров. Каждый образец в наборе данных окажется довольно короткой подстрокой полного текста, а сеть RNN будет разворачиваться только по длине этих подстрок. Прием называется *укороченным обратным распространением во времени* (*truncated backpropagation through time*). Давайте вызовем метод `window()`, чтобы создать набор данных из коротких окон текста:

```
n_steps = 100
window_length = n_steps + 1 # цель = вход, сдвинутый на 1 символ вперед
dataset = dataset.window(window_length, shift=1, drop_remainder=True)
```



Вы можете попробовать настроить `n_steps`: сети RNN легче обучать на более коротких входных последовательностях, но, разумеется, сеть RNN не сумеет узнавать шаблоны длиннее `n_steps`, так что не делайте значение `n_steps` слишком малым.

По умолчанию метод `window()` создает неперекрывающиеся окна, но для получения как можно более крупного обучающего набора мы применяем

`shift=1`, в результате чего первое окно содержит символы с 0 до 100, второе — символы с 1 до 101 и т.д. Для гарантирования того, что все окна имеют длину в точности 101 символ (позволяя нам создавать пакеты безо всякого дополнения), мы устанавливаем `drop_remainder=True` (иначе последние 100 окон будут содержать 100 символов, 99 символов и так далее вплоть до 1 символа).

Метод `window()` создает набор данных, содержащий окна, каждое из которых также представлено как набор данных. Мы получаем так называемый *набор данных с вложениями*, похожий на список списков. Это удобно, когда необходимо трансформировать каждое окно, вызывая его методы набора данных (скажем, для их тасования или группирования в пакеты). Тем не менее, использовать набор данных с вложениями напрямую для обучения нельзя, т.к. наша модель на входе ожидает тензоры, а не наборы данных. Таким образом, нам придется вызвать метод `flat_map()`: он преобразует набор данных с вложениями в *плоский набор данных* (не содержащий в себе наборов данных). Например, пусть `{1, 2, 3}` представляет набор данных, содержащий последовательность тензоров 1, 2 и 3. Выровняв набор данных с вложениями `[[1, 2], [3, 4, 5, 6]]`, вы получите плоский набор данных `[1, 2, 3, 4, 5, 6]`. Кроме того, метод `flat_map()` принимает в качестве аргумента функцию, которая позволяет перед выравниванием трансформировать каждый набор данных из набора данных с вложениями. Скажем, если вы передадите методу `flat_map()` функцию `lambda ds: ds.batch(2)`, то она трансформирует набор данных с вложениями `[[1, 2], [3, 4, 5, 6]]` в плоский набор данных `[[1, 2], [3, 4], [5, 6]]`, т.е. набор данных из тензоров размера 2. Имея это в виду, мы готовы выровнять наш набор данных:

```
dataset = dataset.flat_map(lambda window: window.batch(window_length))
```

Обратите внимание, что мы вызываем `batch(window_length)` на каждом окне: поскольку все окна имеют в точности длину `window_length`, для каждого из них мы получим одиночный тензор. Теперь набор данных содержит непрерывно следующие друг за другом окна, каждое по 101 символу. Так как градиентный спуск работает лучше всего, когда образцы в обучающем наборе являются независимыми и имеющими идентичное распределение (см. главу 4), окна понадобится перетасовать. Затем мы можем сгруппировать окна в пакеты и отделить входы (первые 100 символов) от цели (последний символ):

```
batch_size = 32
dataset = dataset.shuffle(10000).batch(batch_size)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
```

На рис. 16.1 подытоживаются описанные до сих пор шаги подготовки набора данных (здесь показаны окна длиной 11, а не 101, и размер пакета составляет 3 вместо 32).

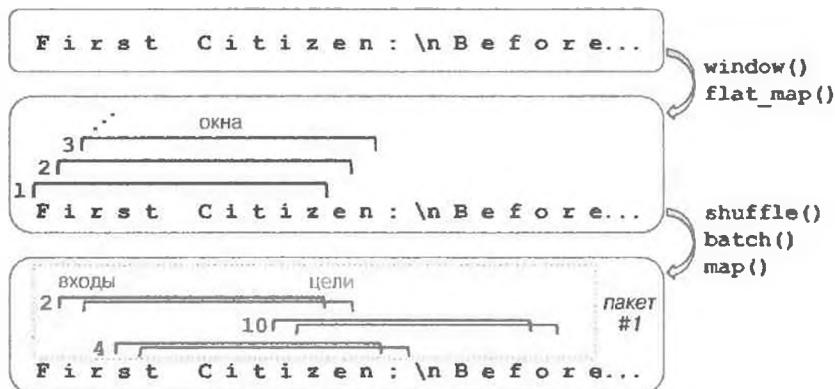


Рис. 16.1. Подготовка набора данных из перетасованных окон

Как обсуждалось в главе 13, категориальные входные признаки должны обычно кодироваться как векторы в унитарном коде или как вложения. Мы будем кодировать каждый символ с применением вектора в унитарном коде из-за того, что несовпадающих символов относительно мало (всего лишь 39):

```
dataset = dataset.map(  
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
```

В заключение нам потребуется добавить предварительную выборку:

```
dataset = dataset.prefetch(1)
```

Вот и все! Подготовка набора данных была самой трудной частью. А теперь создадим модель.

## Построение и обучение модели Char-RNN

Чтобы спрогнозировать следующий символ на основе 100 предшествующих символов, мы можем использовать сеть RNN с 2 слоями GRU из 128 элементов каждый и 20%-ным отключением на входах (`dropout`) и скрытых состояниях (`recurrent_dropout`). При необходимости позже мы можем подстроить указанные гиперпараметры. Выходной слой представляет собой распределенный во времени слой `Dense`, подобный тому, что мы видели в главе 15. На этот раз он обязан иметь 39 элементов (`max_id`), т.к. в тексте

присутствуют 39 несовпадающих символов, а мы хотим выдавать вероятность для каждого возможного символа (на каждом временном шаге).

На каждом временном шаге выходные вероятности должны в сумме составлять 1, поэтому мы применяем к выходам слоя Dense многопеременную функцию активации. Затем мы можем скомпилировать модель, используя потерю "sparse\_categorical\_crossentropy" и оптимизатор Adam. Наконец-то мы готовы обучить модель в течение нескольких эпох (в зависимости от оборудования процесс может занять много часов):

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True,
                      input_shape=[None, max_id],
                      dropout=0.2, recurrent_dropout=0.2),
    keras.layers.GRU(128, return_sequences=True,
                      dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
model.compile(loss="sparse_categorical_crossentropy",
              optimizer="adam")
history = model.fit(dataset, epochs=20)
```

## Использование модели Char-RNN

Теперь у нас есть модель, которая способна прогнозировать следующий символ в тексте, написанном Шекспиром. Чтобы передать ей какой-то текст, мы сначала должны его предварительно обработать, как поступали ранее, а потому давайте создадим для этого небольшую функцию:

```
def preprocess(texts):
    X = np.array(tokenizer.texts_to_sequences(texts)) - 1
    return tf.one_hot(X, max_id)
```

Воспользуемся моделью для прогнозирования следующей буквы в примере текста:

```
>>> X_new = preprocess(["How are yo"])
>>> Y_pred = model.predict_classes(X_new)
>>> tokenizer.sequences_to_texts(Y_pred + 1)[0][-1]
# первое предложение, последний символ
'u'
```

Успешно! Модель угадала правильно. Давайте применим ее для генерации нового текста.

## Генерирование поддельного шекспировского текста

Чтобы сгенерировать новый текст, используя модель Char-RNN, мы могли бы передать ей какой-то текст, заставить модель выработать прогноз наиболее вероятной следующей буквы, добавить эту букву в конец текста, затем предоставить модели расширенный текст для оценки следующей буквы и т.д. Но на практике такой подход часто приводит к тому, что те же самые слова повторяются снова и снова. Взамен мы можем выбирать следующий символ случайнным образом с вероятностью, равной оценке вероятности, с применением функции `tf.random.categorical()` из TensorFlow. В результате будет генерироваться более разнообразный и интересный текст. Функция `categorical()` производит выборку случайных индексов классов для заданных логарифмических вероятностей классов (логитов). Для достижения большего контроля над разнообразием генерируемого текста мы можем делить логиты на число, называемое *температурой* (*temperature*), которое допускает любую желаемую подстройку: температура, близкая к 0, будет благоприятствовать высоковероятным символам, тогда как очень высокая температура обеспечит назначение всем символам равной вероятности. В показанной ниже функции `next_char()` такой подход используется для выбора следующего символа с целью его добавления во входной текст:

```
def next_char(text, temperature=1):
    X_new = preprocess([text])
    y_proba = model.predict(X_new)[0, -1:, :]
    rescaled_logits = tf.math.log(y_proba) / temperature
    char_id =
        tf.random.categorical(rescaled_logits, num_samples=1) + 1
    return tokenizer.sequences_to_texts(char_id.numpy())[0]
```

Далее мы можем написать небольшую функцию, которая будет многократно вызывать `next_char()` для получения следующего символа и присоединять его к заданному тексту:

```
def complete_text(text, n_chars=50, temperature=1):
    for _ in range(n_chars):
        text += next_char(text, temperature)
    return text
```

Теперь мы готовы к генерации какого-нибудь текста! Давайте опробуем разные температуры:

```
>>> print(complete_text("t", temperature=0.2))
```

```
the belly the great and who shall be the belly the
>>> print(complete_text("w", temperature=1))
thing? or why you gremio.
who make which the first
>>> print(complete_text("w", temperature=2))
th no cce:
yeolg-hormer firi. a play asks.
fol rusb
```

Очевидно, наша модель шекспировского текста лучше всего работает при температуре, близкой к 1. Для генерации более убедительного текста можете попробовать создать больше слоев GRU и больше нейронов на слой, дальше обучать и добавить некоторую регуляризацию (скажем, установить recurrent\_dropout=0.3 в слоях GRU). Кроме того, в текущий момент модель неспособна узнавать шаблоны длиннее n\_steps, равного всего лишь 100 символам. Вы могли бы попытаться увеличить окно, но тогда затруднилось бы обучение, вдобавок даже ячейки LSTM и GRU не могут обрабатывать очень длинные последовательности. В качестве альтернативы можно применять сеть RNN с запоминанием состояния.

## Сеть RNN с запоминанием состояния

До сих пор мы использовали только сети RNN без запоминания состояния: на каждом цикле обучения модель начинает со скрытого состояния, заполненного нулями, затем на каждом временном шаге обновляет это состояние и после последнего временного шага отбрасывает его, т.к. больше в нем не нуждается. А что, если мы сообщим сети RNN о необходимости предохранять финальное состояние после обработки одного пакета обучающих образцов и задействовать его как начальное состояние для следующего пакета? Таким способом модель может узнавать установленные шаблоны, несмотря на их обратное распространение только через короткие последовательности. Это называется *сетью RNN с запоминанием состояния*. Посмотрим, как ее построить.

Прежде всего, обратите внимание на то, что сеть RNN с запоминанием состояния имеет смысл лишь в том случае, если каждая входная последовательность в пакете начинается именно там, где прервалась соответствующая последовательность в предыдущем пакете. В итоге первым действием, которое нам предстоит предпринять для построения сети RNN с запоминанием состояния, будет применение упорядоченных и неперекрывающихся входных последовательностей (вместо перетасованных и перекрывающихся по-

следовательностей, используемых при обучении сетей RNN без запоминания состояния). Тогда во время создания объекта Dataset при вызове метода `window()` мы обязаны применять `shift=n_steps` (а не `shift=1`). Кроме того, вполне очевидно мы не должны вызывать метод `shuffle()`. К сожалению, группирование в пакеты при подготовке набора данных для сети RNN с запоминанием состояния гораздо сложнее, чем для сети RNN без запоминания состояния. В действительности, если бы мы вызвали `batch(32)`, то 32 идущих друг за другом окна попали бы в тот же самый пакет и следующий пакет не продолжил бы каждое окно после того, как оно закончилось. Первый пакет содержал бы окна с 1 по 32, а второй — окна с 33 по 64, поэтому если вы возьмете, скажем, первое окно каждого пакета (т.е. окна 1 и 33), то заметите, что они не следуют друг за другом. Простейшее решение проблемы предусматривает использование “пакетов”, содержащих единственное окно:

```
dataset = tf.data.Dataset.from_tensor_slices(encoded[:train_size])
dataset = dataset.window(window_length, shift=n_steps,
                         drop_remainder=True)
dataset = dataset.flat_map(lambda window: window.batch(window_length))
dataset = dataset.batch(1)
dataset = dataset.map(lambda windows: (windows[:, :-1], windows[:, 1:]))
dataset = dataset.map(
    lambda X_batch, Y_batch: (tf.one_hot(X_batch, depth=max_id), Y_batch))
dataset = dataset.prefetch(1)
```

Первый шаг иллюстрируется на рис. 16.2.

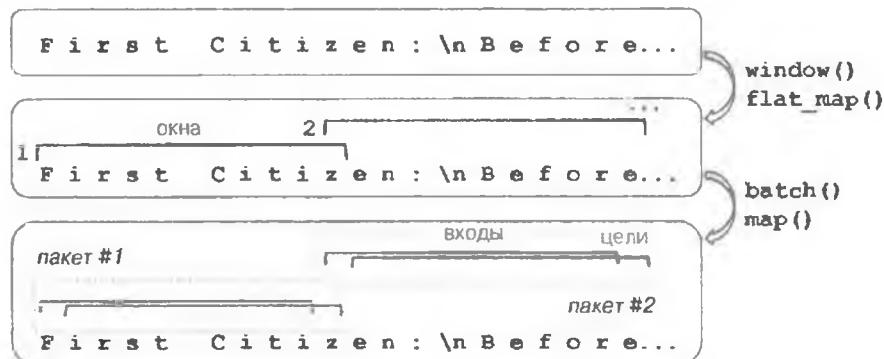


Рис. 16.2. Подготовка набора данных из следующих друг за другом фрагментов последовательности для сети RNN с запоминанием состояния

Группирование в пакеты является более сложным, но отнюдь не невозможным. Например, мы могли бы разрезать текст Шекспира на 32 текста

равной длины, создать для каждого из них по набору данных с идущими друг за другом входными последовательностями и в заключение применить `tf.train.Dataset.zip(datasets).map(lambda *windows: tf.stack(windows))`, чтобы создать правильные идущие друг за другом пакеты, причем  $n$ -тая входная последовательность в пакете начинается в точности, где закончилась  $n$ -тая входная последовательность в предыдущем пакете (полный код находится в тетради Jupiter для настоящей главы).

А теперь давайте создадим сеть RNN с запоминанием состояния. Во-первых, нам необходимо устанавливать `stateful=True` при создании каждого рекуррентного слоя. Во-вторых, сети RNN с запоминанием состояния должен быть известен размер пакета (поскольку она предохраняет состояние для каждой входной последовательности в пакете), так что в первом слое потребуется установить аргумент `batch_input_shape`. Обратите внимание, что мы можем не указывать второе измерение, т.к. входы способны иметь любую длину:

```
model = keras.models.Sequential([
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2,
                     batch_input_shape=[batch_size, None, max_id]),
    keras.layers.GRU(128, return_sequences=True, stateful=True,
                     dropout=0.2, recurrent_dropout=0.2),
    keras.layers.TimeDistributed(keras.layers.Dense(max_id,
                                                    activation="softmax"))
])
```

В конце каждой эпохи нам нужно сбросить состояние, прежде чем возвращаться в начало текста. Для этого мы можем использовать простой обратный вызов:

```
class ResetStatesCallback(keras.callbacks.Callback):
    def on_epoch_begin(self, epoch, logs):
        self.model.reset_states()
```

Далее мы компилируем и подгоняем модель (для большего количества эпох, потому что каждая эпоха намного короче предшествующих, и есть только один образец на пакет):

```
model.compile(loss="sparse_categorical_crossentropy",
               optimizer="adam")
model.fit(dataset, epochs=50, callbacks=[ResetStatesCallback()])
```



После того, как модель обучена, ее можно применять в целях выработывания прогнозов для пакетов того же размера, который использовался во время обучения. Чтобы обойти такое ограничение, создайте идентичную модель без запоминания состояния и скопируйте в нее веса модели с запоминанием состояния.

Построив модель уровня символов, самое время взглянуть на модели уровня слов и заняться распространенной задачей обработки естественного языка: *смысловым анализом* (*sentiment analysis*). В процессе мы выясним, как обрабатывать последовательности переменной длины с применением маскирования.

## Смысловой анализ

Если набор данных MNIST является первой учебной задачей компьютерного зрения, то набор данных IMDb с рецензиями на фильмы представляет собой первую учебную задачу обработки естественного языка. Он состоит из 50 000 рецензий на английском языке (25 000 для обучения, 25 000 для испытаний), извлеченных из знаменитой базы данных фильмов в Интернете (<https://imdb.com/>), наряду с простой двоичной целью для каждой рецензии, которая указывает, отрицательная она (0) или же положительная (1). Подобно MNIST набор данных рецензий IMDb популярен по двум веским причинам: он достаточно прост, чтобы с ним можно было справиться на лэптопе за разумное время, но достаточно сложен, чтобы быть забавным и полезным. В Keras предлагается простая функция для его загрузки:

```
>>> (X_train, y_train), (X_test, y_test) =  
        keras.datasets.imdb.load_data()  
>>> X_train[0][:10]  
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

А где рецензии на фильмы? Легко заметить, что набор данных уже предварительно обработан: X\_train состоит из списка рецензий, каждая из которых представлена в виде массива NumPy целых чисел, где каждое число представляет слово. Все знаки пунктуации были удалены, затем слова были преобразованы в нижний регистр, разделены по пробелам и в заключение проиндексированы по частоте (так что меньшие индексы соответствуют часто встречающимся словам). Целые числа 0, 1 и 2 являются особыми:

они представляют соответственно маркер дополнения, маркер начала последовательности (*start-of-sequence* — `SOS`) и неизвестные слова. Если вы хотите визуализировать рецензию, тогда можете декодировать ее примерно так:

```
>>> word_index = keras.datasets.imdb.get_word_index()  
>>> id_to_word = {id_ + 3: word for word, id_ in word_index.items()}  
>>> for id_, token in enumerate("<pad>", "<sos>", "<unk>"):  
...     id_to_word[id_] = token  
...  
>>> " ".join([id_to_word[id_] for id_ in X_train[0][:10]])  
'<sos> this film was just brilliant casting location scenery story'
```

В реальном проекте вам пришлось бы самостоятельно предварительно обрабатывать текст с использованием того же самого класса `Tokenizer`, что и ранее, но на этот раз устанавливая `char_level=False` (принимается по умолчанию). При кодировании слов он отфильтровывает массу символов, включая большинство пунктуации, разрывы строк и табуляции (что можно изменить, устанавливая аргумент `filters`). Самое главное, класс `Tokenizer` применяет пробелы для идентификации границ слов. Это нормально для английского и многих других систем письма (письменных языков), которые используют пробелы между словами, но не все системы письма применяют пробелы таким способом. В китайском языке не используются пробелы между словами, во вьетнамском пробелы применяются даже внутри слов, а в языках вроде немецкого множество слов часто соединяются вместе без пробелов. Даже в английском языке пробелы не всегда будут лучшим способом разбиения текста: подумайте о “San Francisco” или “#ILoveDeepLearning”.

К счастью, есть варианты получше! В статье 2018 года (<https://homl.info/subword>)<sup>4</sup> Таку Кудо представил методику обучения без учителя для разбиения на лексемы текста и обратного процесса на уровне частей слов в независимом от языка стиле, трактуя пробелы подобно остальным символам. При таком подходе, даже если ваша модель сталкивается со словом, которое никогда не видела ранее, она все еще способна разумно предположить, что оно означает. Например, во время обучения модель могла не встречать слово “smartest” (“умнейший”), но возможно она узнала слово “smart” (“умный”)

<sup>4</sup> Таку Кудо, *Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates* (Регуляризация по частям слов: улучшение моделей нейронного машинного перевода с помощью множества кандидатов-частей слов), препринт arXiv:1804.10959 (2018 г.).

и также то, что суффикс “est” означает “самый”, поэтому способна вывести смысл слова “smartest”. Проект *SentencePiece* от Google (<https://github.com/google/sentencepiece>) предоставляет реализацию с открытым кодом, описанную в статье 2018 года (<https://hml.info/sentencepiece>)<sup>5</sup> Таку Кудо и Джона Ричардсона.

Еще один вариант был предложен в более ранней статье Рико Сеннириха и др. (<https://hml.info/rarewords>)<sup>6</sup>, где исследовались другие способы создания кодировок частей слов (например, использование кодировки *парами байтов* (*byte pair encoding*)). И последнее, но не менее важное — в июне 2019 года команда разработчиков TensorFlow выпустила библиотеку *TF.Text* (<https://hml.info/tftext>), которая реализует разнообразные стратегии лексического анализа, включая *WordPiece* (<https://hml.info/wordpiece>)<sup>7</sup> (вариант кодировки парами байтов).

Если вы хотите развернуть свою модель на мобильном устройстве или в веб-браузере, но не желаете каждый раз писать отличающуюся функцию предварительной обработки, тогда предварительную обработку придется организовывать с применением только операций TensorFlow, чтобы она могла быть включена в саму модель. Давайте посмотрим, как это делается. Для начала загрузим исходные рецензии IMDb в виде текста (байтовых строк), воспользовавшись проектом TensorFlow Datasets (представленным в главе 13):

```
import tensorflow as tf
datasets, info = tfds.load("imdb_reviews", as_supervised=True,
                           with_info=True)
train_size = info.splits["train"].num_examples
```

<sup>5</sup> Таку Кудо и Джон Ричардсон, *SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing* (*SentencePiece: простой и независимый от языка лексический анализатор и сборщик на уровне частей слов для обработки текста нейронными сетями*), препринт arXiv:1808.06226 (2018 г.).

<sup>6</sup> Рико Сеннирих и др., *Neural Machine Translation of Rare Words with Subword Units* (Нейронный машинный перевод редких слов с помощью элементов на основе частей слов), *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics 1* (2016 г.): с. 1715–1725.

<sup>7</sup> Юнхуэй Ву и др., *Google’s Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation* (Система нейронного машинного перевода Google: преодоление разрыва между человеческим и машинным переводом), препринт arXiv:1609.08144 (2016 г.).

Далее напишем функцию предварительной обработки:

```
def preprocess(X_batch, y_batch):
    X_batch = tf.strings.substr(X_batch, 0, 300)
    X_batch = tf.strings.regex_replace(X_batch, b"<br\\s*/?>", b" ")
    X_batch = tf.strings.regex_replace(X_batch, b"[^a-zA-Z]", b" ")
    X_batch = tf.strings.split(X_batch)
    return X_batch.to_tensor(default_value=b"<pad>"), y_batch
```

Функция начинается с усечения рецензий, оставляя в каждой только первые 300 символов: это ускорит обучение и не особо сильно повлияет на эффективность, потому что определить, положительная рецензия или нет, обычно можно по начальным одному-двум предложениям. Затем она с помощью регулярных выражений заменяет пробелами дескрипторы `<br />`, любые символы, отличающиеся от букв, и кавычки.

Например, текст "Well, I can't<br />" превращается в "Well I can't". В заключение функция `preprocess()` разбивает рецензии по пробелам, что возвращает зубчатый тензор, и преобразует его в плотный тензор, дополняя все рецензии маркером дополнения "`<pad>`", чтобы они имели одну и ту же длину.

Далее нам необходимо создать словарь, для чего потребуется однократно пройти по всему обучающему набору, применяя функцию `preprocess()`, и посредством объекта `Counter` подсчитать количество вхождений каждого слова:

```
from           import Counter
vocabulary = Counter()
for X_batch, y_batch in datasets["train"].batch(32).map(preprocess):
    for review in X_batch:
        vocabulary.update(list(review.numpy()))
```

Давайте взглянем на три наиболее часто встречающихся слова:

```
>>> vocabulary.most_common()[:3]
[(b'<pad>', 215797), (b'the', 61137), (b'a', 38564)]
```

Великолепно! Однако возможно, что для достижения приемлемой эффективности нашей модели не нужно знать все слова в словаре, поэтому давайте усечем словарь, оставив только 10 000 самых распространенных слов:

```
vocab_size = 10000
truncated_vocabulary = [
    word for word, count in vocabulary.most_common()[:vocab_size]]
```

Теперь мы должны добавить шаг предварительной обработки, чтобы заменить каждое слово его идентификатором (т.е. индексом в словаре). В частности, как мы поступали в главе 13, мы создадим таблицу поиска, использующую 1000 участков вне словаря (oov):

```
words = tf.constant(truncated_vocabulary)
word_ids = tf.range(len(truncated_vocabulary), dtype=tf.int64)
vocab_init = tf.lookup.KeyValueTensorInitializer(words, word_ids)
num_oov_buckets = 1000
table = tf.lookup.StaticVocabularyTable(vocab_init, num_oov_buckets)
```

Затем мы можем применять созданную таблицу для нахождения идентификаторов нескольких слов:

```
>>> table.lookup(tf.constant(
...     [b"This movie was faaaaaantastic".split()])
<tf.Tensor: [...], dtype=int64, numpy=array([[ 22, 12, 11, 10054]])>
```

Обратите внимание, что слова “this”, “movie” и “was” нашлись в таблице, так что их идентификаторы меньше 10 000, в то время как слово “faaaaaantastic” не было найдено, а потому оно отображается на один из участков oov с идентификатором, большим или равным 10 000.



Библиотека TF Transform (см. главу 13) предоставляет ряд удобных функций для работы с такими словарями. Скажем, возьмем функцию `tft.compute_and_apply_vocabulary()`: она проходит по набору данных, чтобы отыскать все несовпадающие слова и построить словарь, после чего генерирует операции TF, требующиеся для кодирования каждого слова с использованием этого словаря.

Итак, все готово к созданию финального обучающего набора. Мы группируем рецензии в пакеты, преобразуем их в короткие последовательности слов с применением функции `preprocess()`, кодируем полученные слова, используя простую функцию `encode_words()`, которая задействует только что построенную таблицу поиска, и в заключение выполняем предварительную выборку следующего пакета:

```
def encode_words(X_batch, y_batch):
    return table.lookup(X_batch), y_batch

train_set = datasets["train"].batch(32).map(preprocess)
train_set = train_set.map(encode_words).prefetch(1)
```

Наконец, мы можем создать модель и обучить ее:

```
embed_size = 128
model = keras.models.Sequential([
    keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size,
                           input_shape=[None]),
    keras.layers.GRU(128, return_sequences=True),
    keras.layers.GRU(128),
    keras.layers.Dense(1, activation="sigmoid")
])
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])
history = model.fit(train_set, epochs=5)
```

Первым слоем является *Embedding*, который будет преобразовывать идентификаторы слов во вложения (введенные в главе 13). Матрица вложений должна иметь одну строку на идентификатор слова (`vocab_size + num_oov_buckets`) и один столбец на измерение вложений (в рассматриваемом примере применяются 128 измерений, но это гиперпараметр, допускающий подстройку). Несмотря на то что входами модели будут двумерные тензоры формы [размер пакета, количество временных шагов], выходом слоя *Embedding* является трехмерный тензор формы [размер пакета, количество временных шагов, размер вложения].

Оставшаяся часть модели довольно прямолинейна: она состоит из двух слоев *GRU*, второй из которых возвращает только выход последнего временного шага. Выходной слой представляет собой одиночный нейрон, использующий сигмоидальную функцию активации для выдачи оценки вероятности того, что рецензия выражает позитивное отношение к фильму. Затем мы достаточно легко компилируем модель и подгоняем ее к подготовленному ранее набору данных в течение нескольких эпох.

## Маскирование

В том виде, как есть, модель должна будет узнать, что маркеры дополнения необходимо игнорировать. Но данный факт нам уже известен! Почему бы не сообщить модели об игнорировании маркеров дополнения, чтобы она могла сконцентрироваться на данных, которые действительно важны? Решение тривиально: просто добавьте `mask_zero=True` при создании слоя *Embedding*. Это означает, что маркеры дополнения (чьи идентификаторы

равны 0)<sup>8</sup> будут игнорироваться всеми расположеннымми далее слоями. Вот и все!

Слой Embedding создает тензор маски, эквивалентный `K.not_equal(inputs, 0)` (здесь `K = keras.backend`): он представляет собой булевский тензор с такой же формой, как у входов, и он равен `False` везде, где идентификаторы слов являются нулевыми, или `True` в противном случае. Затем тензор маски автоматически распространяется моделью по все последующие слои при условии, что предохраняется измерение времени. Таким образом, в нашем примере оба слоя GRU автоматически получат эту маску, но поскольку второй слой GRU не возвращает последовательности (он возвращает только выход последнего временного шага), маска не будет передаваться слою Dense. Каждый слой может обрабатывать маску по-разному, но в целом они просто игнорируют маскированные временные шаги (т.е. временные шаги, для которых маска равна `False`). Скажем, когда рекуррентный слой сталкивается с маскированным временным шагом, он просто копирует выход из предыдущего временного шага. Если маска распространяется вплоть до выхода (в моделях, которые выдают последовательности, что не касается настоящего примера), тогда она будет применяться также и к потерям, поэтому маскированные временные шаги не будут вносить вклад в потерю (их потеря составит 0).



Слои LSTM и GRU имеют оптимизированную реализацию для графических процессоров, основанную на библиотеке cuDNN от Nvidia. Тем не менее, такая реализация не поддерживает маскирование. Если в вашей модели используется маска, тогда эти слои будут переключаться на (гораздо более медленную) стандартную реализацию. Обратите внимание, что оптимизированная реализация также требует применения стандартных значений для нескольких гиперпараметров: `activation`, `recurrent_activation`, `recurrent_dropout`, `unroll`, `use_bias` и `reset_after`.

Все слои, которые получают маску, обязаны поддерживать маскирование (иначе генерируется исключение). К ним относятся все рекуррентные слои, а

<sup>8</sup> Их идентификаторы равны 0 лишь потому, что они являются самыми часто встречающимися "словами" в наборе данных. Вероятно, было бы неплохо гарантировать, что маркеры дополнения всегда кодируются как 0, даже если они не будут наиболее часто встречающимися.

также слой `TimeDistributed` и ряд других. Любой слой, поддерживающий маскирование, должен иметь атрибут `supports_masking`, равный `True`. Если вы хотите реализовать собственный специальный слой с поддержкой маскирования, тогда к методу `call()` потребуется добавить аргумент `mask` (и вполне очевидно каким-то образом задействовать маску внутри метода). Кроме того, понадобится установить `self.supports_masking = True` в конструкторе. Если ваш слой не начинается со слоя `Embedding`, то вместо него вы можете использовать слой `keras.layers.Masking`: он устанавливает маску в `K.any(K.not_equal(inputs, 0), axis=-1)` и временные шаги, где последнее измерение заполнено нулями, будут маскироваться в последующих слоях (опять-таки при условии, что измерение времени существует).

Применение слоев маскирования и автоматическое распространение маски лучше всего работают для простых моделей `Sequential`. Маскирование не всегда работает для более сложных моделей вроде тех, где необходимо смешивать слои `Conv1D` с рекуррентными слоями. В таких случаях придется явно вычислять маску и передавать ее соответствующим слоям, используя либо API-интерфейс `Functional`, либо API-интерфейс `Subclassing`. Например, показанная модель идентична предыдущей за исключением того, что она построена с применением API-интерфейса `Functional` и маскирование в ней поддерживается вручную:

```
K = keras.backend
inputs = keras.layers.Input(shape=[None])
mask = keras.layers.Lambda(lambda inputs: K.not_equal(inputs, 0))
(inputs)
z = keras.layers.Embedding(vocab_size + num_oov_buckets, embed_size)
(inputs)
z = keras.layers.GRU(128, return_sequences=True)(z, mask=mask)
z = keras.layers.GRU(128)(z, mask=mask)
outputs = keras.layers.Dense(1, activation="sigmoid")(z)
model = keras.Model(inputs=[inputs], outputs=[outputs])
```

После обучения в течение нескольких эпох модель станет достаточно хорошей в оценке того, является рецензия положительной или нет. Если вы используете обратный вызов `TensorBoard()`, тогда можете визуализировать вложения в `TensorBoard` по мере их узнавания: просто-таки завораживает смотреть на то, как слова наподобие “awesome” (“потрясающий”) и “amazing” (“изумительный”) постепенно группируются с одной стороны пространства

вложений, в то время как слова вроде “awful” (“отвратительный”) и “terrible” (“ужасный”) собираются в группу с другой стороны. Некоторые слова не настолько положительны, как вы могли ожидать (по крайней мере, при текущей модели), такие как слово “good” (“хороший”), предположительно из-за того, что многие отрицательные рецензии содержат оборот “not good” (“не хороший”). Впечатляет, что модель способна выучить полезные вложения слов, основываясь всего лишь на 25 000 рецензий. Представьте себе, насколько хорошими были бы вложения, если бы мы располагали миллиардами рецензий, чтобы обучать на них! К сожалению, такого количества рецензий у нас нет, но может быть, нам удастся повторно задействовать вложения слов, выученные на каком-то другом крупном корпусе текстов (например, статей из Википедии), даже если он не состоит из рецензий на фильмы? В конце концов, слово “amazing” обычно имеет одинаковый смысл, применяется оно в обсуждении фильмов или чего-то другого. Более того, возможно вложения оказались бы полезными для смыслового анализа, даже когда они были выучены в другой задаче: поскольку такие слова, как “awesome” и “amazing”, имеют похожий смысл, вполне вероятно, что они образуют группу в пространстве вложений и для других задач (скажем, при прогнозировании следующего слова в предложении). Если все положительные слова и все отрицательные слова формируют кластеры, тогда это будет полезно для смыслового анализа. Таким образом, вместо того, чтобы использовать настолько много параметров, чтобы узнать вложения слов, давайте посмотрим, сумеем ли мы просто повторно применить заранее обученные вложения.

## Повторное использование заранее обученных вложений

Проект TensorFlow Hub облегчает повторное использование компонентов заранее обученных моделей в собственных моделях. Такие компоненты моделей называются *модулями*. Просто просмотрите хранилище TF Hub (<https://tfhub.dev/>), найдите необходимый модуль и скопируйте пример кода в свой проект, после чего модуль автоматически загрузится вместе с заранее обученными весами и будет включен в вашу модель. Легко!

Например, давайте применим модуль с вложениями предложений nnlm-en-dim50 версии 1 в нашей модели смыслового анализа:

```
import          as  
model = keras.Sequential([
```

```

hub.KerasLayer(
    "https://tfhub.dev/google/tf2-preview/nnlm-en-dim50/1",
    dtype=tf.string, input_shape=[], output_shape=[50]),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(1, activation="sigmoid")
)
model.compile(loss="binary_crossentropy", optimizer="adam",
               metrics=["accuracy"])

```

Слой `hub.KerasLayer` загружает модуль из указанного URL-адреса. Модуль `nnlm-en-dim50` представляет собой *кодировщик предложений*: он получает на входе строки и кодирует каждую в виде одиночного вектора (в данном случае 50-мерного вектора). Внутренне модуль анализирует строку (разбивая на слова по пробелам) и кодирует каждое слово с использованием матрицы вложений, которая была заранее обучена на гигантском корпусе текстов: корпусе Google News 7B (длиной семь миллиардов слов!). Затем он рассчитывает среднее всех вложений слов, а результатом будет вложение предложения<sup>9</sup>. Чтобы создать хорошую модель смыслового анализа, далее мы можем добавить два простых слоя `Dense`.

По умолчанию слой `hub.KerasLayer` не является обучаемым, но вы можете изменить это, установив `trainable=True` при его создании, что позволит его точно подстроить к имеющейся задаче.



Не все модули TF Hub поддерживают версию библиотеки TensorFlow 2, поэтому удостоверьтесь в том, что выбираете модуль, который ее поддерживает.

Затем мы можем загрузить набор данных IMDb с рецензиями — нет необходимости в его предварительной обработке (кроме группирования в пакеты и предварительной выборки) — и непосредственно обучить модель:

```

datasets, info = tfds.load("imdb_reviews", as_supervised=True,
                           with_info=True)
train_size = info.splits["train"].num_examples
batch_size = 32
train_set = datasets["train"].batch(batch_size).prefetch(1)
history = model.fit(train_set, epochs=5)

```

<sup>9</sup> Точнее говоря, вложение предложения эквивалентно среднему вложению слов, умноженному на квадратный корень из количества слов в предложении. Это компенсирует тот факт, что среднее *n* векторов становится короче с ростом *n*.

Обратите внимание, что в последней части URL-адреса модуля TF Hub указано, что нам нужна его версия 1. Ведение версий гарантирует, что выпуск новой версии модуля не приведет к нарушению работы нашей модели. Удобно то, что если вы просто введете данный URL-адрес в веб-браузере, то получите документацию по этому модулю. По умолчанию TF Hub будет кешировать загруженные файлы во временном каталоге на локальной системе. Вы можете отдать предпочтение их загрузке в какой-то постоянный каталог, чтобы не загружать их заново после каждой очистки системы. В таком случае укажите в переменной среды `TFHUB_CACHE_DIR` желаемый каталог (например, `os.environ["TFHUB_CACHE_DIR"] = "./my_tfhub_cache"`).

До сих пор мы рассматривали временные ряды, генерацию текста с применением Char-RNN и смысловой анализ с использованием моделей RNN уровня слов, обучая собственные вложения слов или повторно применяя предварительно обученные вложения. Далее мы взглянем на еще одну важную задачу NLP: *нейронный машинный перевод (NMT)*. Сначала мы будем использовать чистую модель “кодировщик–декодировщик”, затем усовершенствуем ее с помощью механизмов внимания и в заключение ознакомимся с замечательной архитектурой “Преобразователь” (*Transformer*).

## Сеть “кодировщик–декодировщик” для нейронного машинного перевода

Давайте исследуем простую модель для нейронного машинного перевода (<https://homl.info/103>)<sup>10</sup>, которая будет переводить английские предложения на французский язык (рис. 16.3).

Английское предложение передается кодировщику, а декодировщик выдаст его перевод на французский язык. Обратите внимание, что французский перевод также применяется как вход в декодировщик, но он смешен 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 предоставляется маркер начала последовательности (*start-of-sequence* — SOS). Декодировщик ожидает, что предложение завершится маркером конца последовательности (*end-of-sequence* — EOS).

<sup>10</sup> Илья Сатскевер и др., *Sequence to Sequence Learning with Neural Networks* (Обучение типа “последовательность в последовательность” с помощью нейронных сетей), препринт arXiv:1409.3215 (2014 г.).

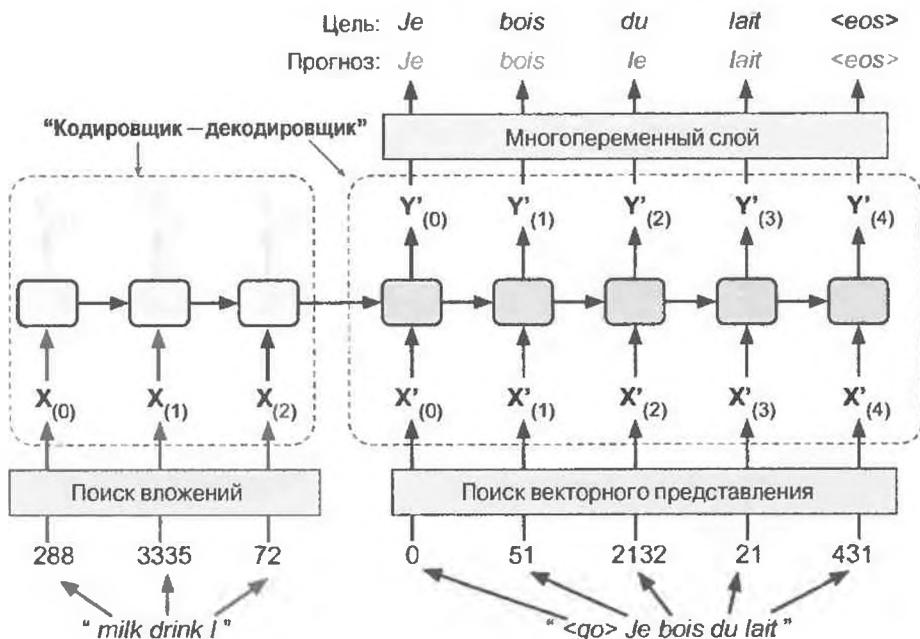


Рис. 16.3. Простая модель для нейронного машинного перевода

Имейте в виду, что перед передачей кодировщику английские предложения выворачиваются наизнанку. Скажем, “I drink milk” превращается в “milk drink I”. 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 его идентификатором (например, 288 для слова “milk”). Затем слой Embedding возвращает вложение слова. Такие вложения 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. Выходом будет слово, с которым связана самая высокая вероятность. Процесс очень похож на обыкновенную задачу классификации, так что вы можете обучать модель с использованием потери “sparse\_categorical\_crossentropy” во многом подобно тому, как поступали в случае модели Char-RNN.

Обратите внимание, что во время выводения (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. 16.4 (это потребует поиска вложения, что на диаграмме не показано).

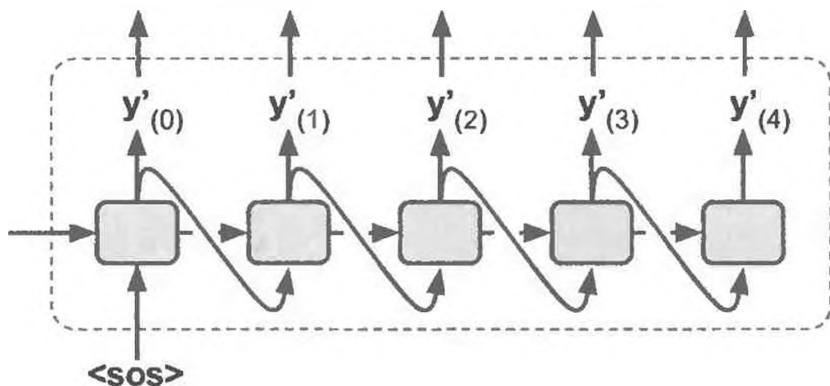


Рис. 16.4. Передача предыдущего выходного слова в качестве входа во время вывода

Итак, теперь у вас есть общая картина. Однако если вы решите реализовать такую модель, то есть несколько дополнительных деталей, которые понадобится учесть.

- До сих пор мы полагали, что все входные последовательности (передаваемые кодировщику и декодировщику) имели постоянную длину. Но длина предложений вполне очевидно может варьироваться. Поскольку обычновенные тензоры имеют фиксированные формы, они могут содержать только предложения той же самой длины. Чтобы справиться с этим, вы можете применять маскирование, как обсуждалось ранее. Тем не менее, если предложения имеют очень разные длины, то вы не сможете просто обрезать их, как делалось для смыслового анализа (потому что нам нужен полный, а не усеченный перевод). Взамен сгруппируйте предложения в участки с похожими длинами (скажем, участок для предложений от 1 до 6 слов, еще один участок для предложений от 7 до 12 слов и т.д.), используя дополнение для самых коротких предложений, чтобы обеспечить одну и ту же длину у всех предложений в участке (ознакомьтесь с функцией `t.f.data.experimental.bucket_by_sequence_length()`). Например, предложение “I drink milk” превращается в “<pad> <pad> milk drink I”.

- Мы хотим игнорировать любой вывод после маркера EOS, так что эти маркеры не должны вносить вклад в потерю (их потребуется маскировать). Например, если модель выдает “Je bois du lait <eos> oui” (“Я пью молоко <eos> да”), тогда потеря для последнего слова должна игнорироваться.
- 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 последующий расчет многопараметрической функции на таком большом векторе потребует очень интенсивных вычислений. Одно из решений предусматривает просмотр лишь логитов, выдаваемых моделью для правильных слов и для случайной выборки неправильных слов, и затем расчет приближенной величины потери на основе только упомянутых логитов. Такой *многопараметрический прием с выборкой* (*sampled softmax technique*) был введен Себастьяном Жаном и др. в 2015 году (<https://hml.info/104>)<sup>11</sup>. В TensorFlow для этого можно применять функцию `tf.nn.sampled_softmax_loss()` во время обучения и использовать нормальную многопараметрическую логистическую функцию во время вывода (многопараметрический прием с выборкой не может применяться во время вывода, т.к. он требует знания цели).

Проект TensorFlow Addons включает много инструментов вида “последовательность в последовательность”, которые позволяют легко строить модели “кодировщик–декодировщик”, готовые к вводу в эксплуатацию. Например, следующий код создает базовую модель “кодировщик–декодировщик”, подобную показанной на рис. 16.3:

```
import           as
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
sequence_lengths = keras.layers.Input(shape=[], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
```

<sup>11</sup> Себастьян Жан и др., *On Using Very Large Target Vocabulary for Neural Machine Translation* (Об использовании очень крупного целевого словаря для нейронного машинного перевода), *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing 1* (2015 г.): с. 1–10.

```

encoder = keras.layers.LSTM(512, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_embeddings)
encoder_state = [state_h, state_c]

sampler = tfa.seq2seq.sampler.TrainingSampler()

decoder_cell = keras.layers.LSTMCell(512)
output_layer = keras.layers.Dense(vocab_size)
decoder = tfa.seq2seq.basic_decoder.BasicDecoder(decoder_cell,
                                                 sampler,
                                                 output_layer=output_layer)

final_outputs, final_state, final_sequence_lengths = decoder(
    decoder_embeddings, initial_state=encoder_state,
    sequence_length=sequence_lengths)
Y_proba = tf.nn.softmax(final_outputs.rnn_output)

model = keras.Model(inputs=[encoder_inputs, decoder_inputs,
                            sequence_lengths],
                     outputs=[Y_proba])

```

Код в основном не требует пояснений, но несколько моментов все же нужно отметить. Первым делом при создании слоя LSTM мы устанавливаем `return_state=True`, так что можем получить его финальное скрытое состояние и передать его декодировщику. Поскольку мы используем ячейку LSTM, она фактически возвращает два скрытых состояния (краткосрочное и долгосрочное). Класс `TrainingSampler` — один из нескольких классов для выборки, доступных в TensorFlow Addons: их роль заключается в том, чтобы на каждом шаге сообщать декодировщику о необходимости делать вид, что предыдущий вывод был. Во время выведения им должно быть вложение маркера, который в действительности был выдан. Во время обучения им должно быть вложение предыдущего маркера цели: именно потому мы применяем `TrainingSampler`. На практике часто неплохо начинать обучение с вложения цели предыдущего временного шага и постепенно переходить к использованию вложения фактического маркера, который был выдан на предыдущем шаге. Идея была предложена в статье 2015 года (<https://homl.info/scheduledsampling>)<sup>12</sup> Сэми Бенджи и др.

Класс `ScheduledEmbeddingTrainingSampler` будет случайным образом выбирать между целью и фактическим выходом с вероятностью, которую вы можете постепенно изменять во время обучения.

---

<sup>12</sup> Сэми Бенджи и др., *Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks* (Запланированная выборка для прогнозирования последовательностей с помощью рекуррентных нейронных сетей), препринт arXiv:1506.03099 (2015 г.).

## Двунаправленные рекуррентные нейронные сети

На каждом временном шаге обычновенный рекуррентный слой просматривает только прошлые и настоящие входы перед генерацией своего выхода. Таким образом, он является “причинным”, т.е. не может заглядывать в будущее. Подобный тип сети RNN имеет смысл применять при прогнозировании временных рядов, но для многих задач NLP вроде нейронного машинного перевода часто предпочтительнее смотреть вперед на несколько слов, прежде чем кодировать заданное слово. Например, возьмем фразы “the Queen of the United Kingdom” (“королева Соединенного Королевства”), “the queen of hearts” (“покорительница сердец”) и “the queen bee” (“пчелиная матка”): для надлежащего кодирования слова “queen” необходимо заглядывать вперед. Чтобы реализовать это, запустите два рекуррентных слоя на тех же самых входах, из которых один читает слова слева направо, а другой — справа налево. Затем просто объединяйте их выходы на каждом временном шаге, обычно путем конкатенации. Результат называется *двунаправленным рекуррентным слоем* (рис. 16.5).

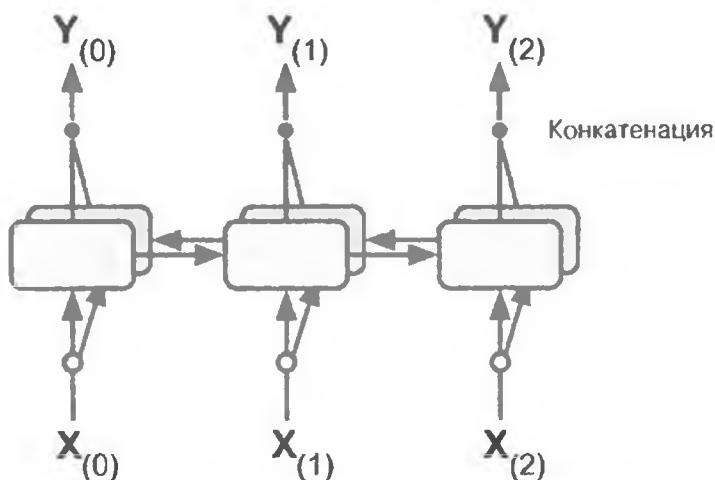


Рис. 16.5. Двунаправленный рекуррентный слой

Чтобы реализовать двунаправленный рекуррентный слой в Keras, поместите рекуррентный слой внутрь слоя keras.layers.Bidirectional. Например, следующий код создает двунаправленный слой GRU:

```
keras.layers.Bidirectional(keras.layers.GRU(10, return_sequences=True))
```



Слой Bidirectional будет создавать клон слоя GRU (но в обратном направлении), запускать оба слоя GRU и выполнять конкатенацию их выводов. Таким образом, хотя слой GRU имеет 10 элементов, слой Bidirectional будет выдавать 20 значений на временной шаг.

## Лучевой поиск

Предположим, вы обучаете модель “кодировщик–декодировщик” и используете ее для перевода французского предложения “Comment vas-tu?” (“Как ты?”) на английский язык. Вы надеетесь, что она выдаст правильный перевод (“How are you?”), но, к сожалению, модель выдает “How will you?” (“Как ты будешь?”). Глядя на обучающий набор, вы замечаете много предложений типа “Comment vas-tu jouer?” (“Как ты будешь играть?”), которые переводятся как “How will you play?”. Таким образом, для модели вовсе не было бессмысленно выдавать “How will” после того, как она увидела “Comment vas”. К несчастью, в данном случае это было ошибкой, и модель не могла вернуться назад и исправить ее, а потому она попыталась как можно лучше закончить предложение. Из-за жадной выдачи наиболее вероятного слова на каждом шаге в итоге получается субоптимальный перевод. Как предоставить модели возможность вернуться назад и исправить ошибки, допущенные ею ранее? Одним из самых распространенных решений является лучевой поиск (*beam search*): он отслеживает список кандидатов из  $k$  наиболее многообещающих предложений (скажем, верхние три) и на каждом шаге декодировщика пытается расширить их на одно слово, сохраняя только  $k$  наиболее вероятных предложений. Параметр  $k$  называется шириной луча (*beam width*).

Например, допустим, вы применяете модель для перевода предложения “Comment vas-tu?”, используя лучевой поиск с шириной луча 3. На первом шаге декодировщика модель выдаст оценку вероятности для каждого возможного слова. Пусть верхние три слова выглядят как “How” (оценка вероятности 75%), “What” (3%) и “You” (1%). Это наш список кандидатов до настоящего момента. Далее мы создаем три копии модели и применяем их для поиска следующего слова для каждого предложения. Каждая модель будет выдавать оценку вероятности на слово в словаре. Первая модель пытается найти следующее слово в предложении “How” и возможно выдаст вероятность 36% для слова “will”, 32% для слова “are”, 16% для слова “do” и т.д. Обратите внимание, что на самом деле они являются *условными* вероятностями с учетом того, что предложение начинается с “How”. Вторая модель

попробует закончить предложение “What”; она может выдать условную вероятность 50% для слова “age” и т.д. Принимая наличие в словаре 10000 слов, каждая модель выдаст 10000 вероятностей.

Далее мы рассчитываем вероятности каждого из 30000 двухсловных предложений, которые модели приняли во внимание ( $3 \times 10000$ ). Для этого мы умножаем оценку условной вероятности каждого слова на оценку вероятности предложения, которое слово заканчивает. Например, оценка вероятности предложения “How” была 75%, тогда как оценка условной вероятности слова “will” (с учетом первого слова “How”) составляла 36%, поэтому оценка вероятности предложения “How will” равна  $75\% \times 36\% = 27\%$ . После расчета вероятностей всех 30 000 двухсловных предложений мы сохраняем только верхние 3. Может быть, все они начинаются со слова “How”: “How will” (27%), “How are” (24%) и “How do” (12%). Прямо сейчас предложение “How will” выигрывает, но “How are” не было исключено.

Затем мы повторяем тот же самый процесс: используем три модели для прогнозирования следующего слова в каждом из трех предложений и рассчитываем вероятности всех 30 000 трехсловных предложений, которые принимались во внимание. Возможно, теперь верхними тремя стали “How are you” (10%), “How do you” (8%) и “How will you” (2%). На следующем шаге мы могли бы получить “How do you do” (7%), “How are you <eos>” (6%) и “How are you doing” (3%). Обратите внимание, что “How will” было исключено, и мы располагаем тремя целиком приемлемыми переводами. Мы повысили эффективность нашей модели “кодировщик–декодировщик” без какого-либо дополнительного обучения, просто за счет ее более разумного применения.

Реализовать лучевой поиск с использованием TensorFlow Addons достаточно легко:

```
beam_width = 10
decoder = tfa.seq2seq.beam_search_decoder.BeamSearchDecoder(
    cell=decoder_cell, beam_width=beam_width,
    output_layer=output_layer)
decoder_initial_state = tfa.seq2seq.beam_search_decoder.tile_batch(
    encoder_state, multiplier=beam_width)
outputs, _ = decoder(
    embedding_decoder, start_tokens=start_tokens, end_token=end_token,
    initial_state=decoder_initial_state)
```

Сначала мы создаем объект BeamSearchDecoder, который включает в себя все клоны декодировщика (в данном случае 10 клонов). Затем мы созда-

ем по одной копии финального состояния кодировщика для каждого клона декодировщика и передаем эти состояния декодировщику вместе с маркерами начала и конца.

При этом вы можете получать хорошие переводы для довольно коротких предложений (особенно, если применяете заранее обученные вложения слов). К сожалению, такая модель будет очень плохо переводить длинные предложения. И снова проблема произрастает из ограниченной краткосрочной памяти сетей RNN. Решить проблему позволяет нововведение, изменяющее правила игры нововведение — *механизмы внимания*.

## Механизмы внимания

Рассмотрим путь, который английское слово “milk” (“молоко”) проходит до своего французского перевода “lait” (рис. 16.3): он действительно длинный! В итоге представление указанного слова (наряду со всеми остальными словами) должно переноситься на много шагов, прежде чем оно фактически будет использоваться. Можем ли мы сделать такой путь короче?

Это было основной идеей в новаторской статье 2014 года (<https://homl.info/attention>)<sup>13</sup>, которую написали Дэмиетри Бахданау и др. Они представили методику, которая позволяет декодировщику на каждом временном шаге фокусироваться на соответствующих словах (как они закодированы кодировщиком). Скажем, на временном шаге, где кодировщик должен выдать слово “lait”, он сосредоточивает свое внимание на слове “milk”. В результате путь от входного слова до его перевода теперь стал гораздо короче, а потому ограничения краткосрочной памяти сетей RNN оказывают намного меньшее влияние. Механизмы внимания произвели коренные перемены в нейронном машинном переводе (и вообще в обработке естественного языка), сделав возможным значительное улучшение в современном положении дел, особенно для длинных предложений (свыше 30 слов)<sup>14</sup>.

<sup>13</sup> Дэмиетри Бахданау и др., *Neural Machine Translation by Jointly Learning to Align and Translate* (Нейронный машинный перевод за счет одновременного обучения выравниванию и переводу), препринт arXiv:1409.0473 (2014 г.).

<sup>14</sup> Самой распространенной метрикой, используемой в NMT, является мера замены двуязычного определения (BiLingual Evaluation Understudy — BLEU), которая сравнивает каждый перевод, произведенный моделью, с несколькими хорошими переводами, сделанными людьми: она подсчитывает количество *n*-грамм (последовательностей из *n* слов), которые появляются в любых целевых переводах, и регулирует меру с целью учета частоты произведенных *n*-грамм в целевых переводах.

На рис. 16.6 показана архитектура данной модели (как мы увидим, слегка упрощенная). Слева находятся кодировщик и декодировщик. Вместо того чтобы просто отправлять декодировщику финальное скрытое состояние кодировщика (что по-прежнему делается, хотя на рисунке не отражено), теперь мы посылаем декодировщику весь вывод кодировщика.

На каждом временном шаге ячейка памяти декодировщика вычисляет взвешенную сумму всех выходов кодировщика: она определяет, на каких словах он будет фокусироваться на текущем шаге. Вес  $\alpha_{(t,i)}$  — это вес  $i$ -того выхода кодировщика в  $t$ -ый временной шаг декодировщика. Например, если вес  $\alpha_{(3,2)}$  намного больше весов  $\alpha_{(3,0)}$  и  $\alpha_{(3,1)}$ , тогда декодировщик будет уделять гораздо большее внимание слову номер 2 ("milk"), чем остальным двум словам, по крайней мере, на данном временном шаге.

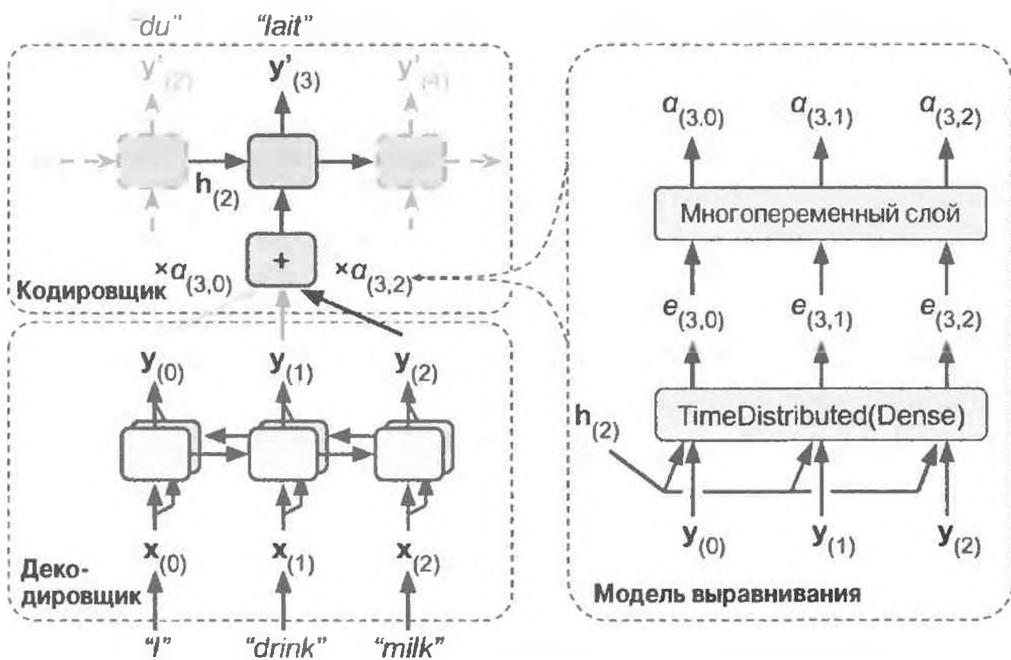


Рис. 16.6. Нейронный машинный перевод с использованием сети “кодировщик–декодировщик” с моделью внимания

Оставшаяся часть декодировщика работает точно так же, как ранее: на каждом временном шаге ячейка памяти получает входы, которые только что обсуждались, плюс скрытое состояние из предыдущего временного шага и в заключение (хотя на диаграмме это не показано) она принимает целевое слово из предыдущего временного шага (или во время выведения выход из предыдущего временного шага).

Но откуда берутся веса  $\alpha_{(t,j)}$ ? На самом деле все довольно просто: они производятся разновидностью небольшой нейронной сети под названием модель выравнивания (*alignment model*) или слой внимания (*attention layer*), которая обучается вместе с остальной частью модели “кодировщик–декодировщик”. Модель выравнивания показана справа на рис. 16.6. Она начинается с распределенного во времени слоя Dense<sup>15</sup> с единственным нейроном, который получает в качестве входа все выходы кодировщика, объединенные с предыдущим скрытым состоянием декодировщика (например,  $h_{(2)}$ ). Этот слой выдает меру (или энергию) для каждого выхода кодировщика (скажем,  $e_{(3,2)}$ ): такая мера определяет, насколько хорошо каждый выход выровнен с предыдущим скрытым состоянием декодировщика. Наконец, все меры проходят через многопеременный слой, чтобы получить финальный вес для каждого выхода кодировщика (например,  $\alpha_{(3,2)}$ ). Все веса для заданного временного шага декодировщика в сумме дают 1 (т.к. многопеременный слой не является распределенным во времени). Такой механизм внимания называется вниманием Бахданау (в честь первого автора статьи). Поскольку он объединяет выход кодировщика с предыдущим скрытым состоянием декодировщика, иногда его называют конкатенативным вниманием (*concatenative attention*) или аддитивным вниманием (*additive attention*).



Если входное предложение имеет длину  $n$  слов и выходное предложение предположительно примерно такой же длины, тогда этой модели придется рассчитать около  $n^2$  весов. К счастью, такая квадратичная вычислительная сложность все еще поддается обработке, потому что даже длинные предложения не содержат тысячи слов.

<sup>15</sup> Вспомните, что распределенный во времени слой Dense эквивалентен обычному слою Dense, который вы применяете независимо на каждом временном шаге (только он гораздо быстрее).

Вскоре после этого в статье 2015 года (<https://homl.info/luongattention>)<sup>16</sup> Минь-Тханг Лыонг и др. представили еще один механизм внимания, получивший широкое распространение. Поскольку цель механизма внимания заключается в измерении подобия между одним из выходов кодировщика и предыдущим скрытым состоянием декодировщика, авторы предложили просто вычислять скалярное произведение (см. главу 4) указанных двух векторов, т.к. оно часто оказывается хорошей мерой подобия, а современное оборудование способно рассчитывать его гораздо быстрее. Чтобы это было возможным, оба вектора обязаны иметь одинаковую размерность. Такой механизм был назван *вниманием Лыонга* (*Luong attention*) в честь первого автора статьи, а иногда его называют *множественным вниманием* (*multiplicative attention*). Скалярное произведение дает оценку, затем все оценки (на заданном временном шаге декодировщика) проходят через многопеременный слой для получения финальных весов, как в случае внимания Бахданау.

Еще одно предложенное ими упрощение заключалось в том, чтобы применять скрытое состояние декодировщика на текущем временном шаге, а не на предыдущем (т.е.  $h_{(t)}$  вместо  $h_{(t-1)}$ ), и далее использовать выход механизма внимания (взгляните на  $\tilde{h}_{(t)}$ ) непосредственно для вычисления прогнозов декодировщика (вместо его применения для расчета текущего скрытого состояния декодировщика). Авторы также предложили разновидность механизма скалярного произведения, где перед его расчетом выходы декодировщика сначала прогоняются через линейную трансформацию (т.е. распределенный во времени слой Dense без члена смещения). Это называется “универсальным” подходом со скалярным произведением. Они сравнили оба подхода со скалярным произведением применительно к механизму конкатенативного внимания (добавив вектор параметров изменения масштаба  $v$ ) и заметили, что разновидности скалярного произведения работают лучше, чем конкатенативное внимание. По указанной причине конкатенативное внимание в настоящее время используется намного меньше. Три механизма внимания подытожены в уравнении 16.1.

<sup>16</sup> Минь-Тханг Лыонг и др., *Effective Approaches to Attention-Based Neural Machine Translation* (Эффективные подходы к нейронному машинному переводу, основанному на внимании), *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015 г.): с. 1412–1421.

## Уравнение 16.1. Механизмы внимания

$$\tilde{h}_{(t)} = \sum_i \alpha_{(t, i)} y_{(i)},$$

где  $\alpha_{(t, i)} = \frac{\exp(e_{(t, i)})}{\sum_{i'} \exp(e_{(t, i')})}$

и  $e_{(t, i)} = \begin{cases} h_{(t)}^T y_{(i)} & \text{скалярное произведение} \\ h_{(t)}^T W y_{(i)} & \text{универсальное} \\ v^T \tanh(W[h_{(t)}; y_{(i)}]) & \text{конкатенативное} \end{cases}$

Вот как добавить внимание Льонга к модели “кодировщик–декодировщик” с применением TensorFlow Addons:

```
attention_mechanism = tfa.seq2seq.attention_wrapper.LuongAttention(  
    units, encoder_state, memory_sequence_length=encoder_sequence_length)  
attention_decoder_cell = tfa.seq2seq.attention_wrapper.AttentionWrapper(  
    decoder_cell, attention_mechanism, attention_layer_size=n_units)
```

Мы просто помещаем ячейку декодировщика внутрь AttentionWrapper и предоставляем желаемый механизм внимания (внимание Льонга в приведенном примере).

## Зрительное внимание

Механизмы внимания теперь используются для разнообразных целей. Одним из их первых приложений помимо NMT было генерирование подписей для изображений с применением зрительного внимания (visual attention; <https://homl.info/visualattention>)<sup>17</sup>: сверточная нейронная сеть сначала обрабатывает изображение и выдает ряд карт признаков, после чего сеть RNN декодировщика с механизмом внимания генерирует подпись, по одному слову за раз. На каждом временном шаге (каждое слово) декодировщик использует модель внимания для фокусирования только на надлежащей части изображения.

<sup>17</sup> Келвин Ксу и др., *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* (Увидеть, обратить внимание и сообщить: генерация подписей для изображений нейронной сетью со зрительным вниманием), *Proceedings of the 32nd International Conference on Machine Learning* (2015 г.): с. 2048–2057.

Например, на рис. 16.7 модель сгенерирует подпись “A woman is throwing a frisbee in a park” (“Женщина бросает летающую тарелку в парке”) и вы можете видеть, на какой части входного изображения декодировщик сосредоточил свое внимание, когда собирался выдать слово “frisbee” (“летающая тарелка”): очевидно, большинство его внимания было сфокусировано на летающей тарелке.



Рис. 16.7. Зрительное внимание: входное изображение (слева) и фокус модели перед выпуском слова “frisbee” (справа)<sup>18</sup>

## Объяснимость

Механизмы влияния обладают еще одним дополнительным преимуществом — они облегчают понимание того, что привело модель к выдаче своего вывода. Это называется *объяснимостью* (*explainability*). Она оказывается особенно полезной, когда модель допускает ошибку: скажем, если изображение собаки, гуляющей по снегу, помечается как “волк, гуляющий по снегу”, то вы в состоянии возвратиться и посмотреть, на чем была сфокусирована модель при выдаче слова “волк”. Вы можете обнаружить, что модель обращала внимание не только на собаку, но и на снег, намекая на возможное объяснение: вероятно, способ, которым она научилась отличать собак от волков, предусматривал проверку, нет ли вокруг обилия снега. Затем вы можете устранить проблему, обучив модель на добавочных изображениях волков без снега и собак со снегом.

<sup>18</sup> Часть рисунка 3 в статье Келвина Ксу и др. Воспроизводится с разрешения авторов.

Описанный пример взят из великолепной работы 2016 года (<https://hml.info/explainclass>)<sup>19</sup> Марко Тулио Рибейро и др., где применялся другой подход к объяснимости: обучение интерпретируемой модели локально вблизи прогноза классификатора.

В ряде приложений объяснимость является не только инструментом для отладки модели; она может быть законным требованием (подумайте о системе, которая принимает решение, предоставлять ли вам заем).

Механизмы внимания настолько мощные, что вы фактически можете строить современные модели, используя только эти механизмы.

## Внимание — это все, что нужно: архитектура “Преобразователь”

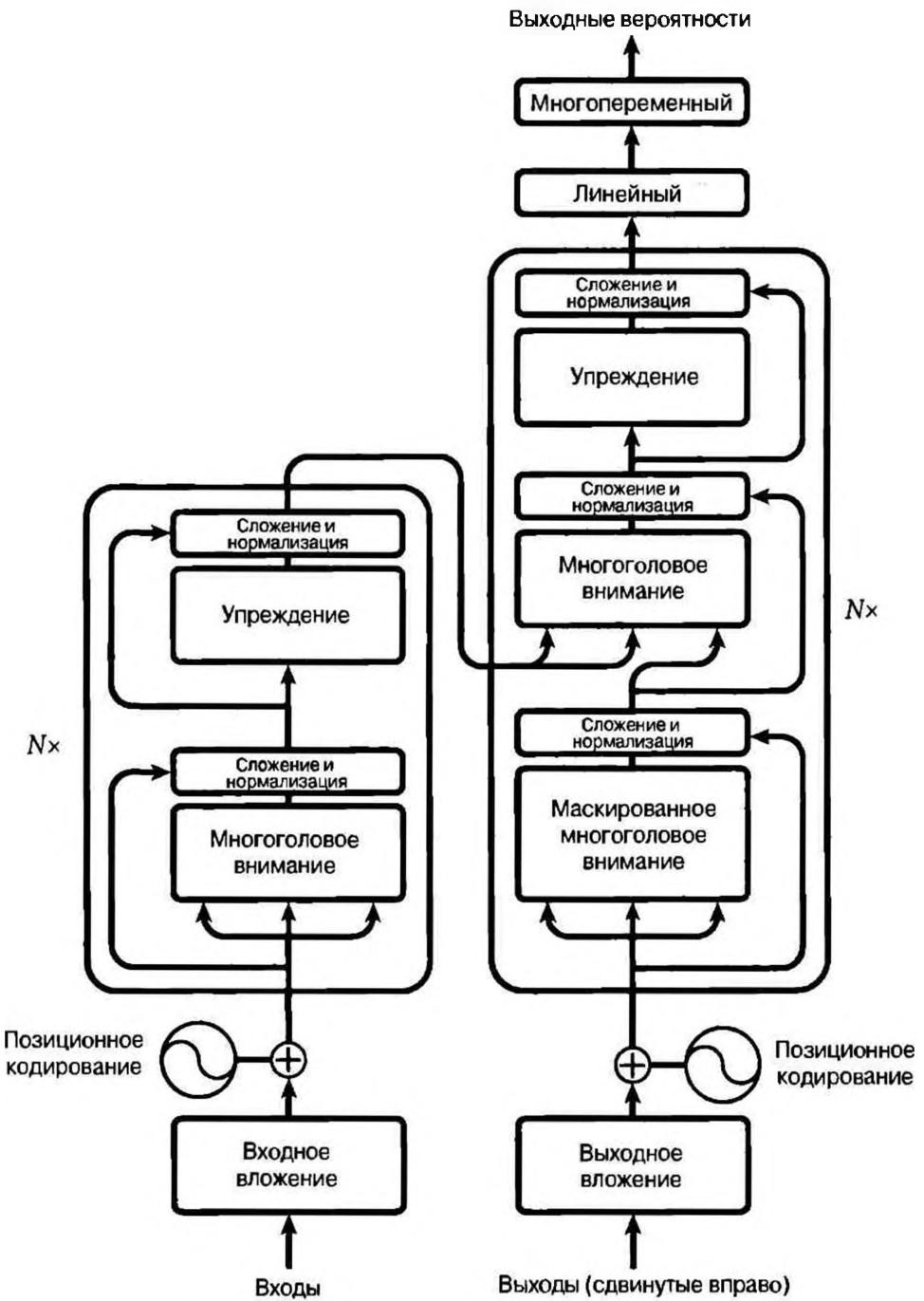
В новаторской статье 2017 года (<https://hml.info/transformer>)<sup>20</sup> команда исследователей из Google предположили, что “внимание — это все, что нужно” (“Attention Is All You Need”). Им удалось создать архитектуру под названием “Преобразователь” (*Transformer*), которая значительно улучшила положение дел в NMT без применения рекуррентных или сверточных слоев<sup>21</sup>, а используя лишь механизмы внимания (плюс слои вложений, плотные слои, слои нормализации и несколько других мелочей). В качестве дополнительного бонуса архитектура “Преобразователь” также была гораздо быстрее в обучении и легче в распараллеливании, поэтому они сумели обучить ее за долю времени и затрат, присущих предшествующим современным моделям.

Архитектура “Преобразователь” представлена на рис. 16.8.

<sup>19</sup> Марко Тулио Рибейро и др., “Why Should I Trust You?”, *Explaining the Predictions of Any Classifier* (“Почему я должен вам доверять?”, объяснение прогнозов любого классификатора), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016 г.): с. 1135–1144.

<sup>20</sup> Ашиш Васвани и др., *Attention Is All You Need* (Внимание — это все, что нужно), *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017 г.): с. 6000–6010.

<sup>21</sup> Так как архитектура “Преобразователь” использует распределенные во времени слои Dense, вы могли бы утверждать, что в ней применяются одномерные сверточные слои с размером ядра 1.



*Рис. 16.8. Архитектура “Преобразователь”<sup>22</sup>*

<sup>22</sup> Рисунок 1 из статьи Ашиша Васвани и др. Воспроизведется с разрешения авторов.

Давайте разберем рис. 16.8.

- В левой части находится кодировщик. Как и ранее, он принимает на входе пакет предложений, представленных в виде последовательностей идентификаторов слов (входной формой является [размер пакета, максимальная длина входного предложения]), а каждое слово кодируется в 512-мерное представление (поэтому выходной формой кодировщика будет [размер пакета, максимальная длина входного предложения, 512]). Обратите внимание, что верхняя часть кодировщика уложена стопкой  $N$  раз (в статье  $N = 6$ ).
- В правой части находится декодировщик. Во время обучения он принимает на входе целевое предложение (также представленное как последовательность идентификаторов слов), сдвинутое на один временной шаг вправо (т.е. маркер начала последовательности вставлен вначале). Кроме того, декодировщик получает выходы кодировщика (обозначены линиями со стрелками, идущими от левой части). Обратите внимание, что верхняя часть декодировщика тоже уложена стопкой  $N$  раз, а финальные выходы стопки кодировщика подаются на каждый из  $N$  таких уровней. Как и ранее, декодировщик выдает вероятность для каждого возможного следующего слова на каждом временном шаге (его выходной формой будет [размер пакета, максимальная длина выходного предложения, длина словаря]).
- Во время выводения декодировщик не может получать цели, так что ему подаются ранее выданные слова (начиная с маркера начала последовательности). Таким образом, модель необходимо вызывать неоднократно, прогнозируя в каждом раунде еще одно слово (которое подается декодировщику в следующем раунде до тех пор, пока не будет выдан маркер конца последовательности).
- Присмотревшись внимательнее, вы заметите, что уже знакомы с большинством компонентов: есть два слоя вложения,  $5 \times N$  обходящих связей, за каждой из которых следует слой нормализации,  $2 \times N$  модулей “упреждения” (“Feed Forward”), состоящих из двух плотных слоев каждый (в первом применяется функция активации ReLU, а второй не имеет функции активации), и выходной слой, представляющий собой плотный слой, который использует многопеременную функцию активации. Все перечисленные слои являются распределенными во времени,

так что каждое слово трактуется независимо от всех остальных. Но как можно перевести предложение, видя только по одному слову за раз? Именно здесь в игру вступают новые компоненты.

- Слой многоголового внимания (*Multi-Head Attention*) кодировщика кодирует взаимосвязь каждого слова с каждым другим словом в том же самом предложении, сосредоточивая большее внимание на самых значимых словах. Например, выход этого слоя для слова “Queen” (“королева”) из предложения “They welcomed the Queen of the United Kingdom” (“Они приветствовали королеву Соединенного Королевства”) зависит от всех слов в предложении, но, вероятно, большее внимание будет уделяться словам “United” (“Соединенное”) и “Kingdom” (“Королевство”), нежели слову “They” (“Они”) или “welcomed” (“приветствовали”). Такой механизм внимания называется *самовниманием (self-attention)*, т.е. предложение уделяет внимание самому себе. Вскоре мы обсудим, как оно в точности работает. Слой маскированного многоголового внимания (*Masked Multi-Head Attention*) декодировщика делает то же самое, но каждому слову разрешено уделять внимание только словам, расположенным перед ним. Наконец, в верхнем слое многоголового внимания декодировщика внимание обращается на слова во входном предложении. Скажем, возможно, декодировщик уделит пристальное внимание слову “Queen” во входном предложении, когда соберется выдавать перевод этого слова.
- Позиционные вложения (*positional embedding*) — это просто плотные векторы (во многом подобно вложениям слов), которые представляют позицию слова в предложении. Позиционное вложение номер  $i$  складывается с вложением  $i$ -го слова в каждом предложении. В итоге модель получает доступ к позиции каждого слова, что необходимо, поскольку слои многоголового внимания не учитывают порядок следования или позицию слов; они смотрят только на их взаимосвязи. Так как все остальные слои являются распределенными во времени, у них нет никакого способа узнать позицию каждого слова (ни относительную, ни абсолютную). Очевидно, относительные и абсолютные позиции слов важны, поэтому нам нужно каким-то образом представить архитектуре “Преобразователь” сведения о них и позиционные вложения обеспечивают хороший способ.

Давайте чуть подробнее рассмотрим оба новых компонента архитектуры “Преобразователь”, начиная с позиционных вложений.

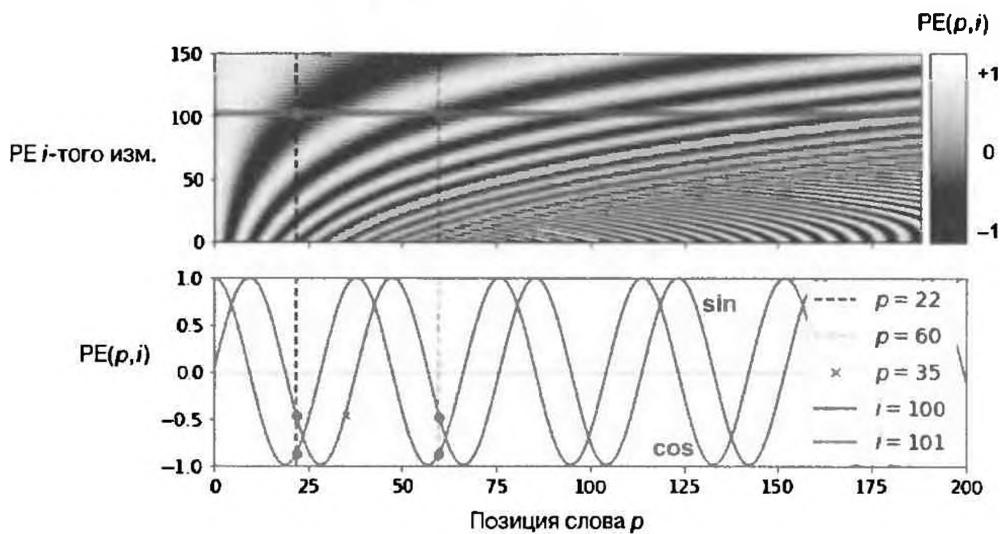
## Позиционные вложения

Позиционное вложение является плотным вектором, который кодирует позицию слова внутри предложения:  $i$ -тое позиционное вложение просто складывается с вложением  $i$ -того слова в предложении. Модель сама может узнать такие позиционные вложения, но авторы в своей статье предпочли применять фиксированные позиционные вложения, определенные с использованием функций синуса и косинуса разных частот. Матрица позиционных вложений  $P$  определена в уравнении 16.2 и представлена внизу на рис. 16.9 (транспонированная), где  $P_{p,i}$  —  $i$ -тый компонент вложения для слова, находящегося в  $p$ -той позиции внутри предложения.

**Уравнение 16.2. Позиционные вложения, определенные с применением функций синуса и косинуса**

$$P_{p,2i} = \sin(p/10000^{2i/d})$$

$$P_{p,2i+1} = \cos(p/10000^{2i/d})$$



**Рис. 16.9. Матрица позиционных вложений, определенных с использованием функций синуса и косинуса (транспонированная, вверху) с фокусом на двух значениях  $i$  (внизу)**

Такое решение показывает ту же самую эффективность, что и изученные позиционные вложения, но оно может быть расширено на произвольно длинные предложения, отчего ему отдают предпочтение. После сложения позиционных вложений и вложений слов оставшаяся часть модели имеет доступ к абсолютной позиции каждого слова в предложении, потому что для каждой позиции имеется уникальное позиционное вложение (например, позиционное вложение для слова, находящегося в 22-й позиции внутри предложения, представлено вертикальной пунктирной линией слева внизу на рис. 16.9, и вы видите, что в данной позиции оно уникально). Кроме того, выбор колебательных функций (синуса и косинуса) позволяет модели узнавать также относительные позиции. Скажем, слова, отстоящие друг от друга на 38 слов (например, расположенные в позициях  $p = 22$  и  $p = 60$ ) всегда имеют одинаковые значения позиционных вложений в измерениях вложений  $i = 100$  и  $i = 101$ , как видно на рис. 16.9. Это объясняет, почему для каждой частоты нам необходим и синус, и косинус: если бы мы применяли только синус (сияя волна при  $i = 100$ ), тогда модель не имела бы возможности проводить различие между позициями  $p = 25$  и  $p = 35$  (помечено крестиком).

В TensorFlow нет слоя вроде `PositionalEmbedding`, но его легко создать. По соображениям эффективности мы предварительно вычисляем матрицу позиционных вложений в конструкторе (поэтому нам нужно знать максимальную длину предложения, `max_steps`, и количество измерений для каждого представления слова, `max_dims`). Затем метод `call()` обрезает такую матрицу вложений до размера входов и складывает ее с входами. Поскольку при создании матрицы позиционных вложений мы добавляем дополнительное первое измерение размера 1, правила ретрансляции обеспечивают добавление матрицы к каждому предложению во входах:

```
class PositionalEmbedding(keras.layers.Layer):
    def __init__(self, max_steps, max_dims, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        if max_dims % 2 == 1: max_dims += 1      # значение max_dims
                                                # обязано быть четным
        p, i = np.meshgrid(np.arange(max_steps),
                           np.arange(max_dims // 2))
        pos_emb = np.empty((1, max_steps, max_dims))
        pos_emb[0, :, ::2] = np.sin(p / 10000**2 * i / max_dims).T
        pos_emb[0, :, 1::2] = np.cos(p / 10000**2 * i / max_dims).T
        self.positional_embedding =
            tf.constant(pos_emb.astype(self.dtype))
```

```
def call(self, inputs):
    shape = tf.shape(inputs)
    return inputs+self.positional_embedding[:, :, shape[-2], :shape[-1]]
```

Далее мы можем создать первые слои в архитектуре “Преобразователь”:

```
embed_size = 512; max_steps = 500; vocab_size = 10000
encoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
decoder_inputs = keras.layers.Input(shape=[None], dtype=np.int32)
embeddings = keras.layers.Embedding(vocab_size, embed_size)
encoder_embeddings = embeddings(encoder_inputs)
decoder_embeddings = embeddings(decoder_inputs)
positional_encoding = PositionalEncoding(max_steps,
                                           max_dims=embed_size)
encoder_in = positional_encoding(encoder_embeddings)
decoder_in = positional_encoding(decoder_embeddings)
```

А теперь давайте углубимся в самый основной компонент модели “Преобразователь”: слой многоголового внимания.

## Многоголовое внимание

Чтобы понять, как работает слой многоголового внимания, сначала мы должны разобраться в слое внимания масштабированного скалярного произведения (*Scaled Dot-Product Attention*), на котором он основан. Предположим, что кодировщик проанализировал входное предложение “They played chess” (“Они играли в шахматы”) и сумел понять, что слово “They” (“Они”) является субъектом, а слово “played” (“играли”) — глаголом, поэтому он за-кодировал такую информацию в представлениях указанных слов. Пусть де-кодировщик уже перевел субъект (*subject*) и считает, что следующим должен перевести глагол (*verb*). Для этого ему необходимо извлечь глагол из входно-го предложения, что аналогично поиску в словаре: как если бы кодировщик создал словарь {“*subject*”: “They”, “*verb*”: “played”, ...} и декодировщику нужно было найти значение, которое соответствует ключу “*verb*”. Однако модель не располагает отдельными маркерами для представления ключей (типа “*sub-ject*” или “*verb*”); она имеет векторизованные представления таких концепций (которые узнает во время обучения), так что ключ, используемый для поиска (называемый *запросом*), не будет полностью совпадать с любым ключом в словаре. Решение предусматривает вычисление меры подобия между запро-сом и каждым ключом в словаре и применение многоголовенной (*softmax*) функции для преобразования таких показателей подобия в веса, дающие в сумме 1. Если ключ, который представляет глагол, явно наиболее подобен

запросу, тогда вес этого ключа будет близок к 1. Затем модель может рассчитать взвешенную сумму соответствующих значений, и если вес ключа "verb" близок к 1, то взвешенная сумма окажется очень близкой к представлению слова "played". Короче говоря, вы можете думать о таком процессе как о дифференцируемом поиске в словаре. Используемая моделью "Преобразователь" мера подобия является просто скалярным произведением, как во внимании Лыонга. На самом деле уравнение то же, что и для внимания Лыонга, за исключением масштабного коэффициента. Его векторизованная форма показана в уравнении 16.3.

### Уравнение 16.3. Внимание масштабированного скалярного произведения

$$\text{Внимание } (\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left( \frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{ключей}}}} \right) \mathbf{V}$$

Разберем уравнение 16.3.

- $\mathbf{Q}$  — матрица, содержащая по одной строке на запрос. Имеет форму  $[n_{\text{запросов}}, d_{\text{ключей}}]$ , где  $n_{\text{запросов}}$  — количество запросов и  $d_{\text{ключей}}$  — количество измерений каждого запроса и каждого ключа.
- $\mathbf{K}$  — матрица, содержащая по одной строке на ключ. Имеет форму  $[n_{\text{ключей}}, d_{\text{ключей}}]$ , где  $n_{\text{ключей}}$  — количество ключей и значений.
- $\mathbf{V}$  — матрица, содержащая по одной строке на значение. Имеет форму  $[n_{\text{ключей}}, d_{\text{значений}}]$ , где  $d_{\text{значений}}$  — количество каждого значения.
- Формой  $\mathbf{Q}\mathbf{K}^T$  является  $[n_{\text{запросов}}, n_{\text{ключей}}]$ : здесь содержится по одному показателю подобия для каждой пары запрос/ключ. Выход многопараметрной функции имеет ту же самую форму, но все строки дают в сумме 1. Финальный выход имеет форму  $[n_{\text{запросов}}, d_{\text{значений}}]$ : есть по одной строке на запрос, где каждая строка представляет результат запроса (взвешенную сумму значений).
- Масштабный коэффициент понижает показатели подобия, чтобы избежать насыщения многопараметрной функции, что могло бы привести к крошечным градиентам.
- Некоторые пары ключ/значение можно маскировать, добавляя очень большое отрицательное значение к соответствующим показателям подобия непосредственно перед вычислением многопараметрной функции. Это полезно в слое маскированного многоголового внимания.

В кодировщике уравнение 16.3 применяется к каждому входному предложению в пакете с **Q**, **K** и **V**, равными списку слов во входном предложении (а потому каждое слово в предложении будет сравниваться с каждым словом в том же предложении, включая самого себя). Аналогично в слое маскированного многоголового внимания декодировщика уравнение 16.3 будет применяться к каждому целевому предложению в пакете с **Q**, **K** и **V**, равными списку слов в целевом предложении, на этот раз с использованием маски, чтобы предотвратить сравнение любого слова со словами, находящимися после него (во время выведения декодировщик будет иметь доступ только к уже выданным, но не к будущим словам, так что во время обучения мы обязаны маскировать будущие выходные маркеры). В верхнем слое внимания декодировщика ключи **K** и значения **V** являются просто списками кодировок слов, выпускаемых кодировщиком, а запросы **Q** — списком кодировок слов, который произведен декодировщиком.

Слой `keras.layers.Attention` реализует внимание масштабированного скалярного произведения, эффективно применяя уравнение 16.3 к множеству предложений в пакете. Его входы подобны **Q**, **K** и **V** за исключением дополнительного измерения пакетов (первое измерение).



В библиотеке TensorFlow, если **A** и **B** — тензоры с более чем двумя измерениями (скажем, формы  $[2, 3, 4, 5]$  и  $[2, 3, 5, 6]$ ), тогда `tf.matmul(A, B)` будет трактовать такие тензоры как массивы  $2 \times 3$ , где каждая ячейка содержит матрицу, и перемножать соответствующие матрицы: матрица в  $i$ -той строке и  $j$ -том столбце в **A** будет умножаться на матрицу в  $i$ -той строке и  $j$ -той столбце в **B**. Поскольку произведение матрицы  $4 \times 5$  и матрицы  $5 \times 6$  будет матрицей  $4 \times 6$ , то `tf.matmul(A, B)` возвратит массив формы  $[2, 3, 4, 6]$ .

Если мы проигнорируем обходящие связи, слои нормализации, модули “упреждения” и тот факт, что это внимание масштабированного скалярного произведения, а не в точности многоголовое внимание, тогда остаток модели “Преобразователь” может быть реализован примерно так:

```
Z = encoder_in
for N in range(6):
    Z = keras.layers.Attention(use_scale=True)([Z, Z])
encoder_outputs = Z
Z = decoder_in
```

```

for N in range(6):
    Z = keras.layers.Attention(use_scale=True, causal=True)([Z, Z])
    Z = keras.layers.Attention(use_scale=True)([Z, encoder_outputs])

outputs = keras.layers.TimeDistributed(
    keras.layers.Dense(vocab_size, activation="softmax"))(Z)

```

Аргумент `use_scale=True` создает дополнительный параметр, который позволяет слою узнать, как надлежащим образом понижать показатели подобия. Это слегка отличается от модели “Преобразователь”, которая всегда понижает показатели подобия на тот же самый коэффициент ( $\sqrt{d_{\text{ключей}}}$ ). При создании второго слоя внимания аргумент `causal=True` гарантирует, что каждый выходной маркер уделяет внимание только предыдущим, но не будущим выходным маркерам.

Настало время взглянуть на финальный фрагмент головоломки: что такое слой многоголового внимания? Его архитектура изображена на рис. 16.10.

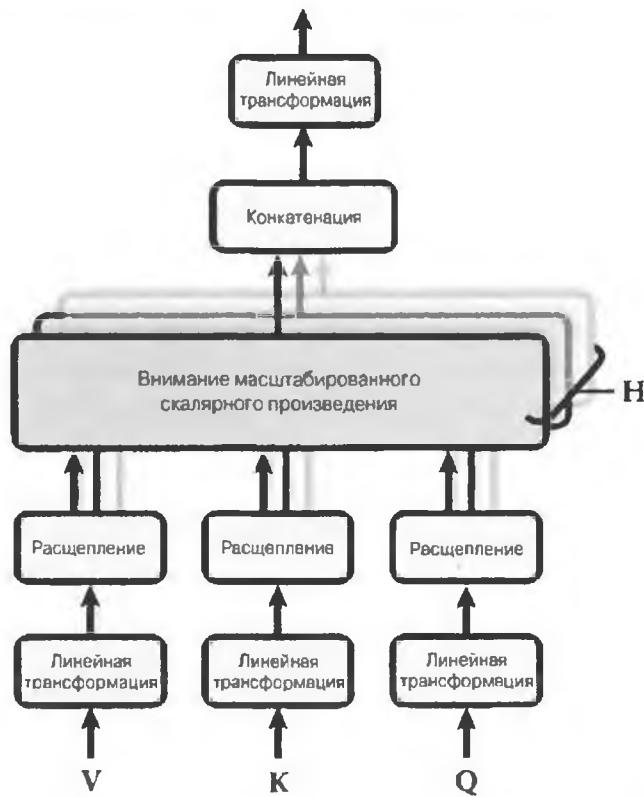


Рис. 16.10. Архитектура слоя многоголового внимания<sup>23</sup>

<sup>23</sup> Правая часть рисунка 2 в статье Ашиша Васвани и др. Воспроизводится с разрешения авторов.

- Как видите, слой многоголового внимания представляет собой просто группу слоев внимания масштабированного скалярного произведения, каждый из которых предваряется линейной трансформацией значений, ключей и запросов (т.е. распределенным во времени слоем Dense без функции активации). Все выходы подвергаются конкатенации и проходят через последнюю линейную трансформацию (снова распределенную во времени). Но почему? Что лежит в основе данной архитектуры? Возьмем слово “played” (“играли”), которое мы обсуждали ранее (в предложении “They played chess” (“Они играли в шахматы”)). Кодировщик был достаточно интеллектуален, чтобы закодировать тот факт, что оно является глаголом. Но благодаря позиционному кодированию представление слова также включает его позицию в тексте и вероятно многие другие признаки, которые полезны для его перевода, такие как тот факт, что оно находится в прошедшем времени. Короче говоря, представление слова кодирует много разных характеристик слова. Если мы просто используем единственный слой внимания масштабированного скалярного произведения, то сможем запрашивать только все эти характеристики одновременно. Именно потому слой многоголового внимания применяет множество разных линейных трансформаций значений, ключей и запросов: модель становится способной применять много разных проекций представления слова на различные подпространства, каждая из которых фокусируется на подмножестве характеристик слова. Возможно, один из слоев линейной трансформации будет проецировать представление слова на подпространство, где остается лишь информация о том, что слово является глаголом, еще один слой линейной трансформации будет извлекать только тот факт, что оно находится в прошедшем времени, и т.д. Затем слои внимания масштабированного скалярного произведения реализуют стадию поиска и в заключение выполняют конкатенацию всех результатов и проецируют их обратно в исходное пространство.

На момент написания главы не было классов вроде `Transformer` или `MultiHeadAttention`, доступных для `TensorFlow 2`. Тем не менее, ознакомьтесь с великолепным руководством `TensorFlow` по построению модели “Преобразователь” для понимания естественного языка (<https://hml.info/transformertuto>). Кроме того, команда разработчиков `TF Hub` в настоящее время занимается переносом в `TensorFlow 2` ряда модулей, основанных на архитектуре “Преобразователь”, и вскоре они станут доступ-

ными. Между тем, я надеюсь, что продемонстрировал вам то, что самостоятельно реализовать модель “Преобразователь” не особенно трудно, к тому же это отличное упражнение!

## Последние новшества в языковых моделях

Год 2018 назвали “временем ImageNet для NLP”: прогресс был поразительным с все более и более крупными архитектурами на основе LSTM и “Преобразователя”, обучаемыми на огромных наборах данных. Я настоятельно рекомендую просмотреть следующие статьи, опубликованные в 2018 году.

- В статье по ELMo (<https://homl.info/elmo>)<sup>24</sup> Мэттью Питерса и др. вводятся вложения из языковых моделей (*Embeddings from Language Models — ELMo*): они представляют собой контекстуализированные вложения слов, выясняемые из внутренних состояний глубокой двунаправленной языковой модели. Например, слово “queen” (“королева”) не будет иметь то же самое вложение в “Queen of the United Kingdom” (“королева Соединенного Королевства”) и в “queen bee” (“пчелиная матка”).
- В статье об ULMFiT (*Universal Language Model Fine-tuning* — точная настройка универсальных языковых моделей; <https://homl.info/ulmfit>)<sup>25</sup> Джереми Ховарда и Себастьяна Рудера продемонстрирована эффективность предварительного обучения без учителя для задач NLP: авторы обучали языковую модель LSTM, используя обучение со своим учителем (т.е. автоматически генерируя метки из данных) на гигантском корпусе текстов, и затем точно настраивали ее на разнообразных задачах. Их модель превзошла современное состояние на шести задачах классификации с большим отрывом (в большинстве случаев сокращая частоту ошибок на 18–24%). Кроме того, они показали, что за счет точной настройки предварительно обученной модели всего лишь на 100 помеченных образцах удалось достичь такой же эффективности, как у модели, обученной с нуля на 10 000 образцов.

<sup>24</sup> Мэттью Питерс и др., *Deep Contextualized Word Representations* (Глубокие контекстуализированные представления слов), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies 1* (2018 г.): с. 2227–2237.

<sup>25</sup> Джереми Ховард и Себастьян Рудер, *Universal Language Model Fine-Tuning for Text Classification* (Точная настройка универсальных языковых моделей для классификации текста), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics 1* (2018 г.): с. 328–339.

- Статья о GPT (Generative Pre-Training — порождающее предварительное обучение; <https://homl.info/gpt>)<sup>26</sup> Алека Рэдфорда и других исследователей из OpenAI также продемонстрировала эффективность предварительного обучения без учителя, но с применением архитектуры, похожей на “Преобразователь”. Авторы предварительно обучали крупную, но довольно простую архитектуру, которая содержала стопку из 12 модулей “Преобразователь” (использующих только слои маскированного многоголового внимания), на большом наборе данных снова с помощью обучения со своим учителем. Затем они точно настраивали ее на разнообразных языковых задачах с применением только незначительных адаптаций для каждой задачи. Задачи были довольно-таки несходными: они включали классификацию текста, логическое следование (*entailment*; вытекает ли из предложения А предложение В)<sup>27</sup>, подобие (например, предложение “Nice weather today” (“Сегодня хорошая погода”) крайне подобно предложению “It is sunny” (“Солнечно”)) и ответы на вопросы (располагая несколькими абзацами текста, дающими определенный контекст, модель должна ответить на ряд вопросов с множественным выбором). Спустя несколько месяцев, в феврале 2019 года, Алек Рэдфорд, Джейфри Ву и другие исследователи из OpenAI опубликовали статью о GPT-2 (<https://homl.info/gpt2>)<sup>28</sup>, где предложили очень похожую архитектуру, но еще крупнее (более 1.5 миллиардов параметров!), и они показали, что могли бы добиться хорошей эффективности на многих задачах без какой-либо точной настройки. Это называется обучением без подготовки (*zero-shot learning — ZSL*). Уменьшенная версия модели GPT-2 (со “всего лишь” 117 миллионами параметров) наряду с заранее обученными весами доступна по адресу <https://github.com/openai/gpt-2>.

<sup>26</sup> Алек Рэдфорд и др., *Improving Language Understanding by Generative Pre-Training* (Улучшение понимания естественного языка за счет порождающего предварительного обучения) (2018 г.).

<sup>27</sup> Например, из предложения “Jane had a lot of fun at her friend’s birthday party” (“Джейн здорово повеселилась на вечеринке по случаю дня рождения своей подруги”) вытекает предложение “Jane enjoyed the party” (“Джейн получила удовольствие от вечеринки”), но оно противоречит “Everyone hated the party” (“Все ненавидят вечеринку”) и не соотносится с “The Earth is flat” (“Земля плоская”).

<sup>28</sup> Алек Рэдфорд и др., *Language Models Are Unsupervised Multitask Learners* (Языковые модели являются многозадачными учениками без учителя) (2019 г.).

- Статья о BERT (Bidirectional Encoder Representations from Transformers — двунаправленные представления кодировщиков из преобразователей; <https://hml.info/bert>)<sup>29</sup> Якоб Девлин и другие исследователи из Google тоже продемонстрировали эффективность предварительного обучения со своим учителем на крупном корпусе текстов с использованием архитектуры, которая похожа на GPT, но не маскирует слои многоголового внимания (вроде кодировщика в архитектуре “Преобразователь”). Это значит, что модель является естественным образом двунаправленной (bidirectional), отсюда “B” в BERT. Более важно то, что авторы предложили две задачи предварительного обучения, которые объясняют большую часть сильной стороны модели.

### *Маскированная языковая модель (masked language model — MLM)*

С каждым словом в предложении связана 15%-ная вероятность быть маскированным, и модель обучается прогнозированию маскированных слов. Например, если исходным предложением является “She had fun at the birthday party” (“Она здорово повеселилась на вечеринке по случаю дня рождения”), тогда модели может быть передано предложение “She <mask> fun at the <mask> party” и она должна спрогнозировать слова “had” и “birthday” (другие выходы будут игнорироваться). Говоря точнее, каждое выбранное слово имеет 80%-ную вероятность быть замаскированным, 10%-ную вероятность быть замененным случайным словом (чтобы уменьшить несходство между предварительным обучением и точной настройкой, т.к. модель не будет видеть маркеры <mask> во время точной настройки) и 10%-ную вероятность остаться в одиночестве (чтобы сместить модель в сторону правильного ответа).

### *Прогнозирование следующего предложения (next sentence prediction — NSP)*

Модель обучается прогнозировать, являются два предложения последовательными или нет. Скажем, она должна прогнозировать, что “The dog sleeps” (“Собака спит”) и “It snores loudly”

<sup>29</sup> Якоб Девлин и др., *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* (BERT: предварительное обучение глубоких двунаправленных преобразователей для понимания естественного языка), *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies 1* (2019 г.).

(“Она громко хранит”) — последовательные предложения, тогда как “The dog sleeps” (“Собака спит”) и “The Earth orbits the Sun” (“Земля вращается вокруг Солнца”) — нет. Это сложная задача, и эффективность модели значительно улучшается, когда она точно настраивается на таких задачах, как ответы на вопросы или логическое следование.

Как видите, главными новшествами в 2018 и 2019 годах были разбиение на лексемы частей слов, переключение с архитектуры на основе LSTM на архитектуру “Преобразователь”, а также предварительное обучение универсальных языковых моделей с применением обучения со своим учителем и затем их точная настройка с помощью очень немногих архитектурных изменений (или вообще без таковых). Все быстро меняется; никто не сможет сказать, какие архитектуры будут преобладать в следующем году. Сегодня это, безусловно, архитектура “Преобразователь”, но завтра превалирующими могут стать сети CNN (например, ознакомьтесь со статьей 2018 года (<https://homl.info/pervasiveattention>)<sup>30</sup> Махи Эльбайад и др., где исследователи использовали маскированные двумерные сверточные слои для задач типа “последовательность в последовательность”). Ими могут оказаться даже сети RNN, если они неожиданно вернутся (например, почитайте статью 2018 года (<https://homl.info/indrnn>)<sup>31</sup> Шуай Ли и др., в которой показано, что за счет обеспечения независимости нейронов друг от друга на отдельно взятом слое RNN появляется возможность обучать намного более глубокие сети RNN, способные узнавать гораздо более длинные предложения).

В следующей главе мы обсудим, как обучать глубокие представления способом без учителя с применением автокодировщиков, и будем использовать порождающие состязательные сети (*generative adversarial network* — GAN) для выпуска изображений и многоного другого!

<sup>30</sup> Маха Эльбайад и др., *Pervasive Attention: 2D Convolutional Neural Networks for Sequence-to-Sequence Prediction* (Проникающее внимание: двумерные сверточные нейронные сети для прогнозирования типа “последовательность в последовательность”), препринт arXiv:1808.03867 (2018 г.).

<sup>31</sup> Шуай Ли и др., *Independently Recurrent Neural Network (IndRNN): Building a Longer and Deeper RNN* (Независимые рекуррентные нейронные сети (IndRNN): построение и углубление сети RNN), *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018 г.): с. 5457–5466.

## Упражнения

1. Приведите доводы за и против применения сети RNN с запоминанием и без запоминания состояния.
2. Почему для автоматического перевода люди используют сети RNN на основе архитектуры “кодировщик–декодировщик”, а не простые сети RNN типа “последовательность в последовательность”?
3. Как вы будете иметь дело с входными последовательностями переменной длины? Что насчет выходных последовательностей переменной длины?
4. Что такое лучевой поиск и для чего вы бы его применяли? Каким инструментом вы можете воспользоваться, чтобы реализовать его?
5. Что такое механизм внимания? Как он помогает?
6. Какой слой в архитектуре “Преобразователь” является самым важным? Для чего он предназначен?
7. Когда может возникнуть необходимость в применении многопеременного приема с выборкой?
8. В своей работе, посвященной ячейкам LSTM (<https://homl.info/93>), Хохрайтер и Шмидхубер применяли вложенные грамматики Ребера (*embedded Reber grammar*). Они являются искусственными грамматиками, которые производят строки наподобие “BPBTSXXVPSEPE”. Ознакомьтесь с хорошим введением в эту тему, написанным Дженнини Опп (<https://homl.info/108>). Выберите отдельную вложенную грамматику Ребера (одну из приведенных на странице Дженнини Опп) и обучите сеть RNN идентифицировать то, соблюдает ли строка правила выбранной грамматики. Сначала вам нужно написать функцию, способную генерировать обучающий пакет с примерно 50% строк, которые соблюдают правила грамматики, и еще примерно 50% строк, которые правила не соблюдают.
9. Обучите модель типа “кодировщик–декодировщик”, которая сможет преобразовывать строку с датой из одного формата в другой (скажем, из “April 22, 2019” в “2019-04-22”).

10. Проштудируйте руководство “Neural machine translation with attention” (“Нейронный машинный перевод с использованием внимания”) из TensorFlow (<https://homl.info/nmttuto>).
11. Воспользуйтесь одной из недавних языковых моделей (например, BERT) для генерации более убедительного шекспировского текста.

Решения приведенных упражнений доступны в приложении А.

# Обучение представлению и порождению с использованием автокодировщиков и порождающих состязательных сетей

Автокодировщики (*autoencoder*) — это искусственные нейронные сети, которые способны узнавать плотные представления входных данных, называемые латентными представлениями (*latent representation*) или кодировками (*coding*), без какого-либо учителя (т.е. обучающий набор не помечен). Обычно такие кодировки имеют намного меньшую размерность, чем входные данные, делая автокодировщики пригодными для понижения размерности (см. главу 8), особенно в целях визуализации. Автокодировщики также действуют как обнаружители признаков и могут использоваться для предварительного обучения без учителя глубоких нейронных сетей (что обсуждалось в главе 11). Наконец, некоторые автокодировщики являются порождающими моделями (*generative model*): они в состоянии случайным образом генерировать новые данные, которые выглядят очень похожими на обучающие данные. Например, вы могли бы обучить автокодировщик на фотографиях лиц, после чего он сумел бы генерировать новые лица. Однако, как правило, сгенерированные изображения оказываются нечеткими и не вполне реалистичными.

Напротив, лица, генерируемые порождающими состязательными сетями (*generative adversarial network* — *GAN*), теперь выглядят настолько убедительными, что трудно поверить в то, что они представляют несуществующих людей. Вы можете судить об этом сами, посетив веб-сайт <https://thispersondoesnotexist.com/>, на котором показаны лица, генериро-

ванные недавно появившейся архитектурой GAN под названием *StyleGAN* (загляните также на веб-сайт <https://thisrentaldoesnotexist.com/>), чтобы посмотреть на сгенерированные изображения апартаментов, бронируемых через Airbnb). В настоящее время сети GAN широко применяются для сверхразрешения (увеличения разрешения изображений), расцвечивания (<https://github.com/jantic/DeOldify>), мощного редактирования изображений (например, замены фотобомберов (тех, кто случайно или намеренно появляются на фотографиях — Примеч. пер.) реалистичным фоном), превращения простого эскиза в фотореалистичное изображение, прогнозирования последующих кадров в видеоролике, дополнения набора данных (для обучения других моделей), генерирования других типов данных (таких как текст, аудиоклип и временной ряд), выявления слабых сторон в других моделях и их укрепления, а также многое другое.

Автокодировщики и сети GAN работают по принципу без учителя, они обучаются плотным представлениям, могут использоваться как порождающие модели и имеют много похожих приложений. Тем не менее, они функционируют совершенно по-разному.

- Автокодировщики просто учатся копировать свои входы в свои выходы. Задача может показаться тривиальной, но мы увидим, что ограничение сети разнообразными путями способно сделать ее довольно трудной. Скажем, вы можете ограничить размер латентных представлений или добавить шум к входам и обучать сеть восстановлению первоначальных входов. Такие ограничения не позволяют автокодировщику тривиально копировать входы напрямую в выходы, что вынуждает его изучать эффективные способы представления данных. Короче говоря, кодировки являются побочными продуктами автокодировщика, изучающего функцию тождественности при некоторых ограничениях.
- Сети GAN состоят из двух нейронных сетей: *генератора* (*generator*), пытающегося генерировать данные, которые выглядят похожими на обучающие данные, и *дискриминатора* (*discriminator*), старающегося различать подлинные данные от фальшивых. Такая архитектура весьма своеобразна в глубоком обучении тем, что во время обучения генератор и дискриминатор соперничают друг с другом: генератор часто сравнивают с преступником, пытающимся изготовить реалистичные фальшивые деньги, тогда как дискриминатор подобен полицейскому следователю, который старается отличить подлинные деньги от фальшивых.

*Состязательное (adversarial) обучение* (обучение соперничающих нейронных сетей) в значительной степени считается одной из наиболее важных идей, возникших в последние годы. В 2016 году Ян Лекун даже отметил, что оно было самой интересной идеей в машинном обучении за прошедшее десятилетие.

В этой главе мы начнем с более глубокого анализа работы автокодировщиков и способов их применения для понижения размерности, выделения признаков, предварительного обучения без учителя или в качестве порождающих моделей, что естественным образом приведет нас к сетям GAN. Первым делом мы построим простую сеть GAN для генерации поддельных изображений, но увидим, что обучение часто оказывается довольно сложным. Мы обсудим главные затруднения, с которыми вы столкнетесь при состязательном обучении, а также несколько основных приемов для обхода возникающих затруднений. Давайте займемся автокодировщиками!

## Эффективные представления данных

Какую из следующих последовательностей чисел вы считаете более легкой для запоминания?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

Поначалу может показаться, что первую последовательность запомнить легче, поскольку она намного короче. Однако если вы пристальнее взглянете на вторую последовательность, то заметите, что она является просто списком четных чисел от 50 до 14. Как только вы заметите этот шаблон, вторая последовательность станет гораздо более легкой для запоминания, нежели первая, потому что вам необходимо запомнить только шаблон (т.е. уменьшающиеся четные числа) плюс начальное и конечное числа (т.е. 50 и 14). Обратите внимание, что если бы вы могли быстро и легко запоминать очень длинные последовательности, то вас особо не волновало бы существование шаблона во второй последовательности. Вы просто бы выучили все числа наизусть и дело с концом. Именно факт того, что нам трудно запоминать длинные последовательности, обеспечивает полезность распознавания шаблонов и надо надеяться проясняет, почему ограничение автокодировщика во время обучения подталкивает его к выявлению и эксплуатации шаблонов в данных.

Взаимосвязь между памятью, восприятием и сопоставлением с шаблоном была превосходно исследована Уильямом Чейзом и Гербертом Саймоном в начале 1970-х годов (<https://homl.info/111>)<sup>1</sup>. Они заметили, что опытные шахматисты были способны запоминать позиции всех фигур в игре, просто глядя на доску в течение каких-то пяти 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, им лишь гораздо легче замечать шахматные шаблоны благодаря своему опыту игры. Выявление шаблонов помогает им эффективно запоминать информацию.

Подобно шахматистам из упомянутого эксперимента с запоминанием автокодировщик просматривает входы, преобразует их в эффективное латентное 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 в латентное представление, и следующего за ним декодировщика (или *порождающей сети (generative network)*), который преобразует латентное представление в выходы (рис. 17.1).

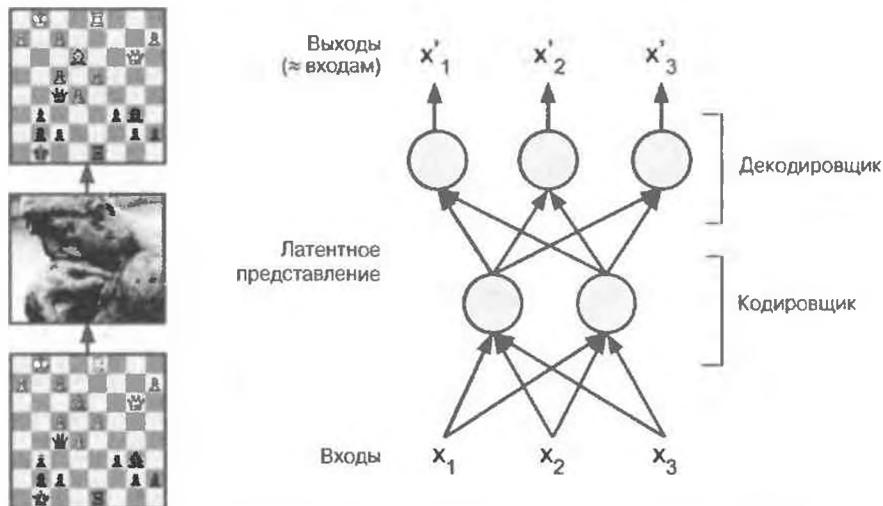


Рис. 17.1. Эксперимент с запоминанием позиций шахматных фигур (слева) и простой автокодировщик (справа)

<sup>1</sup> Уильям Чейз и Герберт Саймон, *Perception in Chess* (Восприятие в шахматах), *Cognitive Psychology* 4, номер 1 (1973 г.): с. 55–81.

Как видите, автокодировщик обычно имеет ту же самую архитектуру, что и многослойный персепtron (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), то говорят, что автокодировщик 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 узнавать самые важные признаки во входных данных (и отбрасывать несущественные признаки).

Давайте посмотрим, как реализовать очень простой понижающий автокодировщик для понижения размерности.

## Выполнение анализа главных компонентов с помощью понижающего линейного автокодировщика

Если автокодировщик использует только линейные активации и функцию издержек в виде среднеквадратической ошибки (MSE), тогда в итоге он выполняет анализ главных компонентов (PCA; см. главу 8).

Следующий код строит простой линейный автокодировщик для выполнения анализа PCA на трехмерном наборе данных, проецируя его в два измерения:

```
encoder = keras.models.Sequential([keras.layers.Dense(2,
                                                       input_shape=[3])])
decoder = keras.models.Sequential([keras.layers.Dense(3,
                                                       input_shape=[2])])
autoencoder = keras.models.Sequential([encoder, decoder])
autoencoder.compile(loss="mse",
                     optimizer=keras.optimizers.SGD(lr=0.1))
```

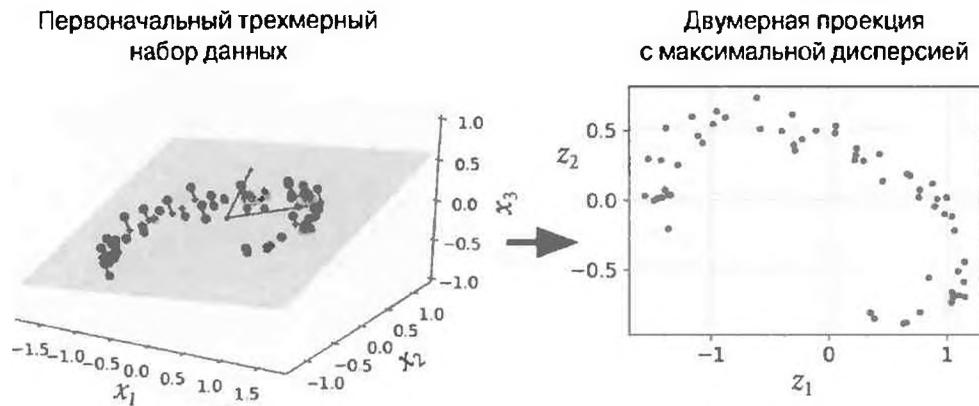
Код на самом деле не очень сильно отличается от всех многослойных персептронов, которые мы строили в предшествующих главах, но есть несколько моментов, о которых нужно упомянуть.

- Мы создали автокодировщик в виде двух подкомпонентов: кодировщика и декодировщика. Оба они представляют собой модели Sequential, каждая с единственным слоем Dense, а автокодировщик является моделью Sequential, содержащей кодировщик, за которым следует декодировщик (не забывайте, что модель может использоваться в качестве слоя в другой модели).
- Количество выходов автокодировщика равно количеству входов (т.е. 3).
- 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 давайте обучим модель на простом сгенерированном трехмерном наборе данных и используем ее для кодирования того же самого набора данных (т.е. его проецирования в два измерения):

```
history = autoencoder.fit(X_train, X_train, epochs=20)
codings = encoder.predict(X_train)
```

Обратите внимание, что тот же самый набор данных  $X_{train}$  применяется как входы и как цели. Слева на рис. 15.2 показан первоначальный 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. 17.2. Анализ PCA, выполненный понижающим линейным автокодировщиком*



Вы можете думать об автокодировщиках как о форме обучения со своим учителем (т.е. использование методики обучения с учителем при автоматической генерации меток, в этом случае просто приравнивая их к входам).

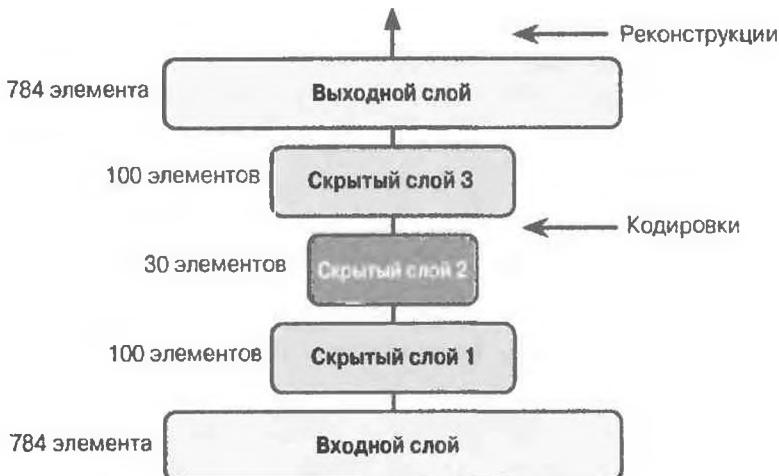
## Многослойные автокодировщики

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 он 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 набора данных MNIST (введенного в главе 3) может иметь 784 входа, за которыми следует скрытый слой из 100 нейронов, центральный скрытый слой из 30 нейронов, еще один скрытый слой из 100 нейронов и выходной слой из 784 нейронов. Такой многослойный автокодировщик показан на рис. 17.3.

### Реализация многослойного автокодировщика с использованием Keras

Многослойный автокодировщик можно реализовать способом, очень похожим на реализацию обычновенного глубокого многослойного персептрона. В частности, могут применяться те же самые приемы, которые использовались для обучения глубоких сетей в главе 11.



*Рис. 17.3. Многослойный автокодировщик*

Скажем, приведенный ниже код строит многослойный автокодировщик для набора данных Fashion MNIST (загруженного и нормализованного в главе 10) с применением функции активации SELU:

```

stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae =
    keras.models.Sequential([stacked_encoder, stacked_decoder])
stacked_ae.compile(loss="binary_crossentropy",
                    optimizer=keras.optimizers.SGD(lr=1.5))
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])

```

Пройдемся по коду.

- Как и ранее, мы разделяем модель автокодировщика на две подмодели: кодировщик и декодировщик.
- Кодировщик берет полутоновые изображения  $28 \times 28$  пикселей, разглаживает их, чтобы каждое изображение было представлено в виде вектора

ра размером 784, затем обрабатывает эти векторы, прогоняя через два слоя Dense с убывающими размерами (100 элементов и 30 элементов) и функцией активации SELU (у вас также может возникнуть желание добавить нормальную инициализацию Лекуна, но сеть не очень глубокая, а потому большой разницы не будет). Для каждого входного изображения кодировщик выдает вектор размера 30.

- Декодировщик берет кодировки размера 30 (выданные кодировщиком), прогоняет их через два слоя Dense с возрастающими размерами (100 элементов и 784 элемента) и изменяет форму финальных векторов на массивы  $28 \times 28$ , так что выходы декодировщика имеют ту же самую форму, что и входы кодировщика.
- При компиляции многослойного автокодировщика вместо среднеквадратической ошибки мы используем потерю двоичной перекрестной энтропии. Мы обращаемся с задачей реконструкции как с задачей многозначной двоичной классификации: интенсивность каждого пикселя представляет вероятность того, что пиксель должен быть черным. Позиционируя ее подобным образом (а не как задачу регрессии), мы обеспечиваем более быстрое схождение модели<sup>2</sup>.
- В заключение мы обучаем модель с применением `X_train` в качестве входов и целей (и аналогично мы используем `X_valid` как входы и цели при проверке).

## Визуализация реконструкций

Удовостовериться в том, что автокодировщик надлежащим образом обучен, можно путем сравнения входов и выходов: отличия не должны быть слишком значительными. Давайте вычертим несколько изображений из проверочного набора, а также их реконструкции:

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")
```

---

<sup>2</sup> У вас может возникнуть искушение применить метрику правильности, но она не будет работать надлежащим образом, поскольку для каждого пикселя эта метрика ожидает, что метка должна быть либо 0, либо 1. Вы можете легко обойти проблему, создав специальную метрику, которая рассчитывает правильность после округления целей и прогнозов до 0 или 1.

```

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)

```

Результатирующие изображения показаны на рис. 17.4.



*Рис. 17.4. Первоначальные изображения (вверху) и их реконструкции (внизу)*

Реконструкции узнаваемы, но потеря в них многовато. Возможно, нам придется дольше обучать модель, сделать кодировщик и декодировщик более глубокими или укрупнить кодировки. Но если мы наделим сеть чрезмерной мощью, то ей удастся вырабатывать идеальные реконструкции, не узнавая никаких полезных шаблонов в данных. А пока мы продолжим с имеющейся моделью.

## Визуализация набора данных Fashion MNIST

Теперь, когда многослойный автокодировщик обучен, мы можем использовать его для понижения размерности набора данных. В случае визуализации это не обеспечит выдающиеся результаты по сравнению с другими алгоритмами понижения размерности (вроде тех, которые обсуждались в главе 8), но крупное преимущество автокодировщиков в том, что они могут обрабатывать большие наборы данных со многими образцами и многими признаками. Таким образом, одна из стратегий предусматривает применение автокодировщика для понижения размерности до приемлемого уровня и затем еще одного алгоритма понижения размерности для визуализации.

Давайте воспользуемся такой стратегией для визуализации набора данных Fashion MNIST. Сначала мы применяем кодировщик из нашего многослойного автокодировщика для понижения размерности до 30, после чего используем реализацию алгоритма t-SNE из Scikit-Learn, чтобы понизить размерность до 2 для визуализации:

```
from           import TSNE  
X_valid_compressed = stacked_encoder.predict(X_valid)  
tsne = TSNE()  
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

Далее можно вычертить набор данных:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1],  
            c=y_valid, s=10, cmap="tab10")
```

На рис. 17.5 показан результирующий график рассеяния (слегка украшенный отображением некоторых изображений). Алгоритм t-SNE идентифицировал несколько кластеров, которые достаточно хорошо соответствуют классам (каждый класс представлен отличающимся цветом).



*Рис. 17.5. Визуализация набора данных Fashion MNIST с применением автокодировщика и затем алгоритма t-SNE*

Итак, автокодировщики могут использоваться для понижения размерности. Еще одно их приложение касается предварительного обучения без учителя.

## 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 высокоэффективную модель с применением малого объема обучающих данных, поскольку ваша нейронная сеть не нуждается в изучении всех низкоуровневых признаков; она просто повторно использует обнаружители признаков, выученные 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. 17.6 показано, как с использованием многослойного автокодировщика провести предварительное обучение без учителя для классификационной нейронной сети. Когда обучается классификатор, а помеченных обучающих данных не особенно много, может возникнуть желание заморозить заранее обученные слои (во всяком случае, самые нижние из них).



Наличие массы непомеченных и небольшого объема помеченных данных — распространенная ситуация. Построение крупных непомеченных наборов данных часто обходится недорого (скажем, простой сценарий способен загрузить миллионы изображений из Интернета), но пометка таких изображений (например, классификация изображений как привлекательных или нет) обычно может надежно производиться только людьми. Пометка образцов является отнимающей много времени и дорогостоящей процедурой, поэтому вполне нормально иметь только несколько тысяч образцов, помеченных людьми.



**Рис. 17.6. Предварительное обучение без учителя с использованием автокодировщиков**

В реализации нет ничего особенного: просто обучите автокодировщик с применением всех обучающих данных (помеченных плюс непомеченных) и затем повторно используйте его слои кодировщика для создания новой нейронной сети (за примером обращайтесь к упражнениям в конце главы).

Далее мы рассмотрим несколько методик для обучения многослойных автокодировщиков.

## Соединение весов

Когда автокодировщик является четко симметричным, подобно только что построенному, распространенная методика заключается в соединении (*i.e.*) весов слоев декодировщика с весами слоев кодировщика. В итоге количество весов уменьшается вдвое, что ускоряет обучение и ограничивает риск переобучения. В частности, если автокодировщик суммарно имеет  $N$  слоев (не считая входного), а  $\mathbf{W}_L$  представляет веса связей  $L$ -того слоя (скажем, слой 1 является первым скрытым слоем, слой  $\frac{N}{2}$  — слоем кодировки и слой  $N$  — выходным слоем), тогда веса слоев декодировщика могут быть определены просто как  $\mathbf{W}_{N-L+1} = \mathbf{W}_L^T$  (при  $L = 1, 2, \dots, \frac{N}{2}$ ).

Чтобы соединить веса между слоями с применением Keras, определим специальный слой:

```

class (keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        self.dense = dense
        self.activation = keras.activations.get(activation)
        super().__init__(**kwargs)
    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias", initializer="zeros",
                                      shape=[self.dense.input_shape[-1]])
        super().build(batch_input_shape)
    def call(self, inputs):
        z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(z + self.biases)

```

Этот специальный слой действует подобно обыкновенному слою Dense, но использует веса еще одного слоя Dense, транспонированные (установка transpose\_b=True эквивалентна транспонированию второго аргумента, но более эффективна, т.к. выполняет транспонирование на лету внутри операции matmul()). Тем не менее, он применяет собственный вектор смещения. Далее мы можем построить новый многослойный автокодировщик во многом подобно предыдущему, но со слоями Dense декодировщика, соединенными со слоями Dense кодировщика:

```

dense_1 = keras.layers.Dense(100, activation="selu")
dense_2 = keras.layers.Dense(30, activation="selu")

tied_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    dense_1,
    dense_2
])

tied_decoder = keras.models.Sequential([
    DenseTranspose(dense_2, activation="selu"),
    DenseTranspose(dense_1, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

tied_ae = keras.models.Sequential([tied_encoder, tied_decoder])

```

Новая модель достигает чуть более низкой ошибки реконструкции, чем предыдущая, с помощью почти половины от количества параметров.

## Обучение одного автокодировщика за раз

Вместо обучения целого многослойного автокодировщика в один присест, как мы только что поступали, допускается обучать по одному неглубокому

автокодировщику за раз, после чего уложить все их стопкой, образовав многослойный автокодировщик (рис. 17.7). В наши дни такая методика используется не настолько часто, но вы по-прежнему можете столкнуться со статьями, где речь идет о “жадном послойном обучении”, и потому неплохо знать, что под ним подразумевается.

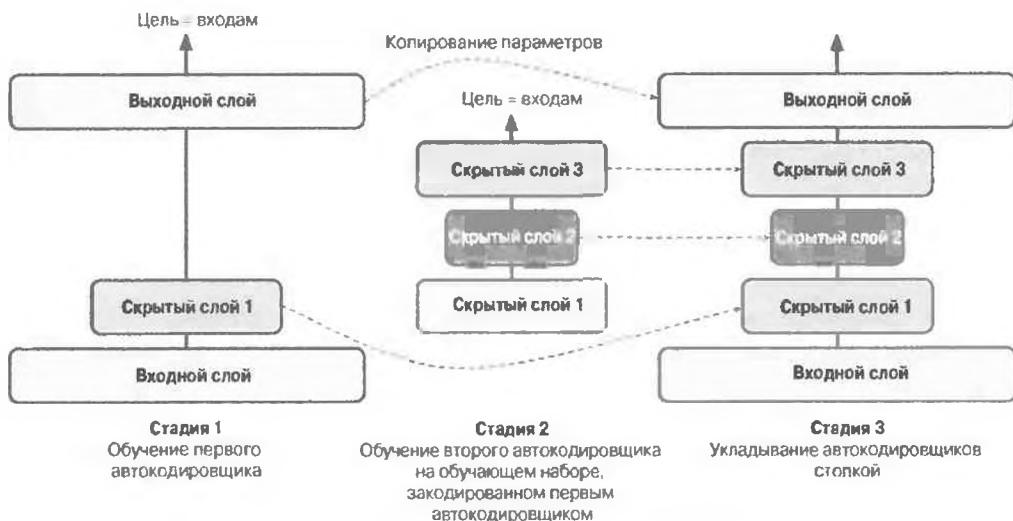


Рис. 17.7. Обучение по одному автокодировщику за раз

В течение первой стадии обучения первый автокодировщик учится реконструировать входы. Затем мы кодируем полный обучающий набор с применением первого автокодировщика, что дает нам новый (сжатый) обучающий набор. Далее мы обучаем на новом наборе данных второй автокодировщик. Это вторая стадия обучения. Наконец, мы строим большой “бутерброд”, используя все автокодировщики, как показано на рис. 17.7 (т.е. укладываем стопкой скрытые слои каждого автокодировщика и затем выходные слои в обратном порядке). В результате мы получаем финальный многослойный автокодировщик (реализацию ищите в разделе “Training One Autoencoder at a Time” (Обучение одного автокодировщика за раз) тетради Jupiter для настоящей главы). Подобным образом мы могли бы обучать и больше автокодировщиков, выстраивая очень глубокий многослойный автокодировщик.

Как обсуждалось ранее, одним из пусковых механизмов текущей волны интереса к глубокому обучению было совершенное в 2006 году Джекфри Хинтоном и др. открытие (<https://homl.info/136>), что глубокие ней-

ронные сети могут быть предварительно обучены в манере без учителя с применением жадного послойного подхода. Для такой цели они использовали ограниченные машины Больцмана (RBM; см. приложение Д), но в 2007 году Йошуа Бенджи и др. показали (<https://homl.info/112>)<sup>3</sup>, что автокодировщики работают в той же степени хорошо. В течение нескольких лет это был единственный эффективный способ обучения глубоких сетей, пока многие методики, представленные в главе 11, не сделали возможным обучение глубокой сети за один заход.

Автокодировщики не ограничены плотными сетями: вы также можете строить сверточные или даже рекуррентные автокодировщики. Давайте взглянем на них.

## Сверточные автокодировщики

Если вы имеете дело с изображениями, тогда рассмотренные до сих пор автокодировщики не будут с ними эффективно работать (разве что изображения окажутся очень маленькими): как отмечалось в главе 14, сверточные сети гораздо лучше подходят для работы с изображениями, чем плотные сети. Таким образом, если вы хотите построить автокодировщик для изображений (например, для предварительного обучения без учителя или понижения размерности), то должны создать *сверточный автокодировщик* (<https://homl.info/convae>)<sup>4</sup>. Кодировщик является обыкновенной сетью CNN, состоящей из сверточных и объединяющих слоев. Обычно он понижает пространственную размерность входов (т.е. высоту и ширину), одновременно увеличивая глубину (т.е. количество карт признаков). Декодировщик обязан выполнять обратные действия (увеличивать масштаб изображения и сокращать его глубину до первоначальных измерений), для чего применяются транспонированные сверточные слои (в качестве альтернативы вы могли бы скомбинировать слои повышения дискретизации со сверточными слоями). Вот простой сверточный автокодировщик для набора данных Fashion MNIST:

<sup>3</sup> Йошуа Бенджи и др., *Greedy Layer-Wise Training of Deep Networks* (Жадное послойное обучение глубоких сетей), *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006 г.): с. 153–160.

<sup>4</sup> Джонатан Маски и др., *Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction* (Многослойные сверточные автокодировщики для выделения иерархических признаков), *Proceedings of the 21st International Conference on Artificial Neural Networks 1* (2011 г.): с. 52–59.

```

conv_encoder = keras.models.Sequential([
    keras.layers.Reshape([28, 28, 1], input_shape=[28, 28]),
    keras.layers.Conv2D(16, kernel_size=3,
                      padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(32, kernel_size=3,
                      padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2),
    keras.layers.Conv2D(64, kernel_size=3,
                      padding="same", activation="selu"),
    keras.layers.MaxPool2D(pool_size=2)
])
conv_decoder = keras.models.Sequential([
    keras.layers.Conv2DTranspose(32, kernel_size=3,
                               strides=2, padding="valid",
                               activation="selu",
                               input_shape=[3, 3, 64]),
    keras.layers.Conv2DTranspose(16, kernel_size=3,
                               strides=2, padding="same",
                               activation="selu"),
    keras.layers.Conv2DTranspose(1, kernel_size=3,
                               strides=2, padding="same",
                               activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
conv_ae = keras.models.Sequential([conv_encoder, conv_decoder])

```

## Рекуррентные автокодировщики

Если вы хотите построить автокодировщик для последовательностей, таких как временные ряды или текст (скажем, для обучения без учителя либо понижения размерности), то рекуррентные нейронные сети (см. главу 15) могут подойти лучше, чем плотные сети. Создать *рекуррентный автокодировщик* довольно легко: кодировщиком обычно будет сеть RNN типа “последовательность в вектор”, которая сжимает входную последовательность в одиночный вектор, а декодировщиком — сеть RNN типа “вектор в последовательность”, выполняющая противоположное действие:

```

recurrent_encoder = keras.models.Sequential([
    keras.layers.LSTM(100, return_sequences=True,
                      input_shape=[None, 28]),
    keras.layers.LSTM(30)
])

```

```
recurrent_decoder = keras.models.Sequential([
    keras.layers.RepeatVector(28, input_shape=[30]),
    keras.layers.LSTM(100, return_sequences=True),
    keras.layers.TimeDistributed(
        keras.layers.Dense(28, activation="sigmoid"))
])
recurrent_ae = keras.models.Sequential([recurrent_encoder,
                                         recurrent_decoder])
```

Такой рекуррентный автокодировщик способен обрабатывать последовательности любой длины с 28 измерениями на временной шаг. Это удобно, т.к. означает, что он может обработать изображения Fashion MNIST, трактуя каждое изображение как последовательность строк: на каждом временном шаге сеть RNN будет обрабатывать одну строку из 28 пикселей. Очевидно, вы могли бы использовать рекуррентный автокодировщик для последовательности любого вида. Обратите внимание, что мы применяем слой `RepeatVector` в качестве первого слоя декодировщика для гарантирования того, что его входной вектор подается декодировщику на каждом временном шаге.

Давайте ненадолго отвлечемся. Ранее мы видели разнообразные типы декодировщиков (базовый, многослойный, сверточный и рекуррентный) и выяснили, как их обучать (либо в один присест, либо слой за слоем). Мы также взглянули на пару приложений: визуализация данных и предварительное обучение без учителя.

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 использовать, включая те, что позволяют кодирующему слою быть таким же большим, как входы, или даже больше, давая в результате *повышающий автокодировщик (overcomplete autoencoder)*. Ниже мы 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. Идея использования автокодировщиков для подавления шума существует с 1980-х годов (например, в 1987 году Ян Лекун упомянул о ней в своей кандидатской диссертации).

В статье 2008 года (<https://homl.info/113>)<sup>5</sup> Паскаль Винсент и др. показали, что автокодировщики могли бы применяться также для выделения признаков. В статье 2010 года (<https://homl.info/114>)<sup>6</sup> Винсент и др. представили *многослойные шумоподавляющие автокодировщики (stacked denoising autoencoders)*.

Шум может быть чистым гауссовым шумом, добавленным к входам, или случайнym выключением входов подобно отключению (введенному в главе 11). На рис. 17.8 приведены оба варианта.

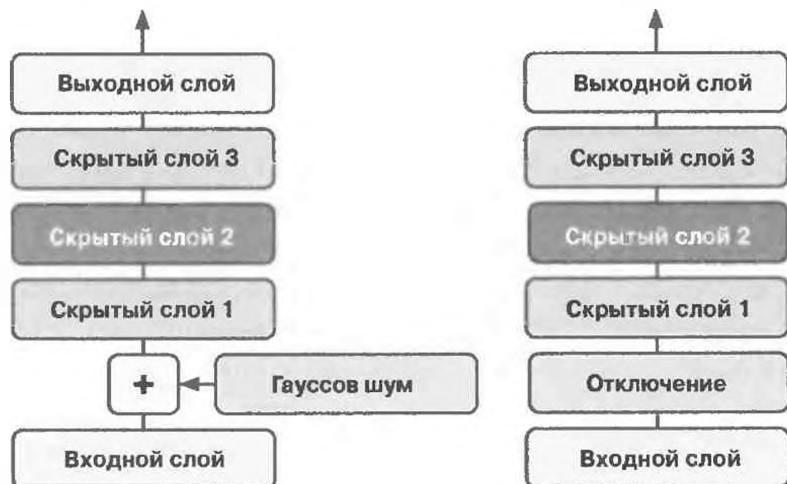


Рис. 17.8. Шумоподавляющие автокодировщики с гауссовым шумом (слева) или отключением (справа)

Реализация прямолинейна: это обычновенный многослойный автокодировщик с дополнительным слоем Dropout, примененным к входам кодировщика (или взамен можно было бы использовать слой GaussianNoise). Вспомните, что слой Dropout активен только во время обучения (как и слой GaussianNoise):

<sup>5</sup> Паскаль Винсент и др., *Extracting and Composing Robust Features with Denoising Autoencoders* (Выделение и компоновка устойчивых признаков с помощью шумоподавляющих автокодировщиков), *Proceedings of the 25th International Conference on Machine Learning* (2008 г.): с. 1096–1103.

<sup>6</sup> Паскаль Винсент и др., *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), *Journal of Machine Learning Research 11* (2010 г.): с. 3371–3408.

```

dropout_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu")
])
dropout_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
dropout_ae = keras.models.Sequential([dropout_encoder,
                                      dropout_decoder])

```

На рис. 17.9 показано несколько зашумленных изображений (с половиной отключенных пикселей), а также изображения, реконструированные шумоподавляющим автокодировщиком на основе отключения. Обратите внимание на то, каким образом автокодировщик угадывает детали, которых на самом деле нет во входе, такие как верх белой рубашки (нижняя строка, четвертое изображение). Как видите, шумоподавляющие автокодировщики можно применять не только для визуализации данных и предварительного обучения без учителя подобно другим автокодировщикам, которые мы обсуждали ранее; их также довольно просто и эффективно использовать для удаления шума из изображений.



*Рис. 17.9. Зашумленные изображения (вверху) и их реконструкции (внизу)*

## Разреженные автокодировщики

Еще одним 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 вынудит 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 сделать все, чтобы их услышали).

Простой подход предусматривает применение в кодирующем слое сигмоидальной функции активации (чтобы ограничить кодировки значениями между 0 и 1), использование крупного кодирующего слоя (скажем, с 300 элементами) и добавление к активациям кодирующего слоя некоторой доли регуляризации  $\ell_1$  (декодировщик остается самым обычным):

```
sparse_11_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_11_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_11_ae =
    keras.models.Sequential([sparse_11_encoder, sparse_11_decoder])
```

Слой `ActivityRegularization` лишь возвращает свои входы, но в качестве побочного эффекта он добавляет потерю при обучении, равную сумме абсолютных значений своих входов (этот слой оказывает влияние только во время обучения). Эквивалентно вы могли бы удалить слой `ActivityRegularization` и установить `activity_regularizer=keras.regularizers.l1(1e-3)` в предыдущем слое. Такой штраф будет стимулировать нейронную сеть выпускать кодировки, близкие к 0, но поскольку она также штрафуется в случае, если не реконструирует входы кор-

ректно, то будет вынуждена выдавать, по крайней мере, несколько ненулевых значений. Применение нормы  $\ell_1$  вместо  $\ell_2$  подтолкнет нейронную сеть к предохранению наиболее важных кодировок и одновременно устраниению тех, которые не нужны для входного изображения (взамен простого сокращения всех кодировок).

Другой подход, часто обеспечивающий лучшие результаты, предусматривает измерение фактической разреженности кодирующего слоя на каждой итерации обучения и штрафование модели, когда измеренная разреженность отличается от целевой разреженности. Мы делаем это путем вычисления средней активации каждого нейрона в кодирующем слое по целому обучавшему пакету. Размер пакета не должен быть слишком маленьким, иначе среднее не будет точным.

Располагая средней активацией на нейрон, мы хотим штрафовать нейроны, которые чересчур активны или недостаточно активны, 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 использование расстояния Кульбака–Лейблера (кратко упомянутого в главе 4), которое имеет намного более сильные градиенты, чем среднеквадратическая ошибка (рис. 17.10).

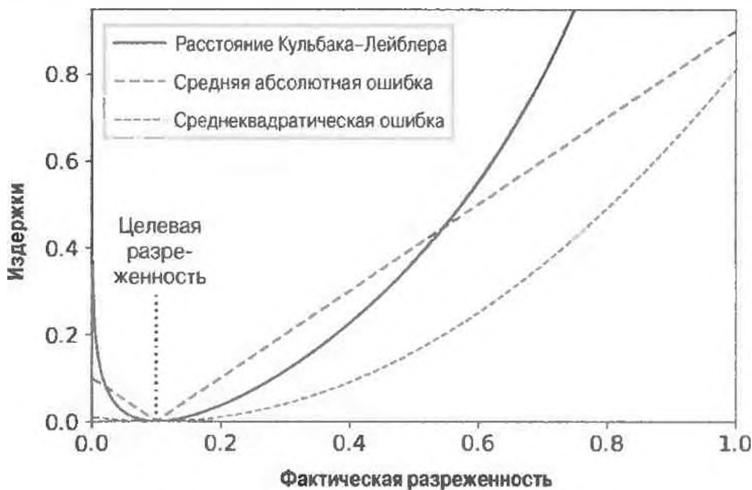


Рис. 17.10. Потеря из-за разреженности

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 17.1.

### Уравнение 17.1. 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 17.2.

### Уравнение 17.2. Расстояние Кульбака–Лейблера между целевой разреженностью $p$ и фактической разреженностью $q$

$$D_{KL}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

Вычислив потерю из-за разреженности для каждого нейрона в кодирующем слое, мы 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.

Теперь у нас есть все необходимое для реализации разреженного автокодировщика, основанного на расстоянии Кульбака–Лейблера. Первым делом создадим специальный регуляризатор для применения регуляризации на базе расстояния Кульбака–Лейблера:

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence
class (keras.regularizers.Regularizer):
    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target
```

```

def __call__(self, inputs):
    mean_activities = K.mean(inputs, axis=0)
    return self.weight * (
        kl_divergence(self.target, mean_activities) +
        kl_divergence(1. - self.target, 1. - mean_activities))

```

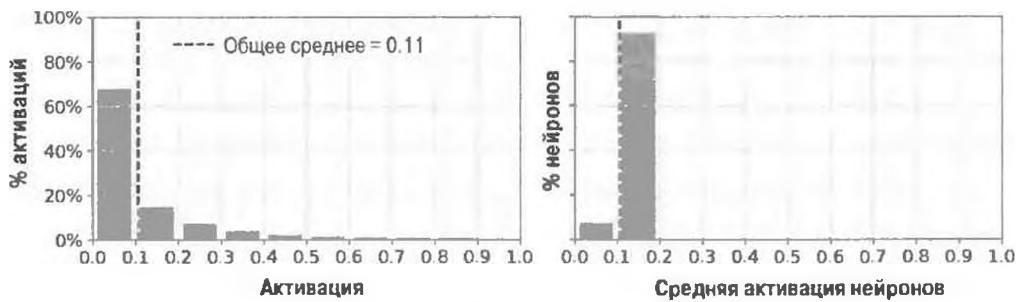
Далее мы можем построить разреженный автокодировщик, используя KL Divergence Regularizer для активаций кодирующего слоя:

```

kld_reg = KL DivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid",
                      activity_regularizer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae =
    keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])

```

После обучения разреженного автокодировщика на наборе данных Fashion MNIST активации нейронов в кодирующем слое в основном близки к 0 (около 70% всех активаций меньше 0.1), а все нейроны имеют среднюю активацию вблизи 0.1 (около 90% всех нейронов имеют среднюю активацию между 0.1 и 0.2), как показано на рис. 17.11.



*Рис. 17.11. Распределение активаций в кодирующем слое (слева) и распределение средней активации на нейрон (справа)*

# Вариационные автокодировщики

Еще одна важная категория автокодировщиков была введена в 2013 году (<https://hml.info/115>) Дидериком Кингма и Максом Веллингом, быстро став одним из самых популярных типов автокодировщиков: *вариационные автокодировщики (variational autoencoder)*<sup>7</sup>.

Вариационные автокодировщики совершенно отличаются от всех автокодировщиков, которые обсуждались до сих пор, перечисленными ниже аспектами.

- Они являются *вероятностными автокодировщиками*, т.е. их выходы отчасти определяются случайно, даже после обучения (в противоположность шумоподавляющим автокодировщикам, которые действуют случайность только во время обучения).
- Более важно то, что они являются *порождающими автокодировщиками*, т.е. способны генерировать новые образцы, которые выглядят так, будто выбраны из обучающего набора.

Обе характеристики делают вариационные автокодировщики похожими на ограниченные машины Больцмана, 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 выбирать новый образец). Действительно, как вытекает из их названия, вариационные автокодировщики выполняют вариационный байесовский вывод (упомянутый в главе 9), который представляет собой эффективный способ проведения приближенного байесовского вывода.

Давайте посмотрим, как они работают. На рис. 17.12 (слева) показан вариационный автокодировщик. Вы можете опознать 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  $\mu$  и стандартное отклонение  $\sigma$* . Фактическая кодировка затем выбирается случайным образом из гауссова распределения со средним  $\mu$  и стандартным отклонением  $\sigma$ . Далее декодировщик декодирует выбранную кодировку обычным образом.

<sup>7</sup> Дидерик Кингма и Макс Веллинг, *Auto-Encoding Variational Bayes* (Автокодирование на основе вариационного байесовского подхода), препринт arXiv:1312.6114 (2013 г.).

В правой части рис. 17.12 приведен образец, проходящий через данный автокодировщик. Сначала кодировщик вырабатывает  $\mu$  и  $\sigma$ , затем случайным образом выбирается кодировка (обратите внимание, что она расположена не точно в позиции  $\mu$ ) и в заключение эта кодировка декодируется; финальный выход напоминает обучающий образец.

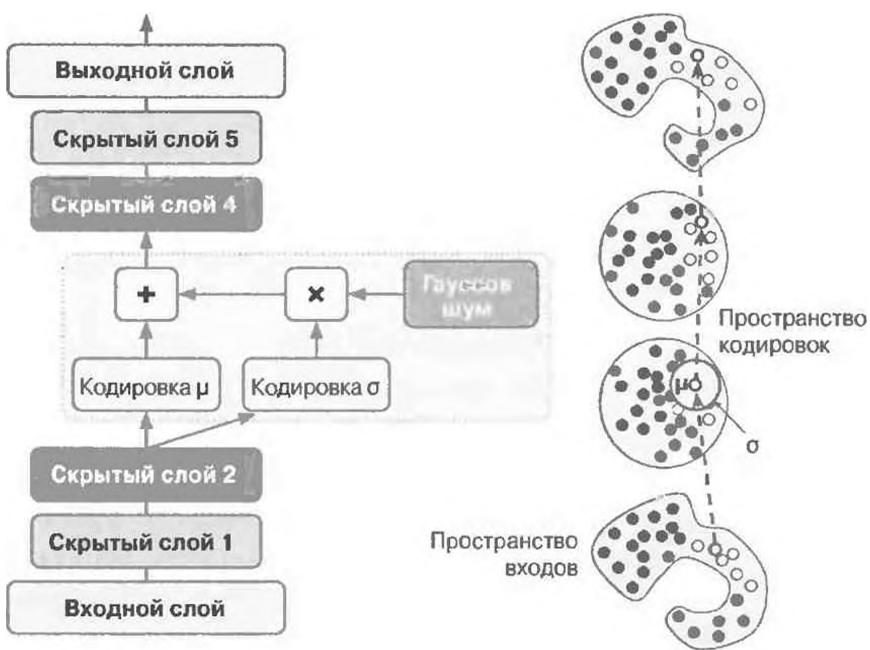


Рис. 17.12. Вариационный автокодировщик (слева) и проходящий через него образец (справа)

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 распределения<sup>8</sup>: во время обучения функция издержек (обсуждается ниже) вынуждает кодировки постепенно мигрировать внутри пространства кодировок (также называемого латентным пространством (*latent space*)), чтобы в итоге выглядеть подобно облаку гауссовых точек. 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!

<sup>8</sup> Вариационные автокодировщики в действительности более универсальны; кодировки не ограничиваются гауссовыми распределениями.

А теперь 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: она является расстоянием Кульбака–Лейблера между целевым (т.е. гауссовым) распределением и фактическим распределением кодировок. Математические выкладки чуть сложнее, чем для разреженного автокодировщика, 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, так что латентная потеря может быть рассчитана с помощью уравнения 17.3<sup>9</sup>.

### Уравнение 17.3. Латентная потеря вариационного автокодировщика

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2$$

В этом уравнении  $\mathcal{L}$  — латентная потеря,  $n$  — размерность кодировок, а  $\mu_i$  и  $\sigma_i$  — средняя величина и стандартное отклонение  $i$ -того компонента кодировок. Векторы  $\mu$  и  $\sigma$  (содержащие все  $\mu_i$  и  $\sigma_i$ ) являются выходом кодировщика, как видно на рис. 17.12 (слева).

Общепринятая подстройка архитектуры вариационного автокодировщика заключается в том, чтобы заставить кодировщик выдавать  $\gamma = \log(\sigma^2)$ , а не  $\sigma$ . Затем латентную потерю можно рассчитывать по уравнению 17.4. Такой подход более численно устойчив и ускоряет обучение.

### Уравнение 17.3. Латентная потеря вариационного автокодировщика, переписанная с использованием $\gamma = \log(\sigma^2)$

$$\mathcal{L} = -\frac{1}{2} \sum_{i=1}^K 1 + \gamma_i - \exp(\gamma_i) - \mu_i^2$$

---

<sup>9</sup> За математическими выкладками обращайтесь к исходной статье о вариационных автокодировщиках или к замечательному руководству Карла Доерша (<https://homeinfo.info/116>) (2016 г.).

Давайте приступим к построению вариационного автокодировщика для набора данных Fashion MNIST (как показано на рис. 17.12, но с применением подстройки  $\gamma$ ). Прежде всего, нам понадобится специальный слой для выборки кодировок при заданных  $\mu$  и  $\gamma$ :

```
class Sampling(keras.layers.Layer):
    def call(self, inputs):
        mean, log_var = inputs
        return K.random_normal(tf.shape(log_var)) *
            K.exp(log_var / 2) + mean
```

Слой Sampling принимает два входа: mean ( $\mu$ ) и log\_var ( $\gamma$ ). Он использует функцию K.random\_normal() для выборки случайного вектора (той же формы, что и  $\gamma$ ) из нормального распределения со средним 0 и стандартным отклонением 1. Далее он умножается на  $\exp(\gamma/2)$  (равно  $\sigma$ , как вы можете проверить), добавляет  $\mu$  и возвращает результат. Это производит выборку вектора кодировок из нормального распределения со средним  $\mu$  и стандартным отклонением  $\sigma$ . Затем мы можем создать кодировщик с применением API-интерфейса Functional, потому что модель не полностью последовательная:

```
codings_size = 10

inputs = keras.layers.Input(shape=[28, 28])
z = keras.layers.Flatten()(inputs)
z = keras.layers.Dense(150, activation="selu")(z)
z = keras.layers.Dense(100, activation="selu")(z)
codings_mean = keras.layers.Dense(codings_size)(z)      # μ
codings_log_var = keras.layers.Dense(codings_size)(z)    # γ
codings = Sampling()([codings_mean, codings_log_var])
variational_encoder = keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Обратите внимание, что слои Dense, которые выдают codings\_mean ( $\mu$ ) и codings\_log\_var ( $\gamma$ ), имеют те же самые входы (т.е. выходы второго слоя Dense). Мы передаем codings\_mean и codings\_log\_var слою Sampling. Наконец, модель variational\_encoder имеет три выхода на случай, если вы захотите проинспектировать значения codings\_mean и codings\_log\_var. Мы будем использовать единственный выход — последний (codings). А теперь давайте построим декодировщик:

```
decoder_inputs = keras.layers.Input(shape=[codings_size])
x = keras.layers.Dense(100, activation="selu")(decoder_inputs)
x = keras.layers.Dense(150, activation="selu")(x)
```

```
x = keras.layers.Dense(28 * 28, activation="sigmoid")(x)
outputs = keras.layers.Reshape([28, 28])(x)
variational_decoder = keras.Model(inputs=[decoder_inputs],
                                 outputs=[outputs])
```

Для этого декодировщика мы могли бы применять API-интерфейс Sequential вместо Functional, т.к. на самом деле он представляет собой простую стопку слоев, практически идентичную многим декодировщикам, которые создавались до сих пор. В заключение построим модель вариационного автокодировщика:

```
_, _, codings = variational_encoder(inputs)
reconstructions = variational_decoder(codings)
variational_ae = keras.Model(inputs=[inputs],
                             outputs=[reconstructions])
```

Обратите внимание, что мы игнорируем первые два выхода кодировщика (нам нужно передавать декодировщику только кодировки). Наконец, мы должны добавить латентную потерю и потерю из-за реконструкции:

```
latent_loss = -0.5 * K.sum(
    1 + codings_log_var - K.exp(codings_log_var)
    - K.square(codings_mean),
    axis=-1)
variational_ae.add_loss(K.mean(latent_loss) / 784.)
variational_ae.compile(loss="binary_crossentropy",
                      optimizer="rmsprop")
```

Сначала мы применяем уравнение 17.4, чтобы вычислить латентную потерю для каждого образца в пакете (суммируя по последней оси). Затем мы подсчитываем среднюю потерю по всем образцам в пакете и делим результат на 784, обеспечивая соответствующий масштаб в сравнении с потерей из-за реконструкции. В действительности подразумевается, что потеря из-за реконструкции вариационного автокодировщика должна быть суммой ошибок реконструкции пикселей, но когда библиотека Keras вычисляет потерю "binary\_crossentropy", она рассчитывает среднее по всем 784 пикселям, а не сумму. Таким образом, потеря из-за реконструкции оказывается в 784 раза меньше, чем нам необходимо. Мы могли бы определить специальную потерю, чтобы вычислять сумму, а не среднее, но проще разделить латентную потерю на 784 (финальная потеря будет в 784 раза меньше, чем должна быть, но это лишь означает, что бы обязаны использовать более высокую скорость обучения).

Обратите внимание, что мы применяем оптимизатор RMSprop, который в данном случае работает хорошо. И наконец-то мы можем обучить автокодировщик!

```
history = variational_ae.fit(X_train, X_train, epochs=50,
                             batch_size=128,
                             validation_data=[X_valid, X_valid])
```

## Генерирование изображений Fashion MNIST

Давайте воспользуемся созданным вариационным автокодировщиком, чтобы сгенерировать изображения, которые выглядят как предметы моды. Нам понадобится только выбрать случайные кодировки из гауссова распределения и декодировать их:

```
codings = tf.random.normal(shape=[12, codings_size])
images = variational_decoder(codings).numpy()
```

На рис. 17.13 показаны 12 сгенерированных изображений.



Рис. 17.13. Изображения Fashion MNIST, сгенерированные вариационным автокодировщиком

Большинство изображений выглядят достаточно убедительными, хотя и слишком размытыми. Остальные не особенно хороши, но не будьте слишком строги в отношении автокодировщика — у него было всего несколько минут на обучение! Дайте ему чуть больше времени на точную настройку и обучение, и генерируемые изображения должны выглядеть лучше.

Вариационные автокодировщики позволяют выполнять *семантическую интерполяцию* (*semantic interpolation*): вместо интерполирования двух изображений на уровне пикселей (как будто бы два изображения были наложены друг на друга) мы можем интерполировать на уровне кодировок. Сначала мы прогоняем оба изображения через кодировщик, затем интерполируем две полученные кодировки и в заключение декодируем интерполированные кодировки, чтобы получить финальное изображение. Оно будет выглядеть как обычное изображение Fashion MNIST, но является промежуточным между исходными изображениями. В приведенном ниже примере кода мы берем 12 только что сгенерированных кодировок, упорядочиваем их в сетке  $3 \times 4$  и используем функцию `tf.image.resize()` из TensorFlow для изменения размеров сетки до  $5 \times 7$ . По умолчанию функция `resize()` будет выполнять билинейную интерполяцию (*bilinear interpolation*), так что каждая вторая строка и столбец будет содержать интерполированную кодировку. Затем мы применяем декодировщик для выпуска всех изображений:

```
codings_grid = tf.reshape(codings, [1, 3, 4, codings_size])
larger_grid = tf.image.resize(codings_grid, size=[5, 7])
interpolated_codings = tf.reshape(larger_grid, [-1, codings_size])
images = variational_decoder(interpolated_codings).numpy()
```

На рис. 17.14 показаны результатирующие изображения. Исходные изображения помещены в рамки, а остальные представляют собой результат семантической интерполяции между соседними изображениями. Например, обратите внимание, что полуботинок в четвертой строке и пятом столбце является удачно сделанной интерполяцией между полуботинками, расположенными выше и ниже него.

В течение нескольких лет вариационные автокодировщики были довольно популярными, но сети GAN со временем выбились в лидеры, в частности оттого, что они способны генерировать гораздо более реалистичные и четкие изображения. Итак, давайте переключим наше внимание на сети GAN.

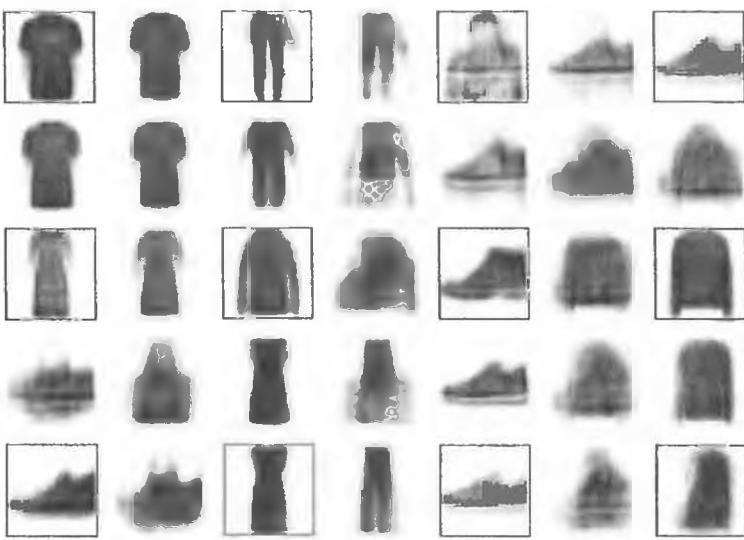


Рис. 17.14. Семантическая интерполяция

## Порождающие состязательные сети

Порождающие состязательные сети были предложены в статье 2014 года (<https://homl.info/gan>)<sup>10</sup> Яном Гудфелло и др., и хотя сама идея взволновала исследователей почти мгновенно, потребовалось несколько лет, чтобы преодолеть трудности, связанные с обучением сетей GAN. Подобно многим замечательным идеям задним числом решение выглядит простым: заставить нейронные сети соперничать друг с другом в надежде, что такое состязание подтолкнет их к успеху. Как показано на рис. 17.15, сеть GAN состоит из двух нейронных сетей, которые описаны ниже.

### Генератор

Принимает случайное распределение в качестве входа (обычно гауссово) и выдает некоторые данные — как правило, изображение. Вы можете считать случайные входы латентными представлениями (т.е. кодировками) изображения, которое должно быть сгенерировано. Таким образом, вы видите, что генератор предлагает такую же функциональность, как декодировщик в вариационном автокодировщике, и может

<sup>10</sup> Ян Гудфелло и др., *Generative Adversarial Nets* (Порождающие состязательные сети), *Proceedings of the 27th International Conference on Neural Information Processing Systems 2* (2014 г.): с. 2672–2680.

использоваться тем же способом для генерации новых изображений (нужно всего лишь передать ему какой-то гауссов шум, и он выдаст совершенно новое изображение). Однако генератор обучается совсем по-другому, что мы вскоре увидим.

## Дискриминатор

Принимает либо фальшивое изображение от генератора, либо подлинное изображение из обучающего набора в качестве входа и обязан отгадать, является изображение фальшивым или подлинным.



Рис. 17.15. Порождающая состязательная сеть

Во время обучения генератор и дискриминатор преследуют противоположные цели: дискриминатор пытается отличить фальшивые изображения от подлинных, тогда как генератор старается производить изображения, которые выглядят в достаточной мере подлинными, чтобы обмануть дискриминатор. Поскольку сеть GAN состоит из двух сетей с разными целями, ее не удастся обучать как обычновенную нейронную сеть. Каждая итерация обучения разделена на две стадии.

- На первой стадии мы обучаем дискриминатор. Из обучающего набора выбирается пакет подлинных изображений и укомплектовывается равным количеством фальшивых изображений, произведенных генератором. Метки устанавливаются в 0 для фальшивых изображений и в 1

для подлинных изображений. Дискриминатор обучается на этом помеченному пакете за один шаг с применением потери двоичной перекрестной энтропии. Важно отметить, что в течение первой стадии обратное распространение оптимизирует веса только дискриминатора.

- На второй стадии мы обучаем генератор. Сначала мы используем его для выпуска еще одного пакета фальшивых изображений и снова применяем дискриминатор для выяснения, являются ли изображения фальшивыми или подлинными. На этот раз мы не добавляем в пакет подлинные изображения, а все метки устанавливаются в 1 (подлинное): другими словами, мы хотим, чтобы генератор производил изображения, которые дискриминатор будет (ошибочно) считать подлинными! Важно отметить, что на время данного шага веса дискриминатора замораживаются, поэтому обратное распространение влияет только на веса генератора.



Генератор никогда фактически не видит подлинные изображения, но постепенно учится выпускать убедительные фальшивые изображения! Все, что он получает — это градиенты, текущие обратно через дискриминатор. К счастью, чем лучше становится дискриминатор, тем больше информации о подлинных изображениях содержится в таких вторичных градиентах, так что генератор может добиться значительного прогресса.

Давайте построим простую сеть GAN для набора данных Fashion MNIST.

Первым делом нам необходимо построить генератор и дискриминатор. Генератор похож на кодировщик автокодировщика, а дискриминатор представляет собой обычный двоичный классификатор (на входе он принимает изображение и заканчивается слоем Dense, содержащим единственный элемент и использующим сигмоидальную функцию активации). Для второй стадии каждой итерации обучения нам также нужна полная модель GAN, содержащая генератор, за которым следует дискриминатор:

```
codings_size = 30
generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu",
                       input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
```

```
discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Далее понадобится скомпилировать модель. Так как дискриминатор является двоичным классификатором, мы свободно можем применять потерю двоичной перекрестной энтропии. Генератор будет обучаться только через модель `gan`, поэтому компилировать его вообще не понадобится. Модель `gan` также представляет собой двоичный классификатор и потому он может использовать потерю двоичной перекрестной энтропии. Важно отметить, что дискриминатор не должен обучаться в течение второй стадии, так что перед компиляцией модели `gan` мы делаем его необучаемым:

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```



Атрибут `trainable` учитывается Keras только при компиляции модели, поэтому после выполнения приведенного выше кода `discriminator` оказывается обучаемым, если мы вызовем его метод `fit()` или `train_on_batch()` (который мы и применим), но он *не* будет обучаемым в случае вызова указанных методов на модели `gan`.

Поскольку цикл обучения необычен, мы не можем использовать стандартный метод `fit()`. Взамен мы реализуем специальный цикл обучения, для чего сначала создадим объект `Dataset` для прохода по изображениям:

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Теперь все готово к написанию цикла обучения. Давайте поместим его внутрь функции `train_gan()`:

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
```

```

# стадия 1 - обучение дискриминатора
noise = tf.random.normal(shape=[batch_size, codings_size])
generated_images = generator(noise)
X_fake_and_real = tf.concat([generated_images, X_batch],
                             axis=0)
y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
discriminator.trainable = True
discriminator.train_on_batch(X_fake_and_real, y1)
# стадия 2 - обучение генератора
noise = tf.random.normal(shape=[batch_size, codings_size])
y2 = tf.constant([[1.]] * batch_size)
discriminator.trainable = False
gan.train_on_batch(noise, y2)
train_gan(gan, dataset, batch_size, codings_size)

```

Как обсуждалось ранее, на каждой итерации можно заметить две стадии.

- На стадии 1 мы подаем генератору гауссов шум для выпуска фальшивых изображений и дополняем результирующий пакет равным количеством подлинных изображений. Цели  $y_1$  устанавливаются в 0 для фальшивых и в 1 для подлинных изображений. Затем мы обучаем дискриминатор на дополненном пакете. Обратите внимание, что мы устанавливаем атрибут `trainable` дискриминатора в `True`: это нужно лишь для того, чтобы избавиться от предупреждения, которое Keras отображает, как только замечает, что атрибут `trainable` теперь равен `False`, но был `True`, когда модель компилировалась (или наоборот).
- На стадии 2 мы подаем сети GAN некоторый гауссов шум. Ее генератор начнет производить фальшивые изображения, после чего дискриминатор попытается отгадать, фальшивые эти изображения или подлинные. Мы хотим, чтобы дискриминатор поверил в то, что фальшивые изображения являются подлинными, а потому цели  $y_2$  установлены в 1. Обратите внимание, что снова во избежание вывода предупреждения мы устанавливаем атрибут `trainable` в `False`.

Вот и все! Если вы отобразите сгенерированные изображения (рис. 17.16), то увидите, что в конце первой эпохи они уже начинают напоминать (очень зашумленные) изображения Fashion MNIST.

К сожалению, на самом деле изображения никогда не становятся намного лучше показанных, и вы даже можете найти эпохи, где сеть GAN, кажется, забывает то, чему научилась. Как так? Оказывается, что обучение сети GAN может быть сложной задачей. Давайте выясним причины.

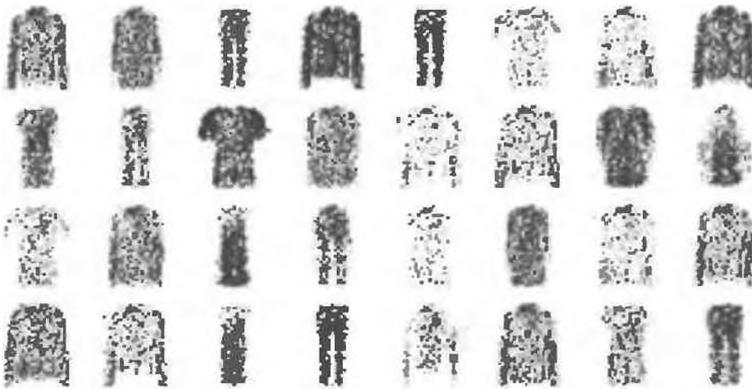


Рис. 17.16. Изображения, сгенерированные сетью GAN после одной эпохи обучения

### Трудности обучения порождающих состязательных сетей

Во время обучения генератор и дискриминатор постоянно пытаются перехитрить друг друга в игре с нулевой суммой. По мере продвижения обучения игра может оказаться в состоянии, которое теоретики игр называют *равновесием Нэша (Nash equilibrium)* в честь математика Джона Нэша: ситуация, когда ни один игрок не может увеличить выигрыш за счет изменения собственной стратегии, исходя из предположения, что остальные игроки свои стратегии не меняют. Например, равновесие Нэша достигается, когда все едут по левой полосе дороги: ни одному водителю не станет лучше, если он будет единственным, кто свернет на другую полосу. Конечно, существует второе возможное равновесие Нэша: когда все едут по правой полосе дороги. Разные начальные состояния и динамика могут приводить к одному или к другому равновесию. В рассматриваемом примере есть единственная стратегия после того, как достигнуто равновесие (т.е. езда по одной полосе вместе со всеми остальными), но равновесие Нэша может включать множество конкурирующих стратегий (скажем, хищник преследует свою жертву, жертва пытается убежать и никому из них не станет лучше от изменения своей стратегии).

Так каким образом это применимо к сетям GAN? Авторы статьи продемонстрировали, что сеть GAN может достичь только единственного равновесия Нэша: когда генератор выпускает совершенно реалистичные изображения, а дискриминатор вынужден угадывать (50% подлинное, 50% фальшивое). Данный факт весьма обнадеживает: может показаться, что вам нужно лишь обучать сеть GAN достаточно долго и в итоге она достигнет равновесия Нэша, давая безупречный генератор. К сожалению, все не так просто: нет никакой гарантии, что равновесие вообще будет достигнуто.

Наибольшая трудность называется *коллапсом мод* (*mode collapse*): когда выходы генератора постепенно становятся менее разнообразными. Как такое может случиться? Предположим, что генератор стал лучше выпускать убедительные изображения полуботинок, нежели изображения любого другого класса. В результате дискриминатор будет чуть больше обманываться в отношении изображений полуботинок, но это побудит генератор производить еще больше изображений полуботинок. Постепенно он станет забывать, как выпускать что-либо другое. Между тем единственными фальшивыми изображениями, которые будет видеть дискриминатор, окажутся изображения полуботинок, так что он тоже станет забывать, как различать фальшивые изображения остальных классов. В конце концов, когда дискриминатору удастся отличать фальшивые изображения полуботинок от подлинных, генератор будет вынужден перейти к другому классу. Затем генератор может стать хорошим в выпуске изображений рубашек, забыв об изображениях полуботинок, и дискриминатор последует за ним. Сеть GAN может понемногу циклически проходить по нескольким классам, никогда толком не становясь очень хорошей с любым из них.

Более того, так как генератор и дискриминатор постоянно сталкиваются друг с другом, в конечном итоге их параметры могут колебаться и становиться нестабильными. Обучение может надлежаще начаться, а позже без видимых причин неожиданно разойтись из-за таких нестабильностей. И поскольку на сложную динамику подобного рода влияют многие факторы, сети GAN крайне чувствительны к гиперпараметрам: их точная настройка может потребовать больших усилий.

Начиная с 2014 года, исследователи были плотно заняты решением упомянутых проблем: по этой теме было опубликовано много статей, в которых предлагались новые функции издержек<sup>11</sup> (хотя в статье 2018 года (<https://homl.info/gansequal>)<sup>12</sup> исследователи из Google ставят под сомнение их эффективность) или методики, направленные на стабилизацию обучения либо избегание проблемы коллапса мод. Например, популярная методика под названием *воспроизведение опыта* (*experience replay*) предусматрива-

<sup>11</sup> Хорошее сравнение основных потерь GAN можно найти в великолепном проекте Хвалсука Ли на GitHub (<https://homl.info/ganloss>).

<sup>12</sup> Марио Лучич и др., *Are GANs Created Equal? A Large-Scale Study* (Созданы ли порождающие состязательные сети равными? Крупномасштабное исследование), *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (2018 г.): с. 698–707.

ет хранение изображений, выпускаемых генератором на каждой итерации, в буфере воспроизведения (который постепенно отбрасывает более старые сгенерированные изображения) и обучение дискриминатора с использованием подлинных изображений плюс фальшивых изображений, взятых из упомянутого буфера (а не фальшивых изображений, производимых текущим генератором). Тем самым уменьшается вероятность того, что дискриминатор будет переобучаться самыми последними выходами генератора. Еще одна распространенная методика называется *мини-пакетным различением* (*mini-batch discrimination*): она измеряет, насколько похожи изображения в пакете, и предоставляет такие статистические данные дискриминатору, а он может легко отклонить целый пакет фальшивых изображений, которым недостает разнородности. Такой подход побуждает генератор выпускать изображения с большим разнообразием, снижая риск коллапса мод. В других статьях просто предлагались специфические архитектуры, которые по стечению обстоятельств работали хорошо.

Короче говоря, это по-прежнему очень активная область исследований и динамические свойства сетей GAN все еще не совсем понятны. Но хорошая новость в том, что достигнут большой прогресс и некоторые результаты по-настоящему удивительны! Итак, давайте взглянем на ряд самых успешных архитектур, начиная с глубоких сверточных сетей GAN, которые пребывали на современном уровне всего несколько лет назад. Затем мы рассмотрим две более поздние (и более сложные) архитектуры.

## Глубокие сверточные порождающие состязательные сети

В первоначальной статье о сетях GAN, опубликованной в 2014 году, проводились эксперименты со сверточными слоями, но только в попытке сгенерировать небольшие изображения. Вскоре после нее многие исследователи пробовали строить сети GAN на основе более глубоких сверточных сетей для изображений покрупнее. Задача оказалась непростой, т.к. обучение было очень нестабильным, но в конце 2015 года Алеку Рэдфорду и др. все же удалось добиться успеха (<https://homl.info/dcgan>)<sup>13</sup> после экспериментирования со многими разными архитектурами и гиперпараметрами.

<sup>13</sup> Алек Рэдфорд и др., *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* (Обучение представлению без учителя с помощью глубоких сверточных порождающих состязательных сетей), препринт arXiv:1511.06434 (2015 г.).

Авторы назвали свою архитектуру глубокими сверточными порождающими состязательными сетями (*deep convolutional GAN* — DCGAN). Ниже приведены основные рекомендации, которые они предложили для построения стабильных сверточных сетей GAN.

- Замените любые объединяющие слои свертками со страйдами (в дискриминаторе) и транспонированными свертками (в генераторе).
- Используйте пакетную нормализацию в генераторе и дискриминаторе кроме выходного слоя генератора и входного слоя дискриминатора.
- Удалите полносвязные скрытые слои для более глубоких архитектур.
- Применяйте функцию активации ReLU в генераторе для всех слоев кроме выходного, на котором должна использоваться функция активации  $\tanh$ .
- Применяйте функцию активации ReLU с утечкой для всех слоев в дискриминаторе.

Описанные рекомендации будут работать во многих случаях, но не всегда, так что вам все еще может потребоваться экспериментирование с различными гиперпараметрами (на самом деле временами достаточно просто изменить начальное случайное число и обучить ту же самую модель заново). Скажем, вот небольшая сеть DCGAN, которая довольно-таки неплохо работает с набором данных Fashion MNIST:

```
codings_size = 100
generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2,
                               padding="same", activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2,
                               padding="same", activation="tanh")
])
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                      activation=keras.layers.LeakyReLU(0.2),
                      input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
```

```
keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                    activation=keras.layers.LeakyReLU(0.2)),
keras.layers.Dropout(0.4),
keras.layers.Flatten(),
keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Генератор принимает кодировки размером 100, проецирует их на 6272 измерения ( $7 \times 7 \times 128$ ) и изменяет форму результата, чтобы получить тензор  $7 \times 7 \times 128$ . Этот тензор подвергается пакетной нормализации и подается транспонированному сверточному слою со страйдом 2, который повышает его дискретизацию от  $7 \times 7$  до  $14 \times 14$  и уменьшает глубину от 128 до 64. Результат снова подвергается пакетной нормализации и подается еще одному транспонированному сверточному слою со страйдом 2, который повышает его дискретизацию от  $14 \times 14$  до  $28 \times 28$  и уменьшает глубину от 64 до 1. Данный слой использует функцию активации  $\tanh$ , так что выходы будут иметь диапазон от -1 до 1. По указанной причине перед обучением сети GAN понадобится масштабировать обучающий набор до того же самого диапазона. Также необходимо изменить его форму, чтобы добавить измерение каналов:

```
X_train = X_train.reshape(-1, 28, 28, 1) * 2. - 1.
# изменить форму и масштабировать
```

Дискриминатор очень похож на обыкновенную сеть CNN для двоичной классификации за исключением того, что вместо слоев объединения по максимуму, понижающих дискретизацию изображения, мы применяем сверточные слои со страйдами (`strides=2`). Также обратите внимание, что мы используем функцию активации ReLU с утечкой.

В целом рекомендации DCGAN соблюdenы, не считая того, что в дискриминаторе мы заменили слои `BatchNormalization` слоями `Dropout` (в противном случае обучение было бы нестабильным) и в генераторе поменяли функцию активации ReLU на функцию SELU. Не стесняйтесь подстраивать представленную архитектуру: вы увидите, насколько она чувствительна к гиперпараметрам (особенно к относительным скоростям обучения двух сетей).

Наконец, чтобы построить набор данных, после чего скомпилировать и обучить модель, мы применяем точно такой же код, как ранее. После 50 эпох обучения генератор выпускает изображения, подобные показанным на рис. 17.17. Они по-прежнему неидеальны, но многие из них выглядят весьма убедительно.



Рис. 17.17. Изображения, сгенерированные сетью DCGAN после 50 эпох обучения

Если вы масштабируете такую архитектуру и обучите ее на крупном наборе данных с лицами, то можете получить вполне реалистичные изображения. На самом деле сети DCGAN способны узнавать довольно значимые латентные представления, как видно на рис. 17.18: было сгенерировано много изображений, из которых вручную отобрано девять (выше слева) — три изображения с мужчинами в очках, три с мужчинами без очков и три с женщинами без очков. Для каждой из трех категорий кодировки, использованные для генерации изображений, были усреднены, а изображения генерировались на основе результирующих средних кодировок (ниже слева). Таким образом, каждое из трех изображений внизу слева представляет среднее трех изображений, расположенных над ним. Но это не просто среднее, вычисленное на уровне пикселей (результатом оказались бы три наложенных друг на друга лица), а среднее, рассчитанное в латентном пространстве, так что изображения по-прежнему выглядят как нормальные лица. Удивительно, но если вы вычислите операцию “мужчина в очках минус мужчина без очков плюс женщина без очков” (где каждый член — одна из средних кодировок) и сгенерируете изображение, соответствующее результирующей кодировке, то получите изображение в центре сетки  $3 \times 3$  лиц справа: с женщиной в очках! Остальные восемь изображений вокруг центрального были сгенерированы на основе того же самого вектора с добавлением порции шума, чтобы проиллюстрировать возможности семантической интерполяции сетей DCGAN. Способность выполнять арифметические действия с лицами похожа на научную фантастику!



**Рис. 17.18. Векторная арифметика для визуальных концепций (часть рис. 7 из статьи, посвященной сетям DCGAN)<sup>14</sup>**



Если вы добавите класс каждого изображения в качестве дополнительного входа к генератору и дискриминатору, то они оба узнают, на что похож каждый класс, а у вас появится возможность контролировать класс каждого изображения, производимого генератором. Это называется *условной порождающей состязательной сетью* (*conditional GAN — CGAN*; <https://homl.info/cgan>)<sup>15</sup>.

Тем не менее, сети DCGAN не безупречны. Например, когда вы пытаетесь генерировать очень крупные изображения с применением сетей DCGAN, то часто сталкиваетесь с локально убедительными признаками, но общими несоответствиями (такими как рубашки, у которых один рукав намного длиннее другого). Как это можно привести в порядок?

### Прогрессивный рост порождающих состязательных сетей

В статье 2018 года (<https://homl.info/progan>)<sup>16</sup> исследователи из Nvidia Теро Каррас и др. описали важную методику: они предложили в на-

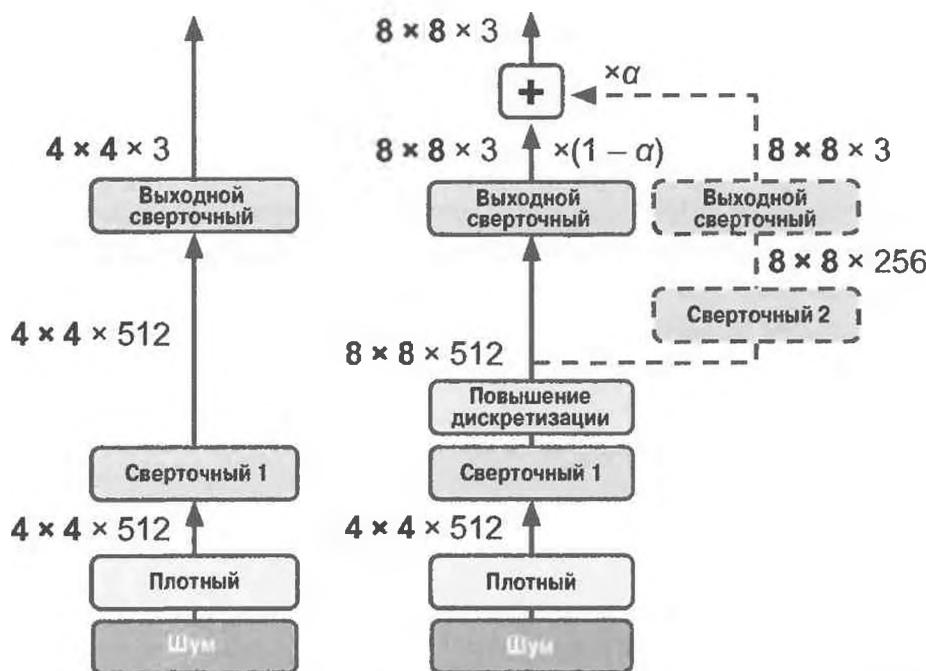
<sup>14</sup> Воспроизведется с разрешения авторов.

<sup>15</sup> Мехди Мирза и Саймон Осиндеро, *Conditional Generative Adversarial Nets* (Условные порождающие состязательные сети), препринт arXiv: 1411.1784 (2014 г.).

<sup>16</sup> Теро Каррас и др., *Progressive Growing of GANs for Improved Quality, Stability, and Variation* (Прогрессивный рост порождающих состязательных сетей для улучшенного качества, стабильности и изменчивости), *Proceedings of the International Conference on Learning Representations* (2018 г.).

чале обучения генерировать небольшие изображения, затем постепенно добавлять к генератору и дискриминатору сверточные слои, чтобы выпускать изображения все больших и больших размеров ( $4 \times 4$ ,  $8 \times 8$ ,  $16 \times 16$ , ...,  $512 \times 512$ ,  $1024 \times 1024$ ). Такой подход напоминает жадное послойное обучение многослойных автокодировщиков. Дополнительные слои добавляются в конец генератора и в начало дискриминатора, а ранее обученные слои остаются обучаемыми.

Скажем, когда выходы генератора растут с  $4 \times 4$  до  $8 \times 8$  (рис. 17.19), к существующему сверточному слою добавляется слой повышения дискретизации (использующий фильтрацию ближайших соседей), так что он выдает карты признаков  $8 \times 8$ , которые затем передаются новому сверточному слою (с дополнением "Same" и страйдом 1, а потому тоже с выходами  $8 \times 8$ ). За этим новым слоем следует новый выходной сверточный слой: он представляет собой обычный сверточный слой с размером ядра 1, который проецирует выходы на желаемое количество каналов (например, 3).



**Рис. 17.19.** Прогрессивный рост сети GAN: генератор GAN выдает цветные изображения  $4 \times 4$  (слева); мы расширяем его для выдачи изображений  $8 \times 8$  (справа)

Чтобы избежать разрушения узнанных весов первого сверточного слоя, когда добавлен новый сверточный слой, финальный выход определен как взвешенная сумма первоначального выходного слоя (который теперь выдает карты признаков  $8 \times 8$ ) и нового выходного слоя. Весом новых выходов будет  $\alpha$ , тогда как весом первоначальных выходов —  $1 - \alpha$ , причем  $\alpha$  медленно увеличивается от 0 до 1. Другими словами, новые сверточные слои (представленные на рис. 17.19 с помощью пунктирных линий) постепенно усиливаются, в то время как первоначальный выходной слой постепенно затухает. Похожая методика усиления/затухания применяется при добавлении нового сверточного слоя к дискриминатору (за которым следует слой объединения по среднему для понижения дискретизации).

В статье Тero Карраса и др. также было представлено несколько других методик, направленных на увеличение разнообразия выходов (во избежание коллапса мод) и повышение стабильности обучения.

### *Слой стандартных отклонений мини-пакета*

Добавляется ближе к концу дискриминатора. Для каждой позиции во входах он вычисляет стандартное отклонение по всем каналам и всем образцам в пакете ( $S = tf.math.reduce_std(inputs, axis=[0, -1])$ ). Рассчитанные стандартные отклонения затем усредняются по всем точкам, чтобы получить единственное значение ( $v = tf.reduce_mean(S)$ ). В заключение к каждому образцу в пакете добавляется дополнительная карта признаков и заполняется вычисленным значением ( $tf.concat([inputs, tf.fill([batch_size, height, width, 1], v)], axis=-1)$ ). Чем это помогает? Что ж, если генератор производит изображения с небольшим разнообразием, тогда будет маленькое стандартное отклонение между картами признаков в дискриминаторе. Благодаря этому слою дискриминатор будет иметь легкий доступ к таким статистическим данным, что уменьшает вероятность его обмана генератором, который производит слишком мало разнообразия. В итоге генератор стимулируется выпускать более разнообразные выходы, сокращая риск коллапса мод.

### *Стабилизированная скорость обучения*

Инициализирует все веса с использованием простого гауссова распределения со средним 0 и стандартным отклонением 1 вместо инициализации Хе. Однако во время выполнения (т.е. каждый раз, когда слой запускается) масштаб весов уменьшается на тот же коэффициент, как в

инициализации Хе: они делятся на  $\sqrt{2/n_{\text{входов}}}$ , где  $n_{\text{входов}}$  — количество входов в слой. В статье было продемонстрировано, что такая методика значительно улучшает эффективность сети GAN, когда применяются RMSProp, Adam или другие адаптивные оптимизаторы градиентов. На самом деле упомянутые оптимизаторы нормализуют обновления градиентов их оценочным стандартным отклонением (см. главу 11), так что параметры, которые имеют больший динамический диапазон<sup>17</sup>, будут дольше обучаться, тогда как параметры с небольшим динамическим диапазоном могут обновляться слишком быстро, приводя к нестабильностям. За счет масштабирования весов как части самой модели, а не просто их масштабирования при инициализации, такой подход гарантирует, что динамический диапазон будет одинаковым для всех параметров на всем протяжении обучения, так что все они обучаются с той же самой скоростью. В результате обучение ускоряется и стабилизируется.

### Слой попиксельной нормализации (*pixelwise normalization layer*)

Добавляется после каждого сверточного слоя в генераторе. Он нормализует каждую активацию, основываясь на всех активациях в том же изображении и в той же позиции, но по всем каналам (разделяя на квадратный корень среднеквадратической активации). В коде TensorFlow это выглядит как `inputs / tf.sqrt(tf.reduce_mean(tf.square(X), axis=-1, keepdims=True) + 1e-8)` (сглаживающий член  $1e-8$  необходим для того, чтобы не было деления на ноль). Такая методика избегает взрывного роста в активациях из-за чрезмерной конкуренции между генератором и дискриминатором.

Сочетание всех описанных методик позволило авторам генерировать чрезвычайно убедительные изображения лиц с высокой четкостью (<https://homl.info/progandemo>). Но что именно мы называем “убедительным”? Оценка является одной из больших трудностей при работе с сетями GAN: хотя можно автоматически оценивать разнообразие генерируемых изображений, оценка их качества — гораздо более сложная и субъективная задача. Один из приемов предусматривает участие оценщиков-людей, но он сопряжен с высокими затратами и расходом времени. Поэтому авторы предложили измерять сходство между локальной структурой генерированных изоб-

<sup>17</sup> Динамический диапазон переменной — это отношение между наибольшим и наименьшим значениями, которые она может принимать.

ражений и обучающих изображений, учитывая каждый масштаб. Такая идея привела их к еще одной революционной инновации: сетям StyleGAN.

## Сети StyleGAN

Современное состояние генерации изображений с высоким разрешением снова получило развитие той же командой исследователей из Nvidia в их статье 2018 года (<https://homl.info/stylegan>)<sup>18</sup>, где была представлена ставшая популярной архитектура StyleGAN. Авторы использовали в генераторе методики *передачи стиля* (*style transfer*) для гарантии того, что генерируемые изображения имеют такую же локальную структуру, как обучающие изображения при каждом масштабе, значительно улучшая качество генерируемых изображений. Дискриминатор и функция потерь не изменялись, только генератор. Давайте взглянем на сеть StyleGAN. Она состоит из двух сетей (рис. 17.20).

### Сеть отображения (*mapping network*)

Восьмислойный персептрон, который отображает латентные представления  $z$  (т.е. кодировки) на вектор  $w$ . Затем вектор  $w$  прогоняется через множество аффинных преобразований (т.е. слоев Dense без функций активации, представленных на рис. 17.20 квадратами с буквой “A”), которые производят множество векторов. Эти векторы управляют стилем генерируемого изображения на различных уровнях, от мелкозернистой текстуры (скажем, цвета волос) до высокоуровневых признаков (например, взрослый или ребенок). Короче говоря, сеть отображения отображает кодировки на множество векторов стилей.

### Сеть синтеза (*synthesis network*)

Отвечает за генерирование изображений. Она имеет константный обученный вход (точнее данный вход будет константным *после обучения*, но *во время обучения* он продолжает подстраиваться обратным распространением). Вход обрабатывается посредством множества сверточных слоев и слоев повышения дискретизации, как было ранее, но здесь присутствуют два трюка. Во-первых, к входу и всем выходам сверточных слоев добавляется некоторый шум (до функции активации).

<sup>18</sup> Теро Каррас и др., *A Style-Based Generator Architecture for Generative Adversarial Networks* (Основанная на стиле архитектура генератора для порождающих состязательных сетей), препринт arXiv:1812.04948 (2018 г.).

Во-вторых, за каждым слоем шума находится слой *адаптивной нормализации образцов* (*Adaptive Instance Normalization* — *AdaIN*): он стандартизирует каждую карту признаков независимым образом (вычитая среднее карты признаков и разделяя на ее стандартное отклонение), затем применяет вектор стилей с целью определения масштаба и смещения каждой карты признаков (вектор стилей содержит один масштаб и один член смещения для каждой карты признаков).

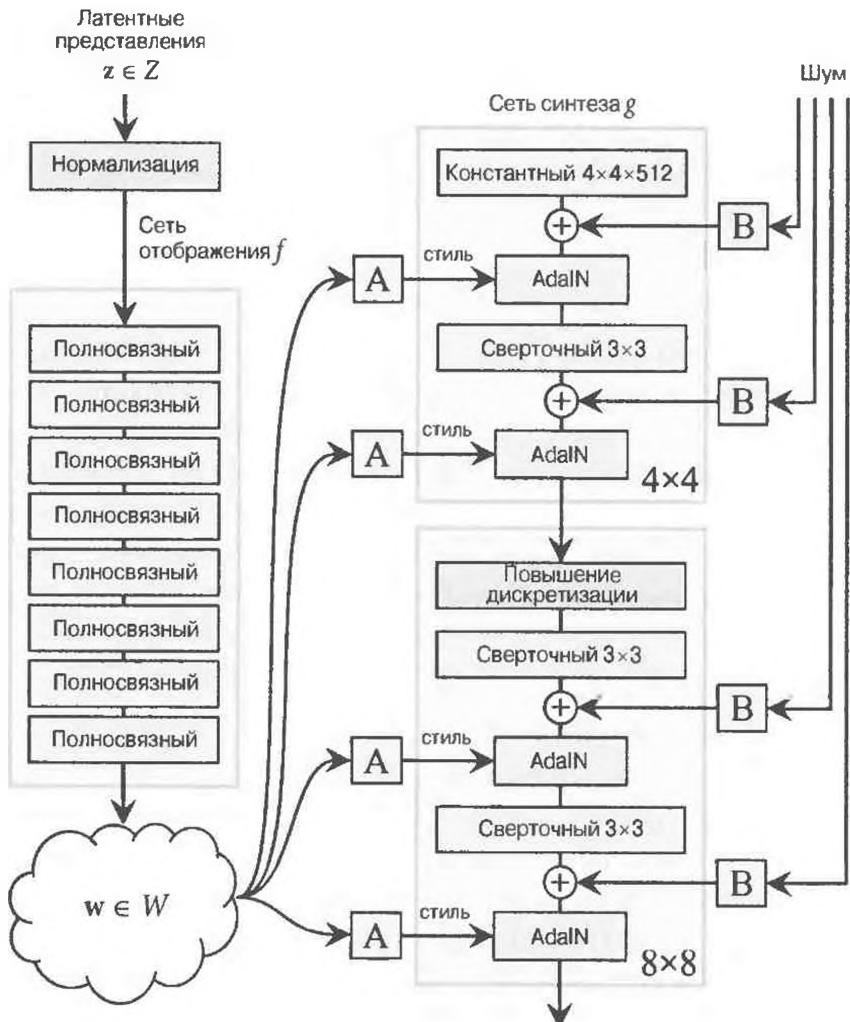


Рис. 17.20. Архитектура генератора сети StyleGAN  
(часть рисунка 1 из статьи о StyleGAN)<sup>19</sup>

<sup>19</sup> Воспроизведется с разрешения авторов.

Идея добавления шума независимо от кодировки очень важна. Некоторые части изображения довольно случайны, скажем, точная позиция каждой веснушки или волоса. В более ранних сетях GAN такая случайность должна либо исходить из кодировок, либо быть каким-то псевдослучайным шумом, производимым самим генератором. Если случайность исходила из кодировок, то это означало, что генератору приходилось выделять значительную часть представительной мощи кодировок для хранения шума — достаточно расточительное решение. Более того, шум должен был иметь возможность протекать через сеть и достигать финальных слоев генератора, что выглядело как ненужное ограничение, которое вероятно замедляло обучение. Наконец, могут появляться визуальные артефакты из-за того, что тот же самый шум используется на разных уровнях. Если взамен генератор пытался выпускать собственный псевдослучайный шум, то такой шум мог не выглядеть очень убедительным, приводя к дополнительным визуальным артефактам. Вдобавок для генерации псевдослучайного шума должна была выделяться часть весов генератора, что также кажется расточительным. За счет добавления дополнительных входов шума все упомянутые проблемы исчезают; сеть GAN способна применять предоставленный шум, чтобы добавить надлежащую величину стохастичности к каждой части изображения.

Добавленный шум отличается на каждом уровне. Каждый вход шума состоит из одиночной карты признаков, полной гауссова шума, которая ретранслируется всем картам признаков (заданного уровня) и перед добавлением масштабируется с использованием выученных масштабных коэффициентов для каждого признака (на рис. 17.20 представлены квадратами с буквой “B”).

В заключение сеть StyleGAN применяет методику под названием *регуляризация смешиванием* (*mixing regularization*) или *смешивание стилей* (*style mixing*), где доля генерируемых изображений производится с использованием двух разных кодировок. В частности, кодировки  $c_1$  и  $c_2$  прошли через сеть отображения, давая два вектора стилей  $w_1$  и  $w_2$ . Затем сеть синтеза генерирует изображение, основываясь на стилях  $w_1$  для первых уровней и на стилях  $w_2$  для оставшихся уровней. Отсекающий уровень выбирается случайным образом. Это не позволяет сети предполагать, что стили на смежных уровнях связаны, а последнее в свою очередь поощряет локальность в сети GAN, т.е. каждый вектор стилей влияет только на ограниченное количество характерных черт в генерируемом изображении.

Существует настолько широкий выбор сетей GAN, что для охвата их всех потребуется отдельная книга. Я надеюсь, что предложенное введение представило вам основные идеи, а главное — желание узнать больше. Если вам никак не дается математическая концепция, тогда возможно есть сообщения в блогах, которые помогут ее лучше понять. Двигайтесь дальше и реализуйте собственную сеть GAN; не огорчайтесь, если поначалу возникнут проблемы с обучением: к сожалению это нормально, а для обеспечения надлежащей работы придется проявить терпение, но результат того стоит. Если вы испытываете затруднения с какой-то деталью реализации, то есть масса реализаций с применением Keras или TensorFlow, с которыми имеет смысл ознакомиться. В сущности, если вы хотите всего лишь быстро получить поразительные результаты, тогда можете просто воспользоваться заранее обученной моделью (например, доступны заранее обученные модели StyleGAN для Keras).

В следующей главе мы перейдем к совершенно другой ветви глубокого обучения: глубокому обучению с подкреплением (*Deep Reinforcement Learning*).

## Упражнения

1. Каковы главные задачи, для которых используются автокодировщики?
2. Предположим, вы хотите обучить классификатор и имеете массу непомеченных обучающих данных, но только 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 эффективность автокодировщика?
4. Что собой представляют понижающий и повышающий автокодировщики? Каков главный риск чрезмерно понижающего автокодировщика? А каков главный риск повышающего автокодировщика?
5. Как соединять веса в многослойном автокодировщике? Какой смысл делать это?
6. Что такое порождающая модель? Можете ли вы назвать вид порождающего автокодировщика?
7. Что такое сеть GAN? Можете ли вы назвать несколько задач, где сети GAN могут вести себя блестяще?

8. Каковы главные трудности при обучении сетей GAN?
9. Попробуйте воспользоваться шумоподавляющим автокодировщиком для предварительного обучения классификатора изображений. Вы можете задействовать MNIST (простейший вариант) или более сложный набор данных с изображениями, такой как CIFAR10 (<https://homl.info/122>), если желаете заняться задачей посложнее. Независимо от выбранного набора данных, выполните следующие шаги.
  - Расщепите набор данных на обучающий и испытательный наборы. Обучите глубокий шумоподавляющий автокодировщик на полном обучающем наборе.
  - Удостоверьтесь в том, что изображения достаточно хорошо реконструируются. 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 500 изображений из обучающего набора. В каком случае она лучше работает — с предварительным обучением или без него?
10. Обучите вариационный автокодировщик на наборе данных с изображениями по своему выбору и примените его для генерации изображений. В качестве альтернативы попробуйте найти непомеченный набор данных, который вас заинтересует, и посмотрите, сумеете ли вы генерировать новые образцы.
11. Обучите сеть DCGAN для работы с выбранным набором данных, содержащим изображения, и используйте его для генерации изображений. Добавьте воспроизведение опыта и посмотрите, помогло ли это. Превратите ее в условную порождающую состязательную сеть, в которой можно контролировать генерируемый класс.

Решения приведенных упражнений доступны в приложении А.



# 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 многие годы произвело много интересных приложений<sup>1</sup>, особенно в игровой отрасли (например, *TD-Gammon* — программу для игры в нарды) и в управлении станками, но нечасто попадало в заголовки новостей. Однако в 2013 году произошел крутой перелом, когда исследователи из британского стартапа под названием DeepMind продемонстрировали систему, которая могла научиться играть практически в любую игру Atari с нуля (<https://homl.info/dqn>)<sup>2</sup> и со временем превосходить людей (<https://homl.info/dqn2>)<sup>3</sup> в большинстве игр, используя только низкоуровневые пиксели в качестве входов и не располагая предварительными знаниями правил игр<sup>4</sup>. Это было первым в серии удивительных трюков, кульминацией которых стал выигрыш их системы AlphaGo у Ли Седоля (легендарного профессионального игрока в го) в марте 2016 года и у Кэ Цзе (чемпиона мира) в мае 2017 года. Ни одна программа даже близко не подбиралась к выигрышу у мастера в данной игре, не говоря уже о чемпионе мира. В наши дни вся область обучения с подкреплением бурлит новыми идеями с широким спектром приложений. В 2014 году стартап DeepMind был приобретен компанией Google за сумму, превышающую 500 миллионов долларов.

<sup>1</sup> Дополнительные детали ищите в книге *Reinforcement Learning: An Introduction* (Обучение с подкреплением: введение) Ричарда Сэттона и Эндрю Барто (MIT Press; <https://homl.info/126>).

<sup>2</sup> Володимир Мніх и др., *Playing Atari with Deep Reinforcement Learning* (Игра в игры Atari с помощью глубокого обучения с подкреплением), препринт arXiv:1312.5602 (2013 г.).

<sup>3</sup> Володимир Мніх и др., *Human-Level Control Through Deep Reinforcement Learning* (Контроль человеческого уровня посредством глубокого обучения с подкреплением), *Nature* 518 (2015 г.): с. 529–533.

<sup>4</sup> Просмотрите видеоролики с демонстрацией обучения системы DeepMind играм *Space Invaders*, *Breakout* и другим (<https://homl.info/dqn3>).

Как же исследователи DeepMind 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, после чего опишем две наиболее важных методики 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 методики для обучения моделей балансировать дышлом в двигающейся телеге. Затем вы ознакомитесь с библиотекой TF-Agents, в которой применяются современные алгоритмы, значительно упрощающие построение мощных систем RL. Мы будем использовать упомянутую библиотеку для обучения агента играть в знаменитую игру *Atari* под названием *Breakout*. В заключение мы взглянем на несколько последних достижений в этой области.

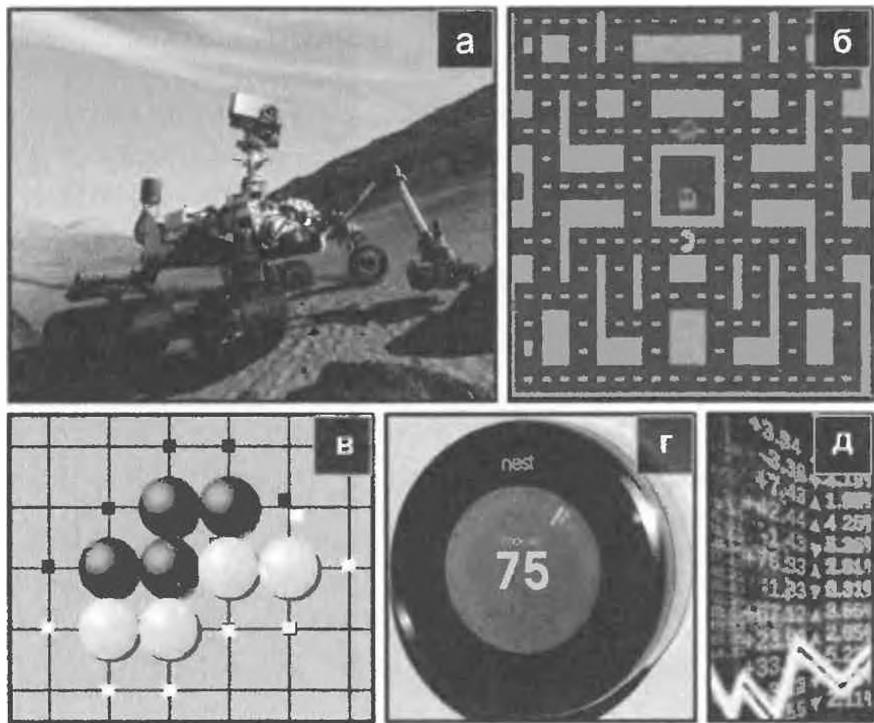
## Обучение для оптимизации наград

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 со временем довести до максимума ожидаемые 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.

Такая довольно обширная установка может применяться к широкому выбору задач. Ниже описано несколько примеров (рис. 18.1).

- Агент может быть программой, которая управляет 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 или двигается в ошибочном направлении.

- 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, по центру и т.д.), наблюдения — экранные снимки, а награды — очки в игре.
- в) Аналогично агент может быть программой, играющей в настольную игру, такую как го.



*Рис. 18.1. Примеры обучения с подкреплением: (а) робототехника, (б) мисс Пэкмен, (в) игрок в го, (г) термостат, (д) автоматический биржевой маклер<sup>5</sup>*

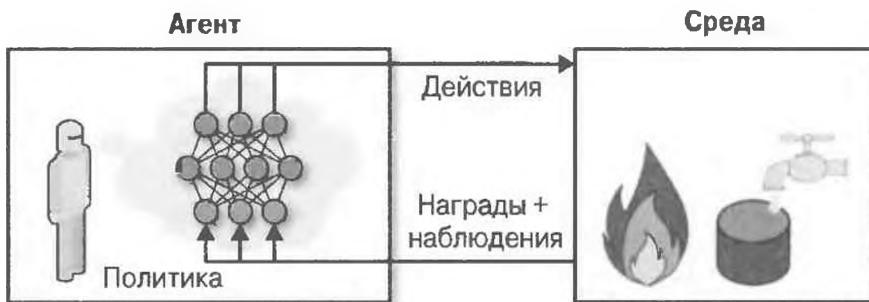
<sup>5</sup> Изображение (а) взято у NASA (публичная собственность). Изображение (б) является экранным снимком из игры Ms. Pac-Man, принадлежащей Atari (законно используется в этой главе). Изображения (в) и (г) воспроизведены из англоязычной Википедии. Изображение (в) было создано пользователем Stevertigo и выпущено под лицензией Creative Commons BY-SA 2.0 (<https://creativecommons.org/licenses/by-sa/2.0/>). Изображение (г) находится в публичной собственности. Изображение (д) воспроизведено из Pixabay и выпущено под лицензией Creative Commons CC0 (<https://creativecommons.org/publicdomain/zero/1.0/>).

- г) Агент вовсе не обязан управлять вещью, двигающейся 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 ежесекундно решать, сколько акций покупать или продавать. Наградами вполне очевидно являются денежные доходы и убытки.

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, системы выдачи рекомендаций, размещение рекламы на веб-странице или управление тем, куда должна сосредоточивать свое внимание система классификации изображений.

## Поиск политики

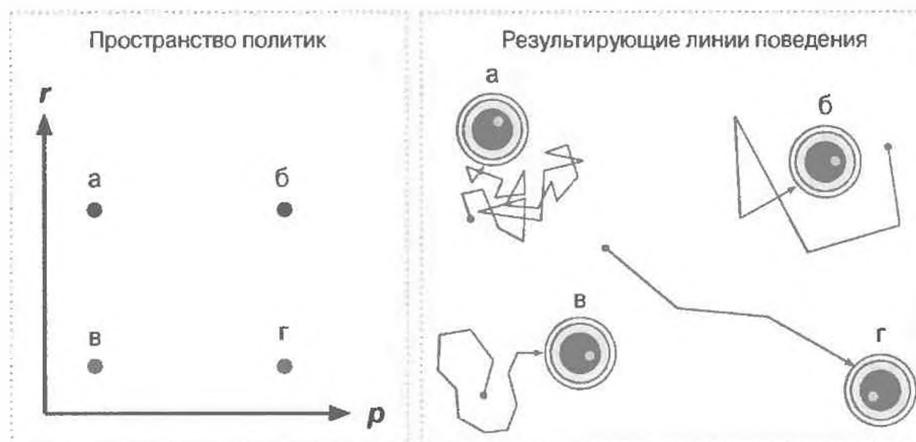
Алгоритм, используемый программным агентом для определения своих действий, называется *политикой* (*policy*). Политикой может служить нейронная сеть, которая в качестве входов принимает наблюдения и выдает действие, подлежащее выполнению (рис. 18.2).



*Рис. 18.2. Обучение с подкреплением, использующее нейросетевую политику*

Политикой может быть любой алгоритм, какой вы только в состоянии вообразить, и он вовсе не обязан быть детерминированным. На самом деле в некоторых случаях ему даже не нужно наблюдать за средой! 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  $-t$  и  $+t$ . 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  $t$ . Один возможный алгоритм обучения мог бы предполагать опробование множества разных значений для указанных параметров и выбор комбинации, которая работает наилучшим образом (рис. 18.3). Это пример *поиска политики* (*policy search*), в данном случае с применением подхода грубой силы. Когда *пространство политик* (*policy space*) слишком велико (как обычно случается), нахождение хорошего набора параметров подобным способом будет похож на поиск иголки в гигантском стоге сена.



**Рис. 18.3.** Четыре точки в пространстве политик (слева) и соответствующее поведение агента (справа)

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 политик<sup>6</sup> и заставить каждую из 20 выживших политик произвести по 4 потомка. Потомок представляет собой просто копию своего родителя<sup>7</sup>, 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<sup>8</sup>.

Еще один подход предусматривает применение методик оптимизации с целью оценки градиентов наград относительно параметров политики и затем подстройки этих параметров, следя по градиентам в направлении более высоких наград<sup>9</sup>. Мы обсудим такой подход, называемый *градиентами политики* (*policy gradient* — PG), более подробно далее в главе. В случае робота-пылесоса вы могли бы чуть повысить  $p$  и выяснить, привело ли повышение  $p$  к увеличению объема пыли, собранной им за 30 минут, и если привело, тогда еще больше повысить  $p$ , а если нет, то 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

<sup>6</sup> Часто лучше предоставить плохим исполнителям небольшой шанс выжить, чтобы сохранить некоторое разнообразие в генофонде.

<sup>7</sup> 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.

<sup>8</sup> Интересным примером генетического алгоритма, используемого для обучения с подкреплением, является алгоритм *нейроэволюции нарастающих топологий* (*Neuro-Evolution of Augmenting Topologies* — NEAT; <https://hml.info/neat>).

<sup>9</sup> Это называется *градиентным подъемом* (*Gradient Ascent*). Он похож на градиентный спуск, но движется в противоположном направлении, доводя до максимума, а не до минимума.

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 обучение параллельно сразу 1 000 роботов обычно сопряжено со слишком высокими затратами. Короче говоря, обучение в реальном мире является трудным и медленным, так что в большинстве случаев нужна имитированная среда, по крайней мере, для запуска обучения. Например, вы можете применять библиотеку вроде PyBullet (<https://pybullet.org/>) или MuJoCo (<http://www.mujoco.org/>) для симуляции трехмерных физических существ.

*OpenAI Gym* (<https://gym.openai.com/>)<sup>10</sup> — 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.

Если вы создавали изолированную среду с использованием `virtualenv`, то перед установкой комплекта инструментов должны ее активировать:

```
$ cd $ML_PATH          # ваш рабочий каталог МО (например, $HOME/ml)
$ source my_env/bin/activate    # в Linux или MacOS
$ .\my_env\Scripts\activate    # в Windows
```

Затем установите OpenAI Gym (если вы не применяете виртуальную среду, тогда вам придется добавить параметр `--user` или располагать правами администратора):

```
$ python3 -m pip install --upgrade gym
```

В зависимости от системы вам также может понадобиться установить библиотеку *Mesa OpenGL Utility (GLU)*; скажем, в Ubuntu 18.04 необходимо выполнить команду `apt install libglu1-mesa`. Библиотека GLU будет нужна для визуализации первой среды. Далее откройте командную оболочку Python или тетрадь Jupyter и создайте среду с помощью `make()`:

<sup>10</sup> OpenAI — 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 уничтожат его).

```
>>> import
>>> env = gym.make("CartPole-v1")
>>> obs = env.reset()
>>> obs
array([-0.01258566, -0.00156614, 0.04207708, -0.00180545])
```

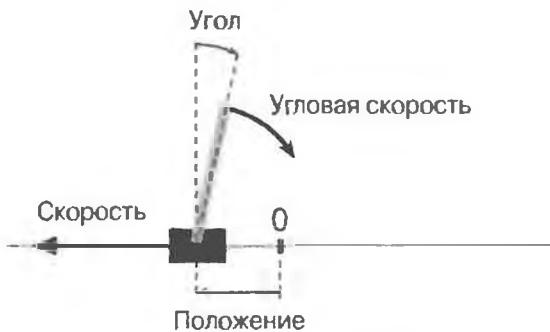
Здесь мы создали среду CartPole. Она представляет собой двумерную симуляцию, в которой телега может ускоряться влево или вправо, чтобы балансировать дышло, размещенное сверху (рис. 18.4). Список всех доступных сред можно получить с использованием `gym.envs.registry.all()`. После того, как среда создана, ее потребуется инициализировать с применением метода `reset()`, который возвращает первое наблюдение. Наблюдения зависят от типа среды. Для среды CartPole каждое наблюдение является одномерным массивом NumPy, содержащим четыре числа с плавающей точкой: они представляют горизонтальное положение телеги (0.0 = центр), скорость телеги (положительное число означает направление вправо), угол дышла (0.0 = вертикально) и угловую скорость дышла (положительное число означает направление по часовой стрелке).

Теперь давайте отобразим среду CartPole, вызвав ее метод `render()` (рис. 18.4). В Windows сначала необходимо установить X Server, такой как VcXsrv или Xming:

```
>>> env.render()
True
```



Если вы используете автоматически работающий сервер (т.е. без экрана), такой как виртуальная машина в облаке, тогда визуализация потерпит неудачу. Единственный способ избежать этого предусматривает применение поддельного сервера X, подобного Xvfb или Xdummy. Например, вы можете установить Xvfb (командой `apt install xvfb` в Ubuntu или Debian) и запустить Python с использованием следующей команды: `xvfb-run -s "-screen 0 1400x900x24" python3`. В качестве альтернативы установите библиотеку pyvirtualdisplay (<https://homl.info/pyvd>), которая является оболочкой Xvfb, и запустите `pyvirtualdisplay.Display(visible=0, size=(1400, 900)).start()` в начале своей программы.



*Рис. 18.4. Среда CartPole*

Если вы хотите, чтобы метод `render()` возвращал визуализированное изображение в виде массива NumPy, то можете установить `mode="rgb_array"` (странны, но эта среда будет также визуализировать и на экране):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape # высота, ширина, каналы (3 = красный, зеленый, синий)
(800, 1200, 3)
```

Выясним у среды, какие действия возможны:

```
>>> env.action_space
Discrete(2)
```

Результат `Discrete(2)` означает, что возможными действиями будут целые числа 0 и 1, которые представляют ускорение влево (0) или вправо (1). Другие среды могут иметь дополнительные дискретные действия или другие типы действий (например, непрерывные). Поскольку дышло кренится вправо (`obs[2] > 0`), ускорим телегу вправо:

```
>>> action = 1 # ускорение вправо
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.01261699,  0.19292789,  0.04204097, -0.28092127])
>>> reward
1.0
>>> done
False
>>> info
{}
```

Метод `step()` выполняет заданное действие и возвращает четыре значения, которые описаны ниже.

## **obs**

Новое наблюдение. Телега теперь двигается 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 эпизод (`episode`) продолжался как можно дольше.

## **done**

Значением будет `True`, когда эпизод закончился. Подобное произойдет, когда дышло даст слишком большой крен, покинет экран или после прохождения 200 шагов (в последнем случае выигрыш ваш). Затем среда должна быть сброшена, прежде чем ее можно будет использовать снова.

## **info**

Специфичный для среды словарь, который может предоставлять дополнительную информацию, пригодную для целей отладки или обучения. Скажем, в некоторых играх он может указывать, сколько жизней имеет агент.



После завершения работы со средой вы должны вызвать ее метод `close()`, чтобы освободить ресурсы.

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(200):
    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. Взглянем на результат:

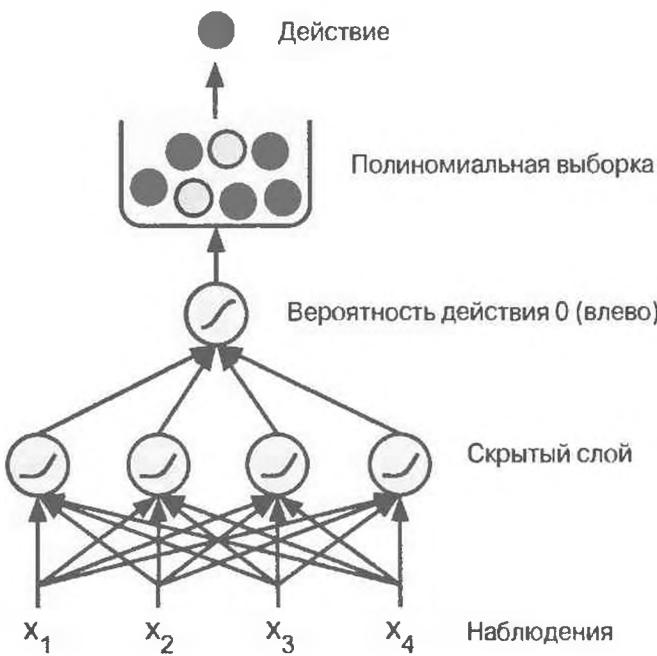
```
>>> import      as
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(41.718, 8.858356280936096, 24.0, 68.0)
```

Даже при 500 попытках данная политика так и не смогла удерживать дышло прямо на протяжении более 68 последовательных шагов. Не особо впечатляет. Если вы просмотрите симуляцию в тетрадях Jupyter (<https://github.com/ageron/handson-m12>), 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.

## Нейросетевые политики

Давайте создадим нейросетевую политику. Подобно политике, жестко за-кодированной 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 оценкам вероятностей (рис. 18.5). В случае среды CartPole есть всего два возможных действия (влево или вправо), так что нам необходим один 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% или 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.



*Рис. 18.5. Нейросетевая политика*

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 `t.f.keras`:

```
import          as
from           import keras

n_inputs = 4    # == env.observation_space.shape[0]

model = keras.models.Sequential([
    keras.layers.Dense(5, activation="elu", input_shape=[n_inputs]),
    keras.layers.Dense(1, activation="sigmoid"),
])

```

После импорта мы используем простую модель `Sequential` для определения политики в форме сети. Количество входов соответствует размеру пространства наблюдений (которое в случае CartPole равно 4) и мы имеем всего лишь пять скрытых элементов, т.к. задача проста. Наконец, мы хотим выдавать единственную вероятность (вероятность движения влево) и потому имеем один выходной нейрон, применяющий сигмоидальную функцию активации. Если бы возможных действий было больше двух, тогда потребовалось бы предусмотреть по одному выходному нейрону на действие и взамен использовать многопеременную функцию активации.

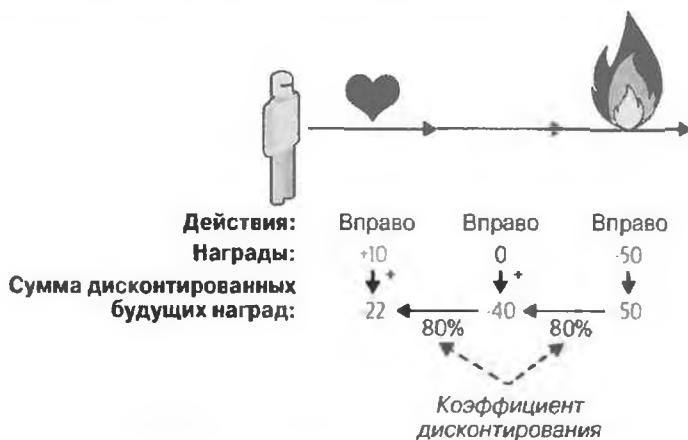
Итак, мы располагаем нейросетевой политикой, которая будет принимать наблюдения и выдавать вероятности действий. Но как ее обучить?

## 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 оценочным и целевым распределением вероятностей. 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 удается балансировать дышло для 100 шагов, тогда как он может узнать, какие из предпринятых им 100 действий были хорошими, а какие плохими? Ему лишь известно то, 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 проблемы заключается в оценке действия на основе суммы всех наград, которые поступили после него, обычно с применением коэффициента дисконтирования (*discount factor*)  $\gamma$  (гамма) на каждом шаге. Такая сумма дисконтированных наград называется *отдачей* (*return*) действия. Возьмем пример на рис. 18.6. 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 коэффициенте дисконтирования  $\gamma = 0.8$  первое действие будет иметь отдачу  $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$ . Если коэффициент дисконтирования близок к 0, то будущие награды окажутся менее значимыми в сравнении с недавними наградами. И наоборот, если коэффициент дисконтирования близок к 1, тогда награды далеко в будущем станут почти такими же весомыми, как недавние награды. Типичные коэффициенты дисконтирования варьируются от 0.9 до 0.99. При коэффициенте дисконтирования 0.95 награды 13 шагов в направлении будущего представляют собой приблизительно половину недавних наград (т.к.  $0.95^{13} \approx 0.5$ ), а при коэффициенте дисконтирования 0.99 награды 69 шагов в направлении будущего дадут половину недавних наград. В среде CartPole действия оказывают относительно краткосрочное влияние, поэтому выбор коэффициента дисконтирования 0.95 выглядит разумным.



**Рис. 18.6. Расчет отдачи действия: сумма дисконтированных будущих наград**

Разумеется, за хорошим действием может следовать несколько плохих действий, которые приведут к быстрому опусканию дышла, в результате чего хорошее действие получит низкую отдачу (подобно тому, как хороший актер иногда может сыграть главную роль в отвратительном фильме). Однако если мы играем в игру достаточное количество раз, то в среднем хорошие действия будут иметь более высокую отдачу, чем плохие действия. Мы хотим оценить, насколько в среднем действие лучше или хуже по сравнению с остальными возможными действиями. Оценка называется *преимуществом действия* (*action advantage*). Для этого мы должны прогнать много эпизодов и нормализовать все отдачи действий (путем вычитания среднего и деления на стандартное отклонение). Затем мы можем корректно предполагать, что действия с отрицательным преимуществом были плохими, а действия с положительным преимуществом — 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, следя по градиентам в сторону более высоких наград. Популярный класс алгоритмов PG, называемый *алгоритмами REINFORCE* (REward Increment = nonnegative Factor  $\times$  Offset Reinforcement  $\times$  Characteristic Eligibility — прирост наград = неотрицательный множитель  $\times$  подкрепление вознаграждения  $\times$  характерный возможный выбор), был введен Рональдом Уильямсом еще в 1992 году (<https://homl.info/132>)<sup>11</sup>. Ниже описан его распространенный вариант.

1. Дайте возможность нейросетевой политике сыграть в игру несколько раз и на каждом шаге вычисляйте градиенты, которые сделают выбранное действие даже более вероятным, но пока не применяйте рассчитанные градиенты.
2. После прогона нескольких эпизодов подсчитайте преимущество каждого действия (используя метод, который описан в предыдущем разделе).

<sup>11</sup> Рональд Уильямс, *Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning* (Простые статистические алгоритмы, следующие по градиентам, для коннекционного обучения с подкреплением), *Machine Learning* 8 (1992 г.) : с. 229–256.

3. Если преимущество действия положительное, то это означает, что действие, видимо, было хорошим, и вы хотите применить вычисленные ранее градиенты, чтобы сделать выбор данного действия в будущем еще более вероятным. Если же преимущество действия отрицательное, то это значит, что действие, видимо, 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 соответствующее преимущество действия.
4. В заключение вычислите все результирующие векторы-градиенты и воспользуйтесь ими для выполнения шага градиентного спуска.

Далее мы реализуем представленный выше алгоритм с применением `tf.keras`. Мы обучим созданную ранее нейросетевую политику, чтобы она научилась балансировать дышло в телеге. Прежде всего, нам нужна функция, которая будет воспроизводить один шаг. Пока мы притворимся, что любое предпринимаемое ею действие является правильным, поэтому можно рассчитать потерю и ее градиенты (вычисленные градиенты просто сохраняются на некоторое время и позже модифицируются в зависимости от того, насколько хорошим или плохим оказалось действие):

```
def play_one_step(env, obs, model, loss_fn):  
    with tf.GradientTape() as tape:  
        left_proba = model(obs[np.newaxis])  
        action = (tf.random.uniform([1, 1]) > left_proba)  
        y_target = tf.constant([[1.]]) - tf.cast(action, tf.float32)  
        loss = tf.reduce_mean(loss_fn(y_target, left_proba))  
        grads = tape.gradient(loss, model.trainable_variables)  
        obs, reward, done, info = env.step(int(action[0, 0].numpy()))  
    return obs, reward, done, grads
```

Давайте пройдемся по функции.

- Внутри блока `GradientTape` (см. главу 12) мы начинаем с вызова модели, передавая ей одиночное наблюдение (с изменением формы наблюдения, чтобы оно превратилось в пакет, содержащий единственный образец, т.к. модель ожидает именно пакет). Результатом будет вероятность движения влево.
- Затем мы производим выборку случайного числа с плавающей точкой между 0 и 1 и проверяем, больше ли оно, чем `left_proba`. Значением

`action` будет `False` с вероятностью `left_proba` или `True` с вероятностью  $1 - \text{left\_proba}$ . После приведения указанного булевского значения к числовому типу действием окажется 0 (влево) или 1 (вправо) с соответствующими вероятностями.

- Далее мы определяем целевую вероятность движения влево: она составляет 1 минус действие (приведенное к числу с плавающей точкой). Если действие равно 0 (влево), тогда целевой вероятностью движения влево будет 1. Если действие равно 1 (вправо), тогда целевой вероятностью движения влево будет 0.
- Затем мы рассчитываем потерю, используя заданную функцию потерь, и с помощью ленты вычисляем градиенты потери относительно обучаемых переменных модели. И снова эти градиенты будут подстраиваться позже, перед их применением, в зависимости от того, насколько хорошим или плохим оказалось действие.
- В заключение мы воспроизводим выбранное действие и возвращаем новое наблюдение, награду, признак окончания эпизода и, конечно же, только что вычисленные градиенты.

Теперь давайте создадим еще одну функцию, которая будет полагаться на функцию `play_one_step()` в воспроизведении множества эпизодов и возвращать все награды и градиенты для каждого эпизода и каждого шага:

```
def play_multiple_episodes(env, n_episodes, n_max_steps, model, loss_fn):  
    all_rewards = []  
    all_grads = []  
    for episode in range(n_episodes):  
        current_rewards = []  
        current_grads = []  
        obs = env.reset()  
        for step in range(n_max_steps):  
            obs, reward, done,  
            grads = play_one_step(env, obs, model, loss_fn)  
            current_rewards.append(reward)  
            current_grads.append(grads)  
            if done:  
                break  
        all_rewards.append(current_rewards)  
        all_grads.append(current_grads)  
    return all_rewards, all_grads
```

Код возвращает список списков наград (один список наград для каждого эпизода, содержащий по одной награде на шаг), а также список списков градиентов (один список градиентов для каждого эпизода, содержащий по одному кортежу градиентов на шаг, а каждый кортеж включает по одному градиентному тензору на обучаемую переменную).

Алгоритм будет использовать функцию `play_multiple_episodes()`, чтобы сыграть в игру несколько раз (скажем, 10 раз), после чего он возвратится назад с целью просмотра всех наград, их дисконтирования и нормализации. Для этого нам понадобится еще пара функций: первая будет вычислять сумму дисконтированных будущих наград на каждом шаге, а вторая займется нормализацией всех дисконтированных наград (отдач) по множеству эпизодов путем вычитания среднего и деления на стандартное отклонение:

```
def discount_rewards(rewards, discount_factor):
    discounted = np.array(rewards)
    for step in range(len(rewards) - 2, -1, -1):
        discounted[step] += discounted[step + 1] * discount_factor
    return discounted

def discount_and_normalize_rewards(all_rewards, discount_factor):
    all_discounted_rewards = (discount_rewards(rewards,
                                                discount_factor)
                               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_factor=0.8)
array([-22, -40, -50])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]],
...                                discount_factor=0.8)
...
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([1.26665318, 1.07277777])]
```

Вызов `discount_rewards()` возвращает именно то, что ожидалось (см. рис. 18.6). Вы можете удостовериться в том, что функция `discount_and_normalize_rewards()` действительно возвращает нормализованные

преимущества для всех действий в обоих эпизодах. Обратите внимание, что первый эпизод был гораздо хуже второго, поэтому все его нормализованные преимущества являются отрицательными; все действия из первого эпизода будут считаться плохими и наоборот — все действия из второго эпизода будут считаться хорошими.

Мы почти готовы к запуску алгоритма! Теперь давайте определим гиперпараметры. Мы выполним 150 итераций обучения, воспроизводя на каждой итерации по 10 эпизодов, каждый из которых будет продолжаться не более 200 шагов. Мы будем применять коэффициент дисконтирования 0.95:

```
n_iterations = 150
n_episodes_per_update = 10
n_max_steps = 200
discount_factor = 0.95
```

Нам также понадобятся оптимизатор и функция потерь. Здесь прекрасно подойдет обычный оптимизатор Adam со скоростью обучения 0.01, и мы будем использовать функцию потерь двоичной перекрестной энтропии, потому что обучаем двоичный классификатор (есть два возможных действия: влево или вправо):

```
optimizer = keras.optimizers.Adam(lr=0.01)
loss_fn = keras.losses.binary_crossentropy
```

Итак, мы готовы построить и запустить цикл обучения!

```
for iteration in range(n_iterations):
    all_rewards, all_grads = play_multiple_episodes(
        env, n_episodes_per_update, n_max_steps, model, loss_fn)
    all_final_rewards = discount_and_normalize_rewards(all_rewards,
                                                       discount_factor)

    all_mean_grads = []
    for var_index in range(len(model.trainable_variables)):
        mean_grads = tf.reduce_mean([
            final_reward * all_grads[episode_index][step][var_index]
            for episode_index, final_rewards
            in enumerate(all_final_rewards)
            for step, final_reward
            in enumerate(final_rewards)], axis=0)
        all_mean_grads.append(mean_grads)
    optimizer.apply_gradients(zip(all_mean_grads,
                                 model.trainable_variables))
```

Разберем показанный выше код.

- На каждой итерации обучения мы вызываем в цикле функцию `play_multiple_episodes()`, которая играет в игру 10 раз и возвращает все награды и градиенты для каждого эпизода и шага.
- Затем мы вызываем функцию `discount_and_normalize_rewards()`, чтобы рассчитать нормализованное преимущество каждого действия (в коде именуется `final_reward`). Это дает меру того, насколько хорошим или плохим было действие с оглядкой на прошлое.
- Далее мы проходим по всем обучаемым переменным и для каждой из них вычисляем взвешенную посредством `final_reward` среднюю величину градиентов по всем эпизодам и всем шагам.
- Наконец, мы применяем рассчитанные средние градиенты, используя оптимизатор: обучаемые переменные модели будут подстраиваться в надежде, что политика станет чуть лучше.

Мы все сделали! Код будет обучать нейросетевую политику, которая успешно научится балансировать дышло в телеге (можете испытать ее — она находится в разделе “Policy Gradients” (Градиенты политики) тетради Jupyter для настоящей главы). Средняя награда на эпизод будет очень близкой к 200 (что по умолчанию является максимумом в данной среде). Успех!



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 без каких-либо колебаний внедрить в агента априорные знания, т.к. они значительно ускорят обучение. Например, поскольку вы знаете, что дышло должно располагаться насколько возможно вертикально, то вы могли бы добавлять отрицательные награды, пропорциональные углу дышла. В итоге награды стали бы гораздо менее разреженными, а обучение ускорилось. Кроме того, если вы уже имеете достаточно успешную политику (скажем, жестко закодированную), тогда у вас может возникнуть желание обучить нейронную сеть с целью ее имитации, прежде чем применять градиенты политики для ее улучшения.

Только что обученный простой алгоритм градиентов политики решает задачу CartPole, но не будет хорошо масштабироваться на более крупные и сложные задачи. В действительности он крайне незэффективен с точки зрения выборки, т.к. ему необходимо исследовать игру очень долгое время, прежде чем он сумеет добиться значительного прогресса. Причина связана с тем фактом, что алгоритм должен прогонять множество эпизодов для оценки преимущества каждого действия, как мы видели. Однако он является основой более мощных алгоритмов, таких как алгоритмы “актор-критик” (*Actor-Critic*), которые мы кратко обсудим в конце главы.

А теперь мы взглянем на еще одно популярное семейство алгоритмов. В то время как алгоритмы PG пытаются напрямую оптимизировать политику, чтобы увеличить награды, рассматриваемые далее алгоритмы менее прямолинейны. Агент учится оценивать ожидаемую отдачу для каждого состояния или для каждого действия в каждом состоянии и затем использует это знание при выработке решения о том, как действовать. Чтобы понять такие алгоритмы, сначала нужно ознакомиться с марковскими процессами принятия решений.

## Марковские процессы принятия решений

В начале двадцатого века математик Андрей Марков изучал стохастические процессы без памяти, называемые цепями Маркова (*Markov chains*). 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 (вот почему мы говорим, что система не имеет памяти).

На рис. 18.7 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, потому что ни одно из других состояний не указывает на  $s_0$ . Если процесс перейдет в состояние  $s_1$ , то затем наиболее вероятно 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).

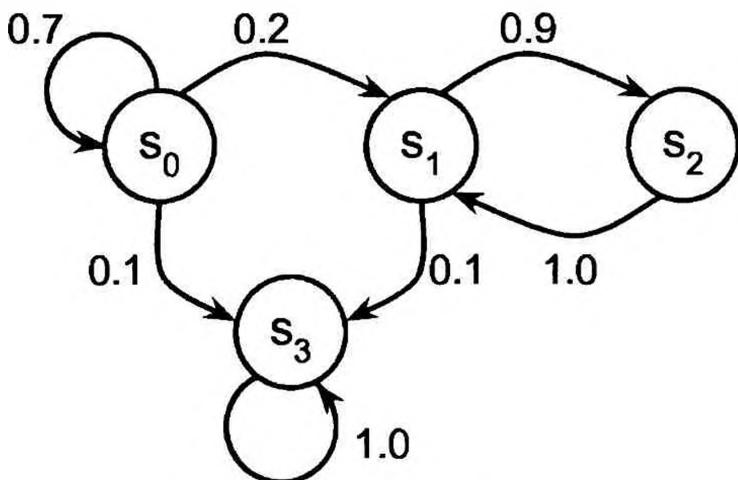


Рис. 18.7. Пример цепи Маркова

Цепи Маркова могут обладать очень разными динамическими свойствами и широко применяются в термодинамике, химии, статистике и многих других областях.

Марковские процессы принятия решений впервые были описаны в 1950-х годах Ричардом Беллманом (<https://homl.info/133>)<sup>12</sup>. 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 награду с течением времени.

Например, представленный на рис. 18.8 процесс MDP имеет три состояния (представлены кругами) и до трех возможных дискретных действий на каждом шаге (представлены ромбами).

Если он начинает с состояния  $s_0$ , то 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. Однако в случае выбора действия  $a_0$  есть вероятность 70% получить награду +10 и остаться в состоянии  $s_0$ . Далее попытка может повторяться снова и снова для получения настолько большой

<sup>12</sup> Ричард Беллман, *A Markovian Decision Process* (Марковский процесс принятия решений), *Journal of Mathematics and Mechanics* 6, номер 5 (1957 г.); с. 679–684.

награды, насколько это возможно, но в какой-то момент он взамен переходит в состояние  $s_1$ . В состоянии  $s_1$  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 попутной наградой  $+40$ . 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 оставаться на месте ( $a_0$ ) либо пройти сквозь огонь ( $a_2$ ).

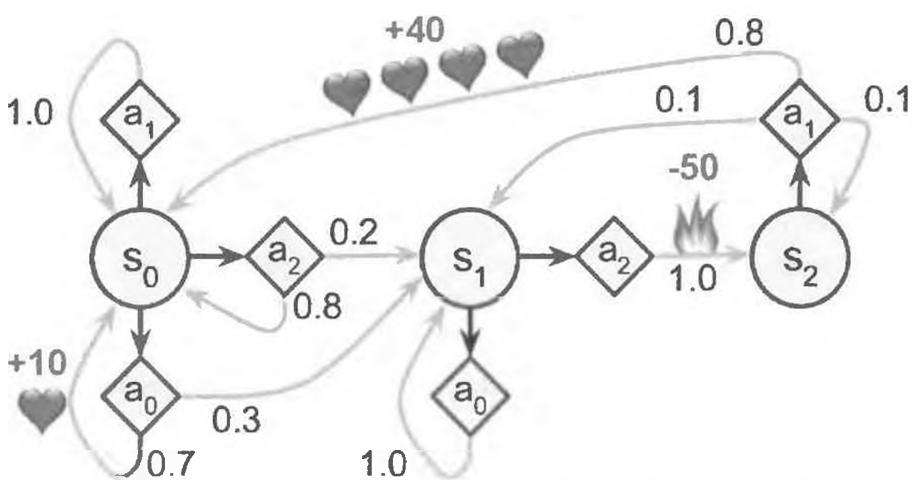


Рис. 18.8. Пример марковского процесса принятия решений

Беллман нашел способ оценки оптимальной ценности состояния (*optimal state value*) для любого состояния  $s$ , обозначаемой  $V^*(s)$ . Она представляет собой сумму всех дисконтированных будущих наград, которые агент может ожидать в среднем после достижения состояния  $s$ , исходя из предположения, что он действует оптимально. Беллман показал, что если агент действует оптимально, то применимо *уравнение оптимальности Беллмана* (*Bellman Optimality Equation*), приведенное в уравнении 18.1. 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 18.1. Уравнение оптимальности Беллмана

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V^*(s')] \quad \text{для всех } s$$

Проанализируем уравнение 18.1.

- $T(s, a, s')$  — вероятность перехода из состояния  $s$  в состояние  $s'$  при условии, что агент выбрал действие  $a$ . Скажем, на рис. 18.8 мы имеем  $T(s_2, a_1, s_0) = 0.8$ .
- $R(s, a, s')$  — награда, которую агент получает, когда переходит из состояния  $s$  в состояние  $s'$  при условии, что он выбрал действие  $a$ . Например, на рис. 18.8 мы имеем  $R(s_2, a_1, s_0) = +40$ .
- $\gamma$  — коэффициент дисконтирования.

Уравнение 18.1 напрямую приводит к алгоритму, который способен 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 18.2. Замечательный результат заключается в том, что при достаточном времени оценки гарантированно сойдутся в оптимальные ценности состояний, соответствующие оптимальной политике.

### Уравнение 18.2. Алгоритм итерации по ценностям

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \cdot V_k(s')] \quad \text{для всех } s$$

В этом уравнении  $V_k(s)$  — оценочная ценность состояния  $s$  на  $k$ -той итерации алгоритма.



Алгоритм итерации по ценностям является примером *динамического программирования*, которое разбивает сложную задачу на легко поддающиеся обработке подзадачи, допускающие итеративное решение.

Знание оптимальных ценностей состояний может быть полезным в частности при оценке политики, но оно не дает нам оптимальную политику для агента. К счастью, Беллман отыскал очень похожий алгоритм для оценки оп-

тимальных ценностей пар “состояние–действие” (*state-action value*), обычно называемых *Q-ценностями* (*Q-Value*) или показателями качества (*Quality Value*). Оптимальная *Q*-ценность пары “состояние–действие” ( $s, a$ ), обозначаемая  $Q^*(s, a)$ , представляет собой сумму дисконтированных будущих наград, 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, предполагая его оптимальное поведение после действия.

Вот как работает прием: снова все оценки *Q*-ценностей изначально инициализируются нулями, а затем обновляются с применением алгоритма *итерации по Q-ценностям* (*Q-Value Iteration*), приведенного в уравнении 18.3.

### Уравнение 18.3. 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. 18.8. Прежде всего, необходимо определить процесс MDP:

```
transition_probabilities = [    # 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], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]]
rewards = [    # форма=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Скажем, чтобы узнать вероятность перехода из состояния  $s_2$  в состояние  $s_0$  после воспроизведения действия  $a_1$ , мы будем искать `transition_probabilities[2][1][0]` (что равно 0.8). Аналогично для получения соответствующей награды мы просмотрим `rewards[2][1][0]` (что равно +40). А чтобы получить список возможных действий в состоянии  $s_2$ , мы будем искать `possible_actions[2]` (в данном случае возможно только действие  $a_1$ ). Затем мы должны инициализировать все *Q*-ценности нулями (кроме невозможных действий, для которых *Q*-ценности устанавливаются в  $-\infty$ ):

```
Q_values = np.full((3, 3), -np.inf) # -np.inf для невозможных действий
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0 # для всех возможных действий
```

Теперь запустим алгоритм итерации по Q-ценностям. Он многократно применяет уравнение 18.3 ко всем Q-ценностям для каждого состояния и каждого возможного действия:

```
gamma = 0.90 # коэффициент дисконтирования
for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                transition_probabilities[s][a][sp]
                * (rewards[s][a][sp] + gamma * np.max(Q_prev[sp]))
                for sp in range(3)])
```

Вот так! Результирующие Q-ценности выглядят следующим образом:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.          , -inf, -4.87971488],
       [-inf, 50.13365013, -inf]])
```

Например, когда агент находится в состоянии  $s_0$  и выбирает действие  $a_1$ , ожидаемая сумма дисконтированных будущих наград составляет приблизительно 17.0.

Давайте для каждого состояния взглянем на действие, которое имеет наивысшую Q-ценность:

```
>>> np.argmax(Q_values, axis=1)      # оптимальное действие
                                         # для каждого состояния
array([0, 0, 1])
```

В итоге мы получаем оптимальную политику для данного процесса MDP, когда используется коэффициент дисконтирования 0.90: в состоянии  $s_0$  выбрать действие  $a_0$ , в состоянии  $s_1$  — действие  $a_0$  (т.е. не двигаться) и в состоянии  $s_2$  — действие  $a_1$  (единственное возможное действие). Интересно отметить, что если мы увеличим коэффициент дисконтирования до 0.95, тогда оптимальная политика изменится: в состоянии  $s_1$  наилучшим действием становится  $a_2$  (пройти сквозь огонь!). Это имеет смысл, т.к. чем больше вы цените будущие награды, тем больше готовы терпеть боль сейчас за обещание будущего блаженства.

# Обучение методом временных разностей

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 обязан 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 Learning* — TD) 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 первоначально агенту известны только возможные состояния и действия и ничего более. 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, которые были фактически получены путем наблюдений (уравнение 18.4).

## Уравнение 18.4. Алгоритм обучения TD

$$V_{k+1}(s) \leftarrow (1 - \alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

или эквивалентно:

$$\begin{aligned} V_{k+1}(s) &\leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s') \\ \text{с } \delta_k(s, r, s') &= r + \gamma \cdot V_k(s') - V_k(s) \end{aligned}$$

В этом уравнении:

- $\alpha$  — скорость обучения (например, 0.01);
- $r + \gamma \cdot V_k(s')$  называется целью TD (*TD target*);
- $\delta_k(s, r, s')$  называется ошибкой TD (*TD error*).

Более лаконичный способ записи первой формы уравнения 18.4 предусматривает использование обозначения  $a \xleftarrow{\alpha} b$ , которое означает  $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$ . Таким образом, первую строку в уравнении 18.4 можно переписать в виде  $V(s) \xleftarrow{\alpha} r + \gamma \cdot V(s')$ .



Обучение TD имеет немало сходств со стохастическим градиентным спуском, в частности похоже тем, что обрабатывает по одному образцу за раз. Кроме того, во многом подобно стохастическому градиентному спуску алгоритм обучения TD может по-настоящему сходиться только при постепенном снижении скорости обучения (иначе он продолжит совершать прыжки возле оптимальных Q-ценностей).

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).

## Q-обучение

Подобным образом алгоритм Q-обучения (*Q-Learning*) является адаптацией алгоритма итерации по Q-ценностям к ситуации, когда вероятности переходов и награды изначально не известны (уравнение 18.5). Алгоритм Q-обучения работает, отслеживая игру агента (скажем, случайно) и постепенно улучшая оценки Q-ценностей. Как только он получит точные оценки Q-ценностей (или достаточно близкие), оптимальная политика выбирает действие, которое имеет наивысшую Q-ценность (т.е. политика жадная).

### Уравнение 18.5. Алгоритм Q-обучения

$$Q(s, a) \leftarrow_a r + \gamma \cdot \max_{a'} Q(s', a')$$

Для каждой пары “состояние-переход”  $(s, a)$  алгоритм Q-обучения отслеживает скользящее среднее наград  $r$ , которые агент получает при покидании состояния  $s$  посредством действия  $a$ , плюс сумму дисконтированных будущих наград, которые он ожидает получить. Чтобы оценить такую сумму, мы берем максимальную из оценок Q-ценностей для следующего состояния  $s'$ , т.к. допускаем, что с этого момента целевая политика будет действовать оптимально.

Давайте реализуем алгоритм Q-обучения. Первым делом нам понадобится заставить агент отслеживать среду, для чего нужна ступенчатая функция, чтобы агент мог выполнить одно действие и получить результирующее состояние и награду:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Теперь реализуем политику исследования агента. Из-за того, что пространство состояний довольно-таки мало, простой случайной политики окажется достаточно. Если алгоритм будет выполняться достаточно долго, тогда агент посетит каждое состояние много раз, а также много раз опробует каждое возможное действие:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

После инициализации Q-ценностей, как делалось ранее, мы готовы к запуску алгоритма Q-обучения со снижением скорости обучения (с применением графика мощности, представленного в главе 11):

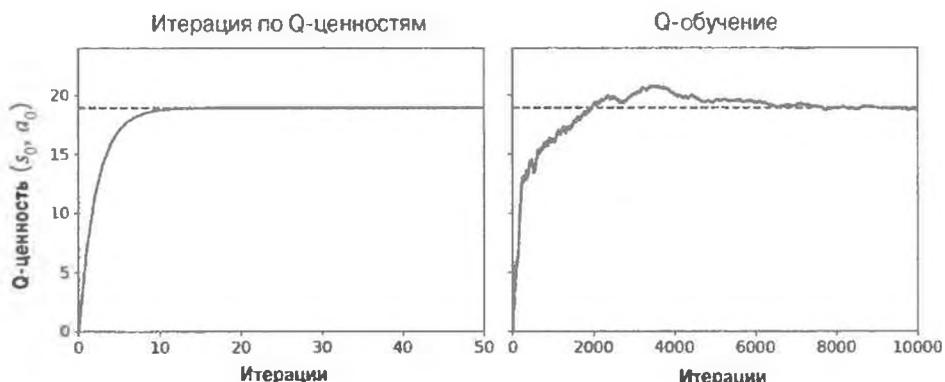
```
alpha0 = 0.05      # начальная скорость обучения
decay = 0.005     # снижение скорости обучения
gamma = 0.90       # коэффициент дисконтирования
state = 0          # начальное состояние

for iteration in range(10000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = np.max(Q_values[next_state])
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

Алгоритм Q-обучения будет сходиться к оптимальным Q-ценностям, но потребует много итераций и возможно немалой подстройки гиперпараметров. Как видно на рис. 18.9, алгоритм итераций по Q-ценностям (слева) сходится очень быстро, за менее, чем 20 итераций, тогда как алгоритму Q-обучения (справа) для схождения понадобится около 8 000 итераций. Очевидно, незнание вероятностей переходов или наград значительно затрудняет нахождение оптимальной политики!

Алгоритм Q-обучения называется алгоритмом *вне политики* (*off-policy*), поскольку политика при обучении не обязательно совпадает с политикой при выполнении: в предыдущем примере кода политика при выполнении (политика исследования) полностью случайна, в то время как политика при

обучении всегда будет выбирать действия с наивысшими Q-ценностями. И наоборот, алгоритм градиентов политики является алгоритмом *внутри политики* (*on-policy*): он исследует мир, используя политику при обучении. Слегка удивляет, что алгоритм Q-обучения 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?



*Рис. 18.9. Сравнение алгоритма итераций по Q-ценностям (слева) и алгоритма Q-обучения (справа)*

## Политики исследования

Разумеется, Q-обучение может работать, только если политика исследования достаточно тщательно анализирует процесс MDP. Хотя чисто случайная политика в итоге гарантированно посетит каждое состояние и пройдет по каждому переходу много раз, это может занять чрезвычайно много времени. Следовательно, более удачным вариантом будет  $\epsilon$ -жадная политика ( $\epsilon$ -greedy policy;  $\epsilon$  — эпсилон): на каждом шаге она ведет себя случайным образом с вероятностью  $\epsilon$  или жадным образом с вероятностью  $1 - \epsilon$  (т.е. выбирая действие с наивысшей Q-ценностью). Преимущество  $\epsilon$ -жадной политики (в сравнении с полностью случайной политикой) заключается в том, что она будет тратить все больше времени на исследование интересных частей среды, т.к. оценки Q-ценностей становятся все лучше, по-прежнему выделяя некоторое время на посещение неизвестных областей процесса MDP. Довольно часто начинают с высокого значения  $\epsilon$  (скажем, 1.0) и постепенно его понижают (например, до 0.05).

Вместо того чтобы полагаться только на шанс исследования, альтернативный подход предусматривает подталкивание политики исследования к испытанию действий, которые ранее не были опробованы часто. Такой прием можно реализовать в виде премии, добавляемой к оценкам Q-ценностей (уравнение 18.6).

### Уравнение 18.6. Q-обучение с применением функции исследования

$$Q(s, a) \leftarrow r + \gamma \cdot \max_{a'} f(Q(s', a'), N(s', a'))$$

В этом уравнении:

- $N(s', a')$  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 связана с тем, что оно плохо масштабируется к крупным (или даже средним) процессам MDP, содержащим много состояний и действий. Скажем, пусть Q-обучение необходимо использовать для того, чтобы научить агента играть в игру Ms. Pac-Man (см. рис. 18.1). Существует около 150 зерен, которые мисс Пэкмен может съесть, и каждое зерно может присутствовать или отсутствовать (т.е. быть съеденным). Таким образом, количество возможных состояний оказывается более чем  $2^{150} \approx 10^{45}$ . А если добавить все возможные комбинации позиций для всех призраков и мисс Пэкмен, то число возможных состояний превысит количество атомов на нашей планете, 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  $\Theta$ ). 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 (например, расстояние до ближайших призраков, направления их передвижения и т.д.). Но в 2013 году система DeepMind (<https://hml.info/dqn>) продемонстрировала, что применение глубоких нейронных сетей может обеспечить гораздо лучшие результаты, особенно при решении 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)*.

А как можно обучить сеть DQN? Рассмотрим приближенную Q-ценность, вычисленную сетью DQN для заданной пары “состояние-действие” ( $s, a$ ). Благодаря Беллману мы знаем, что желательно, чтобы эта приближенная Q-ценность находилась как можно ближе к награде  $r$ , которую мы фактически наблюдаем после выполнения действия  $a$  в состоянии  $s$ , плюс дисконтированная ценность от оптимальной игры, начиная с данного момента. Чтобы оценить такую сумму дисконтированных будущих наград, мы можем просто прогнозировать 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), дисконтируем ее и в результате имеем оценку суммы дисконтированных будущих наград. Складывая награду  $r$  и оценку дисконтированной будущей ценности, мы получаем целевую Q-ценность  $y(s, a)$  для пары “состояние-действие” ( $s, a$ ), как показано в уравнении 18.7.

### Уравнение 18.7. Целевая Q-ценность

$$Q_{\text{целевая}}(s, a) = r + \gamma \cdot \max_{a'} Q_{\theta}(s', a')$$

Располагая целевой Q-ценностью, мы можем запустить шаг обучения с использованием любого алгоритма градиентного спуска. В частности, обычно мы будем пытаться довести до минимума квадратичную ошибку между оценочной Q-ценностью  $Q(s, a)$  и целевой Q-ценностью (или потерю Хьюбера, чтобы понизить чувствительность алгоритма к крупным ошибкам). Вот и весь базовый алгоритм глубокого Q-обучения! Давайте посмотрим, как его реализовать для решения со средой CartPole.

# Реализация глубокого Q-обучения

Первое, что нам понадобится — глубокая Q-сеть. Теоретически необходима нейронная сеть, которая берет пару “состояние-действие” и выдает приближенную Q-ценность, но на практике гораздо эффективнее применять нейронную сеть, которая получает состояние и выдает по одной приближенной Q-ценности для каждого возможного действия. При решении задачи со средой CartPole мы не нуждаемся в особо сложной нейронной сети; достаточно будет пары скрытых слоев:

```
env = gym.make("CartPole-v0")
input_shape = [4]      # == env.observation_space.shape
n_outputs = 2          # == env.action_space.n

model = keras.models.Sequential([
    keras.layers.Dense(32, activation="elu", input_shape=input_shape),
    keras.layers.Dense(32, activation="elu"),
    keras.layers.Dense(n_outputs)
])
```

Чтобы выбрать действие, используя такую сеть DQN, мы ищем действие с наибольшей спрогнозированной Q-ценностью. Для гарантии того, что агент исследует среду, мы будем применять  $\epsilon$ -жадную политику (т.е. выбирать случайное действие с вероятностью  $\epsilon$ ):

```
def epsilon_greedy_policy(state, epsilon=0):
    if np.random.rand() < epsilon:
        return np.random.randint(2)
    else:
        Q_values = model.predict(state[np.newaxis])
        return np.argmax(Q_values[0])
```

Вместо обучения сети DQN на основе только самого последнего опыта мы будем хранить все данные опыта в буфере воспроизведения (или памяти воспроизведения) и на каждой итерации обучения делать выборку оттуда случайногго обучающего пакета. Прием помогает сократить корреляции между опытами в обучающем пакете, что чрезвычайно содействует обучению. Для этого мы просто будем использовать очередь с двусторонним доступом (*deque*):

```
from           import deque
replay_buffer = deque(maxlen=2000)
```



Очередь с двусторонним доступом представляет собой связный список, в котором каждый элемент указывает на следующий и предыдущий элементы. В результате вставка и удаление элементов выполняются очень быстро, но чем длиннее очередь с двусторонним доступом, тем медленнее будет произвольный доступ. Если вам необходим очень крупный буфер воспроизведения, тогда применяйте кольцевой буфер; за реализацией обращайтесь в раздел “Deque vs Rotating List” (Очередь с двусторонним доступом или циклический список) тетради Jupiter для настоящей главы.

Каждый опыт будет состоять из пяти элементов: состояние (`state`), предпринятое агентом действие (`action`), результирующая награда (`reward`), следующее достигнутое им состояние (`next_state`) и, наконец, булевское значение, которое показывает, закончился ли эпизод в данной точке (`done`). Нам будет нужна небольшая функция для выборки случайного пакета опытов из буфера воспроизведения. Она будет возвращать пять массивов NumPy, соответствующих пяти элементам опытов:

```
def sample_experiences(batch_size):
    indices = np.random.randint(len(replay_buffer), size=batch_size)
    batch = [replay_buffer[index] for index in indices]
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

Давайте также создадим функцию, которая будет воспроизводить одиничный шаг с использованием  $\epsilon$ -жадной политики и затем сохранять результирующий опыт в буфере воспроизведения:

```
def play_one_step(env, state, epsilon):
    action = epsilon_greedy_policy(state, epsilon)
    next_state, reward, done, info = env.step(action)
    replay_buffer.append((state, action, reward, next_state, done))
    return next_state, reward, done, info
```

В заключение мы создадим еще одну функцию, которая будет осуществлять выборку пакета опытов из буфера воспроизведения и обучать сеть DQN, выполняя на этом пакете одиничный шаг градиентного спуска:

```

batch_size = 32
discount_factor = 0.95
optimizer = keras.optimizers.Adam(lr=1e-3)
loss_fn = keras.losses.mean_squared_error

def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)
    target_Q_values = (rewards +
        (1 - dones) * discount_factor * max_next_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    with tf.GradientTape() as tape:
        all_Q_values = model(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1,
                               keepdims=True)
    loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Пройдемся по коду.

- Первым делом мы определяем ряд гиперпараметров, а также создаем оптимизатор и функцию потерь.
- Потом мы создаем функцию `training_step()`. Она начинается с выборки пакета опытов, после чего применяет сеть DQN, чтобы спрогнозировать Q-ценность для каждого возможного действия в каждом следующем состоянии каждого опыта. Поскольку мы предполагаем, что агент будет вести игру оптимально, то сохраняем только максимальную Q-ценность для каждого следующего состояния. Далее мы используем уравнение 18.7, чтобы рассчитать целевую Q-ценность для пары “состоиние-действие” каждого опыта.
- Затем мы хотим применить сеть DQN с целью вычисления Q-ценности для каждой испытанной пары “состоиние-действие”. Тем не менее, сеть DQN также будет выдавать Q-ценности для других возможных действий, а не только для действия, которое фактически было выбрано агентом. Следовательно, нам необходимо маскировать все Q-ценности, которые не нужны. Функция `tf.one_hot()` облегчает преобразование массива индексов действий в такую маску. Например, если первые три опыта содержат действия 1, 1, 0 соответственно, тогда маска будет начинаться с `[0, 1], [0, 1], [1, 0], ...`. Мы можем умножить вы-

ход сети DQN на эту маску, что обнулит все Q-ценности, которые нас не интересуют. Наконец, мы суммируем по оси 1, чтобы избавиться от нулей, сохраняя только Q-ценности испытанных пар “состояние-действие”. Итогом будет тензор `Q_values`, содержащий одну Q-ценность для каждого опыта в пакете.

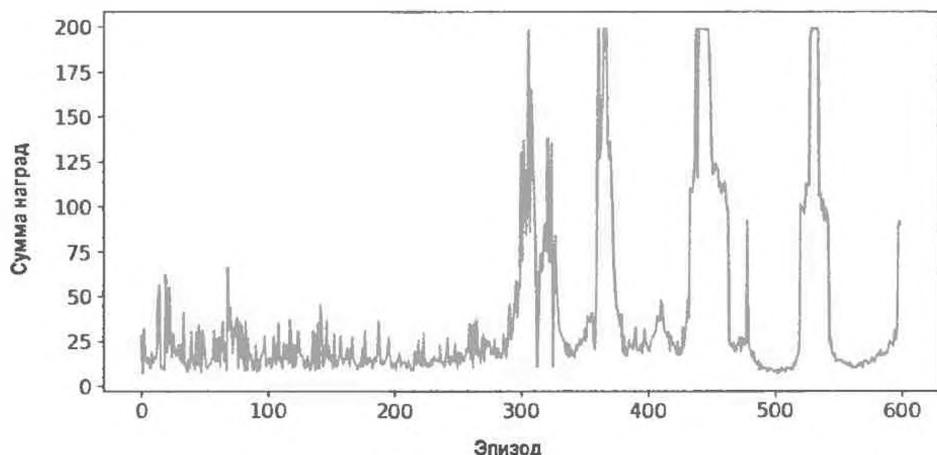
- Далее мы вычисляем потерю: это среднеквадратическая ошибка между целевыми и спрогнозированными Q-ценностями для испытанных пар “состояние-действие”.
- В заключение мы выполняем шаг градиентного спуска, чтобы довести до минимума потерю относительно обучаемых переменных модели.

Самая трудная часть закончена. Теперь обучение модели будет прямолинейным:

```
for episode in range(600):
    obs = env.reset()
    for step in range(200):
        epsilon = max(1 - episode / 500, 0.01)
        obs, reward, done, info = play_one_step(env, obs, epsilon)
        if done:
            break
    if episode > 50:
        training_step(batch_size)
```

Мы прогнали 600 эпизодов, включающих максимум по 200 шагов. На каждом шаге мы сначала рассчитываем значение `epsilon` для ε-жадной политики: оно будет линейно уменьшаться от 1 до 0.01 за чуть меньше, чем 500 эпизодов. Затем мы вызываем функцию `play_one_step()`, которая будет использовать ε-жадную политику для выбора действия, после чего выполняем его и записываем опыт в буфер воспроизведения. Если эпизод закончен, тогда мы выходим из цикла. Пройдя 50-й эпизод, мы вызываем функцию `training_step()` для обучения модели на пакете, выбранном из буфера воспроизведения. Причина, по которой мы играем 50 эпизодов без обучения, заключается в том, чтобы предоставить буферу воспроизведения какое-то время на заполнение (если не подождать достаточно долго, то в буфере воспроизведения будет не хватать разнообразия). Вот и все, мы только что реализовали алгоритм глубокого Q-обучения!

На рис. 18.10 показаны суммарные награды, которые агент получает в течение каждого эпизода.



*Рис. 18.10. Кривая обучения алгоритма глубокого Q-обучения*

Как видите, алгоритм не добивается заметного прогресса для почти 300 эпизодов (отчасти из-за того, что значение  $\epsilon$  первоначально было очень высоким), а затем его эффективность неожиданно резко подскочила до 200 (максимально возможная эффективность в данной среде). Отличная новость: алгоритм работал хорошо и на самом деле выполнялся гораздо быстрее, чем алгоритм градиентов политики! Но подождите,... всего лишь через несколько эпизодов он забыл все, что знал, и его эффективность упала ниже 25! Это называется *катастрофическим забыванием* и является одной из крупных проблем, с которыми сталкиваются практически все алгоритмы RL: по мере исследования среды алгоритм обновляет свою политику, но то, что он узнал в одной части среды, может привести в негодность то, что он узнал в других частях среды. Опыты довольно-таки связаны и среда обучения продолжает меняться — что не будет идеальным для градиентного спуска! Если увеличить размер буфера воспроизведения, тогда алгоритм станет менее подверженным упомянутой проблеме. Также может помочь понижение скорости обучения. Но правда в том, что обучение с подкреплением является трудным: обучение часто нестабильно, а вам может понадобиться опробовать много значений гиперпараметров и начальных случайных значений, прежде чем вы отыщете комбинацию, которая работает хорошо. Скажем, если в предыдущем примере вы пытаетесь изменить количество нейронов на слой с 32 на 30 или 34, то эффективность никогда не поднимется выше 100 (сеть DQN может быть более стабильной с одним скрытым слоем, а не с двумя).



Обучение с подкреплением является общеизвестно трудным в значительной степени из-за нестабильностей обучения и огромной чувствительности к выбору значений гиперпараметров и начальных случайных значений<sup>13</sup>. Как выразился исследователь Андрей Карпатый: “[Обучение с учителем] жаждет работать. [...] Обучение с подкреплением приходится заставлять работать”. Вам понадобится время, терпение, упорство и возможно также толика везения. Это главная причина, по которой RL не настолько широко принято, как обыкновенное глубокое обучение (например, сверточные сети). Но помимо системы AlphaGo и игр Atari существует несколько реальных приложений: скажем, компания Google применяет RL для оптимизации затрат на свои центры обработки данных, кроме того, RL используется в ряде робототехнических приложений для подстройки гиперпараметров и в системах выдачи рекомендаций.

Вас может интересовать, почему мы не вычерчивали график потери. Оказывается, что потеря — плохой индикатор эффективности модели. Потеря может уменьшаться, но агент может работать хуже (например, подобное происходит, когда агент застрял в одной небольшой области среды и сеть DQN начинает переобучаться в этой области). И наоборот, потеря может увеличиваться, но агент может работать лучше (скажем, если сеть DQN недооценила Q-ценности и начинает корректно увеличивать свои прогнозы, то агент, вероятно, будет работать лучше, получая больше наград, но потеря может расти, поскольку сеть DQN также устанавливает цели, которые тоже будут больше).

Базовый алгоритм глубокого Q-обучения, который мы применяли до сих пор, оказался бы слишком нестабильным, чтобы учиться играть в игры Atari. Так каким же образом исследователи DeepMind задействовали его? Что ж, они модифицировали сам алгоритм!

## Варианты глубокого Q-обучения

Давайте взглянем на несколько вариантов алгоритма глубокого Q-обучения, которые способны стабилизировать и ускорить обучение.

<sup>13</sup> В замечательной публикации 2018 г.а (<https://homl.info/r1hard>) Алекса Ирпана безупречно изложены крупнейшие сложности и ограничения, присущие обучению с подкреплением.

## Фиксированные цели Q-ценности

В базовом алгоритме глубокого Q-обучения модель используется для вырабатывания прогнозов и для установки собственных целей. В итоге может возникнуть ситуация сродни собаке, гоняющейся за собственным хвостом. Такая цепь обратной связи способна сделать сеть нестабильной: она может расходиться, колебаться, замораживаться и т.д. Чтобы решить проблему, исследователи DeepMind в своей статье 2013 года применяли две сети DQN, а не одну: первая была динамической моделью (*online model*), которая обучалась на каждом шаге и использовалась для перемещения агента, а вторая — целевой моделью (*target model*), предназначеннной только для определения целей. Целевая модель являлась просто клоном динамической модели:

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Затем в функции `training_step()` необходимо лишь изменить одну строку, чтобы при расчете Q-ценностей следующих состояний вместо динамической модели задействовать целевую модель:

```
next_Q_values = target.predict(next_states)
```

Наконец, в цикле обучения потребуется копировать веса динамической модели в целевую модель через регулярные интервалы (например, каждые 50 эпизодов):

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Так как целевая модель обновляется гораздо реже динамической модели, цели Q-ценностей более стабильны, упоминаемая ранее цепь обратной связи ослабляется и ее влияние становится менее серьезным. Этот подход был одним из главных достижений исследователей DeepMind в их статье 2013 года, позволив агентам научиться играть в игры Atari на пиксельном уровне. Чтобы стабилизировать обучение, они применяли крошечную скорость обучения 0.00025, обновляли целевую модель только каждые 10 000 шагов (вместо 50, как было в предыдущем примере кода) и использовали очень большой буфер воспроизведения размером 1 миллион опытов. Кроме того, они крайне медленно уменьшали `epsilon`, с 1 до 0.1 за 1 миллион шагов, и позволили алгоритму выполнять в течение 50 миллионов шагов.

Позже в главе мы будем применять библиотеку TF-Agents для обучения агента DQN игре в *Breakout*, используя такие гиперпараметры, но прежде чем заняться этим, давайте взглянем на еще один вариант сети DQN, которому снова удалось превзойти текущее положение дел.

## Двойная глубокая Q-сеть

В статье 2015 года (<https://homl.info/doubledqn>)<sup>14</sup> исследователи DeepMind модифицировали алгоритм DQN, увеличив его эффективность и отчасти стабилизировав обучение. Они назвали такой вариант *Double DQN* (двойная глубокая Q-сеть). Обновление было основано на наблюдении, что целевая сеть склонна к завышению оценок Q-ценности. Действительно, предположим, что все действия одинаково хороши: Q-ценности, оцененные целевой моделью, должны быть идентичными, но поскольку они являются приближениями, то чисто случайно одни могут оказаться больше других. Целевая модель всегда выбирает наибольшую Q-ценность, которая будет чуть выше средней Q-ценности, с высокой вероятностью завышая оценку настоящей Q-ценности (что немного похоже на учет высоты самой высокой случайной волны при измерении глубины бассейна). Чтобы исправить это, исследователи DeepMind предложили при выборе наилучших действий для следующих состояний применять динамическую модель вместо целевой модели, а целевую модель использовать только для оценки Q-ценностей, связанных с наилучшими действиями. Вот обновленная функция `training_step()`:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values =
        (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
        (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # остальной код такой же, как ранее
```

<sup>14</sup> Хадо ван Хасселт и др., *Deep Reinforcement Learning with Double Q-Learning* (Глубокое обучение с подкреплением с помощью двойного Q-обучения), *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015 г.): с. 2094–2100.

Спустя несколько месяцев было предложено еще одно усовершенствование алгоритма DQN.

## Воспроизведение опытов по приоритетам

Почему бы вместо равномерной выборки опытов из буфера воспроизведения ни выбирать важные опыты более часто? Такая идея называется *выборкой по значимости* (*importance sampling — IS*) или *воспроизведением опытов по приоритетам* (*prioritized experience replay — PER*) и была представлена в статье 2015 года (<https://hml.info/prioreplay>)<sup>15</sup>, написанной исследователями DeepMind (снова!).

В частности, опыты считаются “важными”, если они с высокой вероятностью приводят к быстрому прогрессу обучения. Но как это можно оценить? Один рациональный подход предусматривает измерение величины ошибки TD:  $\delta = r + \gamma \cdot V(s') - V(s)$ . Крупная ошибка TD указывает на то, что переход  $(s, r, s')$  является крайне неожиданным и потому вероятно на нем стоит учиться<sup>16</sup>. Когда опыт записывается в буфер воспроизведения, его приоритет устанавливается в очень большое значение, чтобы гарантировать хотя бы однократную его выборку. Однако после того как он выбран (и всякий раз, когда он выбран), вычисляется ошибка TD ( $\delta$ ), а приоритет данного опыта устанавливается в  $p = |\delta|$  (плюс маленькая константа для гарантии того, что каждый опыт имеет ненулевую вероятность быть выбранным). Вероятность  $P$  выборки опыта с приоритетом  $p$  пропорциональна  $p^\zeta$ , где  $\zeta$  представляет собой гиперпараметр, который управляет тем, насколько жаждой должна быть выборка по значимости: когда  $\zeta = 0$ , мы получаем просто равномерную выборку, а когда  $\zeta = 1$  — то полноценную выборку по значимости. Авторы в своей статье применяли  $\zeta = 0.6$ , но оптимальное значение будет зависеть от задачи.

Тем не менее, есть одна загвоздка: поскольку выборки будут смещены в сторону важных опытов, мы обязаны скомпенсировать такое смещение во время обучения, понижая веса опытов согласно их важности, иначе модель просто будет переобучаться важными опытами. Точнее говоря, мы хотим, чтобы важные опыты выбирались чаще, но это также означает, что им пот-

<sup>15</sup> Том Шоль и др., *Prioritized Experience Replay* (Воспроизведение опытов по приоритетам), препринт arXiv:1511.05952 (2015 г.).

<sup>16</sup> Также может оказаться, что награды зашумлены, и в таком случае существуют более удачные методы для оценки важности опыта (примеры ищите в статье).

ребуется назначать более низкие веса во время обучения. Таким образом, мы определяем вес при обучении каждого опыта как  $w = (n P)^{-\beta}$ , где  $n$  — количество опытов в буфере воспроизведения, а  $\beta$  — гиперпараметр, который управляет желательной степенью компенсации смещения при выборке по значимости (0 означает вообще без компенсации, 1 означает полную компенсацию). Авторы в своей статье использовали значение  $\beta = 0.4$  в начале обучения и линейно увеличивали его до  $\beta = 1$  к концу обучения. Опять-таки оптимальное значение будет зависеть от задачи, но если вы увеличиваете одно, то обычно захотите увеличить и другое.

Теперь давайте рассмотрим последний важный вариант алгоритма DQN.

## Соревнующаяся глубокая Q-сеть

Алгоритм соревнующейся глубокой Q-сети (*Dueling DQN* — *DDQN*; не путайте ее с сетью Double DQN, хотя обе методики можно легко комбинировать) был представлен в еще одной статье 2015 года (<https://hml.info/ddqn>)<sup>17</sup> от исследователей DeepMind. Чтобы понять, как он работает, мы сначала должны отметить, что Q-ценность пары “состояние-действие”  $(s, a)$  может быть выражена в виде  $Q(s, a) = V(s) + A(s, a)$ , где  $V(s)$  — ценность состояния  $s$  и  $A(s, a)$  — преимущество выполнения действия  $a$  в состоянии  $s$  по сравнению со всеми остальными возможными действиями в этом состоянии. Кроме того, ценность состояния равна Q-ценности наилучшего действия  $a^*$  для данного состояния (т.к. мы допускаем, что оптимальная политика выберет наилучшее действие) и потому  $V(s) = Q(s, a^*)$ , откуда вытекает, что  $A(s, a^*) = 0$ . В соревнующейся сети DQN модель оценивает ценность состояния и преимущество каждого возможного действия. Поскольку наилучшее действие должно иметь преимущество 0, модель вычитает максимальное спрогнозированное преимущество из всех спрогнозированных преимуществ. Ниже приведена простая модель Dueling DQN, реализованная с применением API-интерфейса Functional:

```
K = keras.backend  
input_states = keras.layers.Input(shape=[4])  
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)  
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)  
state_values = keras.layers.Dense(1)(hidden2)
```

<sup>17</sup> Зию Ванг и др., *Dueling Network Architectures for Deep Reinforcement Learning* (Архитектуры соревнующихся сетей для глубокого обучения с подкреплением), препринт arXiv:1511.06581 (2015 г.).

```

raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1,
                                      keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])

```

Оставшаяся часть алгоритма является той же, что и ранее. На самом деле вы можете построить *двойную соревнующуюся глубокую Q-сеть (Double Dueling DQN)* и скомбинировать ее с воспроизведением опытов по приоритетам! В более общем смысле комбинирование допускают многие методики RL, как исследователи DeepMind продемонстрировали в своей статье 2017 года (<https://homl.info/rainbow>)<sup>18</sup>. Авторы статьи скомбинировали шесть разных методик внутри агента, названного *Rainbow* (*Радуга*), который в значительной степени превзошел положение дел на тот момент.

К сожалению, реализация всех этих методик, их отладка, точная настройка и, конечно же, обучение моделей может требовать огромного объема работы. Таким образом, вместо того, чтобы заново изобретать колесо, часто лучше повторно использовать масштабируемые и хорошо протестированные библиотеки, подобные TF-Agents.

## Библиотека TF-Agents

Библиотека TF-Agents (<https://github.com/tensorflow/agents>) представляет собой основанную на TensorFlow библиотеку для обучения с подкреплением, которая была разработана в Google и в 2018 году сделана проектом с открытым кодом. Подобно OpenAI Gym она предоставляет много готовых сред (включая оболочки для всех сред OpenAI Gym) плюс поддерживает библиотеку PyBullet (для симуляции трехмерных физических сущностей), библиотеку DM Control от DeepMind (основанную на механизме физических сущностей MuJoCo) и библиотеку ML-Agents от Unity (эмulatingую многочисленные трехмерные среды). Вдобавок она реализует множество алгоритмов RL, в том числе REINFORCE, DQN и DDQN, а также разнообразные компоненты RL вроде эффективных буферов воспроизведения и метрик. Библиотека TF-Agents является быстрой, масштабируемой, легкой в применении и настраиваемой: вы можете создавать собственные среды и

<sup>18</sup> Маттео Хессель и др., *Rainbow: Combining Improvements in Deep Reinforcement Learning* (*Rainbow: объединение улучшений в глубоком обучении с подкреплением*), препринт arXiv:1710.02298 (2017 г.).

нейронные сети и настраивать почти все компоненты. В настоящем разделе мы будем использовать TF-Agents для обучения агента играть в *Breakout*, знаменитую игру Atari (рис. 18.11<sup>19</sup>), с применением алгоритма DQN (при желании вы можете легко переключиться на другой алгоритм).

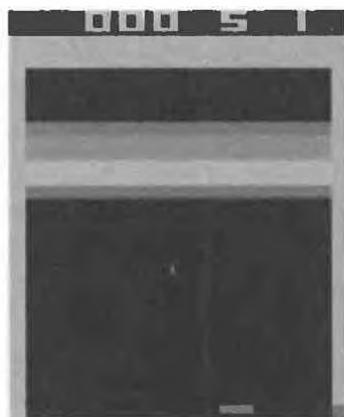


Рис. 18.11. Знаменитая игра *Breakout*

## Установка TF-Agents

Мы начнем с установки библиотеки TF-Agents. Для этого можно использовать pip (как всегда, если вы работаете с виртуальной средой, то сначала активизируйте ее; если же нет, тогда необходимо указать параметр `--user` или располагать правами администратора):

```
$ python3 -m pip install --upgrade tf-agents
```



На время написания главы библиотека TF-Agents библиотека по-прежнему была довольно новой и ежедневно совершенствовалась, так что к моменту чтения вами книги API-интерфейс может слегка измениться. Однако общая картина должна остаться той же самой, равно как и большинство кода. Если что-то нарушит его работу, то я соответствующим образом обновлю тетрадь Jupyter, а потому следите за ней.

<sup>19</sup> Если вы не знакомы с игрой Breakout, то знайте, что она проста: шарик отскакивает и разбивает кирпичи, когда ударяется об них. Вы управляете пластиной вблизи нижней части экрана. Пластина может перемещаться влево или вправо, а вы должны заставить шарик разбить все кирпичи, одновременно предотвращая его касание с нижней границей экрана.

Далее мы создадим среду TF-Agents, которая будет просто оболочкой среды Breakout из OpenAI Gym. Для этого сначала придется установить зависимости Atari библиотеки OpenAI Gym:

```
$ python3 -m pip install --upgrade 'gym[atari]'
```

Среди прочих библиотек приведенная выше команда установит atari-py, т.е. интерфейс Python для игровой среды обучения (*Arcade Learning Environment — ALE*), которая представляет собой фреймворк, построенный поверх эмулятора Atari 2600 под названием Stella.

## Среды TF-Agents

Если все прошло успешно, тогда у вас должна появиться возможность импортирования TF-Agents и создания среды Breakout:

```
>>> from                                     import suite_gym
>>> env = suite_gym.load("Breakout-v4")
>>> env
<tf_agents.environments.wrappers.TimeLimit at 0x10c523c18>
```

Это просто оболочка вокруг среды OpenAI Gym, к которой можно обращаться через атрибут gym:

```
>>> env.gym
<gym.envs.atari.atari_env.AtariEnv at 0x24dcab940>
```

Среды TF-Agents очень похожи на среды OpenAI Gym, но есть несколько отличий. Прежде всего, метод reset() не возвращает наблюдение; взамен он возвращает объект TimeStep, который умеет в себе наблюдение, а также дополнительную информацию:

```
>>> env.reset()
TimeStep(step_type=array(0, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[0., 0., 0.], [0., 0., 0.], ...]],  
dtype=float32))
```

Метод step() тоже возвращает объект TimeStep:

```
>>> env.step(1)    # запуск
TimeStep(step_type=array(1, dtype=int32),
         reward=array(0., dtype=float32),
         discount=array(1., dtype=float32),
         observation=array([[0., 0., 0.], [0., 0., 0.], ...]], dtype=float32))
```

Атрибуты `reward` и `observation` объяснений не требуют; они такие же, как для OpenAI Gym (за исключением того, что `reward` представлен в виде массива NumPy). Атрибут `step_type` равен 0 для первого временного шага в эпизоде, 1 для промежуточных временных шагов и 2 для финального временного шага. Чтобы выяснить, является ли временной шаг финальным, вы можете вызвать его метод `is_last()`. Наконец, атрибут `discount` указывает коэффициент дисконтирования для применения на данном временном шаге. В рассматриваемом примере он равен 1, так что дисконтирования вообще не будет. Вы можете определить коэффициент дисконтирования, устанавливая параметр `discount` при загрузке среды.



Вы можете в любой момент получить доступ к текущему временному шагу среды посредством вызова ее метода `current_time_step()`.

## Спецификации среды

Удобно то, что среда TF-Agents предоставляет спецификации наблюдений, действий и временных шагов, включая их формы, типы данных и имена, а также их минимальные и максимальные значения:

```
>>> env.observation_spec()
BoundedArraySpec(shape=(210, 160, 3), dtype=dtype('float32'), name=None,
                 minimum=[[0. 0. 0.], [0. 0. 0.], ...],
                 maximum=[[255., 255., 255.], [255., 255., 255.], ...])
>>> env.action_spec()
BoundedArraySpec(shape=(), dtype=dtype('int64'), name=None,
                  minimum=0, maximum=3)
>>> env.time_step_spec()
TimeStep(step_type=ArraySpec(shape=(), dtype=dtype('int32'),
                             name='step_type'),
         reward=ArraySpec(shape=(), dtype=dtype('float32'),
                          name='reward'),
         discount=BoundedArraySpec(shape=(), ...,
                                    minimum=0.0, maximum=1.0),
         observation=BoundedArraySpec(shape=(210, 160, 3), ...))
```

Как видите, наблюдения — это просто снимки экрана Atari, представленные в виде массивов NumPy формы [210, 160, 3]. Чтобы визуализировать среду, вы можете вызвать `env.render(mode="human")`, а если хотите получить обратно изображение в форме массива NumPy, тогда просто вызо-

вите `env.render(mode="rgb_array")` (в отличие от OpenAI Gym такой режим является стандартным).

Доступны четыре действия. Среды Atari из Gym имеют дополнительный метод, который можно вызвать, чтобы узнать, чему соответствует каждое действие:

```
>>> env.gym.get_action_meanings()  
['NOOP', 'FIRE', 'RIGHT', 'LEFT']
```



Спецификации могут быть экземплярами класса спецификаций, вложенными списками или словарями спецификаций. Если спецификация вложенная, тогда указанный объект должен соответствовать вложенной структуре спецификации. Скажем, если спецификацией наблюдения является `{"sensors": ArraySpec(shape=[2]), "camera": ArraySpec(shape=[100, 100])}`, то допустимым наблюдением будет `{"sensors": np.array([1.5, 3.5]), "camera": np.array(...)}`. Пакет `tf.nest` предлагает инструменты для обработки таких вложенных структур (также называемых *гнездами* (*nest*)).

Наблюдения довольно крупные, а потому мы понизим их дискретизацию и преобразуем в полутоновые, что ускорит обучение и уменьшит расход ОЗУ. Для этого мы можем использовать оболочку среды (*environment wrapper*).

## Оболочки сред и предварительная обработка Atari

Библиотека TF-Agents предлагает в пакете `tf_agents.environments.wrappers` несколько оболочек сред. Как и следует из их названия, они содержат внутри себя среду, направляя ей все вызовы, но также добавляют дополнительную функциональность. Ниже кратко описаны некоторые доступные оболочки.

### ActionClipWrapper

Вырезает действия в спецификацию действий.

### ActionDiscretizeWrapper

Квантует непрерывное пространство действий в дискретное пространство действий. Например, если пространство действий исходной среды представляет собой непрерывный диапазон от  $-1.0$  до  $+1.0$ , но вы хотите применять алгоритм, который поддерживает только диск-

ретные пространства действий, такой как DQN, тогда оберните среду, используя `discrete_env = ActionDiscretizeWrapper(env, num_actions=5)`, и новая оболочка `discrete_env` будет иметь дискретное пространство действий: 0, 1, 2, 3, 4. Указанные действия соответствуют действиям -1.0, -0.5, 0.0, 0.5 и 1.0 в исходной среде.

### ActionRepeat

Повторяет каждое действие в течение *n* шагов, одновременно накапливая награды. Во многих средах это помогает значительно ускорить обучение.

### RunStats

Записывает статистические данные о среде, такие как количество шагов и число эпизодов.

### TimeLimit

Прерывает работу среды, если она выполняется дольше максимального количества шагов.

### VideoWrapper

Записывает видеоролик среды.

Чтобы создать обернутую среду, потребуется создать оболочку, передав конструктору обернутую среду. Вот и все! Скажем, приведенный далее код поместит нашу среду в оболочку `ActionRepeat`, так что каждое действие будет повторяться четыре раза:

```
from                                     import ActionRepeat
repeating_env = ActionRepeat(env, times=4)
```

Библиотека OpenAI Gym располагает собственными оболочками сред в пакете `gym.wrappers`. Тем не менее, они предназначены для обворачивания сред Gym, но не сред TF-Agents, поэтому для их применения первым делом понадобится поместить среду Gym внутрь оболочки Gym и затем обернуть результатирующую среду с помощью оболочки TF-Agents. Такую работу выполнит функция `suite_gym.wrap_env()` при условии, что ей предоставятся среда Gym, а также список оболочек Gym и/или список оболочек TF-Agents. В качестве альтернативы функция `suite_gym.load()` создаст среду Gym и обернет ее, если ей передать ряд оболочек. Каждая оболочка будет

создаваться без аргументов, так что в случае, когда необходимо установить какие-то аргументы, потребуется передать `lambda`. Например, следующий код создает среду Breakout, которая выполняется максимум для 10 000 шагов в течение каждого эпизода, а каждое действие повторяется четыре раза:

```
from           import TimeLimit
limited_repeating_env = suite_gym.load(
    "Breakout-v4",
    gym_env_wrappers=[lambda env: TimeLimit(env,
                                                max_episode_steps=10000)],
    env_wrappers=[lambda env: ActionRepeat(env, times=4)])
```

В большинстве статей, где используются среды Atari, применяются стандартные шаги предварительной обработки, поэтому библиотека TF-Agents предоставляет удобную оболочку `AtariPreprocessing`, которая реализует их. Ниже представлен список поддерживаемых шагов предварительной обработки.

### *Преобразование в шкалу полутононов и понижение дискретизации*

Наблюдения преобразуются в шкалу полутононов и подвергаются понижению дискретизации (по умолчанию до  $84 \times 84$  пикселей).

### *Объединение по максимуму*

Последние два кадра игры объединяются по максимуму с использованием фильтра  $1 \times 1$ . Это делается для устранения мерцания, происходящего в ряде игр Atari из-за ограниченного количества спрайтов, которые эмулятор Atari 2600 способен отображать в каждом кадре.

### *Пропуск кадров*

Агент видит только каждые  $n$  кадров игры (по умолчанию  $n = 4$ ), а его действия повторяются для каждого кадра с накоплением всех наград. Это фактически ускоряет игру с точки зрения агента и также ускоряет обучение, потому что награды меньше задерживаются.

### *Конец при утрате жизни*

В некоторых играх награды основаны лишь на счете очков, так что агент не получает безотлагательного штрафа за утрату жизни. Одно из решений предусматривает немедленное окончание игры всякий раз, когда утрачивается жизнь. О действительных преимуществах такой стратегии ведутся споры, поэтому по умолчанию она отключена.

Поскольку в стандартной среде Atari уже применяются случайный пропуск кадров и объединение по максимуму, нам понадобится загрузить необработанный вариант без пропуска по имени "BreakoutNoFrameskip-v4". Кроме того, одиночного кадра из игры *Breakout* будет недостаточно, чтобы знать направление и скорость движения шарика, что крайне затруднит для агента проведение игры надлежащим образом (если только он не является агентом RNN, который предохраняет внутреннее состояние между шагами). Один из способов справиться с этим заключается в использовании оболочки среды, которая будет выдавать наблюдения, состоящие из уложенных друг на друга вдоль измерения каналов множества кадров. Такая стратегия реализуется оболочкой FrameStack4, возвращающей стопки из четырех кадров. Давайте создадим обернутую среду Atari!

```
from import suite_atari
from
import AtariPreprocessing
from import FrameStack4

max_episode_steps = 27000 # <=> 108 тысяч кадров ALE,
# т.к. 1 шаг = 4 кадра
environment_name = "BreakoutNoFrameskip-v4"

env = suite_atari.load(
    environment_name,
    max_episode_steps=max_episode_steps,
    gym_env_wrappers=[AtariPreprocessing, FrameStack4])
```

Результат всей предварительной обработки показан на рис. 18.12. Вы заметите, что разрешение стало намного меньшим, но достаточным, чтобы играть в игру. В добавок кадры уложены стопкой вдоль измерения каналов, так что красный представляет кадр из трех шагов тому назад, зеленый — из двух шагов тому назад, синий является предыдущим кадром, а розовый — текущим кадром<sup>20</sup>. Из этого одиночного наблюдения агент может видеть, что шарик перемещается в направлении нижнего левого угла, и агент должен продолжать передвигать пластины влево (как он поступал на предшествующих шагах).

<sup>20</sup> Так как есть только три основных цвета, просто отобразить изображение с четырьмя цветовыми каналами не удастся. По указанной причине, чтобы получить представленное здесь изображение RGB, для последнего канала я смешал первые три. На самом деле розовый является смесью синего и красного, но агент видит четыре независимых канала.



*Рис. 18.12. Предварительно обработанное наблюдение Breakout*

В заключение мы можем поместить среду внутрь TFPyEnvironment:

```
from
    import TFPyEnvironment
tf_env = TFPyEnvironment(env)
```

Это сделает среду пригодной к использованию изнутри графа TensorFlow (“за кулисами” класс TFPyEnvironment полагается на функцию `tf.py_function()`, что позволяет графу вызывать произвольный код Python). Благодаря классу TFPyEnvironment библиотека TF-Agents поддерживает среды чисто Python и среды, основанные на TensorFlow. В более общем смысле библиотека TF-Agents поддерживает и предоставляет компоненты чисто Python и компоненты, основанные на TensorFlow (агенты, буферы воспроизведения, метрики и т.д.).

Теперь, имея подходящую среду Breakout со всей надлежащей предварительной обработкой и поддержкой TensorFlow, мы должны создать агента DQN и другие компоненты, которые будут нужны для его обучения. Давайте рассмотрим архитектуру системы, которую мы построим.

## Структура обучения

Обучающая программа TF-Agents обычно разделена на две части, которые выполняются параллельно, как показано на рис. 18.13: слева драйвер исследует среду с применением политики сбора для выбора действий и собирает траектории (т.е. опыты), отправляя их наблюдателю, который сохраняет их

в буфере воспроизведения; справа агент извлекает пакеты траекторий из буфера воспроизведения и обучает ряд сетей, которыми пользуется политика сбора. Выражаясь кратко, левая часть исследует среду и собирает траектории, тогда как правая часть узнает и обновляет политику сбора.

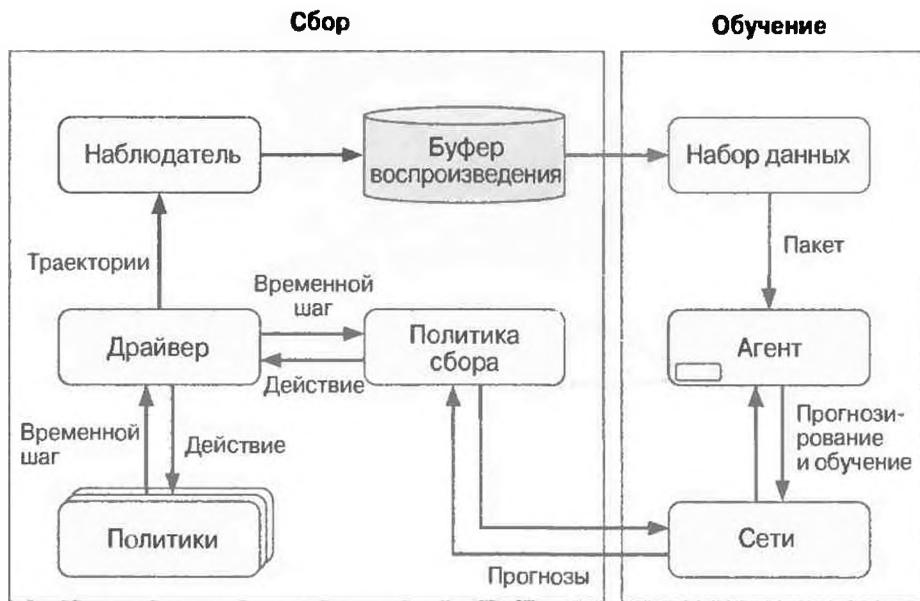


Рис. 18.13. Типовая структура обучения в TF-Agents

После просмотра рис. 18.13 неминуемо возникнет несколько вопросов, на которые я постараюсь ответить ниже.

- Почему существует множество сред? Вместо исследования единственной среды обычно вы хотите, чтобы драйвер прозондировал несколько копий среды параллельно, извлекая преимущества из мощи всех имеющихся ядер центрального процессора, поддерживая занятость обучающих графических процессоров и предоставляя алгоритму обучения менее связанные траектории.
- Что такое *траектория*? Это сжатое представление *перехода* от одного временного шага к следующему, или последовательность следующих друг за другом переходов от временного шага  $t$  к временному шагу  $n+t$ . Траектории, собранные драйвером, передаются наблюдателю, который сохраняет их в буфере воспроизведения, и позже они выбираются агентом и применяются для обучения.

- Почему нам нужен наблюдатель? Может ли драйвер сохранять траектории напрямую? На самом деле он мог бы, но тогда архитектура стала бы менее гибкой. Скажем, что, если вы не хотите использовать буфер воспроизведения? Что, если вы желаете применять траектории для чего-то еще, вроде расчета метрик? Фактически наблюдатель представляет собой всего лишь функцией, которая принимает траекторию как аргумент. Вы можете использовать наблюдатель для сохранения траекторий в буфер воспроизведения, для их записи в файл TFRecord (см. главу 13), для вычисления метрик или для чего-нибудь другого. Более того, вы можете передать драйверу множество наблюдателей, и он будет ретранслировать траектории всем наблюдателям.



Хотя приведенная на рис. 18.13 структура является самой распространенной, вы можете настраивать ее по своему усмотрению и даже заменять отдельные компонентами собственными. В сущности, если только вы не занимаетесь исследованием новых алгоритмов RL, тогда почти наверняка захотите применять специальную среду для своей задачи. Вам необходимо только создать специальный класс, унаследованный от класса `PyEnvironment` из пакета `tf_agents.environments.py_environment`, и переопределить соответствующие методы, такие как `action_spec()`, `observation_spec()`, `_reset()` и `_step()` (пример ищите в разделе “Creating a Custom TF-Agents Environment” (Создание специальной среды TF-Agents) тетради Jupiter для настоящей главы).

Теперь мы создадим все компоненты структуры: сначала глубокую Q-сеть, затем агент DQN (который позаботится о создании политики сбора), далее буфер воспроизведения и наблюдатель для записи в него, затем несколько метрик обучения, далее драйвер и в заключение набор данных. После того как все компоненты окажутся на месте, мы заполним буфер воспроизведения рядом начальных траекторий и запустим главный цикл обучения. Итак, давайте начнем с создания глубокой Q-сети.

## Создание глубокой Q-сети

Библиотека TF-Agents предоставляет много сетей в пакете `tf_agents.networks` и его подпакетах. Мы будем использовать класс `tf_agents.networks.q_network.QNetwork`:

```

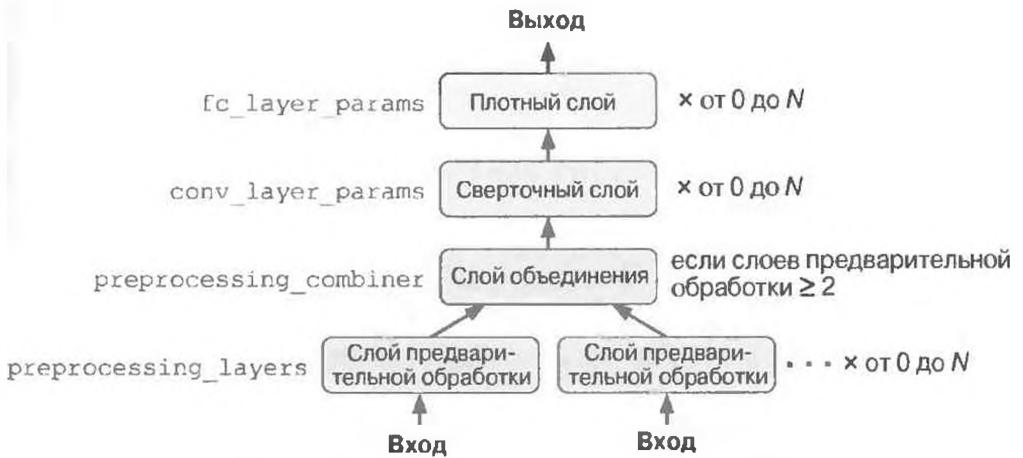
from import QNetwork
preprocessing_layer = keras.layers.Lambda(
    lambda obs: tf.cast(obs, np.float32) / 255.)
conv_layer_params=[(32, (8, 8), 4), (64, (4, 4), 2), (64, (3, 3), 1)]
fc_layer_params=[512]
q_net = QNetwork(
    tf_env.observation_spec(),
    tf_env.action_spec(),
    preprocessing_layers=preprocessing_layer,
    conv_layer_params=conv_layer_params,
    fc_layer_params=fc_layer_params)

```

Класс `QNetwork` принимает в качестве входа наблюдение и выдает по одной Q-ценности на действие, поэтому мы должны предоставить ему спецификации наблюдений и действий. Он начинается со слоя предварительной обработки: простого слоя `Lambda`, который приводит наблюдения к 32-битным числам с плавающей точкой и нормализует их (значения будут находиться в диапазоне от 0.0 до 1.0). Наблюдения содержат беззнаковые байты, занимающие в 4 раза меньше пространства, чем 32-битные числа с плавающей точкой — вот почему мы не приводили наблюдения к 32-битным числам с плавающей точкой ранее; мы хотим сэкономить ОЗУ в буфере воспроизведения. Далее класс `QNetwork` применяет три сверточных слоя: первый имеет 32 фильтра  $8 \times 8$  и использует страйд 4, второй — 64 фильтра  $4 \times 4$  и страйд 2, а третий — 64 фильтра  $3 \times 3$  и страйд 1. Наконец, он применяет плотный слой с 512 элементами, за которым следует выходной плотный слой с 4 элементами, по одному на Q-ценность, подлежащую выдаче (т.е. по одному на действие). Все сверточные и все плотные слои кроме выходного по умолчанию используют функцию активации `ReLU` (вы можете выбрать другую функцию, установив аргумент `activation_fn`). В выходном слое функция активации отсутствует.

“За кулисами” класс `QNetwork` состоит из двух частей: кодирующей сети, обрабатывающей наблюдения, за которой следует выходной плотный слой, выдающий по одной Q-ценности на действие. Класс `EncodingNetwork` из библиотеки `TF-Agent` реализует архитектуру нейронной сети, которую можно обнаружить в разнообразных агентах (рис. 18.14).

Входов может быть один или большее количество. Например, если наблюдение состоит из данных каких-то датчиков плюс изображений из камеры, тогда вы будете иметь два входа.



*Рис. 18.14. Архитектура кодирующей сети*

Каждый вход может требовать ряда шагов предварительной обработки, в случае чего вы можете указать через аргумент `preprocessing_layers` список слоев Keras, с одним слоем предварительной обработки на вход, и сеть будет применять каждый слой к соответствующему входу (если входу необходимо множество слоев предварительной обработки, то разрешено передавать целую модель, т.к. модель Keras всегда можно использовать в качестве слоя). Если входов два или больше, тогда вам придется также передавать через аргумент `preprocessing_combiner` дополнительный слой для объединения выходов из слоев предварительной обработки в единственный выход.

Кодирующая сеть будет необязательно последовательно применять список свертоек при условии, что вы передали их параметры через аргумент `conv_layer_params`. Параметры должны выглядеть как список из трех кортежей (по одному на сверточный слой), указывающих количество фильтров, размер ядра и страйд. После этих сверточных слоев кодирующая сеть будет необязательно применять последовательность плотных слоев, если вы установили аргумент `fc_layer_params`: он должен быть списком, содержащим количество нейронов для каждого плотного слоя. Дополнительно вы можете передать список солями отключения (по одной на плотный слой) через аргумент `dropout_layer_params`, если хотите применять отключение после каждого плотного слоя. Класс `QNetwork` получает выход кодирующей сети и передает его выходному плотному слою (с одним элементом на действие).



Класс `QNetwork` обладает достаточной гибкостью, чтобы строить множество разных архитектур, но вы всегда можете создать собственный класс сети, если нуждаетесь в большей гибкости: расширьте класс `tf_agents.networks.Network` и реализуйте его подобно обычному специальному слою Keras. Класс `tf_agents.networks.Network` является подклассом класса `keras.layers.Layer`, который добавляет функциональность, требуемую рядом агентов, такую как возможность легкого создания неглубоких копий сети (т.е. копирования архитектуры сети, но не ее весов). Например, класс `DQNAgent` использует это для создания копии динамической модели.

Теперь, располагая сетью DQN, мы готовы к построению агента DQN.

## Создание агента DQN

В библиотеке TF-Agents реализованы многие типы агентов, находящиеся в пакете `tf_agents.agents` и его подпакетах. Мы будем применять класс `tf_agents.agents.dqn.dqn_agent.DqnAgent`:

```
from                                     import DqnAgent
train_step = tf.Variable(0)
update_period = 4    # обучать модель каждые 4 шага
optimizer = keras.optimizers.RMSprop(lr=2.5e-4, rho=0.95,
                                      momentum=0.0,
                                      epsilon=0.00001, centered=True)
epsilon_fn = keras.optimizers.schedules.PolynomialDecay(
    initial_learning_rate=1.0,           # начальное значение ε
    decay_steps=250000 // update_period, # <=> 1,000,000 кадров ALE
    end_learning_rate=0.01)             # конечное значение ε
agent = DqnAgent(tf_env.time_step_spec(),
                  tf_env.action_spec(),
                  q_network=q_net,
                  optimizer=optimizer,
                  target_update_period=2000,   # <=> 32 000 кадров ALE
                  td_errors_loss_fn=
                      keras.losses.Huber(reduction="none"),
                  gamma=0.99,                 # коэффициент дисконтирования
                  train_step_counter=train_step,
                  - epsilon_greedy=
                      lambda: epsilon_fn(train_step))
agent.initialize()
```

Давайте пройдемся по коду.

- Сначала мы создаем переменную, которая будет подсчитывать количество шагов обучения.
- Затем мы строим оптимизатор, используя те же самые гиперпараметры, как в статье 2015 года о сети DQN.
- Далее мы создаем объект `PolynomialDecay`, который будет вычислять значение  $\epsilon$  для  $\epsilon$ -жадной политики сбора, имея текущий шаг обучения (обычно оно применяется для снижения скорости обучения, отсюда и имена аргументов, но в той же степени хорошо подходит для снижения любой другой величины). Значение  $\epsilon$  будет уменьшаться с 1.0 до 0.01 (значение, используемое в статье 2015 года о сети DQN) в 1 миллионе кадров ALE, что соответствует 250 000 шагов, т.к. мы применяем пропуск кадров с периодом 4. Кроме того, мы будем обучать агент каждые 4 шага (т.е. 16 кадров ALE), потому  $\epsilon$  фактически снижается в течение 62 500 шагов обучения.
- Затем мы строим объект `DQNAgent`, передавая ему спецификации временных шагов и действий, сеть `QNetwork` для обучения, оптимизатор, количество шагов обучения между обновлениями целевой модели, используемую функцию потерь, коэффициент дисконтирования, переменную `train_step` и функцию, которая возвращает значение  $\epsilon$  (она не должна принимать какие-либо аргументы, поэтому для передачи `train_step` потребуется `lambda`).
- Обратите внимание, что функция потерь обязана возвращать ошибку на образец, не среднюю ошибку, а потому мы устанавливаем `reduction= "none"`.
- В заключение мы инициализируем агент.

Давайте создадим буфер воспроизведения и наблюдатель, который будет в него записывать.

## Создание буфера воспроизведения и соответствующего наблюдателя

Библиотека TF-Agents предлагает в своем пакете `tf_agents.replay_buffers` разнообразные реализации буферов воспроизведения. Некоторые из них написаны на чистом Python (имена их модулей начинаются с `py_`), а другие основаны на TensorFlow (имена их модулей начинаются с `tf_`). Мы

будем применять класс `TFUniformReplayBuffer` из пакета `tf_agents.replay_buffers.tf_uniform_replay_buffer`. Он представляет высокопроизводительную реализацию буфера воспроизведения с равномерной выборкой<sup>21</sup>:

```
from import tf_uniform_replay_buffer  
replay_buffer = tf_uniform_replay_buffer.TFUniformReplayBuffer(  
    data_spec=agent.collect_data_spec,  
    batch_size=tf_env.batch_size,  
    max_length=1000000)
```

Рассмотрим каждый аргумент по отдельности.

#### `data_spec`

Спецификация данных, которые будут сохраняться в буфере воспроизведения. Агенту DQN известно, как будут выглядеть собираемые данные, и он делает спецификацию данных доступной через свой атрибут `collect_data_spec`, так что именно это мы предоставляем буферу воспроизведения.

#### `batch_size`

Количество траекторий, добавляемых на каждом шаге. В нашем случае количество равно 1, поскольку драйвер просто будет запускать по одному действию на шаг, а также собирать одну траекторию. Если бы среда была *пакетной*, т.е. средой, которая на каждом шаге принимает пакет действий и возвращает пакет наблюдений, тогда драйверу пришлось бы на каждом шаге сохранять пакет траекторий. Поскольку мы используем буфер воспроизведения TensorFlow, он должен знать размер пакетов, которые будет обрабатывать (для построения вычислительного графа). Примером пакетной среды является `ParallelPyEnvironment` (из пакета `tf_agents.environments.parallel_py_environment`): он запускает множество сред параллельно в отдельных процессах (они могут быть разными до тех пор, пока имеют те же самые спецификации действий и наблюдений) и на каждом шаге принимает пакет действий и выполняет их в средах (одно действие на среду), после чего возвращает все результирующие наблюдения.

<sup>21</sup> На момент написания главы пока не было буферов воспроизведения опытов по приоритетам, но с высокой вероятностью они вскоре появятся.

## max\_length

Максимальный размер буфера воспроизведения. Мы создаем большой буфер воспроизведения, который может хранить 1 миллион траекторий (как делалось в статье 2015 года о сети DQN). Он потребует много памяти.



Когда мы сохраняем две последовательных траектории, они содержат два последовательных наблюдения, каждое с четырьмя кадрами (т.к. мы применяем оболочку FrameStack4), но, к сожалению, три из четырех кадров во втором наблюдении избыточны (они уже присутствуют в первом наблюдении). Другими словами, мы расходуем приблизительно в четыре раза больше ОЗУ, чем необходимо. Чтобы избежать этого, взамен можно использовать класс PyHashedReplayBuffer из пакета `tf_agents.replay_buffers.py_hashed_replay_buffer`: он устраниет дублирование данных в сохраненных траекториях вдоль последней оси наблюдений.

Теперь можно создать наблюдатель, который будет записывать траектории в буфер воспроизведения. Наблюдатель представляет собой просто функцию (или вызываемый объект), принимающую аргумент с траекторией, так что мы можем напрямую применять в качестве наблюдателя метод `add_method()`, привязанный к объекту `replay_buffer`:

```
replay_buffer_observer = replay_buffer.add_batch
```

Если вы хотите создать собственный наблюдатель, то могли бы написать любую функцию с аргументом `trajectory`. Если он должен иметь состояние, тогда вы можете создать класс с методом `__call__(self, trajectory)`. Скажем, ниже показан простой наблюдатель, который будет инкрементировать счетчик при каждом вызове (за исключением ситуации, когда траектория представляет границу между двумя эпизодами, что не считается шагом), и через каждые 100 инкрементов он отображает прогресс до заданного итога (возврат каретки \r вместе с `end=""` обеспечивает отображение счетчика в той же строке):

```
class Counter:
    def __init__(self, total):
        self.counter = 0
        self.total = total
```

```
def __call__(self, trajectory):
    if not trajectory.is_boundary():
        self.counter += 1
    if self.counter % 100 == 0:
        print("\r{} / {}".format(self.counter, self.total), end="")
```

А теперь создадим несколько метрик обучения.

## Создание метрик обучения

В библиотеке TF-Agents реализовано несколько метрик RL в пакете `tf_agents.metrics`, часть чисто на Python и часть на основе TensorFlow. Давайте создадим ряд метрик для подсчета количества эпизодов, числа выполненных шагов и что еще более важно — средней отдачи на эпизод и средней длины эпизода:

```
from import tf_metrics

train_metrics = [
    tf_metrics.NumberOfEpisodes(),
    tf_metrics.EnvironmentSteps(),
    tf_metrics.AverageReturnMetric(),
    tf_metrics.AverageEpisodeLengthMetric(),
]
```



Дисконтирование наград имеет смысл при обучении и для реализации политики, т.к. оно делает возможным соблюдение баланса между важностью непосредственных наград и будущих наград. Однако после того как эпизод закончился, мы можем оценить, насколько хорошим он был в целом, суммируя недисконтированные награды. По указанной причине класс `AverageReturnMetric` вычисляет сумму недисконтированных наград для каждого эпизода и отслеживает потоковое среднее таких сумм по всем эпизодам, которые встречает.

Вы можете получить значение каждой метрики в любое время, вызвав ее метод `result()` (например, `train_metrics[0].result()`). В качестве альтернативы вы можете вывести все метрики посредством вызова `log_metrics(train_metrics)` (эта функция находится в пакете `tf_agents.eval.metric_utils`):

```
>>> from import log_metrics
>>> import logging
```

```
>>> logging.getLogger().setLevel(logging.INFO)
>>> log_metrics(train_metrics)
[...]
NumberOfEpisodes = 0
EnvironmentSteps = 0
AverageReturn = 0.0
AverageEpisodeLength = 0.0
```

Следующим мы создадим драйвер сбора.

## Создание драйвера сбора

Благодаря рис. 18.13 выяснилось, что драйвер — это объект, который исследует среду с использованием заданной политики, собирает опыты и ретранслирует их наблюдателям. Вот что происходит на каждом шаге:

- драйвер передает текущий временной шаг политике сбора, которая применяет данный временной шаг для выбора действия и возвращает объект *шага действия* (*action step*), содержащий действие;
- затем драйвер передает действие среде, которая возвращает следующий временной шаг;
- наконец, драйвер создает объект траектории для представления этого перехода и ретранслирует его всем наблюдателям.

Некоторые политики, такие как политики RNN, обеспечивают запоминание состояния: они выбирают действие на основе заданного временного шага и собственного внутреннего состояния. Политики с запоминанием состояния возвращают в шаге действия собственное состояние вместе с выбранным действием. На следующем временном шаге драйвер передаст это состояние обратно политике. Более того, драйвер сохраняет состояние политики внутри траектории (в поле *policy\_info*), так что в итоге оно оказывается в буфере воспроизведения. Такое решение жизненно важно при обучении политики с запоминанием состояния: когда агент делает выборку траектории, он обязан установить состояние политики в состояние, в котором она пребывала на момент выбранного временного шага.

Кроме того, как обсуждалось ранее, среда может быть пакетной, и тогда драйвер передает политике *пакетный временной шаг* (т.е. объект временного шага, содержащий пакет наблюдений, пакет типов шагов, пакет наград и пакет дисконтированных наград; все четыре пакета имеют один и тот же размер). Драйвер также передает пакет предшествующий состояний политики.

Затем политика возвращает *пакетный шаг действия*, содержащий пакет действий и пакет состояний политики. В заключение драйвер создает *пакетную траекторию* (т.е. траекторию, которая содержит пакет типов шагов, пакет наблюдений, пакет действий, пакет наград и в более общем случае пакет для каждого атрибута траектории; все пакеты имеют один и тот же размер).

Существуют два главных класса драйверов: `DynamicStepDriver` и `DynamicEpisodeDriver`. Первый собирает опыты для заданного количества шагов, а второй — для заданного количества эпизодов. Мы хотим собирать опыты для четырех шагов для каждой итерации обучения (как было сделано в статье 2015 года о сети DQN), поэтому создадим объект `DynamicStepDriver`:

```
from
    import DynamicStepDriver

collect_driver = DynamicStepDriver(
    tf_env,
    agent.collect_policy,
    observers=[replay_buffer_observer] + training_metrics,
    num_steps=update_period)    # собирать 4 шага для каждой
                                # итерации обучения
```

Мы передаем ему среду для игры, политику сбора агента, список наблюдателей (включая наблюдатель буфера воспроизведения и метрики обучения) и количество шагов, подлежащих выполнению (в данном случае четыре). Сейчас мы могли бы запустить его, вызвав метод `run()`, но буфер воспроизведения лучше разогреть с помощью опытов, собранных с использованием чисто случайной политики. Для этого мы можем применить класс `RandomTFPolicy` и создать второй драйвер, который будет запускать политику для 20000 шагов (что эквивалентно 80000 кадров симулятора, как делалось в статье 2015 года о сети DQN). Мы можем использовать наш наблюдатель `ShowProgress` для отображения прогресса:

```
from                                     import RandomTFPolicy
initial_collect_policy = RandomTFPolicy(tf_env.time_step_spec(),
                                         tf_env.action_spec())
init_driver = DynamicStepDriver(
    tf_env,
    initial_collect_policy,
    observers=[replay_buffer.add_batch, ShowProgress(20000)],
    num_steps=20000)    # <=> 80 000 кадров ALE
final_time_step, final_policy_state = init_driver.run()
```

Мы почти готовы к запуску цикла обучения! Нам лишь необходим последний компонент: набор данных.

## Создание набора данных

Для выборки пакета траекторий из буфера воспроизведения вызовите его метод `get_next()`. Он возвратит пакет траекторий плюс объект `BufferInfo`, который содержит идентификаторы выборки и вероятности их выборки (может оказаться полезным для ряда алгоритмов, таких как PER). Например, следующий код произведет выборку небольшого пакета из двух траекторий (подэпизодов), каждый из которых содержит три последовательных шага. Такие подэпизоды показаны на рис. 18.15 (каждый содержит три последовательных шага из эпизода):

```
>>> trajectories, buffer_info = replay_buffer.get_next(  
...     sample_batch_size=2, num_steps=3)  
...  
>>> trajectories._fields  
('step_type', 'observation', 'action', 'policy_info',  
 'next_step_type', 'reward', 'discount')  
>>> trajectories.observation.shape  
TensorShape([2, 3, 84, 84, 4])  
>>> trajectories.step_type.numpy()  
array([[1, 1, 1],  
       [1, 1, 1]], dtype=int32)
```

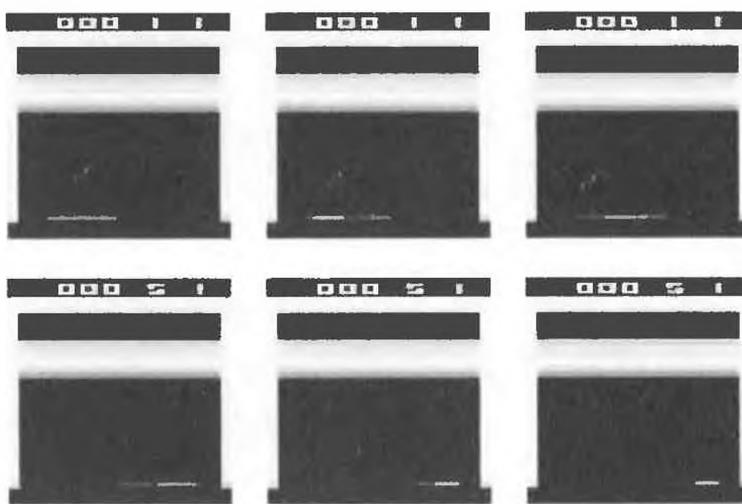
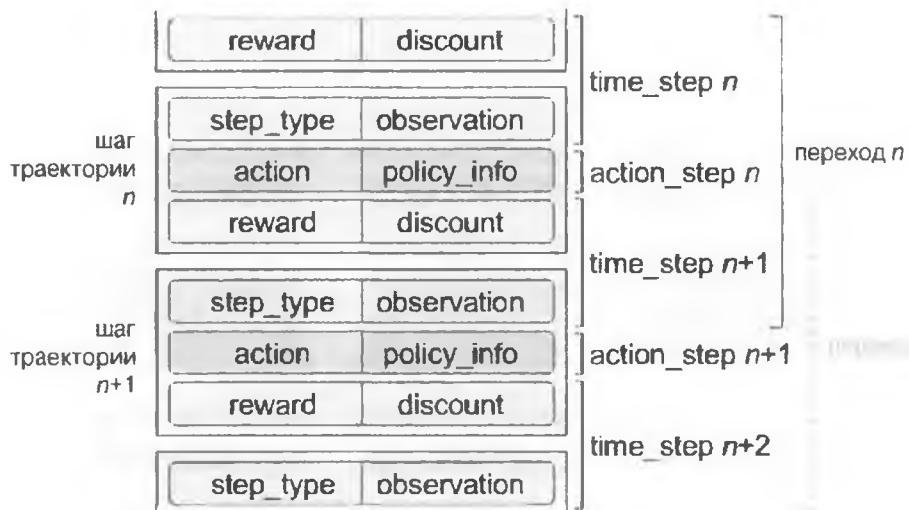


Рис. 18.15. Две траектории, содержащие три последовательных шага каждая

Объект `trajectories` представляет собой именованный кортеж с семью полями. Каждое поле содержит тензор, первыми двумя измерениями которого являются 2 и 3 (т.к. есть две траектории, каждая с тремя шагами). Становится понятным, почему форма поля `observation` выглядит как [2, 3, 84, 84, 4]: две траектории, каждая с тремя шагами и наблюдением каждого шага  $84 \times 84 \times 4$ . Подобным образом тензор `step_type` имеет форму [2, 3]: в рассматриваемом примере обе траектории содержат три последовательных шага в середине эпизода (типы 1, 1, 1). Во второй траектории вы едва видите шарик в левом нижнем углу первого наблюдения, а в следующих двух наблюдениях он исчезает, так что агент собирается утратить жизнь, но эпизод не закончится немедленно, поскольку у него все еще остается несколько жизней.

Каждая траектория — это лаконичное представление последовательности идущих друг за другом временных шагов и шагов действий, предназначенное для избегания избыточности. Как так? Что ж, согласно рис. 18.16 переход  $n$  состоит из временного шага  $n$ , шага действия  $n$  и временного шага  $n+1$ , тогда как переход  $n+1$  образован из временного шага  $n+1$ , шага действия  $n+1$  и временного шага  $n+2$ .



Примечание: шаг траектории  $n$  также включает  $\text{next\_step\_type } n = \text{step\_type } n+1$

**Рис. 18.16. Траектории, переходы, временные шаги и шаги действий**

Если мы просто сохраним эти два перехода прямо в буфере воспроизведения, то временной шаг  $n + 1$  будет дублироваться. Во избежание такого дублирования  $n$ -ный шаг траектории включает только тип и наблюдение из временного шага  $n$  (не его награду и дисконтированную награду), но не содержит наблюдение из временного шага  $n + 1$  (тем не менее, он содержит копию типа следующего временного шага; она является единственным дублированием).

Итак, если у вас есть пакет траекторий, где каждая траектория имеет  $t + 1$  шагов (от временного шага  $n$  до временного шага  $n + t$ ), тогда он содержит все данные от временного шага  $n$  до временного шага  $n + t$  кроме награды и дисконтированной награды из временного шага  $n$  (но включает награду и дисконтированную награду из временного шага  $n + t + 1$ ). Такой пакет представляет  $t$  переходов (от  $n$  до  $n + 1$ , от  $n + 1$  до  $n + 2$ , ..., от  $n + t - 1$  до  $n + t$ ).

Функция `to_transition()` в модуле `tf_agents.trajectories.trajectory` преобразует пакетную траекторию в список, содержащий пакетный временной шаг (`time_step`), пакетный шаг действия (`action_step`) и пакетный следующий временной шаг (`next_time_step`). Обратите внимание, что вторым измерением является 2, а не 3, т.к. между временными шагами  $t + 1$  существуют  $t$  переходов (не переживайте, если слегка запутались; вы поймете, в чем дело):

```
>>> from                                     import to_transition
>>> time_steps, action_steps,
       next_time_steps = to_transition(trajectories)
>>> time_steps.observation.shape
TensorShape([2, 2, 84, 84, 4])  # 3 временных шага = 2 перехода
```



Выбранная траектория в действительности может перекрывать два (и более) эпизода! В таком случае она будет содержать *граничные переходы*, т.е. переходы, где `step_type` равно 2 (конец) и `next_step_type` равно 0 (начало). Разумеется, библиотека TF-Agents надлежащим образом обрабатывает траектории подобного рода (например, переустанавливая состояние политики, когда встречается граница). Метод `is_boundary()` траектории возвращает тензор, который указывает, является каждый шаг границей или нет.

В главном цикле обучения вместо вызова метода `get_next()` мы будем применять `tf.data.Dataset`, что позволит воспользоваться преимуществами

вами мощи API-интерфейса Data (скажем, параллелизмом и упреждающей выборкой). Для этого мы вызываем метод `as_dataset()` буфера воспроизведения:

```
dataset = replay_buffer.as_dataset(  
    sample_batch_size=64,  
    num_steps=2,  
    num_parallel_calls=3).prefetch(3)
```

На каждом шаге обучения мы будем выбирать пакеты из 64 траекторий (как делалось в статье 2015 года о сети DQN) с двумя шагами каждая (т.е. 2 шага = 1 полный переход, включая наблюдение следующего шага). Набор данных обработает три элемента параллельно и выберет с упреждением три пакета.



Для алгоритмов внутри политики, таких как градиенты политики, каждый опыт, применяемый в обучении, должен выбираться один раз и затем отбрасываться. В этом случае вы по-прежнему можете использовать буфер воспроизведения, но вместо работы с Dataset на каждой итерации вызывать метод `gather_all()` буфера воспроизведения с целью получения тензора, содержащего все записанные до сих пор траектории, после чего применять их для выполнения шага обучения и в заключение очищать буфер воспроизведения, вызывая его метод `clear()`.

Теперь, когда все компоненты на месте, мы готовы обучить модель!

## Создание цикла обучения

Чтобы ускорить обучение, мы преобразуем основные функции в функции TF Function. Для этого мы будем использовать функцию `tf_agents.utils.common.function()`, которая является оболочкой `tf.function()` с некоторыми дополнительными экспериментальными вариантами:

```
from                                     import function  
collect_driver.run = function(collect_driver.run)  
agent.train = function(agent.train)
```

Давайте создадим небольшую функцию, которая запустит главный цикл обучения для `n_iterations` итераций:

```

def train_agent(n_iterations):
    time_step = None
    policy_state =
        agent.collect_policy.get_initial_state(tf_env.batch_size)
    iterator = iter(dataset)
    for iteration in range(n_iterations):
        time_step, policy_state =
            collect_driver.run(time_step, policy_state)
        trajectories, buffer_info = next(iterator)
        train_loss = agent.train(trajectories)
        print("\r{} loss:{:.5f}".format(
            iteration, train_loss.loss.numpy()), end="")
        if iteration % 1000 == 0:
            log_metrics(train_metrics)

```

Функция сначала запрашивает у политики сбора ее начальное состояние (для заданного размера пакета среды, равного 1 в рассматриваемом случае). Поскольку политика не запоминает состояние, возвращается пустой кортеж (так что мы могли бы написать `policy_state = ()`). Затем мы создаем итератор по набору данных и запускаем цикл обучения. На каждой итерации мы вызываем метод `run()` драйвера, передавая ему текущий временной шаг (первоначально `None`) и текущее состояние политики. Он запустит политику сбора и соберет опыт для четырех шагов (как мы сконфигурировали ранее), ретранслируя собранные траектории буферу воспроизведения и метрикам. Далее мы выбираем из набора данных один пакет траекторий и передаем его методу `train()` агента. Он возвращает объект `train_loss`, который может варьироваться в зависимости от типа агента. После этого мы отображаем номер итерации и потерю при обучении, а каждые 1 000 итераций выводим все метрики. Теперь можно просто вызвать `train_agent()` для некоторого количества итераций и наблюдать, как агент постепенно учится играть в *Breakout*!

```
train_agent(10000000)
```

Процесс потребует большой вычислительной мощности и немалого терпения (в зависимости от имеющегося оборудования он может длиться часы или даже дни) плюс вам может понадобиться запускать алгоритм несколько раз с различными начальными случайными числами, чтобы получить приемлемые результаты, но после завершения агент обретет сверхчеловеческие способности (по крайней мере, в игре *Breakout*). Вы также можете попробовать обучить агента DQN на других играх Atari: он в состоянии достичь

сверхчеловеческих навыков в большинстве игр-боевиков, но не настолько хороши в играх с длинными сюжетными линиями<sup>22</sup>.

## Обзор ряда популярных алгоритмов обучения с подкреплением

Прежде чем завершить главу, давайте бегло взглянем на несколько популярных алгоритмов RL.

### Алгоритмы “актер-критик”

Семейство алгоритмов RL, которые комбинируют градиенты политики с глубокими Q-сетями. Агент типа “актер-критик” содержит две нейронных сети: сеть политики и сеть DQN. Сеть DQN обучается нормально, изучая опыты агента. Сеть политики обучается не так, как обычные градиенты политики (и гораздо быстрее): вместо оценки ценности каждого действия путем прохождения через множество эпизодов, суммирования дисконтируемых будущих наград и их нормализации агент (актер) полагается на ценности действий, оцененные сетью DQN (критиком). Процесс похож на то, как спортсмен (агент) учится с помощью тренера (сети DQN).

*Асинхронный алгоритм “актер-критик” на основе преимущества*

(*Asynchronous Advantage Actor-Critic — A3C*;

<https://homl.info/a3c>)<sup>23</sup>

Важный вариант алгоритма “актер-критик”, представленный исследователями DeepMind в 2016 году, где множество агентов обучаются параллельно, исследуя разные копии среды. Через регулярные интервалы, но асинхронно (отсюда название) каждый агент передает ряд обновлений весов в главную сеть, после чего получает из этой сети самые последние веса. Таким образом, каждый агент вносит свой вклад в совершенствование главной сети и извлекает пользу из того, что изучили остальные агенты. Кроме того, вместо оценки Q-ценностей сеть DQN оценивает преимущество каждого действия (*Advantage* в названии алгоритма), что стабилизирует обучение.

<sup>22</sup> На рисунке 3 в статье 2015 г.а исследователей из DeepMind сравнивается эффективность этого алгоритма для различных игр Atari (<https://homl.info/dqn2>).

<sup>23</sup> Владимир Мных и др., *Asynchronous Methods for Deep Reinforcement Learning* (Асинхронные методы для глубокого обучения с подкреплением), *Proceedings of the 33rd International Conference on Machine Learning* (2016 г.): с. 1928–1937.

## *Расширенный алгоритм “актер-критик” (Advantage Actor Critic — A2C; <https://homl.info/a2c>)*

Вариант алгоритма А3С, из которого убрана асинхронность. Вся модель обновляется синхронно, поэтому обновления градиентов выполняются по более крупным пакетам, что позволяет модели эффективнее распоряжаться мощью графического процессора.

## *Мягкий алгоритм “актер-критик” (Soft Actor-Critic — SAC; <https://homl.info/sac>)<sup>24</sup>*

Вариант алгоритма “актер-критик”, который предложили в 2018 году Туомас Хаарной и другие исследователи из Калифорнийского университета в Беркли. Он не только узнает награды, но также доводит до максимума энтропию своих действий. Другими словами, алгоритм SAC старается быть настолько непредсказуемым, насколько возможно, и в то же время по-прежнему получать как можно больше наград. Это побуждает агента исследовать среду, что ускоряет обучение и снижает вероятность многократного выполнения того же самого действия, когда сеть DQN производит неидеальные оценки. Алгоритм SAC продемонстрировал поразительную эффективность выборки (в противоположность предшествующих алгоритмов, которые обучались очень медленно). Алгоритм SAC доступен в библиотеке TF-Agents.

## *Оптимизация проксимальных политик (Proximal Policy Optimization — PPO; <https://homl.info/ppo>)<sup>25</sup>*

Алгоритм, основанный на алгоритме А2С, который урезает функцию потерь во избежание чрезмерно больших обновлений весов (что часто приводит к нестабильностям обучения). Алгоритм PPO является упрощением ранее существовавшего алгоритма оптимизации политик областей доверия (*Trust Region Policy Optimization — TRPO*; <https://>

<sup>24</sup> Туомас Хаарной и др., *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor* (Мягкий алгоритм “актер-критик”: глубокое обучение с подкреплением вне политики на основе максимальной энтропии с помощью стохастического актера), *Proceedings of the 35th International Conference on Machine Learning* (2018 г.): с. 1856–1865.

<sup>25</sup> Джон Шульман и др., *Proximal Policy Optimization Algorithms* (Алгоритмы оптимизации проксимальных политик), препринт arXiv:1707.06347 (2017 г.).

`homl.info/trpo`<sup>26</sup>), также предложенного Джоном Шульманом и другими исследователями из OpenAI. Компания OpenAI попала в новости в апреле 2019 благодаря своему искусственному интеллекту под названием OpenAI Five, основанному на алгоритме PPO, который одержал победу над чемпионами мира в многопользовательской игре *Dota 2*. Алгоритм PPO также доступен в библиотеке TF-Agents.

### *Исследование, основанное на любопытстве (<https://homl.info/curiosity>)<sup>27</sup>*

При обучении с подкреплением часто возникает проблема малочисленности наград, которая делает обучение очень медленным и неэффективным. Дипак Патхак и другие исследователи из Калифорнийского университета в Беркли предложили захватывающий способ решения указанной проблемы: почему бы не игнорировать награды и просто сделать агент чрезвычайно любопытным в том, что касается исследования среды? Таким образом, награды становятся присущими агенту, а не поступающими из среды. Точно так же стимулирование любопытства у ребенка с большей вероятностью даст лучшие результаты, чем просто его вознаграждение за хорошие оценки. Как это работает? Агент постоянно пытается спрогнозировать исход своих действий и ищет ситуации, когда исход не совпадает с его прогнозами. Другими словами, он желает быть удивленным. Если исход оказывается предсказуемым (скучным), то агент ищет в другом месте. Однако если исход непредсказуем, но агент замечает, что не имеет над ним никакого контроля, тогда через некоторое время исход также становится скучным. Благодаря одному лишь любопытству авторам статьи удалось обучить агента играть во многие видеоигры: хотя агент не получает штраф за проигрыш, игра начинается заново, что для него скучно, поэтому он учится избегать такой ситуации.

<sup>26</sup> Джон Шульман и др., *Trust Region Policy Optimization* (Оптимизация политик областей доверия), *Proceedings of the 32nd International Conference on Machine Learning* (2015 г.): с. 1889–1897.

<sup>27</sup> Дипак Патхак и др., *Curiosity-Driven Exploration by Self-Supervised Prediction* (Исследование, направляемое любопытством, через прогнозирование со своим учителем), *Proceedings of the 34th International Conference on Machine Learning* (2017 г.): с. 2778–2787.

Мы раскрыли в главе многие темы: градиенты политики, цепи Маркова, марковские процессы принятия решений, Q-обучение, приближенное Q-обучение и глубокое Q-обучение вместе с его основными вариантами (фиксированные цели Q-ценностей, двойная глубокая Q-сеть, соревнующаяся глубокая Q-сеть и воспроизведение опытов по приоритетам). Мы обсудили, как применять библиотеку TF-Agents для полномасштабного обучения агентов и в заключение кратко рассмотрели ряд других популярных алгоритмов. Обучение с подкреплением — огромная и захватывающая область, в которой ежедневно появляются новые идеи и алгоритмы, так что я надеюсь, что эта глава пробудила у вас любопытство: есть целый мир, который нужно исследовать!

## 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 возможных приложения RL, которые не были упомянуты в настоящей главе? Что является средой в каждом из них? Что собой представляет агент? Что собой представляют вероятные действия? Что собой представляют награды?
3. Что такое коэффициент дисконтирования? Может ли оптимальная политика измениться, если модифицировать коэффициент дисконтирования?
4. Как бы вы измеряли эффективность агента обучения с подкреплением?
5. В чем заключается проблема присваивания коэффициентов доверия? Когда она возникает? Как ее можно смягчить?
6. В чем смысл применения буфера воспроизведения?
7. Что собой представляет алгоритм RL вне политики?
8. Воспользуйтесь градиентами политики, чтобы заняться средой LunarLander-v2 из OpenAI Gym. Вам понадобится установить зависимости Box2D (`python3 -m pip install gym[box2d]`).
9. Воспользуйтесь библиотекой TF-Agents для обучения агента, который сумеет достичь сверхчеловеческого уровня в среде SpaceInvaders-v4 с применением любого из доступных алгоритмов.

10. Располагая лишней сотней долларов, вы можете купить Raspberry Pi 3 и несколько недорогих робототехнических компонентов, установить TensorFlow на Pi и неплохо позабавиться! Например, почитайте забавную статью Лукаса Бивальда (<https://homl.info/2>) либо взгляните на GoPiGo или BrickPi. Начните с простых целей, например, заставьте робота покружиться в поисках самого яркого угла (если в нем есть светолучевой датчик) или ближайшего объекта (при наличии в нем датчика звуковой локации) и двигаться в этом направлении. Затем вы можете приступить к применению глубокого обучения: скажем, если робот имеет камеру, тогда вы можете попытаться реализовать алгоритм выявления объектов, чтобы он обнаруживал людей и перемещался в направлении, где они находятся. Можете также задействовать RL, чтобы агент самостоятельно научился использовать моторы для достижения такой цели. Повеселитесь!

Решения приведенных упражнений доступны в приложении А.

# Широкомасштабное обучение и развертывание моделей TensorFlow

Имея в своем распоряжении эффективную модель, которая вырабатывает замечательные прогнозы, что вы будете с ней делать? Вам нужно поместить ее в производственную среду! Задача может оказаться такой же простой, как запуск модели на пакете данных и написание сценария, который будет запускать модель каждую ночь. Однако часто все бывает намного сложнее. Возможно, различным частям вашей инфраструктуры необходимо использовать имеющуюся модель на живых данных и тогда вам придется поместить модель внутрь веб-службы: в таком случае любая часть инфраструктуры может запрашивать модель в любой момент через API-интерфейс REST (или какой-то другой протокол), как обсуждалось в главе 2. Но с течением времени вам понадобится регулярно заново обучать модель на свежих данных и передавать обновленную версию в производственную среду. Вы должны обеспечить поддержку версий модели, элегантного перехода от одной модели к другой, возможности отката к предыдущей модели в случае возникновения проблем и параллельного запуска множества разных моделей для проведения экспериментов A/B<sup>1</sup>. Если созданный вами продукт становится успешным, тогда ваша служба может начать получать много запросов в секунду (*queries per second* — QPS) и ее потребуется масштабировать для поддержки высокой нагрузки. Как мы увидим позже в главе, замечательным решением задачи масштабирования службы будет применение сервера TF Serving, либо на собственной аппаратной инфраструктуре, либо через облачную службу вроде платформы AI Platform инфраструктуры Google Cloud. Он позаботится об эффективном обслуживании вашей модели, об элегантном переходе между

<sup>1</sup> Эксперимент A/B заключается в тестировании двух разных версий продукта на различных подгруппах пользователей с целью проверки, какая версия работает лучше, и получения других сведений.

версиями и о других аспектах. Если вы используете облачную платформу, то будете также иметь в своем распоряжении множество дополнительных средств, таких как мощные инструменты мониторинга.

Кроме того, при наличии большого объема обучающих данных и интенсивных в вычислительном плане моделей время обучения может быть чрезмерно долгим. Если ваш продукт нуждается в быстрой адаптации к изменениям, тогда длительное время обучения может оказаться препятствием (например, подумайте о системе выдачи рекомендаций, раскручивающей новости прошедшей недели). Наверное, еще более важно то, что долгое время обучения не позволит вам экспериментировать с новыми идеями. В машинном обучении (подобно многим другим областям) трудно узнать заранее, какие идеи будут работать, поэтому вы должны опробовать их как можно больше и как можно быстрее. Один из способов ускорить обучение предусматривает применение аппаратных ускорителей, таких как графические или тензорные процессоры. Чтобы двигаться еще быстрее, вы можете обучать модель на нескольких машинах, оснащенных множеством аппаратных ускорителей. Позже выяснится, что задачу упрощает мощный API-интерфейс стратегий распределения (*Distribution Strategies*) из библиотеки TensorFlow.

В настоящей главе мы посмотрим, как развертывать модели сначала на сервере TF Serving и затем на платформе AI Platform инфраструктуры Google Cloud. Мы также бегло взглянем на развертывание моделей в мобильных приложениях, встраиваемых устройствах и веб-приложениях. В заключение мы обсудим, каким образом ускорять вычисления за счет использования графических процессоров (ГП) и как обучать модели на множестве устройств и серверов с применением API-интерфейса Distribution Strategies. Тем для обсуждения много, так что давайте начнем!

## Обслуживание модели TensorFlow

После обучения модели TensorFlow вы можете легко использовать ее в любом коде Python: в случае модели `t.f.keras` просто вызывайте ее метод `predict()`! Но по мере роста вашей инфраструктуры наступает момент, когда модель предпочтительнее поместить внутрь небольшой службы, единственная роль которой состоит в вырабатывании прогнозов, и заставить остаток инфраструктуры запрашивать ее (скажем, через API-интерфейс REST

или gRPC)<sup>2</sup>. Такой подход обеспечивает отделение вашей модели от остальной части инфраструктуры, делая возможным легкое переключение между версиями модели или надлежащее масштабирование службы (независимо от остатка инфраструктуры), проведение экспериментов A/B и гарантия того, что все программные компоненты полагаются на одинаковые версии модели. Вдобавок упрощаются тестирование и разработка, а также многое другое. Вы могли бы создать собственную микро-службу с применением любой желаемой технологии (например, используя библиотеку Flask), но зачем заново изобретать колесо, если можно всего лишь прибегнуть к услугам TF Serving?

## Использование TensorFlow Serving

TF Serving — очень эффективный и проверенный в полевых условиях сервер моделей, который написан на языке C++. Он способен выдерживать высокую нагрузку, обслуживать множество версий моделей и наблюдать за хранилищем моделей, чтобы автоматически развертывать самые последние версии, и т.д. (рис. 19.1).

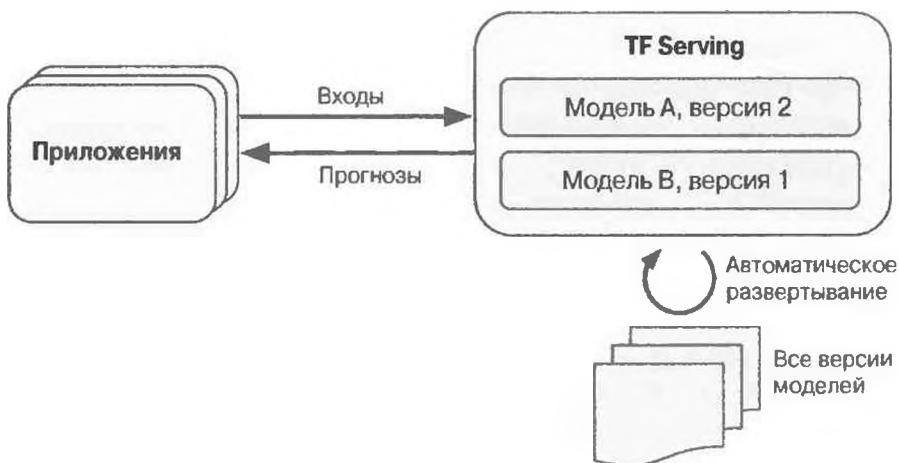


Рис. 19.1. Сервер TF Serving может обслуживать множество моделей и автоматически развертывать последнюю версию каждой модели

<sup>2</sup> API-интерфейс REST (или RESTful) использует стандартные команды HTTP, такие как GET, POST, PUT и DELETE с входными и выходными данными в формате JSON. Протокол gRPC сложнее, но и эффективнее. Обмен данными производится с применением протокольных буферов (см. главу 13).

Итак, давайте предположим, что вы обучили модель MNIST, основанную на `tf.keras`, и хотите развернуть ее с помощью TF Serving. Первым делом вам понадобится экспорттировать эту модель в формат `SavedModel` (представление сохраненной модели) из TensorFlow.

## Экспорттирование в формат `SavedModel`

Библиотека TensorFlow предлагает простую функцию `tf.saved_model.save()` для экспорттирования моделей в формат `SavedModel`. Функции потребуется только предоставить модель, указав ее имя и номер версии, после чего она сохранит вычислительный граф и веса модели:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit([...])

model_version = "0001"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Обычно лучше включать в финальную модель, которая будет экспорттироваться, все слои предварительной обработки, чтобы после развертывания в производственной среде она могла поглощать данные в их естественной форме. Такой прием позволяет избежать необходимости в отдельном выполнении предварительной обработки внутри приложения, где применяется модель. Встраивание шагов предварительной обработки внутрь модели также упрощает их обновление в будущем и снижает риск несоответствия между моделью и шагами предварительной обработки, которые для нее требуются.



Поскольку представление `SavedModel` хранит вычислительный граф, его можно использовать только с моделями, которые основаны исключительно на операциях TensorFlow кроме операции `tf.py_function()` (она содержит произвольный код Python). Также не разрешены динамические модели `tf.keras` (см. приложение Ж), потому что они не могут быть преобразованы в вычислительные графы. Динамические модели необходимо обслуживать с применением других инструментов (скажем, Flask).

Представление `SavedModel` поддерживает версию модели. Она хранится как каталог, содержащий файл `saved_model.pb`, в котором определен

вычислительный граф (представленный в виде сериализированного протокольного буфера), и подкаталог `variables` со значениями переменных. Для моделей, содержащих большое количество весов, значения переменных могут быть разнесены по множеству файлов. Представление `SavedModel` также включает подкаталог `assets`, который может содержать дополнительные данные, такие как файлы словарей, имена классов или примеры образцов для этой модели. Ниже показана структура каталогов (подкаталог `assets` в текущем примере отсутствует):

```
my_mnist_model
└── 0001
    ├── assets
    ├── saved_model.pb
    └── variables
        ├── variables.data-00000-of-00001
        └── variables.index
```

Как и можно было ожидать, представление `SavedModel` загружается с использованием функции `tf.saved_model.load()`. Тем не менее, возвращаемый объект является не моделью Keras, а представлением `SavedModel`, включающим вычислительный граф и значения переменных. Его можно применять подобно функции, и оно будет вырабатывать прогнозы (обеспечьте передачу входов в виде тензоров и также устанавливайте аргумент `training`, как правило, в `False`):

```
saved_model = tf.saved_model.load(model_path)
y_pred = saved_model(X_new, training=False)
```

В качестве альтернативы вы можете поместить функцию прогнозирования этого представления `SavedModel` в оболочку модели Keras:

```
inputs = keras.layers.Input(shape=...)
outputs = saved_model(inputs, training=False)
model = keras.models.Model(inputs=[inputs], outputs=[outputs])
y_pred = model.predict(X_new)
```

Библиотека TensorFlow также предлагает небольшой инструмент командной строки `saved_model_cli`, предназначенный для инспектирования представлений `SavedModel`:

```
$ export ML_PATH="$HOME/ml" # указывает на этот проект, где бы он ни был
$ cd $ML_PATH
$ saved_model_cli show --dir my_mnist_model/0001 --all
```

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

Определение MetaGraphDef с tag-set: serve содержит следующие

SignatureDef:

signature\_def['\_\_saved\_model\_init\_op']:

[...]

signature\_def['serving\_default']:

The given SavedModel SignatureDef contains the following input(s):

Определение SignatureDef данного представления SavedModel

содержит следующие входы:

inputs['flatten\_input'] tensor\_info:

  dtype: DT\_FLOAT

  shape: (-1, 28, 28)

  name: serving\_default\_flatten\_input:0

The given SavedModel SignatureDef contains the following output(s):

Определение SignatureDef данного представления SavedModel

содержит следующие входы:

outputs['dense\_1'] tensor\_info:

  dtype: DT\_FLOAT

  shape: (-1, 10)

  name: StatefulPartitionedCall:0

Method name is: tensorflow/serving/predict

Имя метода: tensorflow/serving/predict

Представление SavedModel содержит один или большее число метаграфов. Метаграф — это вычислительный граф плюс ряд определений сигнатур функций (включая имена входов и выходов, типы и формы). Каждый метаграф идентифицируется комплектом *меток* (*tag*). Например, вам может понадобиться метаграф с полным вычислительным графом, включающим операции обучения (помеченный, скажем, с помощью "train") и еще один метаграф, который содержит подрезанный вычислительный граф с только операциями прогнозирования, в том числе операциями, специфичными для ГП (такой метаграф может быть помечен как "serve", "gru"). Однако когда вы передаете функции `tf.saved_model.save()` модель `tf.keras`, функция по умолчанию сохраняет намного более простое представление SavedModel: она сохраняет единственный метаграф, помеченный как "serve", который содержит два определения сигнатур — функции инициализации (по имени `__saved_model_init_op`, о которой вы не должны беспокоиться) и стандартной функции обслуживания (по имени `serving_default`). При сохранении модели `tf.keras` стандартная функция обслуживания соответствует функции `call()` модели, которая, как должно быть понятно, вырабатывает прогнозы.

Инструмент `saved_model_cli` также можно использовать для вырабатывания прогнозов (с целью тестирования, не в интересах производственной среды). Пусть вы имеете массив NumPy (`X_new`), содержащий три изображения рукописных цифр, для которых нужно вырабатывать прогнозы. Сначала вы должны экспорттировать его в формат при из NumPy:

```
np.save("my_mnist_tests.npy", X_new)
```

Далее введите команду `saved_model_cli` следующего вида:

```
$ saved_model_cli run --dir my_mnist_model/0001 --tag_set serve \
                      --signature_def serving_default \
                      --inputs flatten_input=my_mnist_tests.npy
[...] Result for output key dense_1:
Результат для выходного ключа dense_1:
[[1.1739199e-04 1.1239604e-07 6.0210604e-04 [...] 3.9471846e-04]
 [1.2294615e-03 2.9207937e-05 9.8599273e-01 [...] 1.1113169e-07]
 [6.4066830e-05 9.6359509e-01 9.0598064e-03 [...] 4.2495009e-04]]
```

Вывод инструмента содержит 10 вероятностей классов для каждого из 3 образцов. Замечательно! Теперь, имея работающее представление SavedModel, необходимо установить TF Serving.

## Установка TensorFlow Serving

Установить TF Serving можно многими способами: применяя образ Docker<sup>3</sup>, используя диспетчер пакетов системы, устанавливая из исходного кода и т.д. Давайте выберем вариант с образом Docker, который настоятельно рекомендуется командой разработчиков TensorFlow как простой в установке, не вносящий путаницу в вашу систему и обеспечивающий высокую производительность. Сначала установите Docker (<https://docker.com/>). Затем загрузите из Docker официальный образ TF Serving:

```
$ docker pull tensorflow/serving
```

---

<sup>3</sup> На тот случай, если вы не знакомы с платформой Docker: она позволяет легко загружать набор приложений, упакованных в образ Docker (включая все зависимости и обычно неплохую стандартную конфигурацию), и запускать их на своей системе с применением механизма Docker. Когда вы запускаете образ, механизм создает контейнер Docker, который сохраняет приложения хорошо изолированными от вашей системы (но при желании им можно предоставить ограниченный доступ). Контейнер Docker похож на виртуальную машину, но намного быстрее и легковеснее, т.к. опирается непосредственно на ядро хоста. Это означает, что образу не нужно содержать или запускать собственное ядро.

Далее вы можете создать контейнер Docker для запуска загруженного образа:

```
$ docker run -it --rm -p 8500:8500 -p 8501:8501 \
    -v "$ML_PATH/my_mnist_model:/models/my_mnist_model" \
    -e MODEL_NAME=my_mnist_model \
    tensorflow/serving
[...]
2019-06-01 [...] loaded servable version {name: my_mnist_model
version: 1}
                    загружена обслуживаемая версия {имя: my_mnist_model
версия: 1}
2019-06-01 [...] Running gRPC ModelServer at 0.0.0.0:8500 ...
                    Запуск gRPC ModelServer на 0.0.0.0:8500 ...
2019-06-01 [...] Exporting HTTP/REST API at:localhost:8501 ...
                    Экспортирование HTTP/REST API на: localhost:8501 ...
[evhttp_server.cc : 237] RAW: Entering the event loop ...
                    RAW: вход в цикл событий ...
```

Вот и все! Сервер TF Serving функционирует. Он загрузил нашу модель MNIST (версии 1) и обслуживает ее посредством gRPC (на порте 8500) и REST (на порте 8501). Ниже описаны все параметры командной строки.

#### **-it**

Делает контейнер интерактивным (так что вы можете нажать <Ctrl+C>, чтобы остановить его) и отображает вывод сервера.

#### **--rm**

Удаляет контейнер, когда вы его остановили (не стоит загромождать свою машину прерванными контейнерами). Тем не менее, образ не удаляется.

#### **-p 8500:8500**

Заставляет механизм Docker переадресовывать TCP-порт 8500 хоста на TCP-порт 8500 контейнера. По умолчанию TF Serving применяет этот порт для обслуживания API-интерфейса gRPC.

#### **-p 8501:8501**

Переадресует TCP-порт 8501 хоста на TCP-порт 8501 контейнера. По умолчанию TF Serving использует этот порт для обслуживания API-интерфейса REST.

**-v "\$ML\_PATH/my\_mnist\_model:/models/my\_mnist\_model"**  
Делает каталог \$ML\_PATH/my\_mnist\_model хоста доступным контейнеру по пути /models/mnist\_model. В среде Windows может потребоваться замена / на \ в пути хоста (но не в пути контейнера).

**-e MODEL\_NAME=my\_mnist\_model**

Устанавливает переменную среды MODEL\_NAME контейнера, так что серверу TF Serving известно, какую модель обслуживать. По умолчанию он будет искать модели в каталоге /models и автоматически обслуживать их последние найденные версии.

## **tensorflow/serving**

Имя образа, подлежащего запуску.

А теперь давайте вернемся к Python и займемся запрашиванием этого сервера с применением сначала API-интерфейса REST и затем API-интерфейса gRPC.

### **Запрашивание TF Serving через API-интерфейс REST**

Начнем с создания запроса. Он должен содержать имя сигнатуры функции, подлежащей вызову, и входные данные:

```
import  
  
input_data_json = json.dumps({  
    "signature_name": "serving_default",  
    "instances": X_new.tolist(),  
})
```

Обратите внимание, что формат JSON полностью основан на тексте, поэтому массив NumPy по имени X\_new должен быть преобразован в список Python и затем сформирован как JSON:

```
>>> input_data_json  
'{"signature_name": "serving_default", "instances": [[[0.0, 0.0, 0.0,  
[...] 0.3294117647058824, 0.725490196078431, [...very long],  
0.0, 0.0, 0.0, 0.0]]]}'
```

Далее мы отправим входные данные серверу TF Serving, посыпая HTTP-запрос POST. Его легко сделать с использованием библиотеки requests (она не входит в состав стандартной библиотеки Python, так что первым делом ее потребуется установить, скажем, применив pip):

```
import  
SERVER_URL = 'http://localhost:8501/v1/models/my_mnist_model:predict'  
response = requests.post(SERVER_URL, data=input_data_json)  
response.raise_for_status() # в случае ошибки генерировать исключение  
response = response.json()
```

Ответом будет словарь с единственным ключом "predictions". Соответствующее ему значение представляет собой список прогнозов. Он является списком Python, а потому мы преобразуем его в массив NumPy и округлим содержащиеся в нем числа с плавающей точкой до второго десятичного разряда:

```
>>> y_proba = np.array(response["predictions"])  
>>> y_proba.round(2)  
array([10. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],  
      [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],  
      [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Ура, у нас есть прогнозы! Модель близка к 100%-ной уверенности в том, что на первом изображении представлена 7, имеет 99%-ную уверенность в том, что на втором изображении находится 2, и на 96% уверена в том, что третье изображение представляет 1.

API-интерфейс REST удобен и прост, но работает хорошо, когда входные и выходные данные не очень велики. Кроме того, практически любое клиентское приложение может делать запросы REST без дополнительных зависимостей, тогда как другие протоколы не всегда доступны настолько легко. Однако он основан на протоколе JSON, который базируется на тексте и довольно многословен. Например, нам пришлось преобразовать массив NumPy в список Python, и каждое число с плавающей точкой в конечном итоге представляется в виде строки. Это крайне неэффективно как с точки зрения времени сериализации/десериализации (для преобразования всех чисел с плавающей точкой в строки и наоборот), так и в плане размера полезной нагрузки: многие числа с плавающей точкой оказываются представленными с использованием более 15 символов, которые транслируются в свыше 120 битов для 32-битных чисел с плавающей точкой! Результатом будет высокая задержка и расход полосы пропускания при передаче крупных массивов NumPy<sup>4</sup>. Итак, давайте взамен будем применять API-интерфейс gRPC.

---

<sup>4</sup> Справедливости ради следует отметить, что проблему можно смягчить, сначала сериализируя данные и кодируя их в формат Base64 перед созданием запроса REST. Вдобавок запросы REST можно сжимать с использованием утилиты gzip, что значительно уменьшил размер полезной нагрузки.



Когда приходится передавать большие объемы данных, намного лучше использовать API-интерфейс gRPC (если клиент его поддерживает), т.к. он базируется на компактном двоичном формате и является эффективным протоколом связи (основанным на кадрировании HTTP/2).

## Запрашивание TF Serving через API-интерфейс gRPC

API-интерфейс gRPC на входе ожидает сериализированный протокольный буфер PredictRequest и выдает сериализированный протокольный буфер PredictResponse. Упомянутые протобуфера представляют собой часть библиотеки tensorflow-serving-api, которую придется установить (скажем, с применением pip). Для начала создадим запрос:

```
from                                     import PredictRequest
request = PredictRequest()
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

Приведенный выше код создает протокольный буфер PredictRequest и заполняет обязательные поля, включая имя модели (определенное ранее), имя сигнатуры функции, которую мы хотим вызывать, и входные данные в форме протокольного буфера Tensor. Функция tf.make\_tensor\_proto() создает протокольный буфер Tensor на основе заданного тензора или массива NumPy, в рассматриваемом случае X\_new.

Далее мы отправим запрос серверу и получим его ответ (для чего понадобится библиотека grpcio, которую можно установить с использованием pip):

```
import
from                                     import prediction_service_pb2_grpc
channel = grpc.insecure_channel('localhost:8500')
predict_service =
    prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

Код довольно прямолинеен: после импортирования мы создаем канал связи gRPC с localhost на TCP-порте 8500, затем создаем службу gRPC на этом канале и применяем ее для отправки запроса с 10-секундным тайм-

аутом (нельзя сказать, что вызов считается синхронным: он будет блокироваться до тех пор, пока не получит ответ или не истечет время тайм-аута). В нашем примере канал является незащищенным (нет шифрования, нет аутентификации), но gRPC и TensorFlow Serving также поддерживают защищенные каналы через SSL/TLS.

Давайте преобразуем протокольный буфер PredictResponse в тензор:

```
output_name = model.output_names[0]
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

Если вы выполните показанный код и выведете `y_proba.numpy()`.  
`round(2)`, то получите в точности те же самые оценки вероятностей классов, что и ранее. И это все, что нужно сделать: с помощью лишь нескольких строк кода вы можете дистанционно обращаться к своей модели TensorFlow, используя либо REST, либо gRPC.

## Развертывание новой версии модели

А теперь давайте создадим новую версию модели и экспортируем представление SavedModel в каталог `my_mnist_model/0002`, как делалось ранее:

```
model = keras.models.Sequential([...])
model.compile([...])
history = model.fit(...)

model_version = "0002"
model_name = "my_mnist_model"
model_path = os.path.join(model_name, model_version)
tf.saved_model.save(model, model_path)
```

Сервер TensorFlow Serving через регулярные интервалы (задержка допускает конфигурирование) проверяет наличие новых версий моделей. Обнаружив новую версию, он автоматически элегантно обработает эволюционный переход: по умолчанию сервер ответит на ожидающие запросы (если они есть) посредством предыдущей версии модели, в то время как новые запросы будут обрабатываться с помощью новой версии<sup>5</sup>.

---

<sup>5</sup> Если представление SavedModel содержит примеры образцов в каталоге `assets/extr`a, тогда вы можете сконфигурировать сервер TF Serving для запуска модели на таких образцах перед началом обслуживания новых запросов с ее помощью. Это называется разогревом модели (*model warmup*): он гарантирует, что все надлежащим образом загружено, избегая длительного времени ответа на первые запросы.

После того, как для всех ожидающих запросов сделаны ответы, предыдущая версия модели выгружается, в чем можно удостовериться, заглянув в журналы TensorFlow Serving:

```
[...]  
reserved resources to load servable {name: my_mnist_model version: 2}  
зарезервированные ресурсы для загрузки обслугиваемой версии {имя:  
my_mnist_model версия: 2}  
[...]  
Reading SavedModel from: /models/my_mnist_model/0002  
Reading meta graph with tags { serve }  
Successfully loaded servable version {name: my_mnist_model version: 2}  
Quiescing servable version {name: my_mnist_model version: 1}  
Done quiescing servable version {name: my_mnist_model version: 1}  
Unloading servable version {name: my_mnist_model version: 1}  
Чтение представления SavedModel из: /models/my_mnist_model/0002  
Чтение метаграфа с метками { serve }  
Успешная загрузка обслугиваемой версии {имя: my_mnist_model версия: 2}  
Замораживание обслугиваемой версии {имя: my_mnist_model версия: 1}  
Окончание замораживания обслугиваемой версии {имя: my_mnist_model  
версия: 1}  
Выгрузка обслугиваемой версии {имя: my_mnist_model версия: 1}
```

Такой подход обеспечивает гладкий переход, но может привести к расходу слишком большого объема ОЗУ (особенно ОЗУ графического процессора, размер которого наиболее ограничен). В этом случае вы можете так сконфигурировать сервер TF Serving, чтобы он обрабатывал все ожидающие запросы посредством предыдущей версии модели и выгружал ее перед загрузкой и применением новой версии модели. Подобная конфигурация будет предотвращать присутствие в памяти обеих версий, но в течение короткого периода служба окажется недоступной.

Итак, сервер TF Serving делает развертывание новых версий моделей довольно простым. Кроме того, если вы обнаруживаете, что версия 2 работает не настолько хорошо, как ожидалось, то откат к версии 1 сводится всего лишь к удалению каталога `my_mnist_model/0002`.



Еще одной замечательной особенностью сервера TF Serving является его возможность автоматического пакетирования, которую можно активизировать с использованием параметра `--enable_batching` при начальном запуске. Когда сервер

TF Serving получает множество запросов в рамках короткого периода времени (задержка поддается конфигурированию), он будет автоматически упаковывать их вместе перед применением модели. Результатом оказывается значительный рост производительности за счет использования мощи ГП. После того, как модель возвратит прогнозы, сервер TF Serving направляет каждый прогноз надлежащему клиенту. Вы можете обеспечить большую полосу пропускания в обмен на короткий период ожидания, увеличив задержку пакетирования (с помощью параметра `--batching_parameters_file`).

Если вы ожидаете получать много запросов в секунду, то у вас может возникнуть желание развернуть TF Serving на множестве серверов и балансировать нагрузку для запросов (рис. 19.2). Решение потребует развертывания и управления многочисленными контейнерами TF Serving на серверах. Один из способов предусматривает применение инструмента вроде Kubernetes (<https://kubernetes.io/>), который представляет собой систему с открытым кодом, предназначенную для упрощения координации контейнеров на множестве серверов. Если вы не хотите приобретать, обслуживать и модернизировать аппаратную инфраструктуру, то будете использовать виртуальные машины на облачной платформе, такой как Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud и другие решения типа “платформа как услуга” (*Platform-as-a-Service — PaaS*). Управление всеми виртуальными машинами, поддержка координации контейнеров (даже с помощью Kubernetes), забота о конфигурации TF Serving, настройка и мониторинг представляют собой работы, которые могут занять полный рабочий день. К счастью, существуют поставщики, способные самостоятельно позаботиться обо всем этом. В главе мы будем применять платформу AI Platform инфраструктуры Google Cloud, потому что в настоящее время она является единственной платформой с тензорными процессорами, поддерживает TensorFlow 2, предлагает подходящий комплект служб искусственного интеллекта (скажем, AutoML, Vision API, Natural Language API) и у меня есть большой опыт работы с ней. Но в данной области имеются и другие поставщики, такие как Amazon AWS SageMaker и Microsoft AI Platform, которые тоже способны обслуживать модели TensorFlow.

А теперь давайте посмотрим, как обслуживать нашу замечательную модель MNIST в облаке!

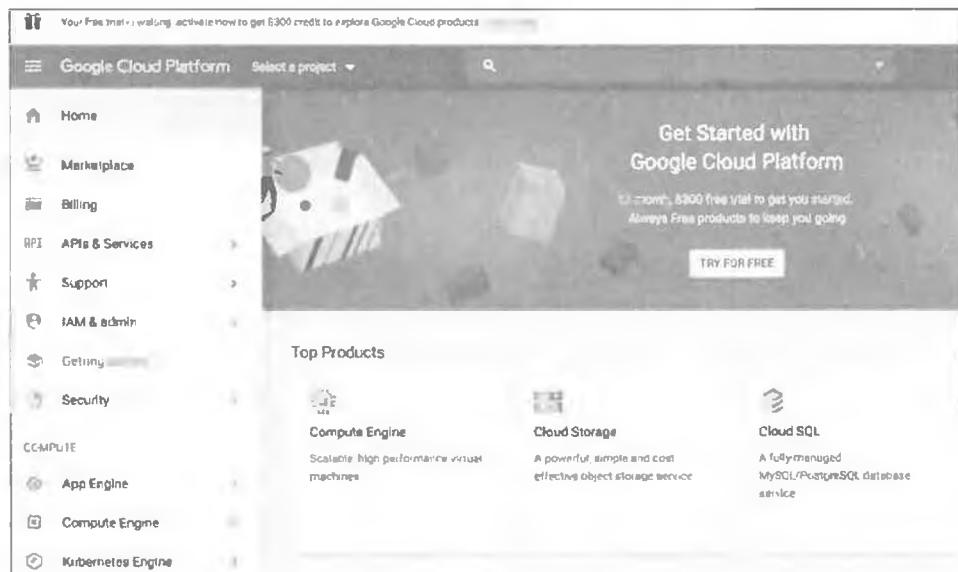


*Рис. 19.2. Масштабирование TF Serving с балансировкой нагрузки*

## **Создание службы прогнозирования на облачной платформе Google**

Прежде чем можно будет развернуть модель, необходимо провести небольшую настройку.

1. Войдите с помощью своего аккаунта Google и откройте консоль облачной платформы Google (*Google Cloud Platform – GCP*), перейдя по ссылке <https://console.cloud.google.com/> (рис. 19.3). Если вы не имеете аккаунта Google, то должны его создать.



*Рис. 19.3. Консоль облачной платформы Google*

2. Если вы впервые работаете с GCP, тогда вам придется прочитать и согласиться с условиями использования. При желании щелкните на ссылке Tour Console (Тур по консоли). На момент написания главы новым пользователям предлагался бесплатный пробный период и кредит \$300, который можно расходовать в течение 12 месяцев. Вам понадобится лишь небольшая его часть для оплаты за службы, применяемые в главе. После подписки на бесплатный пробный период вам все равно потребуется создать профиль оплаты и ввести номер своей кредитной карты: он используется для целей верификации (возможно, чтобы воспрепятствовать многократной подписке на бесплатный пробный период), но счет вам выставляться не будет. В случае запроса активизируйте и обновите платежный аккаунт.
3. Если вы работали с GCP ранее и бесплатный пробный период истек, тогда службы, применяемые в данной главе, обойдутся в определенную сумму. Она не должна быть слишком большой, особенно если вы не забудете отключить службы, когда они уже не нужны. Прежде чем запускать любую службу, удостоверьтесь в том, что понимаете и соглашаетесь с условиями ценообразования. Настоящим я снимаю с себя всякую ответственность, если стоимость служб окажется выше ожидаемой вами! Также убедитесь в том, что ваш платежный аккаунт активен. Для проверки откройте меню навигации слева, щелкните на пункте Billing (Оплата) и удостоверьтесь, что настроили способ оплаты и что платежный аккаунт активен.
4. Каждый ресурс в GCP принадлежит проекту. Это включает все виртуальные машины, которые вы можете использовать, сохраняемые файлы и запускаемые задачи обучения. Когда вы создаете аккаунт, GCP автоматически создает проект по имени My First Project (Мой первый проект). При желании можете изменить его отображаемое имя на странице настроек проекта: выберите в меню навигации (слева) пункт IAM & admin → Settings (IAM и администрирование → Настройки), измените отображаемое имя проекта и щелкните на кнопке Save (Сохранить). Обратите внимание, что проект также имеет уникальный идентификатор и номер. Вы можете выбрать идентификатор проекта при его создании, но изменить его позже нельзя. Номер проекта генерируется автоматически и не может быть изменен. Если вы хотите создать новый проект, тогда щелкните на имени проекта в верхней части страницы.

цы, затем щелкните на кнопке **New Project** (Создать проект) и введите идентификатор проекта. Удостоверьтесь в активности оплаты для нового проекта.



Всегда настраивайте будильник, чтобы напоминать себе об отключении служб, когда известно, что они будут необходимы в течение лишь нескольких часов, иначе вы можете оставить их функционирующими на протяжении дней или месяцев, приводя к потенциально значительным расходам.

5. Имея аккаунт GCP с активной оплатой, вы можете приступить к использованию служб. Первым делом потребуется *облачное хранилище Google* (*Google Cloud Storage — GCS*): сюда будут помещаться представления *SavedModel*, обучающие данные и т.д. Прокрутите меню навигации вниз до раздела *Storage* (Хранилище) и выберите пункт *Storage → Browser* (Хранилище → Обзор). Все ваши файлы попадают в один или большее число *сегментов* (*bucket*). Щелкните на кнопке *Create Bucket* (Создать сегмент) и выберите имя сегмента (может возникнуть необходимость сначала активизировать API-интерфейс *Storage*). Для сегментов GCS применяется единственное всемирное пространство имен, поэтому простые имена вроде “*machine-learning*” почти наверняка не будут доступными. Удостоверьтесь в том, что имя сегмента соответствует соглашениям об именах DNS, т.к. оно может использоваться в записях DNS. Кроме того, имена сегментов являются публичными, поэтому не помещайте сюда ничего конфиденциального. Для обеспечения уникальности принято указывать в качестве префикса доменное имя или название компании либо просто применять случайное число как часть имени. Выберите место, где хотите разместить сегмент, и оставьте стандартные варианты для остальных параметров. Щелкните на кнопке *Create* (Создать).
6. Загрузите в свой сегмент созданную ранее папку *my\_mnist\_model* (включая одну или больше версий). Для этого просто выберите *Storage → Browser*, щелкните на сегменте и перетащите на него папку *my\_mnist\_model* со своей системы (рис. 19.4). В качестве альтернативы можете щелкнуть на кнопке *Upload folder* (Загрузить папку) и выбрать папку *my\_mnist\_model*, чтобы загрузить ее. По умолчанию максимальный размер представления *SavedModel* составляет 250 Мбайт, но разрешено запрашивать более высокую квоту.



Рис. 19.4. Загрузка представления *SavedModel* в облачное хранилище *Google*

7. Теперь понадобится сконфигурировать платформу AI Platform (ранее известную как ML Engine), чтобы ей было известно, какие модели и версии вы хотите использовать. Прокрутите меню навигации вниз до раздела Artificial Intelligence (Искусственный интеллект) и выберите пункт AI Platform → Models (Платформа AI → Модели). Щелкните на кнопке Activate API (Активизировать API-интерфейс), подождите несколько минут до завершения активизации и щелкните на кнопке Create model (Создать модель). Укажите детальные сведения о модели (рис. 19.5) и щелкните на кнопке Create (Создать).
8. Имея модель на платформе AI Platform, вам нужно создать версию модели. Щелкните на только что созданной модели в списке моделей, щелкните на кнопке Create version (Создать версию) и укажите подробные сведения о версии (рис. 19.6): имя, описание, версию Python (3.5 или выше), фреймворк (TensorFlow), версию фреймворка (2.0, если доступна, или 1.13)<sup>6</sup>, версию исполняющей среды MO (2.0, если доступна, или 1.13), тип машины (пока что выберите Single core CPU (ЦП с единственным ядром)), путь к модели в GCS (полный путь к папке с фактической версией, например, gs://my-mnist-model-bucket/my\_mnist\_model/0002/), масштабирование (выберите automatic (автоматическое)) и минимальное количество контейнеров, которые должны всегда запускаться (оставьте это поле пустым). Затем щелкните на кнопке Save (Сохранить).

<sup>6</sup> На момент написания главы версия TensorFlow 2 пока не была доступна на платформе AI Platform, но все в порядке: вы можете использовать версию 1.13 и она будет вполне нормально выполнять ваши представления *SavedModel* из TF 2.

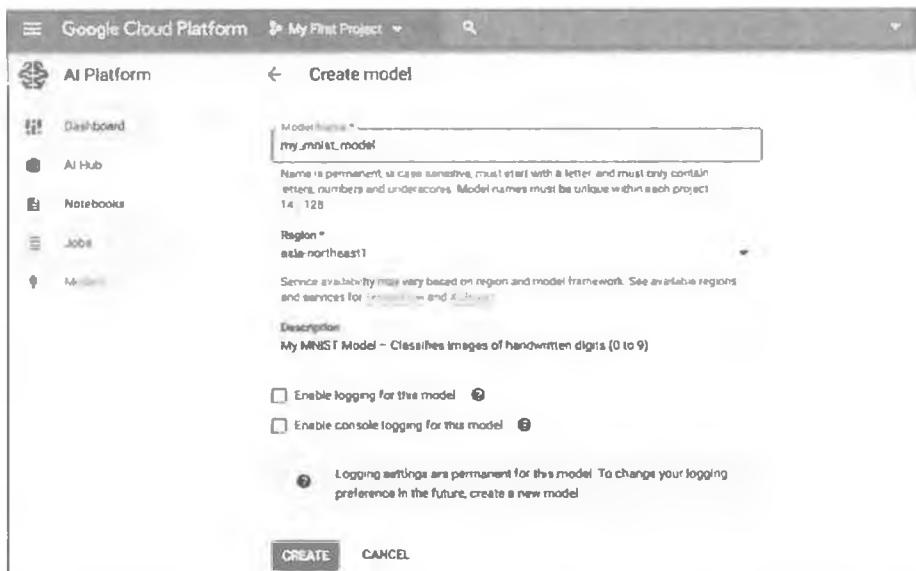


Рис. 19.5. Создание новой модели на платформе AI Platform инфраструктуры Google Cloud

The screenshot shows the 'Create version' page. At the top is a back arrow and the title 'Create version'. Below is a note: 'To create a new version of your model, make necessary adjustments to your saved model file before exporting and store your exported model in Cloud Storage.' with a 'Learn more' link. The 'Name' field contains 'v0001'. A note below says 'Name cannot be changed, is case sensitive, must start with a letter, and may only contain letters, numbers, and underscores 5 / 128'. The 'Description' field contains 'Dense net with 2 layers (100, 10 units)'. The 'Python version' dropdown is set to '3.5'. A note below says 'Select the Python version you used to train the model'. The 'Framework' dropdown is set to 'TensorFlow'.

Рис. 19.6. Создание новой версии модели на платформе AI Platform инфраструктуры Google Cloud

Примите поздравления, вы развернули свою первую модель в облаке! Поскольку было выбрано автоматическое масштабирование, платформа AI Platform будет запускать больше контейнеров TF Serving, когда количество запросов в секунду растет, и балансировать нагрузку между контейнерами. Если QPS снижается, платформа AI Platform автоматически будет останавливать контейнеры. Следовательно, затраты напрямую связаны с QPS (а также с выбранным типом машины и объемом данных, хранимых в GCS). Подобная модель ценообразования особенно полезна для нерегулярных пользователей и для служб с важными скачками использования, равно как и для стартапов: цена остается низкой до тех пор, пока стартап действительно не начнется.



Если вы не используете службу прогнозирования, то платформа AI Platform остановит все контейнеры. Это значит, что вы будете платить только за занимаемый объем хранилища (несколько центов за гигабайт в месяц). Обратите внимание, что когда вы запрашиваете службу, платформе AI Platform понадобится запустить контейнер TF Serving, что займет несколько секунд. Если такая задержка неприемлема, тогда при создании версии модели вам придется установить минимальное количество контейнеров TF Serving в 1. Разумеется, это означает, что одна машина будет выполняться постоянно, а потому месячная плата окажется выше.

Теперь давайте запрашивать службу прогнозирования!

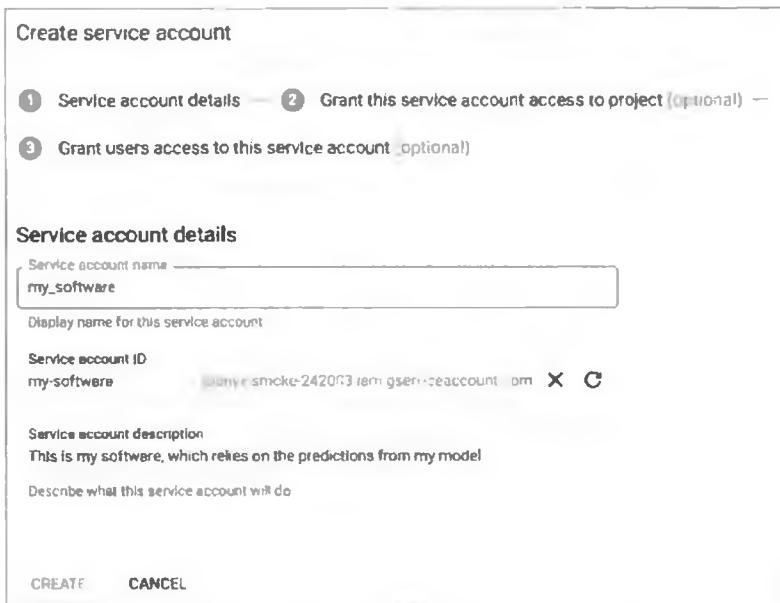
## Использование службы прогнозирования

“За кулисами” платформа AI Platform просто запускает TF Serving, так что в принципе вы могли бы применять тот же самый код, как и ранее, если знаете, какой URL запрашивать. Существует лишь одна проблема: платформа GCP также заботится о шифровании и аутентификации. Шифрование основано на SSL/TLS, а аутентификация — на маркерах: в каждом запросе серверу должен посыпаться секретный маркер аутентификации. Итак, прежде чем ваш код сможет задействовать службу прогнозирования (или любую другую службу GCP), он должен получить маркер. Вскоре мы выясним, как это сделать, но сначала вам нужно сконфигурировать аутентификацию и выдать своему приложению права доступа на платформе GCP. У вас есть два варианта для аутентификации.

- Ваше приложение (т.е. клиентский код, который будет запрашивать службу прогнозирования) могло бы проводить аутентификацию с использованием учетных данных пользователя с вашим именем и паролем в Google. Применение учетных данных пользователя давало бы вашему приложению в точности те же права, как на платформе GCP, что определенно больше, чем ему необходимо. Кроме того, вам пришлось бы разместить свои учетные данные в приложении и тогда любой, имеющий доступ, сможет похитить их и получить полный доступ к вашему аккаунту GCP. Короче говоря, не выбирайте этот вариант; он приемлем лишь в очень редких случаях (например, когда ваше приложение нуждается в доступе к аккаунту GCP своего пользователя).
- Клиентский код может выполнять аутентификацию с помощью *сервисного аккаунта* (*service account*), который представляет приложение, а не пользователя. Обычно ему выдаются крайне ограниченные права доступа: строго то, что необходимо, но не более. Такой вариант является рекомендуемым.

Итак, давайте создадим сервисный аккаунт для приложения: выберите в меню навигации пункт IAM & admin → Service accounts (IAM и администрирование → Сервисные аккаунты), щелкните на кнопке Create Service Account (Создать сервисный аккаунт), заполните форму (название, идентификатор и описание сервисного аккаунта) и щелкните на кнопке Create (Создать), как показано на рис. 19.7. Затем этому аккаунту потребуется предоставить права доступа. Выберите роль ML Engine Developer (Разработчик ML Engine): она разрешит сервисному аккаунту вырабатывать прогнозы и не многим более. Дополнительно вы можете выдать ряду пользователей доступ к сервисному аккаунту (это полезно, когда ваш пользовательский аккаунт GCP является частью организации, и вы хотите разрешить другим пользователям в организации развертывать приложения, основанные на этом сервисном аккаунте, или самостоятельно управлять сервисным аккаунтом). Далее щелкните на кнопке Create Key (Создать ключ), чтобы экспорттировать секретный ключ сервисного аккаунта, выбрав тип ключа JSON и щелкнув на кнопке Create. Секретный ключ загрузится в виде файла JSON. Не забудьте сохранить его в надежном месте!

Замечательно! Теперь давайте напишем небольшой сценарий, который будет запрашивать службу прогнозирования.



*Рис. 19.7. Создание нового сервисного аккаунта в Google IAM*

Для упрощения доступа к своим службам Google предоставляет несколько библиотек.

### *Клиентская библиотека для API-интерфейса Google*

Довольно тонкий уровень, расположенный поверх OAuth 2.0 (<https://oauth.net/>), для аутентификации и REST. Эту библиотеку можно использовать со всеми службами GCP, включая платформу AI Platform. Ее можно установить с применением pip: библиотека называется `google-api-python-client`.

### *Клиентские библиотеки для инфраструктуры Google Cloud*

Они являются чуть более высокоуровневыми: каждая предназначена для отдельной службы, такой как GCS, Google BigQuery, Google Cloud Natural Language и Google Cloud Vision. Все эти библиотеки могут быть установлены с использованием pip (скажем, клиентская библиотека для GCS называется `google-cloud-storage`). Когда для заданной службы доступна специальная клиентская библиотека, то ее рекомендуется применять вместо клиентской библиотеки для API-интерфейса Google, т.к. она реализует всю установившуюся практику, и часто будет использовать gRPC, а не REST, чтобы улучшить производительность.

На момент написания главы клиентская библиотека для платформы AI Platform отсутствовала, так что мы будем применять клиентскую библиотеку для API-интерфейса Google. Библиотеке потребуется секретный ключ сервисного аккаунта; вы можете сообщить ей, где он находится, устанавливая переменную среды `GOOGLE_APPLICATION_CREDENTIALS` либо перед запуском сценария, либо внутри самого сценария:

```
import  
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] =  
    "my_service_account_key.json"
```



Если вы развертываете свое приложение на виртуальной машине в Google Cloud Engine (GCE), внутри контейнера, использующего Google Cloud Kubernetes Engine, как веб-приложение в Google Cloud App Engine или как микро-службу в Google Cloud Functions, но переменная среды `GOOGLE_APPLICATION_CREDENTIALS` не установлена, тогда библиотека будет применять стандартный сервисный аккаунт для службы хостинга (например, стандартный сервисный аккаунт GCE, если ваше приложение выполняется в GCE).

Затем нужно создать объект ресурса, который будет выступать в качестве оболочки для доступа к службе прогнозирования<sup>7</sup>:

```
import  
project_id = "onyx-smoke-242003"      # укажите здесь идентификатор  
                                         # своего проекта  
model_id = "my_mnist_model"  
model_path = "projects/{}/models/{}".format(project_id, model_id)  
ml_resource = googleapiclient.discovery.build("ml", "v1").projects()
```

Обратите внимание, что к `model_path` можно добавить `/versions/0001` (или другой номер версии) с целью указания версии модели, которую желательно запрашивать: это полезно для проведения экспериментов А/Б или тестирования новой версии на небольшой группе пользователей до открытия к ней широкого доступа. Далее давайте напишем маленькую функцию, которая будет использовать объект ресурса для вызова службы прогнозирования и получения прогнозов:

<sup>7</sup> Если вы получаете сообщение об ошибке, уведомляющее о том, что модуль `google.appengine` не был найден, тогда укажите `cache_discovery=False` в вызове метода `build()`; см. <https://stackoverflow.com/q/55561354>.

```
def predict(X):
    input_data_json = {"signature_name": "serving_default",
                       "instances": X.tolist()}
    request = ml_resource.predict(name=model_path, body=input_data_json)
    response = request.execute()
    if "error" in response:
        raise RuntimeError(response["error"])
    return np.array([pred[output_name]
                    for pred in response["predictions"]])
```

Функция принимает массив NumPy, содержащий входные изображения, и подготавливает словарь, который клиентская библиотека преобразует в формат JSON (как мы поступали ранее). Затем она подготовит запрос прогноза и выполнит его; если ответ содержит ошибку, то функция генерирует исключение, а иначе извлекает прогнозы для каждого образца и помещает их в массив NumPy. Посмотрим, работает ли все, как было задумано:

```
>>> Y_probas = predict(X_new)
>>> np.round(Y_probas, 2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. ],
       [0. , 0. , 0.99, 0.01, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0.96, 0.01, 0. , 0. , 0. , 0. , 0.01, 0.01, 0. ]])
```

Да! Теперь у вас есть удачно сделанная служба прогнозирования, которая функционирует в облаке и способна автоматически масштабироваться для любого значения QPS, к тому же ее можно защищенным образом запрашивать откуда угодно. Более того, когда вы ею не пользуетесь, она почти ничего не стоит: вы будете платить лишь несколько центов в месяц за гигабайт, занимаемый в GCS. И вдобавок вы можете получать подробные журналы и метрики с применением Google Stackdriver (<https://cloud.google.com/stackdriver/>).

Но что, если вы хотите развернуть свою модель в мобильном приложении или на встроенном устройстве?

## Развертывание модели на мобильном или встроенном устройстве

Если вам необходимо развернуть свою модель на мобильном или встроенном устройстве, то крупная модель просто может требовать слишком долгого времени на загрузку и расходовать чересчур много памяти и ресурсов центрального процессора. Вместе все это приведет к тому, что ваше прило-

жение перестанет отвечать на запросы, нагреет устройство и разрядит его аккумулятор. Чтобы избежать ситуации подобного рода, вы должны создать мобильную, облегченную и эффективную модель, не жертвуя значительной долей ее точности. Библиотека TFLite (<https://tensorflow.org/lite>) предлагает несколько инструментов<sup>8</sup>, которые помогают развертывать модели на мобильных и встроенных устройствах и преследуют три главных цели:

- уменьшение размера модели для сокращения времени загрузки и объема потребляемой памяти;
- снижение количества вычислений, требующихся для каждого прогноза, для уменьшения задержки, использования аккумулятора и нагрева;
- подгонка модели к ограничениям, специфичным для устройства.

Чтобы уменьшить размер модели, преобразователь моделей TFLite может взять представление SavedModel и сжать его до более легковесного формата, основанного на FlatBuffers (плоские буферы; <https://google.github.io/flatbuffers/>) — эффективной межплатформенной библиотеке сериализации (немного напоминающей Protocol Buffers), которая первоначально была разработана компанией Google для создания игр. Она спроектирована так, что вы можете загружать плоские буферы в ОЗУ безо всякой предварительной обработки, сокращая время загрузки и уменьшая объем занимаемой памяти. После того, как модель загружена в мобильное или встроенное устройство, интерпретатор TFLite запустит ее для вырабатывания прогнозов. Вот как можно преобразовать представление SavedModel в плоский буфер и сохранить его в файле .tflite:

```
converter = tf.lite.TFLiteConverter.from_saved_model(saved_model_path)
tflite_model = converter.convert()
with open("converted_model.tflite", "wb") as f:
    f.write(tflite_model)
```



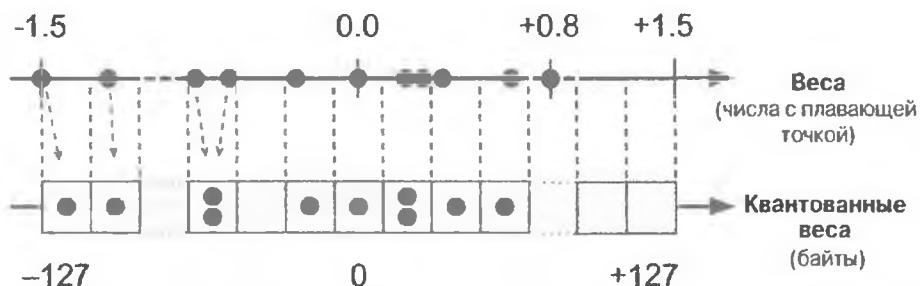
Вы также можете сохранять модель `tf.keras` прямо в плоский буфер с применением метода `from_keras_model()` класса `TFLiteConverter`.

<sup>8</sup> Также взгляните на инструмент трансформации графов TensorFlow (Graph Transform Tool; <https://homl.info/tfgtt>), предназначенный для модификации и оптимизации вычислительных графов.

Преобразователь в добавок оптимизирует модель, сжимая ее и сокращая связанную с ней задержку. Он отсекает все операции, которые не нужны для вырабатывания прогнозов (такие как операции обучения), и всякий раз, когда возможно, оптимизирует вычисления; например,  $3 \times a + 4 \times a + 5 \times a$  преобразуется в  $(3 + 4 + 5) \times a$ . Преобразователь также пытается по возможности объединять операции. Скажем, когда только возможно слои пакетной нормализации в итоге сворачиваются в операции сложения и умножения предыдущего слоя. Чтобы получить хорошее представление о том, насколько библиотека TFLite способна оптимизировать модель, загрузите одну из заранее обученных моделей TFLite (<https://hml.info/litemodels>), распакуйте архив, откройте великолепный инструмент визуализации графов Netron (<https://lutzroeder.github.io/netron/>) и загрузите файл .pb для просмотра исходной модели. Граф большой и сложный, не так ли? Затем откройте оптимизированную модель .tflite и восхищайтесь ее изяществом!

Еще один способ уменьшения размера модели ( помимо того, чтобы просто использовать менее крупные нейросетевые архитектуры) предусматривает применение чисел с меньшей шириной в битах: например, если вместо обычновенных чисел с плавающей точкой (32 бита) использовать числа, которые вдвое короче (16 битов), то размер модели сократится в 2 раза за счет (обычно небольшого) падения точности. Кроме того, обучение станет быстрее и будет расходоваться приблизительно половина объема ОЗУ графического процессора.

Преобразователь TFLite может продвинуться даже дальше, выполнив квантование весов модели до 8-битных целых чисел с фиксированной точкой! Это приводит к четырехкратному сокращению размера в сравнении с применением 32-битных чисел с плавающей точкой. Простейший подход называется *квантованием после обучения* (*post-training quantization*): он просто дискретизирует веса после обучения, используя довольно-таки базовую, но эффективную методику симметричного квантования. Методика заключается в поиске максимальной абсолютной величины веса  $m$  и отображение диапазона с плавающей точкой от  $-m$  до  $+m$  на диапазон с фиксированной точкой (целочисленный) от  $-127$  до  $+127$ . Скажем, если веса простираются от  $-1.5$  до  $+0.8$ , тогда байты  $-127$ ,  $0$  и  $+127$  будут соответствовать числам с плавающей точкой  $-1.5$ ,  $0.0$  и  $+0.8$  (рис. 19.8). Обратите внимание, что когда применяется симметричное квантование,  $0.0$  всегда отображается на  $0$  (также имейте в виду, что битовые значения от  $+68$  до  $+127$  использоваться не будут, поскольку они отображаются на числа с плавающей точкой больше  $+0.8$ ).



**Рис. 19.8. Превращение 32-битных чисел с плавающей точкой в 8-битные целые числа с применением симметричного квантования**

Чтобы выполнить такое квантование после обучения, просто добавьте `OPTIMIZE_FOR_SIZE` в список оптимизация преобразователя перед вызовом метода `convert()`:

```
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
```

Такая методика значительно снижает размер модели, поэтому загружать и сохранять ее можно будет намного быстрее. Тем не менее, во время выполнения квантованные веса перед использованием преобразуются обратно в числа с плавающей точкой (восстановленные числа с плавающей точкой не полностью идентичны исходным числам, но отличаются незначительно, так что потеря точности обычно оказывается приемлемой). Во избежание их повторного вычисления восстановленные числа с плавающей точкой каждый раз кешируются, а потому расход ОЗУ не уменьшается. И нет никакого понижения в скорости вычислений.

Самый эффективный способ уменьшить задержку и снизить энергопотребление предусматривает квантование также активаций, чтобы вычисления можно было выполнять полностью с целыми числами без потребности в любых операциях с плавающей точкой. Даже когда применяется одинаковая ширина в битах (например, 32-битные целые числа вместо 32-битных чисел с плавающей точкой), целочисленные вычисления используют меньше циклов центрального процессора, потребляют меньше энергии и производят меньше тепла. И если вы сократите ширину в битах (скажем, до 8-битных целых чисел), то можете получить огромный выигрыш в скорости. Кроме того, некоторые устройства ускорения нейронных сетей (вроде Edge TPU) способны обрабатывать только целые числа, так что полное квантование весов и активаций будет обязательным. Его можно делать после обучения. Оно требу-

ет шага калибровки для нахождения максимальной абсолютной величины активаций, поэтому вам нужно предоставить библиотеке TFLite репрезентативную выборку обучающих данных (не обязательно крупную), а она прогонит данные через модель и соберет статистические показатели, требующиеся для квантования (такой шаг обычно выполняется быстро).

Главная проблема с квантованием связана с тем, что в результате немного снижается точность: оно эквивалентно добавлению шума к весам и активациям. Если точность падает слишком сильно, тогда вам может потребоваться применять обучение с учетом квантования (*quantization-aware training*). Это означает добавление в модель фиктивных операций квантования, чтобы она могла научиться игнорировать шум от квантования во время обучения; финальные веса станут более устойчивыми к квантованию. Вдобавок шаг калибровки может быть предпринят автоматически на стадии обучения, что упростит весь процесс.

Я объяснял основные концепции TFLite, но путь к кодированию мобильных приложений или встраиваемых программ требует написания отдельной книги. К счастью, такая книга существует: если вы хотите узнать больше о построении приложений TensorFlow для мобильных и встроенных устройств, то почитайте книгу O'Reilly под названием *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers* (TinyML: машинное обучение с помощью TensorFlow на Arduino и сверхмалых мощных микроконтроллерах; <https://homl.info/tinyml>), написанную Питом Уорденом (он возглавляет команду разработчиков TFLite) и Дениэлом Ситунаяке.

## TensorFlow в браузере

Что, если вы хотите использовать свою модель на веб-сайте, запуская ее прямо в браузере пользователя? Это может быть полезно во многих сценариях, примеры которых приведены ниже.

- Когда ваше веб-приложение часто применяется в ситуациях, где подключение пользователя является нестабильным или медленным (скажем, веб-сайт для путешественников), так что запуск модели непосредственно на клиентской стороне оказывается единственным способом сделать ваш веб-сайт надежным.

- Когда необходимо, чтобы ответы вашей модели были как можно более быстрыми (например, для онлайновой игры). Устранение потребности запрашивать сервер для выработывания прогнозов определенно уменьшит задержку и сделает веб-сайт более отзывчивым.
- Когда веб-служба выпускает прогнозы на основе каких-то конфиденциальных пользовательских данных, и вы хотите защитить приватность пользователя, делая прогнозы на клиентской стороне, чтобы конфиденциальные данные не покидали пределы машины пользователя<sup>9</sup>.

Для всех описанных сценариев вы можете экспортировать модель в специальный формат, который способна загружать JavaScript-библиотека TensorFlow.js (<https://tensorflow.org/js>). Затем эта библиотека может использовать вашу модель, чтобы вырабатывать прогнозы прямо в браузере пользователя. Проект TensorFlow.js включает инструмент `tensorflowjs_converter`, позволяющий преобразовывать файл с моделью TensorFlow SavedModel или Keras в формат *TensorFlow.js Layers*: он представляет собой каталог, где содержится набор разбитых файлов весов в двоичной форме и файл `model.json`, который описывает архитектуру модели и ссылается на файлы весов. Формат *TensorFlow.js Layers* оптимизирован для эффективной загрузки через веб-сеть. Пользователи могут загружать модель и вырабатывать прогнозы в браузере с применением библиотеки TensorFlow.js. Вот фрагмент кода, который даст вам представление о том, как выглядит API-интерфейс на JavaScript:

```
import * as tf from '@tensorflow/tfjs';
const model =
  await tf.loadLayersModel('https://example.com/tfjs/model.json');
const image = tf.fromPixels(webcamElement);
const prediction = model.predict(image);
```

И снова, чтобы оценить данную тему надлежащим образом, потребуется отдельная книга. Если вы желаете лучше узнать TensorFlow.js, тогда почитайте книгу O'Reilly под названием *Practical Deep Learning for Cloud, Mobile, and Edge* (Практическое глубокое изучение для облака, мобильных и конечных устройств; <https://hml.info/tfjsbook>), написанную Анирудхом Коулом и др.

<sup>9</sup> Если вас интересует эта тема, тогда взгляните на *федеративное обучение (federated learning)* (<https://tensorflow.org/federated>).

## Использование графических процессоров для ускорения вычислений

В главе 11 мы обсуждали несколько методик, способных значительно ускорить обучение: лучшая инициализация весов, пакетная нормализация, более сложно устроенные оптимизаторы и т.д. Но даже с помощью всех приведенных методик обучение крупной нейронной сети на одной машине с единственным ЦП может занимать дни, если не недели.

В этом разделе мы выясним, как ускорять модели за счет применения ГП. Мы также посмотрим, каким образом распределять вычисление между устройствами, включая ЦП и множество плат ГП (рис. 19.9). Пока мы будем запускать все на одной машине, но позже в главе обсудим способы распределения вычислений между несколькими серверами.

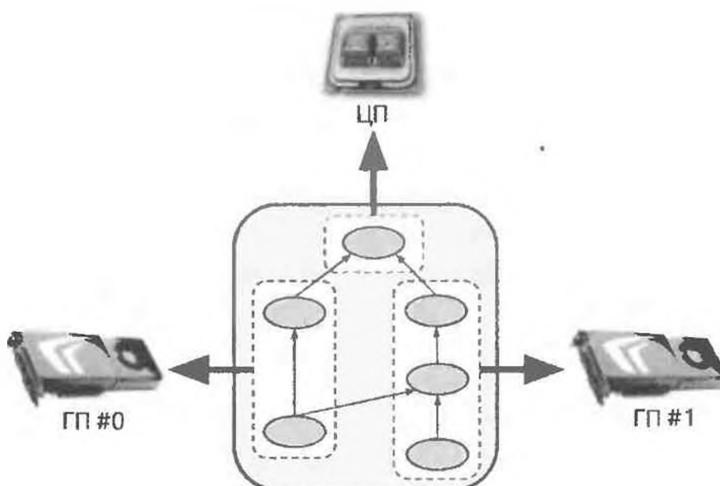


Рис. 19.9. Выполнение графа TensorFlow на множестве устройств параллельно

Вместо того чтобы ждать завершения алгоритма обучения долгие дни или недели, благодаря ГП ожидание может быть сведено всего к нескольким минутам или часам. Использование ГП не только экономит огромное количество времени, но также означает возможность более легкого экспериментирования с разнообразными моделями и частого повторного обучения моделей на свежих данных.



Часто можно получить значительный прирост производительности, просто подключая к машине платы ГП. На самом деле во многих случаях этого будет достаточно; вам вообще не придется задействовать множество машин. Например, обычно вы сможете обучить нейронную сеть быстрее с применением четырех ГП на единственной машине, чем восьми ГП на нескольких машинах, из-за дополнительной задержки, связанной с взаимодействием через сеть в многомашинной среде. Аналогично использовать один мощный ГП часто предпочтительнее множества менее быстрых ГП.

Первый шаг заключается в том, чтобы заполучить в свои руки ГП. Есть два варианта: приобрести собственный ГП (или несколько) или применить оснащенные ГП виртуальные машины в облаке. Давайте начнем с первого варианта.

## Получение собственного графического процессора

Если вы решили приобрести плату ГП, тогда потратьте какое-то время на то, чтобы сделать правильный выбор. Тим Деттмерс сделал великолепную публикацию в своем блоге (<https://homl.info/66>), которая поможет с выбором, и он достаточно регулярно ее обновляет: я настоятельно рекомендую внимательно прочитать ее. На момент написания главы библиотека TensorFlow поддерживала только платы Nvidia с возможностью CUDA Compute Capability 3.5+ (<https://homl.info/cudagpus>), а также тензорные процессоры Google, но может быть расширена для поддержки других производителей. Кроме того, хотя тензорные процессоры в настоящее время доступны только на платформе GCP, весьма вероятно, что в ближайшем будущем станут продаваться платы, подобные тензорным процессорам, и TensorFlow может их поддерживать. Короче говоря, проконсультируйтесь с документацией по TensorFlow (<https://tensorflow.org/install>), чтобы посмотреть, какие устройства поддерживаются на данном этапе.

В случае выбора платы ГП производства Nvidia вам придется установить подходящие драйверы Nvidia и несколько библиотек Nvidia<sup>10</sup>. В их число входят библиотека CUDA (*Compute Unified Device Architecture — архитектура устройств для унифицированных вычислений*), которая позволяет разработчикам использовать ГП, поддерживающие CUDA, для всех видов вычисле-

<sup>10</sup> За подробными и актуальными инструкциями по установке обращайтесь в документацию, т.к. они довольно часто меняются.

ний (не только для ускорения обработки графики), и библиотека cuDNN (*CUDA Deep Neural Network* — глубокая нейронная сеть CUDA), ускоренная с помощью ГП библиотека примитивов для сетей DNN. Библиотека cuDNN предлагает оптимизированные реализации общих вычислений DNN, такие как слои активации, нормализация, прямые и обратные свертки и организация пула (см. главу 14). Она является частью 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. 19.10).

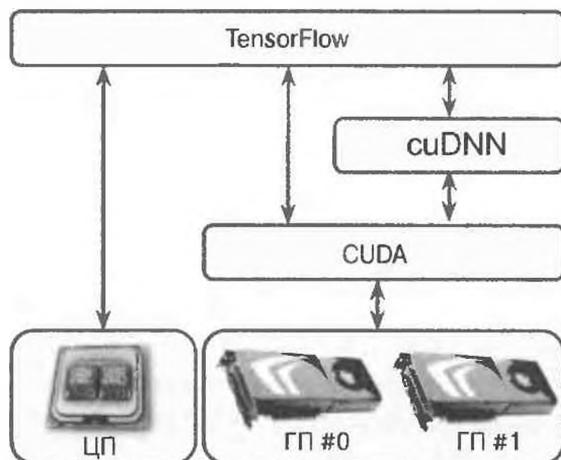


Рис. 19.10. Библиотека TensorFlow использует CUDA и cuDNN для управления платами ГП и ускорения сетей DNN

После установки платы (или плат) ГП вместе со всеми обязательными драйверами и библиотеками вы можете воспользоваться командой `nvidia-smi` для проверки правильности установки CUDA. Она выводит список доступных плат ГП, а также процессов, выполняющихся на каждой плате:

```
$ nvidia-smi
Sun Jun 2 10:05:22 2019
+-----+
| NVIDIA-SMI 418.67      Driver Version: 410.79      CUDA Version: 10.0 |
|-----+-----+-----+
| GPU  Name      Persistence-M| Bus-Id      Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|-----+-----+-----+
|  0  Tesla T4           Off  | 00000000:00:04.0 Off |                    0 |
| N/A   61C    P8    17W /  70W |        0MiB / 15079MiB |      0%     Default |
+-----+
```

			GPU Memory	
			Usage	
=====				
No running processes found				
Выполняющиеся процессы не обнаружены				
=====				

На момент написания главы вам также потребуется установить версию TensorFlow для ГП (т.е. библиотеку `tensorflow-gpu`); однако, продолжается работа по созданию единой процедуры установки для машин с только ЦП и машин с ГП, поэтому загляните в документацию по установке, чтобы выяснить, какая библиотека должна быть установлена. В любом случае, поскольку корректная установка каждой обязательной библиотеки оказывается несколько долгой и сложной (а если вы не установите правильные версии библиотек, то все пойдет не так), TensorFlow предлагает образ Docker со всем необходимым внутри. Тем не менее, для того, чтобы контейнер Docker имел доступ к ГП, вам понадобится установить драйверы Nvidia на машине хоста.

Для проверки, видит ли библиотека TensorFlow имеющиеся ГП, прогоните следующие тесты:

```
>>> import tensorflow as tf
>>> tf.test.is_gpu_available()
True
>>> tf.test.gpu_device_name()
'/device:GPU:0'
>>> tf.config.experimental.list_physical_devices(device_type='GPU')
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Функция `is_gpu_available()` проверяет, доступен ли, по меньшей мере, один ГП. Функция `gpu_device_name()` выдает имя первого ГП: по умолчанию операции будут выполняться на этом ГП.

Функция `list_physical_devices()` возвращает список всех устройств ГП (только один в приведенном примере)<sup>11</sup>.

---

<sup>11</sup> Многие примеры кода в главе используют экспериментальные API-интерфейсы. В будущих версиях они с высокой вероятностью переместятся в основной API-интерфейс. Таким образом, если экспериментальная функция отказывается работать, тогда попробуйте просто удалить слово `experimental` и надо надеяться она заработает. Если же нет, то возможно API-интерфейс слегка изменился; просмотрите тетрадь Jupyter, где гарантированно содержится корректный код.

А что, если вы не хотите тратить время и деньги на приобретение собственной платы ГП? Тогда просто воспользуйтесь виртуальной машиной, оснащенной ГП, в облаке!

## Использование виртуальной машины, оснащенной графическим процессором

В настоящее время все крупные облачные платформы предлагают виртуальные машины, оснащенные ГП, среди которых есть заранее сконфигурированные варианты со всеми необходимыми драйверами и библиотеками (включая TensorFlow). Облачная платформа Google (GCP) навязывает разнообразные квоты относительно ГП, как в масштабе всего мира, так и в зависимости от региона: вы не можете просто создать тысячи виртуальных машин, оснащенных ГП, без предварительного санкционирования со стороны Google<sup>12</sup>. По умолчанию мировая квота ГП составляет ноль, так что вы не можете применять какие-либо виртуальные машины, оснащенные ГП. Следовательно, самое первое, что вам придется сделать — запросить более высокую мировую квоту. Откройте меню навигации в консоли GCP и выберите пункт IAM & admin → Quotas (IAM и администрирование → Квоты). Щелкните на списке Metric (Показатель), щелкните на ссылке None (Очистить) для снятия отметки со всех показателей; выполните поиск GPU и выберите вариант GPUs (all regions) (ГП (все регионы)), чтобы увидеть соответствующую квоту. Если значение квоты равно нулю (или недостаточно для ваших потребностей), тогда отметьте флагок рядом с ним (он должен быть единственным выбранным) и щелкните на кнопке Edit quotas (Редактировать квоту). Предоставьте запрошенную информацию и щелкните на кнопке Submit request (Отправить запрос). Может пройти несколько часов (а то и несколько дней), пока ваш запрос квоты будет обработан и (как правило) принят. По умолчанию существует также квота в размере одного ГП для региона и для типа ГП. Вы можете запрашивать увеличение и таких квот: щелкните на Metric, щелкните на None, поищите GPU и выберите желаемый тип ГП (скажем, NVIDIA P4 GPUs). Затем щелкните на списке Location (Местоположение), щелкните на ссылке None (Очистить) для снятия отметки со всех местоположений и щелкните на желаемом местоположении; от-

<sup>12</sup> Предположительно эти квоты предназначены для того, чтобы остановить плохих парней, у которых может возникнуть соблазн использовать платформу GCP с похищенными кредитными картами для майнинга криптовалют.

метьте флажки рядом с квотами, которые хотите изменить, и щелкните на кнопке *Edit quotas* (Редактировать квоту), чтобы сформировать запрос.

После утверждения ваших запросов квот ГП вы можете моментально создать виртуальную машину, оснащенную одним или большим числом ГП, используя образы виртуальных машин для глубокого обучения (*Deep Learning VM Images*) платформы AI Platform инфраструктуры Google Cloud: перейдите по ссылке <https://console.cloud.google.com/compute/instances>, щелкните на кнопке *View Console* (Просмотреть консоль), щелкните на *Launch on Compute Engine* (Запускать на Compute Engine) и заполните форму конфигурации виртуальной машины. Обратите внимание, что в некоторых местоположениях присутствуют не все типы ГП, а в некоторых ГП нет вообще (измените местоположение, чтобы увидеть типы доступных ГП, если таковые имеются). Обязательно выберите TensorFlow 2.0 в качестве фреймворка и отметьте флажок *Install NVIDIA GPU driver automatically on first startup* (Установить драйвер ГП NVIDIA автоматически при первом запуске). Неплохо также отметить флажок *Enable access to JupyterLab via URL instead of SSH* (Разрешить доступ к JupyterLab через URL вместо SSH): это позволит очень легко запускать тетрадь Jupyter, выполняющуюся на данной оснащенной ГП виртуальной машине, на базе JupyterLab (альтернативный веб-интерфейс для запуска тетрадей Jupyter). После создания виртуальной машины прокрутите меню навигации вниз до раздела *Artificial Intelligence* (Искусственный интеллект) и выберите пункт *AI Platform → Notebooks* (AI Platform → Экземпляр Notebook). Как только экземпляр Notebook появится в списке (что может занять несколько минут, поэтому изредка щелкайте на кнопке обновления текущей страницы, пока он не появится), щелкните на ссылке *Open JupyterLab* (Открыть JupyterLab). В результате пользовательский интерфейс JupyterLab запустится на виртуальной машине и подключит к себе ваш браузер. На этой виртуальной машине вы сможете создавать тетради и выполнять любой желаемый код, извлекая преимущества из его ГП!

Но если вы хотите только прогнать несколько коротких тестов или просто совместно использовать тетради со своими коллегами, тогда вам нужно испытать среду Colaboratory.

## Среда Colaboratory

Самый простой и дешевый способ получить доступ к виртуальной машине, оснащенной ГП, предусматривает применение среды *Colaboratory*

(или для краткости *Colab*). Она бесплатна! Достаточно перейти по ссылке <https://colab.research.google.com/> и создать новую тетрадь Python 3, что в итоге приводит к созданию тетради Jupyter на вашем Google Диске (в качестве альтернативы вы могли бы открыть любую тетрадь из GitHub, из Google Диска или даже загрузить собственные тетради). Пользовательский интерфейс Colab похож на интерфейс Jupyter, но вы можете разделять и использовать тетради подобно обычным документам Google; есть также несколько других незначительных отличий (например, можно создавать удобные графические элементы с применением специальных комментариев в коде).

Когда вы открываете тетрадь Colab, она запускается на бесплатной виртуальной машине Google, выделенной для вас, которая называется *исполняющей средой Colab (Colab Runtime)*. По умолчанию исполняющая среда предназначена только для ЦП, но вы можете изменить ее тип, выбрав пункт меню *Runtime → Change runtime type* (Исполняющая среда → Изменить тип исполняющей среды), указав GPU (Графический процессор) в раскрывающемся списке *Hardware accelerator* (Аппаратный ускоритель) и щелкнув на кнопке *Save* (Сохранить). На самом деле вы могли бы даже выбрать тензорный процессор! (Да, как ни странно, вы можете бесплатно использовать тензорный процессор; однако, речь о тензорных процессорах пойдет позже в главе, так что пока просто выберите ГП.)

Если вы запускаете множество тетрадей Colab с применением исполняющей среды того же самого типа (рис. 19.11), тогда они будут использовать одну и ту же исполняющую среду Colab. Таким образом, когда одна тетрадь записывает в файл, то остальные будут иметь возможность читать из этого файла. Важно понимать последствия в плане безопасности: запуск недежной тетради Colab, написанной хакером, может привести к тому, что она прочитает конфиденциальные данные, производимые другими тетрадями, и раскроет их хакеру. Если данные включают секретные ключи доступа для каких-то ресурсов, тогда хакер добудет доступ к таким ресурсам. Более того, если вы установили библиотеку в исполняющей среде Colab, то другие тетради тоже будут иметь установленную библиотеку. В зависимости от того, что вы хотите делать, это может быть замечательным или раздражающим (скажем, вы не сумеете без труда применять различные версии той же библиотеки в разных тетрадях Colab).



*Рис. 19.11. Исполняющие среды Colab и тетради*

Со средой Colab связаны ограничения: как сформулировано в часто задаваемом вопросе, “Среда Colaboratory предназначена для интерактивного использования. Долго выполняющиеся фоновые вычисления, особенно на ГП, могут быть остановлены. Пожалуйста, не применяйте Colaboratory для майнинга криптовалют”. Веб-интерфейс будет автоматически отключаться от исполняющей среды Colab, если вы оставите его без присмотра на какое-то время (~30 минут). Когда вы снова подключитесь к исполняющей среде Colab, она может оказаться сброшенной, поэтому удостоверьтесь, что всегда загружаете любые данные, которые вам интересны. Даже если вы никогда не отключаетесь, то все равно исполняющая среда Colab автоматически прекратит работу через 12 часов, т.к. она не предназначена для длительных вычислений. Невзирая на такие ограничения, она является фантастическим инструментом, который позволяет легко прогонять тесты, быстро получать результаты и сотрудничать с коллегами.

## Управление оперативной памятью графического процессора

По умолчанию библиотека TensorFlow автоматически захватывает все ОЗУ во всех доступных ГП, когда вы первый раз запускаете вычисление. Она поступает так, чтобы ограничить фрагментацию оперативной памяти графических процессоров. Таким образом, если вы попытаетесь запустить вторую программу TensorFlow (или любую программу, которой требуется ГП), то ОЗУ быстро закончится. Это происходит не так часто, как вы могли по-

думать, потому что на машине чаще всего выполняется единственная программа TensorFlow: обычно сценарий обучения, узел TF Serving или тетрадь Jupyter. Если по какой-то причине вам необходимо запускать множество программ (скажем, для обучения двух разных моделей параллельно на той же самой машине), тогда понадобится распределять оперативную память графического процессора между процессами более равномерно.

При наличии на машине множества плат ГП простое решение предусматривает назначение каждой из них одиночного процесса, для чего можно установить переменную среды CUDA\_VISIBLE\_DEVICES так, чтобы каждый процесс видел только надлежащую плату (платы) ГП. Также установите переменную среды CUDA\_DEVICE\_ORDER в PCI\_BUS\_ID для гарантии того, что каждый идентификатор всегда относится к той же самой плате ГП. Например, если у вас есть четыре платы ГП, то вы могли бы запустить две программы, назначив каждой из них два ГП, за счет выполнения команд следующего вида в двух отдельных окнах терминала:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1  
python3 program_1.py
```

# и в другом терминале:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2  
python3 program_2.py
```

Затем программа 1 будет видеть только платы ГП 0 и 1, именованные как /gpu:0 и /gpu:1, а программа 2 — только платы ГП 2 и 3, именованные как /gpu:1 и /gpu:0 ( обратите внимание на порядок). Все будет работать хорошо (рис. 19.12). Разумеется, вы также можете определить такие переменные в Python, устанавливая `os.environ["CUDA_DEVICE_ORDER"]` и `os.environ["CUDA_VISIBLE_DEVICES"]`, при условии, что делаете это перед использованием TensorFlow.

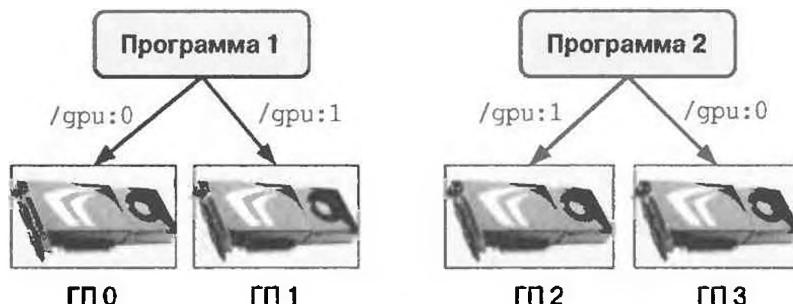


Рис. 19.12. Каждая программа получает в свое распоряжение два ГП

Еще один вариант заключается в том, чтобы сообщить TensorFlow о необходимости захватывать лишь конкретный объем оперативной памяти ГП, что должно делаться немедленно после импортирования TensorFlow. Например, чтобы заставить TensorFlow захватывать только 2 гибайта ОЗУ в каждом ГП, вам потребуется создать *виртуальное устройство ГП* (также называемое *логическим устройством ГП*) для каждого физического устройства ГП и установить его предел памяти в 2 гибайта (т.е. 2048 мебайтов):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):  
    tf.config.experimental.set_virtual_device_configuration(gpu,  
        [tf.config.experimental.VirtualDeviceConfiguration(  
            memory_limit=2048)])
```

Теперь (предполагая наличие четырех ГП, каждый с минимум 4 гибайта ОЗУ) две программы вроде приведенной выше могут выполняться параллельно, причем каждая задействует все четыре платы ГП (рис. 19.13).

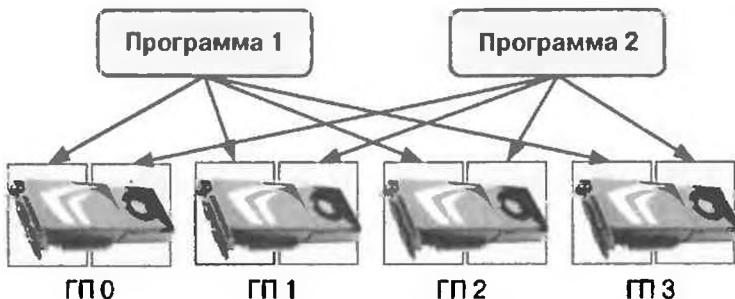


Рис. 19.13. Каждая программа получает в свое распоряжение все четыре ГП, но только 2 гибайта ОЗУ в каждом ГП

Запустив команду `nvidia-smi` во время функционирования программ, вы должны увидеть, что каждый процесс удерживает 2 гибайта ОЗУ на каждой плате:

```
$ nvidia-smi  
[...]  
+-----+  
| Processes:                               GPU Memory |  
| GPU      PID  Type  Process name        Usage     |  
+=====+=====+=====+=====+=====+  
| 0        2373  C    /usr/bin/python3    2241MiB  |  
| 0        2533  C    /usr/bin/python3    2241MiB  |  
| 1        2373  C    /usr/bin/python3    2241MiB  |  
| 1        2533  C    /usr/bin/python3    2241MiB  |  
[...]
```

Очередной вариант предусматривает указание TensorFlow о том, что память должна захватываться, только когда она нужна (это также необходимо делать немедленно после импортирования TensorFlow):

```
for gpu in tf.config.experimental.list_physical_devices("GPU"):  
    tf.config.experimental.set_memory_growth(gpu, True)
```

По-другому это можно сделать, установив переменную среды `TF_FORCE_GPU_ALLOW_GROWTH` в `true`. В результате библиотека TensorFlow никогда не будет освобождать память, после того, как захватила ее (опять-таки во избежание фрагментации памяти), кроме ситуации, когда программа завершается. В таких условиях может быть труднее гарантировать детерминированное поведение (скажем, одна программа может потерпеть аварийный отказ из-за того, что потребление памяти другой программой резко увеличилось), поэтому в производственной среде, вероятно, имеет смысл придерживаться одного из предшествующих вариантов. Тем не менее, в ряде случаев данный вариант очень удобен: например, когда вы применяете машину для запуска множества тетрадей Jupyter, часть которых использует TensorFlow. Именно потому переменная среды `TF_FORCE_GPU_ALLOW_GROWTH` устанавливается в `true` в исполняющей среде Colab.

Наконец, в определенных ситуациях может требоваться разделение ГП на два и более виртуальных ГП — скажем, если вы хотите протестировать алгоритм распределения (это удобный способ прогнать примеры кода из оставшейся части главы даже при наличии единственного ГП, как в исполняющей среде Colab). Следующий код разделяет первый ГП на два виртуальных устройства, каждое с 2 гигабайтами ОЗУ (код должен выполняться немедленно после импортирования TensorFlow):

```
physical_gpus = tf.config.experimental.list_physical_devices("GPU")  
tf.config.experimental.set_virtual_device_configuration(  
    physical_gpus[0],  
    [tf.config.experimental.VirtualDeviceConfiguration(  
        memory_limit=2048),  
     tf.config.experimental.VirtualDeviceConfiguration(  
        memory_limit=2048)])
```

Затем два виртуальных устройства будут называться `/gpu:0` и `/gpu:1`, причем в каждое из них можно помещать операции и переменные, как если бы действительно существовали два независимых ГП. А теперь посмотрим, каким образом библиотека TensorFlow решает, на каких устройствах размещать переменные и выполнять операции.

## Размещение операций и переменных на устройствах

В официальном описании TensorFlow (<https://hml.info/67>)<sup>13</sup> представлен дружественный алгоритм динамического размещения (*dynamic placer*), 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 ОЗУ в каждом устройстве, коммуникационная задержка при передаче данных в устройства и из них, а также подсказки и ограничения со стороны пользователя. На практике этот алгоритм оказался менее эффективным, чем небольшой набор правил размещения, задаваемый пользователем, так что команда разработчиков TensorFlow в итоге отказалась от динамического размещения.

Однако в целом `tf.keras` и `tf.data` выполняют хорошую работу по размещению операций и переменных там, где они будут на своем месте (скажем, интенсивные вычисления в ГП и предварительную обработку данных в ЦП). Но вы также можете вручную размещать операции и переменные на каждое устройство, если желаете иметь больший контроль.

- Как только что упоминалось, в общем случае вы захотите поместить операции предварительной обработки данных в ЦП и операции нейронных сетей в ГП.
- ГП обычно обладают довольно ограниченной полосой пропускания, поэтому важно избегать излишней передачи данных в ГП и из них.
- Нарращивать объем оперативной памяти ЦП машины легко и достаточно дешево, поэтому обычно ее предостаточно, тогда как оперативная память ГП встроена в плату ГП: она является дорогостоящим и потому ограниченным ресурсом. Таким образом, если переменная не нужна в последующих нескольких шагах обучения, то она вероятно должна быть размещена в ЦП (например, наборы данных, как правило, помещаются в ЦП).

По умолчанию все переменные и все операции будут помещаться в первый ГП (по имени `/gpu:0`), исключая переменные и операции, которые

<sup>13</sup> Мартин Абади и др., *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems* (TensorFlow: крупномасштабное машинное обучение в гетерогенных распределенных системах), официальное описание Google Research (2015 г.).

не имеют ядра ГП<sup>14</sup>: они помещаются в ЦП (по имени /сри:0). Атрибут device тензора или переменной сообщает, где она была размещена<sup>15</sup>:

```
>>> a = tf.Variable(42.0)
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable(42)
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

Пока что можете благополучно игнорировать префикс /job:localhost/replica:0/task:0 (он позволяет размещать операции на других машинах, когда применяется кластер TensorFlow; позже в главе мы обсудим задания, реплики и задачи). Легко заметить, что первая переменная была помещена в ГП 0, который является стандартным устройством. Тем не менее, вторая переменная разместилась в ЦП: причина связана с отсутствием ядер ГП для целочисленных переменных (или для операций, затрагивающих целочисленные тензоры), так что библиотека TensorFlow возвратилась к ЦП.

Если вы хотите поместить операцию на устройство, отличающееся от стандартного, тогда используйте контекст `tf.device()`:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable(42.0)
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```



Центральный процессор всегда трактуется как одиночное устройство (/сри:0), даже когда на машине имеется множество ядер ЦП. Любая операция, размещенная в ЦП, может выполняться параллельно на множестве ядер, если для нее предусмотрено многопоточное ядро.

Если вы явно попытаетесь поместить операцию или переменную в несуществующее устройство, либо для операции или переменной отсутствует ядро, тогда генерируется исключение. Однако в некоторых случаях пред-

<sup>14</sup> Как объяснялось в главе 12, ядро представляет собой реализацию переменной или операции для конкретного типа данных и типа устройства. Скажем, есть ядро ГП для операции `float32tf.matmul()`, но нет ядра ГП для `int32tf.matmul()` (имеется только ядро ЦП).

<sup>15</sup> Можно также вывести все размещения на устройствах с помощью `tf.debugging.set_log_device_placement(True)`.

почтильнее возвратиться к ЦП; например, если ваша программа способна выполнятся на машинах, имеющих только ЦП, и на машинах, оснащенных ГП, то у вас может возникнуть желание, чтобы библиотека TensorFlow игнорировала `tf.device("/gpu:*)` на машинах, где есть только ЦП. Для этого можно вызвать `tf.config.set_soft_device_placement(True)` сразу после импортирования TensorFlow: когда запрос размещения терпит неудачу, библиотека TensorFlow возвратится к своим стандартным правилам размещения (т.е. ГП 0, если он существует и есть ядро ГП, а иначе ЦП 0).

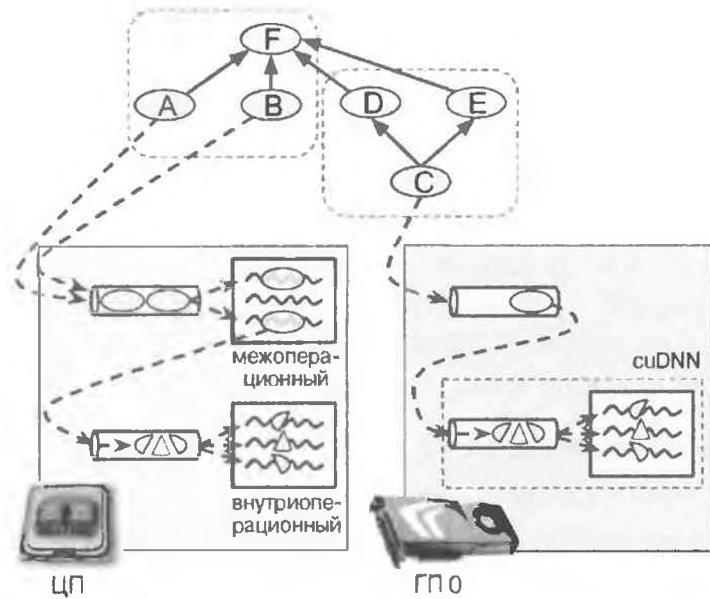
Теперь давайте выясним, как библиотека TensorFlow в точности будет выполнять все операции на множестве устройств.

## Параллельное выполнение на множестве устройств

Как было указано в главе 12, одно из преимуществ применения функций TF Function связано с параллелизмом. Посмотрим на это более внимательно. Когда библиотека TensorFlow запускает функцию TF Function, она начинает с анализа ее графа, чтобы найти список операций, которые необходимо вычислить, и подсчитывает количество зависимостей для каждой из них. Затем TensorFlow добавляет каждую операцию с нулем зависимостей (т.е. каждую исходную операцию) в очередь вычисления устройства этой операции (рис. 19.14). После вычисления операции счетчик зависимостей каждой операции, зависящей от вычисленной, декрементируется. Как только счетчик зависимостей какой-то операции достигает нуля, операция помещается в очередь вычисления данного устройства. И после того, как все узлы, которые нужны библиотеке TensorFlow, были вычислены, она возвращает их выходы.

Операции в очереди вычисления ЦП отправляются в пул потоков, называемый межоперационным пулем потоков (*inter-op thread pool*). Если ЦП имеет несколько ядер, тогда операторы фактически будут вычисляться параллельно. Есть операции, располагающие многопоточными ядрами ЦП: такие ядра расщепляют их задачи на множество подопераций, которые помещаются в другую очередь вычисления и отправляются во второй пул потоков, называемый внутриоперационным пулем потоков (*intra-op thread pool*) и разделяемый всеми многопоточными ядрами ЦП. Короче говоря, множество операций и подопераций могут вычисляться параллельно на разных ядрах ЦП.

Для ГП ситуация чуть проще. Операции в очереди вычисления ГП вычисляются последовательно. Но большинство операций имеют многопоточные ядра ГП, как правило, реализованные посредством библиотек, от которых зависит TensorFlow, наподобие CUDA и cuDNN.



*Рис. 19.14. Распараллеленное выполнение графа TensorFlow*

Эти реализации располагают собственными пулами потоков, и они обычно задействуют столько потоков ГП, сколько могут (что является причиной отсутствия потребности в межоперационном пуле потоков для ГП: каждая операция уже занимает большинство потоков ГП).

Например, на рис. 19.14 операции A, B и C являются исходными операциями, так что они могут быть выполнены немедленно. Операции A и B помещаются в ЦП, а потому посылаются в очередь выполнения ЦП, после чего отправляются в межоперационный пул потоков и незамедлительно вычисляются параллельно. Так вышло, что операция A имеет многопоточное ядро; ее вычисления расщепляются на три части, которые выполняются параллельно внутриоперационным пулом потоков. Операция C попадает в очередь выполнения ГП 0 и в рассматриваемом примере ее ядро ГП использует библиотеку cuDNN, которая управляет собственным внутриоперационным пулом потоков и запускает операцию в нескольких потоках ГП параллельно. Предположим, что операция C завершается первой. Счетчики зависимостей операций D и E декрементируются и достигают нуля, так что обе операции ставятся в очередь вычисления ГП 0 и выполняются параллельно. Обратите внимание, что C выполняется только раз, хотя от нее зависят операции D и E. Пусть операция B финиширует первой. Тогда счетчик зависимостей операции

F уменьшается с 4 до 3, а поскольку имеет ненулевое значение, операция пока не выполняется. Как только операции A, D и E завершаются, счетчик зависимостей F достигает 0, операция F помещается в очередь выполнения ЦП и выполняется. В заключение TensorFlow возвращает запрошенные выходы.

Дополнительная толика магии, предпринимаемая TensorFlow, происходит при модификации функцией `TF Function` ресурса с запоминанием состояния, такого как переменная: подобная модификация гарантирует, что порядок выполнения соответствует порядку в коде, даже если явная зависимость между операторами отсутствует. Скажем, если ваша функция `TF Function` содержит `v.assign_add(1)` и затем `v.assign(v * 2)`, то библиотека TensorFlow обеспечит выполнение указанных операций именно в таком порядке.



Вы можете контролировать количество потоков в межоперационном пуле потоков, вызывая функцию `tf.config.threading.set_inter_op_parallelism_threads()`. Для установки количества внутриоперационных пулов потоков применяйте функцию `tf.config.threading.set_intra_op_parallelism_threads()`. Это полезно, если вы не желаете, чтобы библиотека TensorFlow использовала все ядра ЦП, или хотите, чтобы она была однопоточной<sup>16</sup>.

Итак, у вас есть все, что необходимо для запуска любой операции на любом устройстве, и эксплуатации в своих интересах моци имеющихся ГП! Ниже перечислено несколько действий, которые вы могли бы предпринять.

- Вы можете обучать несколько моделей параллельно, каждую на собственном ГП: просто напишите для всех моделей сценарии обучения и запустите их параллельно, установив `CUDA_DEVICE_ORDER` и `CUDA_VISIBLE_DEVICES`, чтобы каждый сценарий видел только единственное устройство ГП. Прием прекрасно подходит для подстройки гиперпараметров, т.к. есть возможность параллельного обучения множества моделей с отличающимися значениями гиперпараметров. Если есть машина с двумя ГП, а обучение одной модели на одном ГП требует одного часа, тогда обучение двух моделей параллельно (каждой на выделенном ГП) займет всего один час. Просто!

<sup>16</sup> Это может быть полезным, когда вы хотите обеспечить идеальную воспроизводимость, как я объясняю в видеоролике <https://homl.info/repro>, основываясь на TF 1.

- Вы можете обучать модель на одном ГП и параллельно выполнять всю предварительную обработку на ЦП, применяя метод `prefetch()` набора данных<sup>17</sup> для заблаговременной подготовки следующих нескольких пакетов, так что они будут готовыми к моменту, когда потребуются ГП (см. главу 13).
- Если ваша модель принимает на входе два изображения и обрабатывает их с использованием двух сетей CNN до объединения их выходов, тогда она, вероятно, будет работать намного быстрее в случае помещения каждой сети CNN в отдельный ГП.
- Вы можете создать эффективный ансамбль: просто поместите в каждый ГП отдельную обученную модель, чтобы получать все прогнозы гораздо быстрее с целью выработывания финального прогноза.

Но что, если вы хотите обучать одиночную модель на множестве ГП?

## Обучение моделей на множестве устройств

Существуют два основных подхода к обучению одиночной модели на множестве устройств: *параллелизм модели* (*model parallelism*), когда модель расщепляется между устройствами, и *параллелизм данных* (*data parallelism*), при котором модель реплицируется на все устройства и каждая реплика обучается на подмножестве данных. Прежде чем обучать модель на множестве ГП, давайте рассмотрим оба варианта более подробно.

### Параллелизм модели

До сих пор мы обучали каждую нейронную сеть на одиночном устройстве. А что, если мы хотим обучить единственную нейронную сеть на множестве устройств? В таком случае модель потребуется разбить на отдельные порции и запускать каждую порцию на отдельном устройстве. К сожалению, параллелизм модели подобного рода оказывается очень сложным и крайне зависит от архитектуры нейронной сети. Для полно связанных сетей такой подход обычно дает немногое (рис. 19.15). Может показаться, что легкий способ расщепления модели заключается в размещении каждого слоя на отдельном устройстве, но он не работает, т.к. каждому слою придется ожидать выход-

<sup>17</sup> На момент написания главы он делал упреждающую выборку данных только в оперативную память ЦП, но вы можете использовать функцию `tf.data.experimental.prefetch_to_device()`, заставив его выбирать данные и помещать их в желаемое устройство, чтобы ГП попусту не растративал время на ожидание передачи данных.

ных данных предыдущего слоя, прежде чем он сможет делать что-либо. Так может быть имеет смысл расщепить модель по вертикали, скажем, размешая левую половину каждого слоя на одном устройстве, а правую — на другом? Чуть лучше, поскольку 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). По всей видимости, это полностью сведет на нет выгоду от параллельного вычисления, потому что коммуникации между устройствами являются медленными (особенно если устройства находятся на разных машинах).

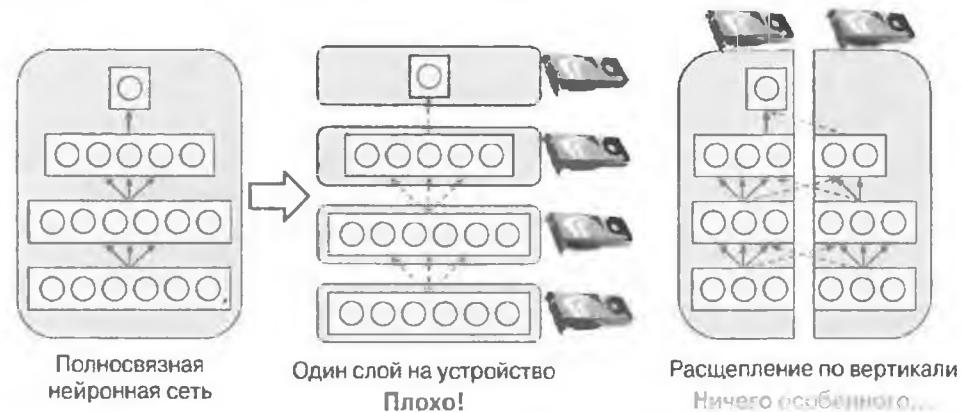
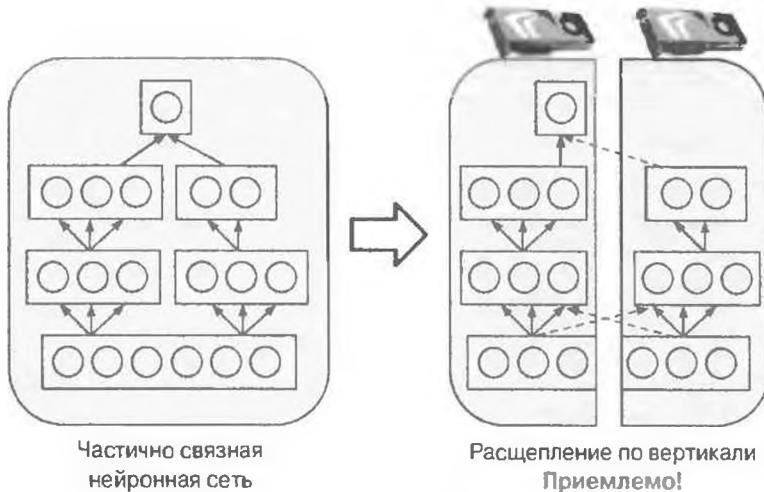


Рис. 19.15. Расщепление полносвязной нейронной сети

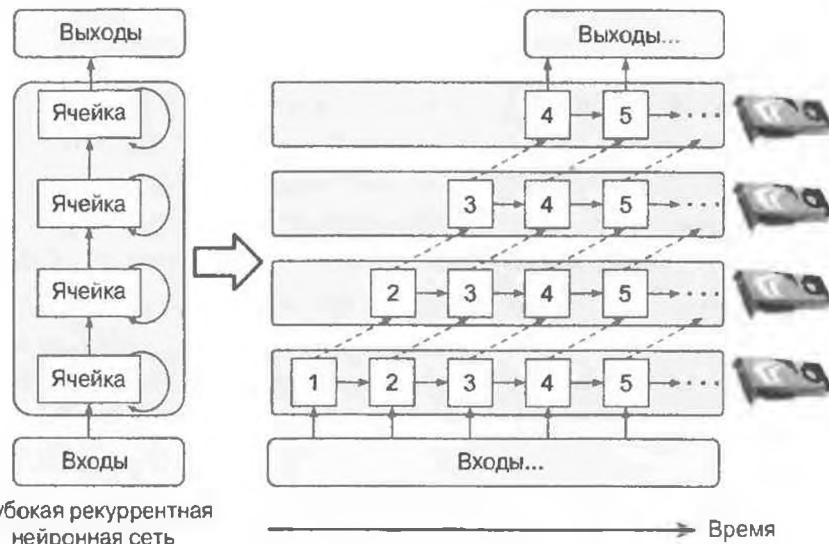
Ряд архитектур нейронных сетей, такие как сверточные нейронные сети (см. главу 14), содержат слои, которые только частично связаны с более низкими слоями, поэтому эффективное распределение порций между устройствами становится более простым (рис. 19.16).

Глубокие рекуррентные нейронные сети (см. главу 15) можно расщеплять на множество ГП чуть эффективнее. Если расщепить сеть горизонтально, размещая каждый слой на своем устройстве, и передать сети на обработку входную последовательность, тогда на первом временном шаге активным будет только одно устройство (обрабатывая первое значение последовательности), на втором временном шаге — два устройства (второй слой будет обрабатывать выход первого слоя для первого значения, а первый слой займется обработкой второго значения), а к моменту распространения сигнала до выходного слоя все устройства будут активными одновременно (рис. 19.17).



*Рис. 19.16. Расщепление частично связной нейронной сети*

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 способна (теоретически) перевешивать штраф в виде коммуникаций. Тем не менее, на практике обыкновенная стопка слоев LSTM, запущенная на единственном ГП, в действительности выполняется гораздо быстрее.



*Рис. 19.17. Расщепление глубокой рекуррентной нейронной сети*

Короче говоря, параллелизм модели может ускорить выполнение или обучение определенных видов нейронных сетей, но не всех. К тому же он требует специального внимания и подстройки вроде обеспечения того, чтобы устройства, которые больше всего нуждаются в коммуникациях, функционировали на одной машине<sup>18</sup>. Давайте рассмотрим намного более простой и в целом более эффективный вариант: параллелизм данных.

## Параллелизм данных

Еще один способ распараллеливания обучения нейронной сети предусматривает ее репликацию на каждое устройство и запуск каждого шага обучения одновременно на всех репликах, применяя для каждой реплики отличающийся мини-пакет. Затем градиенты, рассчитанные каждой репликой, усредняются, а результат используется для обновления параметров модели. Прием называется параллелизмом данных. Существует много вариантов этой идеи, и мы взглянем на самые важные из них.

### Параллелизм данных с использованием стратегии зеркального отображения

Пожалуй, наиболее простой подход предусматривает полное зеркальное отображение всех параметров модели по всем ГП и применение в точности тех же самых обновлений параметров на каждом ГП. В таком случае все реплики всегда остаются совершенно идентичными. Прием называется *стратегией зеркального отображения (mirrored strategy)* и оказывается довольно эффективным, особенно когда используется на одной машине (рис. 19.18).

Сложная часть такого подхода связана с эффективным вычислением среднего всех градиентов из всех ГП и распространение результата во все ГП. Цель достигается с использованием алгоритма *AllReduce*, относящегося к классу алгоритмов, в которых множество узлов сотрудничают для эффективного выполнения операции приведения (такой как вычисление среднего, суммы и максимума), одновременно гарантируя то, что все узлы получают тот же самый финальный результат. К счастью, как мы вскоре увидим, доступны готовые реализации таких алгоритмов.

---

<sup>18</sup> Если вы заинтересованы в дополнительном исследовании параллелизма модели, тогда испытайтe Mesh TensorFlow (<https://github.com/tensorflow/mesh>).

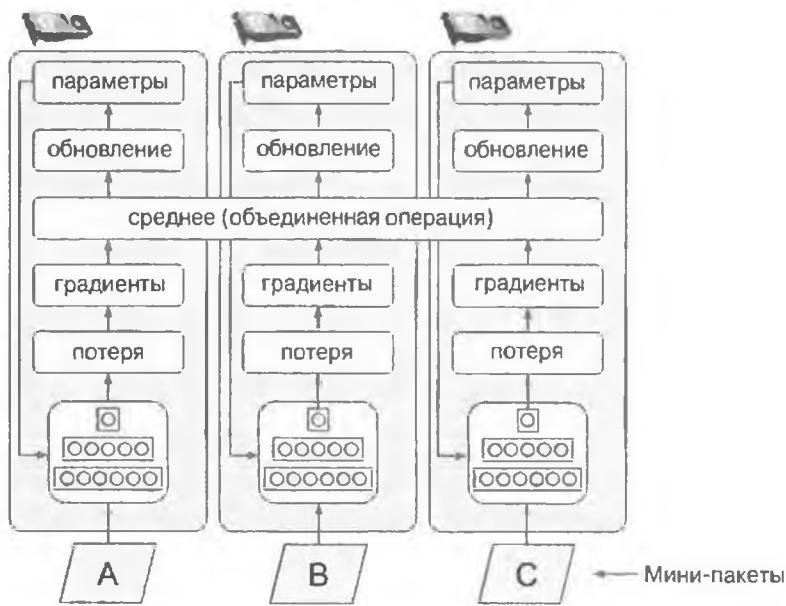


Рис. 19.18. Параллелизм данных с применением стратегии зеркального отображения

### Параллелизм данных с использованием централизованных параметров

Другой подход предполагает хранение параметров модели вне устройств ГП, выполняющих вычисления (называемых *рабочими процессорами*), например, в ЦП (рис. 19.19).

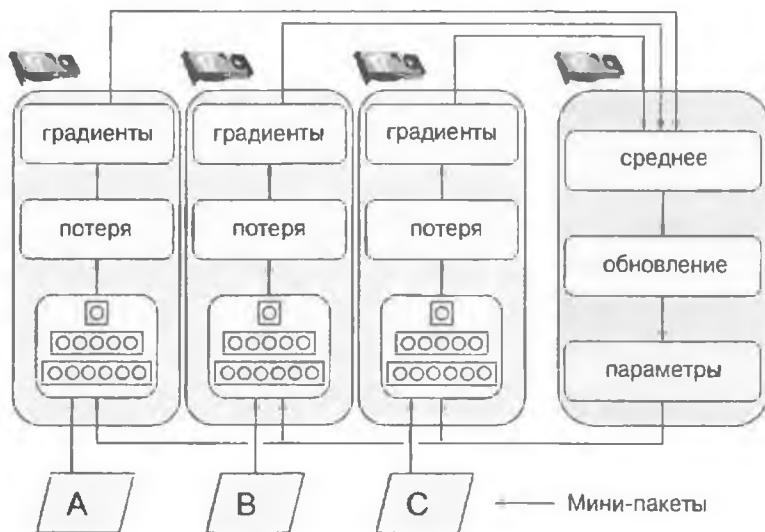


Рис. 19.19. Параллелизм данных с централизованными параметрами

В распределенной конфигурации вы можете размещать все параметры на одном или большем числе серверов, имеющих только ЦП, которые называются серверами параметров, а единственная их роль связана с хранением и обновлением параметров.

В то время как стратегия зеркального отображения навязывает синхронные обновления весов по всем ГП, такой централизованный подход разрешает синхронные или асинхронные обновления. Давайте рассмотрим достоинства и недостатки обоих вариантов.

## Синхронные обновления

При синхронных обновлениях агрегатор ожидает, пока все градиенты не станут доступными, прежде чем вычислить средние градиенты, и передает их оптимизатору, который обновит параметры модели. После того как реплика завершит вычисление своих градиентов, она обязана ожидать обновления параметров, чтобы быть в состоянии продолжить со следующим мини-пакетом. Недостаток заключается 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 на серверах параметров.



Чтобы сократить время ожидания на каждом шаге, вы могли бы проигнорировать градиенты из нескольких самых медленных реплик (как правило, ~10%). Скажем, вы могли бы запускать 20 реплик, но на каждом шаге агрегировать градиенты только из 18 наиболее быстрых реплик и просто игнорировать градиенты из оставшихся двух. Как только параметры обновлены, первые 18 реплик могут незамедлительно возобновить работу, не ожидая двух самых медленных реплик. Такая конфигурация обычно описывается как имеющая 18 реплик плюс две запасные реплики (*spare replica*)<sup>19</sup>.

<sup>19</sup> Такое название слегка сбивает с толку, поскольку 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). Тем не менее, это означает, что в случае аварийного отказа какого-то сервера обучение вполне нормально будет продолжено.

## Асинхронные обновления

При асинхронных обновлениях всякий раз, когда реплика завершает вычисление градиентов, они немедленно используются для обновления параметров модели. Агрегирование и синхронизация отсутствуют (удален шаг “среднее” на рис. 19.19). Реплики 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 во 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 обновляются 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. 19.20). Сильно просроченные градиенты называют *устаревшими градиентами* (*stale gradient*): они могут замедлять схождение, привносить эффекты шума и раскачивания (кривая обучения может содержать временные колебания) или даже делать алгоритм обучения расходящимся.

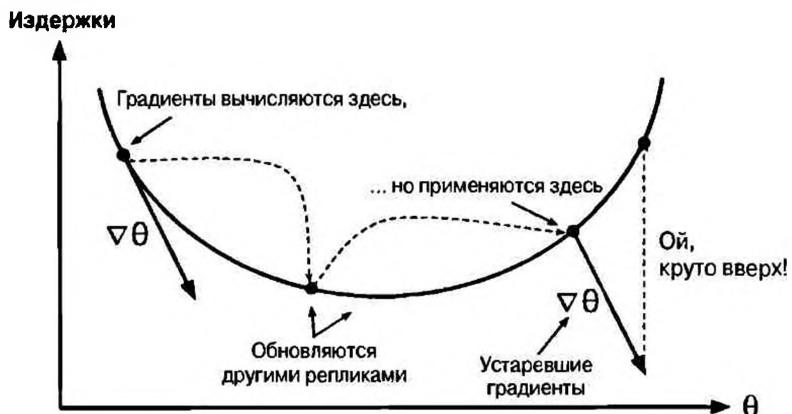


Рис. 19.20. Устаревшие градиенты во время применения асинхронных обновлений

Существует несколько способов ослабить влияние устаревших градиентов.

- 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* (*warming 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 2016 году (<https://homl.info/68>)<sup>20</sup>, было проведено сравнение различных подходов и сделан вывод о том, что применение синхронных обновлений с несколькими запасными репликами оказалось эффективнее, чем использование асинхронных обновлений; помимо более быстрого схождения получалась лучшая 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 с централизованными параметрами по-прежнему 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. Подобным образом, когда применяется стратегия зеркального отображения, производимые каждым ГП градиенты должны разделяться со всеми остальными ГП. К сожалению, всегда наступает момент, когда добавление дополнительного ГП вообще не улучшает производительность, т.к. время, затрачиваемое на перемещение данных в и из оперативной памяти ГП (и через сеть в распределенной конфигурации), будет перевешивать ускорение, достигнутое расщеплением вычислительной нагрузки. В такой точке добавление дополнительных ГП просто ухудшит ситуацию с насыщением полосы пропускания и на самом деле замедлит обучение.

<sup>20</sup> Цзяньминь Чэн и др., *Revisiting Distributed Synchronous SGD* (Еще раз о распределенном синхронном стохастическом градиентном спуске), препринт arXiv:1604.00981 (2016 г.).



Отдельные модели, которые обычно относительно невелики и обучены на очень крупном обучающем наборе, часто лучше обучать на одной машине с единственным мощным ГП, имеющим память с большой полосой пропускания.

Насыщение является более серьезным для больших плотных моделей, поскольку они имеют много параметров и градиентов, подлежащих перемещению. 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. Джекф Дин, инициатор и руководитель проекта Google Brain, сообщил (<https://homl.info/69>) 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:

- нейронный машинный перевод: 6-кратное ускорение на 8 ГП;
- Inception/ImageNet: 32-кратное ускорение на 50 ГП;
- RankBrain: 300-кратное ускорение на 500 ГП.

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.), так что за ближайшие несколько лет, по всей видимости, будет достигнут большой прогресс в области распараллеливания нейронных сетей.

Между тем, для ослабления проблемы насыщения, вероятно, вы пожелаете использовать несколько мощных ГП вместо множества слабых ГП, и также должны группировать свои ГП на небольшом количестве очень хорошо взаимосвязанных серверов. Вдобавок вы можете попробовать понизить точность значений с плавающей точкой с 32 битов (`tf.float32`) до 16 битов (`tf.bfloat16`). В итоге вдвое сократится объем передаваемых данных, зачастую не сильно влияя на скорость схождения или эффективность модели.

Наконец, если вы применяете централизованные параметры, то можете разнести (разделить) параметры по множеству серверов: добавление дополнительных серверов параметров снизит сетевую нагрузку на каждом сервере и ограничит риск насыщения полосы пропускания.

А теперь давайте обучим модель на множестве ГП!

## Обучение и масштабирование с использованием API-интерфейса *Distribution Strategies*

Многие модели способны довольно хорошо обучаться на единственном ГП или даже на ЦП. Но если обучение оказывается очень медленным, тогда вы можете попытаться распределить обучение между множеством ГП на одной машине. Если оно все еще остается слишком медленным, то попробуйте задействовать более мощные ГП или добавить на машину дополнительные ГП. Если ваша модель выполняет интенсивные вычисления (такие как перемножение крупных матриц), тогда она будет работать гораздо быстрее на мощных ГП, и вы даже могли бы применять тензорные процессоры на платформе AI Platform инфраструктуры Google Cloud, которые обычно обеспечивают еще более быструю работу таких моделей. Но если на вашей машине невозможно разместить добавочные ГП и тензорные процессоры не подходят (скажем, из-за того, что модель не извлекает много пользы от тензорных процессоров, или вы хотите использовать собственную аппаратную инфраструктуру), то можете попробовать обучать модель на нескольких серверах, каждый с множеством ГП (а когда всего перечисленного недостаточно, в качестве последнего средства можете попытаться добавить параллелизм модели, правда, это требует намного больших усилий). В настоящем разделе мы выясним, как обучать модели в широких масштабах, начиная с множества ГП на одной машине (или тензорных процессоров) и затем перейдя к множеству ГП на многих машинах.

К счастью, в состав TensorFlow входит очень простой API-интерфейс, который самостоятельно позаботится обо всех сложностях — *API-интерфейс стратегий распределения (Distribution Strategies)*. Чтобы обучить модель Keras на всех доступных ГП (пока на одной машине) с применением параллелизма данных со стратегией зеркального отображения, создайте объект `MirroredStrategy`, вызовите его метод `scope()` для получения контекста распределения и поместите внутрь этого контекста код создания и компиляции своей модели. Затем вызывайте метод `fit()` модели обычным образом:

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    mirrored_model = tf.keras.Sequential([...])
    mirrored_model.compile(...)

batch_size = 100    # должно делиться на количество реплик
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

“За кулисами” библиотека `tf.keras` осведомлена о распределениях, так что в контексте `MirroredStrategy` ей известно, что она должна реплицировать все переменные и операции на все доступные устройства ГП. Обратите внимание, что метод `fit()` будет автоматически распределять каждый обучающий пакет по всем репликам, а потому важно, чтобы размер пакета делился на количество реплик. Вот и все! Обучение в целом станет значительно быстрее, чем в случае использования единственного устройства, а изменения кода были действительно минимальными.

Закончив обучение модели, вы можете применять ее для эффективного вырабатывания прогнозов: вызовите метод `predict()` и он автоматически распределит пакет по всем репликам, вырабатывая прогнозы параллельно (опять-таки размер пакета должен делиться на количество реплик). Если вы вызовите метод `save()` модели, то она сохранится как обыкновенная модель, а не отображенная модель с множеством реплик. Таким образом, при загрузке она будет выполняться как обыкновенная модель на одиночном устройстве (по умолчанию ГП 0 или ЦП, когда ГП отсутствуют). Если вы хотите загрузить модель и выполнить ее на всех доступных устройствах, тогда должны вызывать функцию `keras.models.load_model()` внутри контекста распределения:

```
with distribution.scope():
    mirrored_model = keras.models.load_model("my_mnist_model.h5")
```

Если вы хотите использовать лишь подмножество всех доступных устройств ГП, то можете передать конструктору класса `MirroredStrategy` соответствующий список:

```
distribution = tf.distribute.MirroredStrategy(["/gpu:0", "/gpu:1"])
```

По умолчанию для операции среднего AllReduce класс `MirroredStrategy` применяет библиотеку групповых коммуникаций NVIDIA (*NVIDIA Collective Communications Library — NCCL*), но вы можете это изменить, устанавливая аргумент `cross_device_ops` в экземпляр класса `tf.distribute`.

HierarchicalCopyAllReduce или класса `tf.distribute.ReductionToOneDevice`. Стандартный вариант NCCL основан на классе `tf.distribute.NcclAllReduce`, который обычно быстрее, но все зависит от количества и типов ГП, так что можете испытать альтернативы<sup>21</sup>.

Если вы хотите попробовать воспользоваться параллелизмом данных с централизованными параметрами, тогда замените класс `MirroredStrategy` классом `CentralStorageStrategy`:

```
distribution = tf.distribute.experimental.CentralStorageStrategy()
```

Дополнительно вы можете установить аргумент `compute_devices` для указания списка устройств, которые желательно задействовать как рабочие процессоры (по умолчанию будут эксплуатироваться все доступные ГП), и аргумент `parameter_device`, чтобы указать устройство, где хотите хранить параметры (по умолчанию будет применяться ЦП или ГП, если он только один).

Теперь давайте посмотрим, как обучать модель на кластере серверов TensorFlow!

## Обучение модели на кластере TensorFlow

Кластер TensorFlow представляет собой группу процессов TensorFlow, функционирующих параллельно, обычно на разных машинах, и взаимодействующих друг с другом для выполнения определенной работы — например, обучения или запуска нейронной сети. Каждый процесс TF в кластере называется *задачей* (*task*) или *сервером* (*TF*). Он имеет IP-адрес, порт и тип (также называемый его *ролью* (*role*) или его *заданием* (*job*)). Типом может быть "worker" (рабочий сервер), "chief" (старший сервер), "ps" (сервер параметров) или "evaluator" (сервер оценки).

- Каждый рабочий сервер выполняет вычисления, обычно на машине с одним и более ГП.
- Старший сервер тоже выполняет вычисления (он является рабочим сервером), но занимается дополнительной работой, такой как запись в журналы TensorBoard или сохранение контрольных точек. В кластере

---

<sup>21</sup> Дополнительные детали об алгоритмах AllReduce можно найти в великолепной статье Ючиро Уено (<https://homl.info/uenopost>) и на странице, посвященной масштабированию с помощью NCCL (<https://homl.info/ncclalgo>).

имеется единственный старший сервер. Если старший сервер не указан, тогда им будет первый рабочий сервер.

- Сервер параметров занимается отслеживанием значений переменных и обычно функционирует на машине, располагающей только ЦП. Такой тип задачи используется лишь с классом ParameterServerStrategy.
- Сервер оценки вполне очевидно заботится о вычислениях.

Чтобы запустить кластер TensorFlow, его сначала потребуется определить, указав IP-адрес, TCP-порт и тип каждой задачи. Например, в приведенной ниже спецификации кластера определяется кластер с тремя задачами (два рабочих сервера и один сервер параметров, как показано на рис. 19.21). Спецификация кластера представляет собой словарь с одним ключом на задание и значениями, которые являются списками адресов задач (*IP-адрес:порт*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",           # /job:worker/task:0
        "machine-b.example.com:2222"             # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"]       # /job:ps/task:0
}
```

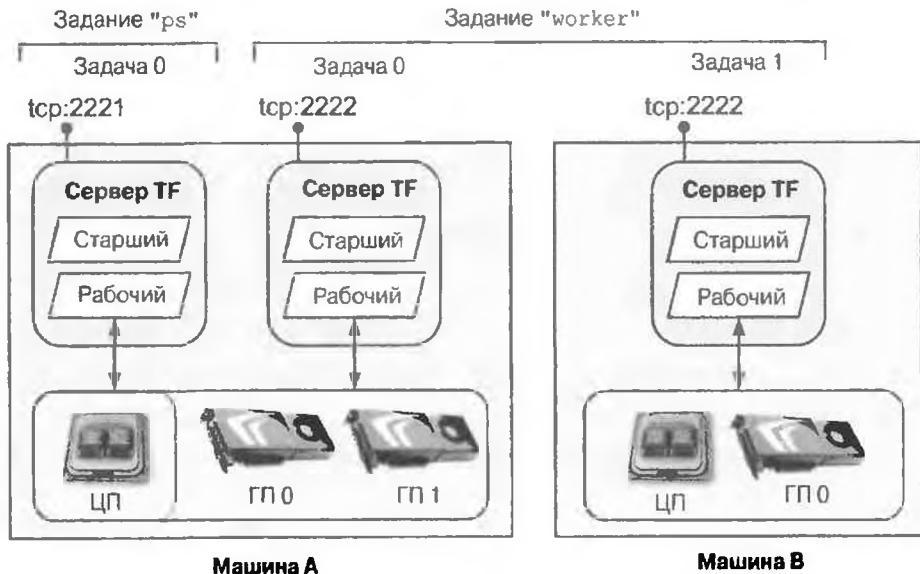


Рис. 19.21. Кластер TensorFlow

В общем случае на одну машину будет приходиться одна задача, но как продемонстрировано в примере, при желании на единственной машине можно сконфигурировать множество задач (если они совместно используют те же самые ГП, тогда удостоверьтесь в надлежащем разделении ОЗУ).



По умолчанию каждая задача в кластере может взаимодействовать с любой другой задачей, поэтому позаботьтесь о конфигурировании своего брандмауэра, чтобы разрешить коммуникации между соответствующими машинами на нужных портах (обычно будет проще применять один и тот же порт на каждой машине).

При запуске задачи вы должны предоставить ей спецификацию кластера, а также указать ее тип и индекс (например, рабочий сервер 0). Самый простой способ указать все сразу (спецификацию кластера плюс тип и индекс текущей задачи) предусматривает установку переменной среды TF\_CONFIG до запуска TensorFlow. Она должна быть словарем в формате JSON, содержащим спецификацию кластера (под ключом "cluster") вместе с типом и индексом текущей задачи (под ключом "task"). Скажем, следующая переменная среды TF\_CONFIG использует только что определенный кластер и указывает, что запускаемой задачей является первый рабочий сервер:

```
import  
import  
  
os.environ["TF_CONFIG"] = json.dumps({  
    "cluster": cluster_spec,  
    "task": {"type": "worker", "index": 0}  
})
```



В общем случае переменную среды TF\_CONFIG желательно определять вне Python, чтобы в код не требовалось включать тип и индекс текущей задачи (это позволяет применять тот же самый код во всех рабочих серверах).

Давайте же обучим модель на кластере! Мы начнем со стратегии зеркального отображения — она удивительно проста! Первым делом понадобится надлежащим образом установить переменную среды TF\_CONFIG для каждой задачи. Сервера параметров быть не должно (удалите ключ "ps" из спецификации кластера) и обычно желательно иметь по одному рабочему серверу на машину. Удостоверьтесь в том, что для каждой задачи установили отлича-

ющийся индекс. В заключение запустите на каждом рабочем сервере следующий код обучения:

```
distribution = tf.distribute.experimental.MultiWorkerMirroredStrategy()  
with distribution.scope():  
    mirrored_model = tf.keras.Sequential([...])  
    mirrored_model.compile([...])  
  
batch_size = 100      # должно делиться на количество реплик  
history = mirrored_model.fit(X_train, y_train, epochs=10)
```

Да, это точно такой же код, который мы использовали ранее, но теперь мы применяем класс MultiWorkerMirroredStrategy (в будущих версиях класс MirroredStrategy возможно будет поддерживать случаи с одной и множеством машин). Когда вы запускаете такой сценарий на первых рабочих серверах, они останутся заблокированными на шаге AllReduce, но как только последний рабочий сервер из числа тех, где было запущено обучение, начнет работу, вы увидите, что все они продвигаются с совершенно одинаковой скоростью (поскольку синхронизируются на каждом шаге).

Для такой стратегии распределения вы можете выбирать одну из двух реализаций AllReduce: кольцевой алгоритм AllReduce, основанный на gRPC для сетевых коммуникаций, и реализация NCCL. Наилучший для использования алгоритм зависит от числа рабочих серверов, количества и типов ГП и сети. По умолчанию для выбора подходящего алгоритма библиотека TensorFlow будет применять ряд эвристических правил, но при желании вы можете указать один алгоритм, передавая CollectiveCommunication.RING или CollectiveCommunication.NCCL (из tf.distribute.experimental) конструктору класса стратегии.

Если вы отдаете предпочтение реализации асинхронного параллелизма данных с серверами параметров, тогда поменяйте стратегию на ParameterServerStrategy, добавьте один или большее число серверов параметров и таким образом сконфигурируйте переменную среды TF\_CONFIG для каждой задачи. Обратите внимание, что хотя рабочие серверы будут функционировать асинхронно, реплики на каждом рабочем сервере работают синхронно.

Наконец, при наличии доступа к тензорным процессорам из Google Cloud (<https://cloud.google.com/tpu/>) вы можете создать примерно такую стратегию TPUStrategy (и затем использовать ее подобно другим стратегиям):

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()  
tf.tpu.experimental.initialize_tpu_system(resolver)  
tpu_strategy = tf.distribute.experimental.TPUStrategy(resolver)
```



Будучи исследователем, вы можете получить право бесплатно работать с тензорными процессорами; дополнительные детали ищите по ссылке <https://tensorflow.org/tfrc>.

Теперь вы можете обучать модели на множестве ГП и серверов: самое время похвалить себя! Если вы хотите обучить крупную модель, тогда вам будут нужны многочисленные ГП на многих серверах, что потребует приобретения массы оборудования или управления большим количеством виртуальных машин в облаке. Во многих случаях менее хлопотно и не так дорого иметь дело с облачной службой, которая сама обеспечит подготовку и управление всей инфраструктурой, когда она понадобится. Давайте посмотрим, как это сделать на платформе GCP.

## Запуск крупных заданий обучения на платформе AI Platform инфраструктуры Google Cloud

Если вы решили применять платформу AI Platform от Google, то можете ввести в действие задание обучения с помощью того же самого кода обучения, который вы запускали бы на собственном кластере TF. Платформа AI Platform позаботится о подготовке и конфигурировании желаемого количества виртуальных машин с ГП (в рамках ваших квот).

Для запуска задания вам будет нужен инструмент командной строки gcloud, который является частью Google Cloud SDK (<https://cloud.google.com/sdk/>). Вы можете либо установить этот комплект SDK на своей машине, либо просто использовать оболочку Google Cloud Shell на GCP. Она представляет собой терминал, который можно применять непосредственно в веб-браузере, и запускается в среде бесплатной виртуальной машины Linux (Debian) с уже установленным и сконфигурированным комплектом SDK. Оболочка Cloud Shell доступна повсюду в GCP: просто щелкните на значке Activate Cloud Shell (Активировать Cloud Shell), расположенном в правой верхней части страницы (рис. 19.22).

Если вы предпочитаете установить комплект SDK на своей машине, то после установки вы должны инициализировать его, выполнив команду gcloud init: вам необходимо войти в GCP и выдать права доступа к сво-

им ресурсам GCP, затем выбрать желаемый проект GCP (когда их больше одного), а также регион, где нужно запустить задание. Команда `gcloud` предоставляет доступ ко всем средствам GCP, включая те, что мы использовали ранее.



Рис. 19.22. Включение Cloud Shell

Вы не обязаны каждый раз открывать веб-интерфейс; можете написать сценарии, которые будут запускать и останавливать виртуальные машины, развертывать модели или выполнять любые другие действия GCP.

Прежде чем вы сможете запустить задание обучения, вам понадобится написать код обучения в точности, как делалось раньше для распределенной конфигурации (например, применяя `ParameterServerStrategy`). Платформа AI Platform позаботится об установке переменной среды `TF_CONFIG` для каждой виртуальной машины. Когда все будет сделано, вы можете развернуть модель и запустить ее на кластере TF с помощью командной строки следующего вида:

```
$ gcloud ai-platform jobs submit training my_job_20190531_164700 \
--region asia-southeast1 \
--scale-tier PREMIUM_1 \
--runtime-version 2.0 \
--python-version 3.5 \
--package-path /my_project/src/trainer \
--module-name trainer.task \
--staging-bucket gs://my-staging-bucket \
--job-dir gs://my-mnist-model-bucket/trained_model \
-- \
--my-extra-argument1 foo --my-extra-argument2 bar
```

Давайте разберем приведенные аргументы. Команда запустит задание обучения с именем `my_job_20190531_164700` в регионе `asia-southeast1`, используя масштабный уровень `PREMIUM_1`: это соответствует 20 рабочим серверам (включая старший сервер) и 11 серверам параметров (можете взглянуть на другие доступные масштабные уровни, пройдя по ссылке <https://homl.info/scaletiers>).

Все виртуальные машины будут основаны на исполняющей среде AI Platform 2.0 (конфигурация виртуальной машины, которая содержит TensorFlow 2.0 и много других пакетов)<sup>22</sup> и Python 3.5. Код обучения находится в каталоге `/my_project/src/trainer`, а команда `gcloud` автоматически встроит его в пакет `pip` и загрузит в хранилище GCS по адресу `gs://my-staging-bucket`. Далее исполняющая среда AI Platform запустит несколько виртуальных машин, развернет в них пакет и выполнит модуль `trainer.task`. Наконец, аргумент `--job-dir` и дополнительные аргументы (т.е. все аргументы, указанные после разделителя `--`) будут переданы программе: старшая задача обычно будет применять аргумент `--job-dir` для выяснения, куда сохранять финальную модель в GCS, в рассматриваемом случае `gs://my-mnist-model-bucket/trained_model`. Готово! Затем вы можете открыть меню навигации в консоли GCP, прокрутить его вниз до раздела *Artificial Intelligence* (Искусственный интеллект) и выбрать пункт *AI Platform → Jobs* (Платформа AI → Задания). Вы должны увидеть свое выполняющееся задание, а щелчок на нем приводит к отображению графиков использования ЦП, ГП и ОЗУ для каждой задачи. Вы можете щелкнуть на кнопке *View Logs* (Просмотреть журналы), чтобы получить доступ к подробным журналам с применением *Stackdriver*.



Если вы поместили обучающие данные в хранилище GCS, то для доступа к ним можете создать набор данных `tf.data.TextLineDataset` или `tf.data.TFRecordDataset`: просто используйте в качестве имен файлов пути внутри GCS (скажем, `gs://my-data-bucket/my_data_001.csv`). Упомянутые наборы данных при доступе к файлам полагаются на пакет `tf.io.gfile`: он поддерживает как локальные файлы, так и файлы GCS (но удостоверьтесь, что применяемый вами сервисный аккаунт имеет доступ в хранилище GCS).

Если хотите исследовать несколько значений гиперпараметров, тогда можете запустить множество заданий и указывать значения гиперпараметров с использованием дополнительных аргументов для своих задач. Тем не менее, для эффективного исследования многих гиперпараметров взамен лучше применять службу подстройки гиперпараметров платформы AI Platform.

<sup>22</sup> На момент написания главы исполняющая среда AI Platform 2.0 еще не была доступной, но может оказаться готовой ко времени, когда вы будете читать книгу. Проверяйте список доступных исполняющих сред (<https://homl.info/runtimes>).

## Служба подстройки гиперпараметров типа “черный ящик” платформы AI Platform

Платформа AI Platform предлагает мощную службу подстройки гиперпараметров на основе байесовской оптимизации под названием Google Vizier (<https://homl.info/vizier>)<sup>23</sup>. Для ее использования при создании задания понадобится передать файл конфигурации YAML (--config tuning.yaml). Например, вот как он может выглядеть:

```
trainingInput:  
  hyperparameters:  
    goal: MAXIMIZE  
    hyperparameterMetricTag: accuracy  
    maxTrials: 10  
    maxParallelTrials: 2  
    params:  
      - parameterName: n_layers  
        type: INTEGER  
        minValue: 10  
        maxValue: 100  
        scaleType: UNIT_LINEAR_SCALE  
      - parameterName: momentum  
        type: DOUBLE  
        minValue: 0.1  
        maxValue: 1.0  
        scaleType: UNIT_LOG_SCALE
```

Такой файл конфигурации YAML сообщает платформе AI Platform о том, что мы хотим довести до максимума метрику "accuracy", задание будет запускать максимум 10 испытаний (каждое испытание будет выполнять наш код обучения, чтобы обучить модель с нуля) и параллельно будут запускаться максимум 2 испытания. Мы хотим подстроить два гиперпараметра: `n_layers` (целое число между 10 и 100) и `momentum` (число с плавающей точкой между 0.1 и 1.0). Аргумент `scaleType` указывает априорное убеждение для значения гиперпараметра: `UNIT_LINEAR_SCALE` предполагает плоское априорное убеждение (т.е. априорного предпочтения нет), в то время как `UNIT_LOG_SCALE` указывает на наличие у нас априорного убеждения о том, что оптимальное значение лежит ближе к максимальному значению (другим

<sup>23</sup> Дениэл Головин и др., *Google Vizier: A Service for Black-Box Optimization* (Google Vizier: служба для оптимизации типа ‘черный ящик’), *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017 г.): с. 1487–1495.

возможным априорным убеждением является UNIT\_REVERSE\_LOG\_SCALE, когда мы уверены в том, что оптимальное значение должно быть ближе к минимальному значению).

Аргументы `n_layers` и `momentum` будут передаваться через командную строку коду обучения и, конечно же, он ожидает использовать их. Вопрос в том, как код обучения передаст метрику обратно платформе AI Platform, чтобы она могла решить, какие значения гиперпараметров применять во время следующего испытания? Дело в том, что платформа AI Platform отслеживает в выходном каталоге (указанном через `--job-dir`) любой файл событий (представленный в главе 10), который содержит сводки для метрики по имени "accuracy" (или метрики, указанной в `hyperparameterMetricTag`), и читает их значения. Таким образом, ваш код обучения просто должен использовать обратный вызов `TensorBoard()`, который вы все равно будете делать в целях мониторинга, и все готово!

После завершения задания все значения гиперпараметров, применяемые в каждом испытании, и результирующая правильность станут доступными в выходе задания (на странице AI Platform → Jobs).



Задания AI Platform также могут использоваться для эффективного выполнения вашей модели на крупных объемах данных: каждый рабочий процесс способен читать часть данных из хранилища GCS, вырабатывать прогнозы и сохранять их в GCS.

Теперь вы располагаете всеми инструментами и знаниями, необходимыми для создания нейронных сетей с современными архитектурами и их широкомасштабного обучения с применением разнообразных стратегий распределения в собственной инфраструктуре или в облаке — и вы можете даже выполнять мощную байесовскую оптимизацию для точной настройки гиперпараметров!

## Упражнения

1. Что содержит представление `SavedModel`? Как бы вы проинспектировали его содержимое?
2. Когда вы должны использовать сервер TF Serving? Каковы его основные характеристики? Назовите несколько инструментов, которые можно применять для его развертывания?

3. Как бы вы развернули модель на множестве экземпляров TF Serving?
4. Когда вы должны использовать API-интерфейс gRPC вместо REST для запрашивания модели, обслуживаемой сервером TF Serving?
5. Какими способами TFLite уменьшает размер модели, чтобы ее можно было запускать на мобильном или встроенном устройстве?
6. Что такое обучение с учетом квантования и зачем оно может понадобиться?
7. Что собой представляют параллелизм модели и параллелизм данных? Почему обычно рекомендуется применять параллелизм данных?
8. Какие стратегии распределения можно использовать при обучении модели на множестве серверов? Как выбрать стратегию для применения?
9. Обучите модель (любую желаемую) и разверните ее на сервере TF Serving или платформе AI Platform инфраструктуры Google Cloud. Напишите клиентский код для ее запрашивания с применением API-интерфейса REST или gRPC. Обновите модель и разверните новую версию. Теперь ваш клиентский код будет запрашивать новую версию. Выполните откат к первой версии.
10. Обучите любую модель с помощью множества ГП на одной машине, используя MirroredStrategy (если у вас нет доступа к ГП, тогда можете применять Colaboratory с исполняемой средой ГП и создать два виртуальных ГП). Снова обучите модель с использованием CentralStorageStrategy и сравните время обучения.
11. Обучите небольшую модель на платформе AI Platform инфраструктуры Google Cloud с применением подстройки гиперпараметров типа “черный ящик”.

## 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-m12` на GitHub или Твиттер @aurelienrgon.

На пути вперед мой лучший совет вам — постоянно совершенствуйте свои навыки: попытайтесь выполнить все упражнения (если вы этого еще сделали), работайте с тетрадями Jupyter, присоединяйтесь к Kaggle.com или какому-то другому сообществу МО, смотрите курсы лекций по МО, читайте статьи, участуйте в конференциях и встречайтесь с экспертами. Также чрезвычайно помогает наличие конкретного проекта, подлежащего выполнению, будь он для работы или ради удовольствия (а в идеале для того и другого), поэтому если есть что-то, что вы всегда мечтали построить, тогда стоит попробовать! Работайте постепенно; не пытайтесь сразу долететь до Луны, но сосредоточьте внимание на своем проекте и стройте его частями. Процесс требует терпения и настойчивости, а заслуженной наградой будет появление шагающего робота, действующего чатбота или чего-то еще.

Моя огромная надежда заключается в том, что книга вдохновит вас на создание замечательного приложения МО, которое принесет пользу всем нам! На что оно будет похоже?

*Орельен Жерон,  
17 июня 2019 года*

# Решения упражнений



Решения упражнений, связанных с написанием кода, доступны в онлайновых тетрадях Jupyter по адресу <https://github.com/ageron/handson-m12>.

## Глава 1. Введение в машинное обучение

1. Машинное обучение — это построение систем, которые способны учиться на основе данных. Обучение означает изменение работы в лучшую сторону на некоторой задаче при наличии определенной меры эффективности.
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 вариантом 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. В противоположность системе пакетного обучения система динамического обучения может обучать последовательно. Это делает ее способной быстро адаптироваться 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 таких 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 крупно 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 понижение 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. Обучающе-развивающий набор используется, когда существует риск несоответствия между обучающими данными и данными, присутствующими в проверочном и испытательном наборах (которые всегда должны быть как можно ближе к данным, применяемым после помещения модели в производственную среду). Обучающе-развивающий набор является частью обучающего набора, которая была удержанна (модель на нем не обучалась). Модель обучается на остальной части обучающего набора и оценивается как на обучающе-развивающем наборе, так и

на проверочном наборе. Если модель хорошо работает на обучающем наборе, но не на обучающе-развивающем наборе, то она, скорее всего, переобучается обучающим набором. Если модель хорошо работает и на обучающем, и на обучающе-развивающем наборе, но не на проверочном наборе, то, вероятно, существует значительное несоответствие между обучающими данными и проверочными + испытательными данными, и вы должны попытаться улучшить обучающие данные, чтобы они больше походили на проверочные + испытательные данные.

19. 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).

## Глава 2. Полный проект машинного обучения

См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-m12>.

## Глава 3. Классификация

См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-m12>.

## 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 или подход с сингулярным разложением, 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 схождение. Чтобы решить проблему, перед обучением модели вы должны масштабировать данные. Обратите внимание, что нормаль-

ное уравнение или подход с сингулярным разложением будет хорошо работать и без масштабирования. Кроме того, если признаки не масштабированы, то регуляризованные модели могут сходиться в субоптимальное решение: 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 выпуклой<sup>1</sup>.
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 продолжат прыжки вперед и назад возле глобального оптимума. Это означает, что даже если вы позволите им выполнятся в течение очень долгого времени, то 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 тоже 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; если она затем не улучшается в течение долгого

<sup>1</sup> Прямая линия между любыми двумя точками, выбранными на кривой, никогда не пересечет кривую.

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, по всей видимости, в том, что ваша модель переобучается обучающим набором. Одна попытка устраниить 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  $\ell_2$  (гребневая регрессия) или штрафа  $\ell_1$  (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  $\alpha$ .
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 обычновенной линейной регрессией.
  - Лассо-регрессия применяет штраф  $\ell_1$ , который стремится довести веса до полных нулей. Итогом будут разреженные модели, где все веса кроме самых важных являются нулевыми. Это способ автоматического выполнения выбора признаков, который хорош, если вы полагаете, что в действительности существенны только немногие

признаки. 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 хотите получить лассо-регрессию, работающую без перебоев, тогда можете применять эластичную сеть со значением `l1_ratio`, близким к 1.
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 на основе логистической регрессии.
  12. См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml2>.

## 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 классификатор 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 классификатора SVM (обученного посредством дополнительной перекрестной проверки с контролем по пяти блокам на обучающих данных). Это добавит к классификатору 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  $H'$ ,  $f'$ ,  $A'$  и  $b'$  (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:

$$\mathbf{H} = \begin{pmatrix} \mathbf{H}' & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{0} & \\ \vdots & & \ddots \end{pmatrix}.$$

- $\mathbf{f}$  равно  $\mathbf{f}'$  с  $m$  дополнительных элементов, которые все равны значению гиперпараметра  $C$ .
- $\mathbf{b}$  равно  $\mathbf{b}'$  с  $m$  дополнительных элементов, которые все равны нулю.
- $\mathbf{A}$  равно  $\mathbf{A}'$  с добавочной единичной матрицей  $\mathbf{I}_m$  размера  $m \times m$ , добавленной справа,  $-\mathbf{I}_m$  чуть ниже ее и остатком, заполненным нулями:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}.$$

Решения упражнений 8, 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 6. Деревья принятия решений

1. Глубина хорошо сбалансированного двоичного дерева, содержащего  $m$  листьев, равна  $\log_2(m)$  с округлением в большую сторону<sup>2</sup>. 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 роди-

<sup>2</sup>  $\log_2$  — это двоичный логарифм;  $\log_2(m) = \log(m) / \log(2)$ .

теля, 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:  $1 - \left(\frac{1}{5}\right)^2 - \left(\frac{4}{5}\right)^2 = 0.32$ . 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  $1 - \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^2 = 0.5$ , что выше, чем у его родителя. Это уравновешивается тем фактом, что другой узел является чистым, а потому общая взвешенная загрязненность Джини составляет  $\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0 = 0.2$ , что ниже, чем загрязненность Джини родителя.

- Если дерево принятия решений переобучается обучающим набором, то уменьшение `max_depth` может быть хорошей идеей, т.к. это ограничит модель, регуляризируя ее.
- 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 недобучается на обучающем наборе, тогда масштабирование входных признаков будет просто пустой тратой времени.
- 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.
- 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 тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## 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 эффективным 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 случайному лесу 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 дополнительная 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 тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 8. Понижение размерности

1. Ниже перечислены главные мотивы для понижения размерности.
  - о Чтобы ускорить последующий алгоритм обучения (в некоторых случаях понижение размерности может даже устранить шум и избыточные признаки, улучшая работу алгоритма обучения).

- о Чтобы визуализировать данные и получить представление о самых важных признаках.
  - о Чтобы сберечь пространство (сжатие).
- Далее указаны основные недостатки понижения размерности.
- о Некоторая информация утрачивается, из-за чего может ухудшиться эффективность последующих алгоритмов обучения.
  - о Понижение размерности может быть сопряжено с большим объемом вычислений.
  - о Оно привносит некоторую сложность в конвейеры машинного обучения.
  - о 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 (как в наборе данных Swiss roll), тогда понижение размерности посредством PCA приведет к утрате слишком большого объема информации. Ведь вы хотите развернуть Swiss roll, а не сплющить его.

5. Сложный вопрос: все зависит от набора данных. Давайте рассмотрим два крайних 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, предохраняя в то же время 95% дисперсии. Теперь представим, что набор данных состоит из совершенно случайных точек, разбросанных по 1000 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 идеи 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 эффективность второго алгоритма. Если понижение размерности не вызывает утрату слишком большого объема информации, то алгоритм должен выполнять 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, подобного LLE. Такой двухэтапный подход обеспечит ту же самую эффективность модели, как и в случае использования только LLE, но за меньшее время.

Решения упражнений 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 9. Методики обучения без учителя

1. В машинном обучении кластеризация является задачей обучения без учителя, направленной на группирование вместе похожих образцов. Понятие подобия зависит от имеющейся задачи: например, в некоторых случаях два близлежащих образца будут считаться похожими, в то время как в других похожие образцы могут находиться на большом расстоянии друг от друга, пока они принадлежат той же плотно упакованной группе. Популярные алгоритмы кластеризации включают K-Means, DBSCAN, агломеративную кластеризацию, BIRCH, Mean-Shift, распространение похожести и спектральную кластеризацию.
2. В число главных приложений алгоритмов кластеризации входят анализ данных, сегментация заказчиков, системы выдачи рекомендаций, поисковые механизмы, сегментация изображений, частичное обучение, понижение размерности, обнаружение аномалий и обнаружение новизны.
3. Метод локтя является простой методикой выбора количества кластеров при использовании K-Means: нужно лишь вычертить график инерции (среднеквадратического расстояния от каждого образца до его ближайшего центроида) как функции количества кластеров и найти точку на кривой, где инерция перестает быстро уменьшаться (“локоть”). Обычно она близка к оптимальному количеству кластеров. Еще один подход предусматривает вычерчивание графика оценки силуэта как функции количества кластеров. На графике часто будет пик и, как правило, оптимальное количество кластеров находится поблизости. Оценка силуэта — это средний коэффициент силуэта по всем образцам. Такой ко-

эффициент варьируется от +1 для образцов, которые находятся внутри своего кластера и далеко от остальных кластеров, до -1 для образцов, очень близких к другому кластеру. Вы также можете вычертить диаграммы силуэтов и провести более основательный анализ.

4. Снабжение метками набора данных сопряжено с высокими затратами и требует времени. По этой причине обычно имеется масса непомеченных образцов и лишь немного помеченных. Распространение меток представляет собой прием, который заключается в копировании некоторых (или всех) меток из помеченных образцов в похожие непомеченные образцы. Это может значительно увеличить количество помеченных образцов и тем самым позволить алгоритму обучения с учителем достичь большей эффективности (форма частичного обучения). Один из подходов заключается в том, чтобы использовать на всех образцах алгоритм кластеризации, такой как K-Means, найти наиболее распространенную метку для каждого кластера или метку самого репрезентативного образца (т.е. ближайшего к центроиду) и распространить ее на непомеченные образцы в том же кластере.
5. Алгоритмы кластеризации K-Means и BIRCH хорошо масштабируются на крупные наборы данных. Алгоритмы кластеризации DBSCAN и Mean-Shift ищут области высокой плотности.
6. Активное обучение полезно, когда имеется масса непомеченных образцов, но снабжение метками оказывается дорогостоящим. В таком (довольно распространенном) случае вместо того, чтобы выбирать образцы для пометки случайным образом, часто предпочтительнее проводить активное обучение, при котором человек-эксперт взаимодействует с алгоритмом обучения, предоставляя метки для индивидуальных образцов, когда алгоритм их запрашивает. Общепринятым подходом является выборка по наименьшей уверенности (см. врезку “Активное обучение” в главе 9).
7. Многие люди используют понятия *обнаружение аномалий* и *обнаружение новизны* взаимозаменямо, но они не совсем одинаковы. При обнаружении аномалий алгоритм обучается на наборе данных, который может содержать выбросы, и цель обычно заключается в том, чтобы идентифицировать такие выбросы (внутри обучающего набора), а также выбросы среди новых образцов. При обнаружении новизны алгоритм обучается на наборе данных, который предположительно “чист”,

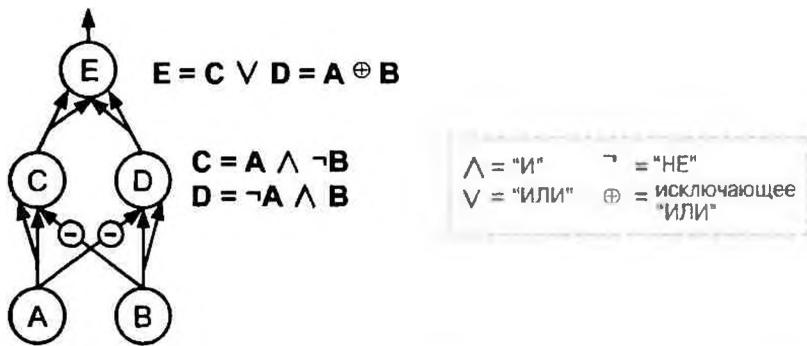
а целью является выявление новинок строго среди новых образцов. Одни алгоритмы работают лучше для обнаружения аномалий (скажем, изолирующий лес), тогда как другие лучше подходят для обнаружения новизны (например, одноклассовый SVM).

8. Модель со смесью гауссовых распределений — это вероятностная модель, которая предполагает, что образцы были сгенерированы из смеси нескольких гауссовых распределений с неизвестными параметрами. Другими словами, допущение заключается в том, что данные сгруппированы в конечное число кластеров, каждый с эллиптической формой (но кластеры могут иметь разные эллиптические формы, размеры, ориентации и плотности), и мы не знаем, к какому кластеру принадлежит каждый образец. Такая модель удобна для оценки плотности, кластеризации и обнаружения аномалий.
9. Один из способов нахождения правильного количества кластеров при использовании модели со смесью гауссовых распределений предусматривает вычерчивание графика с байесовским информационным критерием (BIC) или с информационным критерием Акаике (AIC) как функции количества кластеров и выбор такого числа кластеров, которое сводит к минимуму BIC или AIC. Еще один способ предполагает применение модели со смесью гауссовых распределений, которая выбирает количество кластеров автоматически.

Решения упражнений 10–13 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 10. Введение в искусственные нейронные сети с использованием Keras

1. Зайдите на веб-сайт с инструментом TensorFlow Playground (<https://playground.tensorflow.org/>) и проделайте то, что описано в упражнении.
2. 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.



3. Классический персепtron 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 персептрана 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 персептран станет эквивалентным классификатору на основе логистической регрессии.
4. Логистическая функция активации была ключевым ингредиентом при обучении первых многослойных персептронов из-за того, 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.
5. В число популярных функций активации входят ступенчатая функция, логистическая (сигмоидальная) функция, функция гиперболического тангенса ( $\tanh$ ) и функция выпрямленного линейного элемента (ReLU; см. рис. 10.8). В главе 11 приводятся другие примеры, такие как ELU и варианты функции ReLU.
6. Обсудим многослойный персептран, 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 выход-

ной слой с 3 искусственными нейронами, при этом все искусственные нейроны используют функцию активации ReLU.

- о Формой входной матрицы  $X$  является  $m \times 10$ , где  $m$  представляет размер обучающего пакета.
  - о Формой вектора весов скрытого слоя  $W_h$  является  $10 \times 50$ , а длина его вектора смещений  $b_h$  составляет 50.
  - о Формой вектора весов выходного слоя  $W_o$  является  $50 \times 3$ , а длина его вектора смещений  $b_o$  составляет 3.
  - о Формой входной матрицы  $Y$  сети является  $m \times 3$ .
- о  $Y = \text{ReLU}(\text{ReLU}(X W_h + b_h) \cdot W_o + b_o)$ . 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*.
7. Чтобы классифицировать почтовые сообщения на спам и не спам, в выходном слое нейронной сети нужен только один нейрон, скажем, 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 многогиперменной функцией активации, которая способна обрабатывать множество классов, выдавая по одной вероятности на класс. Если вы желаете, чтобы нейронная сеть прогнозировала цены на дома, как в главе 2, то нужно предусмотреть один выходной нейрон и вообще не использовать в выходном слое функцию активации<sup>3</sup>.
8. 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 обучаю-

<sup>3</sup> 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.  $\exp(\log v) = v$ ).

щих 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, рассчитывая значение каждого узла для текущего обучающего пакета, и затем осуществляет обратный проход, вычисляя все градиенты за один раз (дополнительные сведения ищите в приложении Г). Так в чем же разница? Дело в том, что обратное распространение относится к полному процессу обучения искусственной нейронной сети с применением множества шагов обратного распространения, каждый из которых рассчитывает 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 самим обратным распространением.

9. Вот список гиперпараметров, которые можно подстраивать в базовом многослойном персептроне: количество скрытых слоев, число нейронов в каждом скрытом слое и функция активации, используемая в каждом скрытом слое и в выходном слое<sup>4</sup>. 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.

Если многослойный персепtron переобучается обучающими данными, тогда можно попробовать уменьшить количество скрытых слоев и число нейронов на скрытый слой.

<sup>4</sup> В главе 11 мы обсуждали много методик, 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 (скажем, доля отключения, когда применяется отключение) и т.д.

10. См. тетради Jupyter, доступные по адресу <https://github.com/ageron/handson-ml2>.

## 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 случного выбора весов является разрушение симметрии: если все веса имеют то же самое начальное значение, даже отличное от нуля, тогда симметрия не разрушена (т.е. все нейроны в заданном слое эквивалентны) и обратное распространение неспособно ее разрушить. 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 подобно весам, что тоже нормально; большой разницы в подходах нет.
3. Ниже указано несколько преимуществ функции активации SELU по сравнению с ReLU.
  - о Функция активации SELU 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.
  - о Функция активации SELU всегда имеет ненулевую производную, избегая проблемы угасающих элементов, которой могут подвергаться элементы ReLU.
  - о Если соблюдаются надлежащие условия (т.е. модель последовательная, веса инициализированы с использованием инициализации Лекуна, входы стандартизированы и отсутствуют несовместимые слои или регуляризации, такие как отключение или регуляризация  $\ell_1$ ), тогда функция активации SELU гарантирует, что модель будет самонормализованной, а это решает проблемы взрывного роста/исчезновения градиентов.

- Функция активации SELU является хорошим стандартным вариантом. 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 SELU 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 17). Кроме того, временами она может извлечь пользу от оптимизированной реализации, а также от аппаратного ускорения. 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 (кроме рекуррентных сетей). Логистическая функция активации также полезна в выходном слое, когда необходимо оценивать вероятность (скажем, для двоичной классификации), но редко используется в скрытых слоях (есть исключения — например, для кодирующего слоя вариационного автокодировщика; см. главу 17). Наконец, многопеременная функция активации удобна в выходном слое при выдаче вероятностей для взаимоисключающих классов, но она редко (если вообще когда-либо) применяется в скрытых слоях.
- Если при использовании оптимизатора SGD 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 его момент приведет к тому, что он попадет прямо за минимум. Затем он замедлится и возвратится, снова ускорится, опять промахнется и т.д. Алгоритм может раскачиваться подобным образом много раз до того, как сойтись, так что в целом его схождение займет гораздо большее время, чем при меньшем значении `momentum`.
- 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  $\ell_1$  во время обучения, что подталкивает оптимизатор к разреженности. Третий способ связан с использованием комплекта TensorFlow Model Optimization Toolkit.

7. Да, отключение замедляет процесс обучения, в общем случае приблизительно в два раза. Тем не менее, оно не влияет на скорость выведения, поскольку включается лишь во время обучения. Отключение MC Dropout в точности похоже на обыкновенное отключение во время обучения, но оно по-прежнему активно при выведении, поэтому каждое выведение слегка замедляется. Более важно то, что когда применяется отключение MC Dropout, вы обычно пожелаете выполнять выведение 10 или больше раз с целью получения лучших прогнозов. Таким образом, вырабатывание прогнозов замедлится в 10 и более раз.

Решения упражнений 8–10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 12. Специальные модели и обучение с помощью TensorFlow

- TensorFlow является библиотекой с открытым кодом, которая предназначена для численных расчетов, особенно хорошо подходит для крупномасштабного машинного обучения и точно под него подогнана. Ядро библиотеки TensorFlow похоже на NumPy, но TensorFlow также предлагает поддержку графических процессоров и распределенных вычислений, возможности анализа вычислительных графов и оптимизации (с переносимым форматом графов, который позволяет обучать модель TensorFlow в одной среде и запускать ее в другой), API-интерфейс оптимизации, основанный на автоматическом дифференцировании в обратном режиме, и несколько мощных API-интерфейсов, таких как `tf.keras`, `tf.data`, `tf.image`, `tf.signal` и т.д. В число других популярных библиотек для глубокого обучения входят PyTorch, MXNet, Microsoft Cognitive Toolkit, Theano, Caffe2 и Chainer.
- Хотя библиотека TensorFlow предлагает большинство функциональности, обеспечиваемой NumPy, по ряду причин она не является готовой заменой NumPy. Во-первых, имена функций не всегда совпадают (например, `tf.reduce_sum()` против `np.sum()`). Во-вторых, некоторые функции не обладают совершенно идентичным поведением (скажем, `tf.transpose()` создает транспонированную копию тензора, тогда как атрибут `T` в NumPy создает транспонированное представление без фактического копирования каких-либо данных). В-третьих, массивы

NumPy изменяемы, в то время как тензоры TensorFlow — нет (но когда нужен изменяемый объект, можно использовать `tf.Variable`).

3. Оба вызова, `tf.range(10)` и `tf.constant(np.arange(10))`, возвращают одномерный тензор, содержащий числа от 0 до 9. Однако первый использует 32-битные целые, тогда как второй — 64-битные целые. На самом деле в TensorFlow по умолчанию используется 32 бита, а в NumPy — 64 бита.
4. Помимо обычных тензоров TensorFlow предлагает другие структуры данных, включая разреженные тензоры, массивы тензоров, зубчатые тензоры, очереди, строковые тензоры и множества. Последние две фактически представлены как обычные тензоры, но TensorFlow предоставляет специальные функции для манипулирования ими (в `tf.strings` и `tf.sets`).
5. Когда нужно определить специальную функцию потерь, в общем случае вы можете просто реализовать ее как обычную функцию Python. Тем не менее, если специальная функция потерь обязана поддерживать гиперпараметры (или любое другое состояние), тогда вы должны создать подкласс класса `keras.losses.Loss` и реализовать методы `__init__()` и `call()`. Если вы хотите, чтобы гиперпараметры функции потерь сохранялись вместе с моделью, то должны также реализовать метод `get_config()`.
6. Во многом подобно специальным функциям потерь большинство метрик могут определяться как обычные функции Python. Но если вы хотите, чтобы специальная метрика поддерживала гиперпараметры (или любое другое состояние), тогда вы должны создать подкласс класса `keras.metrics.Metric`. Кроме того, если вычисление метрики по целой эпохе не эквивалентно расчету средней метрики по всем пакетам в этой эпохе (например, как для метрик точности и полноты), то вы должны создать подкласс класса `keras.metrics.Metric` и реализовать методы `__init__()`, `update_state()` и `result()`, чтобы отслеживать скользящую метрику в течение каждой эпохи. Вам также следует реализовать метод `reset_states()`, если ему необходимо делать нечто большее, чем просто переустановить все переменные в 0.0. Когда нужно, чтобы состояние сохранялось вместе с моделью, тогда понадобится также реализовать метод `get_config()`.

7. Вы обязаны проводить различие между внутренними компонентами своей модели (т.е. слоями или многократно используемыми блоками слоев) и самой моделью (т.е. объектом, который будет обучаться). Первые должны быть подклассами класса `keras.layers.Layer`, а вторые — подклассами класса `keras.models.Model`.
8. Написание собственного специального цикла обучения — довольно сложная задача, поэтому вы должны ею заниматься только в случае настоятельной необходимости. Библиотека Keras предоставляет несколько инструментов для настройки обучения без потребности в написании собственного цикла обучения: обратные вызовы, специальные регуляризаторы, специальные ограничения, специальные потери и т.д. Вы должны их применять всякий раз, когда это возможно: написание специального цикла обучения чревато ошибками, к тому же многократно использовать написанный вами специальный код будет труднее. Однако в ряде случаев написание специального цикла обучения оказывается необходимым — например, если вы хотите применять для разных частей своей нейронной сети разные оптимизаторы, как в статье об обучении Wide & Deep (<https://homl.info/widedeep>). Специальный цикл обучения также может быть полезен при отладке или попытке понять, как на самом деле работает обучение.
9. Специальные компоненты Keras должны допускать преобразование в функции TF Function, т.е. максимально возможно придерживаться операций TF и соблюдать все правила, перечисленные в разделе “Правила TF Function” главы 12. Если возникает крайняя необходимость включить в специальный компонент произвольный код Python, тогда вы можете либо поместить его внутрь операции `tf.py_function()` (но это снизит производительность и ограничит переносимость вашей модели), либо установить `dynamic=True` при создании специального слоя или модели (либо установить `run_eagerly=True` при вызове метода `compile()` модели).
10. Список правил, которые нужно соблюдать при создании функции TF Function, приведен в разделе “Правила TF Function” главы 12.
11. Создавать динамическую модель Keras может быть полезно при отладке, т.к. она не компилирует любой специальный компонент в функцию TF Function и у вас появляется возможность использовать любой от-

ладчик Python для отладки своего кода. Она также может быть удобной, если вы хотите включить в модель (или в код обучения) произвольный код Python, в том числе обращения к внешним библиотекам. Чтобы сделать модель динамической, во время ее создания потребуется установить `dynamic=True`. В качестве альтернативы вы можете установить `run_eagerly=True` при вызове метода `compile()` модели. Превращение модели в динамическую не позволит Keras применять любые средства графов TensorFlow, так что обучение и выводение замедлятся, а у вас исчезнет возможность экспортования вычислительного графа, в результате чего переносимость модели ограничится.

Решения упражнений 12 и 13 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 13. Загрузка и предварительная обработка данных с помощью TensorFlow

1. Поглощение крупного набора данных и его эффективная предварительная обработка могут оказаться сложной инженерной задачей. API-интерфейс Data делает ее довольно простой. Он предлагает много средств, включая загрузку данных из разнообразных источников (таких как текстовые или двоичные файлы), чтение данных из множества источников в параллельном режиме, трансформация данных, чередование записей, тасование данных, формирование из них пакетов и упреждающая выборка.
2. Разделение крупного набора данных на множество файлов делает возможным его тасование на крупнозернистом уровне перед тасованием на более мелком уровне, используя буфер тасования. Оно также позволяет обрабатывать огромные наборы данных, которые не умещаются внутри одной машины. Кроме того, манипулировать тысячами небольших файлов проще, чем одним гигантским файлом; например, облегчается разделение данных на множество поднаборов. Наконец, если данные разделены на множество файлов, распространенных по множеству серверов, то становится возможной одновременная загрузка нескольких файлов из разных серверов, что улучшает потребление полосы пропускания.

3. Вы можете воспользоваться TensorBoard для визуализации данных профилирования: если ГП утилизируется не полностью, тогда узким местом, вероятно, является ваш входной конвейер. Вы можете устранить проблему, обеспечив чтение и предварительную обработку данных во множестве потоков параллельно, а также упреждающую выборку нескольких пакетов. Если этого недостаточно для использования ГП на 100% во время обучения, то удостоверьтесь в том, что ваш код предварительной обработки оптимизирован. Можете также попробовать сохранить набор данных во множестве файлов TFRecord и при необходимости выполнять предварительную обработку заблаговременно, чтобы не делать ее на лету на стадии обучения (помочь в этом может TF Transform). В случае потребности применяйте машину с большим количеством ядер ЦП и объемом ОЗУ, а также достаточно широкой полосой пропускания ГП.
4. Файл состоит из последовательности произвольных двоичных записей: в каждой записи вы можете хранить абсолютно любые двоичные данные, какие только хотите. Тем не менее, на практике большинство файлов TFRecord содержат последовательности сериализованных протокольных буферов. Это позволяет задействовать в своих интересах преимущества протокольных буферов, такие как тот факт, что их можно легко читать среди множества платформ и языков, а их определение может позже обновляться с поддержкой обратной совместимости.
5. Преимущество формата протобуфера Example в том, что библиотека TensorFlow предоставляет ряд операций для его разбора (функции `tf.io.parse*example()`), не заставляя вас определять собственный формат. Он достаточно гибкий, чтобы представлять образцы в большинстве наборов данных. Однако если он не покрывает ваш сценарий использования, тогда вы можете определить собственный протокольный буфер, скомпилировать его с применением `protoc` (установив аргументы `--descriptor_set_out` и `--include_imports` для экспортования дескриптора протобуфера) и задействовать функцию `tf.io.decode_proto()` для разбора сериализованных протобуферов (пример ищите в разделе “Custom protobuf” (“Специальный протобуфер”) тетради для главы 13). Такой подход сложнее и требует развертывания дескриптора наряду с моделью, но он вполне реален.

6. При использовании записей TFRecord вы обычно будете активизировать сжатие, если файлы TFRecord требуется загружать в сценарии обучения, т.к. сжатие делает файлы меньше и тем самым сокращает время загрузки. Но если файлы находятся на той же самой машине, что и сценарий обучения, то в целом предпочтительнее отказаться от сжатия во избежание излишней траты вычислительной мощности ЦП на распаковку.
7. Давайте рассмотрим доводы за и против каждого варианта предварительной обработки.
  - о Если вы предварительно обрабатываете данные при создании файлов данных, то сценарий обучения будет выполняться быстрее, т.к. ему не придется делать предварительную обработку на лету. В некоторых случаях предварительно обработанные данные также оказываются намного меньше исходных данных, что позволит сэкономить пространство и ускорить загрузку. Также полезно материализовать предварительно обработанные данные, скажем, для инспектирования или архивации. Тем не менее, у такого подхода есть несколько минусов. Во-первых, не так-то легко экспериментировать с различной логикой предварительной обработки, если вам необходимо генерировать предварительно обработанный набор данных для каждого варианта. Во-вторых, если вы хотите выполнять дополнение данных, то придется материализовывать много вариантов вашего набора данных, что приведет к расходу большого объема дискового пространства и длительного времени генерации. В-третьих, обученная модель будет ожидать предварительно обработанных данных, поэтому вам потребуется добавить в свое приложение код предварительной обработки, который должен быть выполнен до обращения к модели.
  - о Если данные предварительно обрабатываются с помощью конвейера `tf.data`, то будет намного легче подстраивать логику предварительной обработки и применять дополнение данных. Вдобавок `tf.data` упрощает построение высокоеффективных конвейеров предварительной обработки (например, с многопоточностью и упреждающей выборкой). Однако предварительная обработка данных подобным образом замедлит обучение. Кроме того, каждый обучающий образец будет предварительно обрабатываться один раз на эпоху, а не просто однократно, если данные были предварительно обработаны

при создании файлов данных. Наконец, обученная модель по-прежнему будет ожидать предварительно обработанных данных.

- Если вы добавите к своей модели слои предварительной обработки, тогда писать код предварительной обработки придется только один раз для обучения и выводения. Если модель должна быть развернута на множестве разных платформ, то вам не нужно будет многократно писать код предварительной обработки. Плюс вы не подвергаетесь риску использовать неправильную логику предварительной обработки для своей модели, поскольку она будет частью самой модели. С другой стороны, предварительная обработка данных замедлит обучение, и каждый обучающий образец будет предварительно обрабатываться однократно на эпоху. Кроме того, по умолчанию операции предварительной обработки для текущего пакета будут выполняться на ГП (вы лишиены преимуществ параллельной предварительной обработки на ЦП и упреждающей выборки). К счастью, предстоящие слои предварительной обработки Keras должны быть способны изымать операции предварительной обработки из слоев предварительной обработки и запускать их как часть конвейера `t.f.data`, так что вы извлечете выгоду из многопоточного выполнения на ЦП и упреждающей выборки.
- Наконец, применение TF Transform для предварительной обработки предоставляет вам многие преимущества предшествующих вариантов: предварительно обработанные данные материализуются, каждый образец предварительно обрабатывается только раз (ускоряя обучение) и слои предварительной обработки генерируются автоматически, а потому писать код предварительной обработки придется лишь однократно. Основной недостаток связан с тем фактом, что вам необходимо научиться пользоваться этим инструментом.

## 8. Давайте посмотрим, как кодировать категориальные признаки и текст.

- Простейший вариант кодирования категориального признака, который имеет естественный порядок, такой как оценка фильма (например, “плохой”, “средний”, “хороший”), предусматривает использование порядковой кодировки: сортировку категорий в их естественном порядке и сопоставление каждой категории с ее рангом (скажем, “плохой” отображается на 0, “средний” — на 1 и “хороший” — на 2). Тем не менее, большинство категориальных признаков не имеет тако-

го естественного порядка. Например, естественный порядок отсутствует для профессий или стран. В этом случае вы можете применять унитарное кодирование или, если категорий много, то вложения.

- Для текста один из вариантов предполагает использование представления в виде мешка слов: предложение представляется с помощью вектора, подсчитывающего количество каждого возможного слова. Так как общеупотребительные слова обычно не очень важны, вы захотите применить методику TF-IDF для уменьшения их веса. Вместо подсчета слов часто подсчитывают *n*-граммы, которые являются последовательностями из *n* следующих друг за другом слов — просто и изящно. В качестве альтернативы можете кодировать каждое слово с использованием вложений слов, возможно заранее обученных. Вместо кодирования слов также можно кодировать каждую букву или маркеры подслов (скажем, разбивая “smartest” на “smart” и “est”). Последние два варианта обсуждаются в главе 16.

Решения упражнений 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12>.

## Глава 14. Глубокое компьютерное зрение с использованием сверточных нейронных сетей

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 вроде 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 встроенным. 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 ядра  $3 \times 3$ , 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 ядра  $3 \times 3$  и 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 ядра  $3 \times 3$  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", размеры по горизонтали и вертикали карт признаков делятся на 2 в каждом слое (при необходимости с округлением в большую сторону). Таким образом, при входных каналах  $200 \times 300$  пикселей картами признаков первого слоя будут  $100 \times 150$ , второго слоя —  $50 \times 75$  и третьего слоя —  $25 \times 38$ . Так как 32 бита соответствуют 4 байтам и первый сверточный слой имеет 100 карт признаков, данный слой займет  $4 \times 100 \times 150 \times 100 = 6$  мил-

лионов байтов (6 Мбайт). Второй слой потребует  $4 \times 50 \times 75 \times 200 = 3$  миллиона байтов (3 Мбайт). Третий слой займет  $4 \times 25 \times 38 \times 400 = 1\,520\,000$  байтов (около 1.5 Мбайт). Однако после расчета слоя память, занимаемая предыдущим слоем, может быть освобождена, поэтому если все хорошо оптимизировано, то потребуется лишь  $6 + 3 = 9$  миллионов байтов (9 Мбайт) памяти (когда второй слой был только что рассчитан, но память, занятая первым слоем, пока еще не освободилась). Но подождите, необходимо также добавить память, занимаемую параметрами сети CNN! Ранее мы выяснили, что сеть имеет 903 400 параметров, каждый из которых задействует до 4 байтов, а потому добавляется 3 613 600 байтов (около 3.6 Мбайт). Следовательно, общий объем требуемой памяти составляет (по меньшей мере) 12 613 600 байтов (около 12.6 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 прохода, до тех пор, пока не начнется обратный шаг. Таким образом, мы должны подсчитать общий объем памяти, который требуют все слои для одиночного образца, и умножить его на 50. С этого момента давайте начнем считать в мегабайтах, а не в байтах. Ранее мы вычислили, что три слоя требуют соответственно 6, 3 и 1.5 Мбайт для каждого образца. В итоге имеем 10.5 Мбайт на образец, поэтому для 50 образцов общий объем памяти составляет 525 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 (36 Мбайт), плюс объем памяти, требуемой для параметров модели, который составляет примерно 3.6 Мбайт (вычислен ранее), плюс объем памяти для градиентов (мы проигнорируем ее, т.к. она будет постепенно освобождаться по мере продвижения обратного распространения вниз по слоям во время обратного прохода). В общей сложности мы имеем приблизительно  $525 + 36 + 3.6 = 564.6$  Мбайт и на самом деле это оптимистичный абсолютный минимум.

3. Если во время обучения сети CNN в вашем графическом процессоре случается нехватка памяти, то вот 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-битные значения с плавающей точкой вместо 32-битных;
  - 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 активируются наиболее сильно, 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 AlexNet гораздо крупнее и глубже и 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. Главное новшество ResNet — ввод обходящих связей, которые сделали возможным выход далеко за пределы 100 слоев. Вероятно, простоту и согласованность ResNet также можно считать новаторскими. Основным новшеством SENet была идея применения блоков SE (двухслойных плотных сетей) после каждого модуля начала в сети GoogLeNet или после каждого остаточного элемента в сети ResNet для повторной калибровки важности карт признаков. Наконец, главным новшеством Xception было использование сепарабельных слоев свертки по глубине, которые ищут пространственные и межканальные образы по отдельности.

- Полностью сверточные сети — это нейронные сети, состоящие исключительно из сверточных и объединяющих слоев. Сети FCN способны эффективно обрабатывать изображения любой ширины и высоты (по крайней мере, выше минимального размера). Они наиболее удобны для выявления объектов и семантической сегментации, потому что им необходимо просматривать изображение только раз (а не многократно прогонять сеть CNN по разным частям изображения). Если у вас есть сеть CNN с рядом плотных слоев наверху, то для создания сети FCN вы можете преобразовать эти плотные слои в сверточные: просто замените самый нижний плотный слой сверточным слоем с размером ядра, равным размеру входа слоя, с одним фильтром на нейрон в плотном слое и с дополнением "valid". Обычно страйд должен быть равен 1, но при желании можете установить его в более высокое значение. Функция активации должна быть такой же, как у плотного слоя. Остальные плотные слои должны быть преобразованы тем же способом, но использовать фильтры  $1 \times 1$ . На самом деле так можно преобразовать обученную сеть CNN, надлежащим образом изменяя форму матриц весов плотных слоев.
- Главная техническая трудность семантической сегментации связана с тем фактом, что большая часть пространственной информации утрачивается в сети CNN по мере прохождения сигнала через каждый слой, особенно в объединяющих слоях со страйдом больше 1. Этую пространственную информацию нужно каким-то образом восстанавливать, чтобы точно прогнозировать класс каждого пикселя.

Решения упражнений 9–12 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-ml2>.

## Глава 15. Обработка последовательностей с использованием рекуррентных и сверточных нейронных сетей

- 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–декодер”)

дировщик”), 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 системы выдачи рекомендаций).
  - 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. Слой RNN обязан иметь трехмерные входы: первое измерение является измерением пакета (его размер — размер пакета), второе измерение представляет время (его размер — количество временных шагов), а третье измерение содержит в себе входы на каждом временном шаге (его размер — количество входных признаков на временной шаг). Например, если вы хотите обрабатывать пакет из 5 временных рядов с 2 значениями на временной шаг (скажем, температура и скорость ветра), тогда формой будет [5, 10, 2]. Выходы тоже трехмерные, с теми же первыми двумя измерениями, но последнее измерение будет равно количеству нейронов. Например, если слой RNN с 32 нейронами обрабатывает пакет, который мы только что обсудили, то выход будет иметь форму [5, 10, 32].
3. Чтобы построить глубокую сеть RNN типа “последовательность в последовательность”, используя Keras, вы должны установить `return_sequences=True` для всех слоев RNN. Чтобы построить сеть RNN типа “последовательность в вектор”, вам потребуется установить `return_sequences=True` для всех слоев RNN кроме верхнего слоя RNN, который обязан иметь `return_sequences=False` (или вообще не устанавливайте этот аргумент, т.к. по умолчанию принимается `False`).

4. Если у вас есть ежедневный одномерный временной ряд, и вы хотите вырабатывать прогнозы для следующих семи дней, то простейшей архитектурой RNN будет стопка слоев RNN (все с `return_sequences=True` кроме верхнего слоя RNN), использующая семь нейронов в выходном слое RNN. Затем эту модель можно обучить с применением случайных окон из временных рядов (например, последовательностей из 30 следующих друг за другом дней как входов и вектора, содержащего значения последующих 7 дней, как цели). В итоге получилась сеть RNN типа “последовательность в вектор”. В качестве альтернативы вы могли бы установить `return_sequences=True` для всех слоев RNN, чтобы создать сеть RNN типа “последовательность в последовательность”. Такую модель можно обучить, используя случайные окна из временных рядов с последовательностями такой же длины, как входы, в качестве целей. Каждая целевая последовательность должна иметь семь значений на временной шаг (скажем, для временного шага  $t$  целью будет вектор, содержащий значения из временных шагов от  $t + 1$  до  $t + 7$ ).
5. Двумя основными затруднениями при обучении сетей RNN являются нестабильные градиенты (подверженные взрывному росту или исчезновению) и очень ограниченная краткосрочная память. При работе с длинными последовательностями эти проблемы усугубляются. Для смягчения проблемы нестабильных градиентов вы можете применять более низкую скорость обучения, использовать насыщаемую функцию активации, такую как гиперболический тангенс (принимаемую по умолчанию), и возможно задействовать отсечение градиентов, нормализацию по слою или отключение на каждом временном шаге. Чтобы решить проблему ограниченной краткосрочной памяти, вы можете применять слои LSTM или GRU (они также помогают справиться с проблемой нестабильных градиентов).
6. Архитектура ячейки LSTM выглядит сложной, но на самом деле она не особо трудна, если вы понимаете лежащую в основе логику. Ячейка имеет вектор краткосрочного состояния и вектор долгосрочного состояния. На каждом временном шаге входы и предыдущее краткосрочное состояние подаются простой ячейке RNN и трем шлюзам: шлюз забывания решает, что удалить из долгосрочного состояния, входной шлюз решает, какая часть выхода простой ячейки RNN должна быть добавле-

на к долгосрочному состоянию, и выходной шлюз решает, какая часть долгосрочного состояния должна выдаваться на этом временном шаге (после прохождения через функцию активации  $\tanh$ ). Новое краткосрочное состояние эквивалентно выходу ячейки. См. рис. 15.9.

7. Слой RNN является по существу последовательным: для вычисления выходов на временном шаге  $t$  он сначала должен вычислить выходы всех более ранних временных шагов. Это делает невозможным распараллеливание. С другой стороны, одномерный сверточный слой сам по себе хорошо подходит для распараллеливания, т.к. он не сохраняет состояние между временными шагами. Другими словами, он не имеет памяти: выход на любом временном шаге может быть вычислен на основе только небольшого окна значений из входов без необходимости знать все прошлые значения. Кроме того, поскольку одномерный сверточный слой не является рекуррентным, он меньше страдает от нестабильных градиентов. Один или большее число одномерных сверточных слоев могут быть полезны в сети RNN для эффективной предварительной обработки входов, скажем, чтобы уменьшить их временное разрешение (понижение дискретизации) и тем самым помочь слоям RNN выявлять долгосрочные образы. На самом деле можно использовать только сверточные слои, например, путем построения архитектуры WaveNet.
8. Одна возможная архитектура для классификации видеороликов на основе их визуального содержимого предусматривает захват (скажем) одного кадра в секунду, прогон каждого кадра через ту же самую сверточную нейронную сеть (например, заранее обученную модель Xception, возможно замороженную, если ваш набор данных не является крупным), передачу последовательности выходов из сети CNN в сеть RNN типа “последовательность в вектор” и прогон ее выхода через многопеременный логистический слой, дающий все вероятности классов. При обучении в качестве функции издержек можно было бы применять перекрестную энтропию. Если вы хотите использовать при классификации также и звук, то можете задействовать стопку одномерных сверточных слоев со страйдом, чтобы понизить временное разрешение с тысяч звукокадров в секунду до только одного в секунду (для соответствия количеству изображений в секунду), и объединять выходную последовательность с входами сети RNN типа “последовательность в вектор” (по последнему измерению).

Решения упражнений 9 и 10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12>.

## Глава 16. Обработка естественного языка с помощью рекуррентных нейронных сетей и внимания

1. Сеть RNN без запоминания состояния может захватывать только образы с длиной, меньше или равной размеру окон, на которых сеть RNN обучалась. И наоборот, сеть RNN с запоминанием состояния способна захватывать более долгосрочные образы. Однако реализовать сеть RNN с запоминанием состояния намного труднее — особенно надлежащим образом подготовить набор данных. Кроме того, сети RNN с запоминанием состояния не всегда работают лучше от части потому, что следующие друг за другом пакеты не являются независимыми и идентично распределенными. Градиентный спуск не испытывает нежных чувств к наборам данных с неидентичным распределением.
2. В целом 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 ate welcome*” (“Добро пожаловать”), но в результате перевода по одному слову за раз получится предложение “*I you in pray*” (“Я тебя умоляю”). Что-что? Гораздо лучше первоначально прочитать 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 частности как поступают люди-переводчики при переводе прямой передачи).
3. Входные последовательности переменной длины можно обрабатывать путем дополнения самых коротких последовательностей, чтобы все последовательности в пакете имели ту же самую длину, и использования маскирования для гарантии того, что сеть RNN проигнорирует дополняющий маркер. Для повышения эффективности можно также

создавать пакеты, содержащие последовательности похожих размеров. Зубчатые тензоры могут хранить последовательности переменной длины и, вероятнее всего, `t f . keras` со временем будет их поддерживать, что значительно упростит обработку входных последовательностей переменной длины (на момент написания книги это было не так). Что касается выходных последовательностей переменной длины, если длина выходной последовательности известна заранее (например, вы знаете, что она такая же, как у входной последовательности), тогда вам просто необходимо сконфигурировать функцию потерь, чтобы она игнорировала лексемы, поступающие после конца последовательности. Аналогично код, который будет использовать модель, должен игнорировать лексемы после конца последовательности. Но обычно длина выходной последовательности не будет известна заранее, поэтому решение предусматривает обучение модели, чтобы в конце каждой последовательности она выдавала маркер конца последовательности.

4. **Лучевой поиск** — это методика, применяемая для повышения эффективности обученной модели “кодировщик–декодировщик”, например, в системе нейронного машинного перевода. Алгоритм отслеживает короткий список из  $k$  наиболее многообещающих выходных предложений (скажем, верхних трех) и на каждом шаге декодировщика пытается расширить их на одно слово, сохраняя только  $k$  наиболее вероятных предложений. Параметр  $k$  называется *шириной луча*: чем он больше, тем больше будет использоваться ЦП и ОЗУ, но также более точной окажется система. Вместо того чтобы на каждом шаге жадно выбирать наиболее вероятное следующее слово для расширения единственного предложения, такая методика позволяет системе одновременно исследовать несколько многообещающих предложений. Кроме того, сама методика хорошо подходит для распараллеливания. Лучевой поиск можно легко реализовать с помощью `TensorFlow Addons`.
5. **Механизм внимания** — это методика, которая первоначально использовалась в моделях “кодировщик–декодировщик” для предоставления декодировщику более прямого доступа к входной последовательности, позволяя ему иметь дело с более длинными входными последовательностями. На каждом временном шаге декодировщика текущее состояние декодировщика и полный выход кодировщика обрабатываются

выровненной моделью, которая выдает меру выравнивания для каждого входного временного шага. Такая мера указывает, какая часть входа наиболее существенна для текущего временного шага декодировщика. Затем взвешенная (согласно мере выравнивания) сумма выхода кодировщика поддается декодировщику, который производит следующее состояние декодировщика и выход для данного временного шага. Главное преимущество применения механизма внимания связано с тем фактом, что модель “кодировщик–декодировщик” способна успешно обрабатывать более длинные входные последовательности. Еще одно преимущество заключается в том, что меры выравнивания облегчают отладку и интерпретацию модели: например, если модель допускает ошибку, то вы можете посмотреть, на каком входе было сосредоточено внимание, и это способно помочь в диагностике проблемы. Механизм внимания также лежит в основе архитектуры “Преобразователь”, в слоях многоголового внимания. См. решение следующего упражнения.

6. Самым важным слоем архитектуры “Преобразователь” является слой многоголового внимания (исходная архитектура “Преобразователь” содержит 18 таких слоев, включая 6 слоев маскированного многоголового внимания). Он лежит в основе языковых моделей вроде BERT и GPT-2. Его цель — позволить модели устанавливать, какие слова наиболее выровнены друг с другом, и затем улучшать представление каждого слова, используя такие контекстуальные подсказки.
7. Многопеременный прием с выборкой применяется при обучении модели для классификации, когда есть много классов (скажем, тысячи). Он предусматривает расчет приближенной величины потери перекрестной энтропии на основе логитов, спрогнозированных моделью для правильного класса, и спрогнозированных логитов для выборки из неправильных слов. В результате обучение значительно ускоряется в сравнении с вычислением многопеременной функции по всем логитам и последующей оценкой потери перекрестной энтропии. После обучения модель может нормально использоваться с применением обычной многопеременной логистической функции для расчета всех вероятностей классов на основе всех логитов.

Решения упражнений 8–11 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12>.

## Глава 17. Обучение представлению и порождению с использованием автокодировщиков и порождающих состязательных сетей

1. Ниже перечислено несколько главных задач, для которых применяются автокодировщики:
  - выделение признаков;
  - предварительное обучение без учителя;
  - понижение размерности;
  - порождающие модели;
  - обнаружение аномалий (автокодировщик обычно плох при восстановлении выбросов).
2. Если вы хотите обучить классификатор и располагаете массой непомеченных обучающих данных, но лишь несколькими тысячами помеченных образцов, тогда можете сначала обучить глубокий автокодировщик на полном наборе данных (помеченных и непомеченных), затем повторно использовать нижнюю половину слоев (т.е. слои до кодирующего включительно) для классификатора и обучить классификатор с применением помеченных данных. Если помеченных данных мало, тогда при обучении классификатора, возможно, вы захотите 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 эффективность автокодировщика, можно измерить потерю из-за реконструкции (например, подсчитать MSE, или средний квадрат результата вычитания входов из выходов). 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 эффективность 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 никаких полезных признаков.
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. 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 параметри-

зированы, генерируя только определенные виды выходов. Примером порождающего автокодировщика является вариационный автокодировщик.

7. Порождающая состязательная сеть — это нейросетевая архитектура, состоящая из двух частей, генератора и дискриминатора, которые председают противоположные цели. Целью генератора является создание образцов, похожих на те, что содержатся в обучающем наборе, для обмана дискриминатора. Дискриминатор обязан проводить различие между настоящими и сгенерированными образцами. На каждой итерации обучения дискриминатор обучается как нормальный двоичный классификатор, затем генератор обучается доведению до максимума ошибки дискриминатора. Сети GAN используются для решения сложных задач обработки изображений, таких как супер-разрешение, расцвечивание, редактирование изображений (замена объектов реалистичным фоном), превращение простого наброска в фотorealisticное изображение или прогнозирование следующих кадров в видеоролике. Они также применяются для дополнения набора данных (чтобы обучать другие модели), для генерации других типов данных (наподобие текста, аудиоклипов и временных рядов) и для идентификации слабых сторон в других моделях и их усиления.
8. Обучение сетей GAN общеизвестно трудно из-за сложных динамических отношений между генератором и дискриминатором. Наибольшей трудностью является коллапс мод, когда генератор производит выходы с очень малым разнообразием. Кроме того, обучение может оказаться крайне нестабильным: нормально стартовать, но затем внезапно начать колебаться или расходиться без какой-либо видимой причины. Сети GAN также очень чувствительны к выбору гиперпараметров.

Решения упражнений 9–11 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12.w>

## Глава 18. 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 обу-

чением с подкреплением, обычным 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 вырабатывания 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 буфер (память) воспроизведения, чтобы гарантировать получение алгоритмом обучения достаточно независимых наблюдений.

## 2. Ниже описаны некоторые возможные приложения обучения с подкреплением, отличающиеся от тех, что упоминались в главе 18.

### *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 в них загружаться со складов, куда они должны ехать, что из них нужно выгружать и т.д. Агент будет получать положительную награду за каждый доставленный вовремя товар и отрицательную награду за просроченную доставку.

- 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 коэффициент дисконтирования. Например, с коэффициентом дисконтирования 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 коэффициенте дисконтирования как о мере того, насколько будущее ценится в сравнении с настоящим. Если коэффициент дисконтирования очень близок к 1, то будущее ценится практически одинаково с настоящим. Если он близок 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 эффективности 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 часто может 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 буфер воспроизведения. Вместо применения при обучении только большей части непосредственного опыта агент будет учиться 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 коэффициента дисконтирования, который 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 узнает ценность политики, которую агент фактически исполняет, включая исследование и эксплуатацию.

Решения упражнений 8–10 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12>.

## Глава 19. Широкомасштабное обучение и развертывание моделей TensorFlow

1. Представление `SavedModel` содержит модель TensorFlow, включая ее архитектуру (вычислительный граф) и веса. Оно хранится в виде каталога с файлом `saved_model.pb`, где определен вычислительный граф (в форме сериализованного протокольного буфера), и подкаталога `variables` со значениями переменных. Для моделей с большим количеством весов значения переменных могут быть разнесены по множеству файлов. Представление `SavedModel` также включает подкаталог `assets`, который может содержать дополнительные данные, такие как файлы словарей, имена классов или примеры образцов для этой модели. Точнее говоря, представление `SavedModel` может содержать один или большее число метаграфов. Метаграф — это вычислительный граф плюс определения сигнатур функций (включающие имена входов, имена выходов, типы и формы). Каждый метаграф идентифицируется комплектом меток. Для инспектирования представления `SavedModel` вы можете использовать инструмент командной строки `saved_model_cli` или просто загрузить его с помощью `tf.saved_model.load()` и исследовать в Python.
2. Сервер TF `Serving` позволяет развертывать множество моделей TensorFlow (или множество версий одной модели) и делать их доступными всем приложениям через API-интерфейс REST или gRPC. Использование модели напрямую в приложениях затруднит развертывание новой версии модели среди всех приложений. Реализация собственной микрослужбы, являющейся оболочкой для модели TF, потребует дополнительной работы и сопряжено со сложностями в плане соответствия характеристикам сервера TF `Serving`. Сервер TF `Serving` обладает многими возможностями: он может отслеживать каталог

и автоматически развертывать помещаемые в него модели, при этом вам не придется изменять или даже перезапускать приложения, чтобы задействовать новые версии моделей; он является быстрым, хорошо протестированным и масштабируемым; он поддерживает тестирование А/Б экспериментальных моделей и развертывание новой версии модели только для подмножества пользователей. Сервер TF Serving также способен группировать индивидуальные запросы в пакеты, чтобы запускать их все вместе на ГП. Для ввода в действие сервера TF Serving вы можете установить его из исходного кода, но гораздо проще воспользоваться образом Docker. Чтобы развернуть кластер образов Docker сервера TF Serving, вы можете применить инструмент координации вроде Kubernetes или такое полнофункциональное решение, как платформа AI Platform инфраструктуры Google Cloud.

3. Для развертывания модели на множестве экземпляров TF Serving понадобится лишь сконфигурировать эти экземпляры TF Serving с целью отслеживания того же самого каталога `models` и затем экспортить новую модель в виде представления `SavedModel` в подкаталог.
4. API-интерфейс gRPC более эффективен, чем API-интерфейс REST. Однако его клиентские библиотеки доступны не настолько широко и если вы активизируете сжатие при использовании API-интерфейса REST, то сможете получить почти ту же производительность. Таким образом, API-интерфейс gRPC наиболее полезен, когда нужна самая высокая возможная производительность и клиенты не ограничены одним лишь API-интерфейсом REST.
5. Для уменьшения размера модели, чтобы ее можно было запускать на мобильном или встроенном устройстве, библиотека TFLite использует несколько методик.
  - Она предоставляет преобразователь, который способен оптимизировать представление `SavedModel`: сжать модель и уменьшить задержку. Чтобы сделать это, преобразователь отсекает все операции, которые не нужны при вырабатывании прогнозов (вроде операций обучения), а также по возможности оптимизирует и объединяет операции.
  - Преобразователь также может выполнять квантование после обучения: такой прием значительно уменьшает размер модели, поэтому она гораздо быстрее загружается и сохраняется.

- Она хранит оптимизированную модель в формате FlatBuffers, который может загружаться в ОЗУ напрямую безо всякого разбора. В результате сокращается время загрузки и объем занимаемой памяти.
6. Обучение с учетом квантования заключается в добавлении к модели фиктивных операций квантования во время обучения. Это дает возможность модели научиться игнорировать шум от квантования; финальные веса будут более устойчивыми к квантованию.
7. Параллелизм модели означает расщепление модели на части и параллельный их запуск на множестве устройств в надежде на то, что модель ускорится во время обучения или выведения. Параллелизм данных означает создание множества точных реплик модели и их развертывание на множестве устройств. На каждой итерации во время обучения каждой реплике предоставляется свой пакет данных, и она рассчитывает градиенты потери относительно параметров модели. При синхронном параллелизме данных градиенты из всех реплик затем собираются вместе, и оптимизатор выполняет шаг градиентного спуска. Параметры можно централизовать (например, на серверах параметров) или копировать по всем репликам и поддерживать в синхронизированном состоянии, используя AllReduce. При асинхронном параллелизме данных параметры централизуются и реплики выполняются независимо друг от друга, причем каждая реплика обновляет централизованные параметры напрямую в конце каждой итерации обучения, не ожидая остальных реплик. Что касается ускорения обучения, то в целом параллелизм данных оказывается лучше, чем параллелизм модели. Причина главным образом в том, что он требует меньше коммуникаций между устройствами. Кроме того, его гораздо легче реализовать и он работает одинаково для любой модели, тогда как параллелизм модели требует анализа модели с целью нахождения наилучшего способа ее расщепления на части.
8. При обучении модели на множестве серверов можно использовать следующие стратегии распределения.
- Стратегия MultiWorkerMirroredStrategy поддерживает параллелизм данных с зеркальным отображением. Модель реплицируется по всем доступным серверам и устройствам, при этом каждая реплика получает свой пакет данных на каждой итерации обучения и рас-

считывает собственные градиенты. Затем с применением реализации AllReduce (по умолчанию библиотеки NCCL) вычисляется средняя величина градиентов по всем репликам, и все реплики выполняют тот же самый шаг градиентного спуска. Такая стратегия является простейшей в использовании, поскольку все серверы и устройства трактуются совершенно одинаково, вдобавок она работает довольно хорошо. В общем случае вы должны придерживаться этой стратегии. Ее главное ограничение в том, что она требует, чтобы модель умещалась в ОЗУ на каждой реплике.

- Стратегия ParameterServerStrategy поддерживает асинхронный параллелизм данных. Модель реплицируется по всем устройствам на всех рабочих процессорах, а параметры разбиваются на все серверы параметров. Каждый рабочий процессор имеет собственный цикл обучения, выполняющийся асинхронно с остальными рабочими процессорами; на каждой итерации обучения каждый рабочий процессор получает собственный пакет данных и извлекает последнюю версию параметров модели из серверов параметров, рассчитывает градиенты относительно параметров модели и отправляет их серверам параметров. В заключение серверы параметров выполняют шаг градиентного спуска, используя эти градиенты. Такая стратегия в целом медленнее предыдущей стратегии и чуть сложнее в развертывании, потому что требует управления серверами параметров. Тем не менее, она удобна при обучении гигантских моделей, которые не умещаются в оперативную память ГП.

Решения упражнений 9–11 ищите в тетрадях Jupyter, доступных по адресу <https://github.com/ageron/handson-m12>.

# **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 приспосабливать 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 эффективность?
6. Согласована ли мера эффективности с бизнес-целью?
7. Какой будет минимальная эффективность, 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 произведите выборку, чтобы получить поддающийся управлению размер).
2. Создайте тетрадь Jupyter для сохранения записи об исследовании данных.
3. Изучите каждый атрибут и его характеристики:
  - о имя;
  - о тип (категориальный, целочисленный/с плавающей точкой, ограниченный/неограниченный, текстовый, структурированный и т.д.);
  - о процент отсутствующих значений;
  - о зашумленность и тип шума (стохастический, выбросы, ошибки округления и т.д.);
  - о полезность для задачи;
  - о 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 незатронутым).
- 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 гиперпараметров.

### 1. Очистите данные.

- Исправьте или удалите выбросы (необязательно).
- Заполните отсутствующие значения (например, нулевым, средним, медианным или каким-то другим значением) или отбросьте их строки (либо столбцы).

### 2. Выберите признаки (необязательно).

- Отбросьте атрибуты, которые не несут в себе никакой полезной информации 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.).
- Объедините признаки в перспективные новые признаки.

### 4. Масштабируйте признаки.

- Стандартизируйте или нормализуйте признаки.

# **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).
  - 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 эффективность.
    - К каждой модели примените перекрестную проверку с контролем по  $N$  блокам и вычислите среднее значение и стандартное отклонение меры эффективности 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. Выполните 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, если вы не уверены в том, чем должны заменяться отсутствующие значения — нулем или медианным значением, либо же нужно просто отбрасывать строки).
- Если значений гиперпараметров, которые подлежат исследованию, очень много, тогда отдавайте предпочтение случайному поиску перед решетчатым поиском. Когда обучение проходит слишком долго, вы можете предпочесть подход с байесовской оптимизацией (например, априорно используя гауссов процесс, как описано в работе Джаспера Сноека и др. (<https://homl.info/134>)<sup>1</sup>).

**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 эффективность на испытательном наборе, чтобы оценить ошибку обобщения.**



Не подстраивайте модель после измерения ошибки обобщения: вы просто начнете переобучать ее испытательным набором.

## Представление своего решения

**1. Документируйте все, что вы сделали.**

**2. Создайте привлекательную презентацию.**

- Удостоверьтесь в том, что сначала прояснили общую картину.

**3. Объясните, почему ваше решение достигает бизнес-цели.**

<sup>1</sup> Джаспер Сноек и др., *Practical Bayesian Optimization of Machine Learning Algorithms* (Практическая байесовская оптимизация алгоритмов машинного обучения), *Proceedings of the 25th International Conference on Neural Information Processing Systems* 2 (2012 г.): с. 2951–2959.

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 красивые 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 эффективность системы через регулярные промежутки времени и подавать сигналы тревоги, когда она падает.
  - Остерегайтесь медленного снижения эффективности: с развитием данных модели склонны к “разложению”.
  - Измерение эффективности может требовать конвейера человеческой оценки (например, через службу краудсорсинга).
  - Наблюдайте также за качеством входных данных (например, неисправный датчик может посыпать 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  $g(x, y, \alpha)$  относительно  $x$ ,  $y$  и  $\alpha$ , 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 В.1. Обобщенный лагранжиан для задачи классификации с жестким зазором

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} w^T w - \sum_{i=1}^m \alpha^{(i)} \left( t^{(i)} (w^T x^{(i)} + b) - 1 \right)$$
$$c \quad \alpha^{(i)} \geq 0 \quad \text{для } i = 1, 2, \dots, m.$$

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  $(\hat{w}, \hat{b}, \hat{\alpha})$ , которые удовлетворяют *условиям ККТ* (*KKT condition*).

- Соблюдают ограничения задачи:  $t^{(i)}((\hat{w})^T x^{(i)} + \hat{b}) \geq 1$  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:  $t^{(i)}((\hat{w})^T x^{(i)} + \hat{b}) = 1$ . 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 ККТ — 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 ККТ, гарантированно будет решением задачи условной оптимизации.

Мы можем вычислить частные производные обобщенного лагранжиана в отношении  $w$  и  $b$  с помощью уравнения В.2.

## Уравнение B.2. Частные производные обобщенного лагранжиана

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^m \alpha^{(i)} t^{(i)} x^{(i)}$$

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = - \sum_{i=1}^m \alpha^{(i)} t^{(i)}$$

Когда эти частные производные равны нулю, мы имеем уравнение B.3.

## Уравнение B.3. Свойства стационарных точек

$$\hat{w} = \sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} x^{(i)}$$

$$\sum_{i=1}^m \hat{\alpha}^{(i)} t^{(i)} = 0$$

Если мы включим полученные результаты в определение обобщенного лагранжиана, то некоторые члены исчезнут, и получится уравнение B.4.

## Уравнение B.4. Двойственная форма задачи SVM

$$\begin{aligned} \mathcal{L}(\hat{w}, \hat{b}, \alpha) &= \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} x^{(i)\top} x^{(j)} - \sum_{i=1}^m \alpha^{(i)} \\ &\text{с } \alpha^{(i)} \geq 0 \text{ для } i = 1, 2, \dots, m. \end{aligned}$$

Цель теперь в том, чтобы найти вектор  $\hat{a}$ , который сводит к минимуму эту функцию с  $\hat{\alpha}^{(i)} \geq 0$  для всех образцов. Такая задача условной оптимизации является искомой двойственной задачей.

После нахождения оптимального  $\hat{a}$  можно вычислить  $\hat{w}$ , используя первую строчку уравнения B.3. Чтобы вычислить  $\hat{b}$ , можно задействовать тот факт, что опорный вектор должен обеспечивать  $t^{(i)}((\hat{w})^\top x^{(i)} + \hat{b}) = 1$ , поэтому если опорным вектором является  $k$ -тый образец (т.е.  $\hat{\alpha}^{(k)} > 0$ ), тогда его можно применять для вычисления  $\hat{b} = t^{(k)} - \hat{w}^\top x^{(k)}$ . Тем не менее, предпочтение часто отдается вычислению среднего по всем опорным векторам, чтобы получить более устойчивое и точное значение, как показано в уравнении B.5.

## Уравнение B.5. Оценка члена смещения с использованием двойственной формы

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^m [t^{(i)} - \hat{w}^\top x^{(i)}]$$

# 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, конечно-разностное приближение (finite difference approximation) автоматическое дифференцирование в прямом режиме и автоматическое дифференцирование в обратном режиме. В библиотеке TensorFlow реализовано автоматическое дифференцирование в обратном режиме, но для его понимания полезно сначала взглянуть на другие варианты. Итак, давайте рассмотрим их, начиная с ручного дифференцирования.

## Ручное дифференцирование

Первый подход к вычислению производных — взять карандаш и лист бумаги и задействовать свое знание исчисления с целью вывода соответствующего уравнения. Для только что определенной функции  $f(x, y)$  задача не слишком трудна; необходимо лишь воспользоваться следующими пятью правилами.

- Производная константы равна 0.
- Производная  $\lambda x$  равна  $\lambda$  (где  $\lambda$  — константа).
- Производная  $x^\lambda$  равна  $\lambda x^{\lambda-1}$ , так что производной  $x^2$  будет  $2x$ .
- Производная суммы функций равна сумме производных этих функций.
- Производная  $\lambda$  раз функции равна  $\lambda$  раз ее производной.

Учитывая указанные правила, мы можем получить уравнение Г.1.

## Уравнение Г.1. Частные производные функции $f(x, y)$

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y \frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

Такой подход может стать крайне утомительным для более сложных функций, увеличивая риск допустить ошибку. К счастью, существуют другие варианты. Давайте посмотрим, что такое конечно-разностное приближение.

## Конечно-разностное приближение

Вспомните, что производная  $h'(x_0)$  функции  $h(x)$  в точке  $x_0$  представляет собой наклон кривой функции в этой точке. Выражаясь более точно, производная определяется как предел наклона прямой линии, проходящей через точку  $x_0$  и еще одну точку  $x$  на кривой функции, когда  $x$  становится бесконечно близкой к  $x_0$  (уравнение Г.2).

### Уравнение Г.2. Определение производной функции $h(x)$ в точке $x_0$

$$h'(x_0) = \lim_{x \rightarrow x_0} \frac{h(x) - h(x_0)}{x - x_0}$$
$$= \lim_{\varepsilon \rightarrow 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon}$$

Итак, если мы хотим вычислить частную производную  $f(x, y)$  относительно  $x$  в точке  $x = 3$  и  $y = 4$ , то можем рассчитать  $f(3 + \varepsilon, 4) - f(3, 4)$  и поделить результат на  $\varepsilon$ , используя для  $\varepsilon$  очень маленькое значение. Такой тип численной аппроксимации производной называется *конечно-разностным приближением*, а приведенное конкретное уравнение — *разностным отношением Ньютона (Newton's difference quotient)*. Это в точности то, что делает следующий 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 является неточным (и для более сложных функций становится еще худшим). Правильные результаты равны соответственно 24 и 10, но взамен мы получаем:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Обратите внимание, что для вычисления обеих частных производных мы обязаны вызывать функцию  $f()$ , по крайней мере, три раза (в предыдущем коде мы вызывали ее четыре раза, но это можно было бы оптимизировать). В случае 1 000 параметров нам пришлось бы вызывать  $f()$ , по крайней мере, 1 001 раз. Когда вы имеете дело с крупными нейронными сетями, способ с конечно-разностным приближением становится слишком неэффективным.

Однако данный метод настолько прост в реализации, что является замечательным инструментом для проверки, корректно ли реализованы другие методы. Например, если он не совпадает с вашей вручную выведенной функцией, то ваша функция, вероятно, содержит ошибку.

До сих пор мы обсудили два способа расчета градиентов: применение ручного дифференцирования и использование конечно-разностного приближения. К сожалению, у обоих были фатальные изъяны при обучении крупномасштабной нейронной сети. Таким образом, давайте перейдем к автоматическому дифференцированию, начиная с прямого режима.

## Автоматическое дифференцирование в прямом режиме

На рис. Г.1 показано, как автоматическое дифференцирование в прямом режиме работает с более простой функцией  $g(x, y) = 5 + xy$ . Слева приведен график для этой функции. После автоматического дифференцирования в прямом режиме мы получаем график справа, который представляет частную производную  $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$  (похожим способом можно было бы получить частную производную относительно  $y$ ).

Алгоритм проходит по вычислительному графу от входов до выходов (отсюда и название “в прямом режиме”). Он начинает с взятия частных производных листовых узлов. Узел константы (5) возвращает константу 0, т.к. производная константы всегда равна 0.

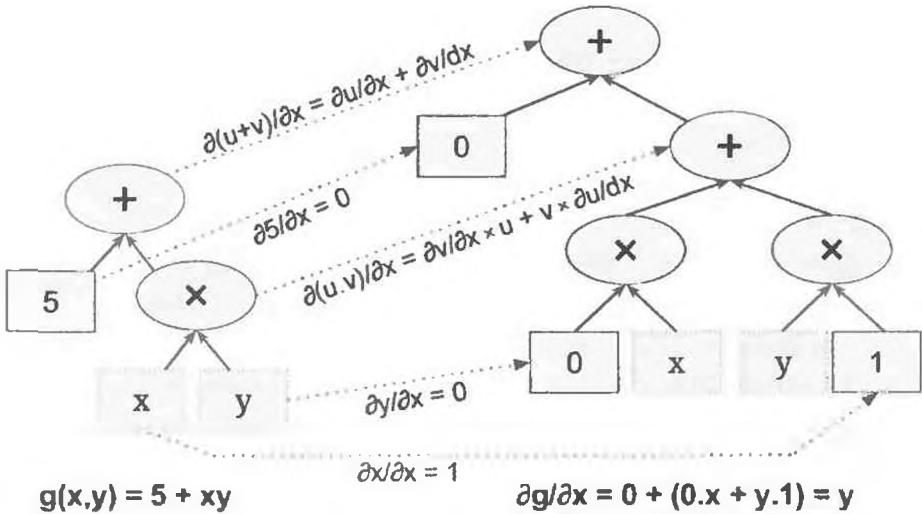


Рис. Г.1. Автоматическое дифференцирование в прямом режиме

Узел переменной  $x$  возвращает константу 1, поскольку  $\frac{\partial x}{\partial x} = 1$ , а узел переменной  $y$  — 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:  $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$ .

Тем не менее, это уравнение может быть упрощено (и значительно). К вычислительному графу можно применить несколько 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 сложной функции автоматическое дифференцирование в прямом режиме может давать гигантский граф, трудно поддающийся упрощению, и приводить к субоптимальной эффективности.

Обратите внимание, что мы начали с вычислительного графа, и дифференцирование в прямом режиме выпустило еще один вычислительный граф. Прием называется *символьным дифференцированием* (*symbolic differentiation*) и обладает двумя интересными характеристиками: во-первых, после того, как вычислительный граф производной был создан, мы можем использовать его любое желаемое количество раз для расчета производных заданной функции для любого значения  $x$  и  $y$ ; во-вторых, мы снова можем запустить автоматическое дифференцирование в прямом режиме на результирующем графе, чтобы получить производные второго порядка (т.е. производные производных), когда они нужны. Мы могли бы даже вычислить производные третьего порядка и т.д.

Но автоматическое дифференцирование в прямом режиме можно также запускать, не создавая граф (т.е. численно, не в символьной форме), просто рассчитывая промежуточные результаты на лету. Один из способов делать это предусматривает применение *дуальных чисел*, которые (странны, но притягательно) представляют собой числа в форме  $a + b\epsilon$ , где  $a$  и  $b$  — вещественные числа, а  $\epsilon$  — 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  $42.0000\ldots 000024$  с бесконечным количеством нулей (разумеется, такое 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

$$\begin{aligned}\lambda(a + b\epsilon) &= \lambda a + \lambda b\epsilon \\ (a + b\epsilon) + (c + d\epsilon) &= (a + c) + (b + d\epsilon) \\ (a + b\epsilon) \times (c + d\epsilon) &= ac + (ad + bc\epsilon) + (bd)\epsilon^2 = ac + (ad + bc)\epsilon\end{aligned}$$

Наиболее важно то, что 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 иллюстрируется вычисление частной производной  $f(x, y)$  относительно  $x$  в точке  $x = 3$  и  $y = 4$  с использованием дуальных чисел. 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 с использованием дуальных чисел**

Таким образом, автоматическое дифференцирование в прямом режиме гораздо точнее конечно-разностного приближения, но оно страдает от того же самого серьезного недостатка, по крайней мере, когда есть много входов и мало выходов (как происходит в случае работы с нейронными сетями): при наличии 1000 параметров для вычисления всех частных производных потребовалось бы 1000 проходов 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 рассчитать значение каждого узла. Затем автоматическое дифференцирование в обратном режиме осуществляет второй проход, но в противоположном направлении (т.е. от выходов к входам), чтобы вычислить все частные производные. Название “обратный режим” происходит от второго прохода через граф, когда градиенты протекают в обратном направлении. 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  $n_7$ :  $f(3, 4) = n_7 = 42$ .

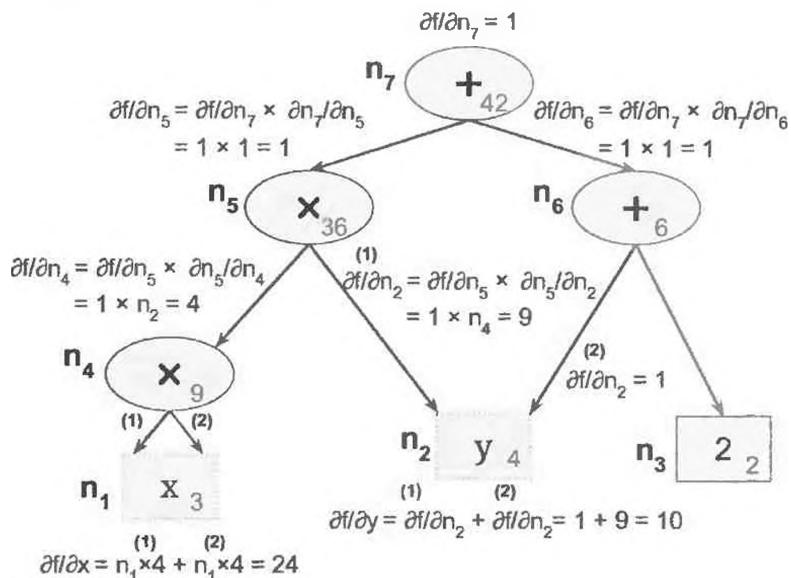


Рис. Г.3. Автоматическое дифференцирование в обратном режиме

Идея заключается в том, чтобы постепенно двигаться вниз по графу, вычисляя частную производную  $f(x, y)$  относительно следующих друг за другом узлов, 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  $\frac{\partial f}{\partial n_7} = 1$ .

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 мощной и точной методикой, 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. При обучении нейронных сетей мы обычно хотим свести к минимуму потерю, так что есть единственный выход (потеря) и потому для расчета градиентов необходимы только два прохода через граф. Автоматическое дифференцирование в обратном режиме также способно обрабатывать функции, которые не являются полностью дифференцируемыми, при условии, что вычисление частных производных запрашивается в точках, где функции дифференцируемы.

На рис. Г.3 числовые результаты вычислялись на лету в каждом узле. Однако это не совсем то, что делает библиотека TensorFlow: она взамен создает новый вычислительный график. Другими словами, она реализует *символьное* автоматическое дифференцирование в обратном режиме. Таким образом, вычислительный график для расчета градиентов потери относительно всех параметров в нейронной сети потребуется генерировать только однократно, после чего его можно запускать снова и снова всякий раз, когда оптимизатору нужно рассчитывать градиенты. Кроме того, появляется возможность вычисления производных более высокого порядка, если они необходимы.



Если вы когда-нибудь пожелаете реализовать новый тип низкоуровневой операции TensorFlow на C++ и хотите сделать его совместимым с автоматическим дифференцированием, тогда вам потребуется предоставить функцию, которая возвращает частные производные выходов функции относительно ее входов. Например, пусть вы реализовали функцию, вычисляющую квадрат своего входа,  $f(x) = x^2$ . В таком случае вам придется предоставить соответствующую производную функцию:  $f'(x) = 2x$ .

# Другие популярные архитектуры искусственных нейронных сетей

В этом приложении представлен краткий обзор нескольких исторически важных архитектур нейронных сетей, которые в наши дни используются намного реже, чем глубокие многослойные персепtronы (глава 10), сверточные нейронные сети (глава 14), рекуррентные нейронные сети (глава 15) или автокодировщики (глава 17). Они часто упоминаются в литературе, и некоторые из них все еще применяются в ряде приложений, а потому такие архитектуры полезно знать. Вдобавок мы обсудим *глубокие сети доверия* (*deep belief nets* — DBN), которые отражали современное состояние глубокого обучения до начала 2010-х годов. Они по-прежнему являются предметом очень активных исследований, поэтому в будущем вполне могут вернуться с удвоенной силой.

## Сети Хопфилда

*Сети Хопфилда* (*Hopfield network*) впервые были введены Уильямом Литтлом в 1974 году и затем популяризированы Джоном 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 образец из числа тех, которые она уже знает. 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.

Сети Хопфилда представляют собой полносвязные графы (рис. Д.1), т.е. каждый нейрон связан со всеми остальными нейронами. Обратите внимание, что изображения имеют  $6 \times 6$  пикселей, поэтому нейронная сеть слева должна содержать 36 нейронов (и 630 связей), но для зрительной ясности показана гораздо меньшая сеть.

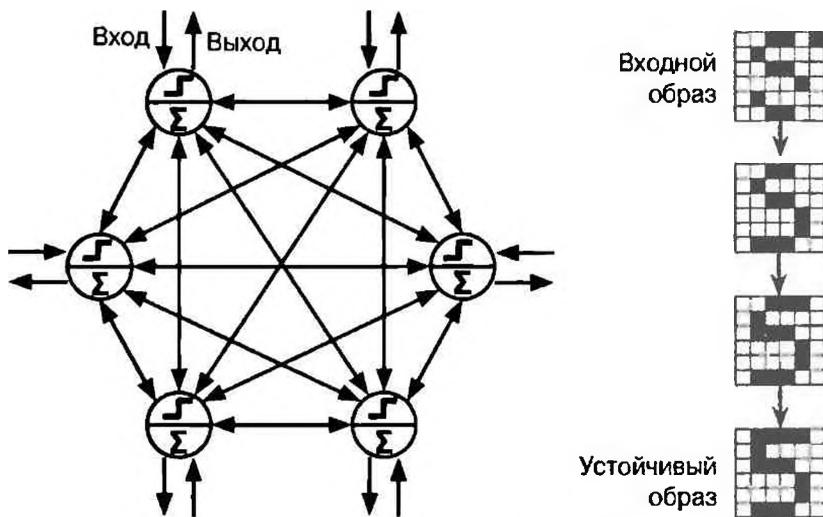


Рис. Д.1. Сеть Хопфилда

Алгоритм обучения работает за счет использования правила Хебба (см. раздел “Персептрон” в главе 10): 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 сни-

жает 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 patterns*).

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  $28 \times 28$ , 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 Джоном Хинтоном и Терренсом Сейновски. Как и сети Хопфилда, они являются полносвязными искусственными нейронными сетями, но основаны на *стохастических нейронах* (*stochastic neurons*): 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 (применяется в статистической механике), отсюда и такое название. Уравнение Д.1 дает вероятность того, что определенный нейрон будет выдавать 1.

**Уравнение Д.1. Вероятность того, что  $i$ -тый нейрон будет выдавать 1**

$$p(s_i^{(\text{следующий шаг})} = 1) = \sigma\left(\frac{\sum_{j=1}^N w_{ij} s_j + b_i}{T}\right)$$

Разберем это уравнение.

- $s_j$  — состояние  $j$ -того нейрона (0 или 1).
- $w_{ij}$  — вес связи между  $i$ -тым и  $j$ -тым нейронами. Обратите внимание, что  $w_{ii} = 0$ .

- $b_i$  — 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%).
- $\sigma$  — 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.

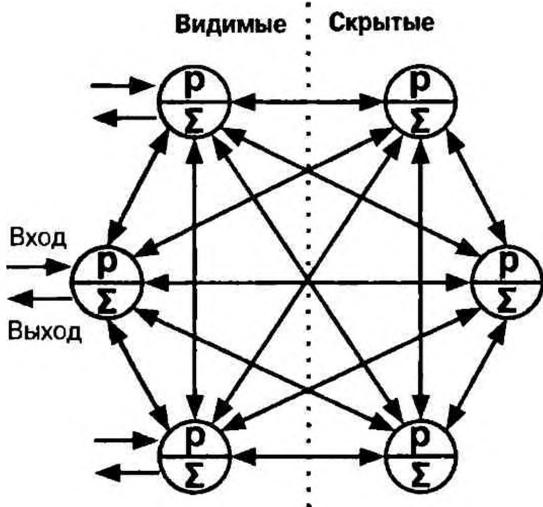


Рис. Д.2. Машина Больцмана

Из-за своей стохастической природы машина Больцмана никогда не стабилизируется в фиксированную конфигурацию; взамен она 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 такого состояния, в котором первона-

чальная конфигурация “забыта”, 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 позволяют 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*).

## Ограниченнные машины Больцмана

Ограниченнная 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.

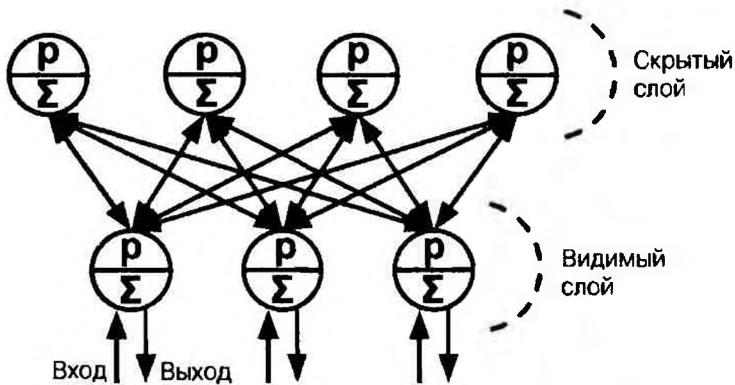


Рис. Д.3. Ограниченнная машина Больцмана

В 2005 году Мигель Кэррера-Перпиньян и Джекфри Хинтон предложили очень эффективный алгоритм обучения, названный *сопоставительным отклонением* (*Contrastive Divergence*; <https://homl.info/135>)<sup>1</sup>. Вот как он работает: для каждого обучающего образца  $x$  алгоритм начинает с того, что передает его сети, устанавливая состояние видимых элементов в  $x_1, x_2, \dots, x_n$ . Затем вычисляется состояние скрытых элементов путем применения статистического уравнения, описанного ранее (см. уравнение Д.1). Это дает скрытый вектор  $h$  (где  $h_i$  — состояние  $i$ -го элемента). Далее вычисляется состояние видимых элементов с использованием того же самого статистического уравнения. Это дает вектор  $x'$ . Потом еще раз вычисляется состояние скрытых элементов, что дает вектор  $h'$ . Теперь можно обновить вес каждой связи, применив правило из уравнения Д.2, где  $\eta$  — скорость обучения.

#### Уравнение Д.2. Обновление весов методом сопоставительного отклонения

$$w_{i,j} \leftarrow w_{i,j} + \eta \left( x h^T - x' h'^T \right)$$

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.

<sup>1</sup> Мигель Кэррера-Перпиньян и Джекфри Хинтон, *On Contrastive Divergence Learning* (Об обучении методом сопоставительного отклонения), *Proceedings of the 10th International Workshop on Artificial Intelligence and Statistics* (2005 г.): с. 59–66.

# Глубокие сети доверия

Несколько слоев машин RBM можно укладывать 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*).

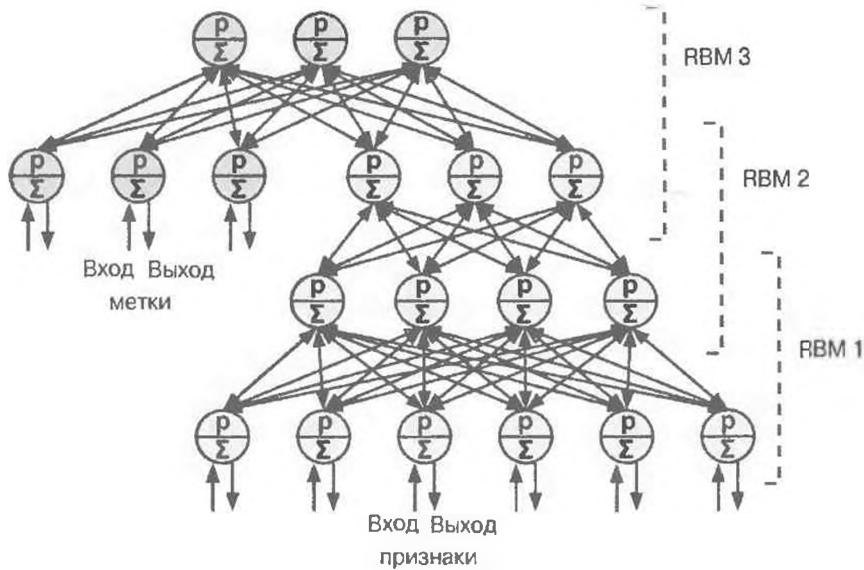
И-Вай Ти, один из студентов Джейфри Хинтона, опытным путем выяснил, что сети DBN можно обучать с помощью алгоритма сопоставительного отклонения по одному слою за раз, начиная с самых нижних слоев и постепенно перемещаясь к верхним слоям. Это вылилось в революционную статью, которая дала толчок началу цунами глубокого обучения в 2006 году (<https://homl.info/136>)<sup>2</sup>.

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.

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 стопкой большее количество машин RBM, но идея должна быть ясна. До сих пор это было полностью обучением без учителя.

<sup>2</sup> Джейфри Хинтон и др., *A Fast Learning Algorithm for Deep Belief Nets* (Быстрый алгоритм обучения для глубоких сетей доверия), *Neural Computation* 18 (2006 г.): с. 1527–1554.



*Рис. Д.4. Глубокая сеть доверия, сконфигурированная для частичного обучения*

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 выветится соответствующая метка. Именно так сеть DBN может применяться для классификации.

Большое преимущество такого частичного обучения заключается в том, что не требуется иметь 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 обычновенный образец класса, чей меточный элемент вы активировали. Такая порождающая способность сетей DBN на самом деле обладает большой мощью. 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 надпись "автомобиль"). После этого поверх обеих сетей DBN укладывалась машина RBM, которая обучалась с помощью набора изображений вместе с их надписями; она учились 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 изображения<sup>3</sup>.

## 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.

---

<sup>3</sup> Дополнительные детали и демонстрацию ищите в видеоролике Джекфри Хинтона: <https://homl.info/137>.

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).

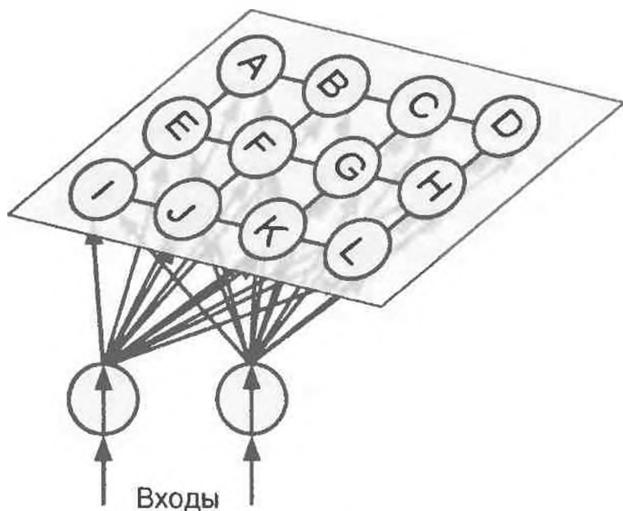


Рис. Д.5. Самоорганизующиеся карты

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. 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 подается в сеть. 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 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 тоже обновляют свой весовой вектор, чтобы слегка приблизится к входному вектору (но они не обновляют свои веса настолько 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 входах<sup>4</sup>.

<sup>4</sup> 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.

# Специальные структуры данных

В этом приложении мы кратко обсудим поддерживаемые TensorFlow структуры данных, выходящие за рамки обычных тензоров с плавающей точкой или целочисленных тензоров. К ним относятся строки, зубчатые тензоры, разреженные тензоры, тензорные массивы и очереди.

## Строки

Тензоры могут содержать в себе байтовые строки, которые особенно удобны для обработки естественного языка (см. главу 16):

```
>>> tf.constant(b"hello world")
<tf.Tensor: id=149, shape=(), dtype=string, numpy=b'hello world'>
```

Если вы попытаетесь создать тензор со строкой Unicode, то библиотека TensorFlow автоматически закодирует его как UTF-8:

```
>>> tf.constant("café")
<tf.Tensor: id=138, shape=(), dtype=string, numpy=b'caf\xc3\xa9'>
```

Также можно создавать тензоры, представляющие строки Unicode. Просто создайте массив 32-битных целых чисел, каждое из которых представляет одиночную кодовую точку Unicode<sup>1</sup>:

```
>>> tf.constant([ord(c) for c in "café"])
<tf.Tensor: id=211, shape=(4,), dtype=int32,
numpy=array([ 99, 97, 102, 233], dtype=int32)>
```



В тензорах типа `tf.string` длина строки не является частью формы тензора. Другими словами, строки считаются атомарными значениями. Однако в строковом тензоре Unicode (т.е. тензоре `int32`) длина строки будет частью формы тензора.

---

<sup>1</sup> Если вы не знаете, что такое кодовые точки Unicode, тогда ознакомьтесь с введением в Unicode по ссылке <https://homl.info/unicode>.

Пакет `tf.strings` содержит несколько функций для манипулирования строковыми тензорами, такие как `length()` для подсчета количества байтов в байтовой строке (или количество кодовых точек, если установить `unit="UTF8_CHAR"`), `unicode_encode()` для преобразования строкового тензора Unicode (т.е. тензора `int32`) в байтовый строковый тензор и `unicode_decode()` для обратного преобразования:

```
>>> b = tf.strings.unicode_encode(u, "UTF-8")
>>> tf.strings.length(b, unit="UTF8_CHAR")
<tf.Tensor: id=386, shape=(), dtype=int32, numpy=4>
>>> tf.strings.unicode_decode(b, "UTF-8")
<tf.Tensor: id=393, shape=(4,), dtype=int32,
    numpy=array([ 99, 97, 102, 233], dtype=int32)>
```

Вы можете также манипулировать тензорами, содержащими множество строк:

```
>>> p = tf.constant(["Café", "Coffee", "caffè", "咖啡"])
>>> tf.strings.length(p, unit="UTF8_CHAR")
<tf.Tensor: id=299, shape=(4,), dtype=int32,
    numpy=array([4, 6, 5, 2], dtype=int32)>
>>> r = tf.strings.unicode_decode(p, "UTF8")
>>> r
tf.RaggedTensor(values=tf.Tensor(
[ 67 97 102 233 67 111 102 102 101 101 99 97
 102 102 232 21654 21857], shape=(17,), dtype=int32),
    row_splits=tf.Tensor([ 0 4 10 15 17], shape=(5,), dtype=int64))
>>> print(r)
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
    [99, 97, 102, 102, 232], [21654, 21857]]>
```

Обратите внимание, что декодированные строки хранятся в `RaggedTensor`. Что это такое?

## Зубчатые тензоры

Зубчатый тензор — это специальный вид тензора, который представляет список массивов разных размеров. В более общем смысле он является тензором с одним и более зубчатых измерений, т.е. измерений, чьи срезы могут иметь отличающиеся длины. В зубчатом тензоре `r` зубчатым будет второе измерение. Во всех зубчатых тензорах первое измерение всегда будет обычным (также называется *однородным измерением*).

Все элементы зубчатого тензора `r` являются обычными тензорами. Например, давайте взглянем на второй элемент зубчатого тензора:

```
>>> print(r[1])
tf.Tensor([ 67 111 102 102 101 101], shape=(6,), dtype=int32)
```

Пакет `tf.ragged` содержит несколько функций для создания и манипулирования зубчатыми тензорами. Ниже мы создаем зубчатый тензор с использованием `tf.ragged.constant()` и сцепляем его с первым зубчатым тензором вдоль оси 0:

```
>>> r2 = tf.ragged.constant([[65, 66], [], [67]])
>>> print(tf.concat([r, r2], axis=0))
<tf.RaggedTensor [[67, 97, 102, 233], [67, 111, 102, 102, 101, 101],
[99, 97, 102, 102, 232], [21654, 21857], [65, 66], [], [67]]>
```

Результат не слишком удивительный: тензоры в `r2` добавились после тензоров в `r` вдоль оси 0. Но что, если мы сцепим `r` и еще один зубчатый тензор вдоль оси 1?

```
>>> r3 = tf.ragged.constant([[68, 69, 70], [71], [], [72, 73]])
>>> print(tf.concat([r, r3], axis=1))
<tf.RaggedTensor [[67, 97, 102, 233, 68, 69, 70], [67, 111, 102,
102, 101, 101, 71], [99, 97, 102, 102, 232], [21654, 21857, 72, 73]]>
```

Обратите внимание, что на этот раз были сцеплены  $i$ -тый тензор в `r` и  $i$ -тый тензор в `r3`. Теперь все выглядит не так типично, поскольку все тензоры могут иметь разные длины.

Вызов метода `to_tensor()` приводит к преобразованию его в обычный тензор с дополнением более коротких тензоров нулями, чтобы получить тензоры с одинаковой длиной (изменить стандартное значение можно установкой аргумента `default_value`):

```
>>> r.to_tensor()
<tf.Tensor: id=1056, shape=(4, 6), dtype=int32, numpy=
array([[ 67,    97,   102,   233,      0,      0],
       [ 67,   111,   102,   102,   101,   101],
       [ 99,    97,   102,   102,   232,      0],
       [21654, 21857,      0,      0,      0,      0]], dtype=int32)>
```

Зубчатые тензоры поддерживают многие операции ТФ. Полный список ищите в документации по классу `tf.RaggedTensor`.

## Разреженные тензоры

Библиотека TensorFlow также способна эффективно представлять *разреженные тензоры* (т.е. тензоры, содержащие главным образом нули). Просто создайте объект `tf.SparseTensor`, указав индексы и значения ненулевых элементов и форму тензора. Индексы должны перечисляться в “порядке чтения” (слева направо и сверху вниз). Если вы не уверены, тогда применяйте `tf.sparse.reorder()`. Разреженный тензор можно преобразовать в плотный тензор (т.е. обычновенный тензор), используя `tf.sparse.to_dense()`:

```
>>> s = tf.SparseTensor(indices=[[0, 1], [1, 0], [2, 3]],
                         values=[1., 2., 3.],
                         dense_shape=[3, 4])
>>> tf.sparse.to_dense(s)
<tf.Tensor: id=1074, shape=(3, 4), dtype=float32, numpy=
array([[0., 1., 0., 0.],
       [2., 0., 0., 0.],
       [0., 0., 0., 3.]], dtype=float32)>
```

Обратите внимание, что разреженные тензоры не поддерживают настолько много операций, как плотные тензоры. Например, вы можете умножить разреженный тензор на любое скалярное значение и получить новый разреженный тензор, но не можете сложить скалярное значение и разреженный тензор, т.к. результатом не будет разреженный тензор:

```
>>> s * 3.14
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x13205d470>
>>> s + 42.0
[...] TypeError: unsupported operand type(s) for +:
'SparseTensor' and 'float'
Ошибка типа: неподдерживаемые типы операндов для +:
SparseTensor и float
```

## Тензорные массивы

Класс `tf.TensorArray` представляет список тензоров. Он может быть удобным в динамических моделях, содержащих циклы, для накопления результатов и расчета статистических данных в более позднее время. Вы можете читать или записывать тензоры в любое место массива:

```
array = tf.TensorArray(dtype=tf.float32, size=3)
array = array.write(0, tf.constant([1., 2.]))
array = array.write(1, tf.constant([3., 10.]))
array = array.write(2, tf.constant([5., 7.]))
tensor1 = array.read(1)      # => возвращает (и выталкивает!)
                            #  tf.constant([3., 10.])
```

Обратите внимание, что чтение элемента выталкивает его из массива и заменяет тензором той же формы, заполненным нулями.



При записи в массив вы должны присваивать результат обратно массиву, как демонстрировалось в приведенном выше примере кода. Если этого не сделать, то код будет нормально работать в энергичном режиме, но не в режиме графа (упомянутые режимы обсуждались в главе 12).

При создании объекта `TensorArray` вам потребуется указать его размер (аргумент `size`), исключая режим графа. В качестве альтернативы можете оставить аргумент `size` неустановленным и взамен установить `dynamic_size=True`, но это приведет к снижению производительности, так что если размер известен заранее, то вы обязаны установить его. Кроме того, понадобится указать `dtype`, и все элементы должны иметь такую же форму, как элемент, записанный в массив первым.

Вы можете упаковать все элементы в обычновенный тензор, вызвав метод `stack()`:

```
>>> array.stack()
<tf.Tensor: id=2110875, shape=(3, 2), dtype=float32, numpy=
array([[1., 2.],
       [0., 0.],
       [5., 7.]], dtype=float32)>
```

## Множества

Библиотека TensorFlow поддерживает множества целых чисел или строк (но не чисел с плавающей точкой). Они представляются с применением обычновенных тензоров. Скажем, множество {1, 5, 9} представлено просто как тензор `[[1, 5, 9]]`. Обратите внимание, что тензор обязан иметь хотя бы два измерения, а множества должны быть в последнем измерении. Например, `[[1, 5, 9], [2, 5, 11]]` — это тензор, содержащий в себе два независимых множества: {1, 5, 9} и {2, 5, 11}. Если некоторые множест-

ва короче остальных, то их потребуется дополнить каким-то значением (по умолчанию 0, но при желании можно использовать любое другое значение).

Пакет `tf.sets` содержит несколько функций для манипулирования множествами. Давайте создадим два множества и получим их объединение (результатом будет разреженный тензор, поэтому для его отображения мы вызываем `to_dense()`):

```
>>> a = tf.constant([[1, 5, 9]])
>>> b = tf.constant([[5, 6, 9, 11]])
>>> u = tf.sets.union(a, b)
>>> u
<tensorflow.python.framework.sparse_tensor.SparseTensor at 0x132b60d30>
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11]], dtype=int32)>
```

Можно также вычислять объединение нескольких пар множеств одновременно:

```
>>> a = tf.constant([[1, 5, 9], [10, 0, 0]])
>>> b = tf.constant([[5, 6, 9, 11], [13, 0, 0, 0, 0]])
>>> u = tf.sets.union(a, b)
>>> tf.sparse.to_dense(u)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13,  0,  0]], dtype=int32)>
```

Если вы предпочитаете применять другое заполняющее значение, тогда при вызове `to_dense()` должны установить аргумент `default_value`:

```
>>> tf.sparse.to_dense(u, default_value=-1)
<tf.Tensor: [...] numpy=array([[ 1,  5,  6,  9, 11],
                               [ 0, 10, 13, -1, -1]], dtype=int32)>
```



Стандартным значением `default_value` является 0, так что при работе с множествами строк устанавливать `default_value` обязательно (скажем, в пустую строку).

В число других функций, доступных в пакете `tf.sets`, входят `difference()`, `intersection()` и `size()`, которые не требуют объяснений. Если вы хотите проверить, содержит ли множество заданные значения, то можете получить пересечение этого множества и значений. Если вы хотите добавить значения в множество, тогда можете вычислить объединение множества и значений.

# Очереди

Очередь представляет собой структуру данных, в которую можно помещать записи данных и позже извлекать их. Библиотека TensorFlow реализует несколько типов очередей в своем пакете `tf.queue`. Раньше они были очень важны при реализации эффективных конвейеров загрузки и предварительной обработки данных, но API-интерфейс `tf.data` фактически сделал их бесполезными (кроме возможно ряда редких случаев), поскольку его гораздо проще использовать и он предлагает все инструменты, которые необходимы для построения эффективных конвейеров. Тем не менее, ради полноты давайте кратко рассмотрим их.

Простейшим видом очереди является очередь типа “первым пришел — первым обслужен” (`first-in, first-out` — FIFO). Чтобы построить ее, вам придется указать максимальное количество записей, которые она будет содержать. Кроме того, каждая запись является кортежем тензоров, поэтому вы обязаны указать тип каждого тензора и дополнительно его форму. Скажем, в следующем примере кода создается очередь FIFO с максимумом тремя записями, каждая из которых содержит кортеж с 32-битным целым числом и строкой. Затем в очередь помещаются две записи, выясняется размер (2 в данный момент) и одна запись извлекается:

```
>>> q = tf.queue.FIFOQueue(3, [tf.int32, tf.string], shapes=[(), ()])
>>> q.enqueue([10, b"windy"])
>>> q.enqueue([15, b"sunny"])
>>> q.size()
<tf.Tensor: id=62, shape=(), dtype=int32, numpy=2>
>>> q.dequeue()
[<tf.Tensor: id=6, shape=(), dtype=int32, numpy=10>,
 <tf.Tensor: id=7, shape=(), dtype=string, numpy=b'windy'>]
```

Можно также помещать и извлекать из очереди сразу несколько записей (последнее требует указания форм при создании очереди):

```
>>> q.enqueue_many([[13, 16], [b'cloudy', b'raining']])
>>> q.dequeue_many(3)
[<tf.Tensor: [...] numpy=array([15, 13, 16], dtype=int32)>,
 <tf.Tensor: [...] numpy=array([b'sunny', b'cloudy', b'raining'], dtype=object)>]
```

Ниже перечислены другие типы очередей.

## **PaddingFIFOQueue**

Такая же очередь, как FIFOQueue, но ее метод `dequeue_many()` поддерживает извлечение из очереди множества записей с разными формами. Она автоматически дополняет самые короткие записи, гарантируя в итоге, что все записи в пакете имеют ту же самую форму.

## **PriorityQueue**

Очередь, записи которой извлекаются в соответствии с приоритетом. Приоритет должен быть 64-битным целым числом, включенным в виде первого элемента каждой записи. Удивительно, но записи с самым низким приоритетом будут извлекаться первыми. Записи с одинаковым приоритетом будут извлекаться в порядке FIFO.

## **RandomShuffleQueue**

Очередь, записи которой извлекаются в случайном порядке. Она была удобной для реализации буфера тасования до появления `tf.data`.

Если очередь уже полна, и вы пытаетесь поместить в нее еще одну запись, то метод `enqueue*` () замораживается до тех пор, пока из очереди не будет извлечена какая-нибудь запись в другом потоке. Аналогично, если очередь пуста, и вы пытаетесь извлечь из нее запись, то метод `dequeue*` () замораживается до тех пор, пока в другом потоке в очередь не будут помещены записи.

# Графы TensorFlow

В этом приложении мы исследуем графы, генерируемые функциями TF Function (см. главу 12).

## Функции TF Function и конкретные функции

Функции TF Function являются полиморфными, что означает поддержку ими входов разных типов (и форм). Например, рассмотрим следующую функцию `tf_cube()`:

```
@tf.function
def tf_cube(x):
    return x ** 3
```

Каждый раз, когда вы вызываете функцию TF Function с новой комбинацией типов или форм входов, она генерирует новую *конкретную функцию* (*concrete function*) с собственным графом, специализированным для этой особой комбинации. Такая комбинация типов и форм аргументов называется *сигнатурой входов* (*input signature*). В случае вызова функции TF Function с сигнатурой входов, которую она уже видела, будет повторно использоваться конкретная функция, сгенерированная ранее. Скажем, если вы вызовите `tf_cube(tf.constant(3.0))`, то функция TF Function повторно применит ту же самую конкретную функцию, которую она использовала для `tf_cube(tf.constant(2.0))` (для скалярных тензоров `float32`). Но она будет генерировать новую конкретную функцию при вызове `tf_cube(tf.constant([2.0]))` или `tf_cube(tf.constant([3.0]))` (для тензоров `float32` с формой `[1]`) и еще одну в случае вызова `tf_cube(tf.constant([[1.0, 2.0], [3.0, 4.0]]))` (для тензоров `float32` с формой `[2, 2]`). Получить конкретную функцию для специфической комбинации входов можно с помощью метода `get_concrete_function()` функции TF Function. Затем ее можно вызы-

вать подобно обычновенной функции, но она будет поддерживать только одну сигнатуру входов (в этом примере скалярные тензоры float32):

```
>>> concrete_function = tf_cube.get_concrete_function(tf.constant(2.0))
>>> concrete_function
<tensorflow.python.eager.function.ConcreteFunction at 0x155c29240>
>>> concrete_function(tf.constant(2.0))
<tf.Tensor: id=19068249, shape=(), dtype=float32, numpy=8.0>
```

На рис. Ж.1 показана функция TF Function по имени `tf_cube()` после вызовов `tf_cube(2)` и `tf_cube(tf.constant(2.0))`: были сгенерированы две конкретные функции, по одной для каждой сигнатуры, каждая с собственным оптимизированным графом функции (*function graph*), `FuncGraph`, и собственным определением функции (*function definition*), `FunctionDef`. Определение функции указывает на части графа, которые соответствуют входам и выходам функции. В каждом графе функции (`FuncGraph`) узлы (ovals) представляют операции (например, возведение в степень, константы или заполнители для аргументов вроде `x`), тогда как ребра (сплошные линии со стрелками между операциями) представляют тензоры, которые будут протекать через граф. Конкретная функция слева специализирована для  $x = 2$ , так что TensorFlow удалось упростить ее до выдачи 8 все время ( обратите внимание, что определение функции даже не имеет входа).

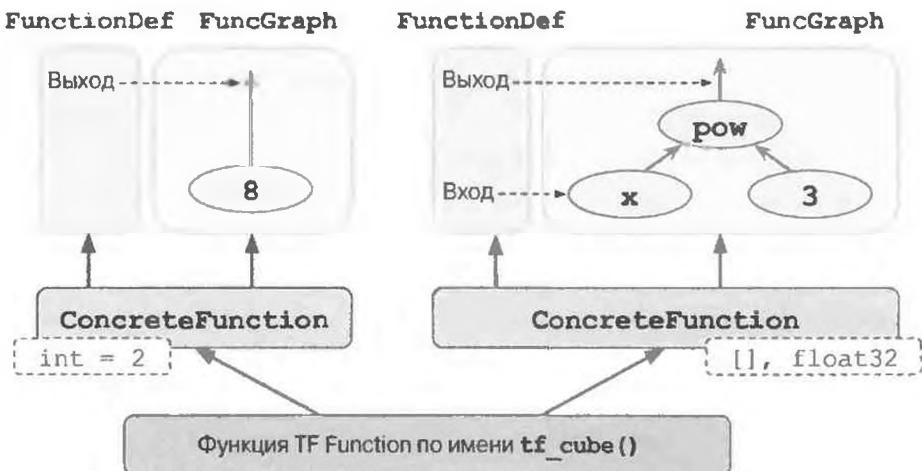


Рис. Ж.1. Функция TF Function по имени `tf_cube()` с ее конкретными функциями и графиками

Конкретная функция справа специализирована для скалярных тензоров float32 и не может быть упрощена. Если мы вызываем `tf_cube(tf.constant(5.0))`, то будет вызвана вторая конкретная функция, операция-заполнитель для `x` выдаст 5.0, затем операция возведения в степень вычислит  $5.0^{**} 3$ , так что выводом окажется 125.0.

Тензоры в этих графах являются *символьными тензорами*, т.е. они не имеют фактического значения, а лишь тип данных, форму и имя. Они представляют будущие тензоры, которые будут протекать через граф после того, как заполнителю `x` передано фактическое значение и граф выполнился. Символьные тензоры делают возможным заблаговременное определение способа соединения операций и также позволяют библиотеке TensorFlow рекурсивно выводить типы и формы данных всех тензоров, имея типы и формы данных их входов.

Теперь давайте продолжим заглядывать “за кулисы” и посмотрим, как получить доступ к определениям и графикам функций, а также исследовать операции и тензоры графа.

## Исследование определений и графов функций

Получить доступ к вычислительному графу конкретной функции можно с применением атрибута `graph`, а извлечь список его операций — с использованием метода `get_operations()` графа:

```
>>> concrete_function.graph
<tensorflow.python.framework.func_graph.FuncGraph at 0x14db5ef98>
>>> ops = concrete_function.graph.get_operations()
>>> ops
[<tf.Operation 'x' type=Placeholder>,
 <tf.Operation 'pow/y' type=Const>,
 <tf.Operation 'pow' type=Pow>,
 <tf.Operation 'Identity' type=Identity>]
```

В этом примере первая операция представляет входной аргумент `x` (он называется *заполнителем (placeholder)*), вторая “операция” — константу 3, третья — операцию возведения в степень (`**`) и последняя — выход функции (это операция эквивалентности, которая ничего не будет делать помимо копирования выхода дополненной операции<sup>1</sup>). Каждая операция имеет список входных и выходных тензоров, к которым легко получить доступ с

<sup>1</sup> Вы можете благополучно проигнорировать ее — она здесь присутствует только по техническим причинам для гарантии того, что функции TF Function не допускают утечки внутренних структур.

применением атрибутов `inputs` и `outputs` операции. Скажем, давайте получим список входов и выходов операции возвведения в степень:

```
>>> pow_op = ops[2]
>>> list(pow_op.inputs)
[<tf.Tensor 'x:0' shape=() dtype=float32>,
 <tf.Tensor 'pow/y:0' shape=() dtype=float32>]
>>> pow_op.outputs
[<tf.Tensor 'pow:0' shape=() dtype=float32>]
```

Итоговый вычислительный граф приведен на рис. Ж.2.

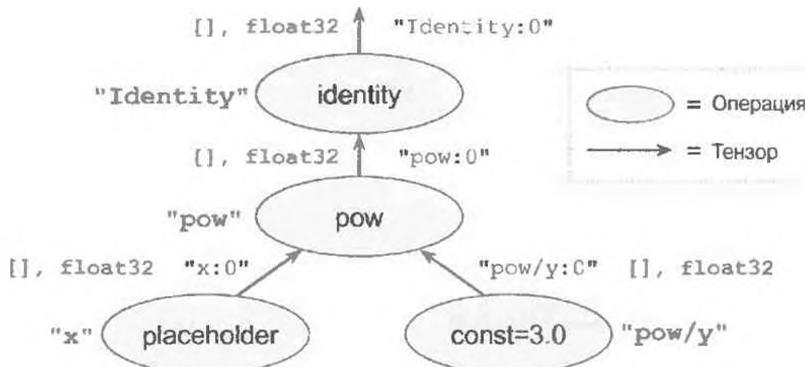


Рис. Ж.2. Пример вычислительного графа

Обратите внимание, что каждая операция имеет имя. По умолчанию им будет название операции (например, "pow"), но вы можете определить его вручную при вызове операции (скажем, `tf.pow(x, 3, name="other_name")`). Если имя уже существует, тогда TensorFlow автоматически добавит уникальный индекс (например, "pow\_1", "pow\_2" и т.д.). Каждый тензор тоже имеет уникальное имя: оно всегда совпадает с названием операции, которая выдает этот тензор, плюс :0, если он является первым выходом операции, :1, если вторым, и т.д. Извлечь операцию или тензор по имени можно с использованием методов `get_operation_by_name()` или `get_tensor_by_name()` графа:

```
>>> concrete_function.graph.get_operation_by_name('x')
<tf.Operation 'x' type=Placeholder>
>>> concrete_function.graph.get_tensor_by_name('Identity:0')
<tf.Tensor 'Identity:0' shape=() dtype=float32>
```

Конкретная функция содержит также определение функции (представленное в виде протокольного буфера<sup>2</sup>), которое включает сигнатуру функции.

<sup>2</sup> Популярный двоичный формат, обсуждавшийся в главе 13.

Эта сигнатура позволяет конкретной функции знать, в какие заполнители помещать входные значения и какие тензоры возвращать:

```
>>> concrete_function.function_def.signature
name: "__inference_cube_19068241"
input_arg {
    name: "x"
    type: DT_FLOAT
}
output_arg {
    name: "identity"
    type: DT_FLOAT
}
```

А теперь давайте пристальнее взглянем на трассировку.

## Более пристальный взгляд на трассировку

Скорректируем функцию `tf_cube()` для отображения ее входа:

```
@tf.function
def tf_cube(x):
    print("x =", x)
    return x ** 3
```

Вызовем ее:

```
>>> result = tf_cube(tf.constant(2.0))
x = Tensor("x:0", shape=(), dtype=float32)
>>> result
<tf.Tensor: id=19068290, shape=(), dtype=float32, numpy=8.0>
```

Выход `result` выглядит хорошо, но посмотрите внимательно на вход: `x` — символьный тензор! Он имеет форму и тип данных, но не значение. Вдобавок он имеет имя ("`x:0`"). Причина в том, что функция `print()` не является операцией TensorFlow, поэтому она будет выполняться только в случае трассировки функции Python, которая происходит в режиме графа, а аргументы заменяются символьными тензорами (того же типа и формы, но без значения). Поскольку функция `print()` не была захвачена в граф, когда мы в следующий раз вызываем `tf_cube()` со скалярными тензорами `float32`, ничего не отображается:

```
>>> result = tf_cube(tf.constant(3.0))
>>> result = tf_cube(tf.constant(4.0))
```

Но если мы вызовем `tf_cube()` с тензором другого типа или формы либо с новым значением Python, то функция снова будет трассироваться и потому `print()` вызывается:

```
>>> result = tf_cube(2)      # новое значение Python: трассируется!
x = 2
>>> result = tf_cube(3)      # новое значение Python: трассируется!
x = 3
>>> result = tf_cube(tf.constant([[1., 2.]]))    # новая форма:
                                                # трассируется!
x = Tensor("x:0", shape=(1, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[3., 4.], [5., 6.]])) # новая форма:
                                                # трассируется!
x = Tensor("x:0", shape=(None, 2), dtype=float32)
>>> result = tf_cube(tf.constant([[7., 8.], [9., 10.]])) # та же форма:
                                                # не трассируется
```



Если ваша функция имеет побочные эффекты Python (скажем, сохраняет журнальные файлы на диске), тогда позаботьтесь о том, чтобы такой код выполнялся только при трассировке функции (т.е. каждый раз, когда функция TF Function вызывается с новой сигнатурой входов). Лучше всего предположить, что функция может трассироваться (или нет) в любое время, когда вызывается функция TF Function.

В ряде случаев может возникнуть желание ограничить функцию TF Function специфической сигнатурой входов. Например, пусть известно, что функция TF Function будет вызываться только с пакетами изображений  $28 \times 28$  пикселей, но пакеты будут иметь очень разные размеры. Вы не хотите, чтобы библиотека TensorFlow генерировала свою конкретную функцию для каждого размера пакета, или рассчитываете на то, что самостоятельно решите, когда применять `None`. В таком случае вы можете указать сигнатуру входов следующего вида:

```
@tf.function(input_signature=[tf.TensorSpec([None, 28, 28], tf.float32)])
def shrink(images):
    return images[:, ::2, ::2] # отбросить половину строк и столбцов
```

Эта функция TF Function будет принимать любой тензор `float32` с формой `[*, 28, 28]` и каждый раз повторно использовать ту же самую конкретную функцию:

```
img_batch_1 = tf.random.uniform(shape=[100, 28, 28])
img_batch_2 = tf.random.uniform(shape=[50, 28, 28])
```

```
preprocessed_images = shrink(img_batch_1)      # Работает нормально.  
                                                # Трассирует функцию.  
preprocessed_images = shrink(img_batch_2)      # Работает нормально.  
                                                # Та же самая конкретная функция.
```

Однако если вы попытаетесь вызвать такую функцию TF Function со значением Python либо тензором непредвиденного типа данных или формы, то получите исключение:

```
img_batch_3 = tf.random.uniform(shape=[2, 2, 2])  
preprocessed_images = shrink(img_batch_3)      # Ошибка значения!  
                                                # Непредвиденная сигнатура.
```

## Использование AutoGraph для захвата потока управления

Если ваша функция содержит простой цикл `for`, тогда что произойдет, как вы думаете? Скажем, давайте напишем функцию, которая будет добавлять 10 к своему входу, просто добавляя единицу 10 раз:

```
@tf.function  
def add_10(x):  
    for i in range(10):  
        x += 1  
    return x
```

Функция работает хорошо, но когда вы просмотрите ее график, то обнаружите, что он не содержит цикла: он лишь включает 10 операций сложения!

```
>>> add_10(tf.constant(0))  
<tf.Tensor: id=19280066, shape=(), dtype=int32, numpy=10>  
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
 <tf.Operation 'add' type=Add>, [...],  
 <tf.Operation 'add_1' type=Add>, [...],  
 <tf.Operation 'add_2' type=Add>, [...],  
 [...]  
 <tf.Operation 'add_9' type=Add>, [...],  
 <tf.Operation 'Identity' type=Identity>]
```

На самом деле все имеет смысл: в результате трассировки функции цикл выполнился 10 раз, поэтому операция `x += 1` запускалась 10 раз, а из-за режима графа данная операция 10 раз была записана в график. Вы можете думать о таком цикле `for` как о “статическом” цикле, который раскручивается во время создания графа.

Если желательно, чтобы взамен граф содержал “динамический” цикл (т.е. такой, который выполняется при запуске графа), тогда можете создать его вручную с применением операции `tf.while_loop()`, но она не особенно понятна (пример ищите в разделе “Using AutoGraph to Capture Control Flow” (Использование AutoGraph для захвата потока управления) тетради Jupiter для главы 12). Вместо нее намного проще использовать средство `AutoGraph` библиотеки TensorFlow, обсуждавшееся в главе 12. В действительности средство `AutoGraph` по умолчанию включено (чтобы отключить его, понадобится передать `autograph=False` в `tf.function()`). Итак, если средство `AutoGraph` включено, то почему оно не захватило цикл `for` в функции `add_10()`? Дело в том, что средство `AutoGraph` захватывает только циклы `for`, которые проходят по диапазону `tf.range()`, а не `range()`. Это дает вам выбор.

- Если вы применяете `range()`, то цикл `for` окажется статическим, т.е. будет выполняться только при трассировке функции. Цикл “развернется” в набор операций для каждой итерации, как уже было показано.
- Если вы используете `tf.range()`, то цикл окажется динамическим, т.е. будет помещен в сам график (но не будет выполняться во время трассировки).

Давайте взглянем на график, который генерируется после замены вызова `range()` вызовом `tf.range()` в функции `add_10()`:

```
>>> add_10.get_concrete_function(tf.constant(0)).graph.get_operations()  
[<tf.Operation 'x' type=Placeholder>, [...],  
<tf.Operation 'range' type=Range>, [...],  
<tf.Operation 'while' type=While>, [...],  
<tf.Operation 'Identity' type=Identity>]
```

Легко заметить, что теперь график содержит операцию цикла `While`, как если бы мы вызывали функцию `tf.while_loop()`.

## Обработка переменных и других ресурсов в функциях TF Function

В библиотеке TensorFlow переменные и другие объекты с запоминанием состояния, такие как очереди или наборы данных, называются *ресурсами*. Функции `TF Function` трактуют их особым образом: любая операция, которая читает или обновляет ресурс, считается запоминающей состояния, и

функции TF Function гарантируют, что операции с запоминанием состояния выполняются в порядке их появления (в противоположность операциям без запоминания состояния, которые могут запускаться параллельно, поэтому порядок их выполнения не гарантируется). Кроме того, когда вы передаете ресурс в качестве аргумента функции TF Function, он передается по ссылке, так что функция может его модифицировать. Например:

```
counter = tf.Variable(0)

@tf.function
def increment(counter, c=1):
    return counter.assign_add(c)

increment(counter)    # значение counter теперь равно 1
increment(counter)    # значение counter теперь равно 2
```

Заглянув в определение функции, вы увидите, что первый аргумент помещен как ресурс:

```
>>> function_def = increment.get_concrete_function(counter).function_def
>>> function_def.signature.input_arg[0]
name: "counter"
type: DT_RESOURCE
```

Внутри функции можно использовать переменную `tf.Variable`, определенную вне функции, не передавая ее явно в качестве аргумента:

```
counter = tf.Variable(0)

@tf.function
def increment(c=1):
    return counter.assign_add(c)
```

Функция TF Function будет трактовать такую переменную как неявный первый аргумент, а потому фактически мы получаем ту же самую сигнатуру (за исключением имени аргумента). Тем не менее, применение глобальных переменных может быстро привести к путанице, поэтому обычно вы должны помещать переменные (и другие ресурсы) внутрь классов. Хорошая новость в том, что `@tf.function` нормально работает и с методами:

```
class Counter:
    def __init__(self):
        self.counter = tf.Variable(0)

@tf.function
def increment(self, c=1):
    return self.counter.assign_add(c)
```



Не используйте `=`, `+=`, `-=` или любую другую операцию присваивания Python с переменными TF. Взамен вы должны применять методы `assign()`, `assign_add()` или `assign_sub()`. Если вы попытаетесь воспользоваться операцией присваивания Python, то при вызове метода получите исключение.

Хорошим примером такого объектно-ориентированного подхода является, конечно же, `tf.keras`. Давайте посмотрим, как применять функции TF `Function` с `tf.keras`.

## Использование функций TF `Function` с `tf.keras`

По умолчанию любая специальная функция, слой или модель, которую вы применяете с `tf.keras`, будет автоматически преобразовываться в функцию TF `Function`; вам вообще не придется что-то делать! Однако в ряде случаев у вас может возникнуть желание отключить такое автоматическое преобразование — скажем, если ваш специальный код нельзя превратить в функцию TF `Function` или вы хотите отладить свой код, что легче всего делать в энергичном режиме. Для этого вы можете просто передать `dynamic=True` при создании модели или любого из ее слоев:

```
model = MyModel(dynamic=True)
```

Если ваша специальная модель или слой будет всегда динамическим, тогда взамен можете вызывать конструктор базового класса с `dynamic=True`:

```
class MyLayer(keras.layers.Layer):
    def __init__(self, units, **kwargs):
        super().__init__(dynamic=True, **kwargs)
        [...]
```

В качестве альтернативы можете передавать `run_eagerly=True` при вызове метода `compile()`:

```
model.compile(loss=my_mse, optimizer="nadam", metrics=[my_mae],
               run_eagerly=True)
```

Теперь вы знаете, каким образом функции TF `Function` поддерживают полиморфизм (с множеством конкретных функций), как графы автоматически генерируются с помощью средства AutoGraph и трассировки, на что похожи сами графы, каким образом исследовать их символьные операции и тензоры, как обрабатывать переменные и ресурсы и каким образом использовать функции TF `Function` с библиотекой `tf.keras`.

# Предметный указатель

## A

Accuracy, 37; 144  
Action, 373; 786; 799; 845  
Activation function, 377; 383  
Active learning, 339  
Actor-Critic, 805  
AdaGrad, 465  
AdaMax, 469  
Adaptive Instance Normalization (AdaIN), 780  
Adaptive learning rate, 466  
Adaptive moment estimation, 467  
Advantage Actor-Critic (A2C), 853  
Adversarial learning, 641  
Affinity, 318  
Agent, 786  
Agglomerative clustering, 343  
Akaike information criterion (AIC), 354  
AlexNet, 602  
Alignment model, 712  
Alpha dropout, 482  
Analyzer, 573  
Anchor box, 634  
Anomaly detection, 48  
Apache Beam, 573  
API-интерфейс  
    Data, 537; 538  
    Functional, 405; 419  
    Keras, 498  
    REST, 133; 859  
    Sequential, 404  
    Subclassing, 411; 419  
Approximate Q-Learning, 815  
Arcade Learning Environment (ALE), 829  
Artificial neural network (ANN), 369  
Artificial neuron, 374  
Association rule learning, 46; 49  
Associative memory network, 992  
Asynchronous Advantage Actor-Critic (A3C), 852  
Autoencoder, 45; 733  
AutoGraph, 531; 1017; 1018  
Automatic differentiation (autodiff), 382  
Autoregressive integrated moving average (ARIMA), 656  
Average Precision (AP), 635

## B

Backpropagation through time (BPTT), 651  
Bagging, 261; 265  
Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH), 344  
Batch GD (BGD), 172  
Batch Gradient Descent, 183  
Batch Normalization (BN), 445  
Bayesian Deep Learning, 362  
Bayesian information criterion (BIC), 354  
Bellman Optimality Equation, 807  
Bias, 173; 376; 604  
Biological neural networks (BNN), 373  
Black box stochastic variational inference (BBSVI), 361  
Blender, 284  
Boltzmann machine, 994  
Boosting, 261  
Bootstrap aggregating, 265  
Bootstrapping, 265  
Bucket, 873

## C

Callback, 413  
Capsule network, 641  
Char-RNN, 681  
Classification And Regression Tree (CART), 250  
Clustering, 317  
Coding, 733  
Comma-separated value (CSV), 90  
Completely Automated Public Turing Test to Tell Computers and Humans Apart (CAPTCHA), 135  
Computational complexity, 178  
Compute Unified Device Architecture (CUDA), 887  
Confusion matrix, 145  
Constrained optimization, 234  
Contrastive Divergence, 997  
Convolutional neural network (CNN), 579  
Core instance, 340  
Cross entropy, 214  
Cross-entropy loss, 388  
Cross-validation, 72  
CUDA Deep Neural Network (CuDNN), 888

## D

Dataset, 538  
Data snooping bias, 96  
Data visualization (DataViz), 290  
Decision boundary, 212  
Decision function, 149  
Decision Stump, 278  
Decision threshold, 149  
Decision Tree, 45; 245  
Decoder, 651  
Deconvolution layer, 638  
Deep autoencoder, 739  
Deep belief net (DBN), 992; 998  
Deep belief network (DBN), 45; 50  
Deep convolutional GAN (DCGAN), 772  
Deep Learning VM Images, 891  
Deep neural network (DNN), 380  
Deep Neuroevolution, 424  
Deep Q-Learning, 816  
Deep Q-network (DQN), 786  
Deep Reinforcement Learning, 782  
Density-based Spatial Clustering of Applications with Noise (DBSCAN), 316; 340  
Depth concatenation layer, 605  
Depth radius, 604  
Deque, 817  
Development set, 72  
Dilation rate, 639  
Discriminator, 734  
Distributed (Deep) Machine Learning Community (DMLC), 283  
Distribution Strategies, 858; 911  
Docker, 863  
Double Dueling DQN, 827  
DQN, 816  
Dropout, 479  
Dropout rate, 480  
Duality, 981  
Dual problem, 236  
Dueling DQN (DDQN), 826  
Dying ReLU, 440

## E

Elastic Net, 198  
Embedded Reber grammar, 731  
Embedding, 116; 538  
Embedding matrix, 566  
Embeddings from Language Models (ELMo), 727  
Encoder, 651

Ensemble, 125; 130; 261  
Entailment, 728  
Environment, 786  
Environment wrapper, 831  
Epoch, 187; 382  
Equality constraint, 981  
Equivariance, 595  
Estimator, 111  
Euclidian norm, 84  
Evidence lower bound (ELBO), 361  
Exclusive OR (XOR), 379  
Expectation-Maximization (EM) algorithm, 348  
Experience replay, 770  
Explainability, 715  
Explained variance ratio, 300  
Exploding gradients, 435

## F

False negative (FN), 146  
False positive (FP), 146  
False positive rate (FPR), 153  
Feedforward neural network (FNN), 380  
FIFO (First In, First Out), 502  
Filter, 585  
Fixup, 453  
Forecasting, 652  
Forget gate, 669  
Frame, 646  
Fully convolutional network (FCN), 630  
Functional, 419

## G

GAN, 776; 966  
Gate controller, 670  
Gated Recurrent Unit (GRU), 672  
Gaussian mixture model (GMM), 316; 346  
Gaussian RBF, 227; 322  
Generative adversarial network (GAN), 459; 730; 733  
Generator, 734  
GitHub, 640  
Global average pooling layer, 597  
Google Cloud AI Platform, 133  
Google Cloud Engine (GCE), 879  
Google Cloud Platform (GCP), 871  
Google Cloud Storage (GCS), 133; 873  
GoogLeNet, 605  
Gradient Boosted Regression Tree (GBRT), 278  
Gradient Boosting, 274; 278  
Gradient Descent (GD), 171

Graphviz, 246  
Grid Search, 126

## H

Hebb's rule, 377  
Hessian, 470  
Hierarchical Cluster Analysis (HCA), 46  
Hierarchical DBSCAN (HDBSCAN), 343  
Hinge loss, 222; 241  
Huber loss, 386  
Hypothesis, 83

## I

ImageNet, 447  
ImageNet Large Scale Visual Recognition Challenge (ILSVRC), 600  
Importance sampling (IS), 825  
Impurity, 247  
Imputation, 652  
Incremental PCA (IPCA), 303  
Independent and identically distributed (IID), 188  
Inference, 61  
Inter-op thread pool, 899  
Intersection over Union (IoU), 627  
Invariance, 594  
Isomap, 311

## J

Jacobian, 470  
JIT (just-in-time), 492

## K

Karush-Kuhn-Tucker (KKT), 982  
Keras, 388; 449; 498; 569; 940; 948  
Kernel, 396; 494  
Kernelized, 231; 364  
Kernel PCA (kPCA), 46; 305  
Kernel trick, 225  
K-Means, 46; 316; 319; 322  
K-Means++, 325  
K-Nearest Neighbors, 45  
Kullback-Leibler (KL) divergence, 215; 360

## L

Label, 43; 338  
Lagrangian, 981  
Landmark, 226  
Lasso Regression, 198  
LeNet-5, 600; 601  
Levenshtein distance, 228

Linear Discriminant Analysis (LDA), 312  
Linear threshold unit (LTU), 375  
Lipschitz continuous, 181  
Locally-Linear Embedding (LLE), 46; 308  
Local Outlier Factor (LOF), 364  
Local response normalization (LRN), 604  
Logistic, 207  
Logistic Regression, 45; 172  
Logit, 208  
Logit Regression, 207  
Log loss, 209  
Log-odds, 208  
Long Short-Term Memory (LSTM), 668

## M

Majority-vote prediction, 260  
Manhattan norm, 84  
Manifold Learning, 290  
Markov chain, 805  
Markov decision process (MDP), 786  
Masked language model (MLM), 729  
Masked Multi-Head Attention, 719  
Matplotlib, 151  
Maximum a-posteriori (MAP), 356  
Maximum likelihood estimate (MLE), 356  
Mean Average Precision (mAP), 635  
Mean Squared Error (MSE), 174  
Mercer's theorem, 239  
Mesa OpenGL Utility (GLU), 791  
Meta learner, 284  
Mini-batch, 189  
Mini-batch discrimination, 771  
Mini-batch Gradient Descent (GD), 172; 189  
Mirrored strategy, 905  
Mixed National Institute of Standards and Technology (MNIST), 139  
Mixing regularization, 781  
Mode collapse, 770  
Model parallelism, 902  
Model warmup, 868  
Momentum, 462  
Momentum optimization, 462  
Monte Carlo (MC) Dropout, 483  
Multiclass classifier, 157  
Multidimensional Scaling (MDS), 311  
Multi-Head Attention, 719  
Multilabel classification, 165  
Multi-Layer Perceptron (MLP), 370; 380  
Multinomial classifier, 157  
Multinomial Logistic Regression, 213

## N

- Nadam, 470
- Naive Bayes classifier, 157
- Nash equilibrium, 769
- Natural language processing (NLP), 461
- Nest, 831
- Nesterov Accelerated Gradient (NAG), 462; 464
- Neural machine translation (NMT), 680; 702
- Neural network, 45
- Neuro-Evolution of Augmenting Topologies (NEAT), 790
- Neurotransmitter, 373
- Newton's difference quotient, 985
- Next sentence prediction (NSP), 729
- No Free Lunch (NFL), 74
- Nonlinear dimensionality reduction (NLDR), 308
- Normal Equation, 174
- Null hypothesis, 254
- NumPy, 498
- NVIDIA Collective Communications Library (NCCL), 912

## O

- Object detection, 628
- Objectness, 629
- Observation, 786
- Offline learning, 51
- One-versus-all (OvA), 157
- One-versus-one (OvO), 157
- One-versus-the-rest (OvR), 157
- Online model, 823
- OpenAI Gym, 790; 791
- Out-of-core learning, 53
- Out-of-sample error, 71
- Output gate, 670
- Output layer, 380
- Overcomplete autoencoder, 750

## P

- Parametric leaky ReLU (PReLU), 441
- Pasting, 266
- Peephole connection, 671
- Percentile, 93
- Perceptron, 375
- Perceptron convergence theorem, 378
- Performance scheduling, 473
- Piecewise constant scheduling, 473
- Placeholder, 1013
- Plate, 347

## Platform-as-a-Service (PaaS), 870

- Policy, 788
- Policy gradient, 786
- Policy gradient (PG), 790
- Policy parameter, 789
- Policy search, 789
- Policy space, 789
- Polynomial Regression, 172; 191
- Polynomial Regression model, 61
- Pooling kernel, 593
- Positional embedding, 719
- Positive class, 146
- Post-training quantization, 882
- Power scheduling, 473
- Precision, 146
- Precision-recall (PR) curve, 153
- Predictor, 43; 112
- Pre-image, 307
- Primal problem, 236
- Principal Component Analysis (PCA), 46; 257; 296
- Prioritized Experience Replay (PER), 825
- Probability Density Function (PDF), 316; 350
- Projection, 290
- Propositional logic, 370
- Protobuf, 553
- Protocol Buffer, 537
- Proximal Policy Optimization (PPO), 853
- PyTorch, 390

## Q

- Q-Learning, 812
- Quadratic Programming (QP), 235
- Quality Value, 809
- Quantization-aware training, 884
- Quartile, 93
- Q-Value, 809

## R

- Radial Basis Function (RBF), 226
- Random Forest, 45; 261; 270
- Randomized leaky ReLU (RReLU), 441
- Randomized PCA, 303
- Randomized Search, 129
- Random Patches method, 270
- Random Projections, 311
- Random Subspaces method, 270
- Recall, 146
- Receiver operating characteristic (ROC), 153
- Reconstruction, 737

Rectified linear unit (ReLU), 384  
Recurrent neural network (RNN), 645  
Recurrent neuron, 646  
Region Proposal Network (RPN), 636  
Regularization, 68  
Regularization term, 198  
Reinforcement Learning (RL), 43; 785  
ReLU, 440; 664

с утечкой, 440  
параметрический, 441  
рандомизированный, 441  
угасающий, 440

Residual Network (ResNet), 609  
Residual unit (RU), 610  
ResNet, 609; 611; 956  
Responsibility, 349  
Restricted Boltzmann Machine (RBM), 45; 50; 996  
Reward, 786  
RMSProp, 467  
Role, 913  
Root Mean Squared Error (RMSE), 82  
RReLU, 441

## S

SAMME (Stagewise Additive Modeling using a Multiclass Exponential loss function), 277  
Scaled Dot-Product Attention, 722  
Scikit-Learn, 300; 191  
средство перекрестной проверки Scikit-Learn, 118; 124; 134  
SE block, 616  
Self-organizing map (SOM), 1000  
Self-supervised learning, 461  
SENet, 616  
Sensitivity, 147; 153  
Sequence, 645  
Shrinkage, 280  
Singular Value Decomposition (SVD), 177; 298  
Softmax, 213; 387; 722  
Soft voting, 265  
Sparsity, 753  
Specificity, 153  
Squeeze-and-Excitation Network, 616  
Stacked autoencoder, 739  
Stacked denoising autoencoder, 751  
Stacking, 261; 283  
Start-of-sequence (SOS), 693; 702  
Stemming, 169  
Stick-Breaking Process (SBP), 358  
Stochastic Gradient Boosting, 283

Stochastic Gradient Descent (SGD), 142; 172; 186  
Strata, 99  
Stride, 583  
StyleGAN, 779  
Subclassing, 411; 419  
Support Vector Clustering, 222  
Support Vector Machine (SVM), 45; 125; 219

## T

Tag, 862  
Task, 913  
t-distributed Stochastic Neighbor Embedding (t-SNE), 46; 311  
Temperature, 688  
Temporal Difference Learning (TD), 811  
Tensor, 496  
TensorBoard, 416  
TensorFlow (TF), 471; 492; 537; 555; 588; 639; 858; 884; 947; 1011; 1018  
TensorFlow Datasets (TFDS), 574  
TensorFlow Extended, 495  
TensorFlow Hub, 495  
TensorFlow Lite, 495  
TensorFlow Model Optimization Toolkit (TF-MOT), 471  
TensorFlow Playground, 388  
TensorFlow Serving, 859  
масштабирование, 871  
установка, 863

Tensor processing unit (TPU), 494  
TF Datasets (TFDS), 538  
TF Function, 529; 532; 573; 1011; 1018  
TF Hub, 700  
TF-IDF, 571  
TF Serving, 970  
TF Transform, 538; 572  
Threshold logic unit (TLU), 375  
Tie, 745  
Tolerance, 185  
Transformer, 112; 680; 702; 716  
True negative rate (TNR), 153  
True negative (TN), 146  
True positive rate (TPR), 147  
True positive (TP), 146  
Trust Region Policy Optimization (TRPO), 853

## V

Variance, 93  
VGGNet, 609  
Visual Geometry Group (VGG), 609

# W

WaveNet, 676  
Weak learner, 263  
Wide & Deep, 406

# Y

YOLO (You Only Look Once), 632

# Z

Zero-shot learning (ZSL), 728

# A

Автокодировщик, 45; 733  
вариационный, 757  
вероятностный, 757  
глубокий, 739  
многослойный, 739; 740  
обучение, 746  
повышающий, 750  
понижающий, 737; 965  
порождающий, 757  
разреженный, 753  
рекуррентный, 749  
сверточный, 748  
шумоподавляющий, 750  
многослойный, 751  
с гауссовым шумом, 751  
с отключением, 751

Автокорреляция, 683

Агент, 50; 786

DQN, 840

Аккаунт

сервисный, 877

Аксон, 372

Алгоритм

AdaGrad, 465  
Adam, 468  
BIRCH, 344; 939  
CART, 250; 255  
DBSCAN, 316; 340  
Fast-MCD, 363  
Isolation Forest, 363  
K-Means, 316; 319; 322; 332; 939  
K-Means++, 325  
LLE, 310  
LOF, 364  
Mean-Shift, 344  
One-class SVM, 364  
PCA, 297; 302; 363

REINFORCE, 799

RL, 791

RMSProp, 467

“актер-критик”, 805; 852; 853

асинхронный (A3C), 852

мягкий (SAC), 853

расширенный (A2C), 853

генетический, 790

динамического обучения, 53

динамического размещения, 897

для линейной регрессии, 191

итерации по Q-ценностям, 809

кластеризации, 939

максимизации ожидания, 348

иерархический, 343

нейроэволюции нарастающих топологий (NEAT), 790

обучения

Q-, 812

без учителя, 46

методом временных разностей, 811

на основе моделей, 926

с обратным распространением, 381

с учителем, 45

оптимизации политик областей доверия

(TRPO), 853

пакетной нормализации, 446

поглощающий (жадный), 251

регрессии методом k ближайших соседей, 61

спектральной кластеризации, 345

ускоренного градиента Нестерова, 464

Альфа-канал, 334

Анализ

главных компонентов (PCA), 257; 296

данных

интеллектуальный (глубинный), 40

инкрементный, 303

линейный дискриминантный, 312

ошибок, 160

рандомизированный, 303

ядерный, 304

Анализатор, 573

Аномалия

обнаружение аномалий, 352; 939

Ансамбль (ensemble), 261; 2724 281

Архитектура

AlexNet, 602

GoogLeNet, 605; 607

LeNet-5, 600; 601

Mask R-CNN, 641  
ResNet, 611; 612  
TensorFlow, 494  
WaveNet, 675; 676  
Xception (Extreme Inception), 613  
YOLOv3, 633  
кодирующей сети, 839  
нейронных сетей  
    сверточных, 597  
персептрона, 377  
Атрибут, 44

**Б**

Бета-распределение, 358  
Библиотека  
    cuDNN, 698; 888  
    GLU, 791  
    Hyperas, 423  
    Hyperband, 423  
    Hyperopt, 423  
    Keras, 388; 510; 567  
    Keras Tuner, 423  
    kopt, 423  
    NumPy, 498  
    OpenAI Gym, 832  
    OpenCV, 163  
    Pillow, 163  
    PyTorch, 390  
    Scikit-Image, 163  
    Scikit-Learn, 110; 139; 147; 250; 267; 300; 343  
    Scikit-Optimize, 423  
    Sklearn-Deap, 423  
    Spearmint, 423  
    Talos, 423  
    TensorFlow, 492; 1018  
        архитектура, 494  
        операции свертки, 639  
    TF-Agents, 827; 831  
    tf.keras, 550  
    TF Transform, 573  
    XGBoost, 283  
    групповых коммуникаций NVIDIA, 912  
Блок (fold), 123  
SE, 616  
остаточный, 515  
управляемый рекуррентный (GRU), 672  
Бустинг, 261; 274  
    градиентный, 274; 278  
    на основе деревьев, 278  
    стохастический, 283

Бутстрэп-агрегирование, 265  
Бутстрэппинг, 265  
Буфер  
    воспроизведения, 817; 836  
    протокольный, 537  
Бэггинг, 261; 265; 267

**В**

Вектор  
    -градиент, 183  
    перекрестной энтропии, 216  
    функции издержек, 183  
момента, 462  
опорный, 220; 930  
параметров модели, 173  
признаков образца, 173  
скалярное произведение векторов, 173  
субградиент, 203  
транспонированный, 173  
целевых значений, 175

Вероятность  
    плотность вероятности, 350

Вес  
    соединение весов, 745

Визуализация  
    данных, 290  
    реконструкций, 741

Виртуальная машина  
    для глубокого обучения, 891

Вложение (embedding), 116; 538  
    многоголовое, 722  
    позиционное, 719; 720  
    слов, 565

Внимание  
    Бахданау, 712  
    зрительное, 714  
    конкатенативное, 712  
    Лyonга, 713  
    механизмы внимания, 714  
    многоголовое, 719  
        маскированное, 719  
    модель выравнивания, 712  
    мультиплекативное, 713  
    самовнимание, 719  
    слой внимания, 712

Временной ряд, 652  
    многомерный, 652  
    одномерный, 652  
Временной шаг  
    пакетный, 845

- Время  
фактическое, 448
- Вставка, 265; 266; 267
- Выборка  
по значимости, 825
- смещение выборки (*sampling bias*), 64
- стратифицированная, 99
- шум выборки (*sampling noise*), 64
- Выведение (*inference*), 61
- Вывод  
байесовский, 362
- стохастический вариационный вывод по принципу “черного ящика”, 361
- Выражение  
регулярное, 695
- Вычисления  
с помощью нейронов, 374
- Вычислительная сложность, 178; 251
- Г**
- Гауссова функция RBF, 226
- Гауссово распределение, 346
- Гауссово ядро RBF, 227
- Генератор, 734; 764
- GAN, 776
- Генерирование изображений Fashion MNIST, 762
- Гессиан, 470
- Гиперпараметр, 69
- настройка, 71
- регуляризации, 253
- Гиперплоскость, 232
- Гипотеза, 83
- нулевая, 254
- Гнездо, 831
- Голосование
- жесткое, 262
- мягкое, 265
- Градиент  
взрывной рост градиентов, 435
- вычисление градиентов с использованием автоматического дифференцирования, 520
- исчезновение градиентов, 435
- неустойчивый, 645
- Нестерова
- ускоренный, 462; 464
- отсечение градиентов, 453
- политики, 786; 799
- устаревший, 908
- Градиентный бустинг, 283
- Градиентный спуск, 178
- мини-пакетный, 189
- пакетный, 182; 183
- полный, 183
- просчеты градиентного спуска, 181
- с масштабированием признаков, 182
- с различными скоростями обучения, 185
- стохастический (SGD), 186
- Граница решений, 211; 212
- линейная, 212
- Граф
- TensorFlow, 1011
- вычислительный, 492
- функции, 1012
- График
- кусочно-линейный, 473
- мощности, 473
- обучения, 187; 472
- одного цикла, 473
- экспоненциальный, 473
- эффективности, 473
- Д**
- Данные
- анализ данных
- интеллектуальный (глубинный), 40
- важность данных, 63
- визуализация данных, 290
- дополнение данных, 168; 602
- загрузка данных, 90
- зашумление, 57
- использование Keras для загрузки набора данных, 392
- набор данных Swiss roll, 293
- набор данных с вложениями, 685
- необоснованная эффективность данных, 62
- нерепрезентативные, 64
- несоответствие данных, 73
- обучающие, 36
- недообучение обучающими данными, 69
- переобучение обучающими данными, 67
- очистка данных, 110
- параллелизм данных, 902; 905
- плоский набор данных, 685
- плохого качества, 66
- специальные структуры данных, 1003
- тасование данных, 541
- хранилище данных, 77
- эффективное представление данных, 735

Двойственность (duality), 981  
Действие, 786  
отдача (return) действия, 798  
пакетный шаг действия, 846  
преимущество действия, 799  
шаг действия, 845

Декодировщик, 651; 736

Дендрит, 372

Дерево

- двоичное, 248
- классификации и регрессии (CART), 250
- принятия решений, 245; 248; 932
  - для регрессии, 255
- регрессии с градиентным бустингом, 278

Диаграмма

- силузтов, 330

Дискриминатор, 734; 765

Дисперсия (variance), 93; 197  
объясненная, 301

Дифференцирование, 656  
автоматическое, 382; 986  
в обратном режиме, 989  
в прямом режиме, 989

ручное, 984

символьное, 988

Допуск (tolerance), 185

### 3

Забывание

- катастрофическое, 821
- Загрязненность (impurity), 247
- Джини, 248; 252; 932
- Задание, 913
- Задача, 913
  - NP-полная, 251
  - двойственная, 236
  - квадратичного программирования, 235
  - многомерной регрессии, 81
  - обнаружение аномалий, 48
  - обнаружение новизны, 48
  - обучение ассоциативным правилам, 49
  - одномерной регрессии, 81
  - понижение размерности, 48
  - прямая, 236
  - регрессии, 44
  - составной регрессии, 81

Закон

больших чисел, 263

Заполнитель, 1013

Запрос, 722

JSON, 133

Зеркальное отображение, 905

### И

Измерение

зубчатое, 1004

Изображение

репрезентативное, 337

Изометрическое отображение (Isomap), 311

Имитация отжига (simulated annealing), 187

Инвариантность, 594

масштабная, 594

ротационная, 594

трансляционная, 594

Инерция модели, 325

Инициализатор, 506

Глоро, 506

Инициализация

Глоро, 437; 438

Лекуна, 438

случайная, 179

Хе, 439

использование вместе с ELU, 445

Инструмент

Apache Beam, 573

TensorBoard, 416

Интервал

доверительный, 131

Интерполяция

семантическая, 763

Интерфейс API

REST, 859

Исключающее "ИЛИ", 379

Испытательный набор, 71; 926

создание, 96

Итерации по ценностям, 808

### К

Капча (CAPTCHA), 135

Карта

признаков, 307; 586

самоорганизующаяся, 1000

Квантование

после обучения, 882

Класс

LinearSVC, 229

SGDClassifier, 229

SVC, 229

вероятность класса, 249

- отрицательный, 146; 207  
положительный, 146; 207  
спрогнозированный, 146  
фактический, 146
- Классификатор**  
AdaBoost  
обучение, 275  
SVM, 219; 224; 228; 232; 931  
безупречный, 146  
двоичный, 142  
многоклассовый, 157  
наивный байесовский, 157  
обучение классификаторов, 262  
полиномиальный, 157  
с голосованием, 262  
жестким, 262
- Классификация**, 43; 139  
SVM, 229  
нелинейная, 223  
истинно отрицательная (TN), 146; 153  
истинно положительная (TP), 146; 147  
ложноотрицательная (FN), 146  
ложноположительная (FP), 146; 153  
многовходовая, 166  
многозадачная, 409  
многозначчная, 164  
многоклассовая, 157  
с жестким зазором, 220  
с мягким зазором, 220  
с широким зазором, 219; 220
- Кластер**, 316  
TensorFlow, 913; 914  
нахождение оптимального количества кластеров, 328
- Кластеризация**, 47; 316  
DBSCAN, 341  
агломеративная, 343  
жесткая, 322  
использование кластеризации  
для предварительной обработки, 334  
для частичного обучения, 336  
методом опорных векторов, 222  
мягкая, 322  
спектральная, 345
- Кодирование**  
категориальных признаков с использованием вложений, 564  
с одним активным состоянием, 115  
суммированием слов в мешок, 538  
унитарное, 115
- Кодировка, 733  
парами байтов, 694
- Кодировщик**, 651; 736  
предложений, 701
- Компилятор**  
JIT, 492
- Конвейеры**, 80  
трансформации, 119
- Коннекционизм**, 371
- Консоль**  
GCP, 871
- Конструирование признаков (feature engineering)**, 66
- Контейнер Docker**, 863
- Контроллер**  
шлюзов, 670
- Коэффициент**  
дисконтирования, 798  
доверия, 797  
корреляции  
Пирсона ( $r$ ), 104  
стандартный, 104  
объясненной дисперсии, 300  
разветвления по входу (fan-in), 438  
разветвления по выходу (fan-out), 438  
расширения, 639  
силуэта, 329
- Кривая**  
ROC, 153  
обучения, 172; 193; 194  
для линейной модели, 195  
для полиномиальной модели десятой степени, 197  
точности-полноты, 153
- Л**
- Лагранжиан, 981  
обобщенный, 982
- Лассо-регрессия**, 201; 929
- Латентное представление**, 733
- Левенштейн (Levenshtein)**, 228
- Лес**  
случайный, 261; 270
- Логарифмическая потеря (log loss)**, 209
- Логарифм коэффициента перевеса**, 208
- Логика высказываний**, 370
- Логистика**, 207
- Логистическая функция активации**, 941
- Логит (logit)**, 208

**Логический элемент**

пороговый (TLU), 375

**Логическое следование (entailment)**, 728

**Луч**, 708

## M

**Маркер**

конца последовательности (EOS), 702  
начала последовательности (SOS), 693; 702

**Марковский процесс**

принятия решений, 786; 805

**Маска**

тензор маски, 698

**Маскирование**, 697

**Массив**

тензорный, 1006

тензоров, 501

**Масштабирование**

по минимаксу, 118

признаков, 117

**Матрица**

вложений, 566

Мура-Пенроуза, 177

обратная, 177

неточностей, 145

параметров, 213

псевдообратная, 177

разреженная, 115

**Машина**

Больцмана, 45; 50; 994

ограниченная, 460; 996

**Машинное обучение (МО)**, См. Обучение, 35; 36; 924

автономное, 51

без учителя, 45

непомеченный обучающий набор, 45

внешнее, 53

динамическое, 52; 54

на основе моделей, 55; 56

на основе образцов, 55

обучающий набор, 44

основные проблемы машинного обучения, 62

пакетное, 51

постепенное, 53

скорость обучения, 54

с подкреплением, 43; 50

с учителем, 43; 45

типы систем машинного обучения, 42

частичное, 43; 49

**Мера F1**, 148

**Мерсер (Mercer)**, 239

теорема Мерсера, 239

условия Мерсера, 239

**Метаграф**, 862; 970

**Метка**, 43; 862

распространение меток, 338

**Метод**

AdaBoost, 274

SVM, 220; 231; 931

динамический, 240

ансамблевый, 130; 264

опорных векторов (SVM), 125; 219; 930

кластеризация, 222

случайных подпространств, 270

случайных участков, 270

сопоставительного отклонения, 997

**Метрика**

обучения, 844

потоковая, 509

с запоминанием состояния, 509

специальная, 507

средней точности (AP), 635

**Механизм**

Docker, 863

внимания, 680; 710; 714; 962

**Минимум**

глобальный, 180

локальный, 180

**Мини-пакет**, 52; 189

**Многозадачная классификация**, 409

**Множества**, 502; 1007

**Множитель**

Каруша-Куна-Таккера (KKT), 982

Лагранжа, 981

**Мода**

коллапс мод, 770

статистическая, 266

**Модель**

Char-RNN, 687

SVC, 222

SVM

параметрически редуцированная, 231

TensorFlow, 858

алгоритмы обучения на основе моделей, 926

архитектура модели, 59

белого ящика, 249

вектор параметров модели, 173

выравнивания, 712

динамическая, 823  
инерция модели, 325  
компиляция модели, 397  
линейная, 57; 199; 201  
  кривые обучения для линейной модели, 195  
  ретрессионная, 171; 173  
    функция издержек MSE, 174  
маскированная языковая, 729  
непараметрическая, 253  
обучение модели, 58; 927  
  на кластере TensorFlow, 913  
параллелизм модели, 902  
параметрическая, 253  
параметры модели, 58  
перекрестная проверка, 143  
полиномиальная, 61; 197; 199; 201  
  ретрессионная, 61  
порождающая, 350; 733; 965; 996  
разогрев модели, 868  
разреженная, 202  
резервное копирование, 136  
система слежения, 135  
со смесью гауссовых распределений, 346; 940  
  байесовская, 357  
специальная, 515  
с регуляризацией, 929  
тип модели, 59  
целевая, 823  
черного ящика, 249  
эффективность модели  
  падение эффективности, 135  
  реальная, 134

Модуль, 700  
  SE-Inception, 616  
  начала, 606; 956

Момент, 462

## Н

Наблюдения, 786  
Набор данных, 538  
  Fashion MNIST  
    визуализация, 742  
  MNIST, 139  
использование Keras для загрузки набора  
  данных, 392  
испытательный, 926  
обучающе-развивающий, 73; 926  
обучающий, 36  
  помеченный, 924  
расширение обучающего набора, 168

плоский, 685  
проводочный, 72; 926  
развивающий, 72  
с вложениями, 685  
Награды (reward), 786  
Недообучение (underfitting), 69  
Нежесткость  
  дополняющая, 982  
Нейрон  
  биологический, 372  
  входной, 376  
  искусственный, 374  
  логические вычисления с помощью  
    нейронов, 374  
  рекуррентный, 646  
  смещения, 376  
Нейронная сеть  
  Wide & Deep, 406  
  биологическая (BNN)  
    архитектура, 373  
  глубокая (DNN), 380  
    обучение, 435  
  двунаправленная рекуррентная, 707  
  искусственная (ANN), 369  
  прямого распространения (FNN), 380  
  рекуррентная (RNN), 645  
  сверточная (CNN), 579  
    архитектура, 597

Нейронный машинный перевод, 680; 702

Нейротрансмиттер, 373

Нейроэволюция

  глубокая, 424

Неустойчивость, 257

Норма, 84

  евклидова, 84

  Манхэттена, 84

Нормализация

  пакетная, 445; 664

  с помощью Keras, 449

по слову, 665

Нормальное уравнение, 174

## О

Облачное хранилище Google (GCS), 873  
Обнаружение аномалий, 939  
Обнаружение новизны, 354; 939  
Обновление  
  асинхронное, 908  
  фиксированное, 453  
Оболочка среды, 831

- Образ  
ложный, 994
- Образец  
центральный, 340
- Обратное распространение, 942  
во времени (BPTT), 651
- Обратный вызов, 413
- Обучающие данные (training data), 36
- Обучающий набор (training set), 36; 71  
для классификации спама, 44  
непомеченный, 45  
помеченный, 924  
расширение, 168
- Обучающий образец (training instance), 36
- Обучение  
 $Q$ -, 812  
глубокое, 816; 817  
приближенное, 815  
с применением функции исследования, 815
- автокодировщик, 746
- активное, 339  
выборка по наименьшей уверенности, 339
- ансамблевое, 125; 261; 934  
с бэггингом, 934
- асимметрия между обучением и обслуживанием, 572
- без подготовки, 728
- без учителя, 315; 938  
предварительное, 744  
с использованием автокодировщиков, 745
- глубоких нейронных сетей, 944
- глубокое  
байесовское, 362  
с подкреплением, 782
- график обучения, 472  
разреженных, 471
- дерева принятия решений, 245
- классификатора, 262  
AdaBoost, 275
- методом временных разностей, 811
- метрики обучения, 844
- моделей, 58; 927  
глубоких, 435  
на кластере TensorFlow, 913  
на множестве устройств, 902
- на основе данных, 36
- на основе многообразий, 290; 294
- однократное, 641
- остаточное, 610
- первого уровня, 285
- передачей знаний, 425; 454  
с помощью Keras, 456
- предварительное  
без учителя, 459  
жадное послойное, 460  
на вспомогательной задаче, 460
- представлению (representation learning), 564
- прекращение обучения  
раннее, 205
- скорость обучения, 472
- смесителя, 285
- создание цикла обучения, 850
- состязательное (adversarial), 641; 735
- с подкреплением, 785; 966
- с помощью TensorFlow, 946
- с учетом квантования, 884
- с учителем, 461
- Объект, 628
- Объектность, 629
- Объяснимость (explainability), 715
- Ограничения, 506
- Операция  
argmax, 214  
свертки TensorFlow, 639
- Оптимизатор, 428; 461  
сравнение оптимизаторов, 471
- Оптимизация  
Adam, 467  
Nadam, 470  
моментная, 462  
Нестерова, 464  
условная, 234
- Оптическое распознавание знаков (OCR), 35
- Ориентир (landmark), 226
- Ответственность (responsibility), 349
- Отклонение  
сопоставительное, 997  
среднее абсолютное, 84  
стандартное, 93
- Отключение (dropout), 479  
альфа-, 482  
доля отключения, 480
- Монте-Карло (MC), 483
- Отношение Ньютона  
разностное, 985
- Отображение  
изометрическое, 311
- Оценка  
апостериорного максимума (MAP), 356  
максимального правдоподобия (MLE), 356

**Оценщики (estimator),** 111  
**Очередь,** 502; 1009  
  простая, 502  
  с двусторонним доступом, 817  
**Ошибка**  
  анализ ошибок, 160  
  восстановления, 302  
  выхода за пределы выборки, 71  
  обобщения (generalization error), 71  
  обучения методом временных разностей, 811  
  остаточная, 278  
  среднеквадратическая (MSE), 174  
  средняя абсолютная (MAE), 84

## П

**Пакет**  
  Graphviz, 246  
  размер пакета, 428  
**Пакетная нормализация (BN),** 445; 664  
**Память**  
  краткосрочная, 667  
  пропускная способность памяти, 549  
  ячейка памяти, 649  
**Параллелизм**  
  данных, 902; 905  
  модели, 902  
**Переменная,** 500  
  латентная, 348  
  наблюдаемая, 348  
  фиктивная, 234  
**Переобучение (overfitting),** 67  
**Пересечение по объединению (IoU),** 627  
**Персептрон,** 375  
  архитектура, 377  
  классический, 941  
  многослойный, 386  
  многослойный (MLP), 370; 380  
    реализация многослойных персепtronов с помощью Keras, 388  
  правило обучения персептрана (обновление весов), 378  
  теорема о сходимости персептрана, 378  
**Платформа**  
  Google Cloud AI Platform, 133  
**Плашка,** 347  
**Плотность вероятности,** 350  
**Погрешность**  
  неустранимая, 197  
**Поддержка,** 166

**Поиск**  
  лучевой, 708; 962  
  без запоминания состояния, 680; 689  
  с запоминанием состояния, 680; 689  
  морфологический, 169  
  рандомизированный, 129  
  решетчатый, 126  
**Показатели эффективности,** 143  
**Полиномиальное отображение второй степени,** 237  
**Политика,** 50; 788  
   $\epsilon$ -жадная, 814  
  градиент политики, 790; 799  
  нейросетевая, 795; 796  
  параметры политики, 789  
  поиск политики, 789  
  проксимальная, 853  
  пространство политик, 789  
  стохастическая, 789  
**Полнота (recall),** 146  
  кривая точности-полноты, 153  
  соотношение точность/полнота, 149  
**Понижение размерности,** 48; 289; 311; 935  
  нелинейное, 308  
**Порог принятия решения,** 149  
**Последовательность,** 645  
**Потенциал**  
  биоэлектрический, 373  
**Потеря**  
  из-за разреженности, 754  
  из-за реконструкции, 518; 737  
  латентная, 759  
  Хьюбера, 386; 502  
**Поток**  
  пул потоков  
    внутриоперационный, 899  
    межоперационный, 899  
**Похожесть (affinity),** 318  
**Правило**  
  обучения персептрана, 378  
  Хебба, 377  
  цепное, 382; 990  
**Правильность (accuracy),** 144  
**Представление,** 564  
  латентное, 733  
**Преобразование**  
  аффинное, 779  
**Преобразователь (Transformer),** 680  
**Приближение**  
  конечно-разностное, 985

- P**
- Признак (feature), 44
    - выбор признаков, 66
    - выделение признаков, 48; 66
    - карта признаков, 307
    - конструирование признаков, 66
    - несущественный, 66
    - пространство признаков, 304
    - разреженный, 229
    - создание новых признаков, 66
  - Проверка
    - перекрестная (cross-validation), 72; 143
    - с удерживанием (holdout validation), 72
  - Проверочный набор, 72; 926
  - Прогноз, 56; 247; 286
    - агрегирование прогнозов, 284
    - классификатора
      - линейного SVM, 232
      - с жестким голосованием, 262
    - линейной регрессионной модели, 173
    - мажоритарный, 260
  - Прогнозатор (predictor), 43; 112
    - вес прогнозатора, 276
  - Прогнозирование, 652
    - наивное, 654
  - Программа AlphaGo, 51
  - Программирование
    - квадратичное, 235
  - Проекция, 290
  - Проектирование
    - случайное, 311
  - Производная
    - липшиц-непрерывная, 181
    - частная, 182
  - Прообраз, 307
  - Пропускная способность памяти, 549
  - Пространство
    - латентное, 758
    - политик, 789
  - Протобуфер, 553
    - Example, 556
    - SequenceExample, 558
    - TensorFlow, 555
  - Процентиль, 93
  - Процессор
    - тензорный, 494
  - Пул потоков
    - внутриоперационный, 899
    - межоперационный, 899
- Равновесие Нэша, 769
  - Радиус глубины, 604
  - Развертывание, 134
  - Различение
    - мини-пакетное, 771
  - Размерность
    - понижение размерности, 289; 292; 935
    - “проклятие размерности”, 289; 290
  - Разреженность (sparsity), 753
  - Разрешение
    - супер-, 640
  - Разум
    - коллективный, 261
  - Рамка
    - ограничивающая, 634
    - опорная, 634
  - Распределение
    - апостериорное, 360
    - бета, 358
    - категориальное, 347
    - нормальное (гауссово), 93
  - Распространение
    - обратное, 942
  - Расстояние (расхождение)
    - Кульбака–Лейблера, 215; 360; 361; 754; 755
    - Левенштейна, 228
  - Регрессия, 44; 254
    - SVM, 230; 231
    - гребневая, 198; 199
    - дерево принятия решений, 255
    - задача регрессии, 44
    - лассо, 198; 201; 929
      - вектор-субградиент лассо-регрессии, 204
      - функция издержек для лассо-регрессии, 201
    - линейная, 172
    - логистическая, 45; 172; 207
      - многопараметрическая, 172; 213; 216
      - полиномиальная, 213
    - методом k ближайших соседей, 61
    - методом наименьшего абсолютного сокращения и выбора, 201
    - полиномиальная, 191; 194
  - Регуляризация, 68
    - $\ell_1$ , 506
    - $\ell_2$ , 478
    - на основе так-нормы, 486
    - смешиванием, 781
    - с ранним прекращением, 205
  - Тихонова, 198

**Режим**  
энергичный, 531  
**Реконструкция**, 737  
визуализация реконструкций, 741  
**Ресурс**, 1018  
**Ретрансляция**, 942  
**Рецепторное поле**, 580  
**Роль**, 913

## C

**Самовнимание**, 719  
**Свертка**, 582  
**Свободный член (intercept term)**, 173  
**Сводка (summary)**, 416  
**Связь**  
обходящая, 610  
смотровая, 671  
сокращенная, 610  
**Сглаживающий член (smoothing term)**, 446  
**Сегмент**, 873  
**Сегментация**  
семантическая, 580; 637  
**Сегментирование**  
изображений, 332  
семантическое, 332  
образцов, 332  
цветов, 333  
**Сервер**  
TF Serving, 859; 970  
оценки, 914  
параметров, 914  
рабочий, 913  
старший, 913  
**Сеть**  
ANN, 369  
DCGAN, 772  
DQN, 786; 816  
двойная, 824  
соревнующаяся, 826; 827  
CGAN, 775  
GAN, 966  
RNN, 646; 650; 961  
символьная, 680; 681  
StyleGAN, 779  
глубокая  
доверия, 992; 998  
капсульная, 641  
кодирующая, 839  
**нейронная**, 888  
CUDA, 888  
Wide & Deep, 406  
биологическая (BNN), 373  
глубокая (DNN), 380; 435  
рекуррентная, 657  
искусственная (ANN), 369  
прямого распространения (FNN), 380  
рекуррентная, 645  
сверточная (CNN), 579; 583  
архитектура, 597  
остаточная, 609  
порождающая, 736  
состязательная, 459; 730; 733; 764; 966  
“последовательность в вектор”, 650  
развертыванием сети во времени, 646  
распознающая, 736  
самонормализованная, 443  
с ассоциативной памятью, 992  
сверточная, 630  
полностью (FCN), 630  
сжатия и возбуждения, 616  
с обходящими связями, 443  
Хопфилда, 992  
эластичная, 198; 204; 930  
**Сжатие**, 280  
**Сигнал**, 79  
**Сигнатура входов**, 1011  
**Синапс**, 372  
**Сингулярное разложение (SVD)**, 177; 298  
**Система**  
слежения, 135  
машинного обучения, 42  
**Скалярное произведение**, 713  
векторов, 173  
**Скорость**  
обучения, 54; 179; , 427; 472  
адаптивная, 466  
сходимости, 185  
**Слово**  
мешок слов, 571  
**Слой**  
адаптивной нормализации образцов  
(AdaIN), 780  
внимания, 712  
масштабированного скалярного  
произведения, 722  
входной, 376; 380  
высший, 380  
выходной, 380

дву направл ен ный рекуррентный, 707  
конкатенации в глубину, 605  
многоголового внимания, 963  
архитектура, 725  
низший, 380  
обращения свертки, 638  
объединения  
по глобальному среднему, 597  
по максимуму, 593  
по среднему, 595  
объединяющий, 582; 592  
повышения дискретизации, 638  
полносвязный (или плотный), 376  
свертки по глубине, 639  
сепарабельный, 613  
сверточный, 582  
сепарабельный, 614  
с множеством карт признаков, 587  
“с отверстиями”, 639  
транспонированный, 638  
скрытый, 380  
специальный, 511  
суживающийся, 606  
Смеситель (blender), 284  
Смешивание, 284  
Смещение, 197; 197  
выборки (sampling bias), 64  
Смысловой анализ, 692  
Соединение весов, 745  
Спам  
фильтр спама, 35; 43  
Специфичность (specificity), 153  
Спуск  
градиентный  
мини-пакетный (Mini-batch GD), 172  
пакетный (BGD), 172  
стохастический (SGD), 142; 172  
Среда (environment), 786  
Colab, 893  
Colaboratory, 891  
оболочка среды, 831  
обучения  
игровая, 829  
создание изолированной среды, 87  
спецификации среды, 830  
Среднее гармоническое, 148  
Средняя абсолютная ошибка (MAE), 84  
Стандартизация, 118  
Стандартное отклонение, 93  
Стекинг, 261; 283; 284

Стиль  
передача стиля, 779  
смешивание стилей, 781  
Страйд, 583  
Стратегия  
“один против всех” (OvA), 157  
“один против одного” (OvO), 157  
“один против остальных” (OvR), 157  
Страты (strata), 99  
Строки, 1003  
Структуры данных  
специальные, 1003  
Субдифференциал, 241  
Супер-разрешение, 640

**Т**

Телодендрон, 372  
Температура, 688  
Тензор, 496; 498; 947  
зубчатый, 501; 1004  
маски, 698  
массив тензоров, 501  
разреженный, 501; 1006  
символьный, 531  
строковый, 501

Теорема  
Байеса, 360  
Мерсера, 239  
об отсутствии бесплатных завтраков, 74  
о сходимости персептрона, 378  
Теория информации Шеннона, 252  
Точность, 37; 146  
в сравнении с полнотой, 152  
кривая точности-полноты, 153  
соотношение точность/полнота, 149  
усредненная средняя (MAP), 635

Траектория, 836  
Трансформаторы (transformer), 112  
специальные, 116  
Трассировка, 531  
Трюк  
ядерный (kernel trick), 225  
для вычисления члена смещения, 240  
для полиномиального отображения второй  
степени, 238

**У**

Узел  
корневой, 247  
листовой, 247

**Уравнение**  
квадратное, 191  
нормальное, 174  
оптимальности Беллмана, 807  
**Установка**  
TensorFlow Serving, 863  
**Устройство ГП**  
виртуальное, 895  
**Ученик**  
мета-, 284  
сильный, 263  
слабый, 263

## Ф

**Файл**  
событий, 416  
**Факториал числа**, 193  
**Фильтр**, 585  
расширенный, 639  
спама, 35; 43  
**Формат**  
SavedModel, 860  
экспортирование в формат SavedModel, 860  
TFRecord, 552  
**Фрейм**, 646  
**Функция**, 375  
confusion\_matrix(), 145  
cross\_val\_predict(), 145; 150  
cross\_val\_score(), 143  
reshape(), 211  
roc\_curve(), 153  
TensorFlow, 529  
TF Function, 573; 1011; 1012; 1019  
активации, 377; 383; 385; 429; 439; 506; 647  
ELU, 442  
ReLU, 945  
    с утечкой, 441  
SELU, 443; 944  
softplus, 385  
логистическая, 941  
ненасыщаемая, 440; 664  
близости (*similarity function*), 226  
выпрямленного линейного элемента  
    (ReLU), 384; 941  
выпуклая, 181  
гауссова радиальная базисная, 322  
гиперболического тангенса (*tanh*), 384; 941  
гипотезы, 173  
знака, 375

издержек, 58; 83; 208  
CART, 250  
MSE  
    для линейной регрессионной модели, 174  
вектор-градиент функции издержек, 183  
в форме перекрестной энтропии, 215  
для гребневой регрессии, 199  
для лассо-регрессии, 201  
линейного классификатора SVM, 241  
частная производная, 183  
исследования, 815  
Лагранжа, 981  
логистическая, 207; 208  
    многопараметрическая, 213  
логит, 208  
многопараметрическая (*softmax*), 387; 722  
непрерывная, 181  
нормализованная экспоненциальная, 213  
определение функции, 1012  
отображающая (*mapping function*), 237  
петлевая (*hinge loss*), 222; 241  
плотности вероятности (PDF), 316; 350  
полезности (*utility function*), 58  
потери, 502  
правдоподобия, 355  
приспособленности (*fitness function*), 58  
прогнозирования системы, 83  
радиальная базисная (RBF), 226  
    Гауссова, 226  
решения (*decision function*), 149; 232  
сигмоидальная, 207  
стоимости (*cost function*), 58  
Хевисайда  
    ступенчатая, 375  
энергетическая, 993

## Х

**Характеристика приемника**  
рабочая (ROC), 153  
**Хранилище**  
GitHub, 640  
TF Hub, 700  
**данных**  
    открытое, 77  
облачное, 873  
    Google (GCS), 133

# Ц

Цветовой канал, 586  
Цель TD, 811  
Центроид, 319; 324  
Цепи Маркова, 805

# Ч

Чатбот, 679  
Число  
  дуальное, 988  
Член  
  регуляризации, 198  
  свободный (intercept term), 173  
  сглаживающий (smoothing term), 446  
  смещения (bias term), 173  
Чувствительность (sensitivity), 147; 153

# Ш

Ширина луча (beam width), 708; 962  
Шкалирование  
  многомерное, 311  
Шлюз  
  входной, 669; 670  
  выходной, 670  
  забывания, 669; 670  
  контроллер шлюзов, 670  
Штрафы, 50  
Шум выборки (sampling noise), 64

# Э

Эквивариантность, 595  
Экземпляр  
  сегментация экземпляров, 640  
Экспортирование в формат SavedModel, 860

# Элемент

ReLU  
  с утечкой, 440  
  утасающий, 440  
Scaled ELU, 443  
видимый, 995  
линейный пороговый (LTU), 375  
остаточный, 610  
скрытый, 995

# Энтропия, 252

  перекрестная, 214; 215; 388  
  вектор-градиент, 216

# Эпоха (epoch), 187; 382

Эффективность  
  показатели эффективности, 143

# Я

Ядерный трюк, 225  
для вычисления члена смещения, 240  
для полиномиального отображения второй  
степени, 238  
Ядро, 238; 396; 494  
RBF  
  Гауссово, 227  
на основе расстояния Левенштейна, 228  
объединяющее, 593  
полиномиальное, 224; 238  
сверточное, 585  
строковое, 228  
Якобиан, 470  
Ячейка  
  памяти, 649  
    долгой краткосрочной (LSTM), 668; 671  
  управляемого рекуррентного блока  
    (GRU), 672

# ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

Андреас Мюллер  
Сара Гвидо



[www.williamspublishing.com](http://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

O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

2 0 0 1 4

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

2 0 0 1 4

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

2 0 0 1 4

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 203334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>



O'REILLY

# Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow

Благодаря серии выдающихся 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 данных. Новое издание книга-бестселлера, опирающееся на конкретные примеры, минимум теории и готовые фреймворки Python производственного уровня, поможет вам получить интуитивное представление о концепциях и инструментах, предназначенных для построения интеллектуальных систем.

Вы освоите широкий спектр методик, которые можно быстро задействовать на практике. Учитывая наличие в каждой главе упражнений, призванных закрепить то, чему вы научились, для начала работы нужен лишь опыт программирования. Весь код доступен на GitHub. Он был обновлен с учетом TensorFlow 2 и последней версии Scikit-Learn.

- Изучите основы машинного обучения на сквозном проекте с применением Scikit-Learn и pandas
- Постройте и обучите нейронные сети с многочисленными архитектурами для классификации и регрессии, используя TensorFlow 2
- Ознакомьтесь с выявлением объектов, семантической сегментацией, механизмами внимания, языковыми моделями, порождающими состязательными сетями и многим другим
- Исследуйте Keras API — официальный высокоДировневый API-интерфейс для TensorFlow 2
- Запускайте в производство модели TensorFlow с применением Data API из TensorFlow, стратегий распределения, TF Transform и TF Serving
- Развортывайте модели на платформе AI Platform инфраструктуры Google Cloud или на мобильных устройствах
- Используйте методики обучения без учителя, такие как понижение размерности, кластеризация и обнаружение аномалий
- Создавайте автономные обучающиеся агенты с помощью обучения с подкреплением, в том числе с применением библиотеки TF-Agents

Все иллюстрации к книге в цветном варианте доступны по адресу <http://go.dialektika.com/mlearning>

Категория: Данные / Наука о данных / Анализ данных / Машинное обучение / Глубокое обучение / Машинное обучение с помощью Python

Уровень: для пользователей средней и высокой квалификации

“Выдающийся ресурс для изучения машинного обучения. Вы найдете здесь ясные и интуитивно понятные объяснения, а также обилие практических советов.”

Франсуа Шолле,  
автор библиотеки Keras, автор книги *Deep Learning with Python*

“Эта книга — замечательное введение в теорию и практику решения задач с помощью нейронных сетей; я рекомендую ее всем, кто заинтересован в освоении практического машинного обучения.”

Пит Уорден,  
руководитель команды мобильной разработки TensorFlow

Орельен Жерон — консультант и инструктор по машинному обучению. Бывший работник компаний 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).



ISBN 978-5-907203-33-4

20014

9 785907 20334

DIALEKTIKA

<http://www.williamspublishing.com>

<http://oreilly.com>

